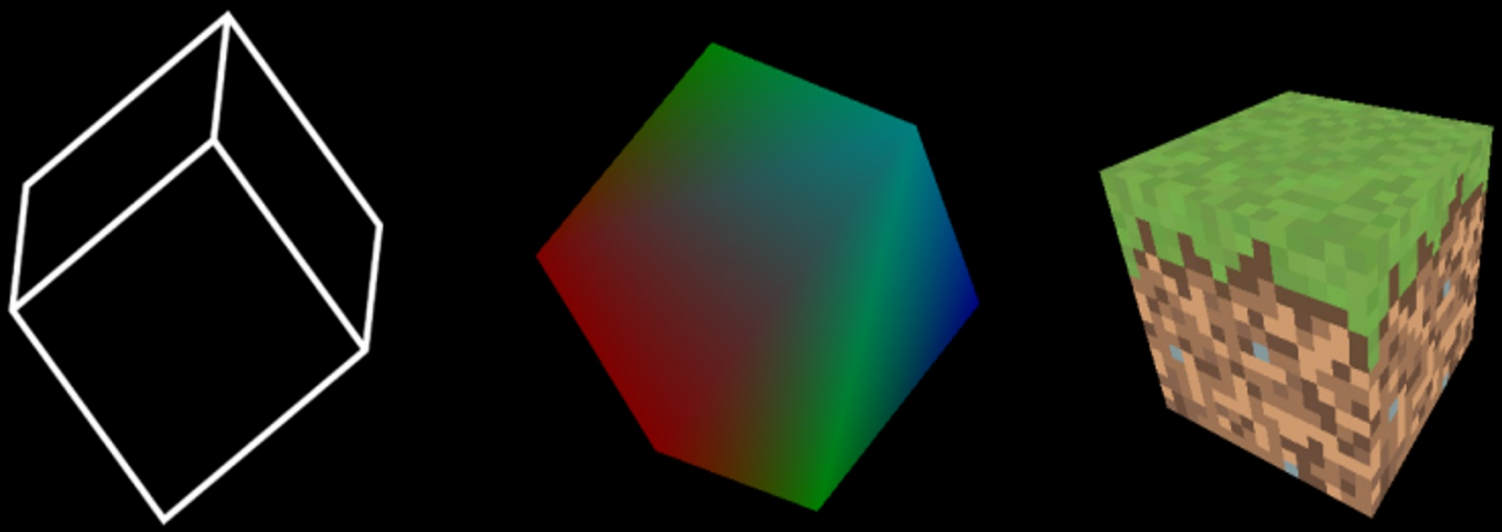
3D Game Development with LWJGL 3

Learn the main concepts involved in writing 3D games using the Lightweight Java Gaming Library



www.dbooks.org

Table of Contents

Introduction	1.1
First steps	1.2
The Game Loop	1.3
A brief about coordinates	1.4
Rendering	1.5
More on Rendering	1.6
Transformations	1.7
Textures	1.8
Camera	1.9
Loading more complex models	1.10
Let there be light	1.11
Let there be even more light	1.12
HUD	1.13
Sky Box and some optimizations	1.14
Height Maps	1.15
Terrain Collisions	1.16
Fog	1.17
Normal Mapping	1.18
Shadows	1.19
Animations	1.20
Particles	1.21
Instanced Rendering	1.22
Audio	1.23
3D Object picking	1.24
Hud revisited - NanoVG	1.25
Optimizations	1.26
Cascaded Shadow Maps	1.27
Assimp	1.28
Deferred Shading	1.29
Appendix A - OpenGL Debugging	1.30

3D Game Development with LWJGL 3

* **IMPORTANT NOTICE** *: The online book is being migrated to new gitbook space: https://ahbejarano.gitbook.io/lwjglgamedev/

This online book will introduce the main concepts required to write a 3D game using the LWJGL 3 library.

LWJGL is a Java library that provides access to native APIs used in the development of graphics (OpenGL), audio (OpenAL) and parallel computing (OpenCL) applications.This library leverages the high performance of native OpenGL applications while using the Java language.

My initial goal was to learn the techniques involved in writing a 3D game using OpenGL. All the information required was there in the internet but it was not organized and sometimes it was very hard to find and even incomplete or misleading.

I started to collect some materials, develop some examples and decided to organize that information in the form of a book.

Source Code

The source code of the samples of this book are in GitHub.

The source code for the book itself is also published in GitHub.

License

The book is licensed under Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

The source code for the book is licensed under Apache v2.0

Support

If you like the book please rate it with a start and share it. If you want to contribute with a donation you can do a donation:

Donate

Or if you prefer Bitcoin: 1Kwe78faWarzGTsWXtdGvjjbS9RmW1j3nb.

Comments are welcome

Suggestions and corrections are more than welcome (and if you do like it please rate it with a star). Please send them using the discussion forum and make the corrections you consider in order to improve the book.

Author

Antonio Hernández Bejarano

Special Thanks

To all the readers that have contributed with corrections, improvements and ideas.

www.dbooks.org

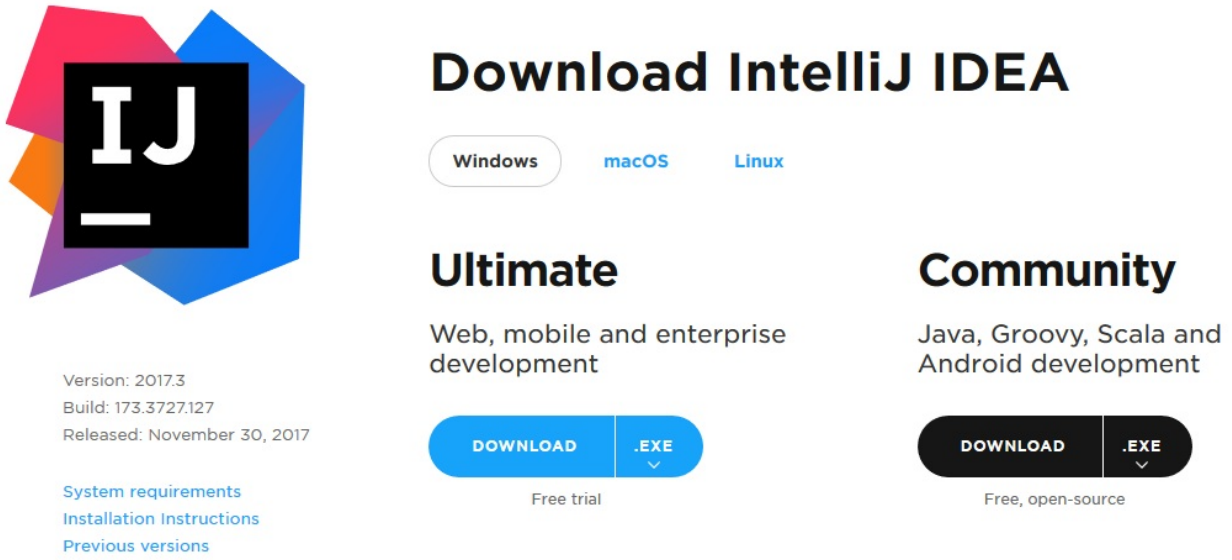
First steps

In this book we will learn the principal techniques involved in developing 3D games. We will develop our samples in Java and we will use the Lightweight Java Game Library (LWJGL). The LWJGL library enables the access to low-level APIs (Application Programming Interface) such as OpenGL.

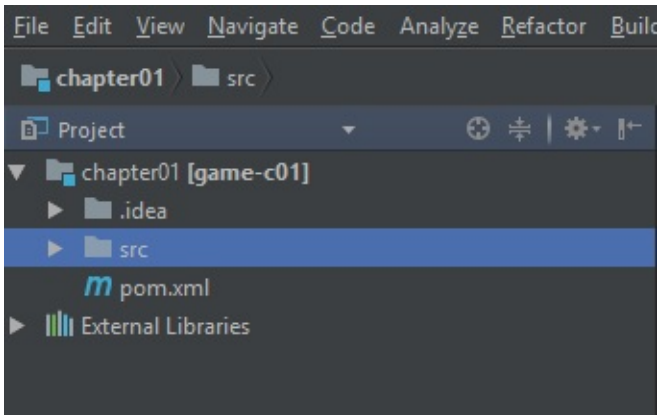
LWJGL is a low level API that acts like a wrapper around OpenGL. If your idea is to start creating 3D games in a short period of time maybe you should consider other alternatives like [JmonkeyEngine]. By using this low level API you will have to go through many concepts and write lots of lines of code before you see the results. The benefit of doing it this way is that you will get a much better understanding of 3D graphics and also you can get better control.

As said in the previous paragraphs we will be using Java for this book. We will be using Java 10, so you need to download the Java SDK from Oracle's pages. Just choose the installer that suits your Operating System and install it. This book assumes that you have a moderate understanding of the Java language.

You may use the Java IDE you want in order to run the samples. You can download IntelliJ IDEA which has good support for Java 10. Since Java 10 is only available, by now, for 64 bits platforms, remeber to download the 64 bits version of IntelliJ. IntelliJ provides a free open source version, the Community version, which you can download from here: https://www.jetbrains.com/idea/download/.



For building our samples we will be using Maven. Maven is already integrated in most IDEs and you can directly open the different samples inside them. Just open the folder that contains the chapter sample and IntelliJ will detect that it is a maven project.



Maven builds projects based on an XML file named `pom.xml` (Project Object Model) which manages project dependencies (the libraries you need to use) and the steps to be performed during the build process. Maven follows the principle of convention over configuration, that is, if you stick to the standard project structure and naming conventions the configuration file does not need to explicitly say where source files are or where compiled classes should be located.

This book does not intend to be a maven tutorial, so please find the information about it in the web in case you need it. The source code folder defines a parent project which defines the plugins to be used and collects the versions of the libraries employed.

LWJGL 3.1 introduced some changes in the way that the project is built. Now the base code is much more modular, and we can be more selective in the packages that we want to use instead of using a giant monolithic jar file. This comes at a cost: You now need to carefully specify the dependencies one by one. But the download page includes a fancy script that generates the pom file for you. In our case, we will just be using GLFW and OpenGL bindings. You can check what the pom file looks like in the source code.

The LWJGL platform dependency already takes care of unpacking native libraries for your platform, so there's no need to use other plugins (such as `mavennatives`). We just need to set up three profiles to set a property that will configure the LWJGL platform. The profiles will set up the correct values of that property for Windows, Linux and Mac OS families.

www.dbooks.org

```
    <profiles>
        <profile>
            <id>windows-profile</id>
            <activation>
                <os>
                    <family>Windows</family>
                </os>
            </activation>
            <properties>
                <native.target>natives-windows</native.target>
            </properties>
        </profile>
        <profile>
            <id>linux-profile</id>
            <activation>
                <os>
                    <family>Linux</family>
                </os>
            </activation>
            <properties>
                <native.target>natives-linux</native.target>
            </properties>
        </profile>
        <profile>
            <id>OSX-profile</id>
            <activation>
                <os>
                    <family>mac</family>
                </os>
            </activation>
            <properties>
                <native.target>natives-osx</native.target>
            </properties>
        </profile>
    </profiles>
```

Inside each project, the LWJGL platform dependency will use the correct property established in the profile for the current platform.

```
        <dependency>
            <groupId>org.lwjgl</groupId>
            <artifactId>lwjgl-platform</artifactId>
            <version>${lwjgl.version}</version>
            <classifier>${native.target}</classifier>
        </dependency>
```

Besides that, every project generates a runnable jar (one that can be executed by typing java -jar name_of_the_jar.jar). This is achieved by using the maven-jar-plugin which creates a jar with a `MANIFEST.MF` file with the correct values. The most important attribute for that file

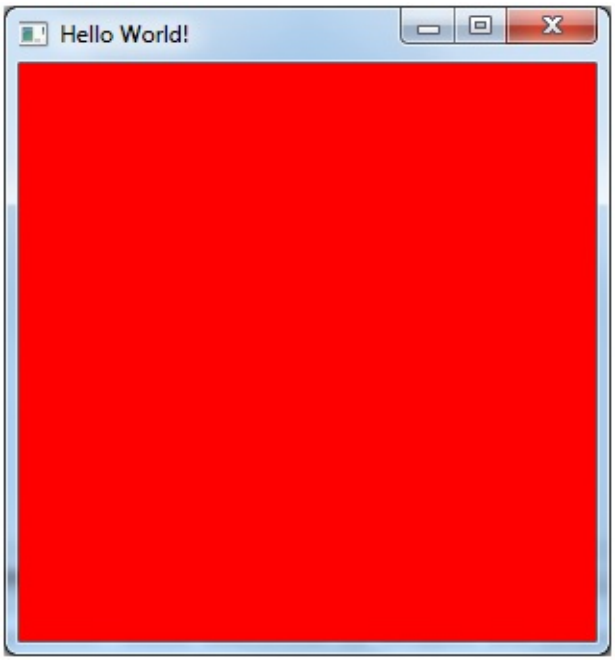
is `Main-Class`, which sets the entry point for the program. In addition, all the dependencies are set as entries in the `Class-Path` attribute for that file. In order to execute it on another computer, you just need to copy the main jar file and the lib directory (with all the jars included there) which are located under the target directory.

The jars that contain LWJGL classes, also contain the native libraries. LWJGL will also take care of extracting them and adding them to the path where the JVM will look for libraries.

Chapter 1 source code is taken directly from the getting started sample in the LWJGL site (http://www.lwjgl.org/guide). You will see that we are not using Swing or JavaFX as our GUI library. Instead of that we are using GLFW which is a library to handle GUI components (Windows, etc.) and events (key presses, mouse movements, etc.) with an OpenGL context attached in a straightforward way. Previous versions of LWJGL provided a custom GUI API but, for LWJGL 3, GLFW is the preferred windowing API.

The samples source code is very well documented and straightforward so we won't repeat the comments here.

If you have your environment correctly set up you should be able to execute it and see a window with a red background.



The source code of this book is published in GitHub.

www.dbooks.org

The Game Loop

In this chapter we will start developing our game engine by creating our game loop. The game loop is the core component of every game. It is basically an endless loop which is responsible for periodically handling user input, updating game state and rendering to the screen.

The following snippet shows the structure of a game loop:

```
while (keepOnRunning) {
    handleInput();
    updateGameState();
    render();
}
```

So, is that all? Are we finished with game loops? Well, not yet. The above snippet has many pitfalls. First of all the speed that the game loop runs at will be different depending on the machine it runs on. If the machine is fast enough the user will not even be able to see what is happening in the game. Moreover, that game loop will consume all the machine resources.

Thus, we need the game loop to try running at a constant rate independently of the machine it runs on. Let us suppose that we want our game to run at a constant rate of 50 Frames Per Second (FPS). Our game loop could be something like this:

```
double secsPerFrame = 1.0d / 50.0d;

while (keepOnRunning) {
    double now = getTime();
    handleInput();
    updateGameState();
    render();
    sleep(now + secsPerFrame – getTime());
}
```

This game loop is simple and could be used for some games but it also presents some problems. First of all, it assumes that our update and render methods fit in the available time we have in order to render at a constant rate of 50 FPS (that is, `secsPerFrame` which is equal to 20 ms.).

Besides that, our computer may be prioritizing other tasks that prevent our game loop from executing for a certain period of time. So, we may end up updating our game state at very variable time steps which are not suitable for game physics.

Finally, sleep accuracy may range to tenth of a second, so we are not even updating at a constant frame rate even if our update and render methods take no time. So, as you see the problem is not so simple.

On the Internet you can find tons of variants for game loops. In this book we will use a not too complex approach that can work well in many situations. So let us move on and explain the basis for our game loop. The pattern used here is usually called Fixed Step Game Loop.

First of all we may want to control separately the period at which the game state is updated and the period at which the game is rendered to the screen. Why do we do this? Well, updating our game state at a constant rate is more important, especially if we use some physics engine. On the contrary, if our rendering is not done in time it makes no sense to render old frames while processing our game loop. We have the flexibility to skip some frames.

Let us have a look at how our game loop looks like:

```
double secsPerUpdate = 1.0d / 30.0d;
double previous = getTime();
double steps = 0.0;
while (true) {
  double loopStartTime = getTime();
  double elapsed = loopStartTime - previous;
  previous = current;
  steps += elapsed;

  handleInput();

  while (steps >= secsPerUpdate) {
    updateGameState();
    steps -= secsPerUpdate;
  }

  render();
  sync(current);
}
```

With this game loop we update our game state at fixed steps. But how do we control that we do not exhaust the computer's resources by rendering continuously? This is done in the sync method:

www.dbooks.org

```
private void sync(double loopStartTime) {
    float loopSlot = 1f / 50;
    double endTime = loopStartTime + loopSlot;
    while(getTime() < endTime) {
        try {
            Thread.sleep(1);
        } catch (InterruptedException ie) {}
    }
}
```

So what are we doing in the above method? In summary we calculate how many seconds our game loop iteration should last (which is stored in the `loopSlot` variable) and we wait for that amount of time taking into consideration the time we spent in our loop. But instead of doing a single wait for the whole available time period we do small waits. This will allow other tasks to run and will avoid the sleep accuracy problems we mentioned before. Then, what we do is:

1. Calculate the time at which we should exit this wait method and start another iteration of our game loop (which is the variable `endTime`).
2. Compare the current time with that end time and wait just one millisecond if we have not reached that time yet.

Now it is time to structure our code base in order to start writing our first version of our Game Engine. But before doing that we will talk about another way of controlling the rendering rate. In the code presented above, we are doing micro-sleeps in order to control how much time we need to wait. But we can choose another approach in order to limit the frame rate. We can use v-sync (vertical synchronization). The main purpose of v-sync is to avoid screen tearing. What is screen tearing? It's a visual effect that is produced when we update the video memory while it's being rendered. The result will be that part of the image will represent the previous image and the other part will represent the updated one. If we enable v-sync we won't send an image to the GPU while it is being rendered onto the screen.

When we enable v-sync we are synchronizing to the refresh rate of the video card, which at the end will result in a constant frame rate. This is done with the following line:

```
glfwSwapInterval(1);
```

With that line we are specifying that we must wait, at least, one screen update before drawing to the screen. In fact, we are not directly drawing to the screen. We instead store the information to a buffer and we swap it with this method:

```
glfwSwapBuffers(windowHandle);
```

So, if we enable v-sync we achieve a constant frame rate without performing the micro-sleeps to check the available time. Besides that, the frame rate will match the refresh rate of our graphics card. That is, if it's set to 60Hz (60 times per second), we will have 60 Frames Per Second. We can scale down that rate by setting a number higher than 1 in the `glfwSwapInterval` method (if we set it to 2, we would get 30 FPS).

Let's get back to reorganize the source code. First of all we will encapsulate all the GLFW Window initialization code in a class named `Window` allowing some basic parameterization of its characteristics (such as title and size). That `Window` class will also provide a method to detect key presses which will be used in our game loop:

```
public boolean isKeyPressed(int keyCode) {
    return glfwGetKey(windowHandle, keyCode) == GLFW_PRESS;
}
```

The `Window` class besides providing the initialization code also needs to be aware of resizing. So it needs to setup a callback that will be invoked whenever the window is resized. The callback will receive the width and height, in pixels, of the framebuffer (the rendering area, in this sample, the display area). If you want the width, height of the framebuffer in screen coordinates you may use the the `glfwSetWindowSizeCallback` method. Screen coordinates don't necessarilly correspond to pixels (for instance, on a Mac with Retina display. Since we are going to use that information when performing some OpenGL calls, we are interested in pixels not in screen coordinates. You can get more infomation in the GLFW documentation.

```
// Setup resize callback
glfwSetFramebufferSizeCallback(windowHandle, (window, width, height) -> {
    Window.this.width = width;
    Window.this.height = height;
    Window.this.setResized(true);
});
```

We will also create a `Renderer` class which will handle our game render logic. By now, it will just have an empty `init` method and another method to clear the screen with the configured clear color:

```
public void init() throws Exception {
}

public void clear() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}
```


www.dbooks.org

Then we will create an interface named `IGameLogic` which will encapsulate our game logic. By doing this we will make our game engine reusable across different titles. This interface will have methods to get the input, to update the game state and to render game-specific data.

```
public interface IGameLogic {

    void init() throws Exception;

    void input(Window window);

    void update(float interval);

    void render(Window window);
}
```

Then we will create a class named `GameEngine` which will contain our game loop code. This class will implement the `Runnable` interface since the game loop will be run inside a separate thread:

```
public class GameEngine implements Runnable {

    //..[Removed code]..

    private final Thread gameLoopThread;

    public GameEngine(String windowTitle, int width, int height, boolean vsSync, IGame
Logic gameLogic) throws Exception {
        gameLoopThread = new Thread(this, "GAME_LOOP_THREAD");
        window = new Window(windowTitle, width, height, vsSync);
        this.gameLogic = gameLogic;
        //..[Removed code]..
    }
```

The `vSync` parameter allows us to select if we want to use v-sync or not. You can see we create a new Thread which will execute the run method of our `GameEngine` class which will contain our game loop:

```
public void start() {
    gameLoopThread.start();
}

@Override
public void run() {
    try {
        init();
        gameLoop();
    } catch (Exception excp) {
        excp.printStackTrace();
    }
}
```

Our `GameEngine` class provides a start method which just starts our Thread so the run method will be executed asynchronously. That method will perform the initialization tasks and will run the game loop until our window is closed. It is very important to initialize GLFW inside the thread that is going to update it later. Thus, in that `init` method our Window and `Renderer` instances are initialized.

In the source code you will see that we created other auxiliary classes such as Timer (which will provide utility methods for calculating elapsed time) and will be used by our game loop logic.

Our `GameEngine` class just delegates the input and update methods to the `IGameLogic` instance. In the render method it delegates also to the `IGameLogic` instance and updates the window.

```
protected void input() {
    gameLogic.input(window);
}

protected void update(float interval) {
    gameLogic.update(interval);
}

protected void render() {
    gameLogic.render(window);
    window.update();
}
```

Our starting point, our class that contains the main method will just only create a `GameEngine` instance and start it.

www.dbooks.org

```
public class Main {

    public static void main(String[] args) {
        try {
            boolean vSync = true;
            IGameLogic gameLogic = new DummyGame();
            GameEngine gameEng = new GameEngine("GAME",
                600, 480, vSync, gameLogic);
            gameEng.start();
        } catch (Exception excp) {
            excp.printStackTrace();
            System.exit(-1);
        }
    }

}
```

At the end we only need to create or game logic class, which for this chapter will be a simpler one. It will just increase / decrease the clear color of the window whenever the user presses the up / down key. The render method will just clear the window with that color.

```
public class DummyGame implements IGameLogic {

    private int direction = 0;

    private float color = 0.0f;

    private final Renderer renderer;

    public DummyGame() {
        renderer = new Renderer();
    }

    @Override
    public void init() throws Exception {
        renderer.init();
    }

    @Override
    public void input(Window window) {
        if ( window.isKeyPressed(GLFW_KEY_UP) ) {
            direction = 1;
        } else if ( window.isKeyPressed(GLFW_KEY_DOWN) ) {
            direction = -1;
        } else {
            direction = 0;
        }
    }

    @Override
    public void update(float interval) {
        color += direction * 0.01f;
        if (color > 1) {
            color = 1.0f;
        } else if ( color < 0 ) {
            color = 0.0f;
        }
    }

    @Override
    public void render(Window window) {
        if ( window.isResized() ) {
            glViewport(0, 0, window.getWidth(), window.getHeight());
            window.setResized(false);
        }
        window.setClearColor(color, color, color, 0.0f);
        renderer.clear();
    }
}
```

In the `render` method we get notified when the window has been resized in order to update the viewport to locate the center of the coordinates to the center of the window.

www.dbooks.org

The class hierarchy that we have created will help us to separate our game engine code from the code of a specific game. Although it may seem unnecessary at this moment, we need to isolate generic tasks that every game will use from the state logic, artwork and resources of a specific game in order to reuse our game engine. In later chapters we will need to restructure this class hierarchy as our game engine gets more complex.

Threading issues

If you try to run the source code provided above in OSX you will get an error like this:

```
Exception in thread "GAME_LOOP_THREAD" java.lang.ExceptionInInitializerError
```

What does this mean? The answer is that some functions of the GLFW library cannot be called in a `Thread` which is not the main `Thread`. We are doing the initializing stuff, including window creation in the `init` method of the `GameEngine class`. That method gets called in the `run` method of the same class, which is invoked by a new `Thread` instead of the one that's used to launch the program.

This is a constraint of the GLFW library and basically it implies that we should avoid the creation of new Threads for the game loop. We could try to create all the Windows related stuff in the main thread but we will not be able to render anything. The problem is that, OpenGL calls need to be performed in the same `Thread` that its context was created.

On Windows and Linux platforms, although we are not using the main thread to initialize the GLFW stuff the samples will work. The problem is with OSX, so we need to change the source code of the `run` method of the `GameEngine` class to support that platform like this:

```
public void start() {
    String osName = System.getProperty("os.name");
    if ( osName.contains("Mac") ) {
        gameLoopThread.run();
    } else {
        gameLoopThread.start();
    }
}
```

What we are doing is just ignoring the game loop thread when we are in OSX and execute the game loop code directly in the main Thread. This is not a perfect solution but it will allow you to run the samples on Mac. Other solutions found in the forums (such as executing the JVM with the `-XstartOnFirstThread` flag seem to not work).

In the future it may be interesting to explore if LWJGL provides other GUI libraries to check if this restriction applies to them. (Many thanks to Timo Bühlmann for pointing out this issue).

Platform Differences (OSX)

You will be able to run the code described above on Windows or Linux, but we still need to do some modifications for OSX. As it's stated in th GLFW documentation:

> The only OpenGL 3.x and 4.x contexts currently supported by OS X are forward-compatible, core profile contexts. The supported versions are 3.2 on 10.7 Lion and 3.3 and 4.1 on 10.9 Mavericks. In all cases, your GPU needs to support the specified OpenGL version for context creation to succeed.

So, in order to support features explained in later chapters we need to add these lines to the `Window` class before the window is created:

```
        glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
        glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
        glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
        glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
```

This will make the program use the highest OpenGL version possible between 3.2 and 4.1. If those lines are not included, a Legacy version of OpenGL is used.

www.dbooks.org

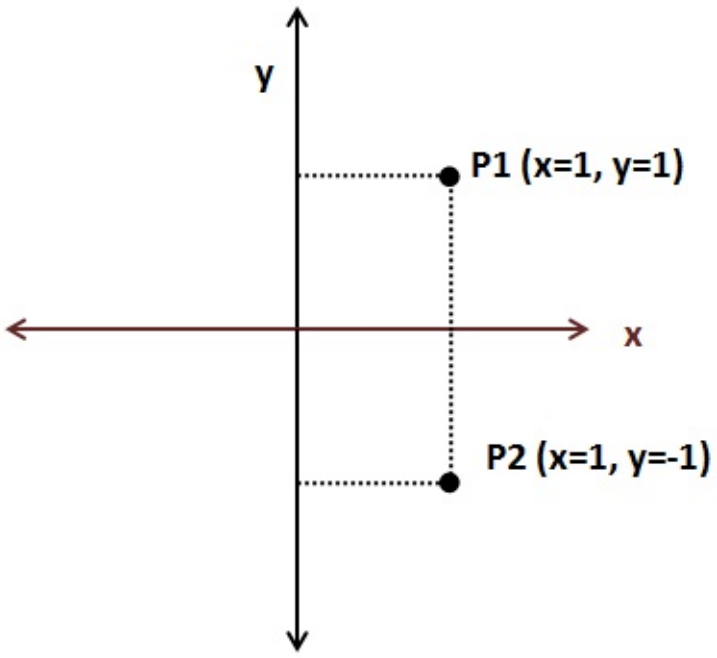
A brief about coordinates

In this chapter we will talk a little bit about coordinates and coordinate systems trying to introduce some fundamental mathematical concepts in a simple way to support the techniques and topics that we will address in subsequent chapters. We will assume some simplifications which may sacrifice preciseness for the sake of legibility.

We locate objects in space by specifying its coordinates. Think about a map. You specify a point on a map by stating its latitude or longitude. With just a pair of numbers a point is precisely identified. That pair of numbers are the point coordinates (things are a little bit more complex in reality, since a map is a projection of a non perfect ellipsoid, the earth, so more data is needed but it's a good analogy).

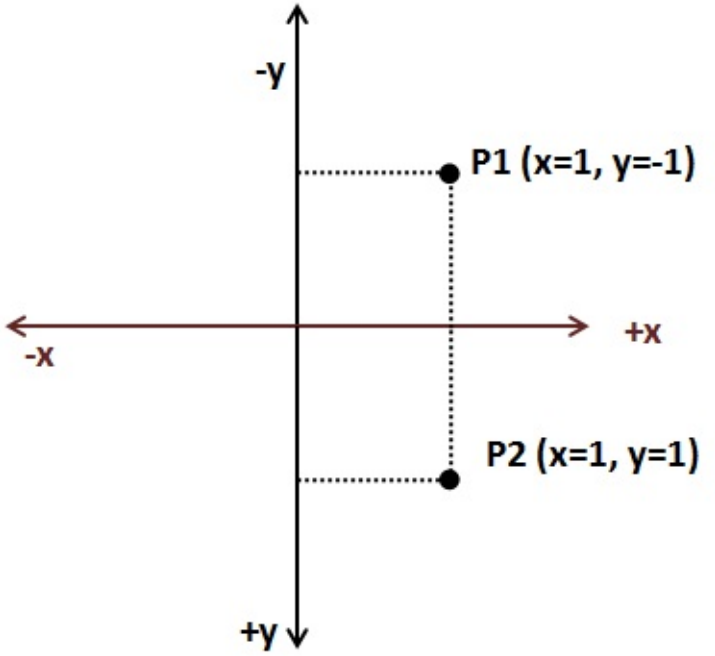
A coordinate system is a system which employs one or more numbers, that is, one or more coordinates to uniquely specify the position of a point. There are different coordinate systems (Cartesian, polar, etc.) and you can transform coordinates from one system to another. We will use the Cartesian coordinate system.

In the Cartesian coordinate system, for two dimensions, a coordinate is defined by two numbers that measure the signed distance to two perpendicular axes, x and y.



Continuing with the map analogy, coordinate systems define an origin. For geographic coordinates the origin is set to the point where the equator and the zero meridian cross. Depending on where we set the origin, coordinates for a specific point are different. A coordinate system may also define the orientation of the axis. In the previous figure, the x

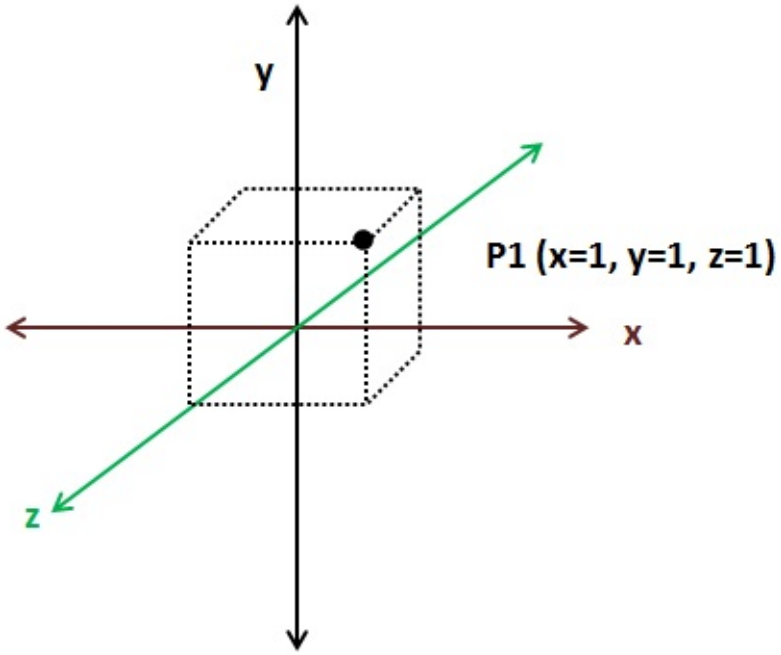
coordinate increases as long as we move to the right and the y coordinate increases as we move upwards. But, we could also define an alternative Cartesian coordinate system with different axis orientation in which we would obtain different coordinates.



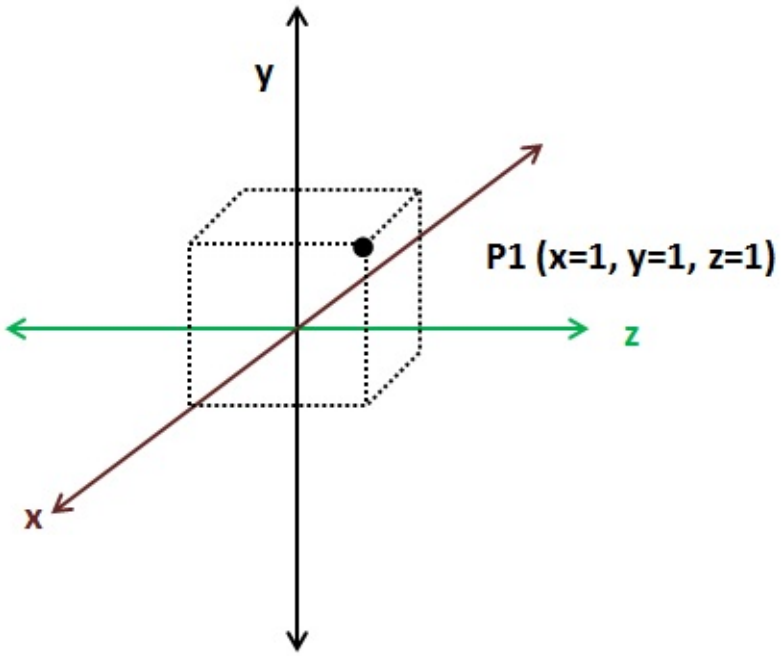
As you can see we need to define some arbitrary parameters, such as the origin and the axis orientation in order to give the appropriate meaning to the pair of numbers that constitute a coordinate. We will refer to that coordinate system with the set of arbitrary parameters as the coordinate space. In order to work with a set of coordinates we must use the same coordinate space. The good news is that we can transforms coordinates from one space to another just by performing translations and rotations.

If we are dealing with 3D coordinates we need an additional axis, the z axis. 3D coordinates will be formed by a set of three numbers (x, y, z).

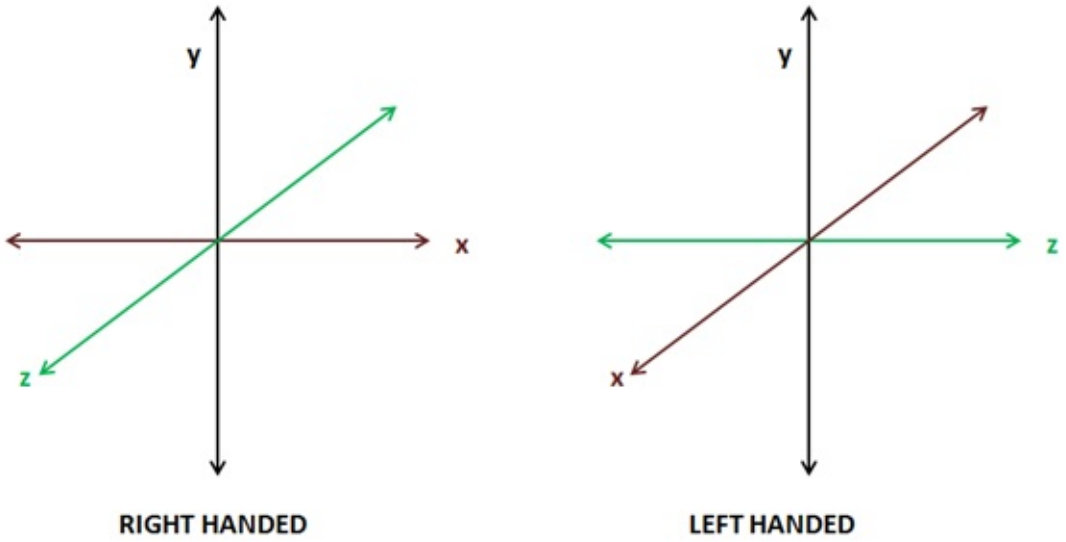
www.dbooks.org



As in 2D Cartesian coordinate spaces we can change the orientation of the axes in 3D coordinate spaces as long as the axes are perpendicular. The next figure shows another 3D coordinate space.



3D coordinates can be classified in two types: left handed and right handed. How do you know which type it is? Take your hand and form a “L” between your thumb and your index fingers, the middle finger should point in a direction perpendicular to the other two. The thumb should point to the direction where the x axis increases, the index finger should point where the y axis increases and the middle finger should point where the z axis increases. If you are able to do that with your left hand, then its left handed, if you need to use your right hand is right-handed.



2D coordinate spaces are all equivalent since by applying rotation we can transform from one to another. 3D coordinate spaces, on the contrary, are not all equal. You can only transform from one to another by applying rotation if they both have the same handedness, that is, if both are left handed or right handed.

Now that we have defined some basic topics let's talk about some commonly used terms when dealing with 3D graphics. When we explain in later chapters how to render 3D models we will see that we use different 3D coordinate spaces, that is because each of those coordinate spaces has a context, a purpose. A set of coordinates is meaningless unless it refers to something. When you examine this coordinates (40.438031, -3.676626) they may say something to you or not. But if I say that they are geometric coordinates (latitude and longitude) you will see that they are the coordinates of a place in Madrid.

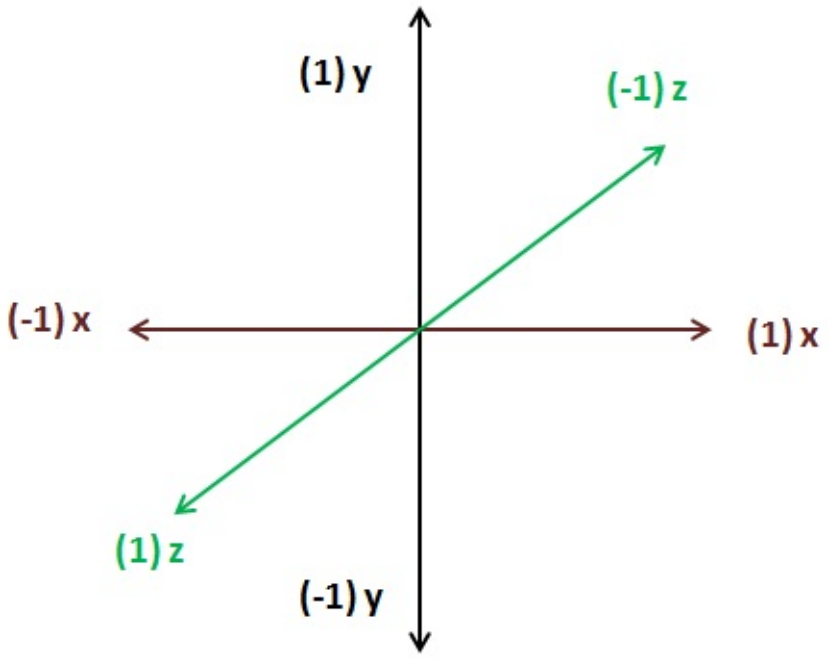
When we will load 3D objects we will get a set of 3D coordinates. Those coordinates are expressed in a 3D coordinate space which is called object coordinate space. When the graphics designers are creating those 3D models they don't know anything about the 3D scene that this model will be displayed in, so they can only define the coordinates using a coordinate space that is only relevant for the model.

When we will be drawing a 3D scene all of our 3D objects will be relative to the so called world space coordinate space. We will need to transform from 3D object space to world space coordinates. Some objects will need to be rotated, stretched or enlarged and translated in order to be displayed properly in a 3D scene.

We will also need to restrict the range of the 3D space that is shown, which is like moving a camera through our 3D space. Then we will need to transform world space coordinates to camera or view space coordinates. Finally these coordinates need to be transformed to screen coordinates, which are 2D, so we need to project 3D view coordinates to a 2D screen coordinate space.

www.dbooks.org

The following picture shows OpenGL coordinates, (the z axis is perpendicular to the screen) and coordinates are between -1 and +1.



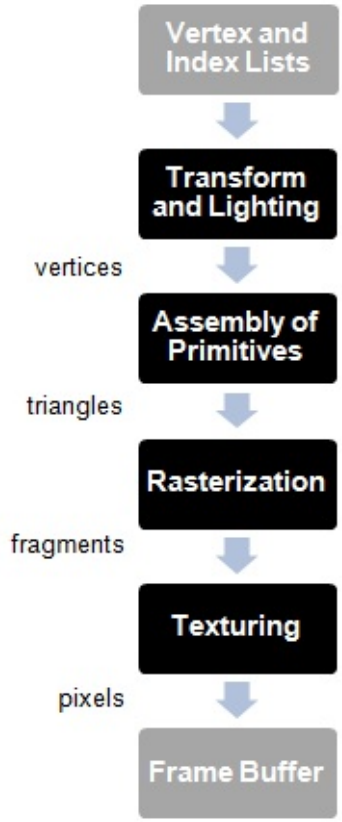
Don’t worry if you don’t have a clear understanding of all these concepts. They will be revisited during next chapters with practical examples.

Rendering

In this chapter we will learn the processes that takes place while rendering a scene using OpenGL. If you are used to older versions of OpenGL, that is fixed-function pipeline, you may end this chapter wondering why it needs to be so complex. You may end up thinking that drawing a simple shape to the screen should not require so many concepts and lines of code. Let me give you an advice for those of you that think that way. It is actually simpler and much more flexible. You only need to give it a chance. Modern OpenGL lets you think in one problem at a time and it lets you organize your code and processes in a more logical way.

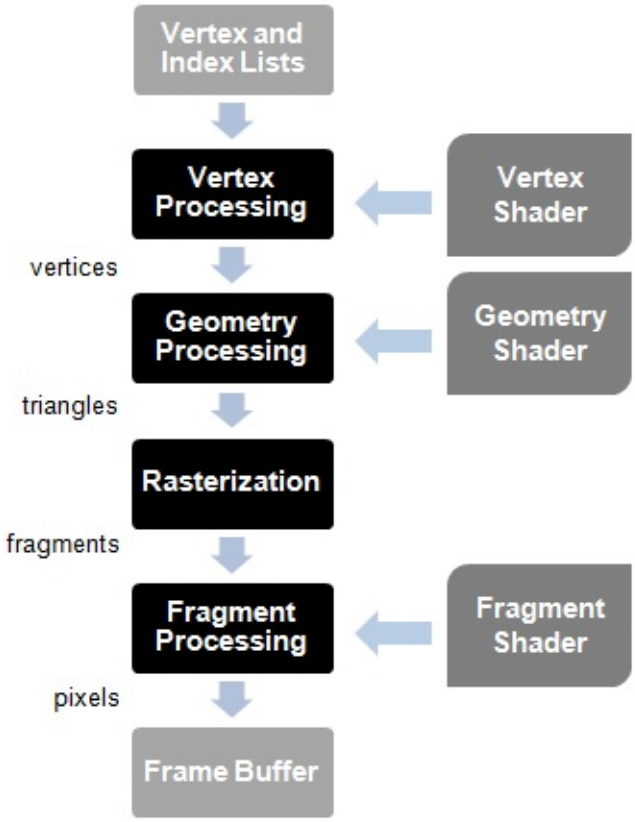
The sequence of steps that ends up drawing a 3D representation into your 2D screen is called the graphics pipeline. First versions of OpenGL employed a model which was called fixed-function pipeline. This model employed a set of steps in the rendering process which defined a fixed set of operations. The programmer was constrained to the set of functions available for each step. Thus, the effects and operations that could be applied were limited by the API itself (for instance, “set fog” or “add light”, but the implementation of those functions were fixed and could not be changed).

The graphics pipeline was composed of these steps:



www.dbooks.org

OpenGL 2.0 introduced the concept of programmable pipeline. In this model, the different steps that compose the graphics pipeline can be controlled or programmed by using a set of specific programs called shaders. The following picture depicts a simplified version of the OpenGL programmable pipeline:



The rendering starts taking as its input a list of vertices in the form of Vertex Buffers. But, what is a vertex? A vertex is a data structure that describes a point in 2D or 3D space. And how do you describe a point in a 3D space? By specifying its x, y and z coordinates. And what is a Vertex Buffer? A Vertex Buffer is another data structure that packs all the vertices that need to be rendered, by using vertex arrays, and makes that information available to the shaders in the graphics pipeline.

Those vertices are processed by the vertex shader whose main purpose is to calculate the projected position of each vertex into the screen space. This shader can generate also other outputs related to colour or texture, but its main goal is to project the vertices into the screen space, that is, to generate dots.

The geometry processing stage connects the vertices that are transformed by the vertex shader to form triangles. It does so by taking into consideration the order in which the vertices were stored and grouping them using different models. Why triangles? A triangle is like the basic work unit for graphic cards. It’s a simple geometric shape that can be combined and transformed to construct complex 3D scenes. This stage can also use a specific shader to group the vertices.

The rasterization stage takes the triangles generated in the previous stages, clips them and transforms them into pixel-sized fragments.

Those fragments are used during the fragment processing stage by the fragment shader to generate pixels assigning them the final color that gets written into the framebuffer. The framebuffer is the final result of the graphics pipeline. It holds the value of each pixel that should be drawn to the screen.

Keep in mind that 3D cards are designed to parallelize all the operations described above. The input data can be processes in parallel in order to generate the final scene.

So let's start writing our first shader program. Shaders are written by using the GLSL language (OpenGL Shading Language) which is based on ANSI C. First we will create a file named “ `vertex.vs` ” (The extension is for Vertex Shader) under the resources directory with the following content:

```
#version 330

layout (location=0) in vec3 position;

void main()
{
    gl_Position = vec4(position, 1.0);
}
```

The first line is a directive that states the version of the GLSL language we are using. The following table relates the GLSL version, the OpenGL that matches that version and the directive to use (Wikipedia: https://en.wikipedia.org/wiki/OpenGL_Shading_Language#Versions).

www.dbooks.org

GLS Version	OpenGL Version	Shader Preprocessor
1.10.59	2.0	#version 110
1.20.8	2.1	#version 120
1.30.10	3.0	#version 130
1.40.08	3.1	#version 140
1.50.11	3.2	#version 150
3.30.6	3.3	#version 330
4.00.9	4.0	#version 400
4.10.6	4.1	#version 410
4.20.11	4.2	#version 420
4.30.8	4.3	#version 430
4.40	4.4	#version 440
4.50	4.5	#version 450

The second line specifies the input format for this shader. Data in an OpenGL buffer can be whatever we want, that is, the language does not force you to pass a specific data structure with a predefined semantic. From the point of view of the shader it is expecting to receive a buffer with data. It can be a position, a position with some additional information or whatever we want. The vertex shader is just receiving an array of floats. When we fill the buffer, we define the buffer chunks that are going to be processed by the shader.

So, first we need to get that chunk into something that's meaningful to us. In this case we are saying that, starting from the position 0, we are expecting to receive a vector composed of 3 attributes (x, y, z).

The shader has a main block like any other C program which in this case is very simple. It is just returning the received position in the output variable `gl_Position` without applying any transformation. You now may be wondering why the vector of three attributes has been converted into a vector of four attributes (vec4). This is because `gl_Position` is expecting the result in vec4 format since it is using homogeneous coordinates. That is, it's expecting something in the form (x, y, z, w), where w represents an extra dimension. Why add another dimension? In later chapters you will see that most of the operations we need to do are based on vectors and matrices. Some of those operations cannot be combined if we do not have that extra dimension. For instance we could not combine rotation and translation operations. (If you want to learn more on this, this extra dimension allow us to combine affine and linear transformations. You can learn more about this by reading the excellent book "3D Math Primer for Graphics and Game development, by Fletcher Dunn and Ian Parberry).

Let us now have a look at our first fragment shader. We will create a file named "`fragment.fs`" (The extension is for Fragment Shader) under the resources directory with the following content:

```
#version 330

out vec4 fragColor;

void main()
{
    fragColor = vec4(0.0, 0.5, 0.5, 1.0);
}
```

The structure is quite similar to our vertex shader. In this case we will set a fixed colour for each fragment. The output variable is defined in the second line and set as a vec4 fragColor. Now that we have our shaders created, how do we use them? This is the sequence of steps we need to follow:

1. Create a OpenGL Program
2. Load the vertex and fragment shader code files.
3. For each shader, create a new shader program and specify its type (vertex, fragment).
4. Compile the shader.
5. Attach the shader to the program.
6. Link the program.

At the end the shader will be loaded in the graphics card and we can use it by referencing an identifier, the program identifier.

```
package org.lwjglb.engine.graph;

import static org.lwjgl.opengl.GL20.*;

public class ShaderProgram {

    private final int programId;

    private int vertexShaderId;

    private int fragmentShaderId;

    public ShaderProgram() throws Exception {
        programId = glCreateProgram();
        if (programId == 0) {
            throw new Exception("Could not create Shader");
        }
    }
```


www.dbooks.org

```
    public void createVertexShader(String shaderCode) throws Exception {
        vertexShaderId = createShader(shaderCode, GL_VERTEX_SHADER);
    }

    public void createFragmentShader(String shaderCode) throws Exception {
        fragmentShaderId = createShader(shaderCode, GL_FRAGMENT_SHADER);
    }

    protected int createShader(String shaderCode, int shaderType) throws Exception {
        int shaderId = glCreateShader(shaderType);
        if (shaderId == 0) {
            throw new Exception("Error creating shader. Type: " + shaderType);
        }

        glShaderSource(shaderId, shaderCode);
        glCompileShader(shaderId);

        if (glGetShaderi(shaderId, GL_COMPILE_STATUS) == 0) {
            throw new Exception("Error compiling Shader code: " + glGetShaderInfoLog(s
haderId, 1024));
        }

        glAttachShader(programId, shaderId);

        return shaderId;
    }

    public void link() throws Exception {
        glLinkProgram(programId);
        if (glGetProgrami(programId, GL_LINK_STATUS) == 0) {
            throw new Exception("Error linking Shader code: " + glGetProgramInfoLog(pr
ogramId, 1024));
        }

        if (vertexShaderId != 0) {
            glDetachShader(programId, vertexShaderId);
        }
        if (fragmentShaderId != 0) {
            glDetachShader(programId, fragmentShaderId);
        }

        glValidateProgram(programId);
        if (glGetProgrami(programId, GL_VALIDATE_STATUS) == 0) {
            System.err.println("Warning validating Shader code: " + glGetProgramInfoLo
g(programId, 1024));
        }

    }

    public void bind() {
        glUseProgram(programId);
    }
```

```
    public void unbind() {
        glUseProgram(0);
    }

    public void cleanup() {
        unbind();
        if (programId != 0) {
            glDeleteProgram(programId);
        }
    }
}
```

The constructor of the `ShaderProgram` creates a new program in OpenGL and provides methods to add vertex and fragment shaders. Those shaders are compiled and attached to the OpenGL program. When all shaders are attached the link method should be invoked which links all the code and verifies that everything has been done correctly.

Once the shader program has been linked, the compiled vertex and fragment shaders can be freed up (by calling `glDetachShader`)

Regarding verification, this is done through the `glValidateProgram` call. This method is used mainly for debugging purposes, and it should be removed when your game reaches production stage. This method tries to validate if the shader is correct given the **current OpenGL state**. This means, that validation may fail in some cases even if the shader is correct, due to the fact that the current state is not complete enough to run the shader (some data may have not been uploaded yet). So, instead of failing, we just print an error message to the standard error output.

`ShaderProgram` also provides methods to activate this program for rendering (bind) and to stop using it (unbind). Finally it provides a cleanup method to free all the resources when they are no longer needed.

Since we have a cleanup method, let us change our `IGameLogic` interface class to add a cleanup method:

```
void cleanup();
```

This method will be invoked when the game loop finishes, so we need to modify the run method of the `GameEngine` class:

www.dbooks.org

```
@Override
public void run() {
    try {
        init();
        gameLoop();
    } catch (Exception excp) {
        excp.printStackTrace();
    } finally {
        cleanup();
    }
}
```

Now we can use our shaders in order to display a triangle. We will do this in the `init` method of our `Renderer` class. First of all, we create the shader program:

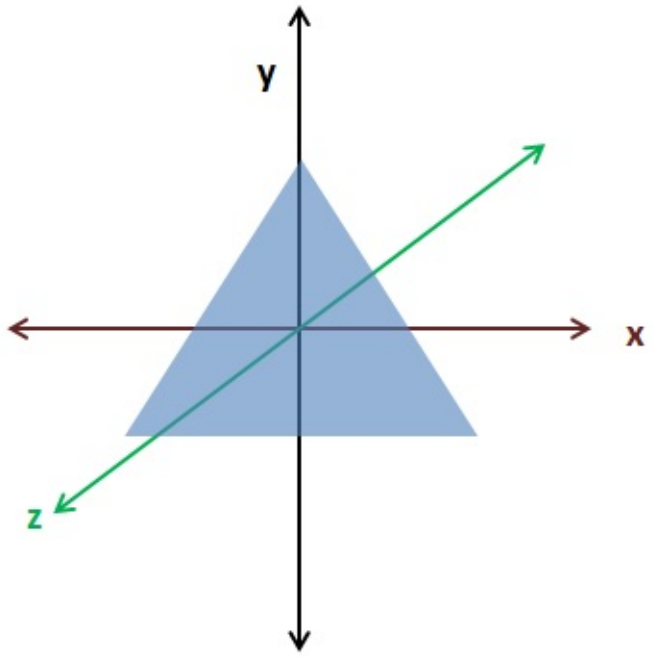
```
public void init() throws Exception {
    shaderProgram = new ShaderProgram();
    shaderProgram.createVertexShader(Utils.loadResource("/vertex.vs"));
    shaderProgram.createFragmentShader(Utils.loadResource("/fragment.fs"));
    shaderProgram.link();
}
```

We have created a utility class which by now provides a method to retrieve the contents of a file from the class path. This method is used to retrieve the contents of our shaders.

Now we can define our triangle as an array of floats. We create a single float array which will define the vertices of the triangle. As you can see there's no structure in that array. As it is right now, OpenGL cannot know the structure of that data. It's just a sequence of floats:

```
float[] vertices = new float[]{
     0.0f,  0.5f, 0.0f,
    -0.5f, -0.5f, 0.0f,
     0.5f, -0.5f, 0.0f
};
```

The following picture depicts the triangle in our coordinates system.



Now that we have our coordinates, we need to store them into our graphics card and tell OpenGL about the structure. We will introduce now two important concepts, Vertex Array Objects (VAOs) and Vertex Buffer Object (VBOs). If you get lost in the next code fragments remember that at the end what we are doing is sending the data that models the objects we want to draw to the graphics card memory. When we store it we get an identifier that serves us later to refer to it while drawing.

Let us first start with Vertex Buffer Object (VBOs). A VBO is just a memory buffer stored in the graphics card memory that stores vertices. This is where we will transfer our array of floats that model a triangle. As we said before, OpenGL does not know anything about our data structure. In fact it can hold not just coordinates but other information, such as textures, colour, etc.
A Vertex Array Objects (VAOs) is an object that contains one or more VBOs which are usually called attribute lists. Each attribute list can hold one type of data: position, colour, texture, etc. You are free to store whichever you want in each slot.

A VAO is like a wrapper that groups a set of definitions for the data that is going to be stored in the graphics card. When we create a VAO we get an identifier. We use that identifier to render it and the elements it contains using the definitions we specified during its creation.

So let us continue coding our example. The first thing that we must do is to store our array of floats into a `FloatBuffer`. This is mainly due to the fact that we must interface with the OpenGL library, which is C-based, so we must transform our array of floats into something that can be managed by the library.

```
FloatBuffer verticesBuffer = MemoryUtil.memAllocFloat(vertices.length);
verticesBuffer.put(vertices).flip();
```


www.dbooks.org

We use the `MemoryUtil` class to create the buffer in off-heap memory so that it's accessible by the OpenGL library. After we have stored the data (with the put method) we need to reset the position of the buffer to the 0 position with the flip method (that is, we say that we've finishing writing to it). Remember, that Java objects, are allocated in a space called the heap. The heap is a large bunch of memory reserved in the JVM's process memory. Memory stored in the heap cannot be accessed by native code (JNI, the mechanism that allows calling native code from Java does not allow that). The only way of sharing memory data between Java and native code is by directly allocating memory in Java.

If you come from previous versions of LWJGL it's important to stress out a few topics. You may have noticed that we do not use the utility class `BufferUtils` to create the buffers. Instead we use the `MemoryUtil` class. This is due to the fact that `BufferUtils` was not very efficient, and has been mantained only for backwards compatibility. Instead, LWJGL 3 proposes two methods for buffer management:

- Auto-managed buffers, that is, buffers that are automatically collected by the Garbage Collector. These buffers are mainly used for short lived operations, or for data that is transferred to the GPU and does not need to be present in the process memory. This is achieved by using the `org.lwjgl.system.MemoryStack` class.
- Manually managed buffers. In this case we need to carefulley free them once we are finished. These buffers are intended for long time operations or for large amounts of data. This is achieved by using the `MemoryUtil` class.

You can consult the details here: https://blog.lwjgl.org/memory-management-in-lwjgl-3/.

In this case, our data is sent to the GPU so we could use auto-managed buffers. But since, later on, we will use them to hold potentially large volumes of data we will need to manually manage them. This is the reason why we are using the `MemoryUtil` class and thus, why we are freeing the buffer in a finally block. In next chapters we will learn how to use auto-managed buffers.

Now we need to create the VAO and bind it.

```
vaoId = glGenVertexArrays();
glBindVertexArray(vaoId);
```

Then we need to create the VBO, bind it and put the data into it.

```
vboId = glGenBuffers();
glBindBuffer(GL_ARRAY_BUFFER, vboId);
glBufferData(GL_ARRAY_BUFFER, verticesBuffer, GL_STATIC_DRAW);
memFree(verticesBuffer);
```

Now comes the most important part. We need to define the structure of our data and store it in one of the attribute lists of the VAO. This is done with the following line.

```
glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
```

The parameters are:

- index: Specifies the location where the shader expects this data.
- size: Specifies the number of components per vertex attribute (from 1 to 4). In this case, we are passing 3D coordinates, so it should be 3.
- type: Specifies the type of each component in the array, in this case a float.
- normalized: Specifies if the values should be normalized or not.
- stride: Specifies the byte offset between consecutive generic vertex attributes. (We will explain it later).
- offset: Specifies an offset to the first component in the buffer.

After we are finished with our VBO we can unbind it and the VAO (bind them to 0)

```
// Unbind the VBO
glBindBuffer(GL_ARRAY_BUFFER, 0);

// Unbind the VAO
glBindVertexArray(0);
```

Once this has been completed we **must** free the off-heap memory that was allocated by the FloatBuffer. This is done by manually calling memFree, as Java garbage collection will not clean up off-heap allocations.

```
if (verticesBuffer != null) {
    MemoryUtil.memFree(verticesBuffer);
}
```

That's all the code that should be in our `init` method. Our data is already in the graphics card, ready to be used. We only need to modify our `render` method to use it each render step during our game loop.

www.dbooks.org

```
public void render(Window window) {
    clear();

    if ( window.isResized() ) {
        glViewport(0, 0, window.getWidth(), window.getHeight());
        window.setResized(false);
    }

    shaderProgram.bind();

    // Bind to the VAO
    glBindVertexArray(vaoId);
    glEnableVertexAttribArray(0);

    // Draw the vertices
    glDrawArrays(GL_TRIANGLES, 0, 3);

    // Restore state
    glDisableVertexAttribArray(0);
    glBindVertexArray(0);

    shaderProgram.unbind();
}
```

As you can see we just clear the window, bind the shader program, bind the VAO, draw the vertices stored in the VBO associated to the VAO and restore the state. That's it.

We also added a cleanup method to our `Renderer` class which frees acquired resources.

```
public void cleanup() {
    if (shaderProgram != null) {
        shaderProgram.cleanup();
    }

    glDisableVertexAttribArray(0);

    // Delete the VBO
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glDeleteBuffers(vboId);

    // Delete the VAO
    glBindVertexArray(0);
    glDeleteVertexArrays(vaoId);
}
```

And, that's all! If you followed the steps carefully you will see something like this.

Our first triangle! You may think that this will not make it into the top ten game list, and you will be totally right. You may also think that this has been too much work for drawing a boring triangle. But keep in mind that we are introducing key concepts and preparing the base infrastructure to do more complex things. Please be patient and continue reading.

www.dbooks.org

More on Rendering

In this chapter we will continue talking about how OpenGL renders things. In order to tidy up our code a little bit let's create a new class called Mesh which, taking as an input an array of positions, creates the VBO and VAO objects needed to load that model into the graphics card.

```
package org.lwjglb.engine.graph;

import java.nio.FloatBuffer;
import static org.lwjgl.opengl.GL11.*;
import static org.lwjgl.opengl.GL15.*;
import static org.lwjgl.opengl.GL20.*;
import static org.lwjgl.opengl.GL30.*;
import org.lwjgl.system.MemoryUtil;

public class Mesh {

    private final int vaoId;

    private final int vboId;

    private final int vertexCount;

    public Mesh(float[] positions) {
        FloatBuffer verticesBuffer = null;
        try {
            verticesBuffer = MemoryUtil.memAllocFloat(positions.length);
            vertexCount = positions.length / 3;
            verticesBuffer.put(positions).flip();

            vaoId = glGenVertexArrays();
            glBindVertexArray(vaoId);

            vboId = glGenBuffers();
            glBindBuffer(GL_ARRAY_BUFFER, vboId);
            glBufferData(GL_ARRAY_BUFFER, verticesBuffer, GL_STATIC_DRAW);
            glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
            glBindBuffer(GL_ARRAY_BUFFER, 0);

            glBindVertexArray(0);
        } finally {
            if (verticesBuffer  != null) {
                MemoryUtil.memFree(verticesBuffer);
            }
        }
    }
```

```
    public int getVaoId() {
        return vaoId;
    }

    public int getVertexCount() {
        return vertexCount;
    }

    public void cleanUp() {
        glDisableVertexAttribArray(0);

        // Delete the VBO
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        glDeleteBuffers(vboId);

        // Delete the VAO
        glBindVertexArray(0);
        glDeleteVertexArrays(vaoId);
    }
}
```

We will create our `Mesh` instance in our `DummyGame` class, removing the VAO and VBO code from `Renderer` `init` method. Our render method in the `Renderer` class will accept also a Mesh instance to render. The `cleanup` method will also be simplified since the `Mesh` class already provides one for freeing VAO and VBO resources.

```
public void render(Mesh mesh) {
    clear();

    if ( window.isResized() ) {
        glViewport(0, 0, window.getWidth(), window.getHeight());
        window.setResized(false);
    }

    shaderProgram.bind();

    // Draw the mesh
    glBindVertexArray(mesh.getVaoId());
    glEnableVertexAttribArray(0);
    glDrawArrays(GL_TRIANGLES, 0, mesh.getVertexCount());

    // Restore state
    glDisableVertexAttribArray(0);
    glBindVertexArray(0);

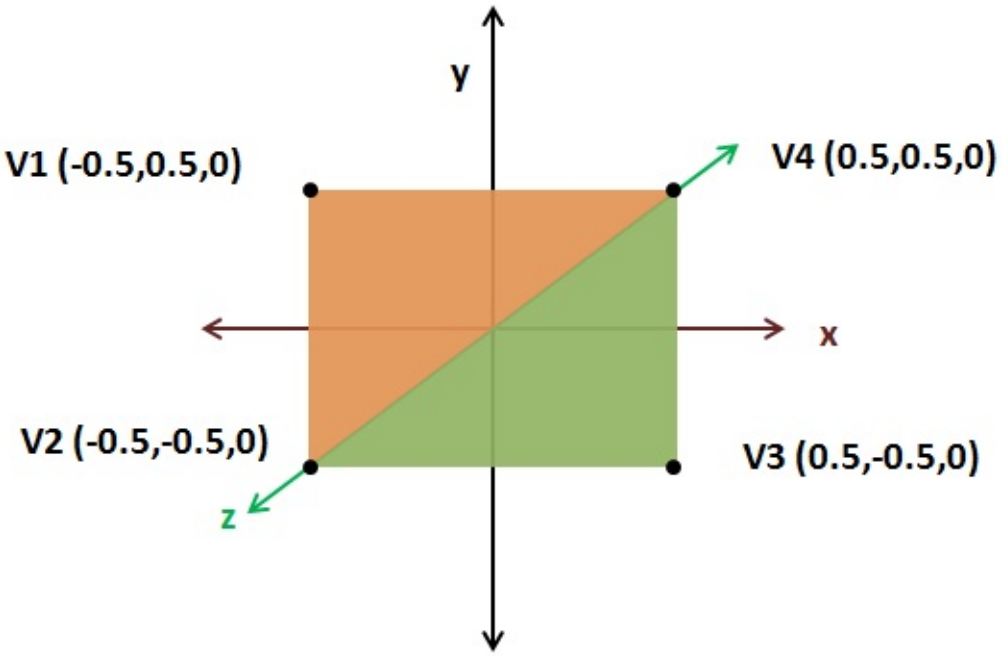
    shaderProgram.unbind();
}

public void cleanup() {
    if (shaderProgram != null) {
        shaderProgram.cleanup();
    }
}
```

One important thing to note is this line:

```
glDrawArrays(GL_TRIANGLES, 0, mesh.getVertexCount());
```

Our `Mesh` counts the number of vertices by dividing the position array by 3 (since we are passing X, Y and Z coordinates)). Now that we can render more complex shapes, let us try to render a more complex shape. Let us render a quad. A quad can be constructed by using two triangles as shown in the next figure.

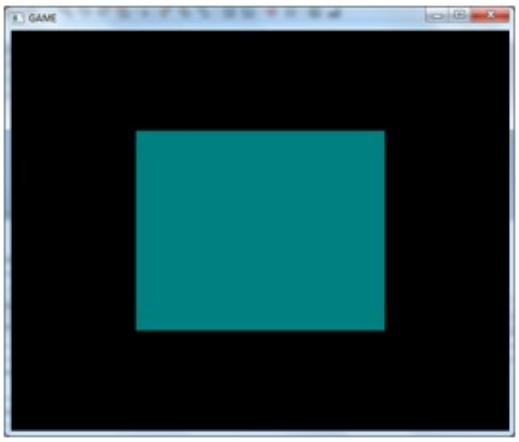


As you can see each of the two triangles is composed of three vertices. The first one formed by the vertices V1, V2 and V4 (the orange one) and the second one formed by the vertices V4, V2 and V3 (the green one). Vertices are specified in a counter-clockwise order, so the float array to be passed will be [V1, V2, V4, V4, V2, V3]. Thus, the init method in our `DummyGame` class will be:

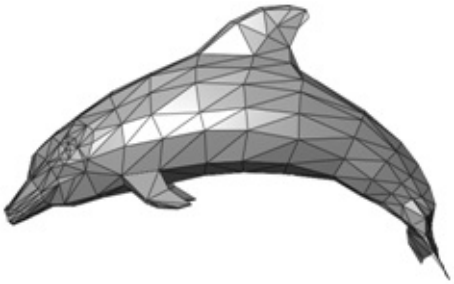
```
@Override
public void init() throws Exception {
    renderer.init();
    float[] positions = new float[]{
        -0.5f,  0.5f, 0.0f,
        -0.5f, -0.5f, 0.0f,
         0.5f,  0.5f, 0.0f,
         0.5f,  0.5f, 0.0f,
        -0.5f, -0.5f, 0.0f,
         0.5f, -0.5f, 0.0f,
    };
    mesh = new Mesh(positions);
}
```

Now you should see a quad rendered like this:

www.dbooks.org



Are we done yet? Unfortunately not. The code above still presents some issues. We are repeating coordinates to represent the quad. We are passing twice V2 and V4 coordinates. With this small shape it may not seem a big deal, but imagine a much more complex 3D model. We would be repeating the coordinates many times. Keep in mind also that now we are just using three floats for representing the position of a vertex. But later on we will need more data to represent the texture, etc. Also take into consideration that in more complex shapes the number of vertices shared between triangles can be even higher like in the figure below (where a vertex can be shared between six triangles).



At the end we would need much more memory because of that duplicate information and this is where Index Buffers come to the rescue. For drawing the quad we only need to specify each vertex once this way: V1, V2, V3, V4). Each vertex has a position in the array. V1 has position 0, V2 has position 1, etc:

V1	V2	V3	V4
0	1	2	3

Then we specify the order in which those vertices should be drawn by referring to their position:

0	1	3	3	1	2
V1	V2	V4	V4	V2	V3

So we need to modify our `Mesh` class to accept another parameter, an array of indices, and now the number of vertices to draw will be the length of that indices array.

```
public Mesh(float[] positions, int[] indices) {
    vertexCount = indices.length;
```

After we have created our VBO that stores the positions, we need to create another VBO which will hold the indices. So we rename the identifier that holds the identifier for the positions VBO and create a new one for the index VBO (`idxVboId`). The process of creating that VBO is similar but the type is now `GL_ELEMENT_ARRAY_BUFFER` .

```
idxVboId = glGenBuffers();
indicesBuffer = MemoryUtil.memAllocInt(indices.length);
indicesBuffer.put(indices).flip();
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, idxVboId);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indicesBuffer, GL_STATIC_DRAW);
memFree(indicesBuffer);
```

Since we are dealing with integers we need to create an `IntBuffer` instead of a `FloatBuffer` .

And that's it. The VAO will contain now two VBOs, one for positions and another one that will hold the indices and that will be used for rendering. Our cleanup method in our `Mesh` class must take into consideration that there is another VBO to free.

```
public void cleanUp() {
    glDisableVertexAttribArray(0);

    // Delete the VBOs
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glDeleteBuffers(posVboId);
    glDeleteBuffers(idxVboId);

    // Delete the VAO
    glBindVertexArray(0);
    glDeleteVertexArrays(vaoId);
}
```

Finally, we need to modify our drawing call that used the `glDrawArrays` method:

```
glDrawArrays(GL_TRIANGLES, 0, mesh. getVertexCount());
```

To another call that uses the method `glDrawElements` :

```
glDrawElements(GL_TRIANGLES, mesh.getVertexCount(), GL_UNSIGNED_INT, 0);
```

The parameters of that method are:

www.dbooks.org

- mode: Specifies the primitives for rendering, triangles in this case. No changes here.
- count: Specifies the number of elements to be rendered.
- type: Specifies the type of value in the indices data. In this case we are using integers.
- indices: Specifies the offset to apply to the indices data to start rendering.

And now we can use our newer and much more efficient method of drawing complex models by just specifying the indices.

```
public void init() throws Exception {
    renderer.init();
    float[] positions = new float[]{
        -0.5f,  0.5f, 0.0f,
        -0.5f, -0.5f, 0.0f,
         0.5f, -0.5f, 0.0f,
         0.5f,  0.5f, 0.0f,
    };
    int[] indices = new int[]{
        0, 1, 3, 3, 1, 2,
    };
    mesh = new Mesh(positions, indices);
}
```

Now let's add some colour to our example. We will pass another array of floats to our `Mesh` class which holds the colour for each coordinate in the quad.

```
public Mesh(float[] positions, float[] colours, int[] indices) {
```

With that array, we will create another VBO which will be associated to our VAO.

```
// Colour VBO
colourVboId = glGenBuffers();
FloatBuffer colourBuffer = memAllocFloat(colours.length);
colourBuffer.put(colours).flip();
glBindBuffer(GL_ARRAY_BUFFER, colourVboId);
glBufferData(GL_ARRAY_BUFFER, colourBuffer, GL_STATIC_DRAW);
memFree(colourBuffer);
glVertexAttribPointer(1, 3, GL_FLOAT, false, 0, 0);
```

Please notice that in the `glVertexAttribPointer` call, the first parameter is now a `"1"`. This is the location where our shader will be expecting that data. (Of course, since we have another VBO we need to free it in the `cleanup` method).

The next step is to modify the shaders. The vertex shader is now expecting two parameters, the coordinates (in location 0) and the colour (in location 1). The vertex shader will just output the received colour so it can be processed by the fragment shader.

```
#version 330

layout (location =0) in vec3 position;
layout (location =1) in vec3 inColour;

out vec3 exColour;

void main()
{
    gl_Position = vec4(position, 1.0);
      exColour = inColour;
}
```

And now our fragment shader receives as an input the colour processed by our vertex shader and uses it to generate the colour.

```
#version 330

in  vec3 exColour;
out vec4 fragColor;

void main()
{
    fragColor = vec4(exColour, 1.0);
}
```

The last important thing to do is to modify our rendering code to use that second array of data:

```
public void render(Window window, Mesh mesh) {
    clear();

    if ( window.isResized() ) {
        glViewport(0, 0, window.getWidth(), window.getHeight());
        window.setResized(false);
    }

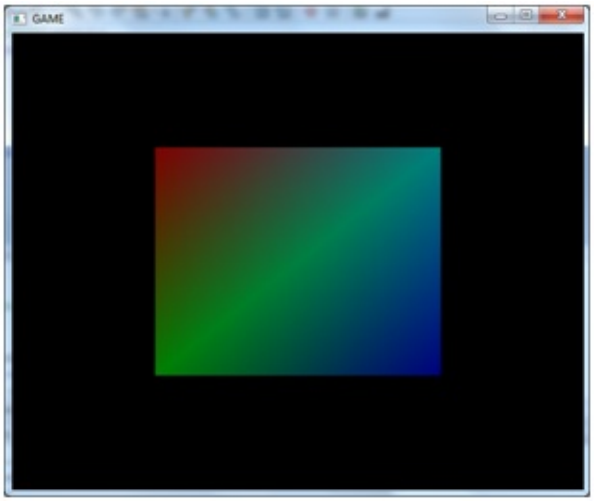
    shaderProgram.bind();

    // Draw the mesh
    glBindVertexArray(mesh.getVaoId());
    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);
    glDrawElements(GL_TRIANGLES, mesh.getVertexCount(), GL_UNSIGNED_INT, 0);
    // ...
```

You can see that we need to enable the VAO attribute at position 1 to be used during rendering. We can now pass an array of colours like this to our `Mesh` class in order to add some colour to our quad.

```
float[] colours = new float[]{
    0.5f, 0.0f, 0.0f,
    0.0f, 0.5f, 0.0f,
    0.0f, 0.0f, 0.5f,
    0.0f, 0.5f, 0.5f,
};
```

And we will get a fancy coloured quad like this.

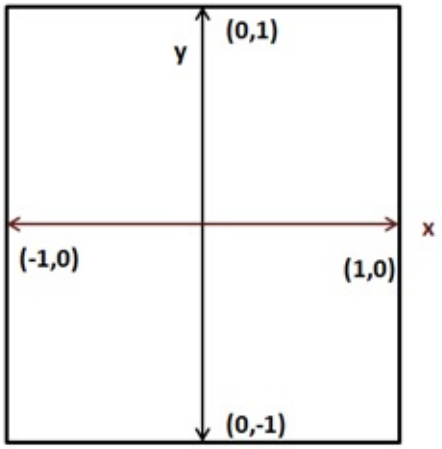


Transformations

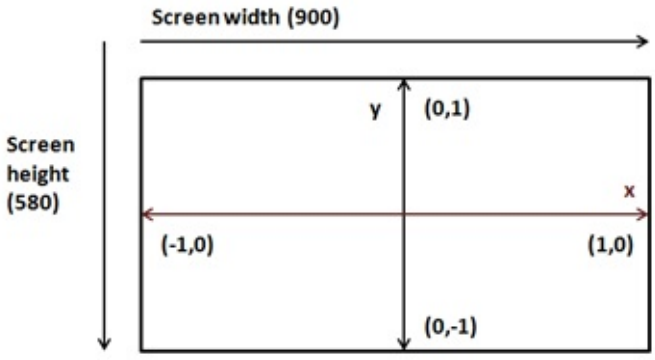
Projecting

Let’s get back to our nice coloured quad we created in the previous chapter. If you look carefully at it, it more resembles a rectangle. You can even change the width of the window from 600 pixels to 900 and the distortion will be more evident. What’s happening here?

If you revisit our vertex shader code we are just passing our coordinates directly. That is, when we say that a vertex has a value for coordinate x of 0.5 we are saying to OpenGL to draw it at x position 0.5 on our screen. The following figure shows the OpenGL coordinates (just for x and y axis).



Those coordinates are mapped, considering our window size, to window coordinates (which have the origin at the top-left corner of the previous figure). So, if our window has a size of 900x480, OpenGL coordinates (1,0) will be mapped to coordinates (900, 0) creating a rectangle instead of a quad.



www.dbooks.org

But, the problem is more serious than that. Modify the z coordinate of our quad from 0.0 to 1.0 and to -1.0. What do you see? The quad is exactly drawn in the same place no matter if it's displaced along the z axis. Why is this happening? Objects that are further away should be drawn smaller than objects that are closer. But we are drawing them with the same x and y coordinates.

But, wait. Should this not be handled by the z coordinate? The answer is yes and no. The z coordinate tells OpenGL that an object is closer or farther away, but OpenGL does not know anything about the size of your object. You could have two objects of different sizes, one closer and smaller and one bigger and further that could be projected correctly onto the screen with the same size (those would have same x and y coordinates but different z). OpenGL just uses the coordinates we are passing, so we must take care of this. We need to correctly project our coordinates.

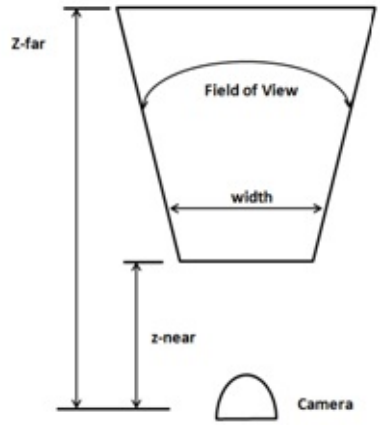
Now that we have diagnosed the problem, how do we do this? The answer is using a projection matrix or frustum. The projection matrix will take care of the aspect ratio (the relation between size and height) of our drawing area so objects won't be distorted. It also will handle the distance so objects far away from us will be drawn smaller. The projection matrix will also consider our field of view and how far the maximum distance is that should be displayed.

For those not familiar with matrices, a matrix is a bi-dimensional array of numbers arranged in columns and rows. Each number inside a matrix is called an element. A matrix order is the number of rows and columns. For instance, here you can see a 2x2 matrix (2 rows and 2 columns).

$$\begin{bmatrix} 1 & 2.3 \\ 0 & -1 \end{bmatrix}$$

Matrices have a number of basic operations that can be applied to them (such as addition, multiplication, etc.) that you can consult in any maths book. The main characteristics of matrices, related to 3D graphics, is that they are very useful to transform points in the space.

You can think about the projection matrix as a camera, which has a field of view and a minimum and maximum distance. The vision area of that camera will be a truncated pyramid. The following picture shows a top view of that area.



A projection matrix will correctly map 3D coordinates so they can be correctly represented on a 2D screen. The mathematical representation of that matrix is as follows (don't be scared).

$$\begin{bmatrix} (1/\tan(fov/2))/a & 0 & 0 & 0 \\ 0 & 1/\tan(fov/2) & 0 & 0 \\ 0 & 0 & -zp/zm & -(2*Zfar*Znear)/zm \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

a = Aspect Ratio
fov = Field Of View
zm = Zfar - Znear
zp = Zfar + Znear

Where aspect ratio is the relation between our screen width and our screen height ($a = width/height$). In order to obtain the projected coordinates of a given point we just need to multiply the projection matrix by the original coordinates. The result will be another vector that will contain the projected version.

So we need to handle a set of mathematical entities such as vectors, matrices and include the operations that can be done on them. We could choose to write all that code by our own from scratch or use an already existing library. We will choose the easy path and use a specific library for dealing with math operations in LWJGL which is called JOML (Java OpenGL Math Library). In order to use that library we just need to add another dependency to our `pom.xml` file.

```
        <dependency>
            <groupId>org.joml</groupId>
            <artifactId>joml</artifactId>
            <version>${joml.version}</version>
        </dependency>
```

And define the version of the library to use.

www.dbooks.org

```
    <properties>
        [...]
        <joml.version>1.9.6</joml.version>
        [...]
    </properties>
```

Now that everything has been set up let's define our projection matrix. We will create an instance of the class `Matrix4f` (provided by the JOML library) in our `Renderer` class. The `Matrix4f` class provides a method to set up a projection matrix named `perspective`. This method needs the following parameters:

- Field of View: The Field of View angle in radians. We will define a constant that holds that value
- Aspect Ratio.
- Distance to the near plane (z-near)
- Distance to the far plane (z-far).

We will instantiate that matrix in our `init` method so we need to pass a reference to our `Window` instance to get its size (you can see it in the source code). The new constants and variables are:

```
    /**
     * Field of View in Radians
     */
    private static final float FOV = (float) Math.toRadians(60.0f);

    private static final float Z_NEAR = 0.01f;

    private static final float Z_FAR = 1000.f;

    private Matrix4f projectionMatrix;
```

The projection matrix is created as follows:

```
float aspectRatio = (float) window.getWidth() / window.getHeight();
projectionMatrix = new Matrix4f().perspective(FOV, aspectRatio,
    Z_NEAR, Z_FAR);
```

At this moment we will ignore that the aspect ratio can change (by resizing our window). This could be checked in the `render` method to change our projection matrix accordingly.

Now that we have our matrix, how do we use it? We need to use it in our shader, and it should be applied to all the vertices. At first, you could think of bundling it in the vertex input (like the coordinates and the colours). In this case we would be wasting lots of space since

the projection matrix should not change even between several render calls. You may also think of multiplying the vertices by the matrix in the java code. But then, our VBOs would be useless and we will not be using the process power available in the graphics card.

The answer is to use “uniforms”. Uniforms are global GLSL variables that shaders can use and that we will employ to communicate with them.

So we need to modify our vertex shader code and declare a new uniform called projectionMatrix and use it to calculate the projected position.

```
#version 330

layout (location=0) in vec3 position;
layout (location=1) in vec3 inColour;

out vec3 exColour;

uniform mat4 projectionMatrix;

void main()
{
    gl_Position = projectionMatrix * vec4(position, 1.0);
    exColour = inColour;
}
```

As you can see we define our `projectionMatrix` as a 4x4 matrix and the position is obtained by multiplying it by our original coordinates. Now we need to pass the values of the projection matrix to our shader. First, we need to get a reference to the place where the uniform will hold its values.

This is done with the method `glGetUniformLocation` which receives two parameters:

- The shader program identifier.
- The name of the uniform (it should match the once defined in the shader code).

This method returns an identifier holding the uniform location. Since we may have more than one uniform, we will store those locations in a Map indexed by the location's name (We will need that location number later). So in the `ShaderProgram` class we create a new variable that holds those identifiers:

```
private final Map<String, Integer> uniforms;
```

This variable will be initialized in our constructor:

```
uniforms = new HashMap<>();
```


www.dbooks.org

And finally we create a method to set up new uniforms and store the obtained location.

```
public void createUniform(String uniformName) throws Exception {
    int uniformLocation = glGetUniformLocation(programId,
        uniformName);
    if (uniformLocation < 0) {
        throw new Exception("Could not find uniform:" +
            uniformName);
    }
    uniforms.put(uniformName, uniformLocation);
}
```

Now, in our `Renderer` class we can invoke the `createUniform` method once the shader program has been compiled (in this case, we will do it once the projection matrix has been instantiated).

```
shaderProgram.createUniform("projectionMatrix");
```

At this moment, we already have a holder ready to be set up with data to be used as our projection matrix. Since the projection matrix won't change between rendering calls we may set up the values right after the creation of the uniform. But we will do it in our render method. You will see later that we may reuse that uniform to do additional operations that need to be done in each render call.

We will create another method in our `ShaderProgram` class to setup the data, named `setUniform`. Basically we transform our matrix into a 4x4 `FloatBuffer` by using the utility methods provided by the JOML library and send them to the location we stored in our locations map.

```
public void setUniform(String uniformName, Matrix4f value) {
    // Dump the matrix into a float buffer
    try (MemoryStack stack = MemoryStack.stackPush()) {
        FloatBuffer fb = stack.mallocFloat(16);
        value.get(fb);
        glUniformMatrix4fv(uniforms.get(uniformName), false, fb);
    }
}
```

As you can see we are creating buffers in a different way here. We are using auto-managed buffers, and allocating them on the stack. This is due to the fact that the size of this buffer is small and that it will not be used beyond this method. Thus, we use the `MemoryStack` class.

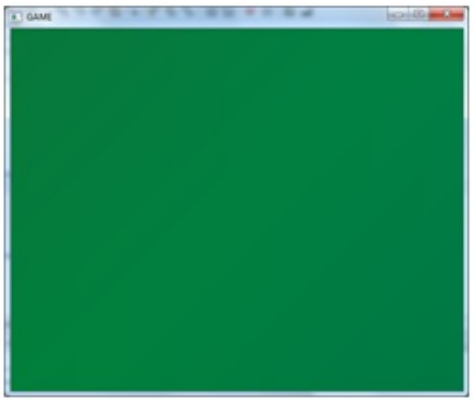
Now we can use that method in the `Renderer` class in the `render` method, after the shader program has been bound:

```
shaderProgram.setUniform("projectionMatrix", projectionMatrix);
```

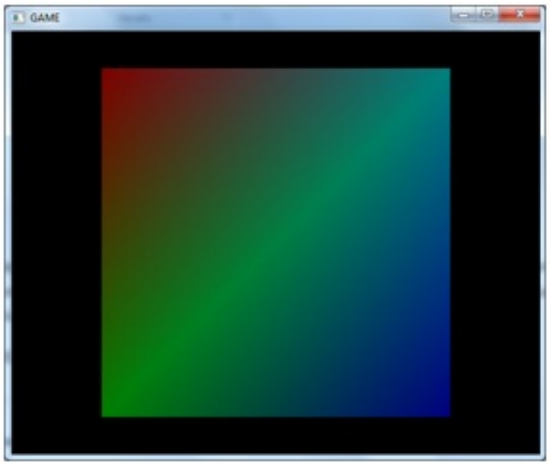
We are almost done. We can now show the quad correctly rendered. So you can now launch your program and will obtain a.... black background without any coloured quad. What's happening? Did we break something? Well, actually no. Remember that we are now simulating the effect of a camera looking at our scene. And we provided two distances, one to the farthest plane (equal to 1000f) and one to the closest plane (equal to 0.01f). Our coordinates are:

```
float[] positions = new float[]{
    -0.5f,  0.5f, 0.0f,
    -0.5f, -0.5f, 0.0f,
     0.5f, -0.5f, 0.0f,
     0.5f,  0.5f, 0.0f,
};
```

That is, our z coordinates are outside the visible zone. Let's assign them a value of `-0.05f`. Now you will see a giant green square like this:



What is happening now is that we are drawing the quad too close to our camera. We are actually zooming into it. If we assign now a value of `-1.05f` to the z coordinate we can now see our coloured quad.



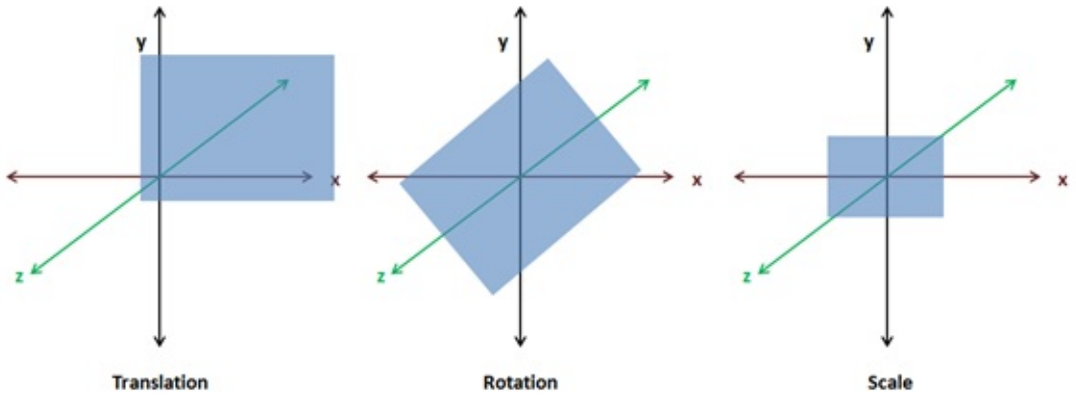
If we continue pushing the quad backwards we will see it becoming smaller. Notice also that our quad does not resemble a rectangle anymore.

Applying Transformations

Let’s recall what we’ve done so far. We have learned how to pass data in an efficient format to our graphics card, and how to project that data and assign them colours using vertex and fragments shaders. Now we should start drawing more complex models in our 3D space. But in order to do that we must be able to load an arbitrary model and represent it in our 3D space at a specific position, with the appropriate size and the required rotation.

So right now, in order to do that representation we need to provide some basic operations to act upon any model:

- Translation: Move an object by some amount on any of the three axes.
- Rotation: Rotate an object by some amount of degrees around any of the three axes.
- Scale: Adjust the size of an object.



The operations described above are known as transformations. And you probable may be guessing that the way we are going to achieve that is by multiplying our coordinates by a set of matrices (one for translation, one for rotation and one for scaling). Those three matrices will be combined into a single matrix called world matrix and passed as a uniform to our vertex shader.

The reason why it is called world matrix is because we are transforming from model coordinates to world coordinates. When you will learn about loading 3D models you will see that those models are defined in their own coordinate systems. They don’t know the size of your 3D space and they need to be placed in it. So when we multiply our coordinates by our matrix what we are doing is transforming from a coordinate system (the model one) to another coordinate system (the one for our 3D world).

That world matrix will be calculated like this (the order is important since multiplication using matrices is not commutative):

$$WorldMatrix\,[TranslationMatrix]\,[RotationMatrix]\,[ScaleMatrix]$$

If we include our projection matrix in the transformation matrix it would be like this:

$$Transf = [ProjMatrix]\,[TranslationMatrix]\,[RotationMatrix]\,[ScaleMatrix] = [ProjMatr$$

The translation matrix is defined like this:

$$\begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation Matrix Parameters:

- dx: Displacement along the x axis.
- dy: Displacement along the y axis.
- dz: Displacement along the z axis.

The scale matrix is defined like this:

$$\begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale Matrix Parameters:

- sx: Scaling along the x axis.
- sy: Scaling along the y axis.
- sz: Scaling along the z axis.

The rotation matrix is much more complex. But keep in mind that it can be constructed by the multiplication of 3 rotation matrices for a single axis, each.

Now, in order to apply those concepts we need to refactor our code a little bit. In our game we will be loading a set of models which can be used to render many objects at different positions according to our game logic (imagine a FPS game which loads three models for different enemies. There are only three models but using these models we can draw as many enemies as we want). Do we need to create a VAO and the set of VBOs for each of those objects? The answer is no. We only need to load it once per model. What we need to do is to draw it independently according to its position, size and rotation. We need to transform those models when we are rendering them.

So we will create a new class named `GameItem` that will hold a reference to a model, to a `Mesh` instance. A `GameItem` instance will have variables for storing its position, its rotation state and its scale. This is the definition of that class.

www.dbooks.org

```java
package org.lwjglb.engine;

import org.joml.Vector3f;
import org.lwjglb.engine.graph.Mesh;

public class GameItem {

    private final Mesh mesh;

    private final Vector3f position;

    private float scale;

    private final Vector3f rotation;

    public GameItem(Mesh mesh) {
        this.mesh = mesh;
        position = new Vector3f(0, 0, 0);
        scale = 1;
        rotation = new Vector3f(0, 0, 0);
    }

    public Vector3f getPosition() {
        return position;
    }

    public void setPosition(float x, float y, float z) {
        this.position.x = x;
        this.position.y = y;
        this.position.z = z;
    }

    public float getScale() {
        return scale;
    }

    public void setScale(float scale) {
        this.scale = scale;
    }

    public Vector3f getRotation() {
        return rotation;
    }

    public void setRotation(float x, float y, float z) {
        this.rotation.x = x;
        this.rotation.y = y;
        this.rotation.z = z;
    }

    public Mesh getMesh() {
        return mesh;
```

```
    }
}
```

We will create another class which will deal with transformations named `Transformation`.

```
package org.lwjglb.engine.graph;

import org.joml.Matrix4f;
import org.joml.Vector3f;

public class Transformation {

    private final Matrix4f projectionMatrix;

    private final Matrix4f worldMatrix;

    public Transformation() {
        worldMatrix = new Matrix4f();
        projectionMatrix = new Matrix4f();
    }

    public final Matrix4f getProjectionMatrix(float fov, float width, float height, fl
oat zNear, float zFar) {
        float aspectRatio = width / height;
        projectionMatrix.identity();
        projectionMatrix.perspective(fov, aspectRatio, zNear, zFar);
        return projectionMatrix;
    }

    public Matrix4f getWorldMatrix(Vector3f offset, Vector3f rotation, float scale) {
        worldMatrix.identity().translate(offset).
                rotateX((float)Math.toRadians(rotation.x)).
                rotateY((float)Math.toRadians(rotation.y)).
                rotateZ((float)Math.toRadians(rotation.z)).
                scale(scale);
        return worldMatrix;
    }
}
```

As you can see this class groups the projection and world matrices. Given a set of vectors that model the displacement, rotation and scale it returns the world matrix. The method `getWorldMatrix` returns the matrix that will be used to transform the coordinates for each `GameItem` instance. That class also provides a method that gets the projection matrix based on the Field Of View, the aspect ratio and the near and far distances.

An important thing to notice is that the `mul` method of the `Matrix4f` class modifies the matrix instance which the method is being applied to. So if we directly multiply the projection matrix with the transformation matrix we will modify the projection matrix itself. This is why

www.dbooks.org

we are always initializing each matrix to the identity matrix upon each call.

In the `Renderer` class, in the constructor method, we just instantiate the `Transformation` with no arguments and in the `init` method we just create the uniform.

```
public Renderer() {
    transformation = new Transformation();
}

public void init(Window window) throws Exception {
    // .... Some code before ...
    // Create uniforms for world and projection matrices
    shaderProgram.createUniform("projectionMatrix");
    shaderProgram.createUniform("worldMatrix");

    window.setClearColor(0.0f, 0.0f, 0.0f, 0.0f);
}
```

In the render method of our `Renderer` class we now receive an array of GameItems:

```
public void render(Window window, GameItem[] gameItems) {
    clear();

        if ( window.isResized() ) {
            glViewport(0, 0, window.getWidth(), window.getHeight());
            window.setResized(false);
        }

    shaderProgram.bind();

    // Update projection Matrix
    Matrix4f projectionMatrix = transformation.getProjectionMatrix(FOV, window.getWidt
h(), window.getHeight(), Z_NEAR, Z_FAR);
    shaderProgram.setUniform("projectionMatrix", projectionMatrix);

    // Render each gameItem
    for(GameItem gameItem : gameItems) {
        // Set world matrix for this item
        Matrix4f worldMatrix =
            transformation.getWorldMatrix(
                gameItem.getPosition(),
                gameItem.getRotation(),
                gameItem.getScale());
        shaderProgram.setUniform("worldMatrix", worldMatrix);
        // Render the mes for this game item
        gameItem.getMesh().render();
    }

    shaderProgram.unbind();
}
```

We update the projection matrix once per `render` call. By doing it this way we can deal with window resize operations. Then we iterate over the `GameItem` array and create a transformation matrix according to the position, rotation and scale of each of them. This matrix is pushed to the shader and the Mesh is drawn. The projection matrix is the same for all the items to be rendered. This is the reason why it's a separate variable in our Transformation class.

We moved the rendering code to draw a Mesh to this class:

```
public void render() {
    // Draw the mesh
    glBindVertexArray(getVaoId());
    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);

    glDrawElements(GL_TRIANGLES, getVertexCount(), GL_UNSIGNED_INT, 0);

    // Restore state
    glDisableVertexAttribArray(0);
    glDisableVertexAttribArray(1);
    glBindVertexArray(0);
}
```

Our vertex shader is modified by simply adding a new `worldMatrix` matrix and it uses it with the `projectionMatrix` to calculate the position:

```
#version 330

layout (location=0) in vec3 position;
layout (location=1) in vec3 inColour;

out vec3 exColour;

uniform mat4 worldMatrix;
uniform mat4 projectionMatrix;

void main()
{
    gl_Position = projectionMatrix * worldMatrix * vec4(position, 1.0);
    exColour = inColour;
}
```

As you can see the code is exactly the same. We are using the uniform to correctly project our coordinates taking into consideration our frustum, position, scale and rotation information.

www.dbooks.org

Another important thing to think about is, why don’t we pass the translation, rotation and scale matrices instead of combining them into a world matrix? The reason is that we should try to limit the matrices we use in our shaders. Also keep in mind that the matrix multiplication that we do in our shader is done once per each vertex. The projection matrix does not change between render calls and the world matrix does not change per `GameItem` instance. If we passed the translation, rotation and scale matrices independently we would be doing many more matrix multiplications. Think about a model with tons of vertices. That’s a lot of extra operations.

But you may now think, that if the world matrix does not change per `GameItem` instance, why didn't we do the matrix multiplication in our Java class? We would multiply the projection matrix and the world matrix just once per GameItem and we send it as a single uniform. In this case we would be saving many more operations. The answer is that this is a valid point right now. But when we add more features to our game engine we will need to operate with world coordinates in the shaders anyway, so it’s better to handle those two matrices in an independent way.

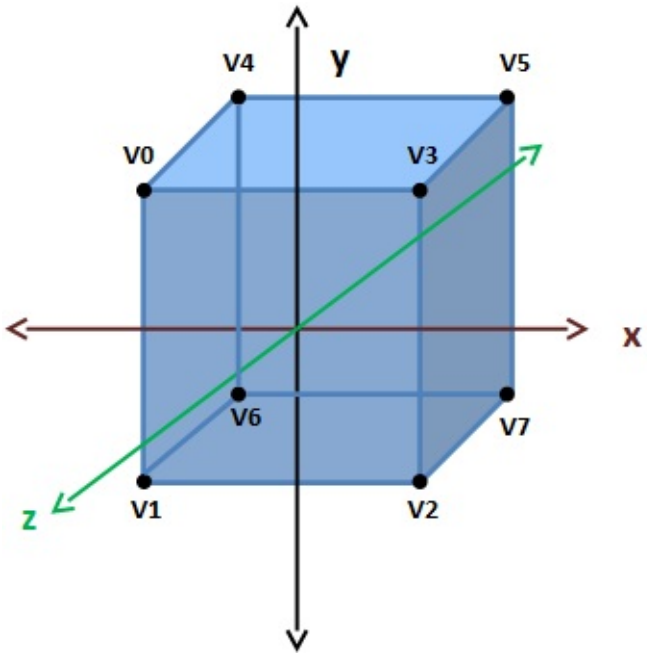
Finally we only need to change the `DummyGame` class to create an instance of `GameItem` with its associated `Mesh` and add some logic to translate, rotate and scale our quad. Since it’s only a test example and does not add too much you can find it in the source code that accompanies this book.

Textures

Create a 3D cube

In this chapter we will learn how to load textures and use them in the rendering process. In order to show all the concepts related to textures we will transform the quad that we have been using in previous chapters into a 3D cube. With the code base we have created, in order to draw a cube we just need to correctly define the coordinates of a cube and it should be drawn correctly.

In order to draw a cube we just need to define eight vertices.



So the associated coordinates array will be like this:

www.dbooks.org

```
float[] positions = new float[] {
    // V0
    -0.5f,  0.5f,  0.5f,
    // V1
    -0.5f, -0.5f,  0.5f,
    // V2
    0.5f, -0.5f,  0.5f,
    // V3
     0.5f,  0.5f,  0.5f,
    // V4
    -0.5f,  0.5f, -0.5f,
    // V5
     0.5f,  0.5f, -0.5f,
    // V6
    -0.5f, -0.5f, -0.5f,
    // V7
     0.5f, -0.5f, -0.5f,
};
```

Of course, since we have 4 more vertices we need to update the array of colours. Just repeat the first four items by now.

```
float[] colours = new float[]{
    0.5f, 0.0f, 0.0f,
    0.0f, 0.5f, 0.0f,
    0.0f, 0.0f, 0.5f,
    0.0f, 0.5f, 0.5f,
    0.5f, 0.0f, 0.0f,
    0.0f, 0.5f, 0.0f,
    0.0f, 0.0f, 0.5f,
    0.0f, 0.5f, 0.5f,
};
```

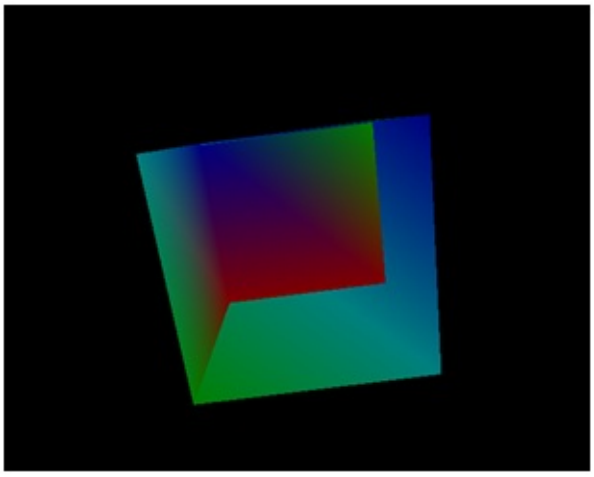
Finally, since a cube is made of six faces we need to draw twelve triangles (two per face), so we need to update the indices array. Remember that triangles must be defined in counter-clock wise order. If you do this by hand, is easy to make mistakes. Allways put the face that you want to define indices for in front of you. Then, idenifie the vertices and draw the triangles in counter-clock wise order.

```
int[] indices = new int[] {
    // Front face
    0, 1, 3, 3, 1, 2,
    // Top Face
    4, 0, 3, 5, 4, 3,
    // Right face
    3, 2, 7, 5, 3, 7,
    // Left face
    6, 1, 0, 6, 0, 4,
    // Bottom face
    2, 1, 6, 2, 6, 7,
    // Back face
    7, 6, 4, 7, 4, 5,
};
```

In order to better view the cube we will change code that rotates the model in the `DummyGame` class to rotate along the three axes.

```
// Update rotation angle
float rotation = gameItem.getRotation().x + 1.5f;
if ( rotation > 360 ) {
    rotation = 0;
}
gameItem.setRotation(rotation, rotation, rotation);
```

And that's all. We are now able to display a spinning 3D cube. You can now compile and run your example and you will obtain something like this.



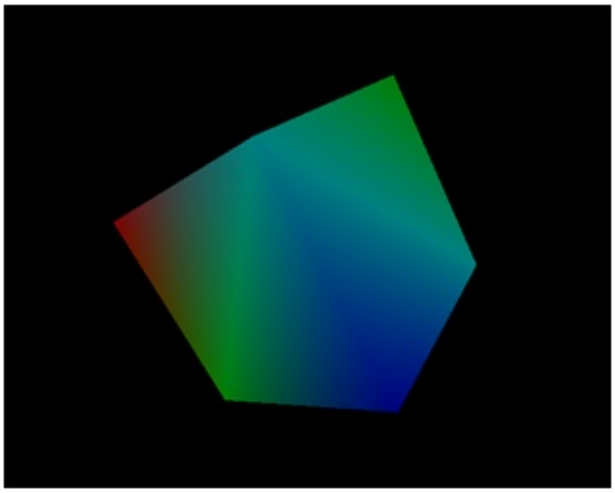
There is something weird with this cube. Some faces are not being painted correctly. What is happening? The reason why the cube has this aspect is that the triangles that compose the cube are being drawn in a sort of random order. The pixels that are far away should be drawn before pixels that are closer. This is not happening right now and in order to do that we must enable depth testing.

www.dbooks.org

This can be done in the `Window` class at the end of the `init` method:

```
glEnable(GL_DEPTH_TEST);
```

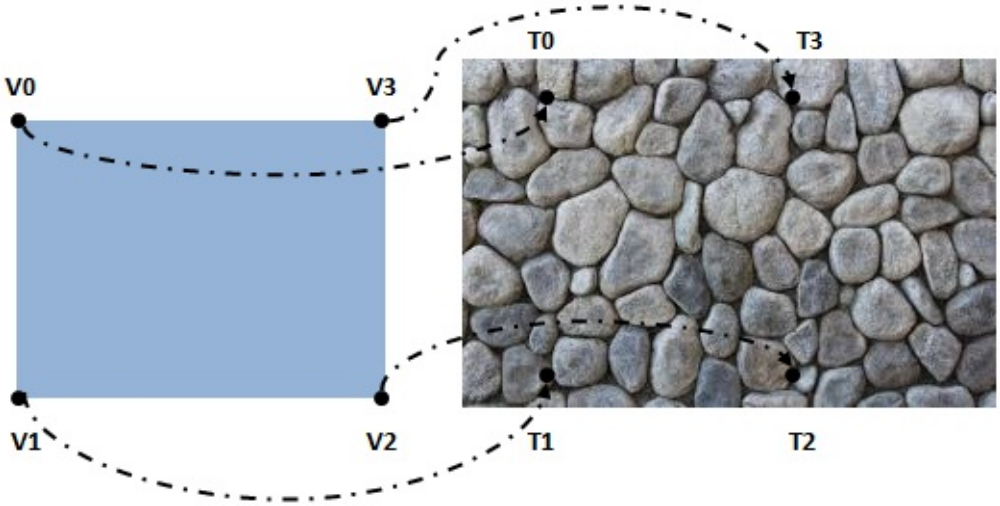
Now our cube is being rendered correctly!



If you see the code for this part of the chapter you may see that we have done a minor reorganization in the `Mesh` class. The identifiers of the VBOs are now stored in a list to easily iterate over them.

Adding texture to the cube

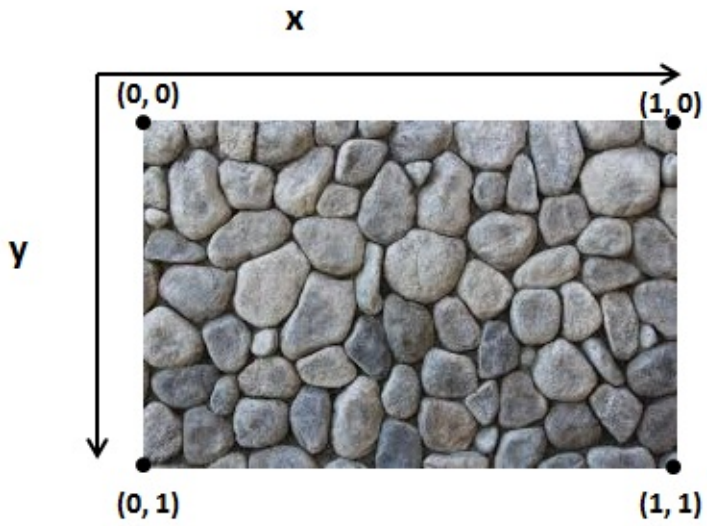
Now we are going to apply a texture to our cube. A texture is an image which is used to draw the colour of the pixels of a certain model. You can think of a texture as a skin that is wrapped around your 3D model. What you do is assign points in the image texture to the vertices in your model. With that information OpenGL is able to calculate the colour to apply to the other pixels based on the texture image.



The texture image does not have to have the same size as the model. It can be larger or

smaller. OpenGL will extrapolate the colour if the pixel to be processed cannot be mapped to a specific point in the texture. You can control how this process is done when a specific texture is created.

So basically what we must do, in order to apply a texture to a model, is assigning texture coordinates to each of our vertices. The texture coordinate system is a bit different from the coordinate system of our model. First of all, we have a 2D texture so our coordinates will only have two components, x and y. Besides that, the origin is setup in the top left corner of the image and the maximum value of the x or y value is 1.



How do we relate texture coordinates with our position coordinates? Easy, in the same way we passed the colour information. We set up a VBO which will have a texture coordinate for each vertex position.

So let's start modifying the code base to use textures in our 3D cube. The first step is to load the image that will be used as a texture. For this task, in previous versions of LWJGL, the Slick2D library was commonly used. At the moment of this writing it seems that this library is not compatible with LWJGL 3 so we will need to follow a more verbose approach. We will use a library called `pngdecoder`, thus, we need to declare that dependency in our `pom.xml` file.

```
<dependency>
    <groupId>org.l33tlabs.twl</groupId>
    <artifactId>pngdecoder</artifactId>
    <version>${pngdecoder.version}</version>
</dependency>
```

And define the version of the library to use.

www.dbooks.org

```
<properties>
    [...]
    <pngdecoder.version>1.0</pngdecoder.version>
    [...]
</properties>
```

One thing that you may see in some web pages is that the first thing we must do is enable the textures in our OpenGL context by calling `glEnable(GL_TEXTURE_2D)`. This is true if you are using the fixed-function pipepline. Since we are using GLSL shaders it is not required anymore.

Now we will create a new `Texture` class that will perform all the necessary steps to load a texture. Our texture image will be located in the resources folder and can be accessed as a CLASSPATH resource and passed as an input stream to the `PNGDecoder` class.

```
PNGDecoder decoder = new PNGDecoder(
     Texture.class.getResourceAsStream(fileName));
```

Then we need to decode the PNG image and store its content into a buffer by using the `decode` method of the `PNGDecoder` class. The PNG image will be decoded in RGBA format (RGB for Red, Green, Blue and A for Alpha or transparency) which uses four bytes per pixel.

The `decode` method requires three parameters:

- `buffer`: The `ByteBuffer` that will hold the decoded image (since each pixel uses four bytes its size will be 4 *width* height).
- `stride`: Specifies the distance in bytes from the start of a line to the start of the next line. In this case it will be the number of bytes per line.
- `format`: The target format into which the image should be decoded (RGBA).

```
ByteBuffer buf = ByteBuffer.allocateDirect(
    4 * decoder.getWidth() * decoder.getHeight());
decoder.decode(buf, decoder.getWidth() * 4, Format.RGBA);
buf.flip();
```

One important thing to remember is that OpenGL, for historical reasons, requires that texture images have a size (number of texels in each dimension) of a power of two (2, 4, 8, 16, ...). Some drivers remove that constraint but it's better to stick to it to avoid problems.

The next step is to upload the texture into the graphics card memory. First of all we need to create a new texture identifier. Each operation related to that texture will use that identifier so we need to bind it.

```
// Create a new OpenGL texture
int textureId = glGenTextures();
// Bind the texture
glBindTexture(GL_TEXTURE_2D, textureId);
```

Then we need to tell OpenGL how to unpack our RGBA bytes. Since each component is one byte in size we need to add the following line:

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

And finally we can upload our texture data:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, decoder.getWidth(),
    decoder.getHeight(), 0, GL_RGBA, GL_UNSIGNED_BYTE, buf);
```

The `glTextImage2D` method has the following parameters:

- `target` : Specifies the target texture (its type). In this case: `GL_TEXTURE_2D` .
- `level` : Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image. More on this later.
- `internal format` : Specifies the number of colour components in the texture.
- `width` : Specifies the width of the texture image.
- `height` : Specifies the height of the texture image.
- `border` : This value must be zero.
- `format` : Specifies the format of the pixel data: RGBA in this case.
- `type` : Specifies the data type of the pixel data. We are using unsigned bytes for this.
- `data` : The buffer that stores our data.

In some code snippets that you may find yow will probably see that filtering parameters are set up before calling the `glTextImage2D` method. Filtering refers to how the image will be drawn when scaling and how pixels will be interpolated.

If those parameters are not set the texture will not be displayed. So before the `glTextImage2D` method you could see something like this:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

This parameter basically says that when a pixel is drawn with no direct one to one association to a texture coordinate it will pick the nearest texture coordinate point.

www.dbooks.org

By this moment we will not set up those parameters. Instead we will generate a mipmap. A mipmap is a decreasing resolution set of images generated from a high detailed texture. Those lower resolution images will be used automatically when our object is scaled.

In order to generate mipmaps we just need to set the following line (in this case after the glTextImage2D method:

```
glGenerateMipmap(GL_TEXTURE_2D);
```

And that's all, we have successfully loaded our texture. Now we need to use it. As we said before we need to pass texture coordinates as another VBO. So we will modify our Mesh class to accept an array of floats, that contains texture coordinates, instead of the colour (we could have colours and texture but in order to simplify it we will strip colours off). Our constructor will be like this:

```
public Mesh(float[] positions, float[] textCoords, int[] indices,
    Texture texture)
```

The texture coordinates VBO is created in the same way as the colour one. The only difference is that it has two elements instead of three:

```
vboId = glGenBuffers();
vboIdList.add(vboId);
textCoordsBuffer = MemoryUtil.memAllocFloat(textCoords.length);
textCoordsBuffer.put(textCoords).flip();
glBindBuffer(GL_ARRAY_BUFFER, vboId);
glBufferData(GL_ARRAY_BUFFER, textCoordsBuffer, GL_STATIC_DRAW);
glVertexAttribPointer(1, 2, GL_FLOAT, false, 0, 0);
```

Now we need to use those textures in our shader. In the vertex shader we have changed the second uniform parameter because now it's a `vec2` (we also changed the uniform name, so remember to change it in the `Renderer` class). The vertex shader, as in the colour case, just passes the texture coordinates to be used by the fragment shader.

```
#version 330

layout (location=0) in vec3 position;
layout (location=1) in vec2 texCoord;

out vec2 outTexCoord;

uniform mat4 worldMatrix;
uniform mat4 projectionMatrix;

void main()
{
    gl_Position = projectionMatrix * worldMatrix * vec4(position, 1.0);
    outTexCoord = texCoord;
}
```

In the fragment shader we must use those texture coordinates in order to set the pixel colours:

```
#version 330

in  vec2 outTexCoord;
out vec4 fragColor;

uniform sampler2D texture_sampler;

void main()
{
    fragColor = texture(texture_sampler, outTexCoord);
}
```

Before analyzing the code let's clarify some concepts. A graphics card has several spaces or slots to store textures. Each of these spaces is called a texture unit. When we are working with textures we must set the texture unit that we want to work with. As you can see we have a new uniform named `texture_sampler`. That uniform has a `sampler2D` type and will hold the value of the texture unit that we want to work with.

In the main function we use the texture `lookup` function named `texture`. This function takes two arguments: a sampler and a texture coordinate and will return the correct colour. The sampler uniform allow us to do multi-texturing. We will not cover that topic right now but we will try to prepare the code to add it easily later on.

Thus, in our `ShaderProgram` class we will create a new method that allows us to set an integer value for a uniform:

www.dbooks.org

```
public void setUniform(String uniformName, int value) {
    glUniform1i(uniforms.get(uniformName), value);
}
```

In the `init` method of the `Renderer` class we will create a new uniform:

```
shaderProgram.createUniform("texture_sampler");
```

Also, in the `render` method of our `Renderer` class we will set the uniform value to 0. (We are not using several textures right now so we are just using unit 0).

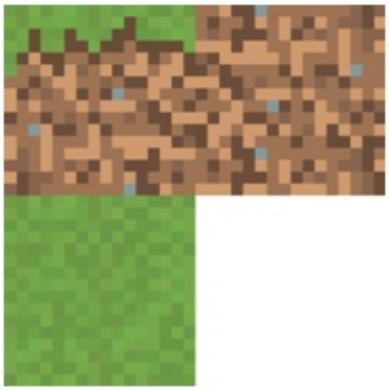
```
shaderProgram.setUniform("texture_sampler", 0);
```

Finally we just need to change the render method of the `Mesh` class to use the texture. At the beginning of that method we put the following lines:

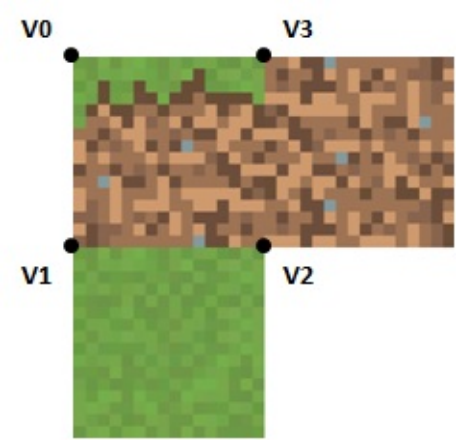
```
// Activate first texture unit
glActiveTexture(GL_TEXTURE0);
// Bind the texture
glBindTexture(GL_TEXTURE_2D, texture.getId());
```

We basically are binding the texture identified by `texture.getId()` to the texture unit 0.

Right now, we have just modified our code base to support textures. Now we need to setup texture coordinates for our 3D cube. Our texture image file will be something like this:

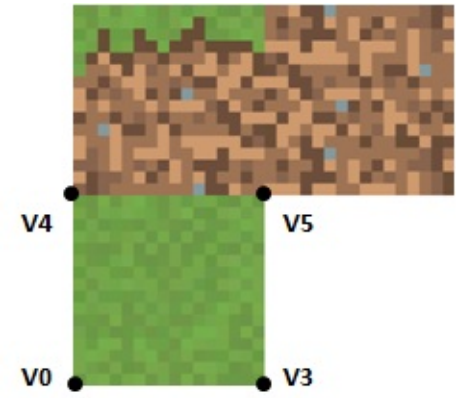


In our 3D model we have eight vertices. Let’s see how this can be done. Let’s first define the front face texture coordinates for each vertex.



Vertex	Texture Coordinate
V0	(0.0, 0.0)
V1	(0.0, 0.5)
V2	(0.5, 0.5)
V3	(0.5, 0.0)

Now, let's define the texture mapping of the top face.

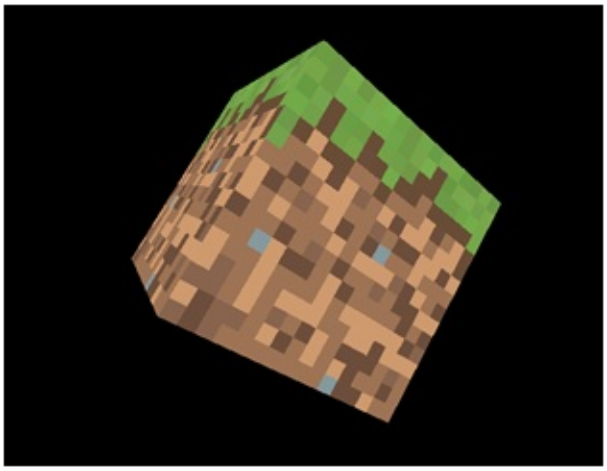


Vertex	Texture Coordinate
V4	(0.0, 0.5)
V5	(0.5, 0.5)
V0	(0.0, 1.0)
V3	(0.5, 1.0)

As you can see we have a problem, we need to setup different texture coordinates for the same vertices (V0 and V3). How can we solve this? The only way to solve it is to repeat some vertices and associate different texture coordinates. For the top face we need to repeat the four vertices and assign them the correct texture coordinates.

www.dbooks.org

Since the front, back and lateral faces use the same texture we will not need to repeat all of these vertices. You have the complete definition in the source code, but we needed to move from 8 points to 20. The final result is like this.



In the next chapters we will learn how to load models generated by 3D modeling tools so we won’t need to define by hand the positions and texture coordinates (which by the way, would be impractical for more complex models).

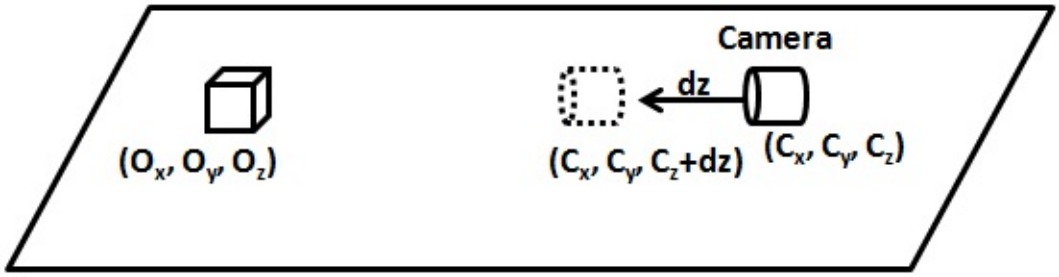
Camera

In this chapter we will learn how to move inside a rendered 3D scene, this capability is like having a camera that can travel inside the 3D world and in fact is the term used to refer to it.

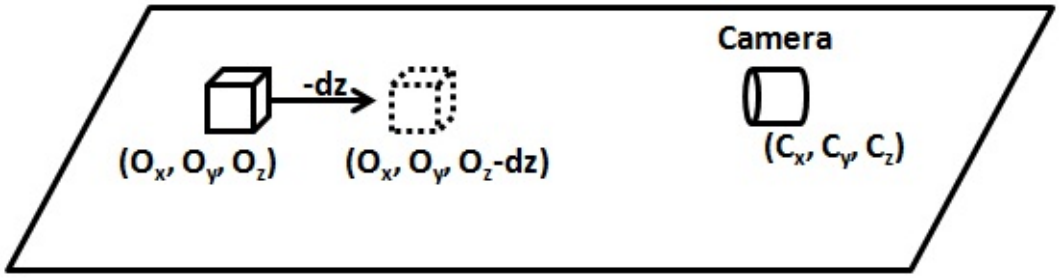
But if you try to search for specific camera functions in OpenGL you will discover that there is no camera concept, or in other words the camera is always fixed, centered in the (0, 0, 0) position at the center of the screen.

So what we will do is a simulation that gives us the impression that we have a camera capable of moving inside the 3D scene. How do we achieve this? Well, if we cannot move the camera then we must move all the objects contained in our 3D space at once. In other words, if we cannot move a camera we will move the whole world.

So, suppose that we would like to move the camera position along the z axis from a starting position (Cx, Cy, Cz) to a position (Cx, Cy, Cz+dz) to get closer to the object which is placed at the coordinates (Ox, Oy, Oz).

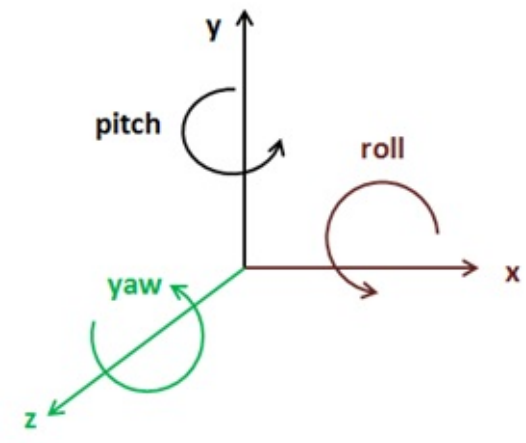


What we will actually do is move the object (all the objects in our 3D space indeed) in the opposite direction that the camera should move. Think about it like the objects being placed in a treadmill.



A camera can be displaced along the three axis (x, y and z) and also can rotate along them (roll, pitch and yaw).

www.dbooks.org



So basically what we must do is to be able to move and rotate all of the objects of our 3D world. How are we going to do this? The answer is to apply another transformation that will translate all of the vertices of all of the objects in the opposite direction of the movement of the camera and that will rotate them according to the camera rotation. This will be done of course with another matrix, the so called view matrix. This matrix will first perform the translation and then the rotation along the axis.

Let's see how we can construct that matrix. If you remember from the transformations chapter our transformation equation was like this:

$$Transf = [ProjMatrix] \cdot [TranslationMatrix] \cdot [RotationMatrix] \cdot [ScaleMatrix] = [ProjM$$

The view matrix should be applied before multiplying by the projection matrix, so our equation should be now like this:

$$Transf = [ProjMatrix] \cdot [ViewMatrix] \cdot [TranslationMatrix] \cdot [RotationMatrix] \cdot [ScaleM$$

Now we have three matrices, let's think a little bit about the life cycles of those matrices. The projection matrix should not change very much while our game is running, in the worst case it may change once per render call. The view matrix may change once per render call if the camera moves. The world matrix changes once per `GameItem` instance, so it will change several times per render call.

So, how many matrices should we push to or vertex shader ? You may see some code that uses three uniforms for each of those matrices, but in principle the most efficient approach would be to combine the projection and the view matrices, let's call it `pv` matrix, and push the `world` and the `pv` matrices to our shader. With this approach we would have the possibility to work with world coordinates and would be avoiding some extra multiplications.

Actually, the most convenient approach is to combine the view and the world matrix. Why this? Because remember that the whole camera concept is a trick, what we are doing is pushing the whole world to simulate world displacement and to show only a small portion of the 3D world. So if we work directly with world coordinates we may be working with world coordinates that are far away from the origin and we may incur in some precision problems.

If we work in what's called the camera space we will be working with points that, although are far away from the world origin, are closer to the camera. The matrix that results of the combination of the view and the world matrix is often called as the model view matrix.

So let's start modifying our code to support a camera. First of all we will create a new class called `Camera` which will hold the position and rotation state of our camera. This class will provide methods to set the new position or rotation state (`setPosition` or `setRotation`) or to update those values with an offset upon the current state (`movePosition` and `moveRotation`)

```
package org.lwjglb.engine.graph;

import org.joml.Vector3f;

public class Camera {

    private final Vector3f position;

    private final Vector3f rotation;

    public Camera() {
        position = new Vector3f(0, 0, 0);
        rotation = new Vector3f(0, 0, 0);
    }

    public Camera(Vector3f position, Vector3f rotation) {
        this.position = position;
        this.rotation = rotation;
    }

    public Vector3f getPosition() {
        return position;
    }

    public void setPosition(float x, float y, float z) {
        position.x = x;
        position.y = y;
        position.z = z;
    }

    public void movePosition(float offsetX, float offsetY, float offsetZ) {
        if ( offsetZ != 0 ) {
            position.x += (float)Math.sin(Math.toRadians(rotation.y)) * -1.0f * offset
Z;
            position.z += (float)Math.cos(Math.toRadians(rotation.y)) * offsetZ;
        }
        if ( offsetX != 0) {
            position.x += (float)Math.sin(Math.toRadians(rotation.y - 90)) * -1.0f * o
ffsetX;
            position.z += (float)Math.cos(Math.toRadians(rotation.y - 90)) * offsetX;
```


www.dbooks.org

```
        }
        position.y += offsetY;
    }

    public Vector3f getRotation() {
        return rotation;
    }

    public void setRotation(float x, float y, float z) {
        rotation.x = x;
        rotation.y = y;
        rotation.z = z;
    }

    public void moveRotation(float offsetX, float offsetY, float offsetZ) {
        rotation.x += offsetX;
        rotation.y += offsetY;
        rotation.z += offsetZ;
    }
}
```

Next in the `Transformation` class we will hold a new matrix to hold the values of the view matrix.

```
private final Matrix4f viewMatrix;
```

We will also provide a method to update its value. Like the projection matrix this matrix will be the same for all the objects to be rendered in a render cycle.

```
public Matrix4f getViewMatrix(Camera camera) {
    Vector3f cameraPos = camera.getPosition();
    Vector3f rotation = camera.getRotation();

    viewMatrix.identity();
    // First do the rotation so camera rotates over its position
    viewMatrix.rotate((float)Math.toRadians(rotation.x), new Vector3f(1, 0, 0))
        .rotate((float)Math.toRadians(rotation.y), new Vector3f(0, 1, 0));
    // Then do the translation
    viewMatrix.translate(-cameraPos.x, -cameraPos.y, -cameraPos.z);
    return viewMatrix;
}
```

As you can see we first need to do the rotation and then the translation. If we do the opposite we would not be rotating along the camera position but along the coordinates origin. Please also note that in the `movePosition` method of the `Camera` class we just not simply increase the camera position by and offset. We also take into consideration the

rotation along the y axis, the yaw, in order to calculate the final position. If we would just increase the camera position by the offset the camera will not move in the direction its facing.

Besides what is mentioned above, we do not have here a full free fly camera (for instance, if we rotate along the x axis the camera does not move up or down in the space when we move it forward). This will be done in later chapters since is a little bit more complex.

Finally we will remove the previous method `getWorldMatrix` and add a new one called `getModelViewMatrix`.

```
public Matrix4f getModelViewMatrix(GameItem gameItem, Matrix4f viewMatrix) {
    Vector3f rotation = gameItem.getRotation();
    modelViewMatrix.identity().translate(gameItem.getPosition()).
        rotateX((float)Math.toRadians(-rotation.x)).
        rotateY((float)Math.toRadians(-rotation.y)).
        rotateZ((float)Math.toRadians(-rotation.z)).
        scale(gameItem.getScale());
    Matrix4f viewCurr = new Matrix4f(viewMatrix);
    return viewCurr.mul(modelViewMatrix);
}
```

The `getModelViewMatrix` method will be called per each `GameItem` instance so we must work over a copy of the view matrix so transformations do not get accumulated in each call (Remember that `Matrix4f` class is not immutable).

In the `render` method of the `Renderer` class we just need to update the view matrix according to the camera values, just after the projection matrix is also updated.

```
// Update projection Matrix
Matrix4f projectionMatrix = transformation.getProjectionMatrix(FOV, window.getWidth(),
 window.getHeight(), Z_NEAR, Z_FAR);
shaderProgram.setUniform("projectionMatrix", projectionMatrix);

// Update view Matrix
Matrix4f viewMatrix = transformation.getViewMatrix(camera);

shaderProgram.setUniform("texture_sampler", 0);
// Render each gameItem
for(GameItem gameItem : gameItems) {
    // Set model view matrix for this item
    Matrix4f modelViewMatrix = transformation.getModelViewMatrix(gameItem, viewMatrix)
;
    shaderProgram.setUniform("modelViewMatrix", modelViewMatrix);
    // Render the mes for this game item
    gameItem.getMesh().render();
}
```


www.dbooks.org

And that's all, our base code supports the concept of a camera. Now we need to use it. We can change the way we handle the input and update the camera. We will set the following controls:

- Keys "A" and "D" to move the camera to the left and right (x axis) respectively.
- Keys "W" and "S" to move the camera forward and backwards (z axis) respectively.
- Keys "Z" and "X" to move the camera up and down (y axis) respectively.

We will use the mouse position to rotate the camera along the x and y axis when the right button of the mouse is pressed.
As you can see we will be using the mouse for the first time. We will create a new class named `MouseInput` that will encapsulate mouse access. Here's the code for that class.

```
package org.lwjglb.engine;

import org.joml.Vector2d;
import org.joml.Vector2f;
import static org.lwjgl.glfw.GLFW.*;

public class MouseInput {

    private final Vector2d previousPos;

    private final Vector2d currentPos;

    private final Vector2f displVec;

    private boolean inWindow = false;

    private boolean leftButtonPressed = false;

    private boolean rightButtonPressed = false;

    public MouseInput() {
        previousPos = new Vector2d(-1, -1);
        currentPos = new Vector2d(0, 0);
        displVec = new Vector2f();
    }

    public void init(Window window) {
        glfwSetCursorPosCallback(window.getWindowHandle(), (windowHandle, xpos, ypos)
-> {
            currentPos.x = xpos;
            currentPos.y = ypos;
        });
        glfwSetCursorEnterCallback(window.getWindowHandle(), (windowHandle, entered) -
> {
            inWindow = entered;
        });
        glfwSetMouseButtonCallback(window.getWindowHandle(), (windowHandle, button, ac
```

```
tion, mode) -> {
            leftButtonPressed = button == GLFW_MOUSE_BUTTON_1 && action == GLFW_PRESS;
            rightButtonPressed = button == GLFW_MOUSE_BUTTON_2 && action == GLFW_PRESS
;
        });
    }

    public Vector2f getDisplVec() {
        return displVec;
    }

    public void input(Window window) {
        displVec.x = 0;
        displVec.y = 0;
        if (previousPos.x > 0 && previousPos.y > 0 && inWindow) {
            double deltax = currentPos.x - previousPos.x;
            double deltay = currentPos.y - previousPos.y;
            boolean rotateX = deltax != 0;
            boolean rotateY = deltay != 0;
            if (rotateX) {
                displVec.y = (float) deltax;
            }
            if (rotateY) {
                displVec.x = (float) deltay;
            }
        }
        previousPos.x = currentPos.x;
        previousPos.y = currentPos.y;
    }

    public boolean isLeftButtonPressed() {
        return leftButtonPressed;
    }

    public boolean isRightButtonPressed() {
        return rightButtonPressed;
    }
}
```

The `MouseInput` class provides an `init` method which should be called during the initialization phase and registers a set of callbacks to process mouse events:

- `glfwSetCursorPosCallback` : Registers a callback that will be invoked when the mouse is moved.
- `glfwSetCursorEnterCallback` : Registers a callback that will be invoked when the mouse enters our window. We will be receiving mouse events even if the mouse is not in our window. We use this callback to track when the mouse is in our window.
- `glfwSetMouseButtonCallback` : Registers a callback that will be invoked when a mouse button is pressed.

www.dbooks.org

The `MouseInput` class provides an input method which should be called when game input is processed. This method calculates the mouse displacement from the previous position and stores it into `Vector2f` `displVec` variable so it can be used by our game.

The `MouseInput` class will be instantiated in our `GameEngine` class and will be passed as a parameter in the `init` and `update` methods of the game implementation (so we need to change the interface accordingly).

```
void input(Window window, MouseInput mouseInput);

void update(float interval, MouseInput mouseInput);
```

The mouse input will be processed in the input method of the `GameEngine` class before passing the control to the game implementation.

```
protected void input() {
    mouseInput.input(window);
    gameLogic.input(window, mouseInput);
}
```

Now we are ready to update our `DummyGame` class to process the keyboard and mouse input. The input method of that class will be like this:

```
@Override
public void input(Window window, MouseInput mouseInput) {
    cameraInc.set(0, 0, 0);
    if (window.isKeyPressed(GLFW_KEY_W)) {
        cameraInc.z = -1;
    } else if (window.isKeyPressed(GLFW_KEY_S)) {
        cameraInc.z = 1;
    }
    if (window.isKeyPressed(GLFW_KEY_A)) {
        cameraInc.x = -1;
    } else if (window.isKeyPressed(GLFW_KEY_D)) {
        cameraInc.x = 1;
    }
    if (window.isKeyPressed(GLFW_KEY_Z)) {
        cameraInc.y = -1;
    } else if (window.isKeyPressed(GLFW_KEY_X)) {
        cameraInc.y = 1;
    }
}
```

It just updates a `Vector3f` variable named `cameraInc` which holds the camera displacement that should be applied.
The update method of the `DummyGame` class modifies the camera position and rotation

according to the processes key and mouse events.

```
@Override
public void update(float interval, MouseInput mouseInput) {
    // Update camera position
    camera.movePosition(cameraInc.x * CAMERA_POS_STEP,
        cameraInc.y * CAMERA_POS_STEP,
        cameraInc.z * CAMERA_POS_STEP);

    // Update camera based on mouse
    if (mouseInput.isRightButtonPressed()) {
        Vector2f rotVec = mouseInput.getDisplVec();
        camera.moveRotation(rotVec.x * MOUSE_SENSITIVITY, rotVec.y * MOUSE_SENSITIVITY
, 0);
    }
}
```

Now we can add more cubes to our world, scale them set them up in a specific location and play with our new camera. As you can see all the cubes share the same mesh.

```
GameItem gameItem1 = new GameItem(mesh);
gameItem1.setScale(0.5f);
gameItem1.setPosition(0, 0, -2);

GameItem gameItem2 = new GameItem(mesh);
gameItem2.setScale(0.5f);
gameItem2.setPosition(0.5f, 0.5f, -2);

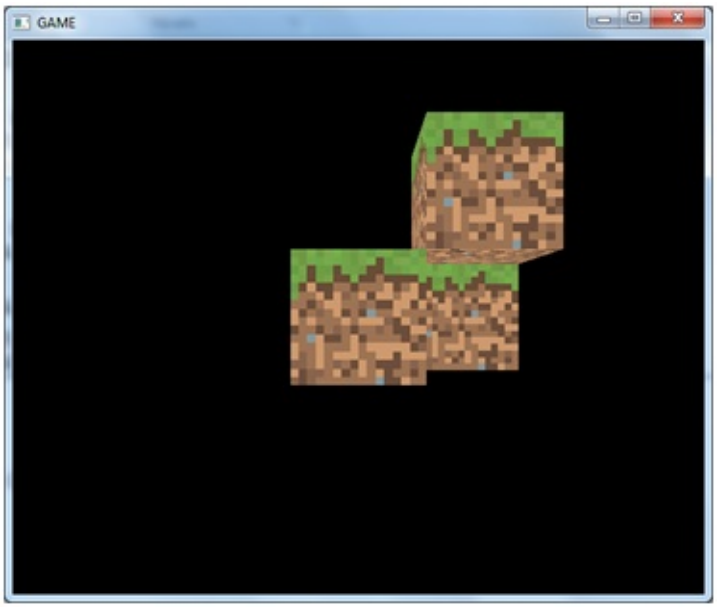
GameItem gameItem3 = new GameItem(mesh);
gameItem3.setScale(0.5f);
gameItem3.setPosition(0, 0, -2.5f);

GameItem gameItem4 = new GameItem(mesh);
gameItem4.setScale(0.5f);

gameItem4.setPosition(0.5f, 0, -2.5f);
gameItems = new GameItem[]{gameItem1, gameItem2, gameItem3, gameItem4};
```

You will get something like this.

www.dbooks.org


GAME

Loading more complex models

In this chapter we will learn to load more complex models defined in external files. Those models will be created by 3D modelling tools (such as Blender). Up to now we were creating our models by hand directly coding the arrays that define their geometry, in this chapter we will learn how to load models defined in OBJ format.

OBJ (or .OBJ) is a geometry definition open file format developed by Wavefront Technologies which has been widely adopted. An OBJ file defines the vertices, texture coordinates and polygons that compose a 3D model. It's a relative easy format to parse since is text based and each line defines an element (a vertex, a texture coordinate, etc.).

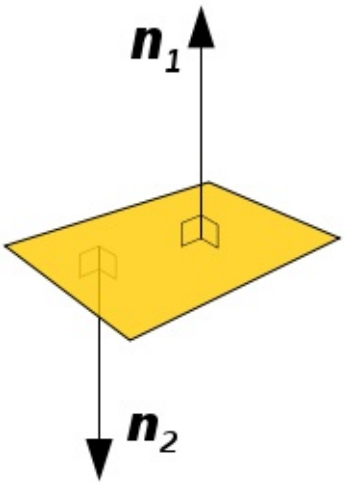
In an .obj file each line starts with a token with identifies the type of element:

- Comments are lines which start with #.
- The token "v" defines a geometric vertex with coordinates (x, y, z, w). Example: v 0.155 0.211 0.32 1.0.
- The token "vn" defines a vertex normal with coordinates (x, y, z). Example: vn 0.71 0.21 0.82. More on this later.
- The token "vt" defines a texture coordinate. Example: vt 0.500 1.
- The token "f" defines a face. With the information contained in these lines we will construct our indices array. We will handle only the case were faces are exported as triangles. It can have several variants:
    - It can define just vertex positions (f v1 v2 v3). Example: f 6 3 1. In this case this triangle is defined by the geometric vertices that occupy positions 6, 3 a and 1. (Vertex indices always starts by 1).
    - It can define vertex positions, texture coordinates and normals (f v1/t1/n1 v2/t2/n2 V3/t3/n3). Example: f 6/4/1 3/5/3 7/6/5. The first block is "6/4/1" and defines the coordinates, texture coordinates and normal vertex. What you see here is the position, so we are saying: pick the geometric vertex number six, the texture coordinate number 4 and the vertex normal number one.

OBJ format has many more entry types (like one to group polygons, defining materials, etc.). By now we will stick to this subset, our OBJ loader will ignore other entry types.

But what is anormal ? Let's define it first. When you have a plane its normal is a vector perpendicular to that plane which has a length equal to one.

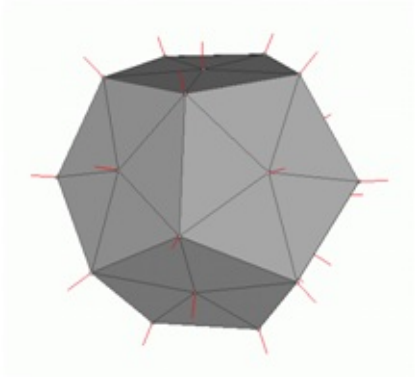
www.dbooks.org



As you can see in the figure above a plane can have two normals, which one should we use ? Normals in 3D graphics are used for lightning, so we should chose the normal which is oriented towards the source of light. In other words we should choose the normal that points out from the external face of our model.

When we have a 3D model, it is composed by polygons, triangles in our case. Each triangle is composed by three vertices. The Normal vector for a triangle will be the vector perpendicular to the triangle surface which has a length equal to one.

A vertex normal is associated to a specific vertex and is the combination of the normals of the surrounding triangles (of course its length is equal to one). Here you can see the vertex models of a 3D mesh (taken from Wikipedia)



Normals will be used for lighting.

So let's start creating our OBJ loader. First of all we will modify our `Mesh` class since now it's mandatory to use a texture. Some of the obj files that we may load may not define texture coordinates and we must be able to render them using a colour instead of a texture. In this case the face definition will be like this: "f v//n".

Our `Mesh` class will have a new attribute named `colour`

```
private Vector3f colour;
```

And the constructor will not require a `Texture` instance any more. Instead we will provide getters and setters for texture and colour attributes.

```
public Mesh(float[] positions, float[] textCoords, float[] normals, int[] indices) {
```

Of course, in the `render` and `cleanup` methods we must check if texture attribute is not null before using it. As you can see in the constructor we pass now a new array of floats named `normals`. How do we use normals for rendering ? The answer is easy it will be just another VBO inside our VAO, so we need to add this code.

```
// Vertex normals VBO
vboId = glGenBuffers();
vboIdList.add(vboId);
vecNormalsBuffer = MemoryUtil.memAllocFloat(normals.length);
vecNormalsBuffer.put(normals).flip();
glBindBuffer(GL_ARRAY_BUFFER, vboId);
glBufferData(GL_ARRAY_BUFFER, vecNormalsBuffer, GL_STATIC_DRAW);
glVertexAttribPointer(2, 3, GL_FLOAT, false, 0, 0);
```

In our `render` method we must enable this VBO before rendering and disable it when we have finished.

```
 // Draw the mesh
glBindVertexArray(getVaoId());
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glEnableVertexAttribArray(2);

glDrawElements(GL_TRIANGLES, getVertexCount(), GL_UNSIGNED_INT, 0);

// Restore state
glDisableVertexAttribArray(0);
glDisableVertexAttribArray(1);
glDisableVertexAttribArray(2);
glBindVertexArray(0);
glBindTexture(GL_TEXTURE_2D, 0);
```

Now that we have finished the modifications in the `Mesh` class we can change our code to use either texture coordinates or a fixed colour. Thus we need to modify our fragment shader like this:

www.dbooks.org

```
#version 330

in  vec2 outTexCoord;
out vec4 fragColor;

uniform sampler2D texture_sampler;
uniform vec3 colour;
uniform int useColour;

void main()
{
    if ( useColour == 1 )
    {
        fragColor = vec4(colour, 1);
    }
    else
    {
        fragColor = texture(texture_sampler, outTexCoord);
    }
}
```

As you can see we have create two new uniforms:

- `colour` : Will contain the base colour.
- `useColour` : It's a flag that we will set to 1 when we don't want to use textures.

In the `Renderer` class we need to create those two uniforms.

```
// Create uniform for default colour and the flag that controls it
shaderProgram.createUniform("colour");
shaderProgram.createUniform("useColour");
```

And like any other uniform, in the `render` method of the `Renderer` class we need to set the values for this uniforms for each `gameItem` .

```
for(GameItem gameItem : gameItems) {
    Mesh mesh = gameItem.getMesh();
    // Set model view matrix for this item
    Matrix4f modelViewMatrix = transformation.getModelViewMatrix(gameItem, viewMatrix)
;
    shaderProgram.setUniform("modelViewMatrix", modelViewMatrix);
    // Render the mes for this game item
    shaderProgram.setUniform("colour", mesh.getColour());
    shaderProgram.setUniform("useColour", mesh.isTextured() ? 0 : 1);
    mesh.render();
}
```

Now we can create a new class named `OBJLoader` which parses OBJ files and will create a `Mesh` instance with the data contained in it. You may find some other implementations in the web that may be a bit more efficient than this one but I think this version is simpler to understand. This will be an utility class which will have a static method like this:

```
public static Mesh loadMesh(String fileName) throws Exception {
```

The parameter `filename` specifies the name of the file, that must be in the CLASSPATH that contains the OBJ model.

The first thing that we will do in that method is to read the file contents and store all the lines in an array. Then we create several lists that will hold the vertices, the texture coordinates, the normals and the faces.

```
List<String> lines = Utils.readAllLines(fileName);

List<Vector3f> vertices = new ArrayList<>();
List<Vector2f> textures = new ArrayList<>();
List<Vector3f> normals = new ArrayList<>();
List<Face> faces = new ArrayList<>();
```

Then will parse each line and depending on the starting token will get a vertex position, a texture coordinate, a vertex normal or a face definition. At the end we will need to reorder that information.

www.dbooks.org

```
for (String line : lines) {
    String[] tokens = line.split("\\s+");
    switch (tokens[0]) {
        case "v":
            // Geometric vertex
            Vector3f vec3f = new Vector3f(
                Float.parseFloat(tokens[1]),
                Float.parseFloat(tokens[2]),
                Float.parseFloat(tokens[3]));
            vertices.add(vec3f);
            break;
        case "vt":
            // Texture coordinate
            Vector2f vec2f = new Vector2f(
                Float.parseFloat(tokens[1]),
                Float.parseFloat(tokens[2]));
            textures.add(vec2f);
            break;
        case "vn":
            // Vertex normal
            Vector3f vec3fNorm = new Vector3f(
                Float.parseFloat(tokens[1]),
                Float.parseFloat(tokens[2]),
                Float.parseFloat(tokens[3]));
            normals.add(vec3fNorm);
            break;
        case "f":
            Face face = new Face(tokens[1], tokens[2], tokens[3]);
            faces.add(face);
            break;
        default:
            // Ignore other lines
            break;
    }
}
return reorderLists(vertices, textures, normals, faces);
```

Before talking about reordering let's see how face definitions are parsed. We have create a class named `Face` which parses the definition of a face. A `Face` is composed by a list of indices groups, in this case since we are dealing with triangles we will have three indices group).



We will create another inner class named `IndexGroup` that will hold the information for a group.

```
protected static class IdxGroup {

    public static final int NO_VALUE = -1;

    public int idxPos;

    public int idxTextCoord;

    public int idxVecNormal;

    public IdxGroup() {
        idxPos = NO_VALUE;
        idxTextCoord = NO_VALUE;
        idxVecNormal = NO_VALUE;
        }
}
```

Our `Face` class will be like this.

www.dbooks.org

```
protected static class Face {

    /**
     * List of idxGroup groups for a face triangle (3 vertices per face).
    */
    private IdxGroup[] idxGroups = new IdxGroup[3];

    public Face(String v1, String v2, String v3) {
        idxGroups = new IdxGroup[3];
        // Parse the lines
        idxGroups[0] = parseLine(v1);
        idxGroups[1] = parseLine(v2);
        idxGroups[2] = parseLine(v3);
    }

    private IdxGroup parseLine(String line) {
        IdxGroup idxGroup = new IdxGroup();

        String[] lineTokens = line.split("/");
        int length = lineTokens.length;
        idxGroup.idxPos = Integer.parseInt(lineTokens[0]) - 1;
        if (length > 1) {
            // It can be empty if the obj does not define text coords
            String textCoord = lineTokens[1];
            idxGroup.idxTextCoord = textCoord.length() > 0 ? Integer.parseInt(textCoor
d) - 1 : IdxGroup.NO_VALUE;
            if (length > 2) {
                idxGroup.idxVecNormal = Integer.parseInt(lineTokens[2]) - 1;
            }
        }

        return idxGroup;
    }

    public IdxGroup[] getFaceVertexIndices() {
        return idxGroups;
    }
}
```

When parsing faces we may see objects with no textures but with vector normals, in this case a face line could be like this `f 11//1 17//1 13//1`, so we need to detect those cases.

Now we can talk about how to reorder the information we have. Finally we need to reorder that information. Our `Mesh` class expects four arrays, one for position coordinates, other for texture coordinates, other for vector normals and another one for the indices. The first three arrays shall have the same number of elements since the indices array is unique (note that the same number of elements does not imply the same length. Position elements, vertex coordinates, are 3D and are composed by three floats. Texture elements, texture

coordinates, are 2D and thus are composed by two floats). OpenGL does not allow us to define different indices arrays per type of element (if so, we would not need to repeat vertices while applying textures).

When you open an OBJ line you will first probably see that the list that holds the vertices positions has a higher number of elements than the lists that hold the texture coordinates and the number of vertices. That's something that we need to solve. Let's use a simple example which defines a quad with a texture with a pixel height (just for illustration purposes). The OBJ file may be like this (don't pay too much attention about the normals coordinate since it's just for illustration purpose).

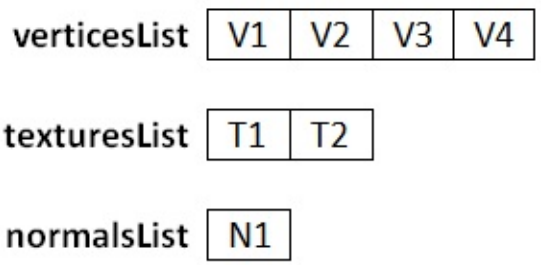
```
v 0 0 0
v 1 0 0
v 1 1 0
v 0 1 0

vt 0 1
vt 1 1

vn 0 0 1

f 1/2/1 2/1/1 3/2/1
f 1/2/1 3/2/1 4/1/1
```

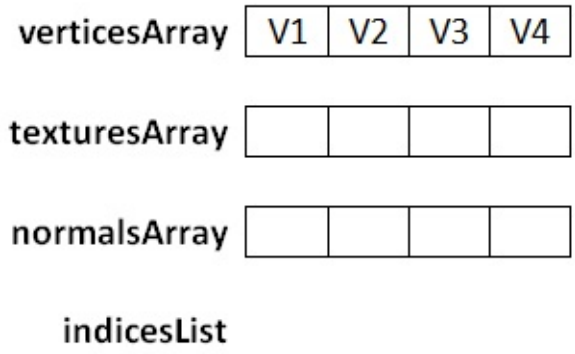
When we have finished parsing the file we have the following lists (the number of each element is its position in the file upon order of appearance)



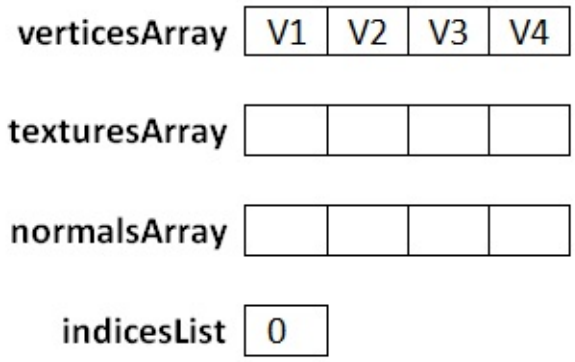
Now we will use the face definitions to construct the final arrays including the indices. A thing to take into consideration is that the order in which textures coordinates and vector normals are defined does not correspond to the orders in which vertices are defined. If the size of the lists would be the same and they were ordered, face definition lines would only just need to include a number per vertex.

So we need to order the data and setup accordingly to our needs. The first thing that we must do is create three arrays and one list, one for the vertices, other for the texture coordinates, other for the normals and the list for the indices. As we have said before the three arrays will have the same number of elements (equal to the number of vertices). The vertices array will have a copy of the list of vertices.

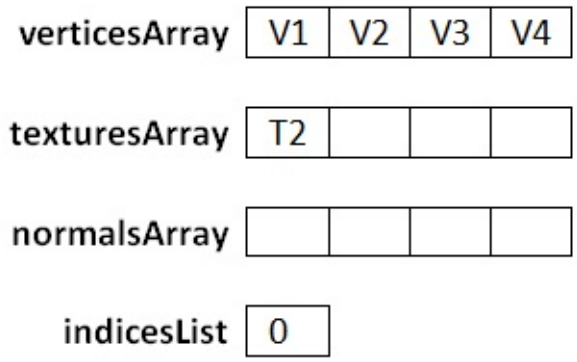
www.dbooks.org



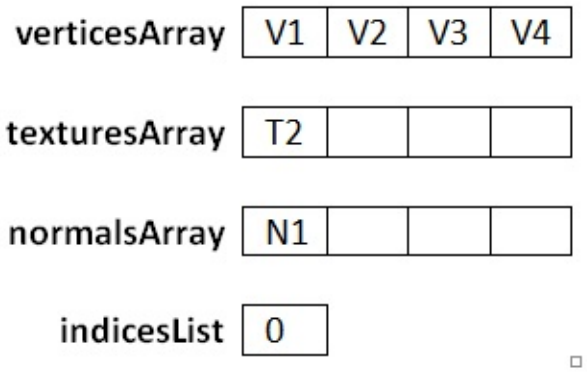
Now we start processing the faces. The first index group of the first face is 1/2/1. We use the first index in the index group, the one that defines the geometric vertex to construct the index list. Let's name it as `posIndex`.
Our face is specifiying that the we should add the index of the element that occupies the first position into our indices list. So we put the value of `posIndex` minus one into the `indicesList` (we must substract 1 since arrays start at 0 but OBJ file format assumes that they start at 1).



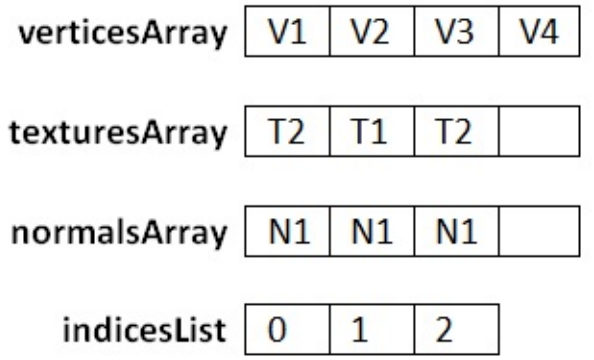
Then we use the rest of the indices of the index group to set up the `texturesArray` and `normalsArray`. The second index, in the index group, is 2, so what we must do is put the second texture coordinate in the same position as the one that occupies the vertex designated posIndex (V1).



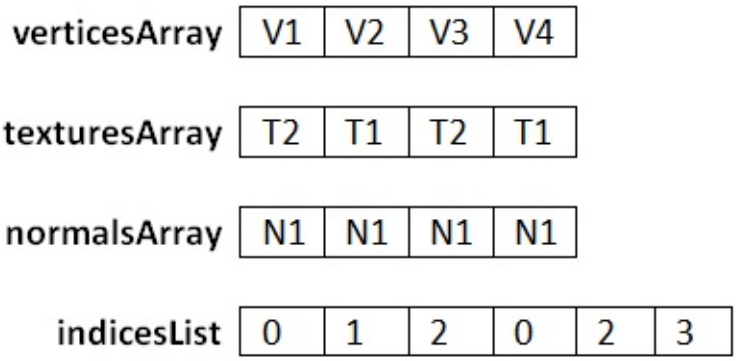
Then we pick the third index, which is 1, so what we must do is put the first vector normal coordinate in the same position as the one that occupies the vertex designated `posIndex` (V1).



After we have processed the first face the arrays and lists will be like this.



After we have processed the second face the arrays and lists will be like this.



The second face defines vertices which already have been assigned, but they contain the same values, so there's no problem in reprocessing this. I hope the process has been clarified enough, it can be some tricky until you get it. The methods that reorder the data are set below. Keep in mind that what we have are float arrays so we must transform those

www.dbooks.org

arrays of vertices, textures and normals into arrays of floats. So the length of these arrays will be the length of the vertices list multiplied by the number three in the case of vertices and normals or multiplied by two in the case of texture coordinates.

```
private static Mesh reorderLists(List<Vector3f> posList, List<Vector2f> textCoordList,
    List<Vector3f> normList, List<Face> facesList) {

    List<Integer> indices = new ArrayList();
    // Create position array in the order it has been declared
    float[] posArr = new float[posList.size() * 3];
    int i = 0;
    for (Vector3f pos : posList) {
        posArr[i * 3] = pos.x;
        posArr[i * 3 + 1] = pos.y;
        posArr[i * 3 + 2] = pos.z;
        i++;
    }
    float[] textCoordArr = new float[posList.size() * 2];
    float[] normArr = new float[posList.size() * 3];

    for (Face face : facesList) {
        IdxGroup[] faceVertexIndices = face.getFaceVertexIndices();
        for (IdxGroup indValue : faceVertexIndices) {
            processFaceVertex(indValue, textCoordList, normList,
                indices, textCoordArr, normArr);
        }
    }
    int[] indicesArr = new int[indices.size()];
    indicesArr = indices.stream().mapToInt((Integer v) -> v).toArray();
    Mesh mesh = new Mesh(posArr, textCoordArr, normArr, indicesArr);
    return mesh;
}

private static void processFaceVertex(IdxGroup indices, List<Vector2f> textCoordList,
    List<Vector3f> normList, List<Integer> indicesList,
    float[] texCoordArr, float[] normArr) {

    // Set index for vertex coordinates
    int posIndex = indices.idxPos;
    indicesList.add(posIndex);

    // Reorder texture coordinates
    if (indices.idxTextCoord >= 0) {
        Vector2f textCoord = textCoordList.get(indices.idxTextCoord);
        texCoordArr[posIndex * 2] = textCoord.x;
        texCoordArr[posIndex * 2 + 1] = 1 - textCoord.y;
    }
    if (indices.idxVecNormal >= 0) {
        // Reorder vectornormals
        Vector3f vecNorm = normList.get(indices.idxVecNormal);
        normArr[posIndex * 3] = vecNorm.x;
        normArr[posIndex * 3 + 1] = vecNorm.y;
        normArr[posIndex * 3 + 2] = vecNorm.z;
    }
}
```

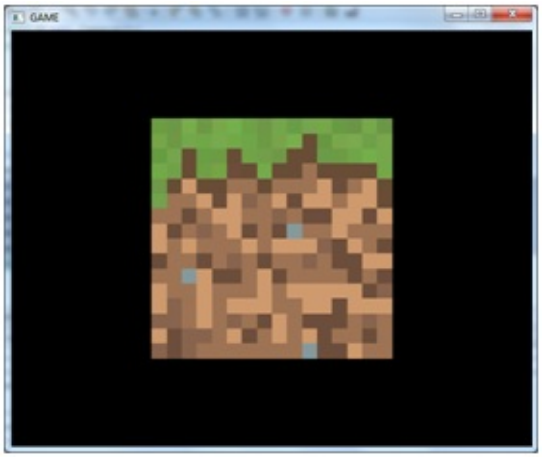

www.dbooks.org

Another thing to notice is that texture coordinates are in UV format so y coordinates need to be calculated as 1 minus the value contained in the file.

Now, at last, we can render obj models. I've included an OBJ file that contains the textured cube that we have been using in previous chapters. In order to use it in the `init` method of our `DummyGame` class we just need to construct a `GameItem` instance like this.

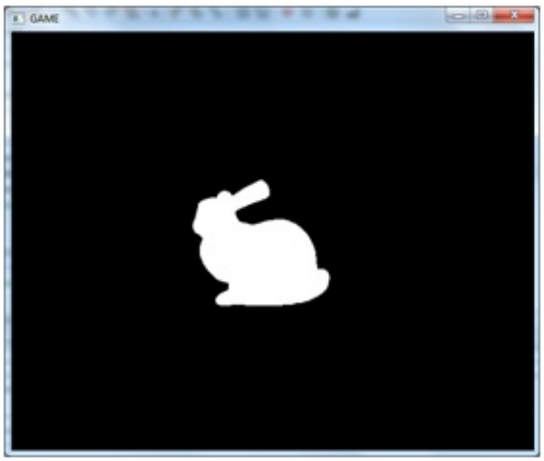
```
Texture texture = new Texture("/textures/grassblock.png");
mesh.setTexture(texture);
GameItem gameItem = new GameItem(mesh);
gameItem.setScale(0.5f);
gameItem.setPosition(0, 0, -2);
gameItems = new GameItem[]{gameItem};
```

And we will get our familiar textured cube.



We can now try with other models. We can use the famous Standford Bunny (it can be freely downloaded) model, which is included in the resources. This model is not textured so we can use it this way:

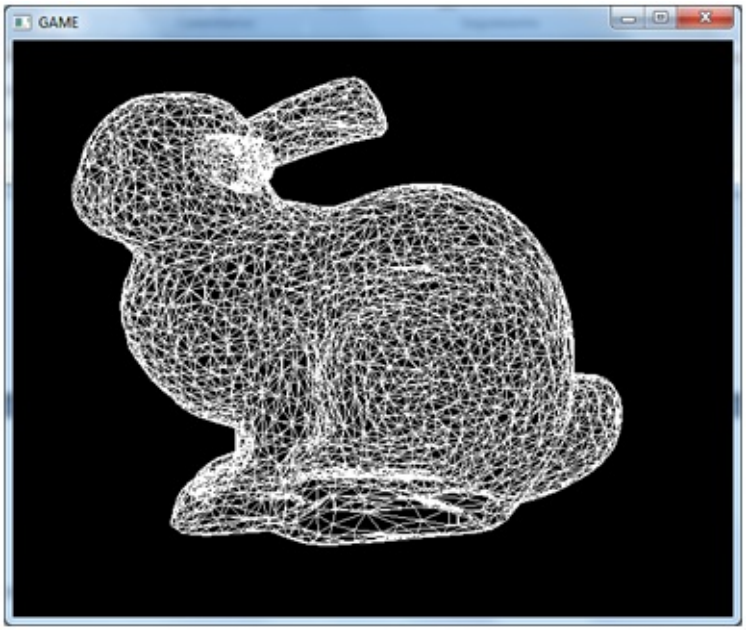
```
Mesh mesh = OBJLoader.loadMesh("/models/bunny.obj");
GameItem gameItem = new GameItem(mesh);
gameItem.setScale(1.5f);
gameItem.setPosition(0, 0, -2);
gameItems = new GameItem[]{gameItem};
```



The model looks a little bit strange because we have no textures and there's no light so we cannot appreciate the volumes but you can check that the model is correctly loaded. In the `Window` class when we set up the OpenGL parameters add this line.

```
glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );
```

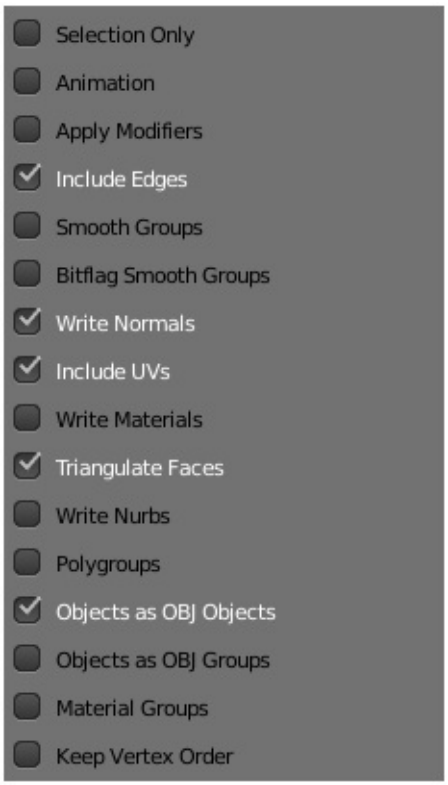
You should now see something like this when you zoom in.



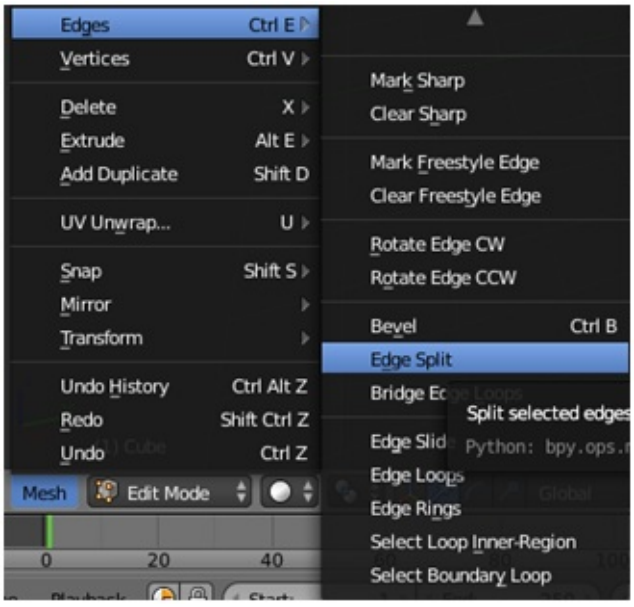
Now you can now see all the triangles that compose the model.

With this OBJ loader class you can now use Blender to create your models. Blender is a powerful tool but it can be some bit of overwhelming at first, there are lots of options, lots of key combinations and you need to take your time to do the most basic things by the first time. When you export the models using blender please make sure to include the normals and export faces as triangles.

www.dbooks.org



Also remeber to split edges when exporting, since we cannot assign several texture coordinates to the same vertex. Also, we need the normals to be defined per each triangle, not asigned to vertices. If you find light problems (next chapters), with some models, you should verify the normals. You can visualize them inside blender.

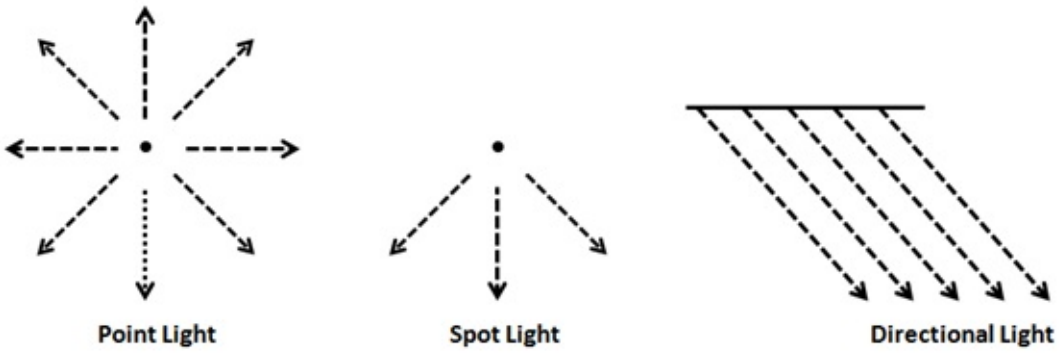


Let there be light

In this chapter we will learn how to add light to our 3D game engine. We will not implement a physical perfect light model because, taking aside the complexity, it would require a tremendous amount of computer recourses, instead we will implement an approximation which will provide decent results. We will use an algorithm named Phong shading (developed by Bui Tuong Phong). Another important thing to point is that we will only model lights but we won't model the shadows that should be generated by those lights (this will be done in another chapter).

Before we start, let us define some light types:

- **Point light**: This type of light models a light source that's emitted uniformly form a point in space in all directions.
- **Spot light**: This type of light models a light source that's emitted from a point in space, but instead of emitting in all directions is restricted to a cone.
- **Directional light**: This type for light models the light that we receive from the sun, all the objects in the 3D the space are hit by parallel ray lights coming from a specific direction. No matter if the object is close or of far away, all the ray lights impact the objects with the same angle.
- **Ambient light**: This type of light comes from everywhere in the space and illuminates all the objects in the same way.



Thus, to model light we need to take into consideration the type of light plus, its position and some other parameters like its colour. Of course, we must also consider the way that objects, impacted by ray lights, absorb and reflect light.

The Phong shading algorithm will model the effects of light for each point in our model, that is for every vertex. This is why it's called a local illumination simulation, and this is the reason which this algorithm will not calculate shadows, it will just calculate the light to be applied to every vertex without taking into consideration if the vertex is behind an object that

www.dbooks.org

blocks the light. We will overcome this in later chapters. But, because of that, is a very simple and fast algorithm that provides very good effects. We will use here a simplified version that does not take into account materials deeply.

The Phong algorithm considers three components for lighting:

- **Ambient light**: models light that comes from everywhere, this will serve us to illuminate (with the require intensity) the areas that are not hit by any light, it's like a background light.
- **Diffuse reflectance**: It takes into consideration that surfaces that are facing the light source are brighter.
- **Specular reflectance**: models how light reflects in polished or metallic surfaces

At the end what we want to obtain is a factor that, multiplied by colour assigned to a fragment, will set that colour brighter or darker depending on the light it receives. Let's name our components as A for ambient, D for diffuse and S for specular. That factor will be the addition of those components:

$$L = A + D + S$$

In fact, those components are indeed colours, that is the colour components that each light component contributes to. This is due to the fact that light components will not only provide a degree of intensity but it can modifiy the colour of model. In our fragment shader we just need to multiply that light colour by the original fragment colour (obtained from a texture or a base colour).

We can assign also different colours, for the same materials, that will be used in the ambient, diffuse and specular components. Hence, these components will be modulated by the colours associated to the material. If the material has a texture, we will simply use a single texture for each of the components.

So the final colour for a non textured material will be:
$L = A * ambientColour + D * diffuseColour + S * specularColour$.

And the final colour for a textured material will be:

$$L = A * textureColour + D * textureColour + S * textureColour$$

Ambient Light component

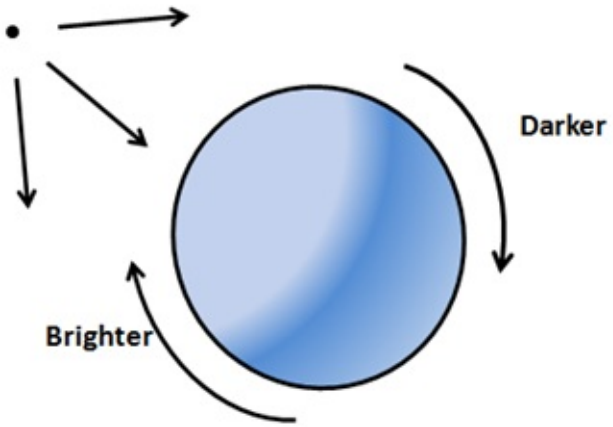
Let's view the first component, the ambient light component it's just a constant factor that will make all of our objects brighter or darker. We can use it to simulate light for a specific period of time (dawn, dusk, etc.) also it can be used to add some light to points that are not hit

directly by ray lights but could be lighted by indirect light (caused by reflections) in an easy way.

Ambient light is the easiest component to calculate, we just need to pass a colour, since it will be multiplied by our base colour it just modulates that base colour. Imagine that we have determined that a colour for a fragment is $(1.0, 0.0, 0.0)$, that is red colour. Without ambient light it will be displayed as a fully red fragment. If we set ambient light to $(0.5, 0.5, 0.5)$ the final colour will be $(0.5, 0, 0)$, that is a darker version of red. This light will darken all the fragments in the same way (it may seem to be a little strange to be talking about light that darkens objects but in fact that is the effect that we get). Besides that, it can add some colour if the RGB components are not the same, so we just need a vector to modulate ambient light intensity and colour.

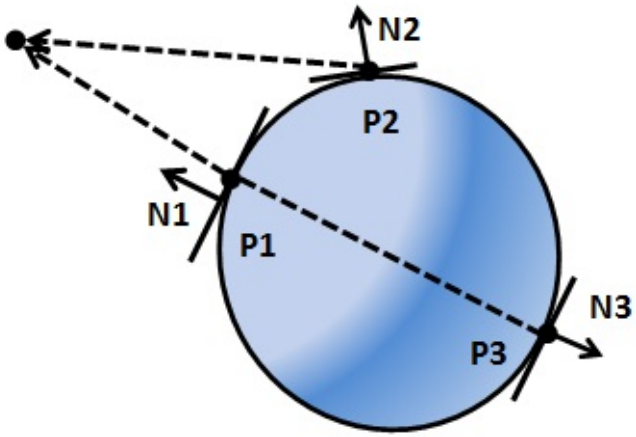
Diffuse refletance

Let's talk now about diffuse reflectance. It models the fact that surfaces which face in a perpendicular way to the light source look brighter than surfaces where light is received in a more indirect angle. Those objects receive more light, the light density (let me call it this way) is higher.



But, how do we calculate this ? Do you remember from previous chapter that we introduced the normal concept ? The normal was the vector perpendicular to a surface that had a length equal to one. So, Let's draw the normals for three points in the previous figure, as you can see, the normal for each point will be the vector perpendicular to the tangent plane for each point. Instead of drawing rays coming from the source of light we will draw vectors from each point to the point of light (that is, in the opposite direction).

www.dbooks.org



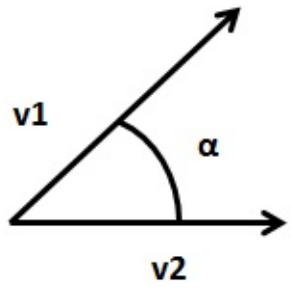
As you can see, the normal associated to $P1$, named $N1$, is parallel to the vector that points to the light source, which models the opposite of the light ray ($N1$ has been sketched displaced so you can see it, but it's equivalent mathematically). $P1$ has an angle equal to 0 with the vector that points to the light source. It's surface is perpendicular to the light source and $P1$ would be the brightest point.

The normal associated to $P2$, named $N2$, has an angle of around 30 degrees with the vector that points the light source, so it should be darker tan $P1$. Finally, the normal associated to $P3$, named $N3$, is also parallel to the vector that points to the light source but both vectors are in the opposite direction. $P3$ has an angle of 180 degrees with the vector that points the light source, and should not get any light at all.

So it seems that we have a good approach to determine the light intensity that gets to a point and it's related to the angle that forms the normal with a vector that points to the light source. How can we calculate this ?

There's a mathematical operation that we can use and it's called dot product. This operation takes two vectors and produces a number (a scalar), that is positive if the angle between them is small, and produces a negative number if the angle between them is wide. If both vectors are normalized, that is the both have a length equal to one, the dot product will be between -1 and 1. The dot product will be one if both vectors look in the same direction (angle 0) and it will be 0 if both vectors form a square angle and will be -1 if both vectors face opposite direction.

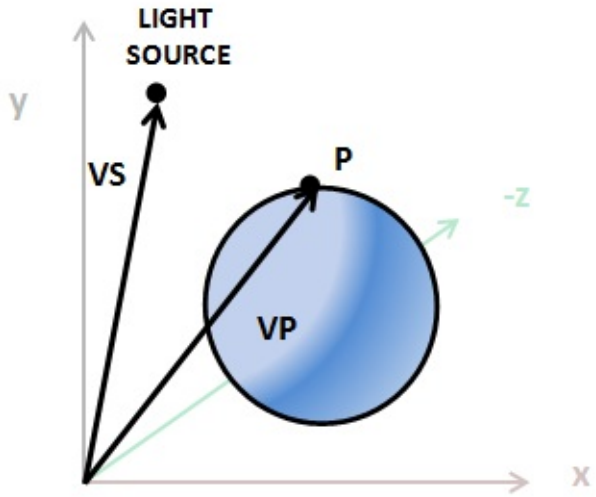
Let's define two vectors, $v1$ and $v2$, and let $alpha$ be the angle between them. The dot product is defined by the following formula.



$$v1 \cdot v2 = |v1| \cdot |v2| \cdot Cos\alpha$$

If both vectors are normalized, their length, their module will be equal to one, so the dot product is equal to the cosine if the angle between them. We will use that operation to calculate the diffuse reflectance component.

So we need to calculate the vector that points to the source of light. How we do this ? We have the position of each point (the vertex position) and we have the position of the light source. First of all, both coordinates must be in the same coordinate space. To simplify, let's assume that they are both in world coordinate space, then those positions are the coordinates of the vectors that point to the vertex position (VP) and to the light source (VS), as shown in the next figure.



If we substract VS from VP we get the vector that we are looking for which it's called L.

Now we can do the dot product between the vector that points to the light source and the normal, that product is called the Lambert term, due to Johann Lambert which was the first to propose that relation to model the brightness of a surface.

Let's summarize how we can calculate it, we define the following variables:

- $vPos$: Position of our vertex in model view space coordinates.
- $lPos$: Position of the light in view space coordinates.

www.dbooks.org

- $intensity$: Intensity of the light (from 0 to 1).
- $lColour$: Colour of the light.
- $normal$: The vertex normal.
- First we need to calculate the vector that points to the light source from current position: $toLightDirection = lPos - vPos$. The result of that operation needs to be normalized

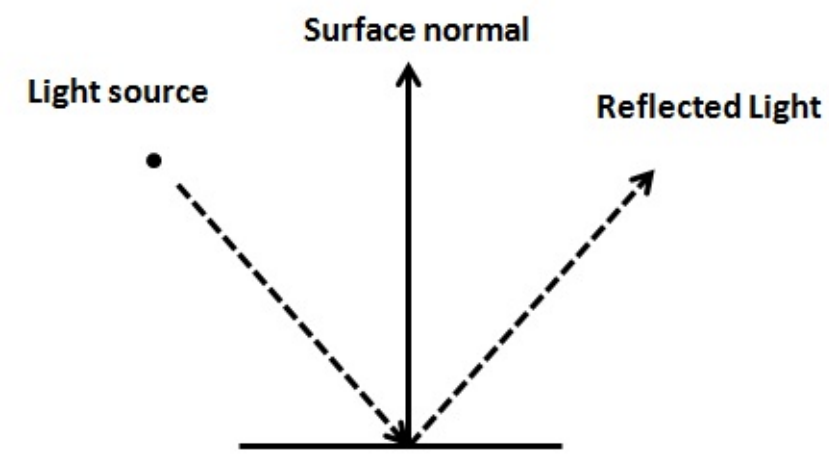
Then we need to calculate the diffuse factor (an scalar): $diffuseFactor = normal \cdot toLightDirection$. It's calculated as dot product between two vectors, since we want it to be between -1 and 1 both vectors need to be normalized. Colours need to be between 0 and 1 so if a value it's lower than 0 we will set it to 0.

Finally we just need to modulate the light colour by the diffuse factor and the light intensity:

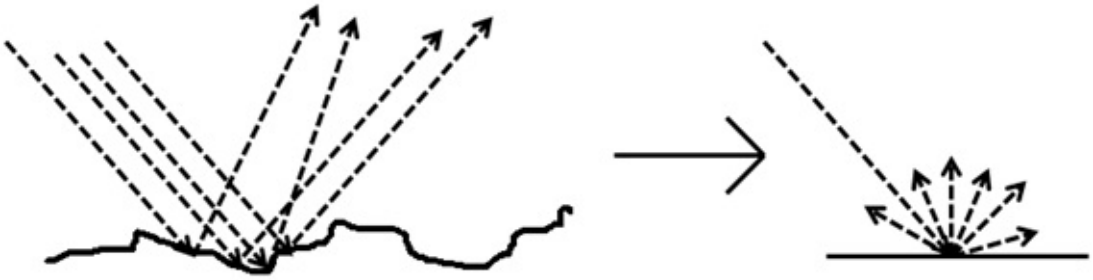
$colour = diffuseColour * lColour * diffuseFactor * intensity$

Specular component

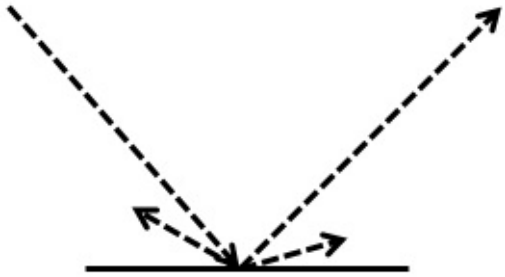
Let's view now the specular component, but first we need to examine how light is reflected. When light hits a surface some part of it is absorbed and the other part is reflected, if you remember from your physics class, reflection is when light bounces off an object.



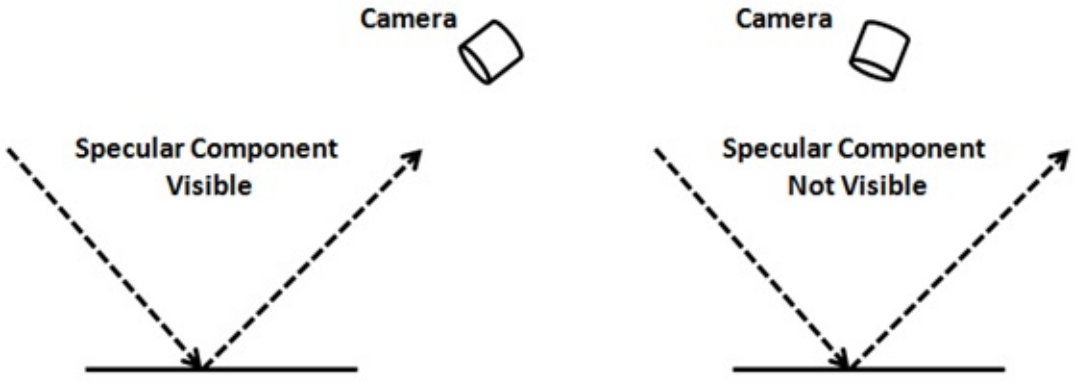
Of course, surfaces are not totally polished, and if you look at closer distance you will see a lot of imperfections. Besides that, you have many ray lights (photons in fact), that impact that surface, and that get reflected in a wide range of angles. Thus, what we see is like a beam of light being reflected from the surface. That is, light is diffused when impacting over a surface, and that's the disuse component that we have been talking about previously.



But when light impacts a polished surface, for instance a metal, the light suffers from lower diffusion and most of it gets reflected in the opposite direction as it hit that surface.



This is what the specular component models, and it depends on the material characteristics. Regarding specular reflectance, it's important to note that the reflected light will only be visible if the camera is in a proper position, that is, if it's in the area of where the reflected light is emitted.



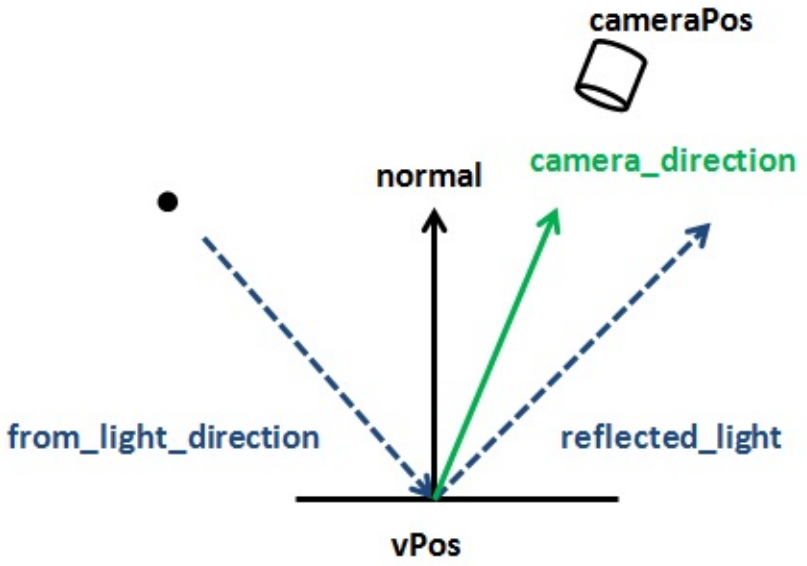
Once the mechanism that's behind sepecular reflection has been explained we are ready to calculate that component. First we need a vector that points from the light source to the vertex point. When we were calculating the difusse component we calculated just the opposite, a vector that points to the light source. $toLightDirection$, so let's calculate it as $fromLightDirection = -(toLightDirection)$.

Then we need to calculate the reflected light that results from the impact of the $fromLightDirection$ into the surface by taking into consideration its normal. There's a GLSL function that does that named `reflect`. So,

www.dbooks.org

$reflectedLight = reflect(fromLightSource, normal)$.

We also need a vector that points to the camera, let's name it $cameraDirection$, and it will be calculated as the difference between the camera position and the vertex position: $cameraDirection = cameraPos - vPos$. The camera position vector and the vertex position need to be in the same coordinate system and the resulting vector needs to be normalized. The following figure sketches the main components we have calculated up to now.



Now we need to calculate the light intensity that we see which we will call $specularFactor$. This component will be higher if the $cameraDirection$ and the $reflectedLight$ vectors are parallel and point in the same direction and will take its lower value if they point in opposite directions. In order to calculate this the dot product comes to the rescue again. So $specularFactor = cameraDirection \cdot reflectedLight$. We only want this value to be between 0 and 1 so if it's lower than 0 it will be set to 0.

We also need to take into consideration that this light must be more intense if the camera is pointing to the reflected light cone. This will be achieved by powering the $specularFactor$ to a parameter named $specularPower$.

$specularFactor = specularFactor^{specularPower}$.

Finally we need to model the reflectivity of the material, which will also modulate the intensity if the light reflected, this will be done with another parameter named reflectance. So the colour component of the specular component will be:
$specularColour * lColour * reflectance * specularFactor * intensity$.

Attenuation

We now know how to calculate the three components that will serve us to model a point light with an ambient light. But our light model is still not complete, the light that an object reflects is independent of the distance that the light is, we need to simulate light attenuation.

Attenuation is a function of the distance and light. The intensity of light is inversely proportional to the square of distance. That fact is easy to visualize, as light is propagating its energy is distributed along the surface of a sphere with a radius that's equal to the distance traveled by the light. The surface of a sphere is proportional to the square of its radius. We can calculate the attenuation factor with this formula:

$1.0/(atConstant + atLinear * dist + atExponent * dist^2)$.

In order to simulate attenuation we just need to multiply that attenuation factor by the final colour.

Implementation

Now we can start coding all the concepts described above, we will start with our shaders. Most of the work will be done in the fragment shader but we need to pass some data from the vertex shader to it. In previous chapter the fragment shader just received the texture coordinates, now we are going to pass also two more parameters:

- The vertex normal (normalized) transformed to model view space coordinates.
- The vertex position transformed to model view space coordinates. This is the code of the vertex shader.

www.dbooks.org

```
#version 330

layout (location=0) in vec3 position;
layout (location=1) in vec2 texCoord;
layout (location=2) in vec3 vertexNormal;

out vec2 outTexCoord;
out vec3 mvVertexNormal;
out vec3 mvVertexPos;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;

void main()
{
    vec4 mvPos = modelViewMatrix * vec4(position, 1.0);
    gl_Position = projectionMatrix * mvPos;
    outTexCoord = texCoord;
    mvVertexNormal = normalize(modelViewMatrix * vec4(vertexNormal, 0.0)).xyz;
    mvVertexPos = mvPos.xyz;
}
```

Before we continue with the fragment shader there's a very important concept that must be highlighted. From the code above you can see that `mvVertexNormal`, the variable contains the vertex normal, is transformed into model view space coordinates. This is done by multiplying the `vertexNormal` by the `modelViewMatrix` as with the vertex position. But there's a subtle difference, the w component of that vertex normal is set to 0 before multiplying it by the matrix: `vec4(vertexNormal, 0.0)`. Why are we doing this ? Because we do want the normal to be rotated and scaled but we do not want it to be translated, we are only interested into its direction but not in its position. This is achieved by setting is w component to 0 and is one of the advantages of using homogeneous coordinates, by setting the w component we can control what transformations are applied. You can do the matrix multiplication by hand and see why this happens.

Now we can start to do the real work in our fragment shader, besides declaring as input parameters the values that come from the vertex shader we are going to define some useful structures to model light and material characteristic. First of all, we will define the structures that model the light.

```
struct Attenuation
{
    float constant;
    float linear;
    float exponent;
};

struct PointLight
{
    vec3 colour;
    // Light position is assumed to be in view coordinates
    vec3 position;
    float intensity;
    Attenuation att;
};
```

A point light is defined by a colour, a position, a number between 0 and 1 which models its intensity and a set of parameters which will model the attenuation equation.

The structure that models a material characteristics is:

```
struct Material
{
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    int hasTexture;
    float reflectance;
};
```

A material is defined by a a set of colours (if we don't use texture to colour the fragments):

- The colout used for the ambient component.
- The colour used for the diffuse component.
- The colour used for the specular component.

A material also is defined by a flag that controls if it has an associated texture or not and a reflectance index. We will use the following uniforms in our fragment shader.

```
uniform sampler2D texture_sampler;
uniform vec3 ambientLight;
uniform float specularPower;
uniform Material material;
uniform PointLight pointLight;
uniform vec3 camera_pos;
```

We are creating new uniforms to set the following variables:

www.dbooks.org

- The ambient light: which will contain a colour that will affect every fragment in the same way.
- The specular power (the exponent used in the equation that was presented when talking about the specular light).
- A point light.
- The material characteristics.
- The camera position in view space coordinates.

We will also define some global variables that will hold the material colour components to be used in the ambient, diffuse and specular components. We use these variables since, if the component has a texture, we will use the same colour for all the components and we do not want to perform redundant texture lookups. The variables are defined like this:

```
vec4 ambientC;
vec4 diffuseC;
vec4 speculrC;
```

We now can define a function that will setup these variables accodring to the material characteristics:

```
void setupColours(Material material, vec2 textCoord)
{
    if (material.hasTexture == 1)
    {
        ambientC = texture(texture_sampler, textCoord);
        diffuseC = ambientC;
        speculrC = ambientC;
    }
    else
    {
        ambientC = material.ambient;
        diffuseC = material.diffuse;
        speculrC = material.specular;
    }
}
```

Now we are going to define a function that, taking as its input a point light, the vertex position and its normal returns the colour contribution calculated for the diffuse and specular light components described previously.

```
vec4 calcPointLight(PointLight light, vec3 position, vec3 normal)
{
    vec4 diffuseColour = vec4(0, 0, 0, 0);
    vec4 specColour = vec4(0, 0, 0, 0);

    // Diffuse Light
    vec3 light_direction = light.position - position;
    vec3 to_light_source  = normalize(light_direction);
    float diffuseFactor = max(dot(normal, to_light_source ), 0.0);
    diffuseColour = diffuseC * vec4(light.colour, 1.0) * light.intensity * diffuseFact
or;

    // Specular Light
    vec3 camera_direction = normalize(-position);
    vec3 from_light_source = -to_light_source;
    vec3 reflected_light = normalize(reflect(from_light_source, normal));
    float specularFactor = max( dot(camera_direction, reflected_light), 0.0);
    specularFactor = pow(specularFactor, specularPower);
    specColour = speculrC * specularFactor * material.reflectance * vec4(light.colour,
1.0);

    // Attenuation
    float distance = length(light_direction);
    float attenuationInv = light.att.constant + light.att.linear * distance +
        light.att.exponent * distance * distance;
    return (diffuseColour + specColour) / attenuationInv;
}
```

The previous code is relatively straight forward, it just calculates a colour for the diffuse component, another one for the specular component and modulates them by the attenuation suffered by the light in its travel to the vertex we are processing.

Please be aware that vertices coordinates are in view space. When calculating the specular component, we must get the direcction to the point of view, that is the camera. This, could be done like this:

```
vec3 camera_direction = normalize(camera_pos - position);
```

But, since `position` is in view space, the camera position is allways at the origin, that is, $(0, 0, 0)$, so we calculate it like this:

```
vec3 camera_direction = normalize(vec3(0, 0, 0) - position);
```

Which can be simplified like this:

www.dbooks.org

```
 vec3 camera_direction = normalize(-position);
```

With the previous function, the main function of the vertex function is very simple.

```
void main()
{
    setupColours(material, outTexCoord);

    vec4 diffuseSpecularComp = calcPointLight(pointLight, mvVertexPos, mvVertexNormal)
;

    fragColor = ambientC * vec4(ambientLight, 1) + diffuseSpecularComp;
}
```

The call to the `setupColours` function will set up the `ambientC`, `diffuseC` and `speculrC` variables with the appropriate colours. Then, we calculate the diffuse and specular components, taking into consideration the attennuation. We do this using a single function call for convenience, as it has been explained above. Final colour is calculated by adding the ambient component (multiplying `ambientC` by the ambient light). As you can see ambient light is not affected by attenuation.

We have introduced some new concepts into our shader that require further explanation, we are defining structures and using them as uniforms. How do we pass those structures ? First of all we will define two new classes that model the properties of a point light and a material, named oh surprise, `PointLight` and `Material`. They are just plain POJOs so you can check them in the source code that accompanies this book. Then, we need to create new methods in the `ShaderProgram` class, first to be able to create the uniforms for the point light and material structures.

```
public void createPointLightUniform(String uniformName) throws Exception {
    createUniform(uniformName + ".colour");
    createUniform(uniformName + ".position");
    createUniform(uniformName + ".intensity");
    createUniform(uniformName + ".att.constant");
    createUniform(uniformName + ".att.linear");
    createUniform(uniformName + ".att.exponent");
}

public void createMaterialUniform(String uniformName) throws Exception {
    createUniform(uniformName + ".ambient");
    createUniform(uniformName + ".diffuse");
    createUniform(uniformName + ".specular");
    createUniform(uniformName + ".hasTexture");
    createUniform(uniformName + ".reflectance");
}
```

As you can see, it's very simple, we just create a separate uniform for all the attributes that compose the structure. Now we need to create another two methods to set up the values of those uniforms and that will take as parameters `PointLight` and Material instances.

```
public void setUniform(String uniformName, PointLight pointLight) {
    setUniform(uniformName + ".colour", pointLight.getColor() );
    setUniform(uniformName + ".position", pointLight.getPosition());
    setUniform(uniformName + ".intensity", pointLight.getIntensity());
    PointLight.Attenuation att = pointLight.getAttenuation();
    setUniform(uniformName + ".att.constant", att.getConstant());
    setUniform(uniformName + ".att.linear", att.getLinear());
    setUniform(uniformName + ".att.exponent", att.getExponent());
}

public void setUniform(String uniformName, Material material) {
    setUniform(uniformName + ".ambient", material.getAmbientColour());
    setUniform(uniformName + ".diffuse", material.getDiffuseColour());
    setUniform(uniformName + ".specular", material.getSpecularColour());
    setUniform(uniformName + ".hasTexture", material.isTextured() ? 1 : 0);
    setUniform(uniformName + ".reflectance", material.getReflectance());
}
```

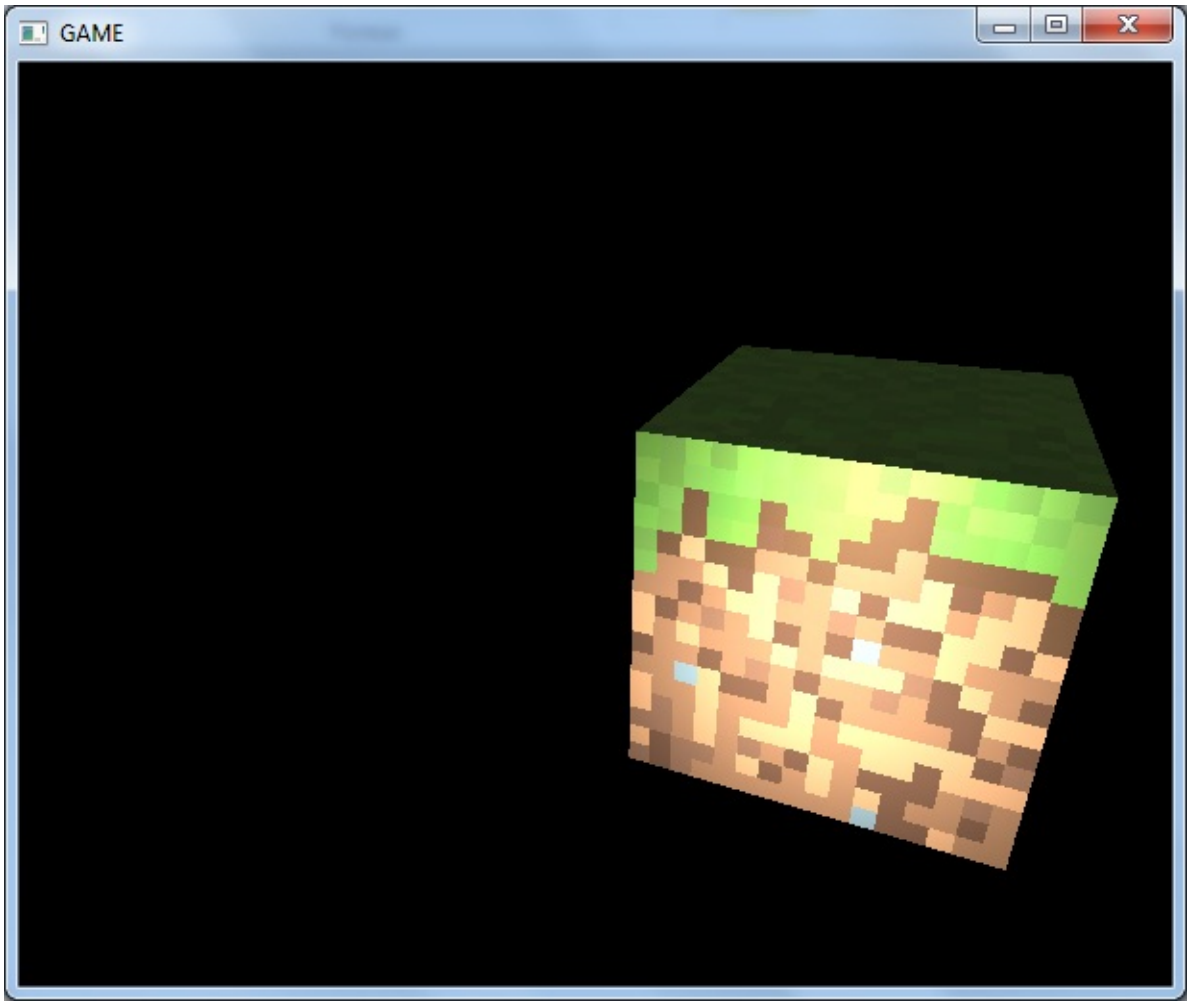
In this chapter source code you will see also that we also have modified the `Mesh` class to hold a material instance and that we have created a simple example that creates a point light that can be moved by using the "N" and "M" keys in order to show how a point light focusing over a mesh with a reflectance value higher than 0 looks like.

Let's get back to our fragment shader, as we have said we need another uniform which contains the camera position, camera_pos. These coordinates must be in view space. Usually we will set up light coordinates in world space coordinates, so we need to multiply them by the view matrix in order to be able to use them in our shader, so we need to create a new method in the `Transformation` class that returns the view matrix so we transform light coordinates.

```
// Get a copy of the light object and transform its position to view coordinates
PointLight currPointLight = new PointLight(pointLight);
Vector3f lightPos = currPointLight.getPosition();
Vector4f aux = new Vector4f(lightPos, 1);
aux.mul(viewMatrix);
lightPos.x = aux.x;
lightPos.y = aux.y;
lightPos.z = aux.z;
shaderProgram.setUniform("pointLight", currPointLight);
```


www.dbooks.org

We will not include the whole source code because this chapter would be too long and it would not contribute too much to clarify the concepts explained here. You can check it in the source code that accompanies this book.

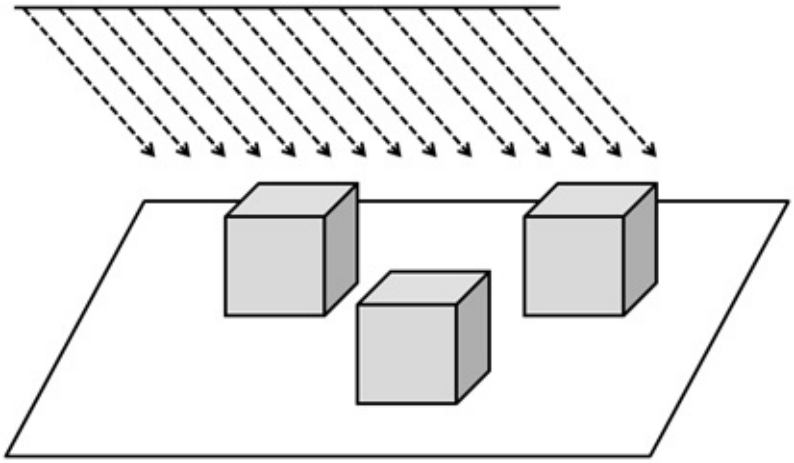


Let there be even more light

In this chapter we are going to implement other light types that we introduced in previous chapter. We will start with directional lightning.

Directional Light

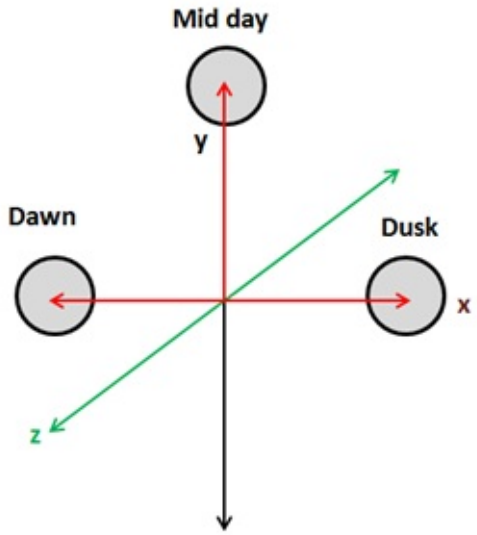
If you recall, directional lighting hits all the objects by parallel rays all coming from the same direction. It models light sources that are far away but have a high intensity such us the Sun.



Another characteristic of directional light is that it is not affected by attenuation. Think again about Sun light, all objects that are hit by ray lights are illuminated with the same intensity, the distance from the sun is so huge that the position of the objects is irrelevant. In fact, directional lights are modeled as light sources placed at the infinity, if it was affected by attenuation it would have no effect in any object (it's colour contribution would be equal to 0).

Besides that, directional light is composed also by a diffuse and specular components, the only differences with point lights is that it does not have a position but a direction and that it is not affected by attenuation. Let's get back to the direction attribute of directional light, and imagine we are modeling the movement of the sun across our 3D world. If we are assuming that the north is placed towards the increasing z-axis, the following picture shows the direction to the light source at dawn, midnight and dusk.

www.dbooks.org



Light directions for the above positions are:

- Dawn: (-1, 0, 0)
- Mid day: (0, 1, 0)
- Dusk: (1, 0, 0)

Side note: You may think that above coordinates are equal to position ones, but they model a vector, a direction, not a position. From the mathematical point of view a vector and a position are not distinguishable but they have a totally different meaning.

But, how do we model the fact that this light is located at the infinity ? The answer is by using the w coordinate, that is, by using homogeneous coordinates and setting the w coordinate to 0:

- Dawn: (-1, 0, 0, 0)
- Mid day: (0, 1, 0, 0)
- Dusk: (1, 0, 0, 0)

This is the same case as when we pass the normals, for normals we set the w component to 0 to state that we are not interested in displacements, just in the direction. Also, when we deal with directional light we need to do the same, camera translations should not affect the direction of a directional light.

So let’s start coding and model our directional light. The first thing that we are going to do is to create a class that models its attributes. It will be another POJO with a copy constructor which stores the direction, the colour and the intensity.

```
package org.lwjglb.engine.graph;

import org.joml.Vector3f;

public class DirectionalLight {

    private Vector3f color;

    private Vector3f direction;

    private float intensity;

    public DirectionalLight(Vector3f color, Vector3f direction, float intensity) {
        this.color = color;
        this.direction = direction;
        this.intensity = intensity;
    }

    public DirectionalLight(DirectionalLight light) {
        this(new Vector3f(light.getColor()), new Vector3f(light.getDirection()), light
.getIntensity());
    }

    // Getters and settes beyond this point...
```

As you can see, we are still using a `Vector3f` to model the direction. Keep calm, we will deal with the w component when we transfer the directional light to the shader. And by the way, the next thing that we will do is to update the `ShaderProgram` to create and update the uniform that will hold the directional light.

In our fragment shader we will define a structure that models a directional light.

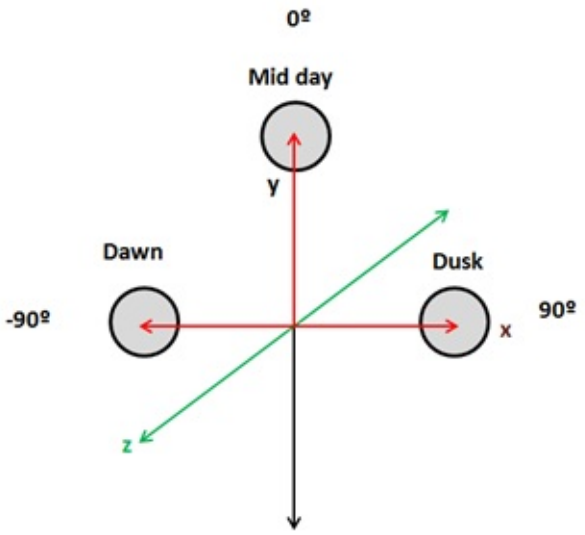
```
struct DirectionalLight
{
    vec3 colour;
    vec3 direction;
    float intensity;
};
```

With that definition the new methods in the `ShaderProgram` class are straight forward.

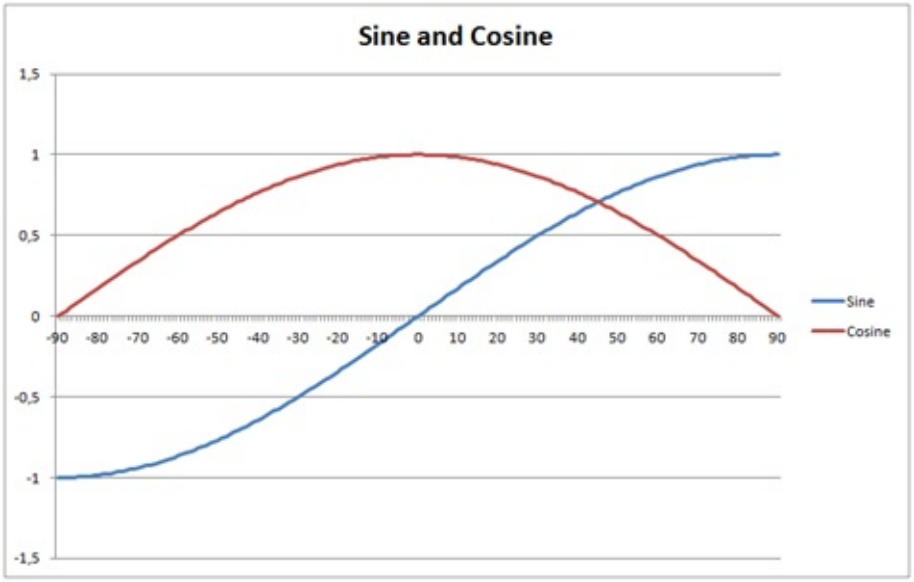
www.dbooks.org

```
// ...
public void createDirectionalLightUniform(String uniformName) throws Exception {
    createUniform(uniformName + ".colour");
    createUniform(uniformName + ".direction");
    createUniform(uniformName + ".intensity");
}
// ...
public void setUniform(String uniformName, DirectionalLight dirLight) {
    setUniform(uniformName + ".colour", dirLight.getColor() );
    setUniform(uniformName + ".direction", dirLight.getDirection());
    setUniform(uniformName + ".intensity", dirLight.getIntensity());
}
```

Now we need to use that uniform. We will model how the sun appears to move across the sky by controlling its angle in our `DummyGame` class.



We need to update light direction so when the sun it's at dawn (-90º) its direction is (-1,0,0) and its x coordinate progressively increases from -1 to 0 and the "y" coordinate increases to 1 as it approaches mid day. Then the "x" coordinate increases to 1 and the "y" coordinates decreases to 0 again. This can be done by setting the x coordinate to the $sine$ of the angle and y coordinate to the $cosine$ of the angle.



We will also modulate light intensity, the intensity will be increasing when it's getting away from dawn and will decrease as it approaches to dusk. We will simulate the night by setting the intensity to 0. Besides that, we will also modulate the colour so the light gets more red at dawn and at dusk. This will be done in the update method of the `DummyGame` class.

```
// Update directional light direction, intensity and colour
lightAngle += 1.1f;
if (lightAngle > 90) {
    directionalLight.setIntensity(0);
    if (lightAngle >= 360) {
        lightAngle = -90;
    }
} else if (lightAngle <= -80 || lightAngle >= 80) {
    float factor = 1 - (float)(Math.abs(lightAngle) - 80)/ 10.0f;
    directionalLight.setIntensity(factor);
    directionalLight.getColor().y = Math.max(factor, 0.9f);
    directionalLight.getColor().z = Math.max(factor, 0.5f);
} else {
    directionalLight.setIntensity(1);
    directionalLight.getColor().x = 1;
    directionalLight.getColor().y = 1;
    directionalLight.getColor().z = 1;
}
double angRad = Math.toRadians(lightAngle);
directionalLight.getDirection().x = (float) Math.sin(angRad);
directionalLight.getDirection().y = (float) Math.cos(angRad);
```

Then we need to pass the directional light to our shaders in the render method of the `Renderer` class.

www.dbooks.org

```
// Get a copy of the directional light object and transform its position to view coord
inates
DirectionalLight currDirLight = new DirectionalLight(directionalLight);
Vector4f dir = new Vector4f(currDirLight.getDirection(), 0);
dir.mul(viewMatrix);
currDirLight.setDirection(new Vector3f(dir.x, dir.y, dir.z));
shaderProgram.setUniform("directionalLight", currDirLight);
```

As you can see we need to transform the light direction coordinates to view space, but we set the w component to 0 since we are not interested in applying translations.

Now we are ready to do the real work which will be done in the fragment shader since the vertex shader does not be modified. We have yet stated above that we need to define a new struct, named `DirectionalLight`, to model a directional light, and we will need a new uniform form that.

```
uniform DirectionalLight directionalLight;
```

We need to refactor our code a little bit, in the previous chapter we had a function called `calcPointLight` that calculate the diffuse and specular components and also applied the attenuation. As we have explained directional light also contributes to the diffuse and specular components but is not affected by attenuation, so we will create a new function named `calcLightColour` that just calculates those components.

```
vec4 calcLightColour(vec3 light_colour, float light_intensity, vec3 position, vec3 to_
light_dir, vec3 normal)
{
    vec4 diffuseColour = vec4(0, 0, 0, 0);
    vec4 specColour = vec4(0, 0, 0, 0);

    // Diffuse Light
    float diffuseFactor = max(dot(normal, to_light_dir), 0.0);
    diffuseColour = diffuseC * vec4(light_colour, 1.0) * light_intensity * diffuseFact
or;

    // Specular Light
    vec3 camera_direction = normalize(camera_pos - position);
    vec3 from_light_dir = -to_light_dir;
    vec3 reflected_light = normalize(reflect(from_light_dir , normal));
    float specularFactor = max( dot(camera_direction, reflected_light), 0.0);
    specularFactor = pow(specularFactor, specularPower);
    specColour = speculrC * light_intensity  * specularFactor * material.reflectance *
vec4(light_colour, 1.0);

    return (diffuseColour + specColour);
}
```

Then the method `calcPointLight` applies attenuation factor to the light colour calculated in the previous function.

```
vec4 calcPointLight(PointLight light, vec3 position, vec3 normal)
{
    vec3 light_direction = light.position - position;
    vec3 to_light_dir  = normalize(light_direction);
    vec4 light_colour = calcLightColour(light.colour, light.intensity, position, to_li
ght_dir, normal);

    // Apply Attenuation
    float distance = length(light_direction);
    float attenuationInv = light.att.constant + light.att.linear * distance +
        light.att.exponent * distance * distance;
    return light_colour / attenuationInv;
}
```

We will create also a new function to calculate the effect of a directional light which just invokes the `calcLightColour` function with the light direction.

www.dbooks.org

```
vec4 calcDirectionalLight(DirectionalLight light, vec3 position, vec3 normal)
{
    return calcLightColour(light.colour, light.intensity, position, normalize(light.di
rection), normal);
}
```

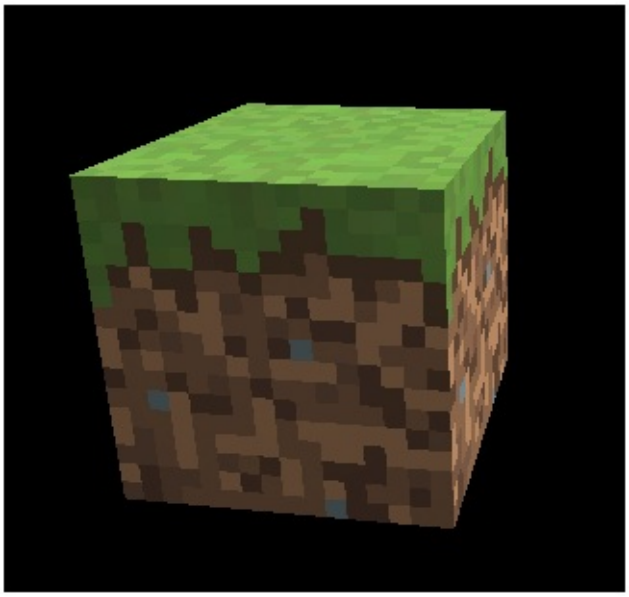
Finally, our main method just aggregates the colour components of the ambient point and directional lights to calculate the fragment colour.

```
void main()
{
    setupColours(material, outTexCoord);

    vec4 diffuseSpecularComp = calcDirectionalLight(directionalLight, mvVertexPos, mvV
ertexNormal);
    diffuseSpecularComp += calcPointLight(pointLight, mvVertexPos, mvVertexNormal);

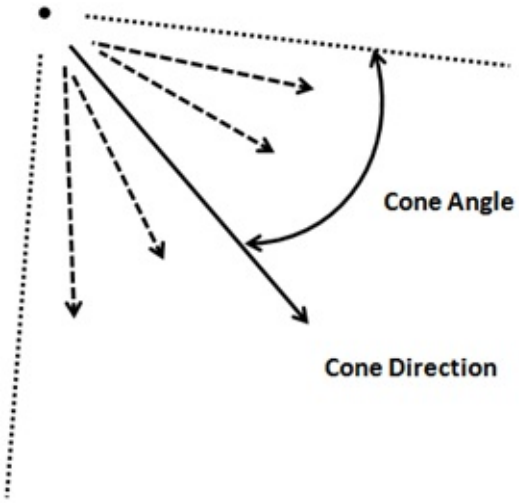
    fragColor = ambientC * vec4(ambientLight, 1) + diffuseSpecularComp;
}
```

And that's it, we can now simulate the movement of the, artificial, sun across the sky and get something like this (movement is accelerated so it can be viewed without waiting too long).

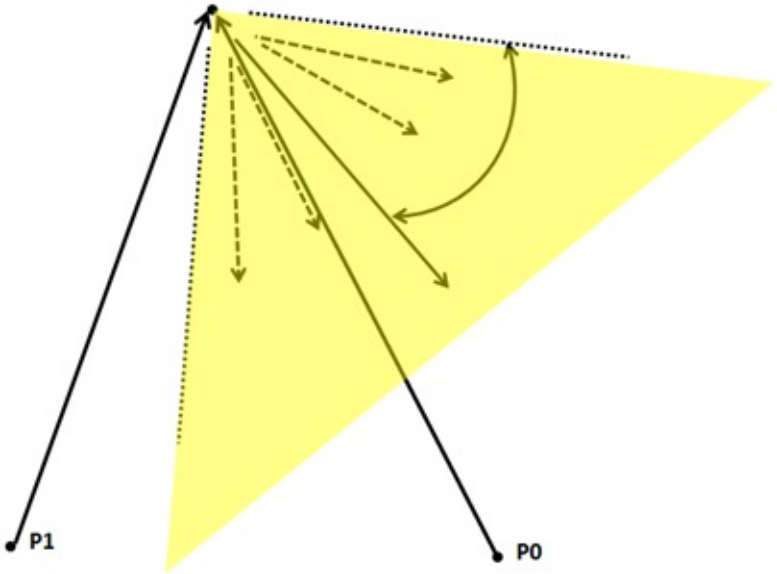


Spot Light

Now we will implement spot lights which are very similar to point lights but the emitted light is restricted to a 3D cone. It models the light that comes out from focuses or any other light source that does not emit in all directions. A spot light has the same attributes as a point light but adds two new parameters, the cone angle and the cone direction.

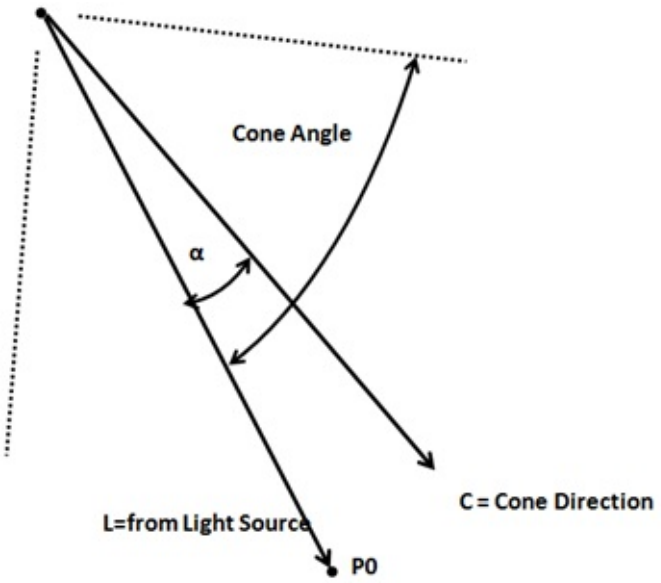


Spot light contribution is calculated in the same way as a point light with some exceptions. The point which the vector that points from the vertex position to the light source is not contained inside the light cone are not affected by the point light.



How do we calculate if it’s inside the light cone or not ? We need to do a dot product again between the vector that points from the light source and the cone direction vector (both of them normalized).

www.dbooks.org



The dot product between L and C vectors is equal to: $L \cdot C = |L| \cdot |C| \cdot Cos(\alpha)$. If, in our spot light definition we store the cosine of the cutoff angle, if the dot product is higher than that value we will know that it is inside the light cone (recall the cosine graph, when α angle is 0, the cosine will be 1, the smaller the angle the higher the cosine).

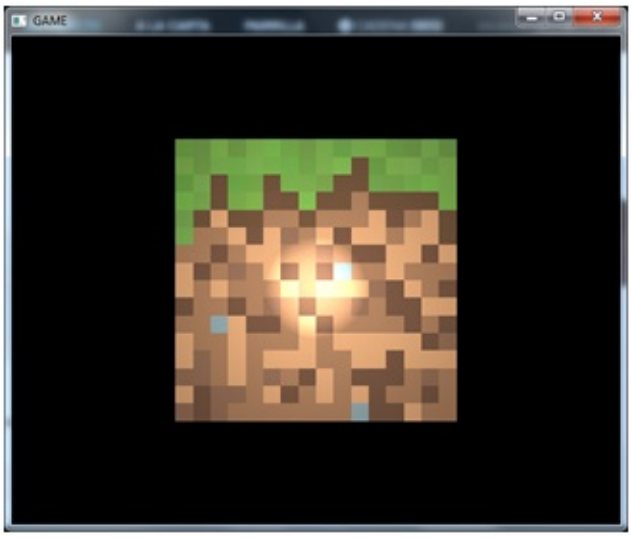
The second difference is that the points that are far away from the cone vector will receive less light, that is, the attenuation will be higher. There are several ways of calculate this, we will chose a simple approach by multiplying the attenuation by the following factor:

$1 - (1 - Cos(\alpha))/(1 - Cos(cutOffAngle)$

(In our fragment shaders we won't have the angle but the cosine of the cut off angle. You can check that the formula above produces values from 0 to 1, 0 when the angle is equal to the cutoff angle and 1 when the angle is 0).

The implementation will be very similar to the rest of lights. We need to create a new class named `SpotLight`, set up the appropriate uniforms, pass it to the shader and modify the fragment shader to get it. You can check the source code for this chapter.

Another important thing when passing the uniforms is that translations should not be applied to the light cone direction since we are only interested in directions. So as in the case of the directional light, when transforming to view space coordinates we must set w component to 0.



Multiple Lights

So at last we have finally implemented all the four types of light, but currently we can only use one instance for each type. This is ok for ambient and directional light but we definitively want to use several point and spot lights. We need to set up our fragment shader to receive a list of lights, so we will use arrays to store that information. Let's see how this can be done.

Before we start, it's important to note that in GLSL the length of the array must be set at compile time so it must be big enough to accommodate all the objects we need later, at runtime. The first thing that we will do is define some constants to set up the maximum number of point and spot lights that we are going to use.

```
const int MAX_POINT_LIGHTS = 5;
const int MAX_SPOT_LIGHTS = 5;
```

Then we need to modify the uniforms that previously store just a single point and spot light to use an array.

```
uniform PointLight pointLights[MAX_POINT_LIGHTS];
uniform SpotLight spotLights[MAX_SPOT_LIGHTS];
```

In the main function we just need to iterate over those arrays to calculate the colour contributions of each instance using the existing functions. We may not pass as many lights as the array length so we need to control it. There are many possible ways to do this, one is to pass a uniform with the actual array length but this may not work with older graphics cards. Instead we will check the light intensity (empty positions in array will have a light intensity equal to 0).

www.dbooks.org

```
for (int i=0; i<MAX_POINT_LIGHTS; i++)
{
    if ( pointLights[i].intensity > 0 )
    {
        diffuseSpecularComp += calcPointLight(pointLights[i], mvVertexPos, mvVertexNor
mal);
    }
}

for (int i=0; i<MAX_SPOT_LIGHTS; i++)
{
    if ( spotLights[i].pl.intensity > 0 )
    {
        diffuseSpecularComp += calcSpotLight(spotLights[i], mvVertexPos, mvVertexNorma
l);
    }
}
```

Now we need to create those uniforms in the `Render` class. When we are using arrays we need to create a uniform for each element of the list. So, for instance, for the $pointLights$ array we need to create a uniform named `pointLights[0]`, `pointLights[1]`, etc. And of ocurse, this translates also to the structure attributes, so we will have `pointLights[0].colour`, `pointLights[1], colour`, etc. The methods to create those uniforms are as follows.

```
public void createPointLightListUniform(String uniformName, int size) throws Exception
{
    for (int i = 0; i < size; i++) {
        createPointLightUniform(uniformName + "[" + i + "]");
    }
}

public void createSpotLightListUniform(String uniformName, int size) throws Exception
{
    for (int i = 0; i < size; i++) {
        createSpotLightUniform(uniformName + "[" + i + "]");
    }
}
```

We also need methods to set up the values of those uniforms.

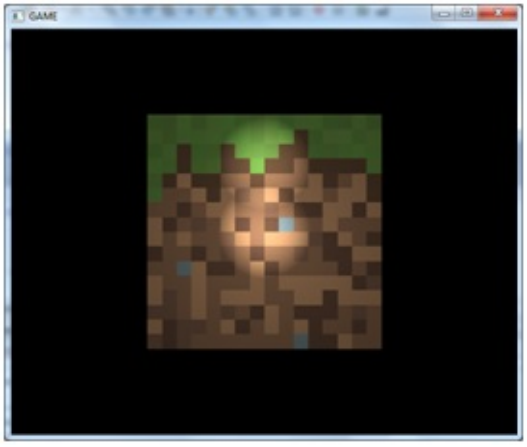
```
public void setUniform(String uniformName, PointLight[] pointLights) {
    int numLights = pointLights != null ? pointLights.length : 0;
    for (int i = 0; i < numLights; i++) {
        setUniform(uniformName, pointLights[i], i);
    }
}

public void setUniform(String uniformName, PointLight pointLight, int pos) {
    setUniform(uniformName + "[" + pos + "]", pointLight);
}

public void setUniform(String uniformName, SpotLight[] spotLights) {
    int numLights = spotLights != null ? spotLights.length : 0;
    for (int i = 0; i < numLights; i++) {
        setUniform(uniformName, spotLights[i], i);
    }
}

public void setUniform(String uniformName, SpotLight spotLight, int pos) {
    setUniform(uniformName + "[" + pos + "]", spotLight);
}
```

Finally we just need to update the `Render` class to receive a list of point and spot lights, and modify accordingly the `DummyGame` class to create those list to see something like this.



Game HUD

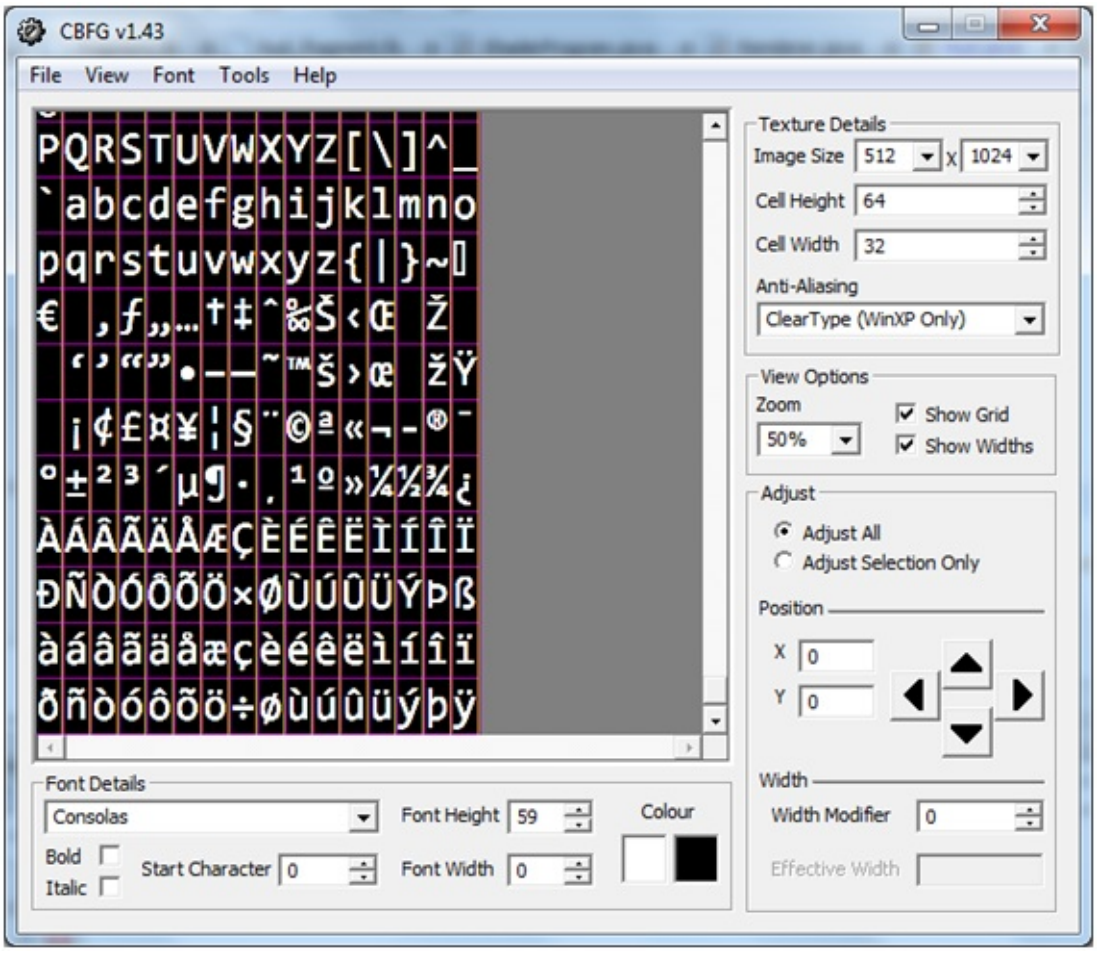
In this chapter we will create a HUD (Heads-Up Display) for our game. That is, a set of 2D shapes and text that are displayed at any time over the 3D scene to show relevant information. We will create a simple HUD that will serve us to show some basic techniques for representing that information.

When you examine the source code for this chpater, you will see also that some little refactoring has been applied to the source code. The changes affect especially the `Renderer` class in order to prepare it for the HUD rendering.

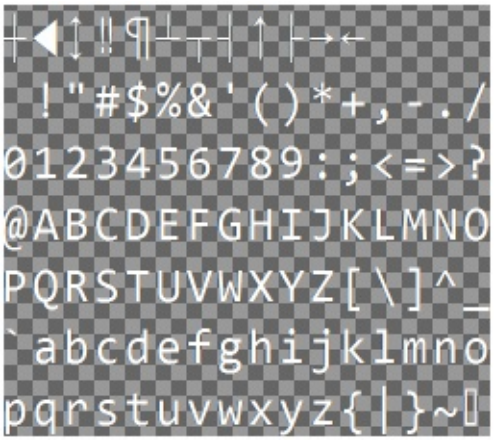
Text rendering

The first thing that we will do to create a HUD is render text. In order to do that, we are going to map a texture that contains alphabet characters into a quad. That quad will be divided by a set of tiles which will represent a single letter. later on, we will use that texture to draw the text in the screen. So the first step is to create the texture that contains the alphabet. You can use many programs out there that can do this task, such as, CBG, F2IBuilder, etc. In this case, We will use Codehead's Bitmap Font Generator (CBFG).

CBG lets you configure many options such as the texture size, the font type, the anti-aliasing to be applied, etc. The following figure depicts the configuration that we will use to generate a texture file. In this chapter we will assume that we will be rendering text encoded in ISO-8859-1 format, if you need to deal with different character sets you will need to tweak a little bit the code.



When you have finished configuring all the settings in CBG you can export the result to several image formats. In this case we will export it as a BMP file and then transform it to PNG so it can be loaded as a texture. When transforming it to PNG we will set up also the black background as transparent, that is, we will set the black colour to have an alpha value equal to 0 (You can use tools like GIMP to do that). At the end you will have something similar as the following picture.



As you can see, the image has all the characters displayed in rows and columns. In this case the image is composed by 15 columns and 17 rows. By using the character code of a specific letter we can calculate the row and the column that is enclosed in the image. The

www.dbooks.org

column can be calculated as follows: $column = code \ mod \ numberOfColumns$. Where mod is the module operator. The row can be calculated as follows: $row = code / numberOfCols$, in this case we will do a integer by integer operation so we can ignore the decimal part.

We will create a new class named `TextItem` that will construct all the graphical elements needed to render text. This is a simplified version that does not deal with multiline texts, etc. but it will allow us to present textual information in the HUD. Here you can see the first lines and the constructor of this class.

```
package org.lwjglb.engine;

import java.nio.charset.Charset;
import java.util.ArrayList;
import java.util.List;
import org.lwjglb.engine.graph.Material;
import org.lwjglb.engine.graph.Mesh;
import org.lwjglb.engine.graph.Texture;

public class TextItem extends GameItem {

    private static final float ZPOS = 0.0f;

    private static final int VERTICES_PER_QUAD = 4;

    private String text;

    private final int numCols;

    private final int numRows;

    public TextItem(String text, String fontFileName, int numCols, int numRows) throws
 Exception {
        super();
        this.text = text;
        this.numCols = numCols;
        this.numRows = numRows;
        Texture texture = new Texture(fontFileName);
        this.setMesh(buildMesh(texture, numCols, numRows));
    }
```

As you can see this class extends the `GameItem` class, this is because we will be interested in changing the text position in the screen and may also need to scale and rotate it. The constructor receives the text to be displayed and the relevant data of the texture file that will be used to render it (the file that contains the image and the number of columns and rows).

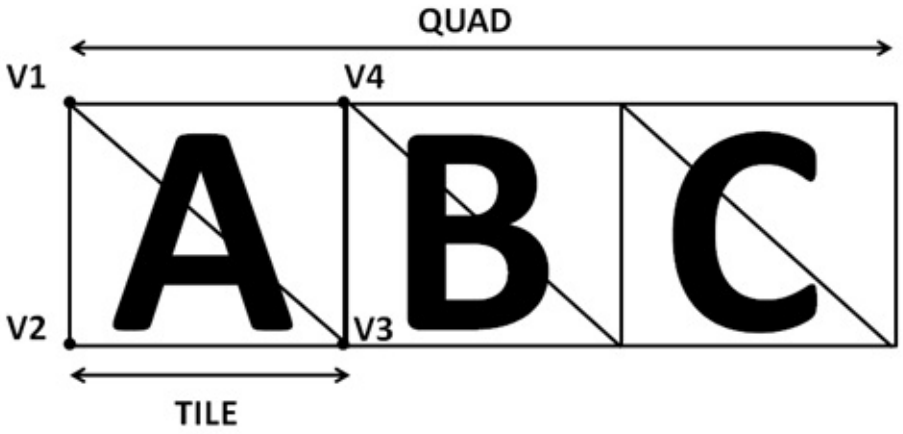
In the constructor we load the texture image file and invoke a method that will create a `Mesh` instance that models our text. Let's examine the `buildMesh` method.

```
private Mesh buildMesh(Texture texture, int numCols, int numRows) {
    byte[] chars = text.getBytes(Charset.forName("ISO-8859-1"));
    int numChars = chars.length;

    List<Float> positions = new ArrayList();
    List<Float> textCoords = new ArrayList();
    float[] normals   = new float[0];
    List<Integer> indices   = new ArrayList();

    float tileWidth = (float)texture.getWidth() / (float)numCols;
    float tileHeight = (float)texture.getHeight() / (float)numRows;
```

The first lines of code create the data structures that will be used to store the positions, texture coordinates, normals and indices of the Mesh. In this case we will not apply lighting so the normals array will be empty. What we are going to do is construct a quad composed by a set of tiles, each of them representing a single character. We need to assign also the appropriate texture coordinates depending on the character code associated to each tile. The following picture shows the different elements that compose the tiles and the quad.



So, for each character we need to create a tile which is formed by two triangles which can be defined by using four vertices (V1, V2, V3 and V4). The indices will be (0, 1, 2) for the first triangle (the lower one) and (3, 0, 2) for the other triangle (the upper one). Texture coordinates are calculated based on the column and the row associated to each character in the texture image. Texture coordinates need to be in the range [0,1] so we just need to divide the current row or the current column by the total number of rows or columns to get the coordinate associated to V1. For the rest of vertices we just need to increase the current column or row by one in order to get the appropriate coordinate.

The following loop creates all the vertex position, texture coordinates and indices associated to the quad that contains the text.

www.dbooks.org

```
for(int i=0; i<numChars; i++) {
    byte currChar = chars[i];
    int col = currChar % numCols;
    int row = currChar / numCols;

    // Build a character tile composed by two triangles

    // Left Top vertex
    positions.add((float)i*tileWidth); // x
    positions.add(0.0f); //y
    positions.add(ZPOS); //z
    textCoords.add((float)col / (float)numCols );
    textCoords.add((float)row / (float)numRows );
    indices.add(i*VERTICES_PER_QUAD);

    // Left Bottom vertex
    positions.add((float)i*tileWidth); // x
    positions.add(tileHeight); //y
    positions.add(ZPOS); //z
    textCoords.add((float)col / (float)numCols );
    textCoords.add((float)(row + 1) / (float)numRows );
    indices.add(i*VERTICES_PER_QUAD + 1);

    // Right Bottom vertex
    positions.add((float)i*tileWidth + tileWidth); // x
    positions.add(tileHeight); //y
    positions.add(ZPOS); //z
    textCoords.add((float)(col + 1)/ (float)numCols );
    textCoords.add((float)(row + 1) / (float)numRows );
    indices.add(i*VERTICES_PER_QUAD + 2);

    // Right Top vertex
    positions.add((float)i*tileWidth + tileWidth); // x
    positions.add(0.0f); //y
    positions.add(ZPOS); //z
    textCoords.add((float)(col + 1)/ (float)numCols );
    textCoords.add((float)row / (float)numRows );
    indices.add(i*VERTICES_PER_QUAD + 3);

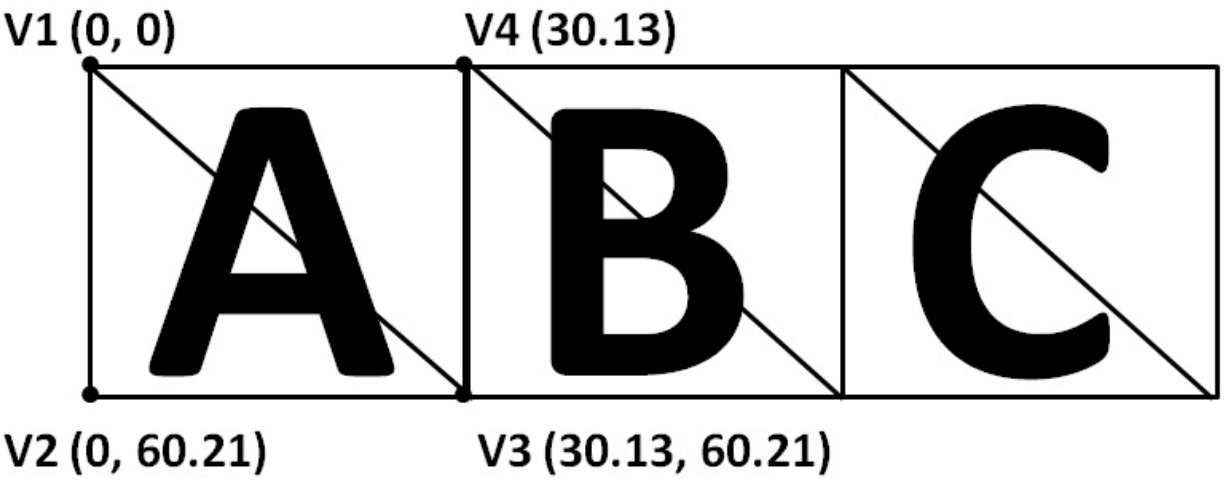
    // Add indices por left top and bottom right vertices
    indices.add(i*VERTICES_PER_QUAD);
    indices.add(i*VERTICES_PER_QUAD + 2);
}
```

The are some important things to notice in the previous fragment of code:

- We will represent the vertices using screen coordinates (remember that the origin of the screen coordinates is located at the top left corner). The y coordinate of the vertices on top of the triangles is lower than the y coordinate of the vertices on the bottom of the triangles.

- We don’t scale the shape, so each tile is at a x distance equal to a character width. The height of the triangles will be the height of each character. This is because we want to represent the text as similar as possible as the original texture. (Anyway we can later scale the result since `TextItem` class inherits from `GameItem`).
- We set a fixed value for the z coordinate, since it will be irrelevant in order to draw this object.

The next figure shows the coordinates of some vertices.



Why do we use screen coordinates ? First of all, because we will be rendering 2D objects in our HUD and often is more handy to use them, and secondly because we will use an orthographic projection in order to draw them. We will explain what is an orthographic projection later on.

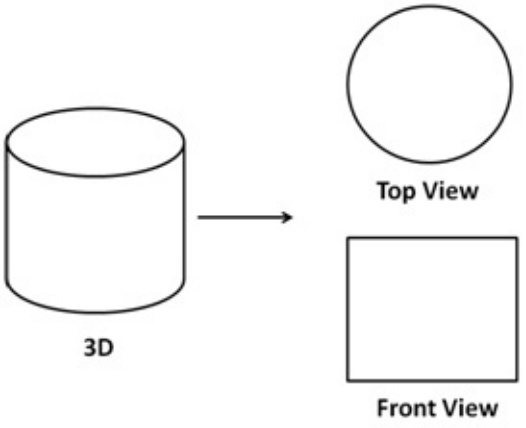
The `TextItem` class is completed with other methods to get the text and to change it at run time. Whenever the text is changed, we need to clean up the previous VAOs (stored in the `Mesh` instance) and create a new one. We do not need to destroy the texture, so we have created a new method in the `Mesh` class to just remove that data.

```
public String getText() {
    return text;
}

public void setText(String text) {
    this.text = text;
    Texture texture = this.getMesh().getMaterial().getTexture();
    this.getMesh().deleteBuffers();
    this.setMesh(buildMesh(texture, numCols, numRows));
}
```


www.dbooks.org

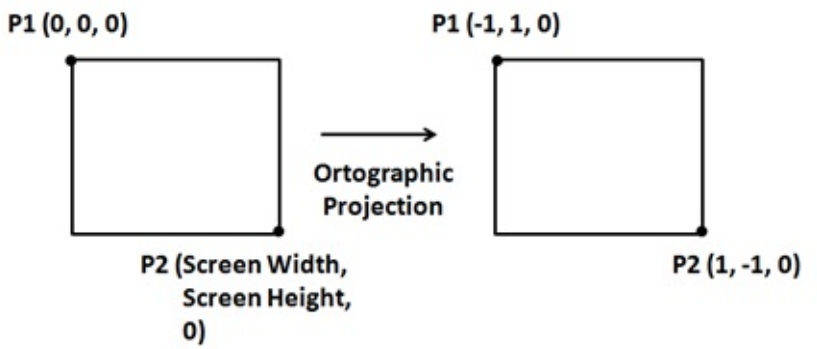
Now that we have set up the infrastucture needed to darw text, How do we do it? The basis is first to render the 3D scene, as in the previous chapters, and then render the 2D HUD over it. In order to render the HUD we will use an orthographic projection (also named orthogonal projection). An Orthographic projection is a 2D representation of a 3D object. You may already have seen some samples in blueprints of 3D objects which show the representation of those objects from the top or from some sides. The following picture shows the orthographic projection of a cylinder from the top and from the front.



This projection is very convenient in order to draw 2D objects because it "ignores" the values of the z coordinates, that is, the distance to the view. With this projection the objects sizes do not decrease with the distance (as in the perspective projection). In order to project an object using an ortographic projection we will need to use another matrix, the orthographic matrix which formula is shown below.

$$\begin{bmatrix} \frac{2}{right-left} & 0 & 0 & \frac{-(right+left)}{(right-left)} \\ 0 & \frac{1}{top-bottom} & 0 & \frac{-(top+bottom)}{(top-bottom)} \\ 0 & 0 & \frac{-2}{far-near} & \frac{-(far+near)}{(far-near)} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix also corrects the distortions that otherwise will be generated due to the fact that our window is not always a perfect square but a rectangle. The right and bottom parameters will be the screen size, the left and the top ones will be the origin. The orthographic projection matrix is used to transform screen coordinates to 3D space coordinates. The following picture shows how this mapping is done.



The properties of this matrix, will allow us to use screen coordinates.

We can now continue with thee implementation of the HUD. The next thing that we should do is create another set of shaders, a vertex and a fragment shaders, in order to draw the objects of the HUD. The vertex shader is actually very simple.

```
#version 330

layout (location=0) in vec3 position;
layout (location=1) in vec2 texCoord;
layout (location=2) in vec3 vertexNormal;

out vec2 outTexCoord;

uniform mat4 projModelMatrix;

void main()
{
    gl_Position = projModelMatrix * vec4(position, 1.0);
    outTexCoord = texCoord;
}
```

It will just receive the vertices positions, the texture coordinates, the indices and the normals and will transform them to the 3D space coordinates using a matrix. That matrix is the multiplication of the ortographic projection matrix and the model matrix, $projModelMatrix = ortographicMatrix \cdot modelMatrix$. Since we are not doing anything with the coordinates in model space, it's much more efficient to multiply both matrices in java code than in the shadere. By doing so we will be doing that multipliaztion once per item insted of doing it for each vertex. Remember that our vertices should be expressed in screen coordinates.

The fragment shader is also very simple.

www.dbooks.org

```
#version 330

in vec2 outTexCoord;
in vec3 mvPos;
out vec4 fragColor;

uniform sampler2D texture_sampler;
uniform vec4 colour;

void main()
{
    fragColor = colour * texture(texture_sampler, outTexCoord);
}
```

It just uses the texture coordinates and multiples that colour by a base colour. This can be used to change the colour of the text to be rendered without the need of creating several texture files. Now that we have created the new pair of shaders we can use them in the `Renderer` class. But, before that, we will create a new interface named `IHud` that will contain all the elements that are to be displayed in the HUD. This interface will also provide a default `cleanup` method.

```
package org.lwjglb.engine;

public interface IHud {

    GameItem[] getGameItems();

    default void cleanup() {
        GameItem[] gameItems = getGameItems();
        for (GameItem gameItem : gameItems) {
            gameItem.getMesh().cleanUp();
        }
    }
}
```

By using that interface our different games can define custom HUDs but the rendering mechanism does not need to be changed. Now we can get back to the `Renderer` class, which by the way has been moved to the engine graphics package because now it's generic enough to not be dependent on the specific implementation of each game. In the `Renderer` class we have added a new method to create, link and set up a new `ShaderProgram` that uses the shaders described above.

```
private void setupHudShader() throws Exception {
    hudShaderProgram = new ShaderProgram();
    hudShaderProgram.createVertexShader(Utils.loadResource("/shaders/hud_vertex.vs"));
    hudShaderProgram.createFragmentShader(Utils.loadResource("/shaders/hud_fragment.fs"
));
    hudShaderProgram.link();

    // Create uniforms for Ortographic-model projection matrix and base colour
    hudShaderProgram.createUniform("projModelMatrix");
    hudShaderProgram.createUniform("colour");
}
```

The `render` method first invokes the method `renderScene` which contains the code from previous chapter that rendered the 3D scene, and a new method, named `renderHud`, to render the HUD.

```
public void render(Window window, Camera camera, GameItem[] gameItems,
    SceneLight sceneLight, IHud hud) {

    clear();

    if ( window.isResized() ) {
        glViewport(0, 0, window.getWidth(), window.getHeight());
        window.setResized(false);
    }

    renderScene(window, camera, gameItems, sceneLight);

    renderHud(window, hud);
}
```

The `renderHud` method is as follows:

www.dbooks.org

```
private void renderHud(Window window, IHud hud) {

    hudShaderProgram.bind();

    Matrix4f ortho = transformation.getOrthoProjectionMatrix(0, window.getWidth(), win
dow.getHeight(), 0);
    for (GameItem gameItem : hud.getGameItems()) {
        Mesh mesh = gameItem.getMesh();
        // Set ortohtaphic and model matrix for this HUD item
        Matrix4f projModelMatrix = transformation.getOrtoProjModelMatrix(gameItem, ort
ho);
        hudShaderProgram.setUniform("projModelMatrix", projModelMatrix);
        hudShaderProgram.setUniform("colour", gameItem.getMesh().getMaterial().getAmbi
entColour());

        // Render the mesh for this HUD item
        mesh.render();
    }

    hudShaderProgram.unbind();
}
```

The previous fragment of code, iterates over the elements that compose the HUD and multiplies the orthographic projection matrix by the model matrix associated to each element. The orthographic projection matrix is updated in each `render` call (because the screen dimensions can change), and it's calculated in the following way:

```
public final Matrix4f getOrthoProjectionMatrix(float left, float right, float bottom,
float top) {
    orthoMatrix.identity();
    orthoMatrix.setOrtho2D(left, right, bottom, top);
    return orthoMatrix;
}
```

In our game package we will create a `Hud` class which implements the `IHud` interface and receives a text in the constructor creating internally a `TexItem` instance.

```
package org.lwjglb.game;

import org.joml.Vector4f;
import org.lwjglb.engine.GameItem;
import org.lwjglb.engine.IHud;
import org.lwjglb.engine.TextItem;

public class Hud implements IHud {

    private static final int FONT_COLS = 15;

    private static final int FONT_ROWS = 17;

    private static final String FONT_TEXTURE = "/textures/font_texture.png";

    private final GameItem[] gameItems;

    private final TextItem statusTextItem;

    public Hud(String statusText) throws Exception {
        this.statusTextItem = new TextItem(statusText, FONT_TEXTURE, FONT_COLS, FONT_R
OWS);
        this.statusTextItem.getMesh().getMaterial().setColour(new Vector4f(1, 1, 1, 1)
);
        gameItems = new GameItem[]{statusTextItem};
    }

    public void setStatusText(String statusText) {
        this.statusTextItem.setText(statusText);
    }

    @Override
    public GameItem[] getGameItems() {
        return gameItems;
    }

    public void updateSize(Window window) {
        this.statusTextItem.setPosition(10f, window.getHeight() - 50f, 0);
    }
}
```

In the `DummyGame` class we create an instance of that class an initialize it with a default text, and we will get something like this.

www.dbooks.org



In the `Texture` class we need to modify the way textures are interpolated to improve text readibility (you will only notice if you play with the text scaling).

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

But the sample is not fnished yet. If you play with the zoom so the text overlaps with the cube you will see this effect.



The text is not drawn with a transparent background. In order to achieve that, we must explicitly enable support for blending so the alpha component can be used. We will do this in the `Window` class when we set up the other initialization parameters with the following fragment of code.

```
// Support for transparencies
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Now you will see the text drawn with a transparent background.



Complete the HUD

Now that we have rendered a text we can add more elements to the HUD. We will add a compass that rotates depending on the direction the camera is facing. In this case, we will add a new GameItem to the Hud class that will have a mesh that models a compass.



The compass will be modeled by an .obj file but will not have a texture associated, instead it will have just a background colour. So we need to change the fragment shader for the HUD a little bit to detect if we have a texture or not. We will be able to do this by setting a new uniform named `hasTexture`.

www.dbooks.org

```
#version 330

in vec2 outTexCoord;
in vec3 mvPos;
out vec4 fragColor;

uniform sampler2D texture_sampler;
uniform vec4 colour;
uniform int hasTexture;

void main()
{
    if ( hasTexture == 1 )
    {
        fragColor = colour * texture(texture_sampler, outTexCoord);
    }
    else
    {
        fragColor = colour;
    }
}
```

To add the compass the the HUD we just need to create a new `GameItem` instance, tn the `Hud` class, that loads the compass model and adds it to the list of items. In this case we will need to scale up the compass. Remember that it needs to be expressed in screen coordinates, so usually you will need to increase its size.

```
// Create compass
Mesh mesh = OBJLoader.loadMesh("/models/compass.obj");
Material material = new Material();
material.setAmbientColour(new Vector4f(1, 0, 0, 1));
mesh.setMaterial(material);
compassItem = new GameItem(mesh);
compassItem.setScale(40.0f);
// Rotate to transform it to screen coordinates
compassItem.setRotation(0f, 0f, 180f);

// Create list that holds the items that compose the HUD
gameItems = new GameItem[]{statusTextItem, compassItem};
```

Notice also that, in order for the compass to point upwards we need to rotate 180 degrees since the model will often tend to use OpenGL space coordinates. If we are expecting screen coordinates it would pointing downwards. The `Hud` class will also provide a method to update the angle of the compass that must take this also into consideration.

```
public void rotateCompass(float angle) {
    this.compassItem.setRotation(0, 0, 180 + angle);
}
```

In the `DummyGame` class we will update the angle whenever the camera is moved. We need to use the y angle rotation.

```
// Update camera based on mouse
if (mouseInput.isRightButtonPressed()) {
    Vector2f rotVec = mouseInput.getDisplVec();
    camera.moveRotation(rotVec.x * MOUSE_SENSITIVITY, rotVec.y * MOUSE_SENSITIVITY, 0)
;

    // Update HUD compass
    hud.rotateCompass(camera.getRotation().y);
}
```

We will get something like this (remember that it is only a sample, in a real game you may probably want to use some texture to give the compass a different look).



Text rendering revisited

Before reviewing other topics let's go back to the text rendering approach we have presented here. The solution is very simple and handy to introduce the concepts involved in rendering HUD elements but it presents some problems:

- It does not support non latin character sets.
- If you want to use several fonts you need to create a separate texture file for each font. Also, the only way to change the text size is either to scale it, which may result in a poor quality rendered text, or to generate another texture file.

www.dbooks.org

- The most important one, characters in most of the fonts do not occupy the same size but we are dividing the font texture in equally sized elements. We have cleverly used “Consolas” font which is monospaced (that is, all the characters occupy the same amount of horizontal space), but if you use a non-monospaced font you will see annoying variable white spaces between the characters.

We need to change our approach an provide a more flexible way to render text. If you think about it, the overall mechanism is ok, that is, the way of rendering text by texturing quads for each character. The issue here is how we are generating the textures. We need to be able to generate those texture dynamically by using the fonts available in the System.

This is where `java.awt.Font` comes to the rescue, we will generate the textures by drawing each letter for a specified font family and size dynamically. That texture will be used in the same way as described previously, but it will solve perfectly all the issues mentioned above. We will create a new class named `FontTexture` that will receive a Font instance and a charset name and will dynamically create a texture that contains all the available characters. This is the constructor.

```
public FontTexture(Font font, String charSetName) throws Exception {
    this.font = font;
    this.charSetName = charSetName;
    charMap = new HashMap<>();

    buildTexture();
}
```

The first step is to handle the non latin issue, given a char set and a font we will build a `String` that contains all the characters that can be rendered.

```
private String getAllAvailableChars(String charsetName) {
    CharsetEncoder ce = Charset.forName(charsetName).newEncoder();
    StringBuilder result = new StringBuilder();
    for (char c = 0; c < Character.MAX_VALUE; c++) {
        if (ce.canEncode(c)) {
            result.append(c);
        }
    }
    return result.toString();
}
```

Let’s now review the method that actually creates the texture, named `buildTexture`.

```
private void buildTexture() throws Exception {
    // Get the font metrics for each character for the selected font by using image
    BufferedImage img = new BufferedImage(1, 1, BufferedImage.TYPE_INT_ARGB);
    Graphics2D g2D = img.createGraphics();
    g2D.setFont(font);
    FontMetrics fontMetrics = g2D.getFontMetrics();

    String allChars = getAllAvailableChars(charSetName);
    this.width = 0;
    this.height = 0;
    for (char c : allChars.toCharArray()) {
        // Get the size for each character and update global image size
        CharInfo charInfo = new CharInfo(width, fontMetrics.charWidth(c));
        charMap.put(c, charInfo);
        width += charInfo.getWidth();
        height = Math.max(height, fontMetrics.getHeight());
    }
    g2D.dispose();
```

We first obtain the font metrics by creating a temporary image. Then we iterate over the `String` that contains all the available characters and get the width, with the help of the font metrics, of each of them. We store that information on a map, `charMap`, which will use as a key the character. With that process we determine the size of the image that will have the texture (with a height equal to the maximum size of all the characters and its with equal to the sum of each character width). `CharSet` is an inner class that holds the information about a character (its width and where it starts, in the x coordinate, in the texture image).

```
    public static class CharInfo {

        private final int startX;

        private final int width;

        public CharInfo(int startX, int width) {
            this.startX = startX;
            this.width = width;
        }

        public int getStartX() {
            return startX;
        }

        public int getWidth() {
            return width;
        }
    }
```


www.dbooks.org

Then we will create an image that will contain all the available characters. In order to do this, we just draw the string over a `BufferedImage`.

```
    // Create the image associated to the charset
    img = new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);
    g2D = img.createGraphics();
    g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALI
AS_ON);
    g2D.setFont(font);
    fontMetrics = g2D.getFontMetrics();
    g2D.setColor(Color.WHITE);
    g2D.drawString(allChars, 0, fontMetrics.getAscent());
    g2D.dispose();
```

We are generating an image which contains all the characters in a single row (we maybe are not fulfilling the premise that the texture should have a size of a power of two, but it should work on most modern cards. In any caseyou could always achieve that by adding some extra empty space). You can even see the image that we are generating, if after that block of code, you put a line like this:

```
ImageIO.write(img, IMAGE_FORMAT, new java.io.File("Temp.png"));
```

The image will be written to a temporary file. That file will contain a long strip with all the available characters, drawn in white over transparent background using anti aliasing.



Finally, we just need to create a `Texture` instance from that image, we just dump the image bytes using a PNG format (which is what the `Texture` class expects).

```
    // Dump image to a byte buffer
    InputStream is;
    try (
        ByteArrayOutputStream out = new ByteArrayOutputStream()) {
        ImageIO.write(img, IMAGE_FORMAT, out);
        out.flush();
        is = new ByteArrayInputStream(out.toByteArray());
    }

    texture = new Texture(is);
}
```

You may notice that we have modified a little bit the `Texture` class to have another constructor that receives an `InputStream`. Now we just need to change the `TextItem` class to receive a `FontTexture` instance in its constructor.

```
public TextItem(String text, FontTexture fontTexture) throws Exception {
    super();
    this.text = text;
    this.fontTexture = fontTexture;
    setMesh(buildMesh());
}
```

The `buildMesh` method only needs to be changed a little bit when setting quad and texture coordinates, this is a sample for one of the vertices.

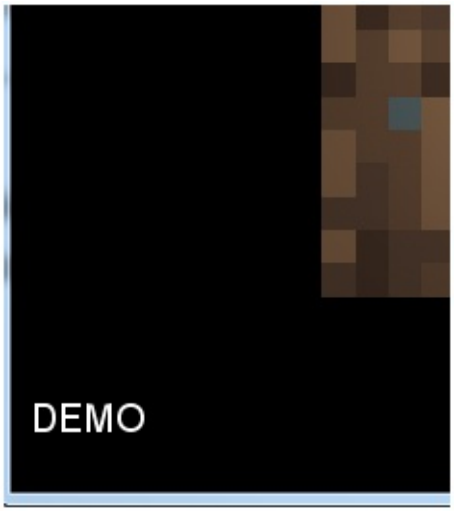
```
    float startx = 0;
    for(int i=0; i<numChars; i++) {
        FontTexture.CharInfo charInfo = fontTexture.getCharInfo(characters[i]);

        // Build a character tile composed by two triangles

        // Left Top vertex
        positions.add(startx); // x
        positions.add(0.0f); //y
        positions.add(ZPOS); //z
        textCoords.add( (float)charInfo.getStartX() / (float)fontTexture.getWidth());
        textCoords.add(0.0f);
        indices.add(i*VERTICES_PER_QUAD);

      // .. More code
      startx += charInfo.getWidth();
    }
```

You can check the rest of the changes directly in the source code. The following picture shows what you will get for an Arial font with a size of 20:



www.dbooks.org

As you can see the quality of the rendered text has been improved a lot, you can play with different fonts and sizes and check it by your own. There's still plenty of room for improvement (like supporting multiline texts, effects, etc.), but this will be left as an exercise for the reader.

You may also notice that we are still able to apply scaling to the text (we pass a model view matrix in the shader). This may not be needed now for text but it may be useful for other HUD elements.

We have set up all the infrastructure needed in order to create a HUD for our games. Now it is just a matter of creating all the elements that represent relevant information to the user and give them a professional look and feel.

OSX

If you try to run the samples in this chapter, and the next ones that render text, you may find that the application blocks and nothing is shown in the screen. This is due to the fact that AWT and GLFW do get along very well under OSX. But, what does it have to do with AWT ? We are using the `Font` class, which belongs to AWT, and just by instantiating it, AWT gets initialized also. In OSX AWT tries to run under the main thread, which is also required by GLFW. This is what causes this mess.

In order to be able to use the `Font` class, GLFW must be initialized before AWT and the samples need to be run in headless mode. You need to setup this property before anything gets intialized:

```
System.setProperty("java.awt.headless", "true");
```

You may get a warning, but the samples will run.

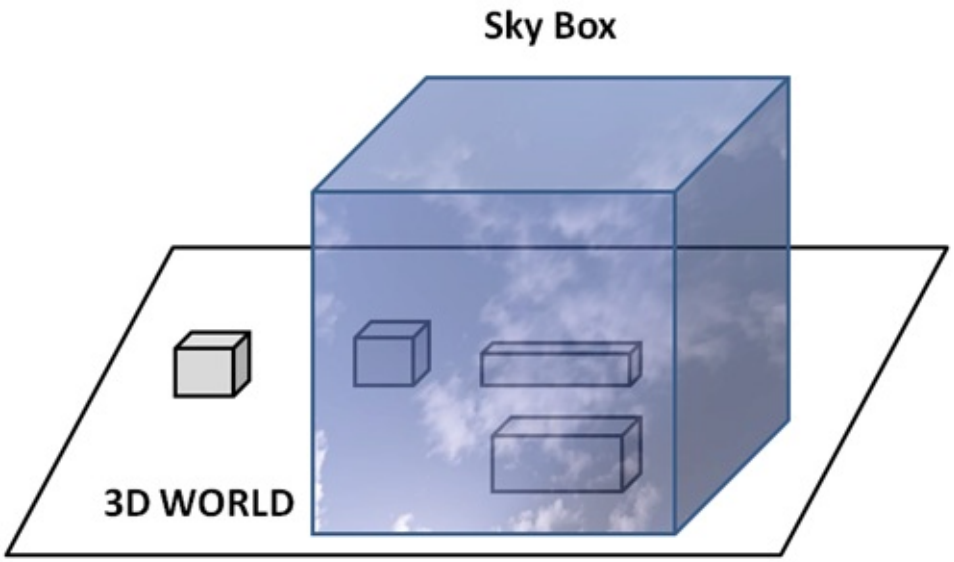
A much more clean approach would be to use the stb library to render text.

Sky Box and some optimizations

Skybox

A skybox will allow us to set a background to give the illusion that our 3D world is bigger. That background is wrapped around the camera position and covers the whole space. The technique that we are going to use here is to construct a big cube that will be displayed around the 3D scene, that is, the centre of the camera position will be the centre of the cube. The sides of that cube will be wrapped with a texture with hills a blue sky and clouds that will be mapped in a way that the image looks a continuous landscape.

The following picture depicts the skybox concept.



The process of creating a sky box can be summarized in the following steps:

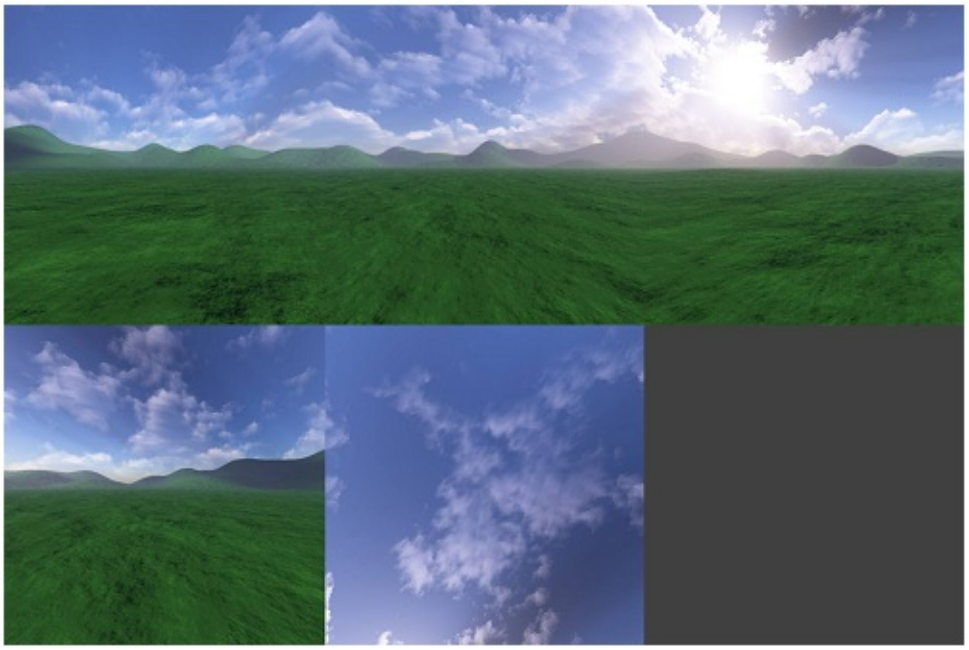
- Create a big cube.
- Apply a texture to it that provides the illusion that we are seeing a giant landscape with no edges.
- Render the cube so its sides are at a far distance and its origin is located at the centre of the camera.

Then, let's start with the texture. You will find that there are lots of pre-generated textures for you to use in the internet. The one used in the sample for this chapter has been downloaded from here: http://www.custommapmakers.org/skyboxes.php. The concrete sample that we

www.dbooks.org

have used is this one: http://www.custommapmakers.org/skyboxes/zips/ely_hills.zip and has been created by Colin Lowndes.

The textures from that site are composed by separate TGA files, one for each side of the cube. The texture loader that we have created expects a single file in PNG format so we need to compose a single PNG image with the images of each face. We could apply other techniques, such us cube mapping, in order to apply the textures automatically. But, in order to keep this chapter as simple as possible, you will have to manuallay arrange them into a single file. The result image will look like this.



After that, we need to create a .obj file which contains a cube with the correct texture coordinates for each face. The picture below shows the tiles associated to each face (you can find the .obj file used in this chapter in the book's source code).



Once the resoures have been set up, we can start coding. We will start by creating a new class named `SkyBox` with a constructor that receives the path to the OBJ model that contains the sky box cube and the texture file. This class will inherit from `GameItem` as the HUD class from the previous chapter. Why it should inherit from `GameItem` ? First of all, for convenience, we can reuse most of the code that deals with meshes and textures. Secondly, because, although the skybox will not move we will be interested in applying rotations and scaling to it. If you think about it a `SkyBox` is indeed a game item. The definition of the `SkyBox` class is as follows.

```
package org.lwjglb.engine;

import org.lwjglb.engine.graph.Material;
import org.lwjglb.engine.graph.Mesh;
import org.lwjglb.engine.graph.OBJLoader;
import org.lwjglb.engine.graph.Texture;

public class SkyBox extends GameItem {

    public SkyBox(String objModel, String textureFile) throws Exception {
        super();
        Mesh skyBoxMesh = OBJLoader.loadMesh(objModel);
        Texture skyBoxtexture = new Texture(textureFile);
        skyBoxMesh.setMaterial(new Material(skyBoxtexture, 0.0f));
        setMesh(skyBoxMesh);
        setPosition(0, 0, 0);
    }
}
```

If you check the source code for this chapter you will see that we have done some refactoring. We have created a class named `Scene` which groups all the information related to the 3D world. This is the definition and the attributes of the `Scene` class, that contains an instance of the `SkyBox` class.

www.dbooks.org

```
package org.lwjglb.engine;

public class Scene {

    private GameItem[] gameItems;

    private SkyBox skyBox;

    private SceneLight sceneLight;

    public GameItem[] getGameItems() {
        return gameItems;
    }

    // More code here...
```

The next step is to create another set of vertex and fragment shaders for the skybox. But, why not reuse the scene shaders that we already have? The answer is that, actually, the shaders that we will need are a simplified version of those shaders, we will not be applying lights to the sky box (or to be more precise, we won't need point, spot or directional lights). Below you can see the skybox vertex shader.

```
#version 330

layout (location=0) in vec3 position;
layout (location=1) in vec2 texCoord;
layout (location=2) in vec3 vertexNormal;

out vec2 outTexCoord;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;

void main()
{
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
    outTexCoord = texCoord;
}
```

You can see that we still use the model view matrix. At is has been explained beore we will scale the skybox, so we need that transformation matrix. You may see some other implementations that increase the size of the cube that models the sky box at start time and do not need to multiply the model and the view matrix. We have chosen this approach because it's more flexible and it allows us to change the size of the skybox at runtime, but you can easily switch to the other approach if you want.

The fragment shader is also very simple.

```
#version 330

in vec2 outTexCoord;
in vec3 mvPos;
out vec4 fragColor;

uniform sampler2D texture_sampler;
uniform vec3 ambientLight;

void main()
{
    fragColor = vec4(ambientLight, 1) * texture(texture_sampler, outTexCoord);
}
```

As you can see, we added an ambient light uniform to the shader. The purpose of this uniform is to modify the colour of the skybox texture to simulate day and night (If not, the skybox will look like if it was midday when the rest of the world is dark).

In the `Renderer` class we just have added a new method to use those shaders and setup the uniforms (nothing new here).

```
private void setupSkyBoxShader() throws Exception {
    skyBoxShaderProgram = new ShaderProgram();
    skyBoxShaderProgram.createVertexShader(Utils.loadResource("/shaders/sb_vertex.vs")
);
    skyBoxShaderProgram.createFragmentShader(Utils.loadResource("/shaders/sb_fragment.
fs"));
    skyBoxShaderProgram.link();

    skyBoxShaderProgram.createUniform("projectionMatrix");
    skyBoxShaderProgram.createUniform("modelViewMatrix");
    skyBoxShaderProgram.createUniform("texture_sampler");
    skyBoxShaderProgram.createUniform("ambientLight");
}
```

And of course, we need to create a new render method for the skybox that will be invoked in the global render method.

www.dbooks.org

```
private void renderSkyBox(Window window, Camera camera, Scene scene) {
    skyBoxShaderProgram.bind();

    skyBoxShaderProgram.setUniform("texture_sampler", 0);

    // Update projection Matrix
    Matrix4f projectionMatrix = transformation.getProjectionMatrix(FOV, window.getWidt
h(), window.getHeight(), Z_NEAR, Z_FAR);
    skyBoxShaderProgram.setUniform("projectionMatrix", projectionMatrix);
    SkyBox skyBox = scene.getSkyBox();
    Matrix4f viewMatrix = transformation.getViewMatrix(camera);
    viewMatrix.m30(0);
    viewMatrix.m31(0);
    viewMatrix.m32(0);
    Matrix4f modelViewMatrix = transformation.getModelViewMatrix(skyBox, viewMatrix);
    skyBoxShaderProgram.setUniform("modelViewMatrix", modelViewMatrix);
    skyBoxShaderProgram.setUniform("ambientLight", scene.getSceneLight().getAmbientLig
ht());

    scene.getSkyBox().getMesh().render();

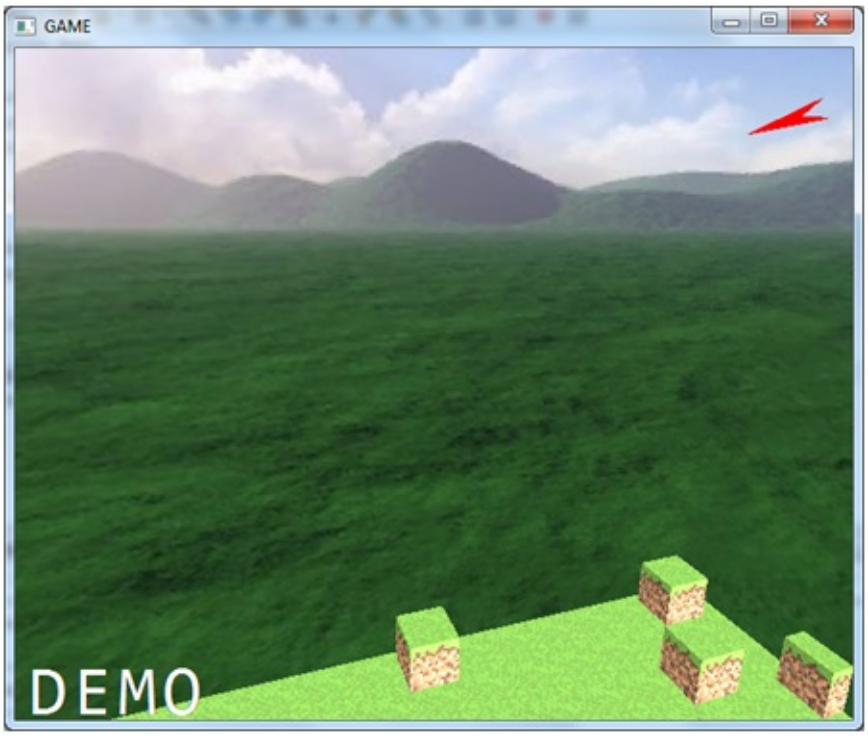
    skyBoxShaderProgram.unbind();
}
```

The method above is quite similar to the other render ones but there's a difference that needs to be explained. As you can see, we pass the projection matrix and the model view matrix as usual. But, when we get the view matrix, we set some of the components to 0. Why we do this ? The reason behind that is that we do not want translation to be applied to the sky box.

Remember that when we move the camera, what we are actually doing is moving the whole world. So if we just multiply the view matrix as it is, the skybox will be displaced when the camera movees. But we do not want this, we want to stick it at the origin coordinates at (0, 0, 0). This is achieved by setting to 0 the parts of the view matrix that contain the translation increments (the `m30`, `m31` and `m32` components).

You may think that you could avoid using the view matrix at all since the sky box must be fixed at the origin. In that case what, you will see is that the skybox will not rotate with the camera, which is not what we want. We need it to rotate but not translate.

And that's all, you can check in the source code for this chapter that in the `DummyGame` class that we have created more block instances to simulate a ground and the skybox. You can also check that we now change the ambient light to simulate light and day. What you will get is something like this.

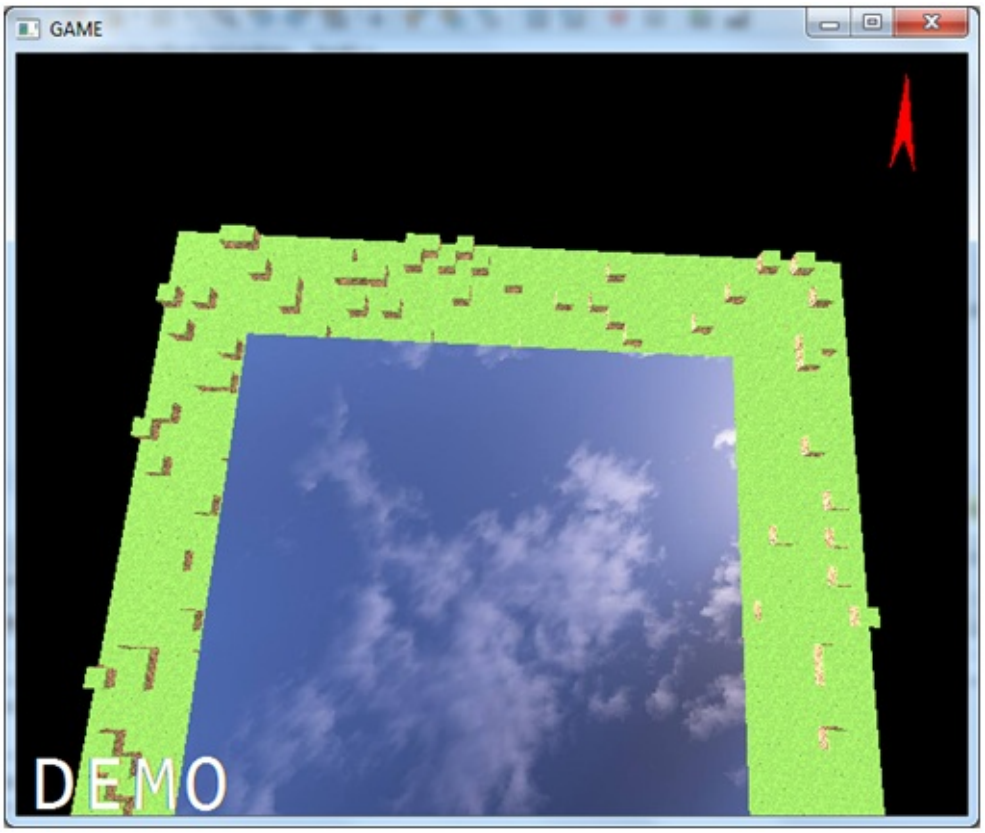


The sky box is a small one so you can easily see the effect of moving through the world (in a real game it should be much bigger). You can see also that the world space objects, the blocks that form the terrain are larger than the skybox, so as you move through it you will see blocks appearing through the mountains. This is more evident because of the relative small size of the sky box we have set. But anyway we will need to alleviate that by adding an effect that hides or blur distant objects (for instance applying a fog effect).

Another reason for not creating a bigger sky box is because we need to apply several optimizations in order to be more efficient (they will be explained later on).

You can play with the render method an comment the lines that prevent the skybox from translating. Then you will be able to get out of the box and see something like this.

www.dbooks.org



Although it is not what a skybox should do it can help you out to understand the concept behind this tecnique. Remember that this is a simple example, you could improve it by adding other effects such as the sun moving through the sky or moving clouds. Also, in order to create bigger worlds you will need to split the world into fragments and only load the ones that are contiguous to the fragment where the player is currently in.

Another point that is worth to mention is when should we render the sky box, before the scene or after? Rendering after the scene has been drawn is more optimal, since most of the fragments will be discarded due to depth testing. That is, non visible skybox fragments, the ones, that will be hidden by scene elements will be discarded. When OpenGL will try to render them, and depth test is enabled, it will discard the ones which are behind some previously rendered fragments, which will have a lower depth value. So the answer might be obvious, right? Just render the skybox after the scene has been rendered. The problem with this approach is handling transparent textures. If we have, in the scene, objects with transparent textures, they will be drawn using the "background" colour, which is now black. If we render the skybox before, the transparent effect will be applied correctly. So, shall we render it before the scene then? Well, yes and no. If you render before the scene is rendered you will solve transparency issues but you will impact performance. In fact, you still may face transparency issues without a sky box. For instance, let's say that you have a transparent item, which overlaps with an object that is far away. If the transparent object is rendered first, you will face also transperent issues. So, maybe another approach can be to

draw transparent items, seperately, after all the other items have been rendered. This is the approach used by some commercial games. So, by now, we will render the skybox after the scene has been rendered, trying to get better performance.

Some optimizations

From the previous example, the fact that the skybox is relative small makes the effect a little bit weird (you can see objects appearing magically behind the hills). So, ok, let's increase the skybox size and the size of our world. Let's scale the size of the skybox by a factor of 50 so the world will be composed by 40,000 GameItem instances (cubes).

If you change the scale factor and rerun the example you will see that performance problem starts to arise and the movement through the 3D world is not smooth. It's time to put an eye on performance (you may know the old saying that states that "premature optimization is the root of all evil", but since this chapter 13, I hope nobody will say that this premature).

Let's start with a concept that will reduce the amount of data that is being rendered, which is named face culling. In our examples we are rendering thousands of cubes, and a cube is made of six faces. We are rendering the six faces for each cube even if they are not visible. You can check this if you zoom inside a cube, you will see its interior like this.



Faces that cannot be seen should be discarded immediately and this is what face culling does. In fact, for a cube you can only see 3 faces at the same time, so we can just discard half of the faces (40,000 *3* 2 triangles) just by applying face culling (this will only be valid if your game does not require you to dive into the inner side of a model, you can see why later on).

Face culling checks, for every triangle if its facing towards us and discards the ones that are not facing that direction. But, how do we know if a triangle is facing towards us or not ? Well, the way that OpenGL does this is by the winding order of the vertices that compose a triangle.

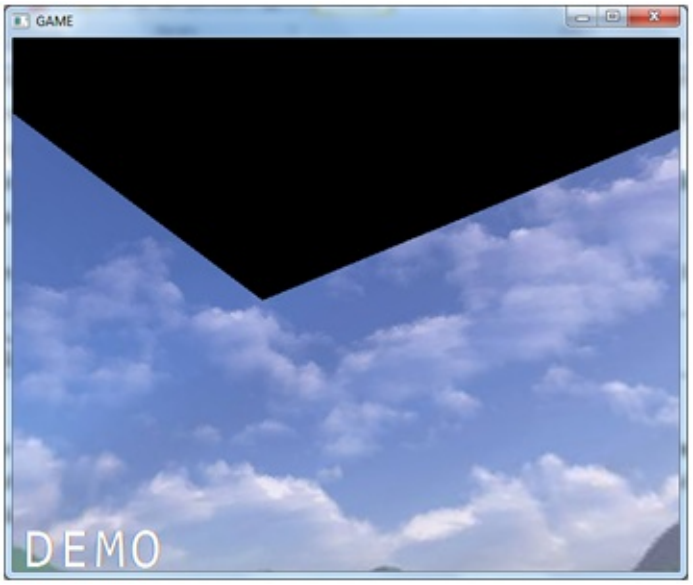
www.dbooks.org

Remember from the first chapters that we may define the vertices of a triangle in clockwise or counter-clockwise order. In OpenGL, by default, triangles that are in counter-clockwise order are facing towards the viewer and triangles that are in clockwise order are facing backwards. The key thing here, is that this order is checked while rendering taking into consideration the point of view. So a triangle that has been defined in counter-clock wise order can be interpreted, at rendering time, as being defined lockwise because of the point of view.

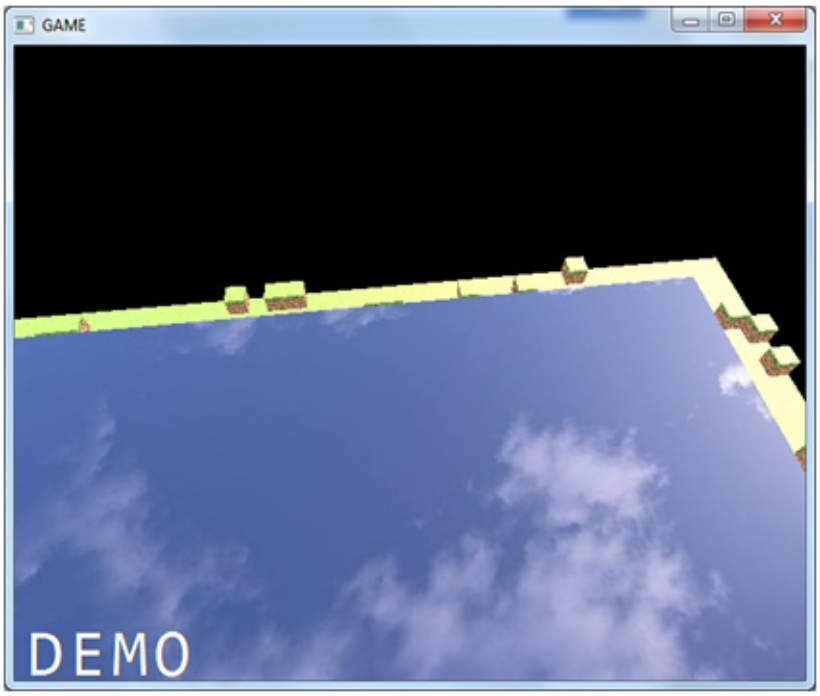
Let's put it in practice, in the `init` method of the `Window` class add the following lines:

```
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
```

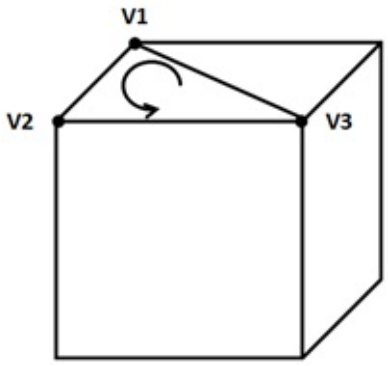
The first line will enable face culling and the second line states that faces that are facing backwards should be culled (removed). With that line if you look upwards you will see something like this.



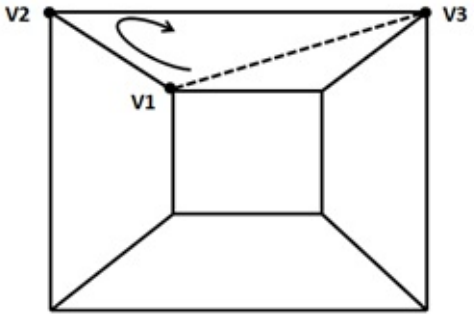
What's happening ? if you review the vertices order for the top face you will see that is has been defined in counter-clockwise order. Well, it was, but remember that the winding refers to the point of view. In fact, if you apply translation also to the skybox so you are able to see it form the upside you will see that the top face is rendered again once you are outside it.



Let's sketch what's happening. The following picture shows one of the triangles of the top face of the skybox cube, which is defined by three vertices defined in counter-clockwise order.



But remember that we are inside the skybox, if we look at the cube form the interior, what we will see is that the vertices are defined in clockwise order.



This is because, the skybox was defined to be looked from the outside. So we need to flip the definition of some of the faces in order to be viewed correctly when face culling is enabled.

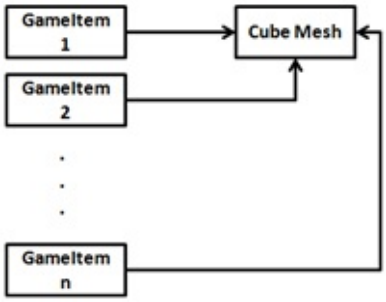
www.dbooks.org

But there's still more room for optimization. Let's review our rendering process. In the `render` method of the `Renderer` class what we are doing is iterate over a `GameItem` array and render the associated `Mesh`. For each `GameItem` we do the following:

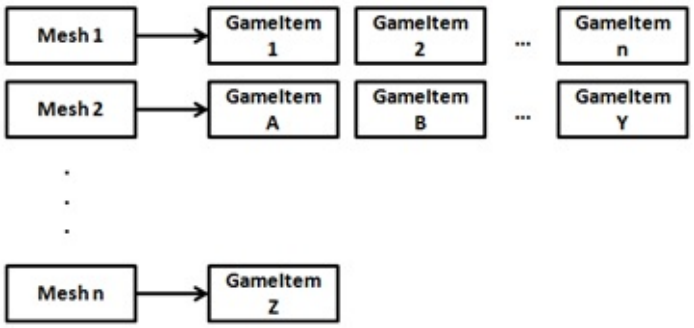
1. Set up the model view matrix (unique per `GameItem`).
2. Get the `Mesh` associated to the `GameItem` and activate the texture, bind the VAO and enable its attributes.
3. Perform a call to draw the triangles.
4. Disable the texture and the VAO elements.

But, in our current game, we reuse the same `Mesh` for the 40,000 GameItems, we are repeating the operations from point 2 to point 4 again and again. This is not very efficient, keep in mind that each call to an OpenGL function is a native call that incurs in some performance overhead. Besides that, we should always try to limit the state changes in OpenGL (activating and deactivating textures, VAOs are state changes).

We need to change the way we do things and organize our structures around Meshes since it will be very frequent to have many GameItems with the same Mesh. Now we have an array of GameItems each of them pointing to the same Mesh. We have something like this.



Instead, we will create a Map of Meshes with a list of the GamItems that share that Mesh.



The rendering steps will be, for each `Mesh`:

1. Set up the model view matrix (unique per `GameItem`).
2. Get the `Mesh` associated to the `GameItem` and Activate the `Mesh` texture, bind the VAO and enable its attributes.
3. For each `GameItem` associated: a. Set up the model view matrix (unique per Game Item). b. Perform a call to draw the triangles.

4. Disable the texture and the VAO elements.

In the Scene class, we will store the following `Map`.

```
private Map<Mesh, List<GameItem>> meshMap;
```

We still have the `setGameItems` method, but instead of just storing the array, we construct the mesh map.

```
public void setGameItems(GameItem[] gameItems) {
    int numGameItems = gameItems != null ? gameItems.length : 0;
    for (int i=0; i<numGameItems; i++) {
        GameItem gameItem = gameItems[i];
        Mesh mesh = gameItem.getMesh();
        List<GameItem> list = meshMap.get(mesh);
        if ( list == null ) {
            list = new ArrayList<>();
            meshMap.put(mesh, list);
        }
        list.add(gameItem);
    }
}
```

The `Mesh` class now has a method to render a list of the associated GameItems and we have split the activating and deactivating code into separate methods.

www.dbooks.org

```
private void initRender() {
    Texture texture = material.getTexture();
    if (texture != null) {
        // Activate firs texture bank
        glActiveTexture(GL_TEXTURE0);
        // Bind the texture
        glBindTexture(GL_TEXTURE_2D, texture.getId());
    }

    // Draw the mesh
    glBindVertexArray(getVaoId());
    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);
    glEnableVertexAttribArray(2);
}

private void endRender() {
    // Restore state
    glDisableVertexAttribArray(0);
    glDisableVertexAttribArray(1);
    glDisableVertexAttribArray(2);
    glBindVertexArray(0);

    glBindTexture(GL_TEXTURE_2D, 0);
}

public void render() {
    initRender();

    glDrawElements(GL_TRIANGLES, getVertexCount(), GL_UNSIGNED_INT, 0);

    endRender();
}

public void renderList(List<GameItem> gameItems, Consumer<GameItem> consumer) {
    initRender();

    for (GameItem gameItem : gameItems) {
        // Set up data requiered by gameItem
        consumer.accept(gameItem);
        // Render this game item
        glDrawElements(GL_TRIANGLES, getVertexCount(), GL_UNSIGNED_INT, 0);
    }

    endRender();
}
```

As you can see we still have the old method that renders the a `Mesh` taking into consideration that we have only one GameItem (this may be used in other cases, this is why it has not been removed). The new method renders a list of GameItems and receives as a

parameter a `Consumer` (a function, that uses the new functional programming paradigms introduced in Java 8), which will be used to setup what's specific for each GameItem before drawing the triangles. We will use this to set up the model view matrix, since we do not want the `Mesh` class to be coupled with the uniforms names and the parameters involved when setting these things up.

In the `renderScene` method of the `Renderer` class you can see that we just iterate over the Mesh map and setup the model view matrix uniform via a lambda.

```
for (Mesh mesh : mapMeshes.keySet()) {
    sceneShaderProgram.setUniform("material", mesh.getMaterial());
    mesh.renderList(mapMeshes.get(mesh), (GameItem gameItem) -> {
        Matrix4f modelViewMatrix = transformation.buildModelViewMatrix(gameItem, viewM
atrix);
        sceneShaderProgram.setUniform("modelViewMatrix", modelViewMatrix);
    }
    );
}
```

Another set of optimizations that we can do is that we are creating tons of objects in the render cycle. In particular, we were creating too many `Matrix4f` instances that holds a copy a the model view matrix for each `GameItem` instance. We will create specific matrices for that in the Transformation class, and reuse the same instance. If you check the code you will see also that we have changed the names of the methods, the `getXX` methods just return the store matrix instance and any method that changes the value of a matrix is called `buildXX` to clarify its purpose.

We have also avoided the construction of new `FloatBuffer` instances each time we set a uniform for a Matrix and removed some other useless instantiations. With all that in place you can see now that the rendering is smoother and more agile.

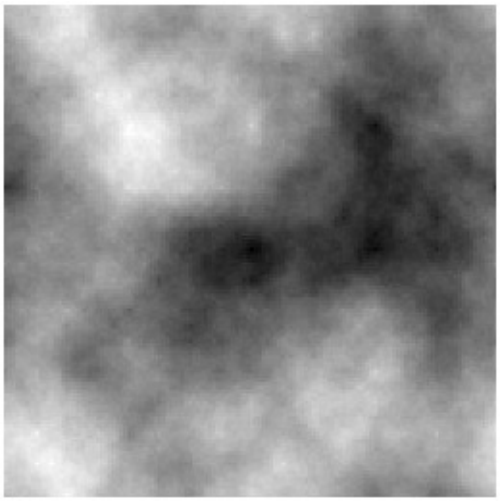
You can check all the details in the source code.

www.dbooks.org

Height Maps

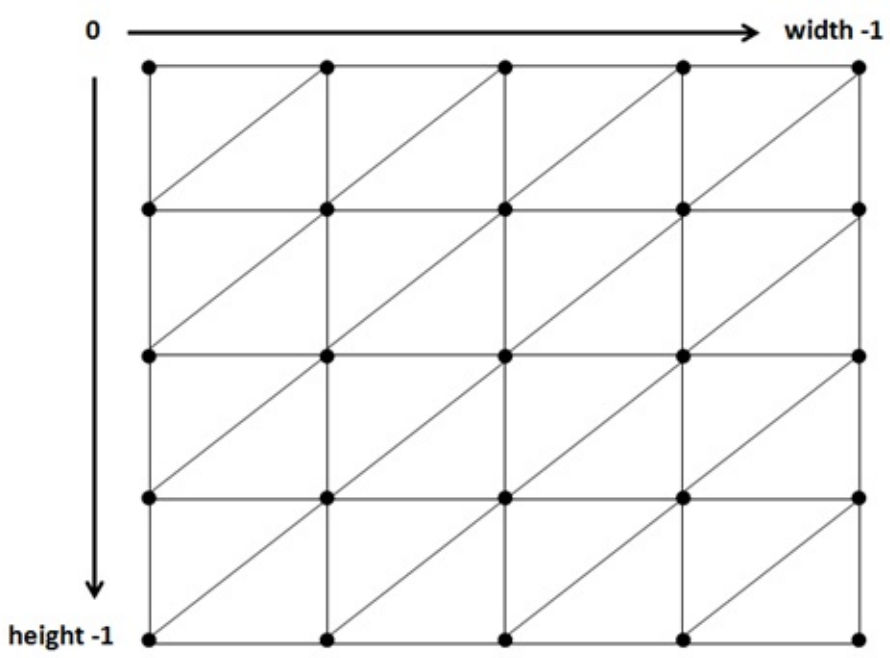
In this chapter we will learn how to create complex terrains using height maps. Before we start, you will notice that some refactoring has been done. We have created some new packages and moved some of the classes to better organize them. You can check the changes in the source code.

So what's a height map? A height map is an image which is used to generate a 3D terrain which uses the pixel colours to get surface elevation data. Height maps images use usually gray scale and can be generated by programs like Terragen. A height map image looks like this.

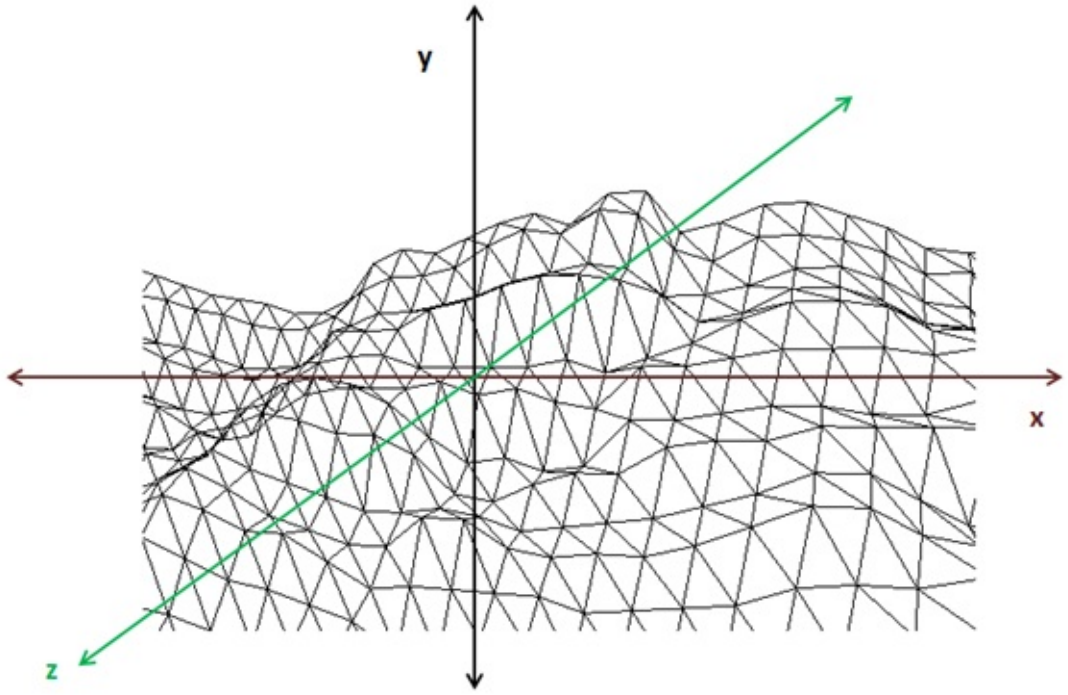


The image above it's like if you were looking at a fragment of land from above. With that image we will build a mesh composed by triangles formed by vertices. The altitude of each vertex will be calculated depending on the colour of each of the image pixels. Black colour will represent the lowest value and white the highest one.

We will be creating a grid of vertices, one for each pixel of the image. Those vertices will be used to form triangles that will compose the mesh as shown in the next figure.



That mesh will form a giant quad that will be rendered across x and z axis using thepixel colours to change the elevation in the y axis.



The process of creating a 3D terrain from a height map can be summarized as follows:

- Load the image that contains the height map. (We will use a `BufferedImage` instance to get access to each pixel).
- For each image pixel create a vertex with its height is based on the pixel colour.
- Assign the correct texture coordinate to the vertex.
- Set up the indices to draw the triangles associated to the vertex.

www.dbooks.org

We will create a class named `HeightMapMesh` that will create a `Mesh` based on a height map image performing the steps described above. Let's first review the constants defined for that class:

```
private static final int MAX_COLOUR = 255 * 255 * 255;
```

As we have explained above, we will calculate the height of each vertex based on the colour of each pixel of the image used as height map. Images are usually greyscale, for a PNG image that means that each RGB component for each pixel can vary from 0 to 255, so we have 256 discrete values to define different heights. This may be enough precision for you, but if it's not we can use the three RGB components to have more intermediate values, in this case the height can be calculated form a range that gets from 0 to 255^3. We will choose the second approach so we are not limited to greyscale images.

The next constants are:

```
private static final float STARTX = -0.5f;

private static final float STARTZ = -0.5f;
```

The mesh will be formed by a set of vertices (one per pixel) whose x and z coordinates will be in the range following range:

- [-0.5, 0.5], that is, [`STARTX` , `-STARTX`] for the x axis.
- [-0.5, 0.5], that is, [`STARTZ` , `-STARTZ`] for the z axis.

Don't worry too much about those values, later on the resulting mesh can be scaled to accommodate its size in the world. Regarding y axis, we will set up two parameters, `minY` and `maxY` , for setting the lowest and highest value that the y coordinate can have. These parameters are not constant because we may want to change them at run time, independently of the scaling applied. At the end, the terrain will be contained in a cube in the range `[STARTX, -STARTX]` , `[minY, maxY]` and `[STARTZ, -STARTZ]` .

The mesh will be created in the constructor of the `HeightMapMesh` class, which is defined like this.

```
public HeightMapMesh(float minY, float maxY, String heightMapFile, String textureFile,
int textInc) throws Exception {
```

It receives the minimum and maximum vale for the y axis, the name of the file that contains the image to be used as height map and the texture file to be used. It also receives an integer named `textInc` that we will discuss later on.

The first thing that we do in the constructor is to load the height map image into a `BufferedImage` instance.

```
this.minY = minY;
this.maxY = maxY;

PNGDecoder decoder = new PNGDecoder(getClass().getResourceAsStream(heightMapFile));
int height = decoder.getHeight();
int width = decoder.getWidth();
ByteBuffer buf = ByteBuffer.allocateDirect(
        4 * decoder.getWidth() * decoder.getHeight());
decoder.decode(buf, decoder.getWidth() * 4, PNGDecoder.Format.RGBA);
buf.flip();
```

Then, we load the texture file into a `ByteBuffer` and setup the variables that we will need to construct the `Mesh`. The `incx` and `incz` variables will have the increment to be applied to each vertex in the x and z coordinates so the `Mesh` covers the range stated above.

```
Texture texture = new Texture(textureFile);

float incx = getWidth() / (width - 1);
float incz = Math.abs(STARTZ * 2) / (height - 1);

List<Float> positions = new ArrayList();
List<Float> textCoords = new ArrayList();
List<Integer> indices = new ArrayList();
```

After that we are ready to iterate over the image, creating a vertex per each pixel, setting up its texture coordinates and setting up the indices to define correctly the triangles that compose the `Mesh`.

www.dbooks.org

```
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        // Create vertex for current position
        positions.add(STARTX + col * incx); // x
        positions.add(getHeight(col, row, width, buf)); //y
        positions.add(STARTZ + row * incz); //z

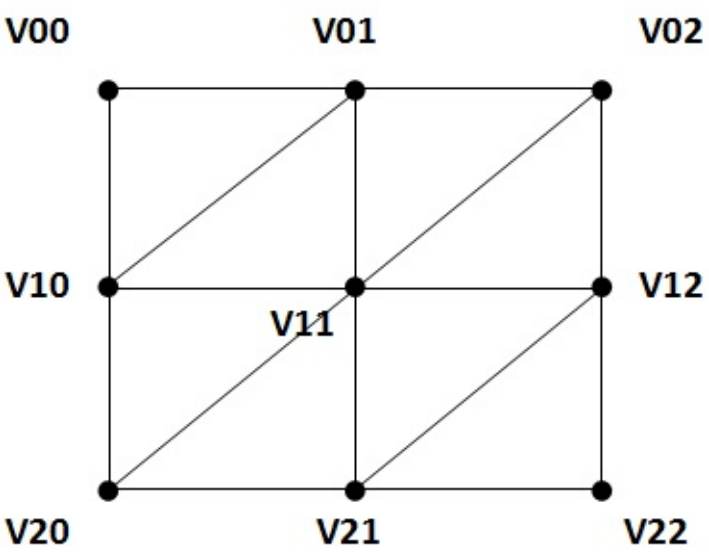
        // Set texture coordinates
        textCoords.add((float) textInc * (float) col / (float) width);
        textCoords.add((float) textInc * (float) row / (float) height);

        // Create indices
        if (col < width - 1 && row < height - 1) {
            int leftTop = row * width + col;
            int leftBottom = (row + 1) * width + col;
            int rightBottom = (row + 1) * width + col + 1;
            int rightTop = row * width + col + 1;

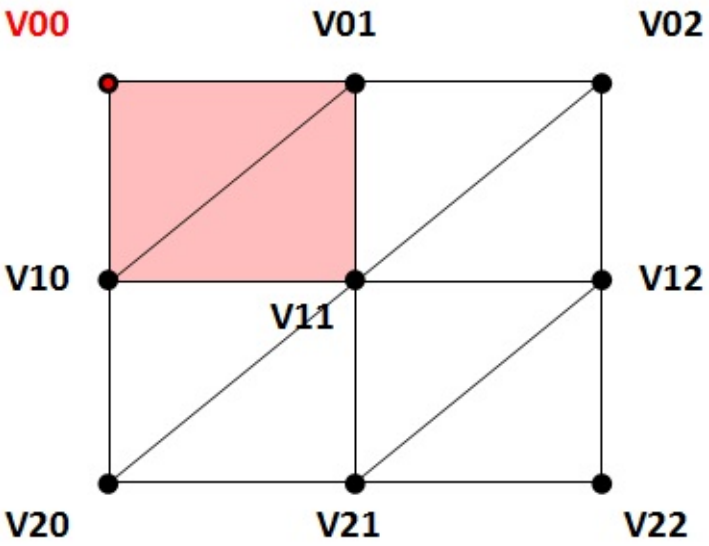
            indices.add(rightTop);
            indices.add(leftBottom);
            indices.add(leftTop);

            indices.add(rightBottom);
            indices.add(leftBottom);
            indices.add(rightTop);
        }
    }
}
```

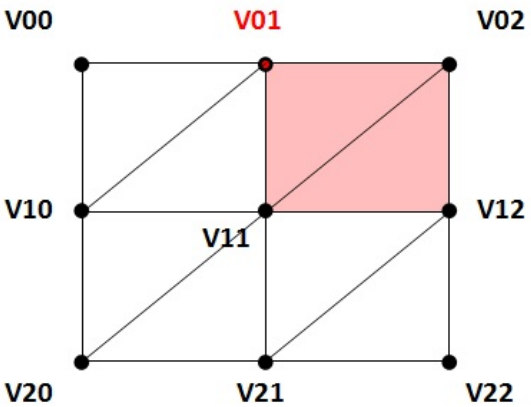
The process of creating the vertex coordinates is self explanatory. Let's ignore at this moment why we multiply the texture coordinates by a number and how the height is calculated. You can see that for each vertex we define the indices of two triangles (except if we are in the last row or column). Let's visualize it with a 3×3 image to visualize how they are constructed. A 3×3 image contains 9 vertices, and thus 4 quads formed by 2×4 triangles. The following picture shows that grid, naming each vertex in the form Vrc (r: row, c: column).

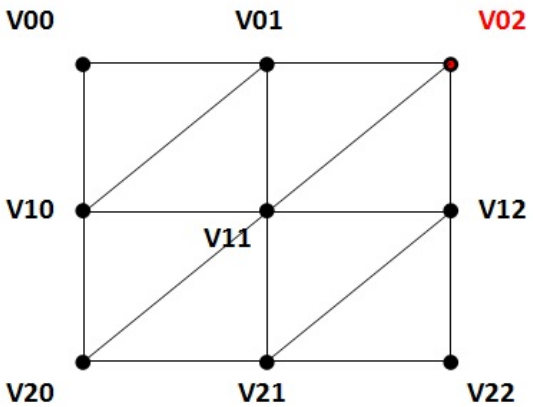


When we are processing the first vertex (V00), we define the indices of the two triangles shaded in red.



When we are processing the second vertex (V01), we define the indices of the two triangles shaded in red. But, when we are processing the the third vertex (V02) we do not need to define more indices, the triangles for that row have already been defined.





www.dbooks.org

You can easily see how the process continues for the rest of vertices. Now, once we have created all the vertices positions, the texture coordinates and the indices we just need to create a `Mesh` and the associated `Material` with all that data.

```
float[] posArr = Utils.listToArray(positions);
int[] indicesArr = indices.stream().mapToInt(i -> i).toArray();
float[] textCoordsArr = Utils.listToArray(textCoords);
float[] normalsArr = calcNormals(posArr, width, height);
this.mesh = new Mesh(posArr, textCoordsArr, normalsArr, indicesArr);
Material material = new Material(texture, 0.0f);
mesh.setMaterial(material);
```

You can see that we calculate the normals taking as an input the vertex positions. Before we see how normals can be calculated, let's see how heights are obtained. We have created a method named `getHeight` which calculates the height for a vertex.

```
private float getHeight(int x, int z, int width, ByteBuffer buffer) {
    byte r = buffer.get(x * 4 + 0 + z * 4 * width);
    byte g = buffer.get(x * 4 + 1 + z * 4 * width);
    byte b = buffer.get(x * 4 + 2 + z * 4 * width);
    byte a = buffer.get(x * 4 + 3 + z * 4 * width);
    int argb = ((0xFF & a) << 24) | ((0xFF & r) << 16)
            | ((0xFF & g) << 8) | (0xFF & b);
    return this.minY + Math.abs(this.maxY - this.minY) * ((float) argb / (float) MAX_C
OLOUR);
    }
```

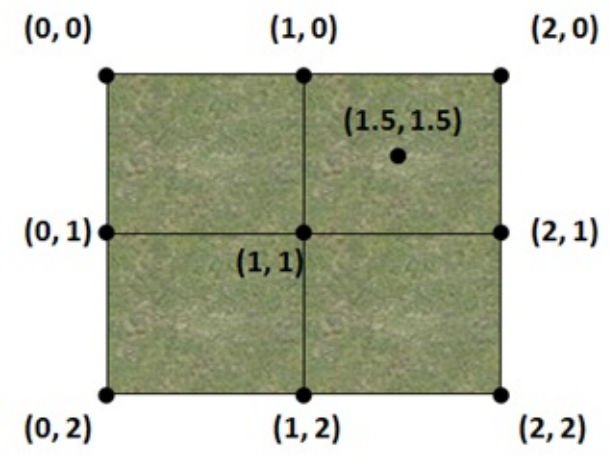
The method receives the x an z coordinates for a pixel, the width of the image and the `ByteBuffer` that conatins it and returns the RGB colour (the sum of the individual R, G and B components) and assigns a value contained between `minY` and `maxY` (`minY` for black colour and `maxY` for white colour).

You may develop asimpler version using a `BufferedImage` which contains handy ethods dor getting RGB values, but we would be using AWT. Remember that AWT does not mix well with OSX so try to avoid using their classes.

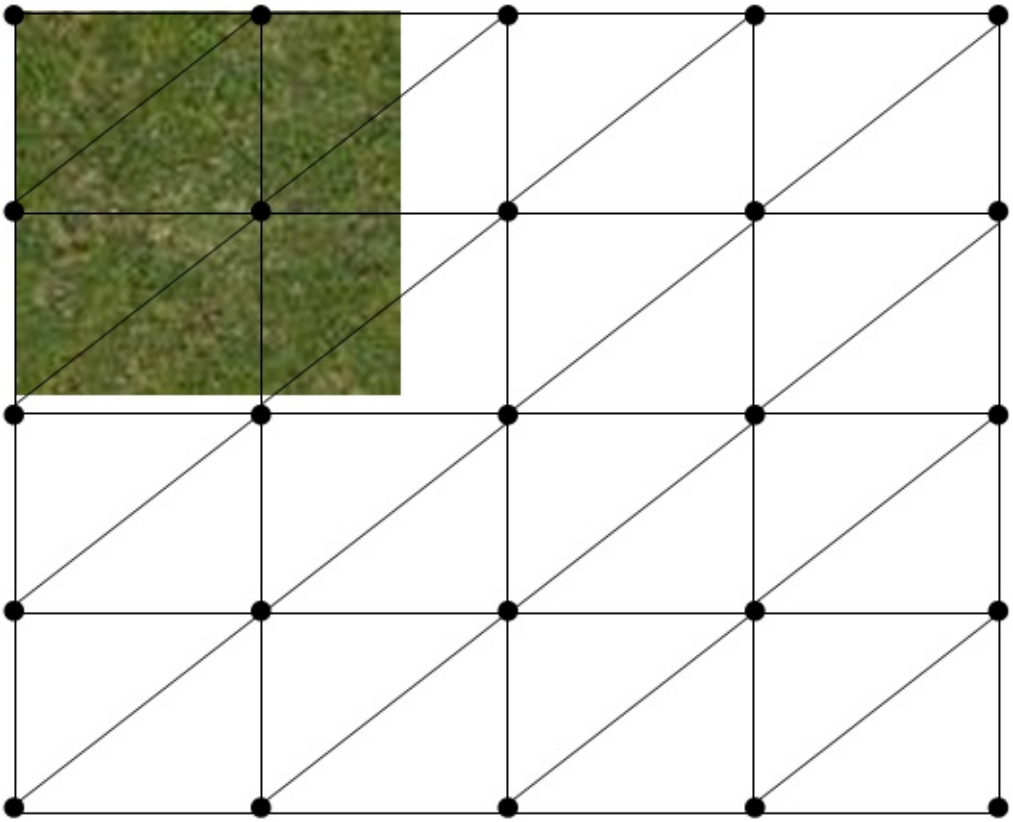
Let's view now how texture coordinates are calculated. The first option is to wrap the texture along the whole mesh, the top left vertex would have (0, 0) texture coordinates and the bottom right vertex would have (1, 1) texture coordinates. The problem with this approach is that the texture should be huge in order to provide good results, if not, it would be stretched too much.

But we can still use a small texture with very good results by employing a very efficient technique. If we set texture coordinates that are beyond the $[1, 1]$ range, we get back to origin and start counting again from the start. The following picture shows this behavior tiling

the same texture in several quads that extend beyond the $[1, 1]$ range.



This is what we will do when setting the texture coordinates. We will be multiplying the texture coordinates (calculated as if the texture just was wrapped covering the whole mesh) by a factor, the `textInc` parameter, to increase the number of pixels of the texture to be used between adjacent vertices.

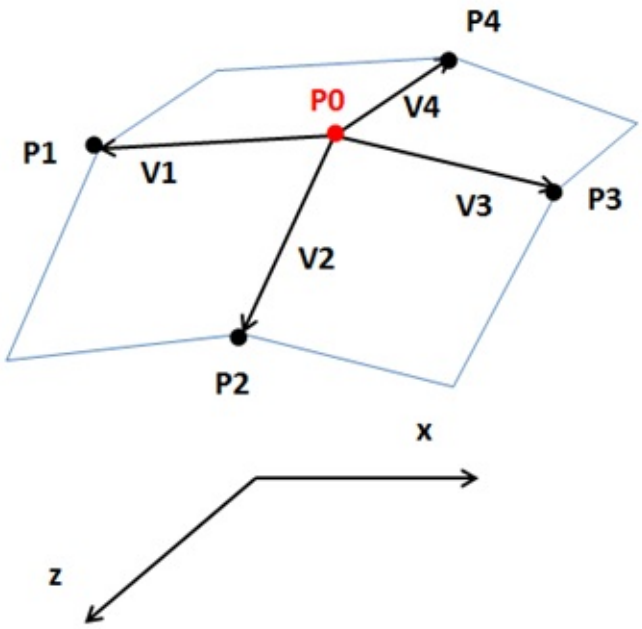


The only thing that's pending now is normal calculation. Remember that we need normals so light can be applied to the terrain correctly. Without normals our terrain will be rendered with the same colour no matter how light hits each point. The method that we will use here may not be the most efficient for height maps but it will help you understand how normals can be auto-calculated. If you search for other solutions you may find more efficient approaches that

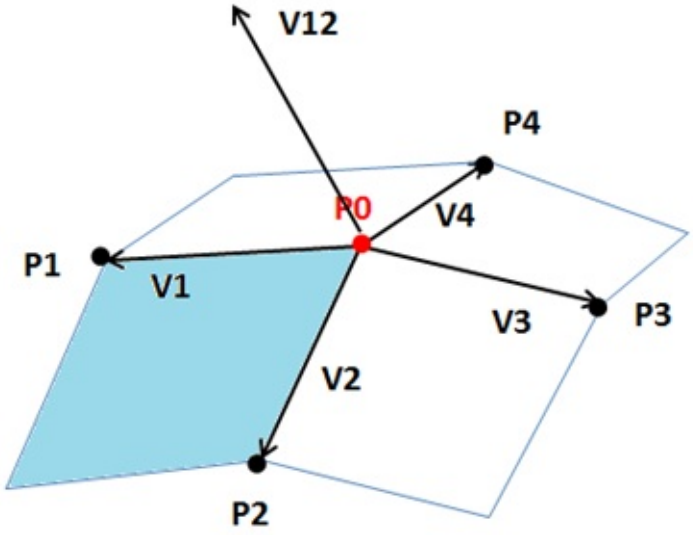
www.dbooks.org

only use the heights of adjacent points without performing cross product operations. Nevertheless since this will only be done at startup, the method presented here will not hurt performance so much.

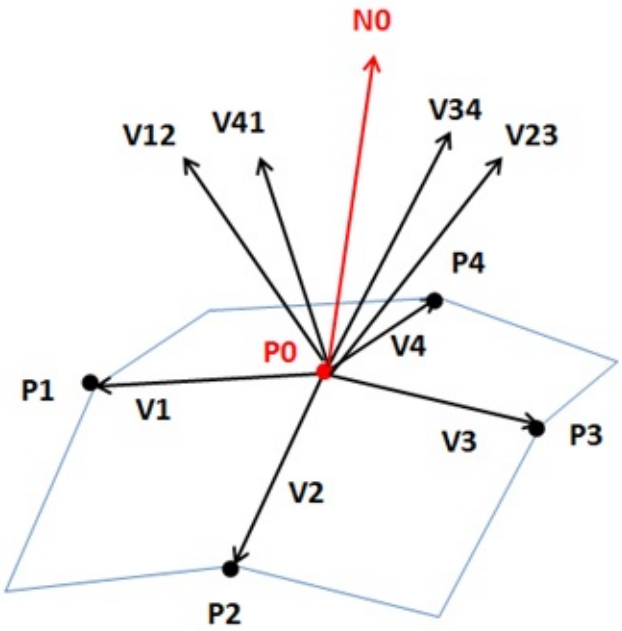
Let's graphically explain how a normal can be calculated. Imagine that we have a vertex named $P0$. We first calculate, for each of the surrounding vertices ($P1$, $P02$, $P3$ and $P4$), the vectors that are tangent to the surface that connects these points. These vectors, ($V1$, $V2$, $V3$ and $V4$), are calculated by subtracting each adjacent point from $P0$ ($V1 = P1 - P0$, etc.)



Then, we calculate the normal for each of the planes that connects the adjacent points. This is done by performing the cross product between the previous calculated vectors. For instance, the normal of the surface that connects $P1$ and $P2$ (shaded in blue) is calculated as the cross product between $V1$ and $V2$, $V12 = V1 \times V2$.



If we calculate the rest of the normals for the rest of the surfaces ($V23 = V2 \times V3$, $V34 = V3 \times V4$ and $V41 = V4 \times V1$), the normal for $P0$ will be the sum (normalized) of all the normals of the surrounding surfaces: $\hat{N0} = \hat{V12} + \hat{V23} + \hat{V34} + \hat{V41}$.



The implementation of the method that calculates the normals is as follows.

```
private float[] calcNormals(float[] posArr, int width, int height) {
    Vector3f v0 = new Vector3f();
    Vector3f v1 = new Vector3f();
    Vector3f v2 = new Vector3f();
    Vector3f v3 = new Vector3f();
    Vector3f v4 = new Vector3f();
    Vector3f v12 = new Vector3f();
    Vector3f v23 = new Vector3f();
```


www.dbooks.org

```
        Vector3f v34 = new Vector3f();
        Vector3f v41 = new Vector3f();
        List<Float> normals = new ArrayList<>();
        Vector3f normal = new Vector3f();
        for (int row = 0; row < height; row++) {
            for (int col = 0; col < width; col++) {
                if (row > 0 && row < height -1 && col > 0 && col < width -1) {
                    int i0 = row*width*3 + col*3;
                    v0.x = posArr[i0];
                    v0.y = posArr[i0 + 1];
                    v0.z = posArr[i0 + 2];

                    int i1 = row*width*3 + (col-1)*3;
                    v1.x = posArr[i1];
                    v1.y = posArr[i1 + 1];
                    v1.z = posArr[i1 + 2];
                    v1 = v1.sub(v0);

                    int i2 = (row+1)*width*3 + col*3;
                    v2.x = posArr[i2];
                    v2.y = posArr[i2 + 1];
                    v2.z = posArr[i2 + 2];
                    v2 = v2.sub(v0);

                    int i3 = (row)*width*3 + (col+1)*3;
                    v3.x = posArr[i3];
                    v3.y = posArr[i3 + 1];
                    v3.z = posArr[i3 + 2];
                    v3 = v3.sub(v0);

                    int i4 = (row-1)*width*3 + col*3;
                    v4.x = posArr[i4];
                    v4.y = posArr[i4 + 1];
                    v4.z = posArr[i4 + 2];
                    v4 = v4.sub(v0);

                    v1.cross(v2, v12);
                    v12.normalize();

                    v2.cross(v3, v23);
                    v23.normalize();

                    v3.cross(v4, v34);
                    v34.normalize();

                    v4.cross(v1, v41);
                    v41.normalize();

                    normal = v12.add(v23).add(v34).add(v41);
                    normal.normalize();
                } else {
                    normal.x = 0;
                    normal.y = 1;
```

```
                normal.z = 0;
            }
            normal.normalize();
            normals.add(normal.x);
            normals.add(normal.y);
            normals.add(normal.z);
        }
    }
    return Utils.listToArray(normals);
}
```

Finally, in order to build larger terrains, we have two options:

- Create a larger height map.
- Reuse a height map and tile it through the 3D space. The height map will be like a terrain block that could be translated across the world like tiles. In order to do so, the pixels of the edge of the height map must be the same (the left edge must be equal to the right side and the top edge must be equal to the bottom one) to avoid gaps between the tiles.

We will use the second approach (and select an appropriate height map). To support this, we will create a class named `Terrain` that will create a square of height map tiles, defined like this.

```
package org.lwjglb.engine.items;

import org.lwjglb.engine.graph.HeightMapMesh;

public class Terrain {

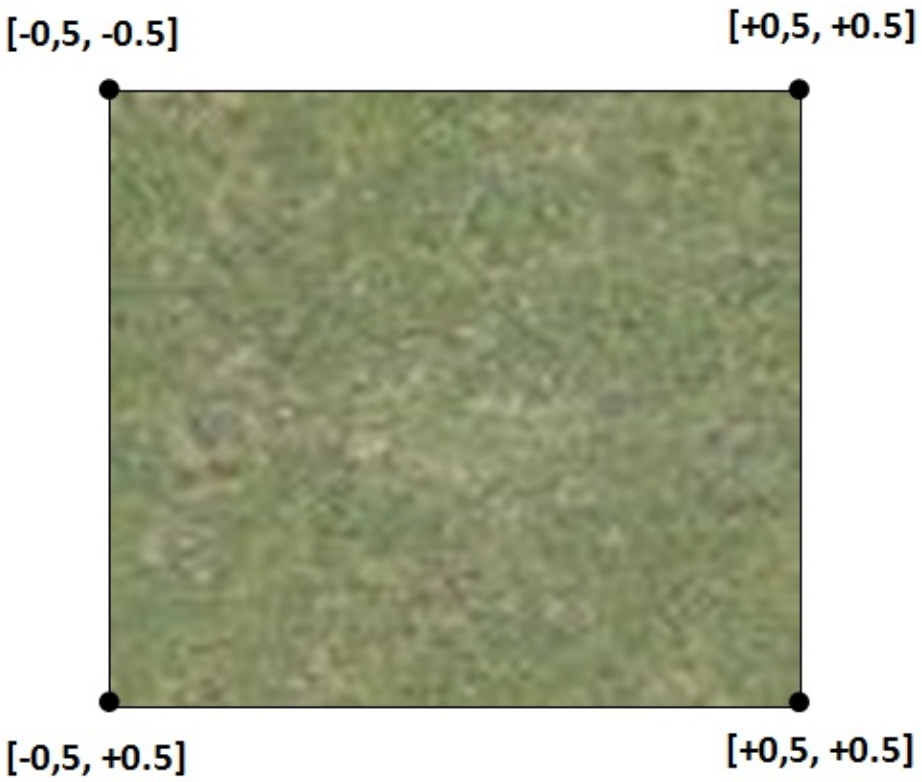
    private final GameItem[] gameItems;

    public Terrain(int blocksPerRow, float scale, float minY, float maxY, String heigh
tMap, String textureFile, int textInc) throws Exception {
        gameItems = new GameItem[blocksPerRow * blocksPerRow];
        HeightMapMesh heightMapMesh = new HeightMapMesh(minY, maxY, heightMap, texture
File, textInc);
        for (int row = 0; row < blocksPerRow; row++) {
            for (int col = 0; col < blocksPerRow; col++) {
                float xDisplacement = (col - ((float) blocksPerRow - 1) / (float) 2) *
 scale * HeightMapMesh.getXLength();
                float zDisplacement = (row - ((float) blocksPerRow - 1) / (float) 2) *
 scale * HeightMapMesh.getZLength();

                GameItem terrainBlock = new GameItem(heightMapMesh.getMesh());
                terrainBlock.setScale(scale);
                terrainBlock.setPosition(xDisplacement, 0, zDisplacement);
                gameItems[row * blocksPerRow + col] = terrainBlock;
            }
        }
    }

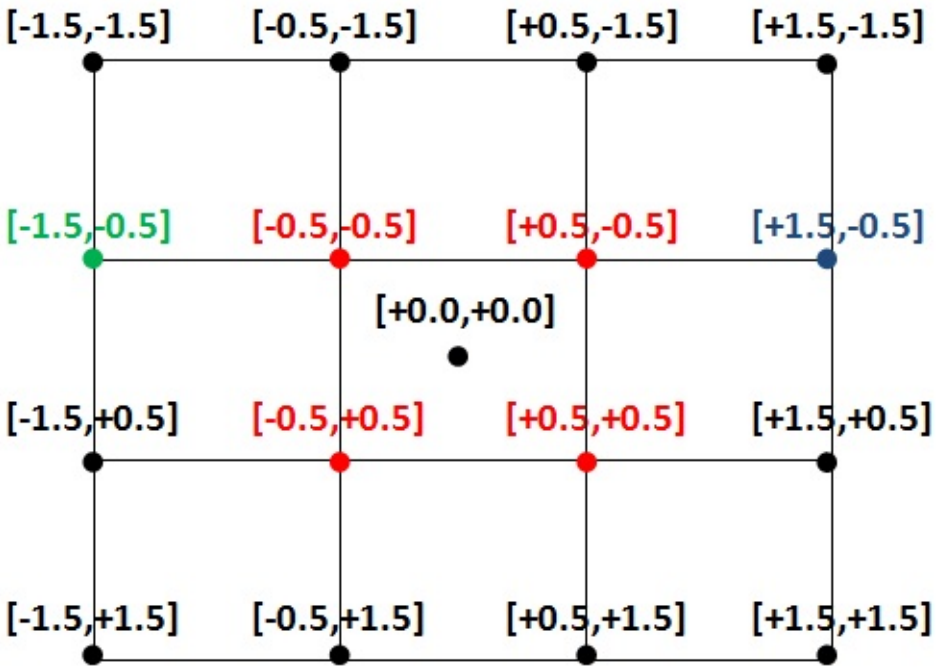
    public GameItem[] getGameItems() {
        return gameItems;
    }
}
```

Let's explain the overall process, we have blocks that have the following coordinates (for x and z and with the constants defined above).



Let's suppose that we are create¡ing a terrain formed by a 3x3 blocks grid. Let's assume also thatwe wont' scale the terrain blocks (that is, the variable `blocksPerRow` will be 3 and the variable `scale` will be 1). We want the grid to be centered at (0, 0) coordinates.

We need to translate the blocks so the vertices will have the following coordinates.



The translation is achieved by calling `setPosition` method, but remember that what we set is a displacement not a position. If you review the figure above you will see that the central block does not require any displacement, it's already positioned in the adequate coordinates.

www.dbooks.org

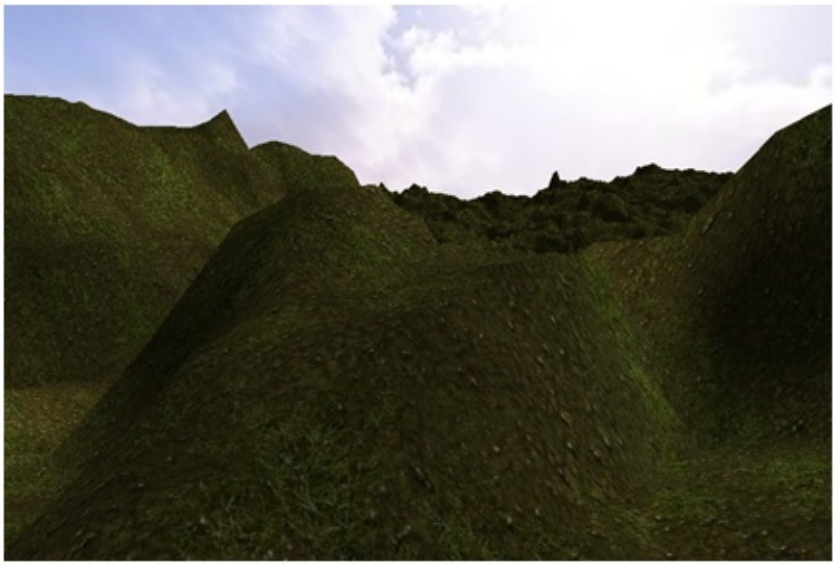
The vertex drawn in green needs a displacement, for the x coordinate, of -1 and the vertex drawn in blue needs a displacement of $+1$. The formula to calculate the x displacement, taking into consideration the scale and the block width, is this one:

$$xDisplacement = (col - (blocksPerRow - 1)/2) \times scale \times width$$

And the equivalent formula for z displacement is:

$$zDisplacement = (row - (blocksPerRow - 1)/2) \times scale \times height$$

If we create a Terrain instance in the `DummyGame` class, we can get something like this.



You can move the camera around the terrain and see how it's rendered. Since we still do not have implemented collision detection you can pass through it and look it from above. Because we have face culling enabled, some parts of the terrain are not rendered when looking from below.

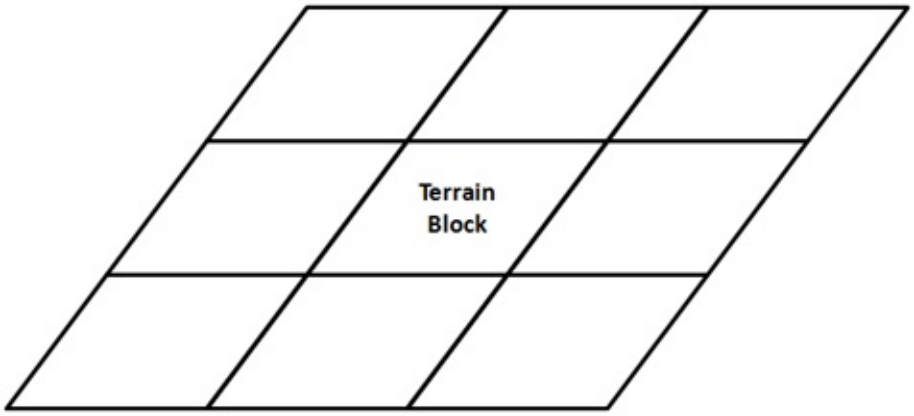
Terrain Collisions

Once we have created a terrain the next step is to detect collisions to avoid traversing through it. If you recall from previous chapter, a terrain is composed by blocks, and each of those blocks is constructed from a height map. The height map is used to set the height of the vertices that compose the triangles that form the terrain.

In order to detect a collision we must compare current position y value with the y value of the point of the terrain we are currently in. If we are above terrain's y value there's no collision, if not, we need to get back. Simple concept, does it ? Indeed it is but we need to perform several calculations before we are able to do that comparison.

The first thing we need to define is what we understand for the term "current position". Since we do not have yet a player concept the answer is easy, the current position will be the camera position. So we already have one of the components of the comparison, thus, the next thing to calculate is terrain height at current position.

As it's been said before, the terrain is composed by a grid of terrain blocks as shown in the next figure.



Each terrain block is constructed from the same height map mesh, but is scaled and displaced precisely to form a terrain grid that looks like a continuous landscape.

So, what we need to do first is determine in which terrain block the current position, the camera, is in. In order to do that, we will calculate the bounding box of each terrain block taking into consideration the displacement and the scaling. Since the terrain will not be displaced or scaled at runtime, we can perform those calculations in the `Terrain` class constructor. By doing this way we access them later at any time without repeating those operations in each game loop cycle.

We will create a new method that calculates the bounding box of a terrain block, named `getBoundingBox`.

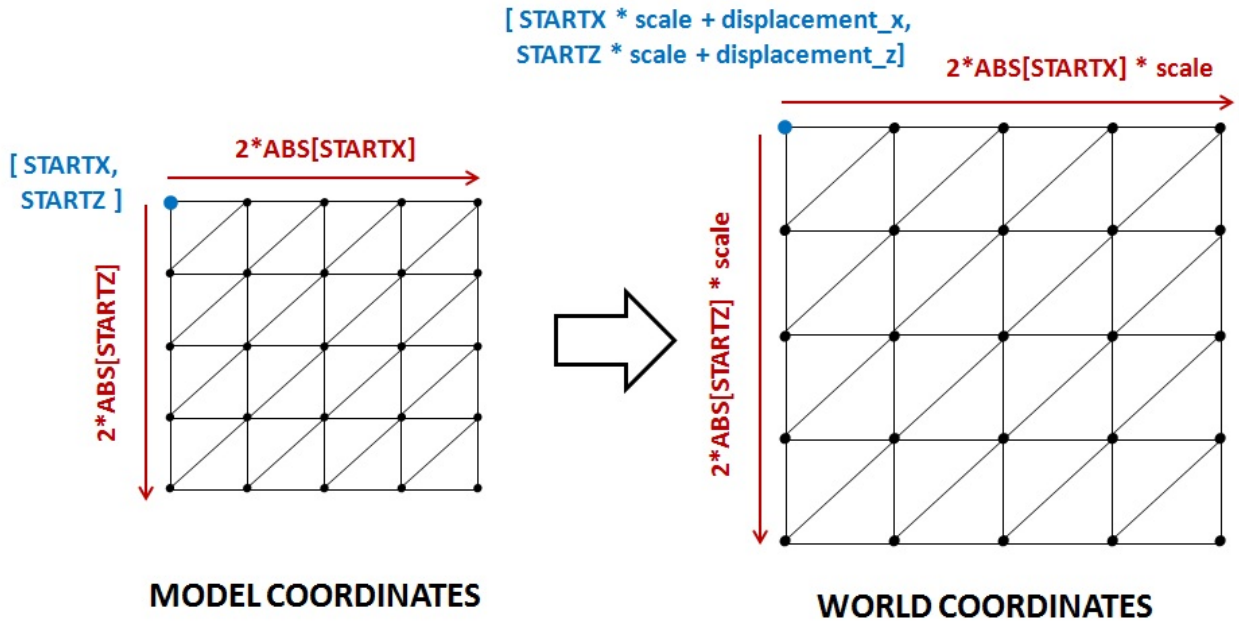
www.dbooks.org

```
private Box2D getBoundingBox(GameItem terrainBlock) {
    float scale = terrainBlock.getScale();
    Vector3f position = terrainBlock.getPosition();

    float topLeftX = HeightMapMesh.STARTX * scale + position.x;
    float topLeftZ = HeightMapMesh.STARTZ * scale + position.z;
    float width = Math.abs(HeightMapMesh.STARTX * 2) * scale;
    float height = Math.abs(HeightMapMesh.STARTZ * 2) * scale;
    Box2D boundingBox = new Box2D(topLeftX, topLeftZ, width, height);
    return boundingBox;
}
```

The `Box2D` class is a simplified version of the `java.awt.Rectangle2D.Float` class; created to avoid using AWT.

Now we need to calculate the world coordinates of the terrain blocks. In the previous chapter you saw that all of our terrain meshes were created inside a quad with its origin set to `[STARTX, STARTZ]`. Thus,we need to transform those coordinates to the world coordinates taking into consideration the scale and the displacement as shown in the next figure.



As it's been said above, this can be done in the Terrain class constructor since it won't change at run time. So we need to add a new attribute which will hold the bounding boxes:

```
private final Box2D[][] boundingBoxes;
```

In the `Terrain` constructor, while we are creating the terrain blocks, we just need to invoke the method that calculates the bounding box.

```
public Terrain(int terrainSize, float scale, float minY, float maxY, String heightMapF
ile, String textureFile, int textInc) throws Exception {
    this.terrainSize = terrainSize;
    gameItems = new GameItem[terrainSize * terrainSize];

    PNGDecoder decoder = new PNGDecoder(getClass().getResourceAsStream(heightMapFile))
;
    int height = decoder.getHeight();
    int width = decoder.getWidth();
    ByteBuffer buf = ByteBuffer.allocateDirect(
            4 * decoder.getWidth() * decoder.getHeight());
    decoder.decode(buf, decoder.getWidth() * 4, PNGDecoder.Format.RGBA);
    buf.flip();

    // The number of vertices per column and row
    verticesPerCol = heightMapImage.getWidth();
    verticesPerRow = heightMapImage.getHeight();

    heightMapMesh = new HeightMapMesh(minY, maxY, buf, width, textureFile, textInc);
    boundingBoxes = new Box2D[terrainSize][terrainSize];
    for (int row = 0; row < terrainSize; row++) {
        for (int col = 0; col < terrainSize; col++) {
            float xDisplacement = (col - ((float) terrainSize - 1) / (float) 2) * scal
e * HeightMapMesh.getXLength();
            float zDisplacement = (row - ((float) terrainSize - 1) / (float) 2) * scal
e * HeightMapMesh.getZLength();

            GameItem terrainBlock = new GameItem(heightMapMesh.getMesh());
            terrainBlock.setScale(scale);
            terrainBlock.setPosition(xDisplacement, 0, zDisplacement);
            gameItems[row * terrainSize + col] = terrainBlock;

            boundingBoxes[row][col] = getBoundingBox(terrainBlock);
        }
    }
}
```

So, with all the bounding boxes pre-calculated, we are ready to create a new method that will return the height of the terrain taking as a parameter the current position. This method will be named `getHeightVector` and its defined like this.

www.dbooks.org

```
public float getHeight(Vector3f position) {
    float result = Float.MIN_VALUE;
    // For each terrain block we get the bounding box, translate it to view coodinates
    // and check if the position is contained in that bounding box
    Box2D boundingBox = null;
    boolean found = false;
    GameItem terrainBlock = null;
    for (int row = 0; row < terrainSize && !found; row++) {
        for (int col = 0; col < terrainSize && !found; col++) {
            terrainBlock = gameItems[row * terrainSize + col];
            boundingBox = boundingBoxes[row][col];
            found = boundingBox.contains(position.x, position.z);
        }
    }

    // If we have found a terrain block that contains the position we need
    // to calculate the height of the terrain on that position
    if (found) {
        Vector3f[] triangle = getTriangle(position, boundingBox, terrainBlock);
        result = interpolateHeight(triangle[0], triangle[1], triangle[2], position.x,
position.z);
    }

    return result;
}
```

The first thing that to we do in that method is to determine the terrain block that we are in. Since we already have the bounding box for each terrain block, the algorithm is simple. We just simply need to iterate over the array of bounding boxes and check if the current position is inside (the class `Box2D` provides a method for this).

Once we have found the terrain block, we need to calculate the triangle which we are in. This is done in the `getTriangle` method that will be described later on. After that, we have the coordinates of the triangle that we are in, including its height. But, we need the height of a point that is not located at any of those vertices but in a place in between. This is done in the $interpolateHeight$ method. We will also explain how this is done later on.

Let's first start with the process of determining the triangle that we are in. The quad that forms a terrain block can be seen as a grid in which each cell is formed by two triangles Let's define some variables first:

- $boundingBox.x$ is the x coordinate of the origin of the bounding box associated to the quad.
- $boundingBox.y$ is the z coordinates of the origin of the bounding box associated to the quad (Although you see a "y", it models the z axis).
- $boundingBox.width$ is the width of the quad
- $boundingBox.height$ is the height of the quad.

- $cellWidth$ is the width of a cell.
- $cellHeight$ is the height of a cell.

All of the variables defined above are expressed in world coordinates. To calculate the width of a cell we just need to divide the bounding box width by the number of vertices per column:

$$cellWidth = \frac{boundingBox.width}{verticesPerCol}$$

And the variable `cellHeight` is calculated analogous

$$cellHeight = \frac{boundingBox.height}{verticesPerRow}$$

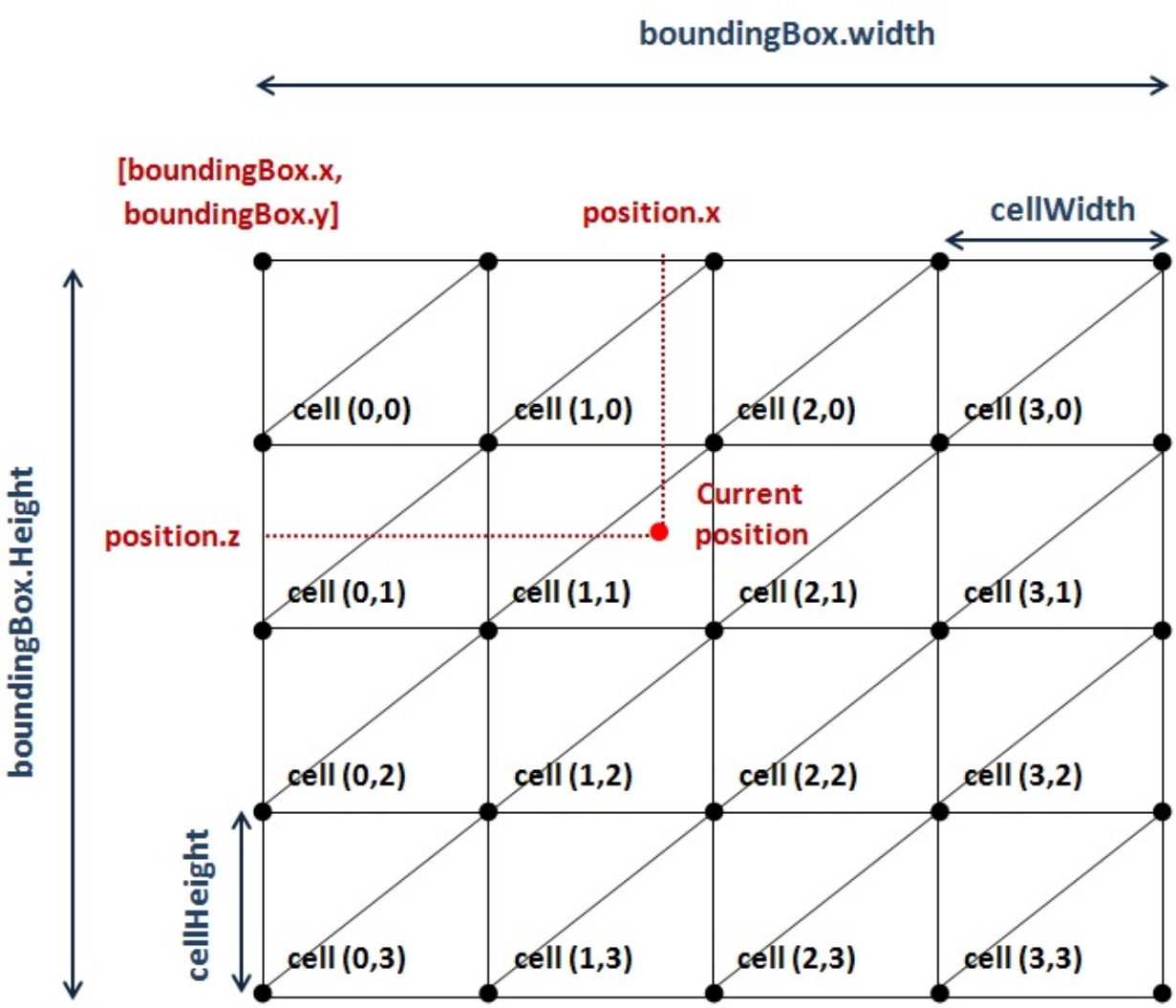
Once we have those variables we can calculate the row and the column of the cell we are currently in width is quite straight forward:

$$col = \frac{position.x - boundingBox.x}{boundingBox.width}$$

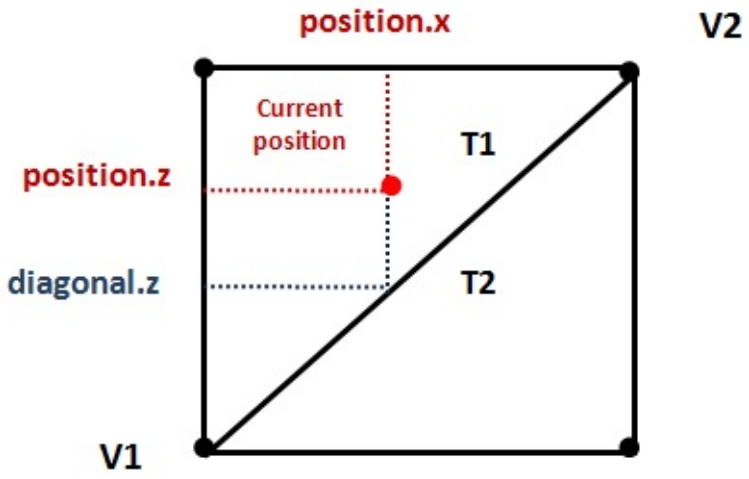
$$row = \frac{position.z - boundingBox.y}{boundingBox.height}$$

The following picture shows all the variables described above for a sample terrain block.

www.dbooks.org



With all that information we are able to calculate the positions of the vertices of the triangles contained in the cell. How we can do this ? Let's examine the triangles that form a single cell.



You can see that the cell is divided by a diagonal that separates the two triangles. The way to determine the triangle associated to the current position, is by checking if the z coordinate is above or below that diagonal. In our case, if current position z value is less than the z

value of the diagonal setting the x value to the x value of current position we are in T1. If it's greater than that we are in T2.

We can determine that by calculating the line equation that matches the diagonal.

If you remember your school math classes, the equation of a line that passes from two points (in 2D) is:

$$y - y1 = m \cdot (x - x1)$$

Where m is the line slope, that is, how much the height changes when moving through the x axis. Note that, in our case, the y coordinates are the z ones. Also note, that we are using 2D coordinates because we are not calculating heights here. We just want to select the proper triangle and to do that x an z coordinates are enough. So, in our case the line equation should be rewritten like this.

$$z - z1 = m \cdot (z - z1)$$

The slope can be calculate in the following way:

$$m = \frac{z1 - z2}{x1 - x2}$$

So the equation of the diagonal to get the z value given a x position is like this:

$$z = m \cdot (xpos - x1) + z1 = \frac{z1 - z2}{x1 - x2} \cdot (zpos - x1) + z1$$

Where $x1$,$x2$,$z1$ and $z2$ are the x and z coordinates of the vertices $V1$ and $V2$ respectively.

So the method to get the triangle that the current position is in, named `getTriangle`, applying all the calculations described above can be implemented like this:

www.dbooks.org

```
protected Vector3f[] getTriangle(Vector3f position, Box2D boundingBox, GameItem terrai
nBlock) {
    // Get the column and row of the heightmap associated to the current position
    float cellWidth = boundingBox.width / (float) verticesPerCol;
    float cellHeight = boundingBox.height / (float) verticesPerRow;
    int col = (int) ((position.x - boundingBox.x) / cellWidth);
    int row = (int) ((position.z - boundingBox.y) / cellHeight);

    Vector3f[] triangle = new Vector3f[3];
    triangle[1] = new Vector3f(
        boundingBox.x + col * cellWidth,
        getWorldHeight(row + 1, col, terrainBlock),
        boundingBox.y + (row + 1) * cellHeight);
    triangle[2] = new Vector3f(
        boundingBox.x + (col + 1) * cellWidth,
        getWorldHeight(row, col + 1, terrainBlock),
        boundingBox.y + row * cellHeight);
    if (position.z < getDiagonalZCoord(triangle[1].x, triangle[1].z, triangle[2].x, tr
iangle[2].z, position.x)) {
        triangle[0] = new Vector3f(
            boundingBox.x + col * cellWidth,
            getWorldHeight(row, col, terrainBlock),
            boundingBox.y + row * cellHeight);
    } else {
        triangle[0] = new Vector3f(
            boundingBox.x + (col + 1) * cellWidth,
            getWorldHeight(row + 2, col + 1, terrainBlock),
            boundingBox.y + (row + 1) * cellHeight);
    }

    return triangle;
}

protected float getDiagonalZCoord(float x1, float z1, float x2, float z2, float x) {
    float z = ((z1 - z2) / (x1 - x2)) * (x - x1) + z1;
    return z;
}

protected float getWorldHeight(int row, int col, GameItem gameItem) {
    float y = heightMapMesh.getHeight(row, col);
    return y * gameItem.getScale() + gameItem.getPosition().y;
}
```

You can see that we have two additional methods. The first one, named `getDiagonalZCoord`, calculates the z coordinate of the diagonal given a x position and two vertices. The other one, named `getWorldHeight`, is used to retrieve the height of the triangle vertices, the y coordinate. When the terrain mesh is constructed the height of each vertex is pre-calculated and stored, we only need to translate it to world coordinates.

Ok, so we have the triangle coordinates that the current position is in. Finally, we are ready to calculate terrain height at current position. How can we do this ? Well, our triangle is contained in a plane, and a plane can be defined by three points, in this case, the three vertices that define a triangle.

The plane equation is as follows: $a \cdot x + b \cdot y + c \cdot z + d = 0$

The values of the constants of the previous equation are:

$$a = (B_y - A_y) \cdot (C_z - A_z) - (C_y - A_y) \cdot (B_z - A_z)$$

$$b = (B_z - A_z) \cdot (C_x - A_x) - (C_z - A_z) \cdot (B_z - A_z)$$

$$c = (B_x - A_x) \cdot (C_y - A_y) - (C_x - A_x) \cdot (B_y - A_y)$$

Where A, B and C are the three vertices needed to define the plane.

Then, with previous equations and the values of the x and z coordinates for the current position we are able to calculate the y value, that is the height of the terrain at the current position:

$$y = (-d - a \cdot x - c \cdot z)/b$$

The method that performs the previous calculations is the following:

```
protected float interpolateHeight(Vector3f pA, Vector3f pB, Vector3f pC, float x, float
 z) {
    // Plane equation ax+by+cz+d=0
    float a = (pB.y - pA.y) * (pC.z - pA.z) - (pC.y - pA.y) * (pB.z - pA.z);
    float b = (pB.z - pA.z) * (pC.x - pA.x) - (pC.z - pA.z) * (pB.x - pA.x);
    float c = (pB.x - pA.x) * (pC.y - pA.y) - (pC.x - pA.x) * (pB.y - pA.y);
    float d = -(a * pA.x + b * pA.y + c * pA.z);
    // y = (-d -ax -cz) / b
    float y = (-d - a * x - c * z) / b;
    return y;
}
```

And that's all ! we are now able to detect the collisions, so in the `DummyGame` class we can change the following lines when we update the camera position:

www.dbooks.org

```
// Update camera position
Vector3f prevPos = new Vector3f(camera.getPosition());
camera.movePosition(cameraInc.x * CAMERA_POS_STEP, cameraInc.y * CAMERA_POS_STEP, came
raInc.z * CAMERA_POS_STEP);
// Check if there has been a collision. If true, set the y position to
// the maximum height
float height = terrain.getHeight(camera.getPosition());
if ( camera.getPosition().y <= height )  {
    camera.setPosition(prevPos.x, prevPos.y, prevPos.z);
}
```

As you can see the concept of detecting terrain collisions is easy to understand but we need to carefully perform a set of calculations and be aware of the different coordinate systems we are dealing with.

Besides that, although the algorithm presented here is valid in most of the cases, there are still situations that need to be handled carefully. One effect that you may observe is the one called tunnelling. Imagine the following situation, we are travelling at a fast speed through our terrain and because of that, the position increment gets a high value. This value can get so high that, since we are detecting collisions with the final position, we may have skipped obstacles that lay in between.



There are many possible solutions to avoid that effect, the simplest one is to split the calculation to be performed in smaller increments.

Fog

Before we deal with more complex topics we will review how to create a fog effect in our game engine. With that effect we will simulate how distant objects get dimmed and seem to vanish into a dense fog.

Let us first examine what are the attributes that define fog. The first one is the fog colour. In the real world the fog has a gray colour, but we can use this effect to simulate wide areas invaded by a fog with different colours. The attribute is the fog's density.

Thus, in order to apply the fog effect we need to find a way to fade our 3D scene objects into the fog colour as long as they get far away from the camera. Objects that are close to the camera will not be affected by the fog, but objects that are far away will not be distinguishable. So we need to be able to calculate a factor that can be used to blend the fog colour and each fragment colour in order to simulate that effect. That factor will need to be dependent on the distance to the camera.

Let's name that factor as $fogFactor$, and set its range from 0 to 1. When $fogFactor$ takes the 1 value, it means that the object will not be affected by fog, that is, it's a nearby object. When $fogFactor$ takes the 0 value, it means that the objects will be completely hidden in the fog.

Then, the equation needed to calculate the fog colour will be:

$$finalColour = (1 - fogFactor) \cdot fogColour + fogFactor \cdot framentColour$$

- $finalColour$ is the colour that results from applying the fog effect.
- $fogFactor$ is the parameters that controls how the fog colour and the fragment colour are blended. It basically controls the object visibility.
- $fogColour$ is the colour of the fog.
- $fragmentColour$, is the colour of the fragment without applying any fog effect on it.

Now we need to find a way to calculate $fogFactor$ depending on the distance. We can chose different models, and the first one could be to use a linear model. That is a model that, given a distance, changes the fogFactor value in a linear way.

The linear model can be defined by the following parameters:

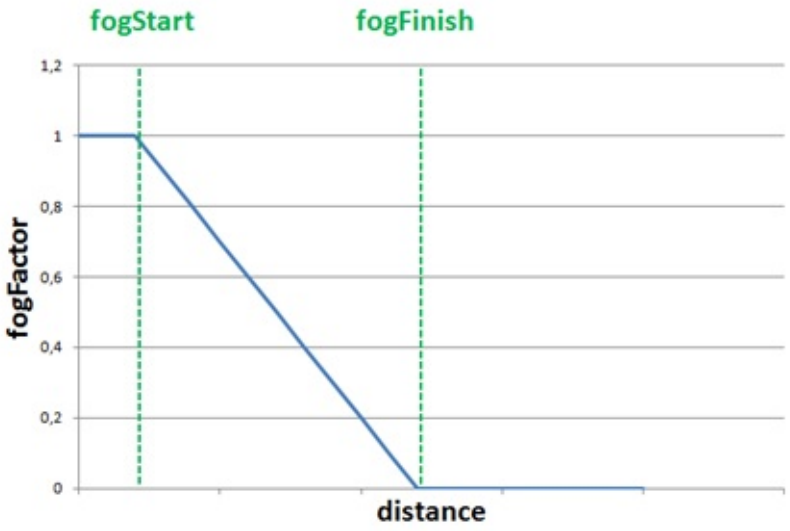
- $fogStart$: The distance at where fog effects starts to be applied.
- $fogFinish$: The distance at where fog effects reaches its maximum value.
- $distance$: Distance to the camera.

With those parameters, the equation to be applied would be:

www.dbooks.org

$$fogFactor = \frac{(fogFinish - distance)}{(fogFinish - fogStart)}$$

For objects at distance lower than $fogStart$ we just simply set the $fogFactor$ to 1. The following graph shows how the $fogFactor$ changes with the distance.



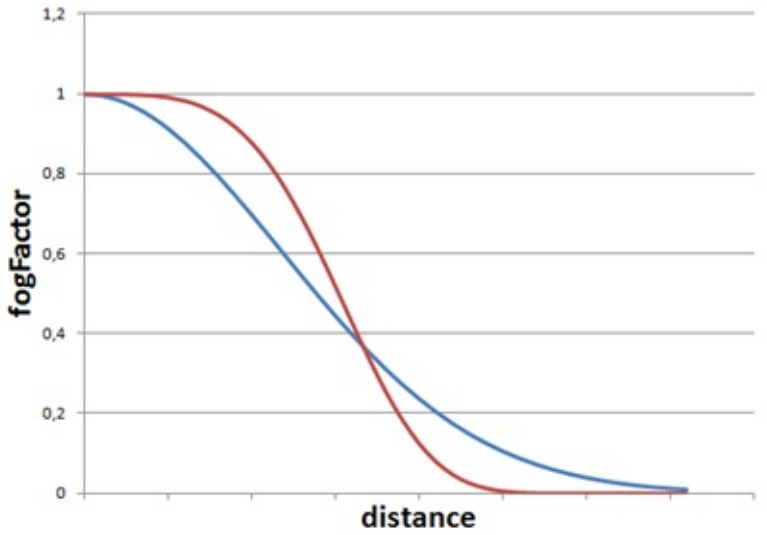
The linear model is easy to calculate but it is not very realistic and it does not take into consideration the fog density. In reality fog tends to grow in a smoother way. So the next suitable model is a exponential one. The equation for that model is as follows:

$$focFactor = e^{-(distance \cdot fogDensity)^{exponent}} = \frac{1}{e^{(distance \cdot fogDensity)^{exponent}}}$$

The new variables that come into play are:

- $fogDensity$ which models the thickness or density of the fog.
- $exponent$ which is used to control how fast the fog increases with distance

The following picture shows two graphs for the equation above for different values of the exponent (2 for the blue line and 4 for the red one)



In our code we will use a formula which sets a value of two for the exponent (you can easily modify the example to use different values).

Now that the theory has been explained we can put it into practice. We will implement the effect in the scene fragment shader since we have there all the variables we need. We will start by defining a struct that models the fog attributes.

```
struct Fog
{
    int active;
    vec3 colour;
    float density;
};
```

The `active` attribute will be used to activate or deactivate the fog effect. The fog will be passed to the shader through another uniform named `fog`.

```
uniform Fog fog;
```

We will create also a new class named `Fog` which is another POJO (Plain Old Java Object) which contains the fog attributes.

www.dbooks.org

```
package org.lwjglb.engine.graph.weather;

import org.joml.Vector3f;

public class Fog {

    private boolean active;

    private Vector3f colour;

    private float density;

    public static Fog NOFOG = new Fog();

    public Fog() {
        active = false;
        this.colour = new Vector3f(0, 0, 0);
        this.density = 0;
    }

    public Fog(boolean active, Vector3f colour, float density) {
        this.colour = colour;
        this.density = density;
        this.active = active;
    }

   // Getters and setters here….
```

We will add a `Fog` instance in the `Scene` class. As a default, the `Scene` class will initialize the `Fog` instance to the constant `NOFOG` which models a deactivated instance.

Since we added a new uniform type we need to modify the `ShaderProgram` class to create and initialize the fog uniform.

```
public void createFogUniform(String uniformName) throws Exception {
    createUniform(uniformName + ".active");
    createUniform(uniformName + ".colour");
    createUniform(uniformName + ".density");
}

public void setUniform(String uniformName, Fog fog) {
    setUniform(uniformName + ".activeFog", fog.isActive() ? 1 : 0);
    setUniform(uniformName + ".colour", fog.getColour() );
    setUniform(uniformName + ".density", fog.getDensity());
}
```

In the `Renderer` class we just need to create the uniform in the `setupSceneShader` method:

```
sceneShaderProgram.createFogUniform("fog");
```

And use it in the `renderScene` method:

```
sceneShaderProgram.setUniform("fog", scene.getFog());
```

We are now able to define fog characteristics in our game, but we need to get back to the fragment shader in order to apply the fog effect. We will create a function named `calcFog` which is defined like this.

```
vec4 calcFog(vec3 pos, vec4 colour, Fog fog)
{
    float distance = length(pos);
    float fogFactor = 1.0 / exp( (distance * fog.density)* (distance * fog.density));
    fogFactor = clamp( fogFactor, 0.0, 1.0 );

    vec3 resultColour = mix(fog.colour, colour.xyz, fogFactor);
    return vec4(resultColour.xyz, colour.w);
}
```

As you can see we first calculate the distance to the vertex. The vertex coordinates are defined in the `pos` variable and we just need to calculate the length. Then we calculate the fog factor using the exponential model with an exponent of two (which is equivalent to multiply it twice). We clamp the `fogFactor` to a range between 0 and 1 and use the `mix` function. In GLSL, the `mix` function is used to blend the fog colour and the fragment colour (defined by variable colour). It's equivalent to apply this equation:

$$resultColour = (1 - fogFactor) \cdot fog.colour + fogFactor \cdot colour$$

We also preserve the w componente, the transparency, of the original colour. We don't want this component to be affected, the fragment should mantain its transparency level.

At the end of the fragment shader after applying all the light effects we just simply assign the returned value to the fragment colour if the fog is active.

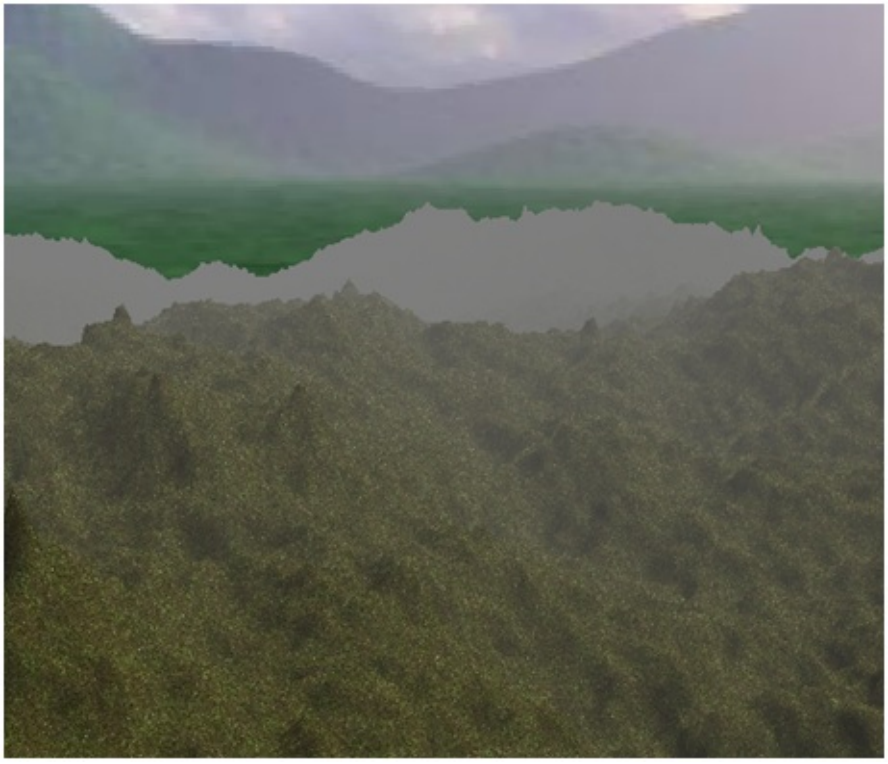
```
if ( fog.activeFog == 1 )
{
    fragColor = calcFog(mvVertexPos, fragColor, fog);
}
```

With all that code completed, we can set up a Fog with the following data:

```
scene.setFog(new Fog(true, new Vector3f(0.5f, 0.5f, 0.5f), 0.15f));
```


www.dbooks.org

And we will get an effect like this:

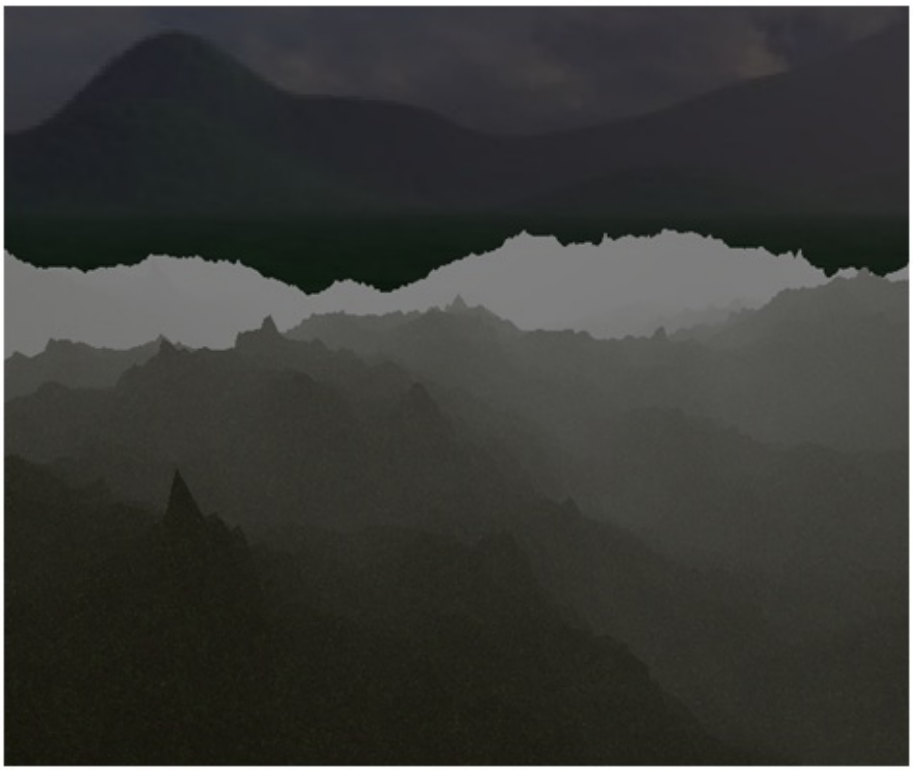


You will see that distant objects get faded in the distance and that fog starts to disappear when you approach to them. There's a problem, though with the skybox, it looks a little bit weird that the horizon is not affected by the fog. There are several ways to solve this:

- Use a different skybox in which you only see a sky.
- Remove the skybox, since you have a dense fog, you should not be able to see a background.

Maybe none of the two solutions fits you, and you can try to match the fog colour to the skybox background but you will end up doing complex calculations and the result will not be much better.

If you let the example run you will see how directional light gets dimmed and the scene darkens, but there's a problem with the fog, it is not affected by light and you will get something like this.



Distant objects are set to the fog colour which is a constant and not affected by light. This fact produces like a glowing in the dark effect (which may be ok for you or not). We need to change the funcion that calculates the fog to take into consideration the light. The function will receive the ambient light and the directional light to modulate the fog colour.

```
vec4 calcFog(vec3 pos, vec4 colour, Fog fog, vec3 ambientLight, DirectionalLight dirLi
ght)
{
    vec3 fogColor = fog.colour * (ambientLight + dirLight.colour * dirLight.intensity)
;
    float distance = length(pos);
    float fogFactor = 1.0 / exp( (distance * fog.density)* (distance * fog.density));
    fogFactor = clamp( fogFactor, 0.0, 1.0 );

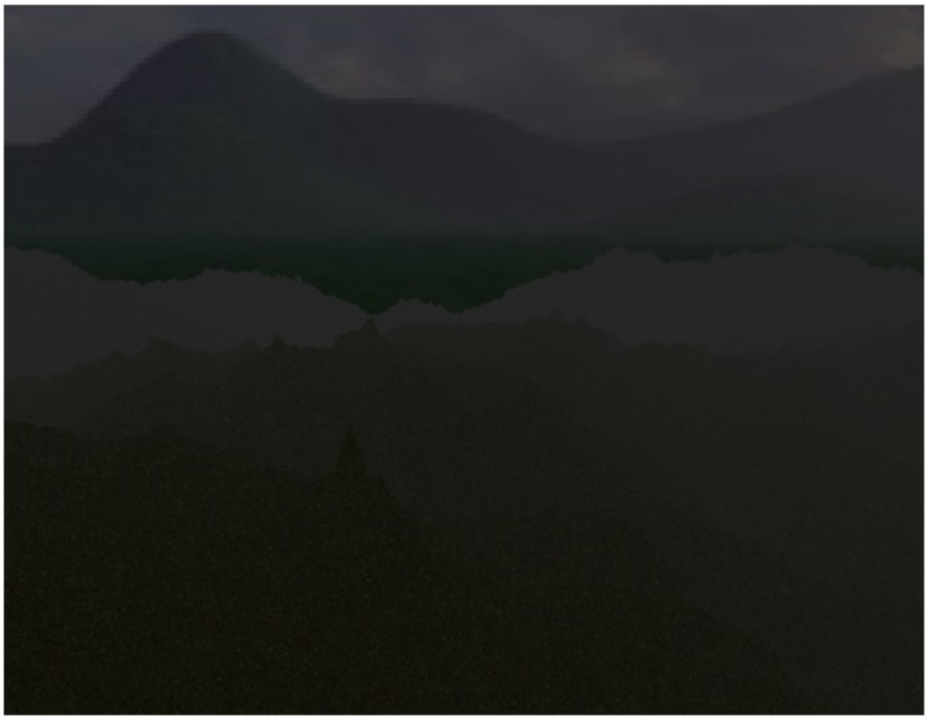
    vec3 resultColour = mix(fogColor, colour.xyz, fogFactor);
    return vec4(resultColour.xyz, 1);
}
```

As you can see with the directional light we just use the colour and the intensity, we are not interested in the direction. With that modification we just need to slightly modify the call to the function like this:

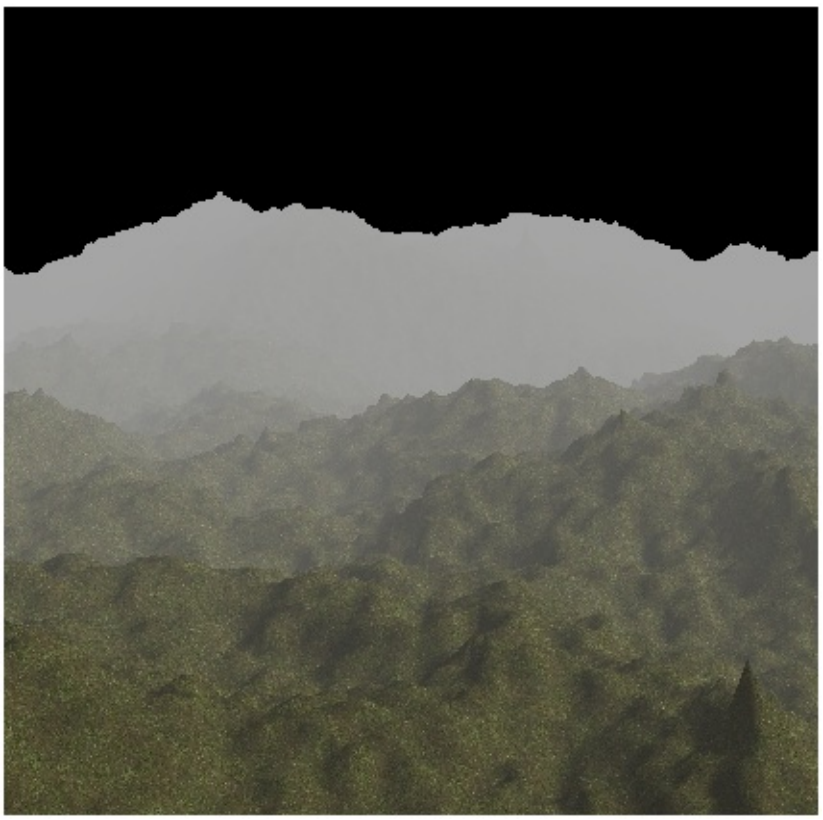
```
if ( fog.active == 1 )
{
    fragColor = calcFog(mvVertexPos, fragColor, fog, ambientLight, directionalLight);
}
```


www.dbooks.org

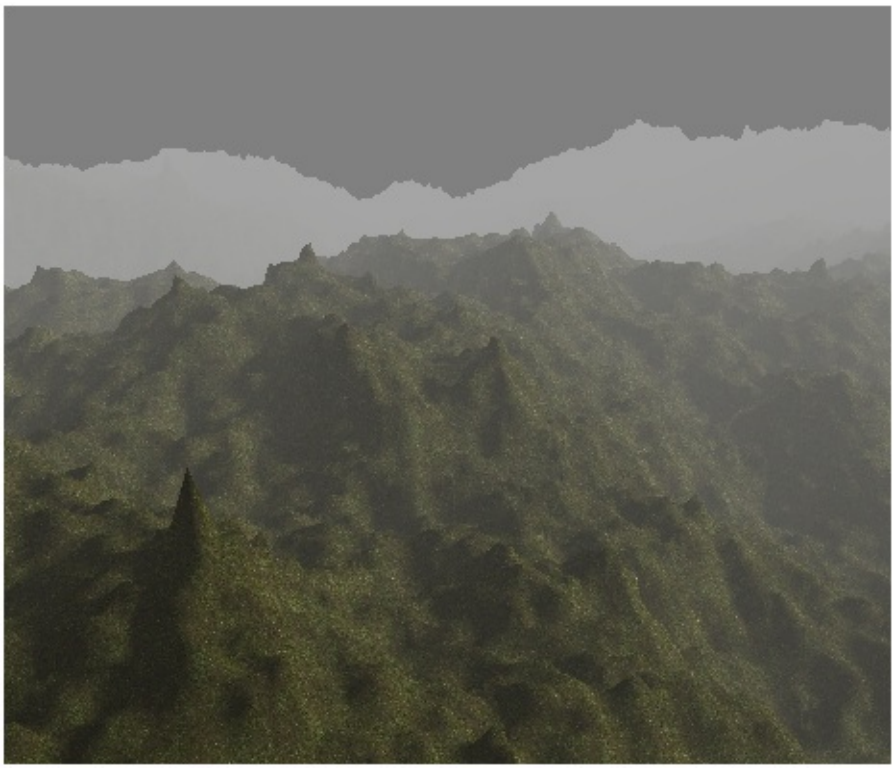
And we will get something like this when the night falls.



One important thing to highlight is that we must wisely choose the fog colour. This is even more important when we have no skybox but a fixed colour background. We should set up the fog colour to be equal to the clear colour. If you uncomment the code that render the skybox and rerun the example you will get something like this.



But if we modify the clear colour to be equal to `(0.5, 0.5, 0.5)` the result will be like this.



www.dbooks.org

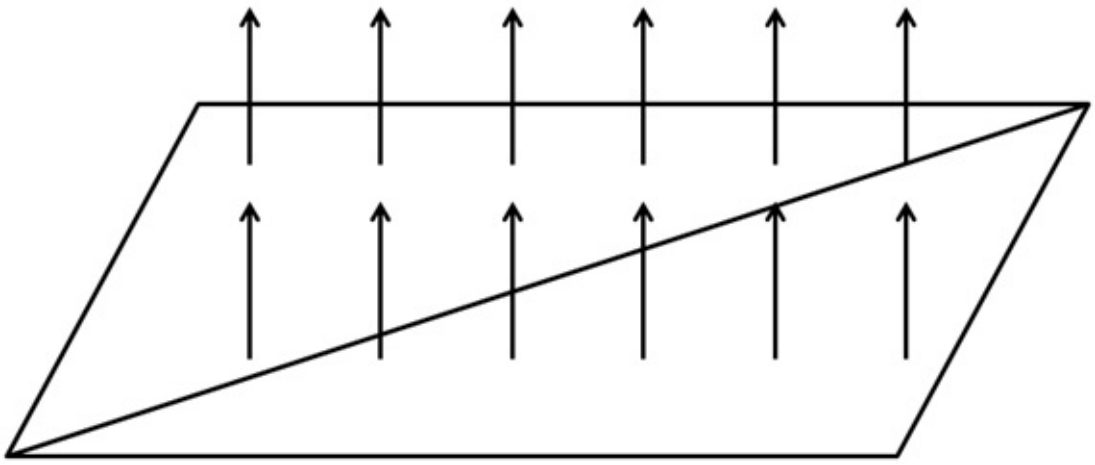
Normal Mapping

In this chapter we will explain a technique that will dramatically improve how our 3D models look like. By now, we are now able to apply textures to complex 3D models, but we are still far away from what real objects look like. Surfaces in the real world are not perfectly plain, they have imperfections that our 3D models currently do not have.

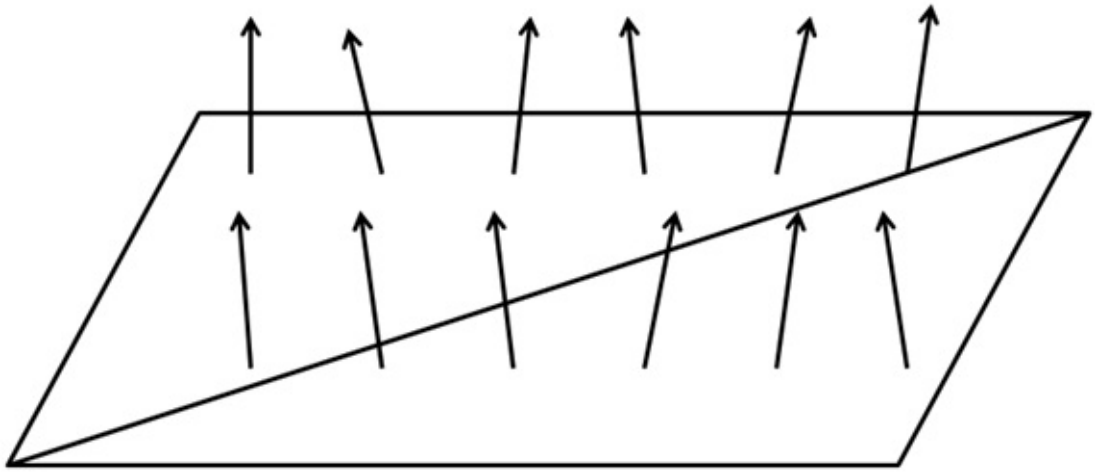
In order to render more realistic scenes we are going to use normal maps. If you look at a flat surface in the real word you will see that those imperfections can be seen even at distance by the way that the light reflects on it. In a 3D scene a flat surface will have no imperfections, we can apply a texture to it but we won't change the way that light reflects on it. That's the thing that makes the difference.

We may think in increasing the detail of our models by increasing the number of triangles and reflect those imperfections but performance will degrade. What we need is a way to change the way light reflects on surfaces to increase the realism. This is achieved with the normal mapping technique.

Let's go back to the plain surface example, a plane cane be defined by two triangles which form a quad. If you remember form the lightning chapters, the element that models how light reflects are surface normals. In this case, we have a single normal for the whole surface, each fragment of the surface uses the same normal when calculating how light affects them. This is shown in the next figure.



If we could change the normals for each fragment of the surface we could model surface imperfections to render them in a more realistic way. This is shown in the next figure.

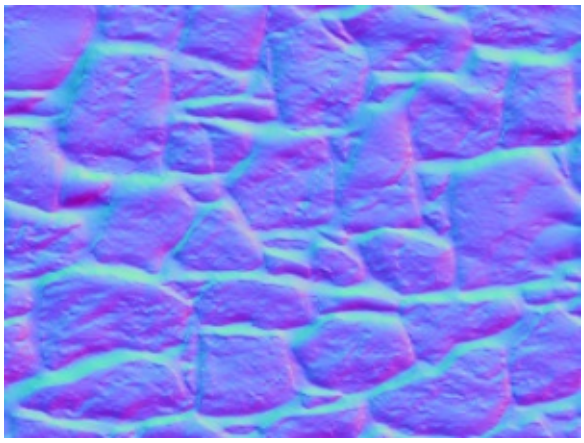


The way we are going to achieve this is by loading another texture which stores the normals for the surface. Each pixel of the normal texture will contain the values of the x, y and z coordinates of the normal stored as an RGB value.

Let's use the following texture to draw a quad.



An example of a normal map texture for the image above could be the following.



As you can see is if like we had applied a colour transformation to the original texture. Each pixel stores normal information using colour components. One thing that you will usually see when viewing normal maps is that the dominant colours tend to blue. This is due to the fact

www.dbooks.org

that normals point to the positive z axis. The z component will usually have a much higher value than the x and y ones for plain surfaces as the normal points out of the surface. Since x, y, z coordinates are mapped to RGB, the blue component will have also a higher value.

So, to render an object using normal maps we just need an extra texture and use it while rendering fragments to get the appropriate normal value.

Let's start changing our code in order to support normal maps. We will add a new texture instance to the `Material` class so we can attach a normal map texture to our game items. This instance will have its own getters and setters and method to check if the material has a normal map or not.

```
public class Material {

    private static final Vector4f DEFAULT_COLOUR = new Vector3f(1.0f, 1.0f, 1.0f, 10.f
);

    private Vector3f ambientColour;

    private Vector3f diffuseColour;

    private Vector3f specularColour;

    private float reflectance;

    private Texture texture;

    private Texture normalMap;

    // … Previous code here

    public boolean hasNormalMap() {
        return this.normalMap != null;
    }

    public Texture getNormalMap() {
        return normalMap;
    }

    public void setNormalMap(Texture normalMap) {
        this.normalMap = normalMap;
    }
}
```

We will use the normal map texture in the scene fragment shader. But, since we are working in view coordinates space we need to pass the model view matrix in order to do the proper transformation. Thus, we need to modify the scene vertex shader.

```
#version 330

layout (location=0) in vec3 position;
layout (location=1) in vec2 texCoord;
layout (location=2) in vec3 vertexNormal;

out vec2 outTexCoord;
out vec3 mvVertexNormal;
out vec3 mvVertexPos;
out mat4 outModelViewMatrix;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;

void main()
{
    vec4 mvPos = modelViewMatrix * vec4(position, 1.0);
    gl_Position = projectionMatrix * mvPos;
    outTexCoord = texCoord;
    mvVertexNormal = normalize(modelViewMatrix * vec4(vertexNormal, 0.0)).xyz;
    mvVertexPos = mvPos.xyz;
    outModelViewMatrix = modelViewMatrix;
}
```

In the scene fragment shader we need to add another input parameter.

```
in mat4 outModelViewMatrix;
```

In the fragment shader, we will need to pass a new uniform for the normal map texture sampler:

```
uniform sampler2D texture_sampler;
```

Also, in the fragment shader, we will create a new function that calculates the normal for the current fragment.

```
vec3 calcNormal(Material material, vec3 normal, vec2 text_coord, mat4 modelViewMatrix)
{
    vec3 newNormal = normal;
    if ( material.hasNormalMap == 1 )
    {
        newNormal = texture(normalMap, text_coord).rgb;
        newNormal = normalize(newNormal * 2 - 1);
        newNormal = normalize(modelViewMatrix * vec4(newNormal, 0.0)).xyz;
    }
    return newNormal;
}
```


www.dbooks.org

The function takes the following parameters:

- The material instance.
- The vertex normal.
- The texture coordinates.
- The model view matrix.

The first thing we do in that function is to check if this material has a normal map associated or not. If not, we just simply use the vertex normal as usual. If it has a normal map, we use the normal data stored in the normal texture map associated to the current texture coordinates.

Remember that the colour we get are the normal coordinates, but since they are stored as RGB values they are contained in the range [0, 1]. We need to transform them to be in the range [-1, 1], so we just multiply by two and subtract 1 . Then, we normalize that value and transform it to view model coordinate space (as with the vertex normal).

And that's all, we can use the returned value as the normal for that fragment in all the lightning calculations.

In the `Renderer` class we need to create the normal map uniform, and in the `renderScene` method we need to set it up like this:

```
...
sceneShaderProgram.setUniform("fog", scene.getFog());
sceneShaderProgram.setUniform("texture_sampler", 0);
sceneShaderProgram.setUniform("normalMap", 1);
...
```

You may notice some interesting thing in the code above. We are setting 0 for the material texture uniform (`texture_sampler`) and 1 for the normal map texture (`normalMap`). If you recall from the texture chapter. We are using more than one texture, so we must set up the texture unit for each separate texture.

We need to take this also into consideration when we are rendering a `Mesh` .

```
private void initRender() {
    Texture texture = material.getTexture();
    if (texture != null) {
        // Activate first texture bank
        glActiveTexture(GL_TEXTURE0);
        // Bind the texture
        glBindTexture(GL_TEXTURE_2D, texture.getId());
    }
    Texture normalMap = material.getNormalMap();
    if ( normalMap != null ) {
        // Activate first texture bank
        glActiveTexture(GL_TEXTURE1);
        // Bind the texture
        glBindTexture(GL_TEXTURE_2D, normalMap.getId());
    }

    // Draw the mesh
    glBindVertexArray(getVaoId());
    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);
    glEnableVertexAttribArray(2);
}
```

As you can see we need to bind to each of the textures available and activate the associated texture unit in order to be able to work with more than one texture. In the `renderScene` method in the `Renderer` class we do not need to explicitly set up the uniform of the texture since it's already contained in the `Material`.

In order to show the improvements that normal maps provide, we have created an example that shows two quads side by side. The right quad has a texture map applied and the left one not. We also have removed the terrain, the skybox and the HUD and setup a directional light with can be changed with the left and right cursor keys so you can see the effect. We have modified the base source code a bit in order to support not having a skybox or a terrain. We have also clamped the light effect in the fragment shader in the rang [0, 1] to avoid over exposing effect of the image.

The result is shown in the next figure.

www.dbooks.org



As you can see the quad that has a normal texture applied gives the impression of having more volume. Although it is, in essence, a plain surface like the other quad, you can see how the light reflects.
But, although the code we have set up, works perfectly with this example you need to be aware of its limitations. The code only works for normal map textures that are created using object space coordinates. If this is the case we can apply the model view matrix transformations to translate the normal coordinates to the view space.

But, usually normal maps are not defined in that way. They usually are defined in the called tangent space. The tangent space is a coordinate system that is local to each triangle of the model. In that coordinate space the z axis always points out of the surface. This is the reason why when you look at a normal map its usually bluish, even for complex models with opposing faces.

We will stick with this simple implementation by now, but keep in mind that you must always use normal maps defined in object space. If you use maps defined in tangent space you will get weird results. In order to be able to work with them we need to setup specific matrices to transform coordinates to the tangent space.

Shadows

Shadow Mapping

Currently we are able to represent how light affects the objects in a 3D scene. Objects that get more light are shown brighter then objects that do not receive light. However we are still not able to cast shadows. Shadows will increase the degree of realism that 3D scene would have so we will add support for it in this chapter.

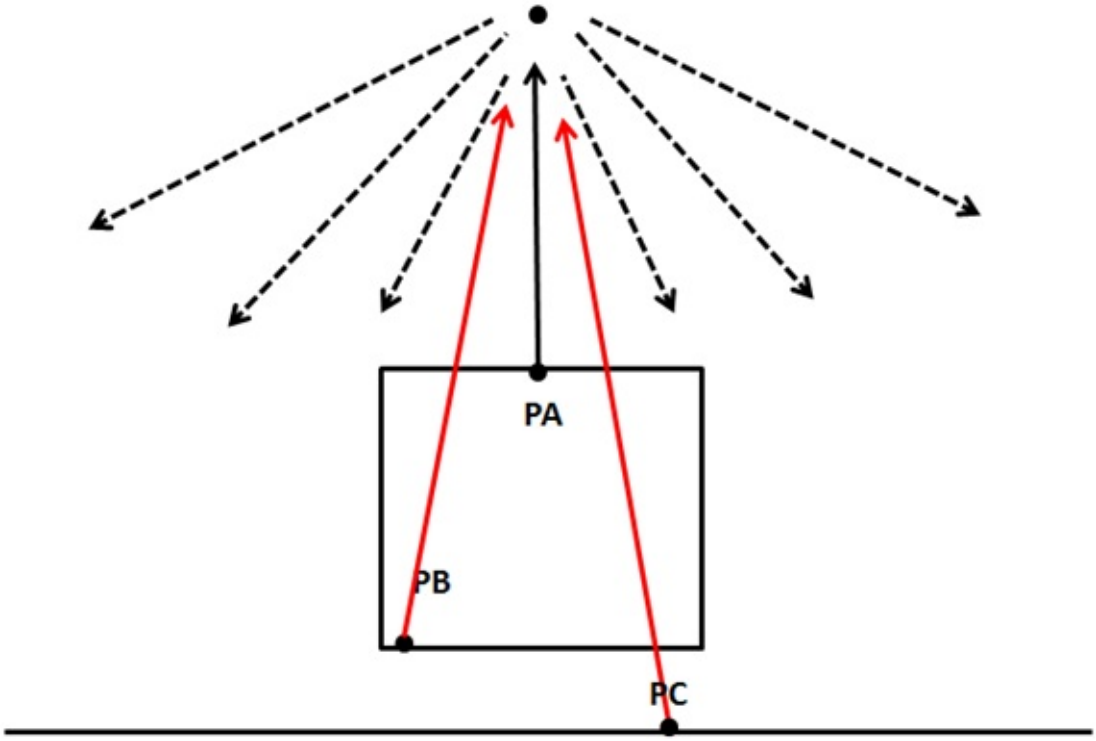
We will use a technique named Shadow mapping which is widely used in games and does not affect severely the engine performance. Shadow mapping may seem simple to understand but it’s somehow difficult to implement it right. Or, to be more precise, it’s very difficult to implement it in a generic ways that cover all the potential cases and produces consistent results.

We will explain here an approach which will serve you to add shadows for most of the cases, but what it’s more important it will serve you to understand its limitations. The code presented here is far from being perfect but I think it will be easy to understand. It is also designed to support directional lights (which in my opinion is the more complex case) but you will learn how it can be extended to support other type of lights (such us point lights). If you want to achieve more advanced results you should use more advance techniques such as Cascaded Shadow Maps. In any case the concepts explained here will serve you as a basis.

So let’s start by thinking in how we could check if a specific area (indeed a fragment) is in shadow or not. While drawing that area if we can cast rays to the light source, if we can reach the light source without any collision then that pixel is in light. If not, then the pixel is in shadow.

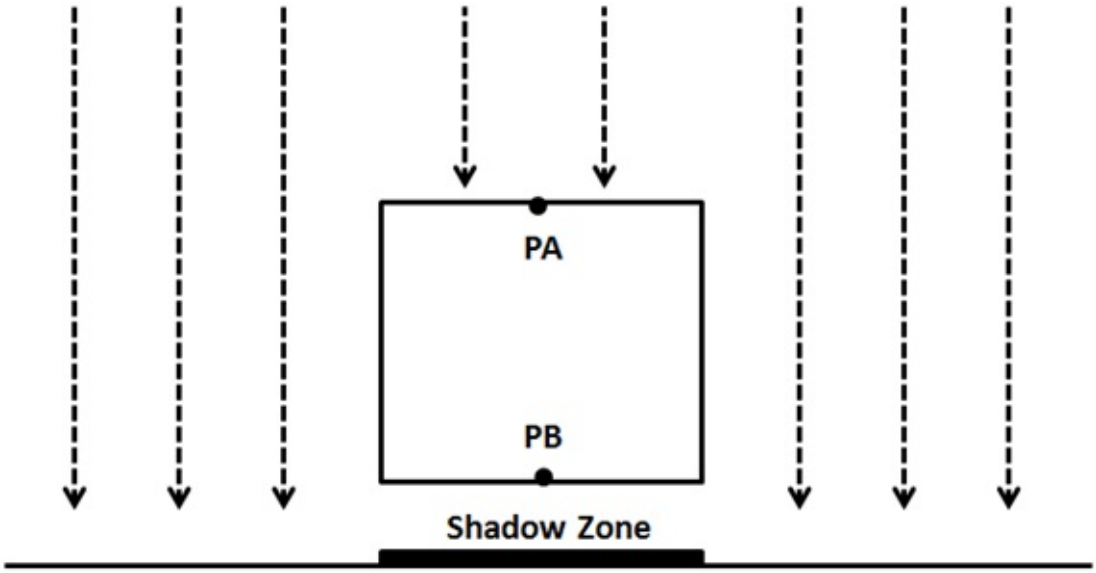
The following picture shows the case for a point light, the point PA can reach the source light, but points PB and PC can’t so they are in shadow.

www.dbooks.org



So, how we can check if we can cast that ray without collisions in an efficient manner? A light source can, theoretically cast infinitely ray lights, so how do we check if a ray light is blocked or not ?
What we can do instead of casting ray lights is to look at the 3D scene from the light's persèctive and render the scene from that location. We can set the camera at the light position and render the scene so we can store the depth for each fragment. This is equivalent to calculate the distance of each fragment to the light source. At the end, what we are doing is storing the minimum distance as seen from the light source as a shadow map.

The following picture shows a cube floating over a plane and a perpendicular light.



The scene as seen from the light perspective would be something like this (the darker the colour, the closer to the light source).



With that information we can render the 3D scene as usual and check the distance for each fragment to the light source with the minimum stored distance. If the distance is less that the value stored in the shadow map, then the object is in light, otherwise is in shadow. We can have several objects that could be hit by the same ray light. But we store the minimum distance.

Thus, shadow mapping is a two step process:

- First we render the scene from the light space into a shadow map to get the minimum distances.
- Second we render the scene from the camera point of view and use that depth map to calculate if objects are in shadow or not.

In order to render the depth map we need to talk about the depth buffer. When we render a scene all the depth information is stored in a buffer named, obviously, depth-buffer (also z-buffer). That depth information is the z value of each of the fragment that is rendered. If you recall from the first chapters what we are doing while rendering a scene is transforming from world coordinates to screen coordinates. We are drawing to a coordinate space which ranges from 0 to 1 for x and y axis. If an object is more distant than other, we must calculate how this affects their x and y coordinates through the perspective projection matrix. This is not calculated automatically depending on the z value, this must be done us. What is actually stored in the z coordinate its the depth of that fragment, nothing less but nothing more.

Besides that, in our source code we are enabling depth testing. In the Window class we have set the following line:

```
glEnable(GL_DEPTH_TEST);
```

By setting this line we prevent fragments that cannot be seen, because they are behind other objects, to be drawn. Before a fragment is drawn its z value is compared with the z value of the z-buffer. If it has a higher z value (it's far away) than the z value of the buffer it's discarded. Remember that this is done in screen space, so we are comparing the z value of a fragment given a pair of x and y coordinates in screen space, that is in the range $[0, 1]$. Thus, the z value is also in that range.

www.dbooks.org

The presence of the depth buffer is the reason why we need to clear the screen before performing any render operation. We need to clear not only the colour but the depth information also:

```
public void clear() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}
```

In order to start building the depth map we want to get that depth information as viewed from the light perspective. We need to setup a camera in the light position, render the scene and store that depth information into a texture so we can access to it later.

Therefore, the first thing we need to do is add support for creating those textures. We will modify the `Texture` class to support the creation of empty textures by adding a new constructor. This constructor expects the dimensions of the texture and the format of the pixels it stores.

```
public Texture(int width, int height, int pixelFormat) throws Exception {
    this.id = glGenTextures();
    this.width = width;
    this.height = height;
    glBindTexture(GL_TEXTURE_2D, this.id);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, this.width, this.height, 0, pix
elFormat, GL_FLOAT, (ByteBuffer) null);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
}
```

We set the texture wrapping mode to `GL_CLAMP_TO_EDGE` since we do not want the texture to repeat in case we exceed the $[0, 1]$ range.

So now that we are able to create empty textures, we need to be able to render a scene into it. In order to do that we need to use Frame Buffers Objects (or FBOs). A Frame Buffer is a collection of buffers that can be used as a destination for rendering. When we have been rendering to the screen we have using OpenGL's default buffer. OpenGL allows us to render to user defined buffers by using FBOs. We will isolate the rest of the code of the process of creating FBOs for shadow mapping by creating a new class named `ShadowMap`. This is the definition of that class.

```
package org.lwjglb.engine.graph;

import static org.lwjgl.opengl.GL11.*;
import static org.lwjgl.opengl.GL30.*;
```

```
public class ShadowMap {

    public static final int SHADOW_MAP_WIDTH = 1024;

    public static final int SHADOW_MAP_HEIGHT = 1024;

    private final int depthMapFBO;

    private final Texture depthMap;

    public ShadowMap() throws Exception {
        // Create a FBO to render the depth map
        depthMapFBO = glGenFramebuffers();

        // Create the depth map texture
        depthMap = new Texture(SHADOW_MAP_WIDTH, SHADOW_MAP_HEIGHT, GL_DEPTH_COMPONENT
);

        // Attach the the depth map texture to the FBO
        glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, dep
thMap.getId(), 0);
        // Set only depth
        glDrawBuffer(GL_NONE);
        glReadBuffer(GL_NONE);

        if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE) {
            throw new Exception("Could not create FrameBuffer");
        }

        // Unbind
        glBindFramebuffer(GL_FRAMEBUFFER, 0);
    }

    public Texture getDepthMapTexture() {
        return depthMap;
    }

    public int getDepthMapFBO() {
        return depthMapFBO;
    }

    public void cleanup() {
        glDeleteFramebuffers(depthMapFBO);
        depthMap.cleanup();
    }
}
```


www.dbooks.org

The `ShadowMap` class defines two constants that determine the size of the texture that will hold the depth map. It also defines two attributes, one for the FBO and one for the texture. In the constructor, we create a new FBO and a new `Texture`. For the FBO we will use as the pixel format the constant `GL_DEPTH_COMPONENT` since we are only interested in storing depth values. Then we attach the FBO to the texture instance.

The following lines explicitly set the FBO to not render any colour. A FBO needs a colour buffer, but we are not going to needed. This is why we set the colour buffers to be used as `GL_NONE`.

```
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
```

Now we are ready to render the scene from the light perspective into FBO in the `Renderer` class. In order to do that, we will create a specific set of vertex and fragments shaders.

The vertex shader, named `depth_vertex.fs`, is defined like this.

```
#version 330

layout (location=0) in vec3 position;
layout (location=1) in vec2 texCoord;
layout (location=2) in vec3 vertexNormal;

uniform mat4 modelLightViewMatrix;
uniform mat4 orthoProjectionMatrix;

void main()
{
    gl_Position = orthoProjectionMatrix * modelLightViewMatrix * vec4(position, 1.0f);
}
```

We expect to receive the same input data as the scene shader. In fact, we only need the position, but to reuse as much as code as possible we will pass it anyway. We also need a pair of matrices. Remember that we must render the scene from the light point of view, so we need to transform our models to light's coordinate space. This is done through the `modelLightViewMatrix` matrix, which is analogous to view model matrix used for a camera. The light is our camera now.

Then we need to transform those coordinates to screen space, that is, we need to project them. And this is one of the differences while calculating shadow maps for directional lights versus point lights. For point lights we would use a perspective projection matrix as if we were rendering the scene normally. Directional lights, instead, affect all objects in the same

way independently of the distance. Directional lights are located at an infinite point and do not have a position but a direction. An orthographic projection does not render distant objects smaller, and because of this characteristic is the most suitable for directional lights.

The fragment shader is even simpler. It just outputs the z coordinate as the depth value.

```
#version 330

void main()
{
    gl_FragDepth = gl_FragCoord.z;
}
```

In fact, you can remove that line, since we are only generating depth values, the depth value it will be automatically returned.

Once we have defined the new shaders for depth rendering we can use them in the `Renderer` class. We define a new method for setting up those shaders, named `setupDepthShader`, which will be invoked where the others shaders are initialized.

```
private void setupDepthShader() throws Exception {
    depthShaderProgram = new ShaderProgram();
    depthShaderProgram.createVertexShader(Utils.loadResource("/shaders/depth_vertex.vs"
));
    depthShaderProgram.createFragmentShader(Utils.loadResource("/shaders/depth_fragmen
t.fs"));
    depthShaderProgram.link();

    depthShaderProgram.createUniform("orthoProjectionMatrix");
    depthShaderProgram.createUniform("modelLightViewMatrix");
}
```

Now we need to create a new method that uses those shaders which will be named `renderDepthMap`. This method will be invoked in the principal render method.

```
public void render(Window window, Camera camera, Scene scene, IHud hud) {
    clear();

    // Render depth map before view ports has been set up
    renderDepthMap(window, camera, scene);

    glViewport(0, 0, window.getWidth(), window.getHeight());

    // Rest of the code here ....
```


www.dbooks.org

If you look at the above code you will see that the new method is invoked at the very beginning, before we have set the view port. This is due to the fact that this new method will change the view port to match the dimensions of the texture that holds the depth map. Because of that, we will always need to set, after the `renderDepthMap` has been finished, the view port to the screen dimensions (without checking if the window has been resized).

Let's define now the `renderDepthMap` method. The first thing that we will do is to bind to the FBO we have created in the `ShadowMap` class and set the view port to match the texture dimensions.

```
glBindFramebuffer(GL_FRAMEBUFFER, shadowMap.getDepthMapFBO());
glViewport(0, 0, ShadowMap.SHADOW_MAP_WIDTH, ShadowMap.SHADOW_MAP_HEIGHT);
```

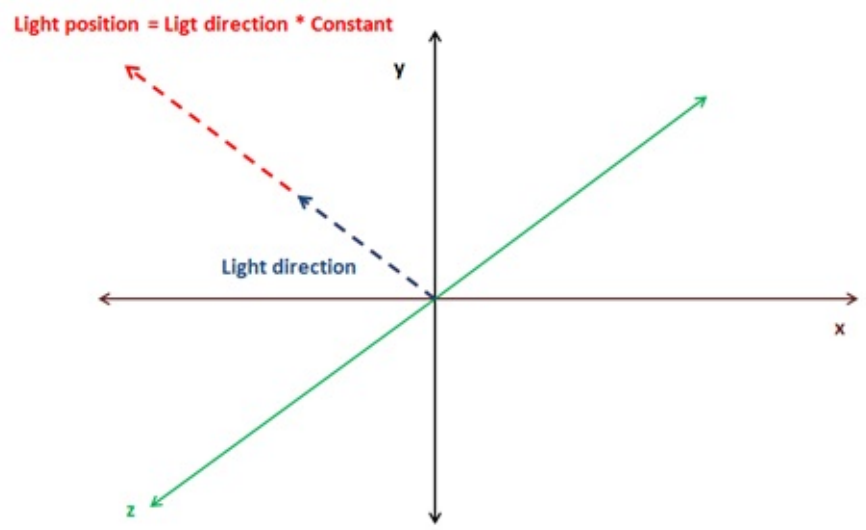
Then we clear the depth buffer contents and bind the depth shaders. Since we are only dealing with depth values we do not need to clear colour information.

```
glClear(GL_DEPTH_BUFFER_BIT);

depthShaderProgram.bind();
```

Now we need to setup the matrices, and here comes the tricky part. We use the light as a camera so we need to create a view matrix which needs a position and three angles. As it has been said at the beginning of the chapter we will support only directional lights, and that type of lights does not define a position but a direction. If we were using point lights this would be easy, the position of the light would be the position of the view matrix, but we do not have that.

We will take a simple approach to calculate the light position. Directional lights are defined by a vector, usually, normalized, which points to the direction where the light is. We will multiply that direction vector by a configurable factor so it defines a point at a reasonable distance for the scene we want to draw. We will use that direction in order to calculate the rotation angle for that view matrix.



This is the fragment that calculates the light position and the rotation angles

```
float lightAngleX = (float)Math.toDegrees(Math.acos(lightDirection.z));
float lightAngleY = (float)Math.toDegrees(Math.asin(lightDirection.x));
float lightAngleZ = 0;
Matrix4f lightViewMatrix = transformation.updateLightViewMatrix(new Vector3f(lightDire
ction).mul(light.getShadowPosMult()), new Vector3f(lightAngleX, lightAngleY, lightAngl
eZ));
```

Next we need to calculate the orthographic projection matrix.

```
Matrix4f orthoProjMatrix = transformation.updateOrthoProjectionMatrix(orthCoords.left,
 orthCoords.right, orthCoords.bottom, orthCoords.top, orthCoords.near, orthCoords.far)
;
```

We have modified the `Transformation` class to include the light view matrix and the orthographic projection matrix. Previously we had a orthographic 2D projection matrix, so we have renamed the previous methods and attributes. You can check the definition in the source code which is straight forward.

Then we render the scene objects as in the `renderScene` method but using the previous matrices to work in light space coordinate system.

www.dbooks.org

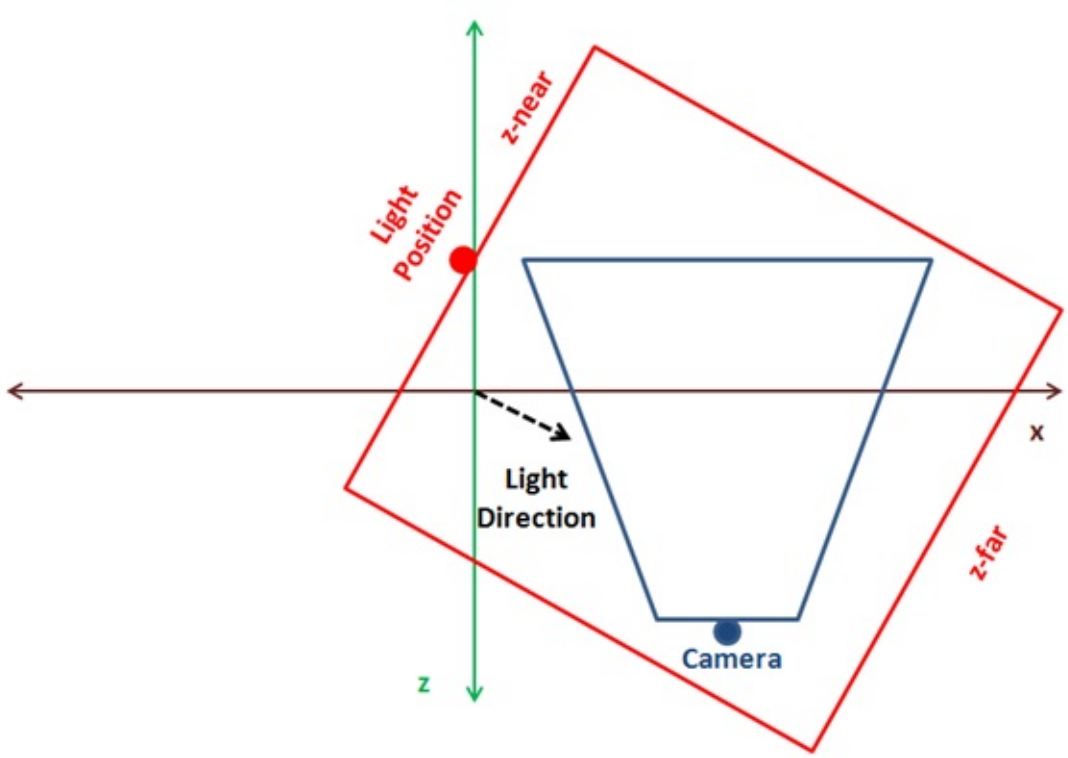
```
depthShaderProgram.setUniform("orthoProjectionMatrix", orthoProjMatrix);
Map<Mesh, List<GameItem>> mapMeshes = scene.getGameMeshes();
for (Mesh mesh : mapMeshes.keySet()) {
    mesh.renderList(mapMeshes.get(mesh), (GameItem gameItem) -> {
        Matrix4f modelLightViewMatrix = transformation.buildModelViewMatrix(gameItem,
lightViewMatrix);
        depthShaderProgram.setUniform("modelLightViewMatrix", modelLightViewMatrix);
    }
    );
}

// Unbind
depthShaderProgram.unbind();
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

The parameterization of the orthographic projection matrix is defined in the directional Light. Think of the orthographic projection matrix as a bounding box that contains all the objects that we want to render. When projecting only the objects that fit into that bounding box will be be visible. That bounding box is defined by 6 parameters: left, right, bottom, top, near, far. Since, the light position is now the origin, these parameters define the distance from that origin to the left or right (x-axis) up or down (y-axis) and to the nearest or farthest plane (z-axis).

One of the trickiest points in getting shadows map to work is determine the light position and the orthographic projection matrix parameters. This is way all these parameters are now defined in the `DirectionalLight` class so it can be set properly according to each scene.

You can implement a more automatic approach, by calculating the centre of the camera frustum, get back in the light direction and build a orthographic projection that contains all the objects in the scene. The following figure shows a 3D scene as looked form above, the camera position and its frustum (in blue) and the optimal light position and bounding box in red.



The problem with the approach above is that is difficult to calculate and if you have small objects and the bounding box is big you may get strange results. The approach presented here is simpler for small scenes and you can tweak it to match your models (even you can chose to explicitly set light's position to avoid strange effects if camera moves far away from the origin). If you want a more generic model that can be applied to any scene you should extend it to support cascading shadow maps.

Let's continue. Before we use the depth maps to actually calculate shadows, you could render a quad with the generated texture to see how a real depth map looks like. You could get something like this for a scene composed by a rotating cube floating over a plane with a perpendicular directional light.



www.dbooks.org

As it's been said before, the darker the colour, the closer to the light position. What's the effect of the light position in the depth map? You can play with the multiplication factor of the directional light and you will see that the size of the objects rendered in the texture do not decrease. Remember that we are using an orthographic projection matrix and objects do not get smaller with distance. What you will see is that all colours get brighter as seen in the next picture.



Does that mean that we can choose a high distance for the light position without consequences? The answer is no. If light is too far away from the objects we want to render, these objects can be out of the bounding box that defines the orthographic projection matrix. In this case you will get a nice white texture which would be useless for shadow mapping. Ok, then we simply increase the bounding box size and everything will be ok, right? The answer is again no. If you chose huge dimensions for the orthographic projection matrix your objects will be drawn very small in the texture, and the depth values can even overlap leading to strange results. Ok, so you can think in increasing the texture size, but, again in this case you are limited and textures cannot grow indefinitely to use huge bounding boxes.

So as you can see selecting the light position and the orthographic projection parameters is a complex equilibrium which makes difficult to get right results using shadow mapping.

Let's go back to the rendering process, once we have calculated the depth map we can use it while rendering the scene. First we need to modify the scene vertex shader. Up to now, the vertex shader projected the vertex coordinates from model view space to the screen space using a perspective matrix. Now we need to project also the vertex coordinates from light space coordinates using a projection matrix to be used in the fragment shader to calculate the shadows.

The vertex shader is modified like this.

```
#version 330

layout (location=0) in vec3 position;
layout (location=1) in vec2 texCoord;
layout (location=2) in vec3 vertexNormal;

out vec2 outTexCoord;
out vec3 mvVertexNormal;
out vec3 mvVertexPos;
out vec4 mlightviewVertexPos;
out mat4 outModelViewMatrix;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform mat4 modelLightViewMatrix;
uniform mat4 orthoProjectionMatrix;

void main()
{
    vec4 mvPos = modelViewMatrix * vec4(position, 1.0);
    gl_Position = projectionMatrix * mvPos;
    outTexCoord = texCoord;
    mvVertexNormal = normalize(modelViewMatrix * vec4(vertexNormal, 0.0)).xyz;
    mvVertexPos = mvPos.xyz;
    mlightviewVertexPos = orthoProjectionMatrix * modelLightViewMatrix * vec4(position
, 1.0);
    outModelViewMatrix = modelViewMatrix;
}
```

We use new uniforms for the light view matrix and the orthographic projection matrix.

In the fragment shader we will create a new function to calculate the shadows that is defined like this.

```
float calcShadow(vec4 position)
{
    float shadowFactor = 1.0;
    vec3 projCoords = position.xyz;
    // Transform from screen coordinates to texture coordinates
    projCoords = projCoords * 0.5 + 0.5;
    if ( projCoords.z < texture(shadowMap, projCoords.xy).r )
    {
        // Current fragment is not in shade
        shadowFactor = 0;
    }

    return 1 - shadowFactor;
}
```


www.dbooks.org

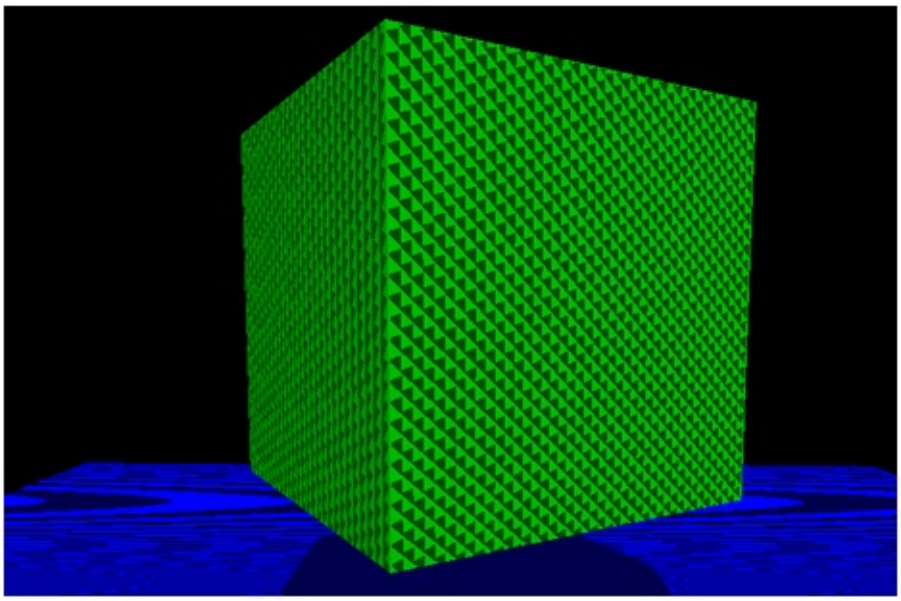
The function receives the position in light view space projected using the orthographic projection matrix. It returns 0 if the position is in shadow and 1 if it's not. First, the coordinates are transformed to texture coordinates. Screen coordinates are in the range $[-1, 1]$, but texture coordinates are in the range $[0, 1]$. With that coordinates we get the depth value from the texture and compare it with the z value of the fragment coordinates. If the z value if the fragment has a lower value than the one stored in the texture that means that the fragment is not in shade.

In the fragment shader, the return value from the `calcShadow` function to modulate the light colour contributions from point, spot and directional lights. The ambient light is not affected by the shadow.

```
float shadow = calcShadow(mlightviewVertexPos);
fragColor = clamp(ambientC * vec4(ambientLight, 1) + diffuseSpecularComp * shadow, 0, 1
);
```

In the `renderScene` method of the `Renderer` class we just need to pass the uniform for the orthographic projection and light view matrices (we need to modify also the method that initializes the shader to create the new uniforms). You can consult this in the book's source code.

If to run the `DummyGame` class, which has been modified to setup a floating cube over a plane with a directional light which angle can be changed by using up and down keys, you should see something like this.



Although shadows are working (you can check that by moving light direction), the implementation presents some problems. First of all, there are strange lines in the objects that are lightened up. This effect is called shadow acne, and it's produced by the limited

resolution of the texture that stores the depth map. The second problem is that the borders of the shadow are not smooth and look blocky. The cause is the same again, the texture resolution. We will solve these problems in order to improve shadow quality.

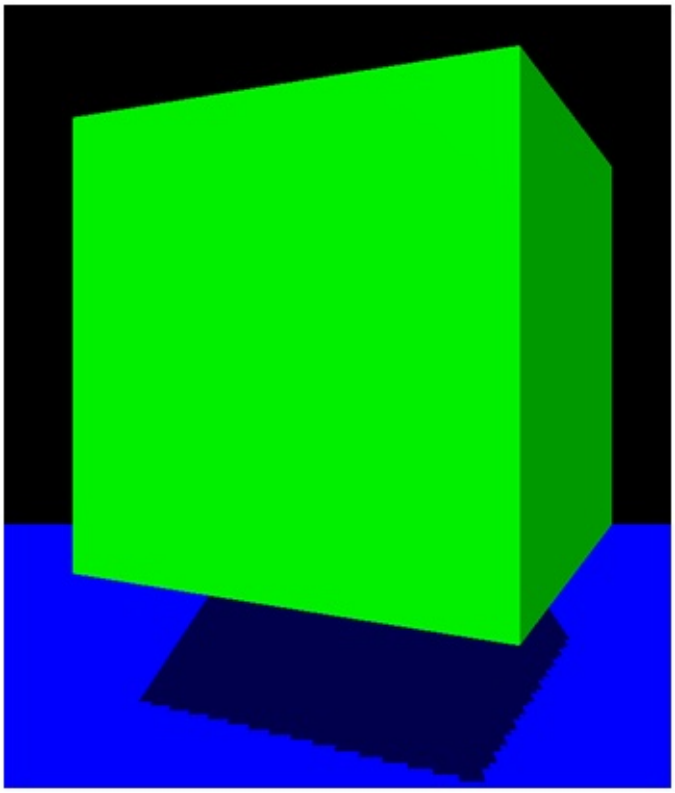
Shadow Mapping improvements

Now that we have the shadow mapping mechanism working, let's solve the problems we have. Let's first start with the shadow acne problem. The depth map texture is limited in size, and because of that, several fragments can be mapped to the same pixel in that texture depth. The texture depth stores the minimum depth, so at the end, we have several fragments that share the same depth in that texture although they are at different distances.

We can solve this by increasing, by a little bit the depth comparison in the fragment shader, we add a bias.

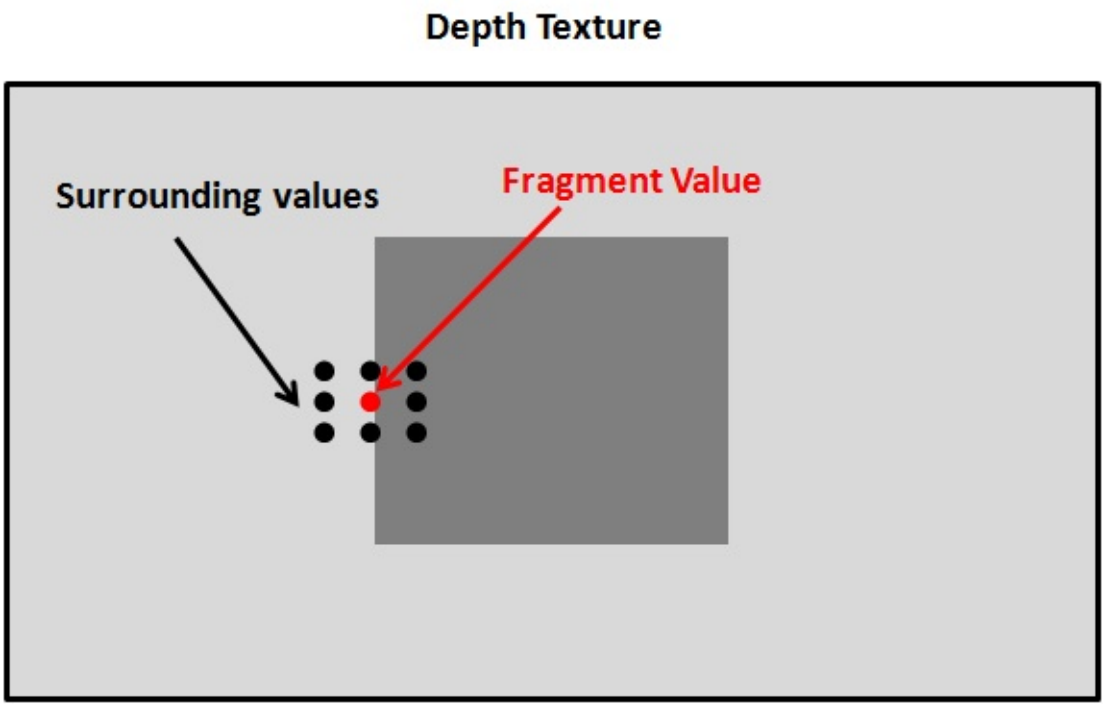
```
float bias = 0.05;
if ( projCoords.z - bias < texture(shadowMap, projCoords.xy).r )
{
    // Current fragment is not in shade
    shadowFactor = 0;
}
```

Now, the shadow acne has disappeared.



www.dbooks.org

Now we are going to solve de shadow edges problem, which is also caused by the texture resolution. For each fragment we are going to sample the depth texture with the fragment's position value and the surrounding values. Then we will calculate the average and assign that value as the shadow value. In this case his value won't be 0 or 1 but can take values in between in order to get smoother edges.

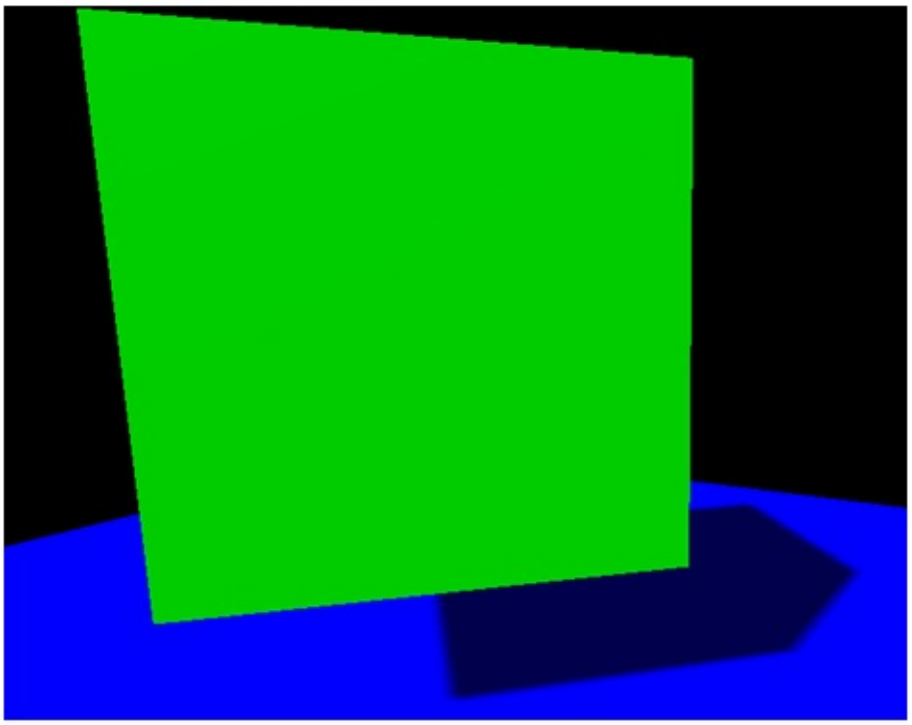


The surrounding values must be at one pixel distance of the current fragment position in texture coordinates. So we need to calculate the increment of one pixel in texture coordinates which is equal to $1/textureSize$.

In the fragment Shader we just need to modify the shadow factor calculation to get an average value.

```
float shadowFactor = 0.0;
vec2 inc = 1.0 / textureSize(shadowMap, 0);
for(int row = -1; row <= 1; ++row)
{
    for(int col = -1; col <= 1; ++col)
    {
        float textDepth = texture(shadowMap, projCoords.xy + vec2(row, col) * inc).r;
        shadowFactor += projCoords.z - bias > textDepth ? 1.0 : 0.0;
    }
}
shadowFactor /= 9.0;
```

The result looks now smoother.



Now our sample looks much better. Nevertheless, the shadow mapping technique presented here can still be improved a lot. You can check about solving peter panning effect (caused by the bias factor) and other techniques to improve the shadow edges. In any case, with the concepts explained here you have a good basis to start modifying the sample.

In order to render multiple lights you just need to render a separate depth map for each light source. While rendering the scene you will need to sample all those depth maps to calculate the appropriate shadow factor.

www.dbooks.org

Animations

Introduction

By now we have just loaded static 3D models, in this chapter we will learn how to animate them. When thinking about animations the first approach is to create different meshes for each model positions, load them up into the GPU and draw them sequentially to create the illusion of animation. Although this approach is perfect for some games it's not very efficient (in terms of memory consumption).

This where skeletal animation comes to play. In skeletal animation the way a model animates is defined by its underlying skeleton. A skeleton is defined by a hierarchy of special points called joints. Those joints are defined by their position and rotation. We have said also that it's a hierarchy, this means that the final position for each joint is affected by their parents. For instance, think on a wrist, the position of a wrist is modified if a character moves the elbow and also if it moves the shoulder.

Joints do not need to represent a physical bone or articulation, they are artifacts that allows the creatives to model an animation. In addition to joints we still have vertices, the points that define the triangles that compose a 3D model. But, in skeletal animation, vertices are drawn based on the position of the joints it is related to.

In this chapter we will use MD5 format to load animated models. MD5 format was create by ID Software, the creators of Doom, and it’s basically a text based file format which is well understood. Another approach would be to use the Collada format, which is a public standard supported by many tools. Collada is an XML based format but as a downside it’s very complex (The specification for the 1.5 version has more than 500 pages). So, we will stick to a much more simple format, MD5, that will allow us to focus in the concepts of the skeletal animation and to create a working sample.

You can also export some models from Blender to MD5 format via specific addons that you can find on the Internet (http://www.katsbits.com/tools/#md5)

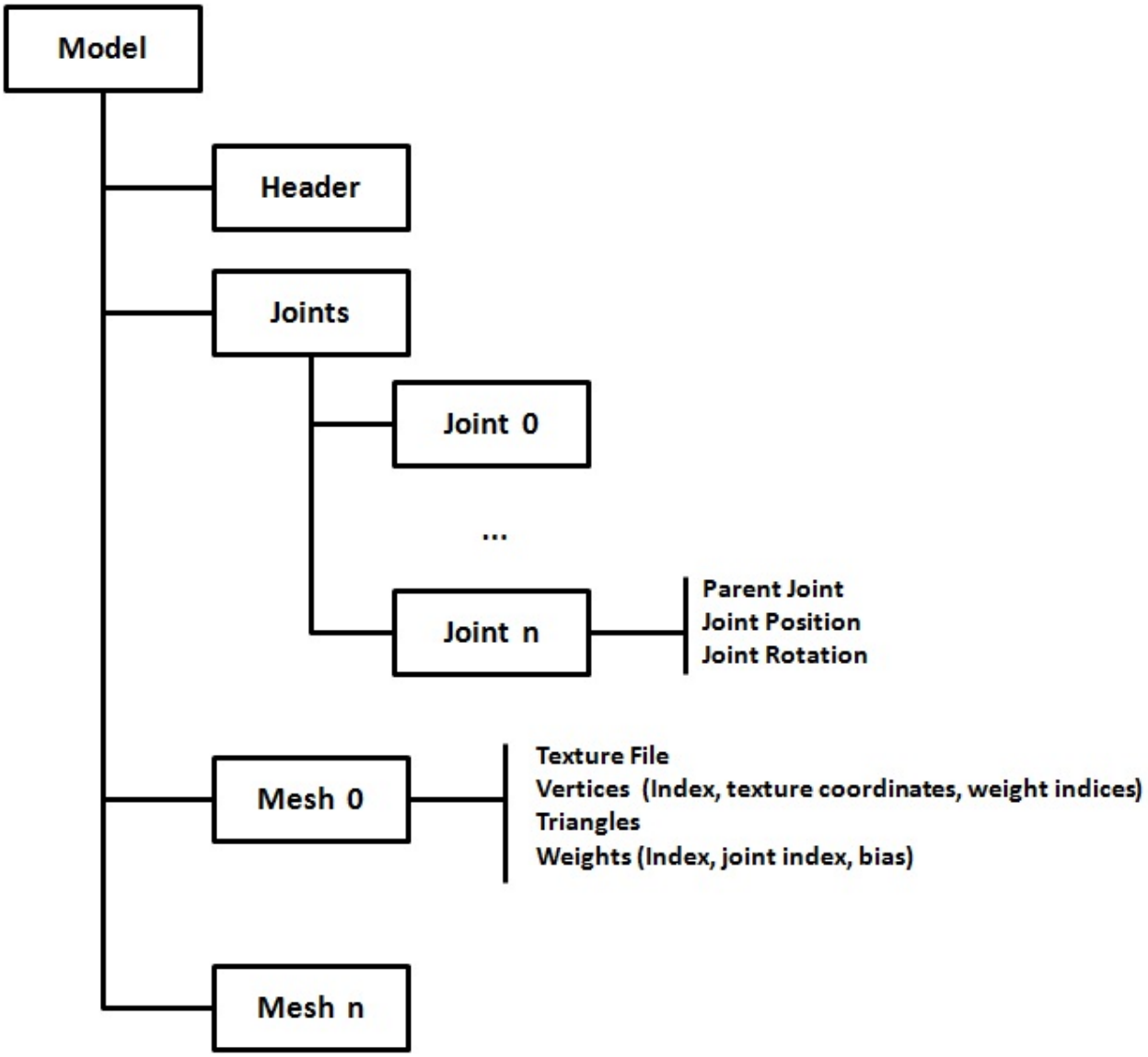
In this chapter I’ve consulted many different sources, but I have found two that provide a very good explanation about how to create an animated model using MD5 files. Theses sources can be consulted at:

- http://www.3dgep.com/gpu-skinning-of-md5-models-in-opengl-and-cg/
- http://ogldev.atspace.co.uk/www/tutorial38/tutorial38.html

So let's start by writing the code that parses MD5 files. The MD5 format defines two types of files:

- The mesh definition file: which defines the joints and the vertices and textures that compose the set of meshes that form the 3D model. This file usually has a extension named ".md5mesh".
- The animation definition file: which defines the animations that can be applied to the model. This file usually has a extension named ".md5anim".

An MD5 file is composed by a header an different sections contained between braces. Let's start examining the mesh definition file. In the resources folder you will find several models in MD5 format. If you open one of them you can see a structure similar like this.



The first structure that you can find in the mesh definition file is the header. You can see below header's content from one of the samples provided:

www.dbooks.org

```
MD5Version 10
commandline ""

numJoints 33
numMeshes 6
```

The header defines the following attributes:

- The version of the MD5 specification that it complies to.
- The command used to generate this file (from a 3D modelling tool).
- The number of Joints that are defined in the joints section
- The number of Meshes (the number of meshes sections expected).

The Joints sections defines the joints, as it names states, their positions and their relationships. A fragment of the joints section of one of the sample models is shown below.

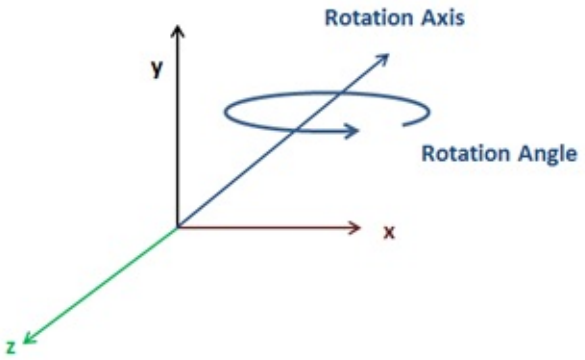
```
joints {
    "origin"    -1 ( -0.000000 0.016430 -0.006044 ) ( 0.707107 0.000000 0.707107 )
    //
    "sheath"    0 ( 11.004813 -3.177138 31.702473 ) ( 0.307041 -0.578614 0.354181 )
     // origin
    "sword"    1 ( 9.809593 -9.361549 40.753730 ) ( 0.305557 -0.578155 0.353505 )
   // sheath
    "pubis"    0 ( 0.014076 2.064442 26.144581 ) ( -0.466932 -0.531013 -0.466932 )
    // origin
              ……
}
```

A Joint is defined by the following attributes:

- Joint name, a textual attribute between quotes.
- Joint parent, using an index which points to the parent joint using its position in the joints list. The root joint has a parent equals to -1.
- Joint position, defined in model space coordinate system.
- Joint orientation, defined also in model space coordinate system. The orientation in fact is a quaternion whose w-component is not included.

Before continuing explaining the rest of the file let’s talk about quaternions. Quaternions are four component elements that are used to represent rotation. Up to now, we have been using Euler angles (yaw, pitch and roll) to define rotations, which basically define rotation around the x, y and z angles. But, Euler angles present some problems when working with rotations, specifically you must be aware of the correct order to apply de rotations and some operations can get very complex.

This where quaternions come to help in order to solve this complexity. As it has been said before a quaternion is defined as a set of 4 numbers (x, y, z, w). Quaternions define a rotation axis and the rotation angle around that axis.



You can check in the web the mathematical definition of each of the components but the good news is that JOML, the math library we are using, provides support for them. We can construct rotation matrices based on quaternions and perform some transformation to vectors with them.

Let's get back to the joints definition, the w component is missing but it can be easily calculated with the help of the rest of the values. You can check the source code to see how it's done.

After the joints definition you can find the definition of the different meshes that compose a model. Below you can find a fragment of a Mesh definition from one of the samples.

```
mesh {
    shader "/textures/bob/guard1_body.png"

    numverts 494
    vert 0 ( 0.394531 0.513672 ) 0 1
    vert 1 ( 0.447266 0.449219 ) 1 2
    ...
    vert 493 ( 0.683594 0.455078 ) 864 3

    numtris 628
    tri 0 0 2 1
    tri 1 0 1 3
    ...
    tri 627 471 479 493

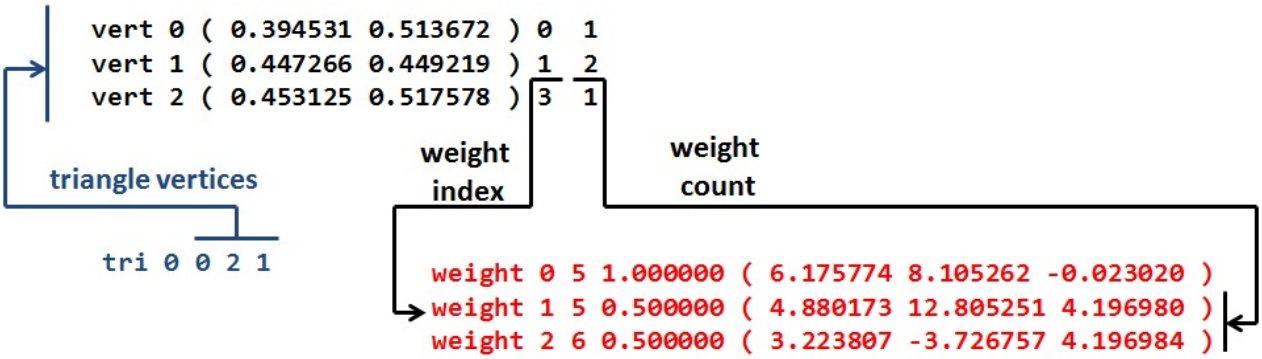
    numweights 867
    weight 0 5 1.000000 ( 6.175774 8.105262 -0.023020 )
    weight 1 5 0.500000 ( 4.880173 12.805251 4.196980 )
    ...
    weight 866 6 0.333333 ( 1.266308 -0.302701 8.949338 )
}
```


www.dbooks.org

Let's review the structure presented above:

- A Mesh starts by defining a texture file. Keep in mind that the path that you will find here is the one used by the tool that created that model. That path may not match the one that is used to load those files. You have two approaches here, either you change the base path dynamically or either you change that path by hand. I've chosen the latter one, the simpler one.
- Next you can find the vertices definition. A vertex is defined by the following attributes:
  - Vertex index.
  - Texture coordinates.
  - The index of the first weight definition that affects this vertex.
  - The number of weights to consider.
- After the vertices, the triangles that form this mesh are defined. The triangles define the way that vertices are organized using their indices.
- Finally, the weights are defined. A Weight definition is composed by:
  - A Weight index.
  - A Joint index, which points to the joint related to this weight.
  - A bias factor, which is used to modulate the effect of this weight.
  - A position of this weight.

The following picture depicts the relation between the components described above using sample data.



Ok, so now that we understand the mesh model file we can parse it. If you look at the source code you will see that a new package has been created to host parsers for model formats. There's one for OBJ files under `org.lwjglb.engine.loaders.obj` and the code for MD5 files

is under `org.lwjglb.engine.loaders.md5`.

All the parsing code is based on regular expressions to extract the information from the MD5 text files. The parsers will create a hierarchy of objects that mimic the structure of the information components contained in the MD5 files. It may not be the most efficient parser in the world but I think it will serve to better understand the process.

The starting class to parse a MD5 model file is `MD5Model` class. This class receives as a parameter in its parse method the contents of a MD5 file an creates a hierarchy that contains the header, the list of joints and the list of meshes with all the subelements. The code is very straightforward so, I won't include it here.

A few comments about the parsing code:

- The subelements of a Mesh are defined as inner classes inside the `MD5Mesh` class.
- You can check how the fourth component of the joints orientation are calculated in the `calculateQuaternion` method form the `MD5Utils` class.

Now that we have parsed a file we must transform that object hierarchy into something that can be processed by the game Engine, we must create a `GameItem` instance. In order to do that we will create a new class named `MD5Loader` that will take a `MD5Model` instance and will construct a `GameItem`.

Before we start, as you noticed, a MD5 model has several Meshes, but our `GameItem` class only supports a single Mesh. We need to change this first, the class `GameItem` now looks like this.

```
package org.lwjglb.engine.items;

import org.joml.Vector3f;
import org.lwjglb.engine.graph.Mesh;

public class GameItem {

    private Mesh[] meshes;

    private final Vector3f position;

    private float scale;

    private final Vector3f rotation;

    public GameItem() {
        position = new Vector3f(0, 0, 0);
        scale = 1;
        rotation = new Vector3f(0, 0, 0);
    }
```


www.dbooks.org

```
    public GameItem(Mesh mesh) {
        this();
        this.meshes = new Mesh[]{mesh};
    }

    public GameItem(Mesh[] meshes) {
        this();
        this.meshes = meshes;
    }

    public Vector3f getPosition() {
        return position;
    }

    public void setPosition(float x, float y, float z) {
        this.position.x = x;
        this.position.y = y;
        this.position.z = z;
    }

    public float getScale() {
        return scale;
    }

    public void setScale(float scale) {
        this.scale = scale;
    }

    public Vector3f getRotation() {
        return rotation;
    }

    public void setRotation(float x, float y, float z) {
        this.rotation.x = x;
        this.rotation.y = y;
        this.rotation.z = z;
    }

    public Mesh getMesh() {
        return meshes[0];
    }

    public Mesh[] getMeshes() {
        return meshes;
    }

    public void setMeshes(Mesh[] meshes) {
        this.meshes = meshes;
    }

    public void setMesh(Mesh mesh) {
        if (this.meshes != null) {
            for (Mesh currMesh : meshes) {
```

```
                currMesh.cleanUp();
            }
        }
        this.meshes = new Mesh[]{mesh};
    }
}
```

With the modification above we can now define the contents for the `MD5Loader` class. This class will have a method named `process` that will receive a `MD5Model` instance and a default colour (for the meshes that do not define a texture) and will return a `GameItem` instance. The body of that method is shown below.

```
public static GameItem process(MD5Model md5Model, Vector4f defaultColour) throws Excep
tion {
    List<MD5Mesh> md5MeshList = md5Model.getMeshes();

    List<Mesh> list = new ArrayList<>();
    for (MD5Mesh md5Mesh : md5Model.getMeshes()) {
        Mesh mesh = generateMesh(md5Model, md5Mesh, defaultColour);
        handleTexture(mesh, md5Mesh, defaultColour);
        list.add(mesh);
    }
    Mesh[] meshes = new Mesh[list.size()];
    meshes = list.toArray(meshes);
    GameItem gameItem = new GameItem(meshes);

    return gameItem;
}
```

As you can see we just iterate over the meshes defined into the `MD5Model` class and transform them into instances of the class `org.lwjglb.engine.graph.Mesh` by using the `generateMesh` method which is the one that really does the work. Before we examine that method we will create an inner class that will serve us to build the positions and normals array.

www.dbooks.org

```
private static class VertexInfo {

    public Vector3f position;

    public Vector3f normal;

    public VertexInfo(Vector3f position) {
        this.position = position;
        normal = new Vector3f(0, 0, 0);
    }

    public VertexInfo() {
        position = new Vector3f();
        normal = new Vector3f();
    }

    public static float[] toPositionsArr(List<VertexInfo> list) {
        int length = list != null ? list.size() * 3 : 0;
        float[] result = new float[length];
        int i = 0;
        for (VertexInfo v : list) {
            result[i] = v.position.x;
            result[i + 1] = v.position.y;
            result[i + 2] = v.position.z;
            i += 3;
        }
        return result;
    }

    public static float[] toNormalArr(List<VertexInfo> list) {
        int length = list != null ? list.size() * 3 : 0;
        float[] result = new float[length];
        int i = 0;
        for (VertexInfo v : list) {
            result[i] = v.normal.x;
            result[i + 1] = v.normal.y;
            result[i + 2] = v.normal.z;
            i += 3;
        }
        return result;
    }
}
```

Let's get back to the `generateMesh` method, the first we do is get the mesh vertices information, the weights and the structure of the joints.

```
private static Mesh generateMesh(MD5Model md5Model, MD5Mesh md5Mesh, Vector4f defaultC
olour) throws Exception {
    List<VertexInfo> vertexInfoList = new ArrayList<>();
    List<Float> textCoords = new ArrayList<>();
    List<Integer> indices = new ArrayList<>();

    List<MD5Mesh.MD5Vertex> vertices = md5Mesh.getVertices();
    List<MD5Mesh.MD5Weight> weights = md5Mesh.getWeights();
    List<MD5JointInfo.MD5JointData> joints = md5Model.getJointInfo().getJoints();
```

Then we need to calculate the vertices position based on the information contained in the weights and joints. This is done in the following block

```
    for (MD5Mesh.MD5Vertex vertex : vertices) {
        Vector3f vertexPos = new Vector3f();
        Vector2f vertexTextCoords = vertex.getTextCoords();
        textCoords.add(vertexTextCoords.x);
        textCoords.add(vertexTextCoords.y);

        int startWeight = vertex.getStartWeight();
        int numWeights = vertex.getWeightCount();

        for (int i = startWeight; i < startWeight + numWeights; i++) {
            MD5Mesh.MD5Weight weight = weights.get(i);
            MD5JointInfo.MD5JointData joint = joints.get(weight.getJointIndex());
            Vector3f rotatedPos = new Vector3f(weight.getPosition()).rotate(joint.getO
rientation());
            Vector3f acumPos = new Vector3f(joint.getPosition()).add(rotatedPos);
            acumPos.mul(weight.getBias());
            vertexPos.add(acumPos);
        }

       vertexInfoList.add(new VertexInfo(vertexPos));
    }
```

Let’s examine what we are doing here. We iterate over the vertices information and store the texture coordinates in a list, no need to apply any transformation here. Then we get the starting and total number of weights to consider to calculate the vertex position.

The vertex position is calculated by using all the weights that is related to. Each weights has a position and a bias. The sum of all bias of the weights associated to each vertex must be equal to 1.0. Each weight also has a position which is defined in joint’s local space, so we need to transform it to model space coordinates using the joint’s orientation and position (like if it were a transformation matrix) to which it refers to.

To sum up, the vertex position can be expressed by this formula:

www.dbooks.org

$$Vpos = \sum_{i=ws}^{ws+wc} (Jt_i \times Wp_i)\dot{W}b_i$$

Where:

- The summation starts from ws (Weight start) up to wc (Weight count) weights.
- Jt_i is the joint's transformation matrix associated to the weight W_i.
- Wp_i is the weight position.
- Wb_i is the weight bias.

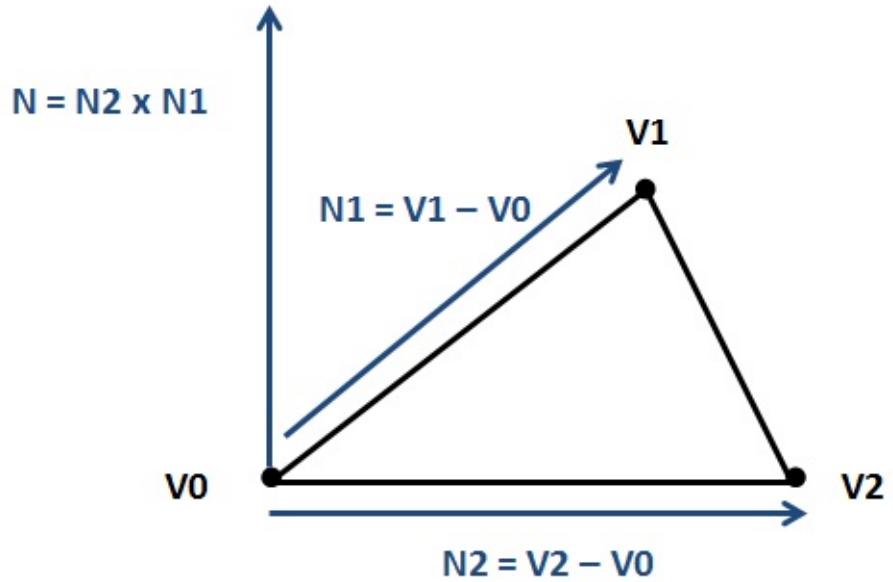
This equation is what we implement in the body of the loop (we do not have the transformation matrix since we have the joint position and rotation separately but the result is the same).

With the code above we will be able to construct the positions and texture coordinates data but we still need to build up the indices and the normals. Indices can be calculated by using the triangles information, just by iterating through the list that holds those triangles.

Normals can be calculated also using triangles information. Let V_0, V_1 and V_2 be the triangle vertices (in object's model space). The normal for the triangle can be calculate according to this formula:

$$N = (V_2 - V_0) \times (V_1 - V_0)$$

Where N should be normalized after. The following figure shows the geometric interpretation of the formula above.



For each vertex we compute its normal by the normalized sum of all the normals of the triangles it belongs to. The code that performs those calculations is shown below.

```
    for (MD5Mesh.MD5Triangle tri : md5Mesh.getTriangles()) {
        indices.add(tri.getVertex0());
        indices.add(tri.getVertex1());
        indices.add(tri.getVertex2());

        // Normals
        VertexInfo v0 = vertexInfoList.get(tri.getVertex0());
        VertexInfo v1 = vertexInfoList.get(tri.getVertex1());
        VertexInfo v2 = vertexInfoList.get(tri.getVertex2());
        Vector3f pos0 = v0.position;
        Vector3f pos1 = v1.position;
        Vector3f pos2 = v2.position;

        Vector3f normal = (new Vector3f(pos2).sub(pos0)).cross(new Vector3f(pos1).sub(
pos0));

        v0.normal.add(normal);
        v1.normal.add(normal);
        v2.normal.add(normal);
     }

     // Once the contributions have been added, normalize the result
     for(VertexInfo v : vertexInfoList) {
        v.normal.normalize();
    }
```

Then we just need to transform the Lists to arrays and process the texture information.

```
     float[] positionsArr = VertexInfo.toPositionsArr(vertexInfoList);
     float[] textCoordsArr = Utils.listToArray(textCoords);
     float[] normalsArr = VertexInfo.toNormalArr(vertexInfoList);
     int[] indicesArr = indices.stream().mapToInt(i -> i).toArray();
     Mesh mesh = new Mesh(positionsArr, textCoordsArr, normalsArr, indicesArr);

     return mesh;
}
```

Going back to the `process` method you can see that there's a method named `handleTexture`, which is responsible for loading textures. This is the definition of that method:

www.dbooks.org

```
private static void handleTexture(Mesh mesh, MD5Mesh md5Mesh, Vector4f defaultColour)
throws Exception {
    String texturePath = md5Mesh.getTexture();
    if (texturePath != null && texturePath.length() > 0) {
        Texture texture = new Texture(texturePath);
        Material material = new Material(texture);

        // Handle normal Maps;
        int pos = texturePath.lastIndexOf(".");
        if (pos > 0) {
            String basePath = texturePath.substring(0, pos);
            String extension = texturePath.substring(pos, texturePath.length());
            String normalMapFileName = basePath + NORMAL_FILE_SUFFIX + extension;
            if (Utils.existsResourceFile(normalMapFileName)) {
                Texture normalMap = new Texture(normalMapFileName);
                material.setNormalMap(normalMap);
            }
        }
        mesh.setMaterial(material);
    } else {
        mesh.setMaterial(new Material(defaultColour, 1));
    }
}
```

The implementation is very straight forward. The only peculiarity is that if a mesh defines a texture named “texture.png” its normal texture map will be defined in a file “texture_normal.png”. We need to check if that file exists and load it accordingly.

We can now load a MD5 file and render it as we render other GameItems, but before doing that we need to disable cull face in order to render it properly since not all the triangles will be drawn in the correct direction. We will add support to the Window class to set these parameters at runtime (you can check it in the source code the changes).

If you load some of the sample models you will get something like this.

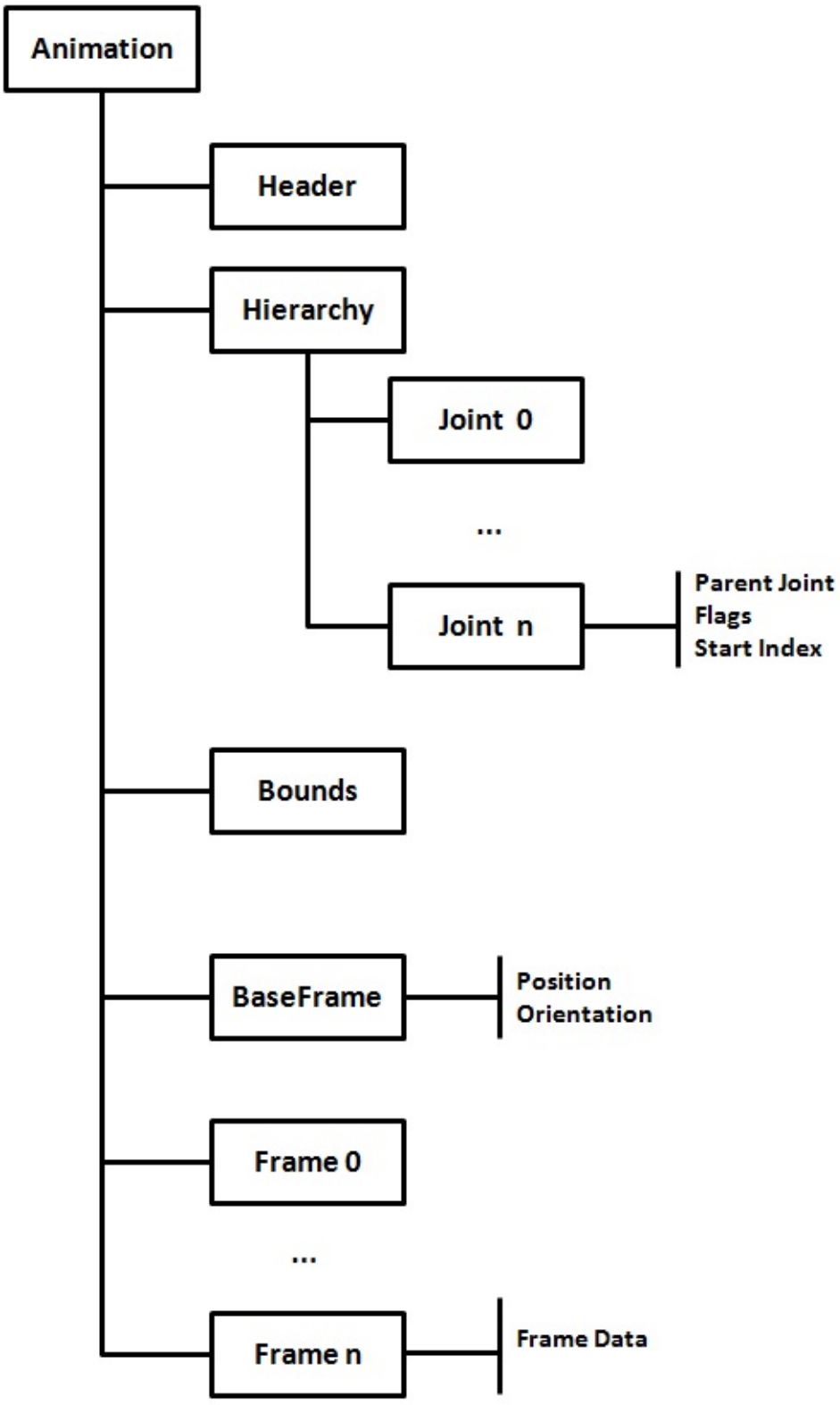


What you see here is the binding pose, it’s the static representation of the MD5 model used for the animators to model them easily. In order to get animation to work we must process the animation definition file.

Animate the model

A MD5 animation definition file, like the model definition one, is composed by a header an different sections contained between braces. If you open one of those files you can see a structure similar like this.

www.dbooks.org



The first structure that you can find in the animation file, as in the case of the mesh definition file, is the header. You can see below header's content from one of the samples provided:

```
MD5Version 10
commandline ""

numFrames 140
numJoints 33
frameRate 24
numAnimatedComponents 198
```

The header defines the following attributes:

- The version of the MD5 specification that it complies to.
- The command used to generate this file (from a 3D modelling tool).
- The number frames defined in the file.
- The number of joints defined in the hierarchy section.
- The frame rate, frames per second, that was used while creating this animation. This parameter can be used to calculate the time between frames.
- The number of components that each frame defines.

The hierarchy section is the one that comes first and defines the joints for this animation. You can see a fragment below:

```
hierarchy {
    "origin"    -1 0 0    //
    "body"    0 63 0    // origin ( Tx Ty Tz Qx Qy Qz )
    "body2"    1 0 0    // body
    "SPINNER"    2 56 6    // body2 ( Qx Qy Qz )
    ....
}
```

A joint. In the hierarchy section, is defined by the following attributes:

- Joint name, a textual attribute between quotes.
- Joint parent, using an index which points to the parent joint using its position in the joints list. The root joint has a parent equals to -1.
- Joint flags, which set how this joint's position and orientation will be changed according to the data defined in each animation frame.
- The start index, inside the animation data of each frame that is used when applying the flags.

The next section is the bounds one. This section defines a bounding box which contains the model for each animation frame. It will contain a line for each of the animation frames and it look like this:

www.dbooks.org

```
bounds {
    ( -24.3102264404 -44.2608566284 -0.181215778 ) ( 31.0861988068 38.7131576538 117.7
417449951 )
    ( -24.3102283478 -44.1887664795 -0.1794649214 ) ( 31.1800289154 38.7173080444 117.
7729110718 )
    ( -24.3102359772 -44.1144447327 -0.1794776917 ) ( 31.2042789459 38.7091217041 117.
8352737427 )
    ....
}
```

Each bounding box is defined by two 3 component vectors in model space coordinates. The first vector defines the minimum bound and the second one the maximum.

The next section is the base frame data. In this section, the position and orientation of each joint is set up before the deformations of each animation frame are applied. You can see a fragment below:

```
baseframe {
    ( 0 0 0 ) ( -0.5 -0.5 -0.5 )
    ( -0.8947336078 70.7142486572 -6.5027675629 ) ( -0.3258574307 -0.0083037354 0.0313
780755 )
    ( 0.0000001462 0.0539700091 -0.0137935728 ) ( 0 0 0 )
    ....
}
```

Each line is associated to a joint and define the following attributes:

- Position of the joint, as a three components vector.
- Orientation of the joint, as the three components of a quaternion (as in the model file).

After that you will find several frame definitions, as many as the value assigned to the numFrames header attribute. Each frame section is like a huge array of floats that will be used by the joints when applying the transformations for each frame. You can see a fragment below.

```
frame 1 {
     -0.9279100895 70.682762146 -6.3709330559 -0.3259022534 -0.0100501738 0.0320306309
     0.3259022534 0.0100501738 -0.0320306309
     -0.1038384438 -0.1639953405 -0.0152553488 0.0299418624
     ....
}
```

The base class that parses a MD5 animation file is named `MD5AnimModel`. This class creates all the objects hierarchy that maps the contents of that file and you can check the source code for the details. The structure is similar to the MD5 model definition file. Now that we are

able to load that information we will use it to generate an animation.

We will generate the animation in the shader, so instead of pre-calculating all the positions for each frame we need to prepare the data we need so in the vertex shader we can compute the final positions.
Let's get back to the process method in the `MD5Loader` class, we need to modify it to take into consideration the animation information. The new definition for that method is shown below:

```
public static AnimGameItem process(MD5Model md5Model, MD5AnimModel animModel, Vector4f
 defaultColour) throws Exception {
    List<Matrix4f> invJointMatrices = calcInJointMatrices(md5Model);
    List<AnimatedFrame> animatedFrames = processAnimationFrames(md5Model, animModel, i
nvJointMatrices);

    List<Mesh> list = new ArrayList<>();
    for (MD5Mesh md5Mesh : md5Model.getMeshes()) {
        Mesh mesh = generateMesh(md5Model, md5Mesh);
        handleTexture(mesh, md5Mesh, defaultColour);
        list.add(mesh);
    }

    Mesh[] meshes = new Mesh[list.size()];
    meshes = list.toArray(meshes);

    AnimGameItem result = new AnimGameItem(meshes, animatedFrames, invJointMatrices);
    return result;
}
```

There are some changes here, the most obvious is that the method now receives a `MD5AnimModel` instance. The next one is that we do not return a `GameItem` instance but and `AnimGameItem` one. This class inherits from the `GameItem` class but adds support for animations. We will see why this was done this way later.

If we continue with the process method, the first thing we do is call the `calcInJointMatrices` method, which is defined like this:

www.dbooks.org

```
private static List<Matrix4f> calcInJointMatrices(MD5Model md5Model) {
    List<Matrix4f> result = new ArrayList<>();

    List<MD5JointInfo.MD5JointData> joints = md5Model.getJointInfo().getJoints();
    for(MD5JointInfo.MD5JointData joint : joints) {
        Matrix4f translateMat = new Matrix4f().translate(joint.getPosition());
        Matrix4f rotationMat = new Matrix4f().rotate(joint.getOrientation());
        Matrix4f mat = translateMat.mul(rotationMat);
        mat.invert();
        result.add(mat);
    }
    return result;
}
```

This method iterates over the joints contained in the MD5 model definition file, calculates the transformation matrix associated to each joint and then it gets the inverse of those matrices. This information is used to construct the AnimationGameItem instance.

Let's continue with the `process` method, the next thing we do is process the animation frames by calling the `processAnimationFrames` method:

```
private static List<AnimatedFrame> processAnimationFrames(MD5Model md5Model, MD5AnimMo
del animModel, List<Matrix4f> invJointMatrices) {
    List<AnimatedFrame> animatedFrames = new ArrayList<>();
    List<MD5Frame> frames = animModel.getFrames();
    for(MD5Frame frame : frames) {
        AnimatedFrame data = processAnimationFrame(md5Model, animModel, frame, invJoin
tMatrices);
        animatedFrames.add(data);
    }
    return animatedFrames;
}
```

This method process each animation frame, defined in the MD5 animation definition file, and returns a list of `AnimatedFrame` instances. The real work is done in the `processAnimationFrame` method. Let's explain what this method will do.

We first, iterate over the joints defined in the hierarchy section in the MD5 animaton file.

```
private static AnimatedFrame processAnimationFrame(MD5Model md5Model, MD5AnimModel ani
mModel, MD5Frame frame, List<Matrix4f> invJointMatrices) {
    AnimatedFrame result = new AnimatedFrame();

    MD5BaseFrame baseFrame = animModel.getBaseFrame();
    List<MD5Hierarchy.MD5HierarchyData> hierarchyList = animModel.getHierarchy().getHi
erarchyDataList();

    List<MD5JointInfo.MD5JointData> joints = md5Model.getJointInfo().getJoints();
    int numJoints = joints.size();
    float[] frameData = frame.getFrameData();
    for (int i = 0; i < numJoints; i++) {
        MD5JointInfo.MD5JointData joint = joints.get(i);
```

We get the position and orientation of the base frame element associated to each joint.

```
        MD5BaseFrame.MD5BaseFrameData baseFrameData = baseFrame.getFrameDataList().get
(i);
        Vector3f position = baseFrameData.getPosition();
        Quaternionf orientation = baseFrameData.getOrientation();
```

In principle, that information should be assigned to the the joint's position and orientation, but it needs to be transformed according to the joint's flag. If you recall, when the structure of the animation file was presented, each joint in the hierarchy section defines a flag. That flag models how the position and orientation information should be changed according to the information defined in each animation frame.

If the first bit of that flag field is equal to 1, we should change the x component of the base frame position with the data contained in the animation frame we are processing. That animation farme defines a bug afloat array, so which I elements should we take. The answer is also in the joints definition which includes a startIndex attribute. If the second bit of the gal is equal to 1, we should change the y component of the base frame position with the value at startIndex + 1, and so on. The next bits are for the z position, and the x, y and z components of the orientation.

www.dbooks.org

```
            int flags = hierarchyList.get(i).getFlags();
            int startIndex = hierarchyList.get(i).getStartIndex();

            if ( (flags & 1 ) > 0) {
                position.x = frameData[startIndex++];
            }
            if ( (flags & 2) > 0) {
                position.y = frameData[startIndex++];
            }
            if ( (flags & 4) > 0) {
                position.z = frameData[startIndex++];
            }
            if ( (flags & 8) > 0) {
                orientation.x = frameData[startIndex++];
            }
            if ( (flags & 16) > 0) {
                orientation.y = frameData[startIndex++];
            }
            if ( (flags & 32) > 0) {
                orientation.z = frameData[startIndex++];
            }
            // Update Quaternion's w component
            orientation = MD5Utils.calculateQuaternion(orientation.x, orientation.y, orien
tation.z);
```

Now we have all information needed to calculate the transformation matrices to get the final position for each joint for the current animation frame. But there's another thing that we must consider, the position of each joint is relative to its parent position, so we need to get the transformation matrix associated to each parent and use it in order to get a transformation matrix that is in model space coordinates.

```
        // Calculate translation and rotation matrices for this joint
        Matrix4f translateMat = new Matrix4f().translate(position);
        Matrix4f rotationMat = new Matrix4f().rotate(orientation);
        Matrix4f jointMat = translateMat.mul(rotationMat);

        // Joint position is relative to joint's parent index position. Use parent mat
rices
        // to transform it to model space
        if ( joint.getParentIndex() > -1 ) {
            Matrix4f parentMatrix = result.getLocalJointMatrices()[joint.getParentInde
x()];
            jointMat = new Matrix4f(parentMatrix).mul(jointMat);
        }

        result.setMatrix(i, jointMat, invJointMatrices.get(i));
    }

    return result;
}
```

You can see that we create an instance of the AnimatedFrame class that holds the information that will be use during animation. This class also uses the inverse matrices, we will see later on why this done this way. An important thing to note is that the setMatrix method of the AnimatedFrame is defined like this.

```
public void setMatrix(int pos, Matrix4f localJointMatrix, Matrix4f invJointMatrix) {
    localJointMatrices[pos] = localJointMatrix;
    Matrix4f mat = new Matrix4f(localJointMatrix);
    mat.mul(invJointMatrix);
    jointMatrices[pos] = mat;
}
```

The variable `localJointMatrix` stores the transformation matrix for the joint that occupies the position “i” for the current frame. The `invJointMatrix` holds the inverse transformation matrix for the joint that occupies the position “i” for the binding pose. We store the result of multiplying the `localJointMatrix` by the invJointMatrix. This result will be used later to compute the final positions. We store also the original joint transformation matrix, the variable `localJointMatrix`, so we can use it to calculate this joint childs transformation matrices.

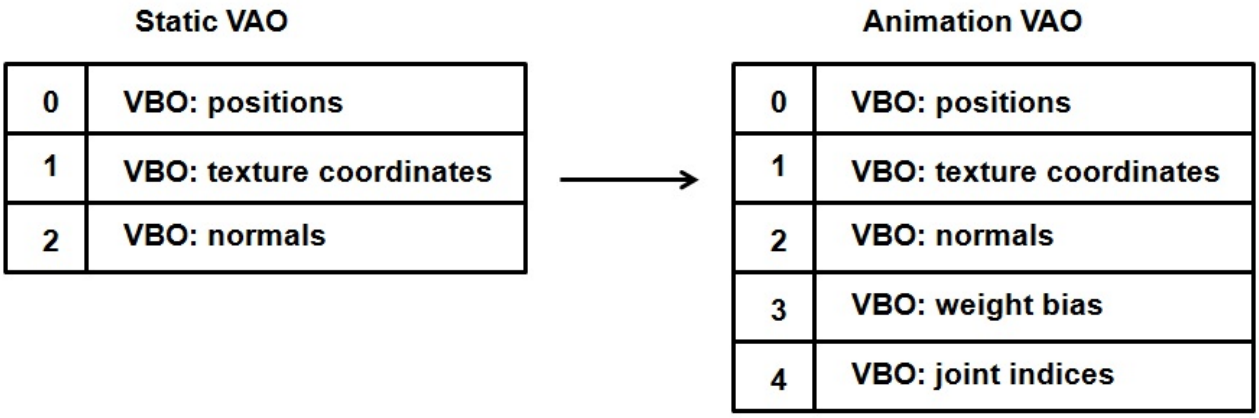
Let's get back to the MD5Loader class. The `generateMesh` method also has changed, we calculate the positions of the binding pose as it has been explained before, but for each vertex we store two arrays:

- An array that holds the weight bias associated to this vertex.
- An array that hold the joint indices associated to this vertex (through the weights).

www.dbooks.org

We limit the size of those arrays to a value of 4. The `Mesh` class has also been modified to receive those parameters and include it in the VAO information processed by the shaders. You can check the details in the source code, but So let's recap what we have done:

- We are still loading the binding pose with their final positions calculated as the sum of the joints positions and orientations through the weights information.
- That information is loaded in the shaders as VBOs but it's complemented by the bias of the weights associated to each vertex and the indices of the joints that affect it. This information is common to all the animation frames, since it's defined in the MD5 definition file. This is the reason why we limit the size of the bias and joint indices arrays, they will be loaded as VBOs once when the model is sent to the GPU.
- For each animation frame we store the transformation matrices to be applied to each joint according to the positions and orientations defined in the base frame.
- We also have calculated the inverse matrices of the transformation matrices associated to the joints that define the binding pose. That is, we know how to undo the transformations done in the binding pose. We will see how this will be applied later.



Now that we have all the pieces to solve the puzzle we just need to use them in the shader. We first need to modify the input data to receive the weights and the joint indices.

```
#version 330

const int MAX_WEIGHTS = 4;
const int MAX_JOINTS = 150;

layout (location=0) in vec3 position;
layout (location=1) in vec2 texCoord;
layout (location=2) in vec3 vertexNormal;
layout (location=3) in vec4 jointWeights;
layout (location=4) in ivec4 jointIndices;
```

We have defined two constants:

- `MAX_WEIGHTS`, defines the maximum number of weights that come in the weights VBO

(an solo the joint indices)

- `MAX_JOINTS`, defines the maximum number of joints we are going to support (more on this later).

Then we define the output data and the uniforms.

```
out vec2 outTexCoord;
out vec3 mvVertexNormal;
out vec3 mvVertexPos;
out vec4 mlightviewVertexPos;
out mat4 outModelViewMatrix;

uniform mat4 jointsMatrix[MAX_JOINTS];
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform mat4 modelLightViewMatrix;
uniform mat4 orthoProjectionMatrix;
```

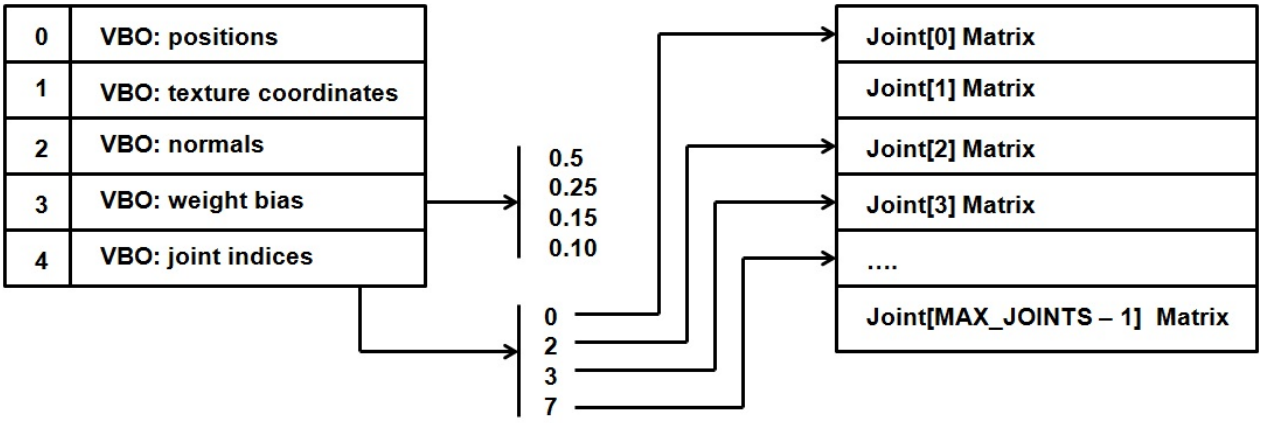
You can see that we have a new uniform named `jointsMatrix` which is an array of matrices (with a maximum length set by the `MAX_JOINTS` constant). That array of matrices holds the joint matrices calculated for all the joints in the present frame, and was calculated in the `MD5Loader` class when processing a frame. Thus, that array holds the transformations that need to be applied to a joint in the present animation frame and will serve as the basis for calculating the vertex final position.

With the new data in the VBOs and this uniform we will transform the binding pose position. This is done in the following block.

```
    vec4 initPos = vec4(0, 0, 0, 0);
    int count = 0;
    for(int i = 0; i < MAX_WEIGHTS; i++)
    {
        float weight = jointWeights[i];
        if(weight > 0) {
            count++;
            int jointIndex = jointIndices[i];
            vec4 tmpPos = jointsMatrix[jointIndex] * vec4(position, 1.0);
            initPos += weight * tmpPos;
        }
    }
    if (count == 0)
    {
        initPos = vec4(position, 1.0);
    }
```


www.dbooks.org

First of all, we get the binding pose position, we iterate over the weights associated to this vertex and modify the position using the weights and the joint matrices for this frame (stored in the jointsMatrix uniform) by using the index that is stored in the input.



So, given a vertex position, we are calculating it's frame position as

$$Vfp = \sum_{i=0}^{MAXWEIGTHS} Wb_i \dot{(} Jfp_i \times Jt_i^{-1}) \times Vpos$$

Where:

- $Wfvp$ is the vertex final position
- Wb is the vertex weight
- Jfp is the joint matrix transformation matrix for this frame
- Jt^{-1} is the inverse of the joint transformation matrix for the binding pose. The multiplication of this matrix and Jfp is what's contained in the `jointsMatrix` uniform.
- $Vpos$ is the vertex position in the binding position.

$Vpos$ is calcualted by usin the Jt matrix, which is the matrix of the joint transformation matrix for the binding pose. So, at the end we are somehow undoing the modificications of the binding pose to apply the transformations for this frame. This is the reason why we need the inverse binding pose matrix.

The shader supports vertices with variable number of weights, up to a maximum of 4, and also supports the rendering of non animated items. In this case, the weights will be equal to 0 and we will get the original position.

```
The rest of the shader stays more or less the same, we just use the updated position a
nd pass the correct values to be used by the fragment shader.
    vec4 mvPos = modelViewMatrix * initPos;
    gl_Position = projectionMatrix * mvPos;
    outTexCoord = texCoord;
    mvVertexNormal = normalize(modelViewMatrix * vec4(vertexNormal, 0.0)).xyz;
    mvVertexPos = mvPos.xyz;
    mlightviewVertexPos = orthoProjectionMatrix * modelLightViewMatrix * vec4(position
, 1.0);
    outModelViewMatrix = modelViewMatrix;
}
```

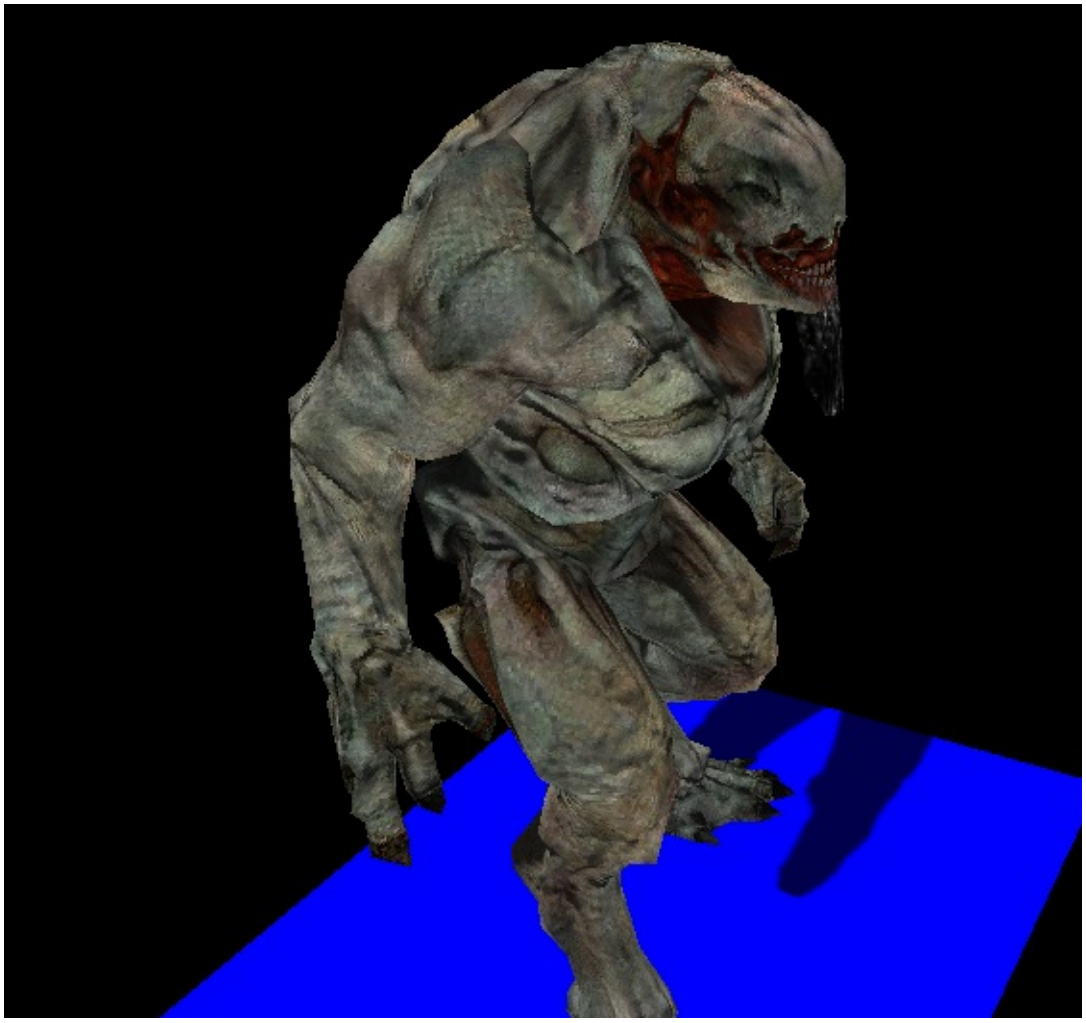
So, in order to test the animation we just need to pass the `jointsMatrix` to the shader. Since this information is stored only in instances of the `AnimGameItem` class, the code is very simple. In the loop that renders the Meshes, we add this fragment.

```
if ( gameItem instanceof AnimGameItem ) {
    AnimGameItem animGameItem = (AnimGameItem)gameItem;
    AnimatedFrame frame = animGameItem.getCurrentFrame();
    sceneShaderProgram.setUniform("jointsMatrix", frame.getJointMatrices());
}
```

Of course, yo will need to create the uniform before using it, you can check the source code for that. If you run the example you will be able to see how the model animates by pressing the space bar (each time the key is pressed a new frame is set and the jointsMatrix uniform changes).

You will see something like this.

www.dbooks.org



Although the animation is smooth, the sample presents some problems. First of all, light is not correctly applied and the shadow represents the binding pose but not the current frame. We will solve all these problems now.

Correcting animation issues

The first issue that will address is the lightning problem. You may have already noticed the case, its due to the fact that we are not transforming normals. Thus, the normals that are used in the fragment shader are the ones that correspond to the binding pose. We need to transform them in the same way as the positions.

This issue is easy to solve, we just need to include the normals in the loop that iterates over the weights in the vertex shader.

```
    vec4 initPos = vec4(0, 0, 0, 0);
    vec4 initNormal = vec4(0, 0, 0, 0);
    int count = 0;
    for(int i = 0; i < MAX_WEIGHTS; i++)
    {
        float weight = jointWeights[i];
        if(weight > 0) {
            count++;
            int jointIndex = jointIndices[i];
            vec4 tmpPos = jointsMatrix[jointIndex] * vec4(position, 1.0);
            initPos += weight * tmpPos;

            vec4 tmpNormal = jointsMatrix[jointIndex] * vec4(vertexNormal, 0.0);
            initNormal += weight * tmpNormal;
        }
    }
    if (count == 0)
    {
        initPos = vec4(position, 1.0);
        initNormal = vec4(vertexNormal, 0.0);
    }
```

Then we just calculate the output vertex normal as always:

```
mvVertexNormal = normalize(modelViewMatrix * initNormal).xyz;
```

The next issue is the shadow problem. If you recall from the shadows chapter, we are using shadow maps to draw shadows. We are rendering the scene from the light perspective in order to create a depth map that tells us if a point is in shadow or not. But, as in the case of the normals, we are just passing the binding pose coordinates and not changing them according to the current frame. This is the reason why the shadow does not corresponds to the current position.

The solution is easy, we just need to modify the depth vertex shader to use the `jointsMatrix` and the weights and joint indices to transform the position. This is how the depth vertex shader looks like.

www.dbooks.org

```
#version 330

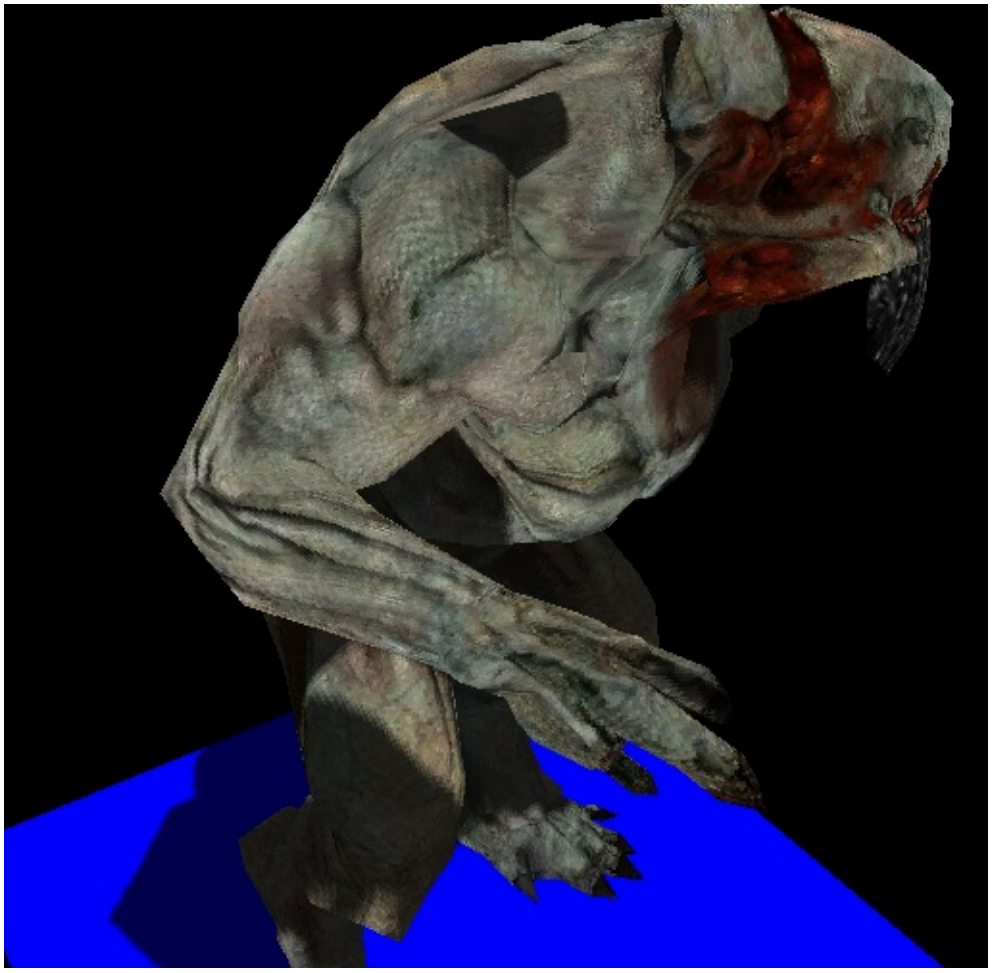
const int MAX_WEIGHTS = 4;
const int MAX_JOINTS = 150;

layout (location=0) in vec3 position;
layout (location=1) in vec2 texCoord;
layout (location=2) in vec3 vertexNormal;
layout (location=3) in vec4 jointWeights;
layout (location=4) in ivec4 jointIndices;

uniform mat4 jointsMatrix[MAX_JOINTS];
uniform mat4 modelLightViewMatrix;
uniform mat4 orthoProjectionMatrix;

void main()
{
    vec4 initPos = vec4(0, 0, 0, 0);
    int count = 0;
    for(int i = 0; i < MAX_WEIGHTS; i++)
    {
        float weight = jointWeights[i];
        if(weight > 0) {
            count++;
            int jointIndex = jointIndices[i];
            vec4 tmpPos = jointsMatrix[jointIndex] * vec4(position, 1.0);
            initPos += weight * tmpPos;
        }
    }
    if (count == 0)
    {
        initPos = vec4(position, 1.0);
    }
    gl_Position = orthoProjectionMatrix * modelLightViewMatrix * initPos;
}
```

You need to modify the `Renderer` class to set up the new uniforms for this shader, and the final result will be much better. The light will be applied correctly and the shadow will change for each animation frame as shown in the next figure.



And that's all, you have now a working example that animates MD5 models. The source code can still be improved and you can modify the matrices that are loaded in each render cycle to interpolate betweeen frames positions. You can check the sources used for this chapter to see how this can be done.

www.dbooks.org

Particles

The basics

In this chapter we will add particle effects to the game engine. With this effect we will be able to simulate rays, fire, dust and clouds. It’s a simple effect to implement that will improve the graphical aspect of any game.

Before we start it's worth to mention that there are many ways to implement particle effects with different, results. In this case we will use billboard particles. This technique uses moving texture quads to represent a particle with the peculiarity that they are always always facing the observer, in our case, the camera. You can also use billboarding technique to show information panels over game items like a mini HUDs.

Let’s start by defining what is a particle. A particle can de defined by the following attributes:

1. A mesh that represents the quad vertices.
2. A texture.
3. A position at a given instant.
4. A scale factor.
5. Speed.
6. A movement direction.
7. A life time or time to live. Once this time has expired the particle ceases to exist.

The first four items are part of the `GameItem` class, but the last three are not. Thus, we will create a new class named `Particle` that extends a `GameItem` instance and that is defined like this.

```
package org.lwjglb.engine.graph.particles;

import org.joml.Vector3f;
import org.lwjglb.engine.graph.Mesh;
import org.lwjglb.engine.items.GameItem;

public class Particle extends GameItem {

    private Vector3f speed;

    /**
     * Time to live for particle in milliseconds.
     */
    private long ttl;
```

```
    public Particle(Mesh mesh, Vector3f speed, long ttl) {
        super(mesh);
        this.speed = new Vector3f(speed);
        this.ttl = ttl;
    }

    public Particle(Particle baseParticle) {
        super(baseParticle.getMesh());
        Vector3f aux = baseParticle.getPosition();
        setPosition(aux.x, aux.y, aux.z);
        aux = baseParticle.getRotation();
        setRotation(aux.x, aux.y, aux.z);
        setScale(baseParticle.getScale());
        this.speed = new Vector3f(baseParticle.speed);
        this.ttl = baseParticle.geTtl();
    }

    public Vector3f getSpeed() {
        return speed;
    }

    public void setSpeed(Vector3f speed) {
        this.speed = speed;
    }

    public long geTtl() {
        return ttl;
    }

    public void setTtl(long ttl) {
        this.ttl = ttl;
    }

    /**
     * Updates the Particle's TTL
     * @param elapsedTime Elapsed Time in milliseconds
     * @return The Particle's TTL
     */
    public long updateTtl(long elapsedTime) {
        this.ttl -= elapsedTime;
        return this.ttl;
    }
}
```

As you can see from the code above, particle's speed and movement direction can be expressed as a single vector. The direction of that vector models the movement direction and its module the speed. The Particle Time To Live (TTL) is modelled as milliseconds counter that will be decreased whenever the game state is updated. The class has also a copy constructor, that is, a constructor that takes an instance of another Particle to make a copy.

www.dbooks.org

Now, we need to create a particle generator or particle emitter, that is, a class that generates the particles dynamically, controls their life cycle and updates their position according to a specific model. We can create many implementations that vary in how particles and created and how their positions are updated (for instance, taking into consideration the gravity or not). So, in order to keep our game engine generic, we will create an interface that all the Particle emitters must implement. This interface, named `IParticleEmitter`, is defined like this:

```
package org.lwjglb.engine.graph.particles;

import java.util.List;
import org.lwjglb.engine.items.GameItem;

public interface IParticleEmitter {

    void cleanup();

    Particle getBaseParticle();

    List<GameItem> getParticles();
}
```

The `IParticleEmitter` interface has a method to clean up resources, named `cleanup`, and a method to get the list of Particles, named `getParticles`. It also as a method named `getBaseParticle`, but What's this method for? A particle emitter will create many particles dynamically. Whenever a particle expires, new ones will be created. That particle renewal cycle will use a base particle, like a pattern, to create new instances. This is what this base particle is used for, This is also the reason why the `Particle` class defines a copy constructor.

In the game engine code we will refer only to the `IParticleEmitter` interface so the base code will not be dependent on the specific implementations. Nevertheless we can create a implementation that simulates a flow of particles that are not affected by gravity. This implementation can be used to simulate rays or fire and is named `FlowParticleEmitter`.

The behaviour of this class can be tuned with the following attributes:

- A maximum number of particles that can be alive at a time.
- A minimum period to create particles. Particles will be created one by one with a minimum period to avoid creating particles in bursts.
- A set of ranges to randomize particles speed and starting position. New particles will use base particle position and speed which can be randomized with values between those ranges to spread the beam.

The implementation of this class is as follows:

```
package org.lwjglb.engine.graph.particles;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import org.joml.Vector3f;
import org.lwjglb.engine.items.GameItem;

public class FlowParticleEmitter implements IParticleEmitter {

    private int maxParticles;

    private boolean active;

    private final List<GameItem> particles;

    private final Particle baseParticle;

    private long creationPeriodMillis;

    private long lastCreationTime;

    private float speedRndRange;

    private float positionRndRange;

    private float scaleRndRange;

    public FlowParticleEmitter(Particle baseParticle, int maxParticles, long creationP
eriodMillis) {
        particles = new ArrayList<>();
        this.baseParticle = baseParticle;
        this.maxParticles = maxParticles;
        this.active = false;
        this.lastCreationTime = 0;
        this.creationPeriodMillis = creationPeriodMillis;
    }

    @Override
    public Particle getBaseParticle() {
        return baseParticle;
    }

    public long getCreationPeriodMillis() {
        return creationPeriodMillis;
    }

    public int getMaxParticles() {
        return maxParticles;
    }

    @Override
```


www.dbooks.org

```
    public List<GameItem> getParticles() {
        return particles;
    }

    public float getPositionRndRange() {
        return positionRndRange;
    }

    public float getScaleRndRange() {
        return scaleRndRange;
    }

    public float getSpeedRndRange() {
        return speedRndRange;
    }

    public void setCreationPeriodMillis(long creationPeriodMillis) {
        this.creationPeriodMillis = creationPeriodMillis;
    }

    public void setMaxParticles(int maxParticles) {
        this.maxParticles = maxParticles;
    }

    public void setPositionRndRange(float positionRndRange) {
        this.positionRndRange = positionRndRange;
    }

    public void setScaleRndRange(float scaleRndRange) {
        this.scaleRndRange = scaleRndRange;
    }

    public boolean isActive() {
        return active;
    }

    public void setActive(boolean active) {
        this.active = active;
    }

    public void setSpeedRndRange(float speedRndRange) {
        this.speedRndRange = speedRndRange;
    }

    public void update(long ellapsedTime) {
        long now = System.currentTimeMillis();
        if (lastCreationTime == 0) {
            lastCreationTime = now;
        }
        Iterator<? extends GameItem> it = particles.iterator();
        while (it.hasNext()) {
            Particle particle = (Particle) it.next();
            if (particle.updateTtl(ellapsedTime) < 0) {
```

```
                it.remove();
            } else {
                updatePosition(particle, ellapsedTime);
            }
        }

        int length = this.getParticles().size();
        if (now - lastCreationTime >= this.creationPeriodMillis && length < maxParticl
es) {
            createParticle();
            this.lastCreationTime = now;
        }
    }

    private void createParticle() {
        Particle particle = new Particle(this.getBaseParticle());
        // Add a little bit of randomness of the parrticle
        float sign = Math.random() > 0.5d ? -1.0f : 1.0f;
        float speedInc = sign * (float)Math.random() * this.speedRndRange;
        float posInc = sign * (float)Math.random() * this.positionRndRange;
        float scaleInc = sign * (float)Math.random() * this.scaleRndRange;
        particle.getPosition().add(posInc, posInc, posInc);
        particle.getSpeed().add(speedInc, speedInc, speedInc);
        particle.setScale(particle.getScale() + scaleInc);
        particles.add(particle);
    }

    /**
     * Updates a particle position
     * @param particle The particle to update
     * @param elapsedTime Elapsed time in milliseconds
     */
    public void updatePosition(Particle particle, long elapsedTime) {
        Vector3f speed = particle.getSpeed();
        float delta = elapsedTime / 1000.0f;
        float dx = speed.x * delta;
        float dy = speed.y * delta;
        float dz = speed.z * delta;
        Vector3f pos = particle.getPosition();
        particle.setPosition(pos.x + dx, pos.y + dy, pos.z + dz);
    }

    @Override
    public void cleanup() {
        for (GameItem particle : getParticles()) {
            particle.cleanup();
        }
    }
}
```

Now we can extend the information that's contained in the `Scene` class to include an array of `ParticleEmitter` instances.

www.dbooks.org

```
package org.lwjglb.engine;

// Imports here

public class Scene {

    // More attributes here

    private IParticleEmitter[] particleEmitters;
```

At this stage we can start rendering the particles. Particles will not be affected by lights and will not cast any shadow. They will not have any skeletal animation, so it makes sense to have specific shaders to render them. The shaders will be very simple, they will just render the vertices using the projection and modelview matrices and use a texture to set the colours.

The vertex shader is defined like this.

```
#version 330

layout (location=0) in vec3 position;
layout (location=1) in vec2 texCoord;
layout (location=2) in vec3 vertexNormal;

out vec2 outTexCoord;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;

void main()
{
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
    outTexCoord = texCoord;
}
```

The fragment shader is defined like this:

```
#version 330

in vec2 outTexCoord;
in vec3 mvPos;
out vec4 fragColor;

uniform sampler2D texture_sampler;

void main()
{
    fragColor = texture(texture_sampler, outTexCoord);
}
```

As you can see they are very simple, they resemble the pair of shaders used in the first chapters. Now, as in other chapters, we need to setup and use those shaders in the `Renderer` class. The shaders setup will be done in a method named `setupParticlesShader` which is defined like this:

```
private void setupParticlesShader() throws Exception {
    particlesShaderProgram = new ShaderProgram();
    particlesShaderProgram.createVertexShader(Utils.loadResource("/shaders/particles_v
ertex.vs"));
    particlesShaderProgram.createFragmentShader(Utils.loadResource("/shaders/particles
_fragment.fs"));
    particlesShaderProgram.link();

    particlesShaderProgram.createUniform("projectionMatrix");
    particlesShaderProgram.createUniform("modelViewMatrix");
    particlesShaderProgram.createUniform("texture_sampler");
}
```

And now we can create the render method named `renderParticles` in the `Renderer` class which is defined like this:

www.dbooks.org

```
private void renderParticles(Window window, Camera camera, Scene scene) {
    particlesShaderProgram.bind();

    particlesShaderProgram.setUniform("texture_sampler", 0);
    Matrix4f projectionMatrix = transformation.getProjectionMatrix();
    particlesShaderProgram.setUniform("projectionMatrix", projectionMatrix);

    Matrix4f viewMatrix = transformation.getViewMatrix();
    IParticleEmitter[] emitters = scene.getParticleEmitters();
    int numEmitters = emitters != null ? emitters.length : 0;

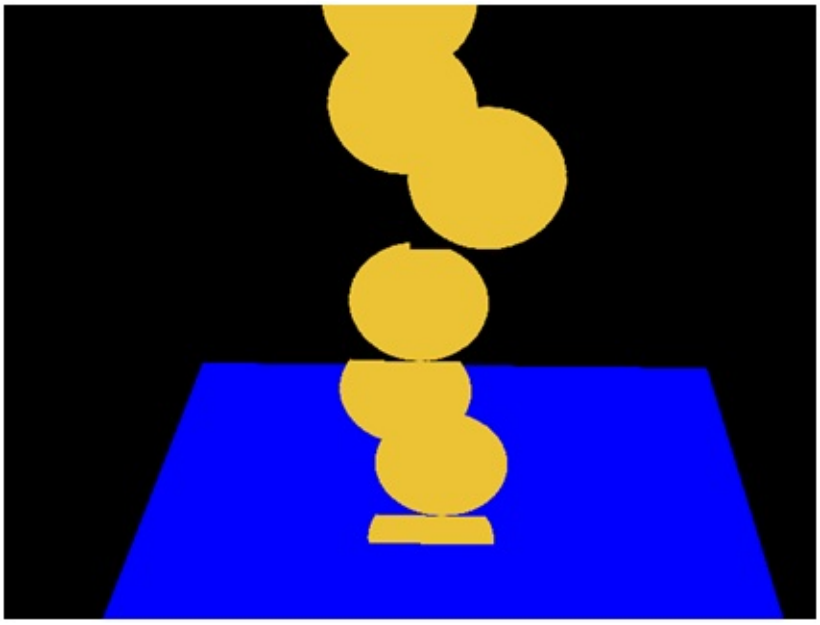
    for (int i = 0; i < numEmitters; i++) {
        IParticleEmitter emitter = emitters[i];
        Mesh mesh = emitter.getBaseParticle().getMesh();

        mesh.renderList((emitter.getParticles()), (GameItem gameItem) -> {
            Matrix4f modelViewMatrix = transformation.buildModelViewMatrix(gameItem, v
iewMatrix);
            particlesShaderProgram.setUniform("modelViewMatrix", modelViewMatrix);
        }
        );
    }
    particlesShaderProgram.unbind();
}
```

The fragment above should be self explanatory if you managed to get to this point, it just renders each particle setting up the required uniforms. We have now created all the methods we need to test the implementation of the particle effect. We just need to modify the `DummyGame` class we can setup a particle emitter and the characteristics of the base particle.

```
Vector3f particleSpeed = new Vector3f(0, 1, 0);
particleSpeed.mul(2.5f);
long ttl = 4000;
int maxParticles = 200;
long creationPeriodMillis = 300;
float range = 0.2f;
float scale = 0.5f;
Mesh partMesh = OBJLoader.loadMesh("/models/particle.obj");
Texture texture = new Texture("/textures/particle_tmp.png");
Material partMaterial = new Material(texture, reflectance);
partMesh.setMaterial(partMaterial);
Particle particle = new Particle(partMesh, particleSpeed, ttl);
particle.setScale(scale);
particleEmitter = new FlowParticleEmitter(particle, maxParticles, creationPeriodMillis
);
particleEmitter.setActive(true);
particleEmitter.setPositionRndRange(range);
particleEmitter.setSpeedRndRange(range);
this.scene.setParticleEmitters(new FlowParticleEmitter[] {particleEmitter});
```

We are using a plain filled circle as the particle's texture by now, to better understand what's happening. If you execute it you will see something like this.



Why some particles seem to be cut off? Why the transparent background does not solve this? The reason for that is depth testing. Some fragments of the particles get discarded because they have a depth buffer value higher than the current value of the depth buffer for that zone. We can solve this by ordering the particle drawings depending in their distance to the camera or we can just disable the depth writing.

Before we draw the particles we just need to insert this line:

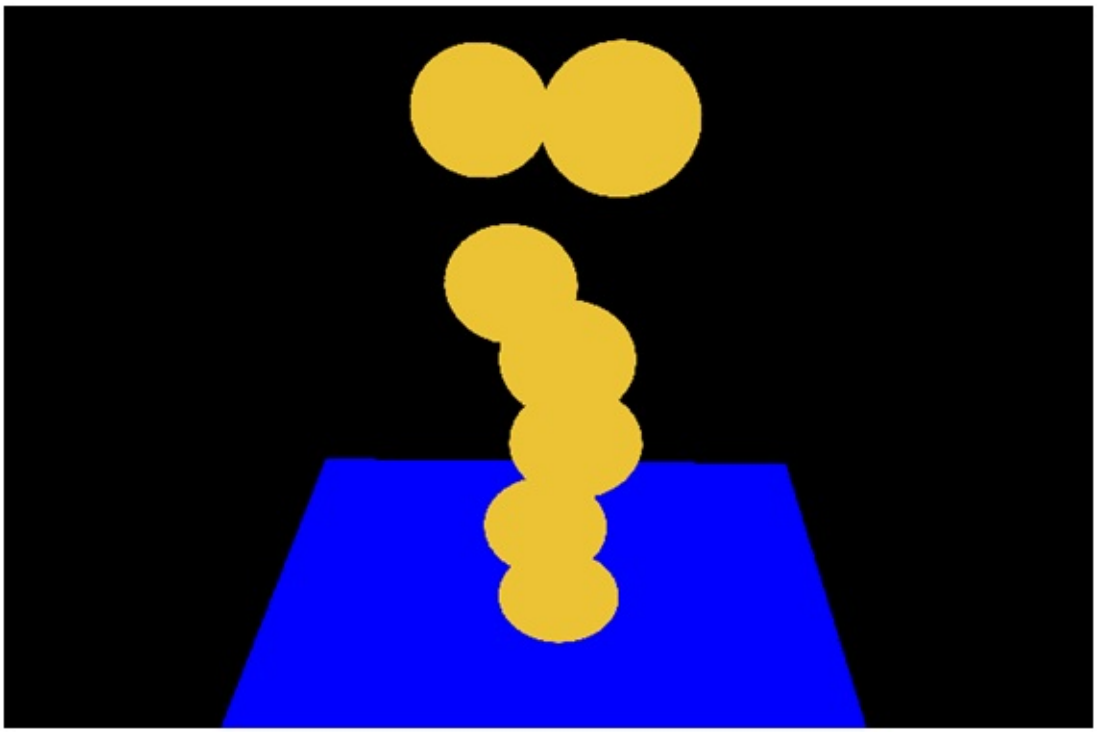
```
glDepthMask(false);
```

And when we are done with rendering we restore the previous value:

```
glDepthMask(true);
```

Then we will get something like this.

www.dbooks.org



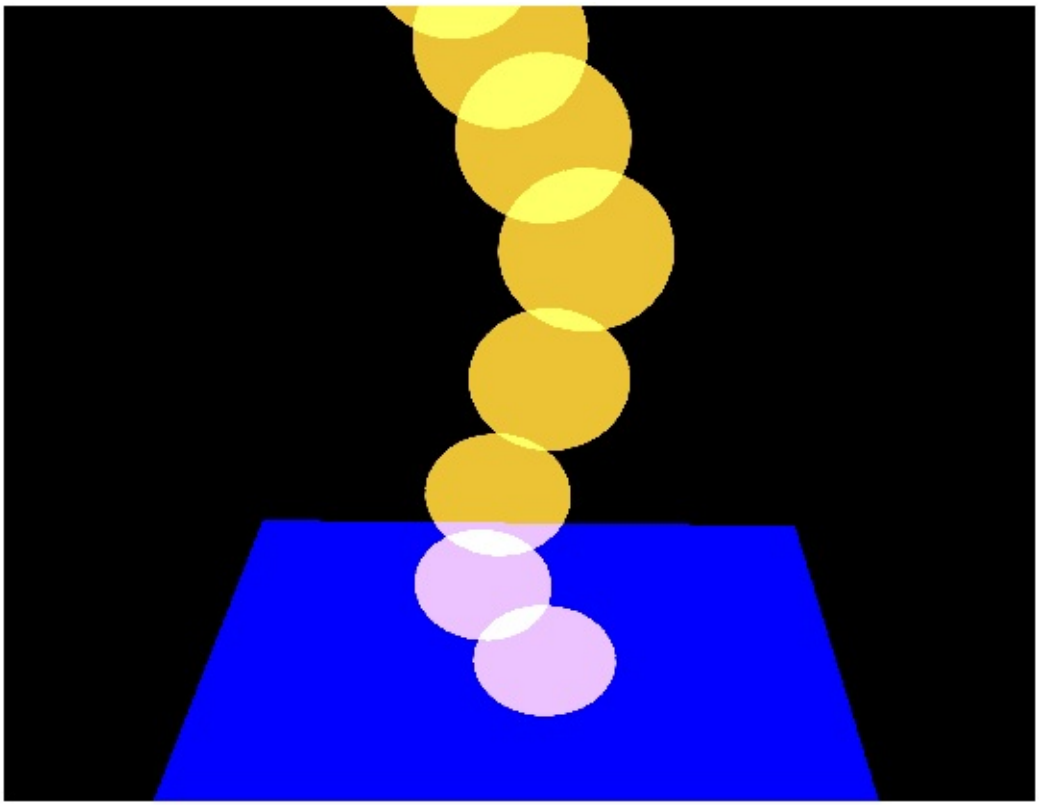
Ok, problem solved. Nevertheless, we still want another effect to be applied, we would want that colours get blended so colours will be added to create better effects. This is achieved with by adding this line before rendering to setup additive blending.

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE);
```

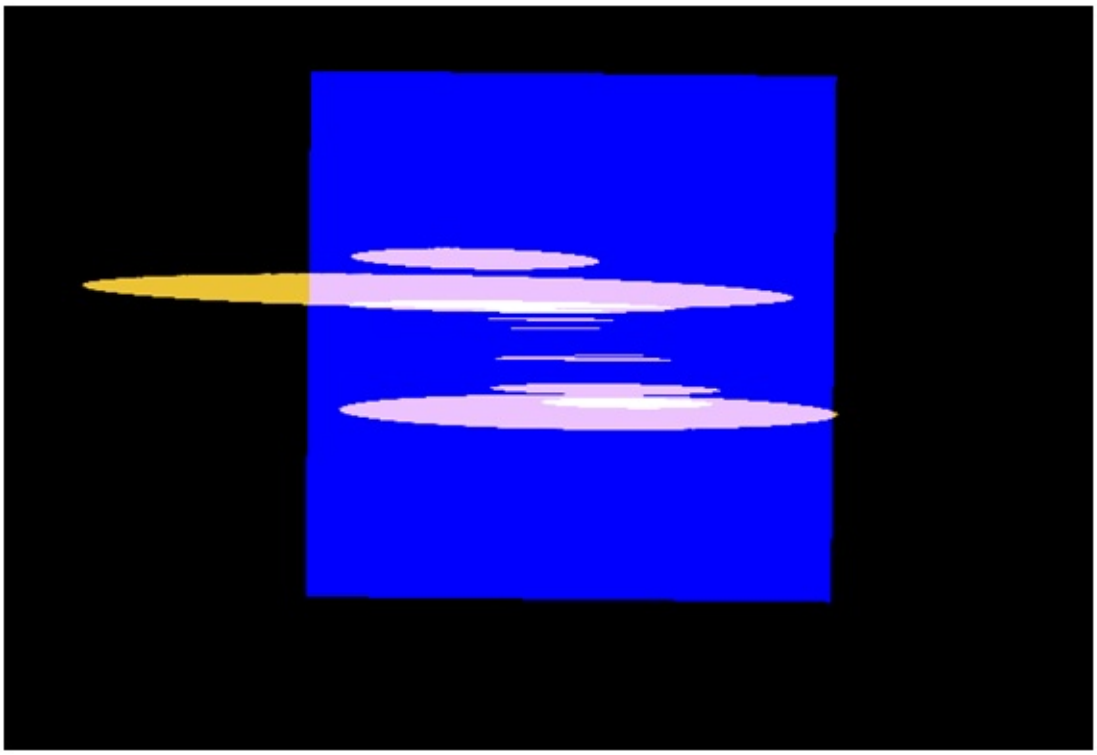
As in the depth case, after we have rendered all the particles we restore the blending function to:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Now we get something like this.



But we have not finished yet. If you have moved the camera over the blue square looking down you may have got something like this.



The particles do not look very good, they should look round but they resemble a sheet of paper. At this points is where we should be applying the billboard technique. The quad that is used to render the particle should always be facing the camera, totally perpendicular to it

www.dbooks.org

as if it there was no rotation at all. The camera matrix applies translation and rotation to every object in the scene, we want to skip the rotation to be applied.

Warning: Maths ahead, you can skip it if you don't feel comfortable with this. Let's review that view matrix once again. That matrix can be represented like this (without any scale applied to it).

$$\begin{bmatrix} r_{00} & r_{10} & r_{20} & dx \\ r_{01} & r_{11} & r_{21} & dy \\ r_{02} & r_{12} & r_{22} & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The red elements represent the camera rotation while the blue ones represent the translation. We need to cancel the effect of the upper left 3x3 matrix contained in the view matrix so it gets to something like this.

$$\begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

So, we have a 3x3 matrix, the upper left red fragment, let's name it M_r and we want it to transform it to the identify matrix: I. Any matrix multiplied by its inverse will give the identify matrix: $M_r \times M_r^{-1} = I$. So we just need to get the upper left 3x3 matrix from the view matrix, and multiply it by its inverse, but we can even optimize this. A rotation matrix has an interesting characteristic, its inverse coincides with its transpose matrix. That is:

$M_r \times M_r^{-1} = M_r \times M_r^T = I$. And a transpose matrix is much more easier to calculate than the inverse. The transpose of a matrix is like if we flip it, we change rows per columns.

$$\begin{bmatrix} r_{00} & r_{10} & r_{20} \\ r_{01} & r_{11} & r_{21} \\ r_{02} & r_{12} & r_{22} \end{bmatrix}^T = \begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix}$$

Ok, let's summarize. We have this transformation: $V \times M$, where V is the view matrix and M is the model matrix. We can express that expression like this:

$$\begin{bmatrix} v_{00} & v_{10} & v_{20} & v_{30} \\ v_{01} & v_{11} & v_{21} & v_{31} \\ v_{02} & v_{12} & v_{22} & v_{32} \\ v_{03} & v_{13} & v_{23} & v_{33} \end{bmatrix} \times \begin{bmatrix} m_{00} & m_{10} & m_{20} & m_{30} \\ m_{01} & m_{11} & m_{21} & m_{31} \\ m_{02} & m_{12} & m_{22} & m_{32} \\ m_{03} & m_{13} & m_{23} & m_{33} \end{bmatrix}$$

We want to cancel the rotation of the view matrix, to get something like this:

$$\begin{bmatrix} 1 & 0 & 0 & mv_{30} \\ 0 & 1 & 0 & mv_{31} \\ 0 & 0 & 1 & mv_{32} \\ mv_{03} & mv_{13} & mv_{23} & mv_{33} \end{bmatrix}$$

So we just need to set the upper left 3x3 matrix for the model matrix as the transpose matrix of the 3x3 upper part of the view matrix:

$$\begin{bmatrix} v_{00} & v_{10} & v_{20} & v_{30} \\ v_{01} & v_{11} & v_{21} & v_{31} \\ v_{02} & v_{12} & v_{22} & v_{32} \\ v_{03} & v_{13} & v_{23} & v_{33} \end{bmatrix} \times \begin{bmatrix} v_{00} & v_{01} & v_{02} & m_{30} \\ v_{10} & v_{11} & v_{12} & m_{31} \\ v_{20} & v_{21} & v_{22} & m_{32} \\ m_{03} & m_{13} & m_{23} & m_{33} \end{bmatrix}$$

But, after doing this, we have removed the scaling factor, indeed what we do really whant to achieve is something like this:

$$\begin{bmatrix} sx & 0 & 0 & mv_{30} \\ 0 & sy & 0 & mv_{31} \\ 0 & 0 & sz & mv_{32} \\ mv_{03} & mv_{13} & mv_{23} & mv_{33} \end{bmatrix}$$

Where sx, sy and sz are the scaling factor. Thus, after we have set set the upper left 3x3 matrix for the model matrix as the transpose matrix of the view matrix, we need to apply scaling again.

And that's all, we just need to change this in the `renderParticlesMethod` like this:

```
        for (int i = 0; i < numEmitters; i++) {
            IParticleEmitter emitter = emitters[i];
            Mesh mesh = emitter.getBaseParticle().getMesh();

            mesh.renderList((emitter.getParticles()), (GameItem gameItem) -> {
                Matrix4f modelMatrix = transformation.buildModelMatrix(gameItem);

                viewMatrix.transpose3x3(modelMatrix);

                Matrix4f modelViewMatrix = transformation.buildModelViewMatrix(modelMa
trix, viewMatrix);
                modelViewMatrix.scale(gameItem.getScale());
                particlesShaderProgram.setUniform("modelViewMatrix", modelViewMatrix);
            }
            );
        }
```

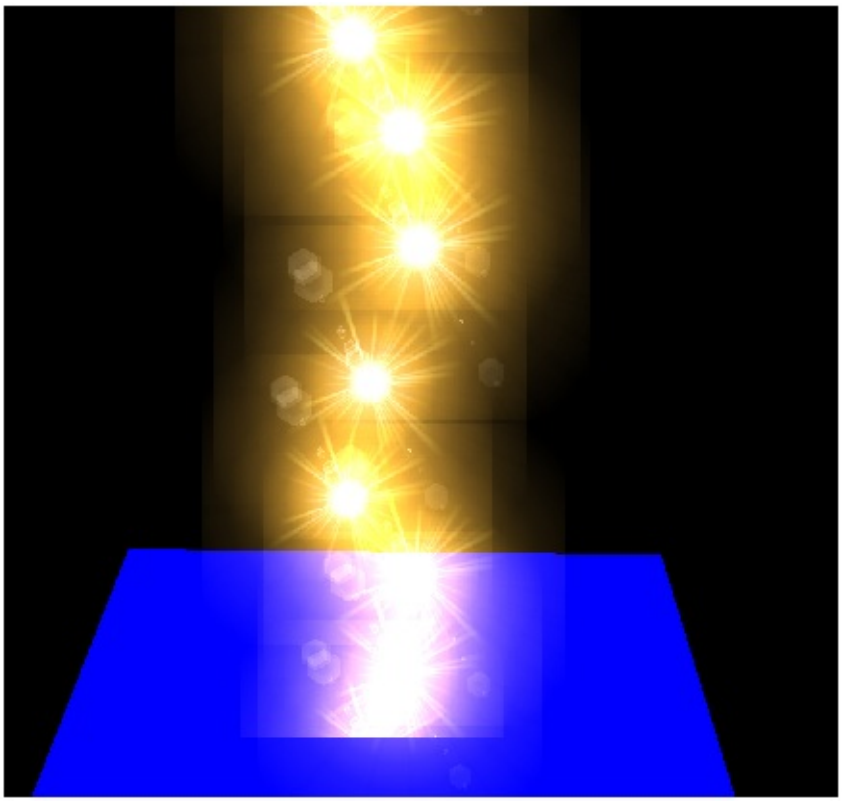
We also have added another method to the `Transformation` class to construct a model view matrix using two matrices instead of a `GameItem` and the view matrix.

With that change, when we look the particles from above we get something like this.



Now we have everything we need to create a more realistic particle effect so let's change the texture to something more elaborated. We will use this image (it was created with GIMP with the lights and shadows filters).

With this texture, we will get something like this.



Much better ! You may notice that we need to adjust the scale, since particles are now always facing the camera the displayed area is always the maximum.

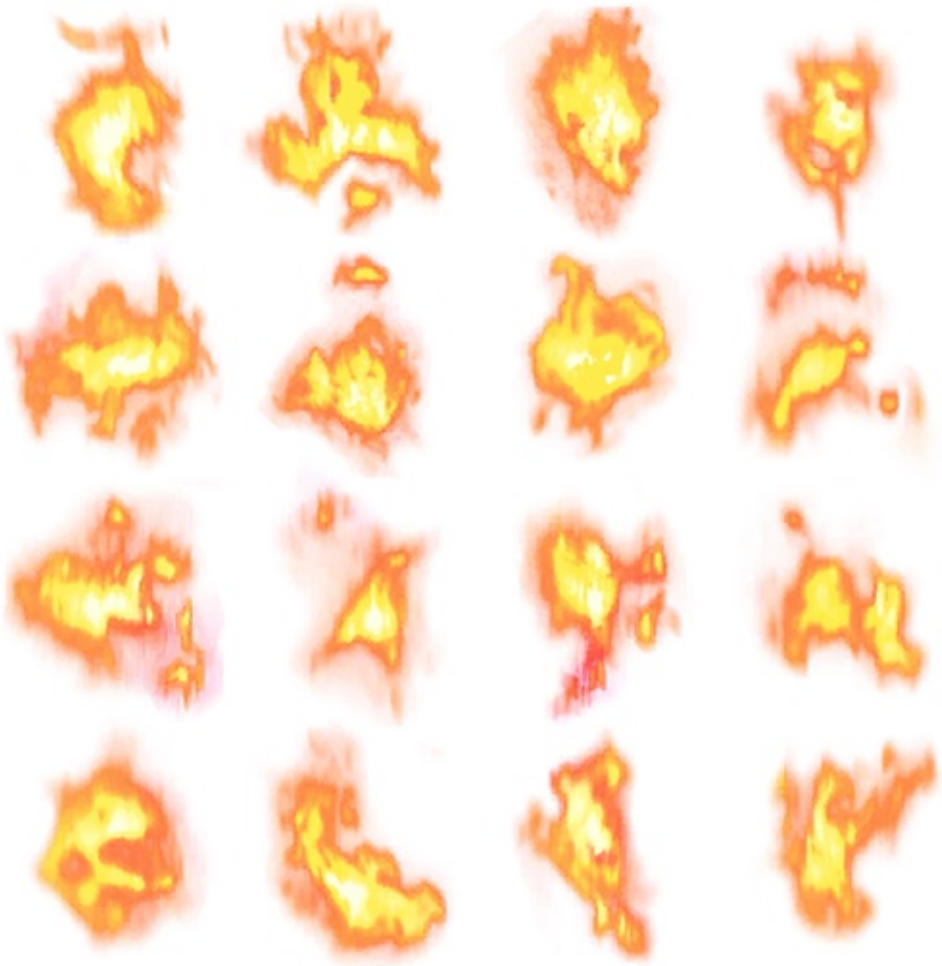
Finally, another conclusion, to get perfect results which can be used in any scene you will need to implement particle ordering and activate depth buffer. In any case, you have here a sample to include this effect in your games.

Texture Atlas

Now that we have set the basic infrastructure for particle effect we can add some animation effects to it. In order to achieve that, we are going to support texture atlas. A texture atlas is a large image that contains all the textures that will be used. With a texture atlas we need only to load a large image and then while drawing the game items we select the portions of that image to be used as our texture. This technique can be applied for instance when we want to represent the same model many times with different textures (think for instance about trees, or rocks). Instead of having many texture instances and switching between them (remember that switching states are always slow) we can use the same texture atlas and just select the appropriate coordinates.

In this case, we are going to use texture coordinates to animate particles. We will iterate over different textures to model a particle animation. All those textures will be grouped into a texture atlas which looks like this.

www.dbooks.org



The texture atlas can be divided into quad tiles. We will assign a tile position to a particle and will change it over time to represent animation. So let's get on it. The first thing that we are going to do is modifying the `Texture` class to specify the number of rows and columns that a texture atlas can have.

```
package org.lwjglb.engine.graph;

// .. Imports hera

public class Texture {

    // More attributes here
    private int numRows = 1;

    private int numCols = 1;

   // More code here

    public Texture(String fileName, int numCols, int numRows) throws Exception  {
        this(fileName);
        this.numCols = numCols;
        this.numRows = numRows;
    }
```

The default case is to have a texture with a number of columns and rows equal to 1, that is, the textures we have dealing with. We also add another constructor to be able to specify the rows and columns.

Then we need to keep track the position in the texture atlas for a `GameItem`, so we just add another attribute to that class with a default value equal to 0.

```
package org.lwjglb.engine.items;

import org.joml.Vector3f;
import org.lwjglb.engine.graph.Mesh;

public class GameItem {

    // More attributes here

    private int textPos;
```

Then we will modify the `Particle` class to be able to iterate automatically through a texture atlas.

```
package org.lwjglb.engine.graph.particles;

import org.joml.Vector3f;
import org.lwjglb.engine.graph.Mesh;
import org.lwjglb.engine.graph.Texture;
import org.lwjglb.engine.items.GameItem;

public class Particle extends GameItem {

    private long updateTextureMillis;

    private long currentAnimTimeMillis;
```

The `updateTextureMillis` attribute models the period of time (in milliseconds) to move to the next position in the texture atlas. The lowest the value the fastest the particle will roll over the textures. The `currentAnimTimeMillis` attribute just keeps track of the time that the particle has maintained a texture position.

Thus, we need to modify the `Particle` class constructor to set up those values. Also we calculate the number of tiles of the texture atlas, which is modelled by the attribute `animFrames`.

www.dbooks.org

```
public Particle(Mesh mesh, Vector3f speed, long ttl, long updateTextureMillis) {
    super(mesh);
    this.speed = new Vector3f(speed);
    this.ttl = ttl;
    this.updateTextureMills = updateTextureMills;
    this.currentAnimTimeMillis = 0;
    Texture texture = this.getMesh().getMaterial().getTexture();
    this.animFrames = texture.getNumCols() * texture.getNumRows();
}
```

Now, we just need to modify the method that checks if the particle has expired to check also if we need to update the texture position.

```
public long updateTtl(long elapsedTime) {
    this.ttl -= elapsedTime;
    this.currentAnimTimeMillis += elapsedTime;
    if ( this.currentAnimTimeMillis >= this.getUpdateTextureMillis() && this.animFrame
s > 0 ) {
        this.currentAnimTimeMillis = 0;
        int pos = this.getTextPos();
        pos++;
        if ( pos < this.animFrames ) {
            this.setTextPos(pos);
        } else {
            this.setTextPos(0);
        }
    }
    return this.ttl;
}
```

Besides that, we also have modified the `FlowRangeEmitter` class to add some randomness to the period of time when we should change the a particle's texture position. You can check it in the source code.

Now we can use that information to set up appropriate texture coordinates. We will do this in the vertex fragment since it outputs those values to be used in the fragment shader. The new version of that shader is defined like this.

```
#version 330

layout (location=0) in vec3 position;
layout (location=1) in vec2 texCoord;
layout (location=2) in vec3 vertexNormal;

out vec2 outTexCoord;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;

uniform float texXOffset;
uniform float texYOffset;
uniform int numCols;
uniform int numRows;

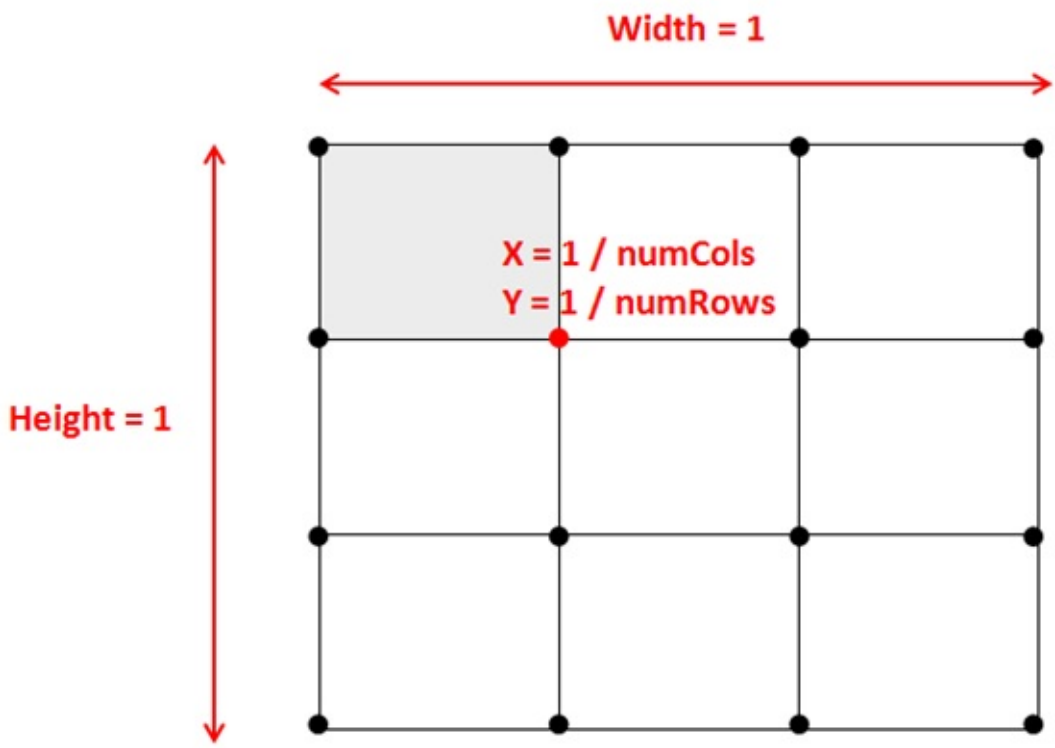
void main()
{
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);

    // Support for texture atlas, update texture coordinates
    float x = (texCoord.x / numCols + texXOffset);
    float y = (texCoord.y / numRows + texYOffset);

    outTexCoord = vec2(x, y);
}
```

As you can see we have now three new uniforms. The uniforms `numCols` and `numRows` just contain the number of columns and rows of the texture atlas. In order to calculate the texture coordinates, we first must scale down these parameters. Each tile will have a width which is equal to $1/numCols$ and a height which is equal to $1/numRows$ as shown in the next figure.

www.dbooks.org

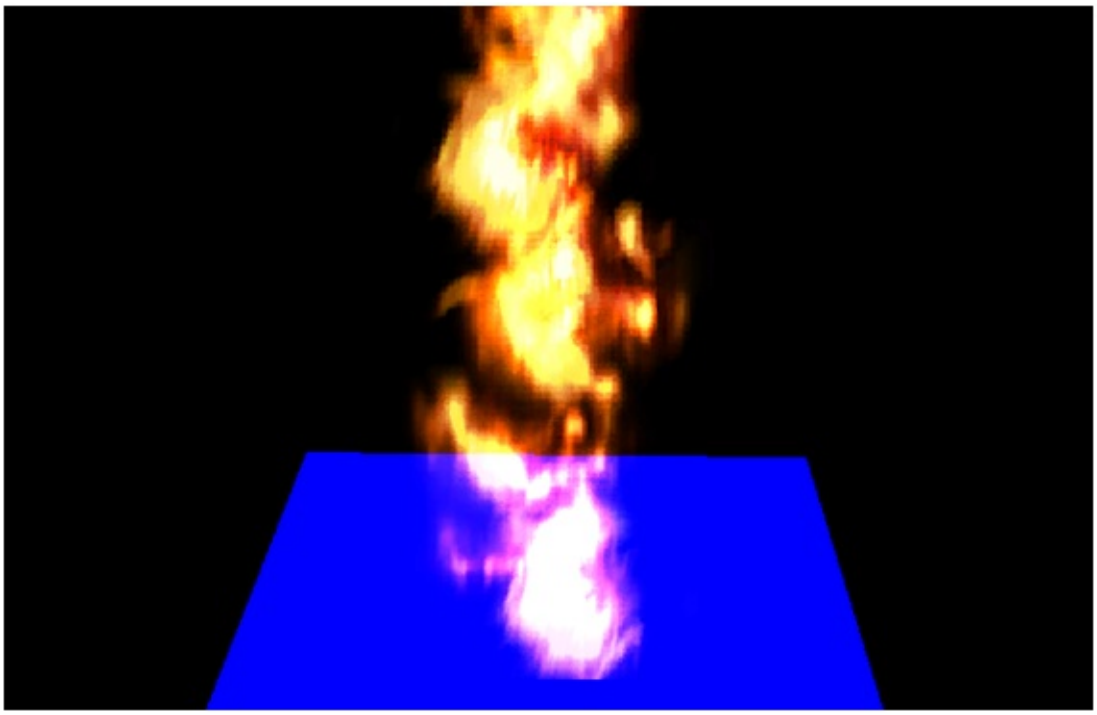


Then we just need to apply and offset depending on the row and column, this is what is modelled by the `texXOffset` and `texYOffset` uniforms.

We will calculate these offsets in the `Renderer` class as shown in the next fragment. We calculate the row and column that each particle is in according to its position and calculate the offset accordingly as a multiple of tile's width and height.

```
mesh.renderList((emitter.getParticles()), (GameItem gameItem) -> {
    int col = gameItem.getTextPos() % text.getNumCols();
    int row = gameItem.getTextPos() / text.getNumCols();
    float textXOffset = (float) col / text.getNumCols();
    float textYOffset = (float) row / text.getNumRows();
    particlesShaderProgram.setUniform("texXOffset", textXOffset);
    particlesShaderProgram.setUniform("texYOffset", textYOffset);
```

Note that if you only need to support perfectly square texture atlas, you will only need two uniforms. The final result looks like this.



Now we have animated particles working. In the next chapter we will learn how to optimize the rendering process. We are rendering multiple elements that have the same mesh and we are performing a drawing call for each of them. In the next chapter we will learn how to do it in a single call. That technique is useful for particles but also for rendering scenes where multiple elements share the same model but are placed in different locations or have different textures.

www.dbooks.org

Instanced Rendering

Lots of Instances

When drawing a 3D scene is frequent to have many models represented by the same mesh but with different transformations. In this case, although they may be simple objects with just a few triangles, performance can suffer. The cause behind this is the way we are rendering them.

We are basically iterating through all the game items inside a loop and performing a call to the function `glDrawElements`. As it has been said in previous chapters, calls to OpenGL library should be minimized. Each call to the `glDrawElements` function imposes an overhead that is repeated again and again for each `GameItem` instance.

When dealing with lots of similar objects it would be more efficient to render all of them using a single call. This technique is called instanced rendering. In order to acomplish that OpenGL provides a set of functions named `glDrawXXXInstanced` to render a set of elements at once. In our case, since we are drawing elements we will use the function named `glDrawElementsInstanced`. This function receives the same arguments as the `glDrawElements` plus one additional parameter which sets the number of instances to be drawn.

This is a sample of how the glDrawElements is used.

```
glDrawElements(GL_TRIANGLES, numVertices, GL_UNSIGNED_INT, 0)
```

And this is how the instanced version can be used:

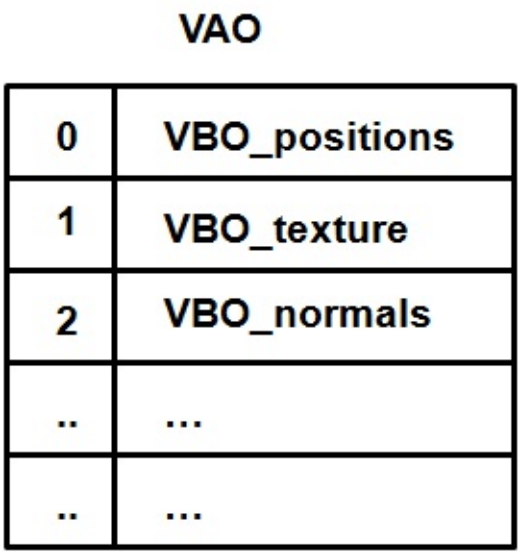
```
glDrawElementsInstanced(GL_TRIANGLES, numVertices, GL_UNSIGNED_INT, 0, numInstances);
```

But you may be wondering now how can you set the different transformations for each of those instances. Now, before we draw each instance we pass the different transformations and instance related data using uniforms. Before a render call is made we need to setup the specific data for each item. How can we do this when rendering all of them at once?

When using instanced rendering, in the vertex shader we can use an input variable that holds the index of the instance that is currently being drawn. With that built-in variable we can, for instance, pass an array of uniforms containing the transformations to be applied to each instance and use a single render call.

The problem with this approach is that it still imposes too much overhead. In addition to that, the number of uniforms that we can pass is limited. Thus, we need to emply another approach, instead of using lists of uniforms we will use instanced arrays.

If you recall from the first chapters, the data for each Mesh is defined by a set of arrays of data named VBOs. The data store in those VBOs is unique per Mesh instance.

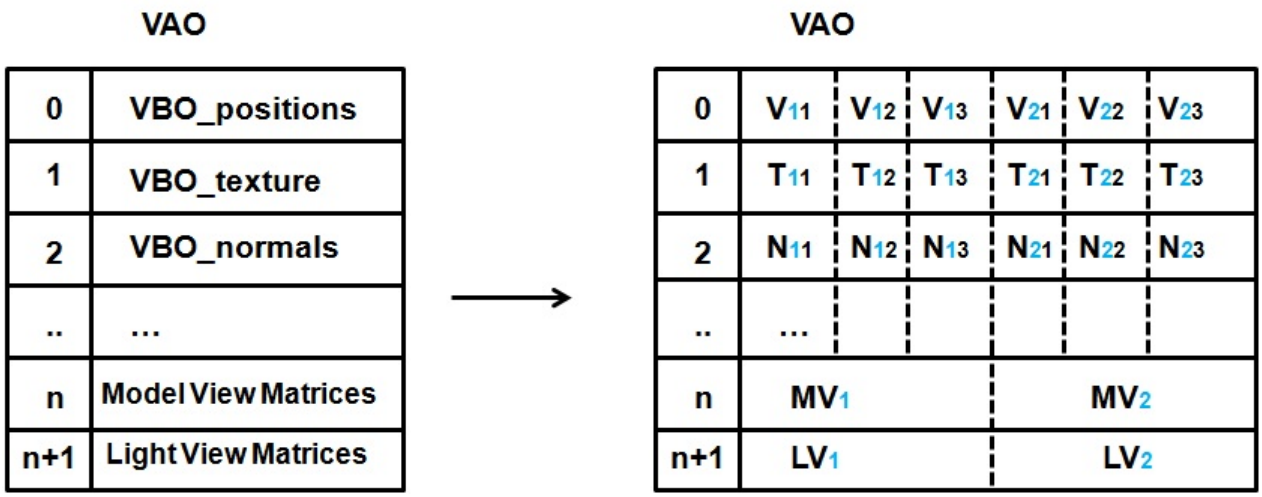


With standard VBOs, inside a shader, we can access the data associated to each vertex (its position, colour, textue, etc.). Whenever the shader is run, the input variables are set to point to the specific data associated to each vertex. With instanced arrays we can set up data that is changed per instance instead of per vertex. If we combine both types we can use regular VBOs to store per vertex information (position, texture coordinates) and VBOs that contain per instance data such as model view matrices.

The next figure shows a Mesh composed by three per vertex VBOs definig the positions, textures and normals. The first index of each of those elements is the instance that it belongs to (in blue colour). The second index represents the vertex position inside a instance.

The Mesh is also defined by two per instance VBOs. One for the model view matrix and the other one for the light view matrix. When rendering the vertices for the firs instance (the 1X, ones), the model view and light view matrices will be the same (the 1). When vertices of the second instance are to be rendered the second model view and light view matrices will be used.

www.dbooks.org



Thus, when rendering the first vertex of the first instance, V11, T11 and N11 would be used for position, texture and normal data and MV1 would be used as a model view matrix. When rendering the second vertex of the same first instance, V12, T12 and N12 would be used for position, texture and normal dara and MV1 wouls still be used as a model view matrix. MV2 and LV2 would not be used until second instance is rendered.

In order to define per instance data we need to call the function `glVertexAttribDivisor` after defining vertex attributes. This function receives two parameters:

- index: The index of the vertex attribute (as issued in the glVertexAttribPointer function).
- Divisor: If this vale contains zero, the data is changed for each vertex while rendering. If it is set to one, the data changes once per instance. If it's set to two it changes every two instances, etc.

So, in order to set data for a instance we need to perform this call after every attribute definition:

```
glVertexAttribDivisor(index, 1);
```

Let's start changing our code base to support instanced rendering. The first step is to create a new class named `InstancedMesh` that inherits from the `Mesh` class. The constructor of this class will be similar to the similar to the `Mesh` one but with an extra parameter, the number of instances.

In the constructor, besides relying in super's constructor, we will create two new VBOs, one for the model view matrix and another one for the light view matrix. The code for creating the model view matrix is presented below.

```
modelViewVBO = glGenBuffers();
vboIdList.add(modelViewVBO);
this.modelViewBuffer = MemoryUtil.memAllocFloat(numInstances * MATRIX_SIZE_FLOATS);
glBindBuffer(GL_ARRAY_BUFFER, modelViewVBO);
int start = 5;
for (int i = 0; i < 4; i++) {
    glVertexAttribPointer(start, 4, GL_FLOAT, false, MATRIX_SIZE_BYTES, i * VECTOR4F_S
IZE_BYTES);
    glVertexAttribDivisor(start, 1);
    start++;
}
```

The first thing that we do is create a new VBO and a new `FloatBuffer` to store the data on it. The size of that buffer is measured in floats, so it will be equal to the number of instances multiplied by the size in floats of a 4x4 matrix, which is equal to 16.

Once the VBO has been bond we start defining the attributes for it. You can see that this is done in a for loop that iterates four times. Each turn of the loop defines one vector the matrix. Why not simply defining a single attribute for the whole matrix? The reason for that is that a vertex attribute cannot contain more than four floats. Thus, we need to split the matrix definition into four pieces. Let's refresh the parameters of the `glVertexAttribPointer`:

- Index: The index of the element to be defined.
- Size: The number of components for this attribute. In this case it's 4, 4 floats, which is the maximum accepted value.
- Type: The type of data (floats in our case).
- Normalize: If fixed-point data should be normalized or not.
- Stride: This is important to understand here, this sets the byte offsets between consecutive attributes. In this case, we need to set it to the whole size of a matrix in bytes. This acts like a mark that packs the data so it can be changed between vertex or instances.
- Pointer: The offset that this attribute definition applies to. In our case, we need to split the matrix definition into four calls. Each vector of the matrix increments the offset.

After defining the vertex attribute, we need to call the `glVertexAttribDivisor` using the same index.

The definition of the light view matrix is similar to the previous one, you can check it in the source code. Continuing with the `InstancedMesh` class definition it's important to override the methods that enable the vertex attributes before rendering (and the one that disables them after).

www.dbooks.org

```
@Override
protected void initRender() {
    super.initRender();
    int start = 5;
    int numElements = 4 * 2;
    for (int i = 0; i < numElements; i++) {
        glEnableVertexAttribArray(start + i);
    }
}

@Override
protected void endRender() {
    int start = 5;
    int numElements = 4 * 2;
    for (int i = 0; i < numElements; i++) {
        glDisableVertexAttribArray(start + i);
    }
    super.endRender();
}
```

The `InstancedMesh` class defines a public method, named `renderListInstanced`, that renders a list of game items, this method splits the list of game items into chunks of size equal to the number of instances used to create the `InstancedMesh`. The real rendering method is called `renderChunkInstanced` and is defined like this.

```
private void renderChunkInstanced(List<GameItem> gameItems, boolean depthMap, Transfor
mation transformation, Matrix4f viewMatrix, Matrix4f lightViewMatrix) {
    this.modelViewBuffer.clear();
    this.modelLightViewBuffer.clear();
    int i = 0;
    for (GameItem gameItem : gameItems) {
        Matrix4f modelMatrix = transformation.buildModelMatrix(gameItem);
        if (!depthMap) {
            Matrix4f modelViewMatrix = transformation.buildModelViewMatrix(modelMatrix
, viewMatrix);
            modelViewMatrix.get(MATRIX_SIZE_FLOATS * i, modelViewBuffer);
        }
        Matrix4f modelLightViewMatrix = transformation.buildModelLightViewMatrix(model
Matrix, lightViewMatrix);
        modelLightViewMatrix.get(MATRIX_SIZE_FLOATS * i, this.modelLightViewBuffer);
        i++;
    }
    glBindBuffer(GL_ARRAY_BUFFER, modelViewVBO);
    glBufferData(GL_ARRAY_BUFFER, modelViewBuffer, GL_DYNAMIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, modelLightViewVBO);
    glBufferData(GL_ARRAY_BUFFER, modelLightViewBuffer, GL_DYNAMIC_DRAW);
    glDrawElementsInstanced(GL_TRIANGLES, getVertexCount(), GL_UNSIGNED_INT, 0, gameIt
ems.size());
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

The method is quite simple, we basically iterate over the game items and calculate the model view and light view matrices. These matrices are dumped into their respective buffers. The contents of those buffers are sent to to the GPU and finally we render all of them with a single call to the `glDrawElementsInstanced` method.

Going back to the shaders, we need to modify the vertex shader to support instanced rendering. We will first add new input parameters for the model and view matrices that will be passed when using instanced rendering.

```
layout (location=5) in mat4 modelViewInstancedMatrix;
layout (location=9) in mat4 modelLightViewInstancedMatrix;
```

As you can see, the model view matrix starts at location 5. Since a matrix is defined by a set of four attributes (each one containing a vector), the light view matrix starts at location 9. Since we want to use a single shader for both non instanced and instanced rendering, we will maintain the uniforms for model and light view matrices. We only need to change their names.

www.dbooks.org

```
uniform int isInstanced;
uniform mat4 modelViewNonInstancedMatrix;
…
uniform mat4 modelLightViewNonInstancedMatrix;
```

We have created another uniform to specify if we are using instanced rendering or not. In the case we are using instanced rendering the code is very simple, we just use the matrices from the input parameters.

```
void main()
{
    vec4 initPos = vec4(0, 0, 0, 0);
    vec4 initNormal = vec4(0, 0, 0, 0);
    mat4 modelViewMatrix;
    mat4 lightViewMatrix;
    if ( isInstanced > 0 )
    {
        modelViewMatrix = modelViewInstancedMatrix;
        lightViewMatrix = modelLightViewInstancedMatrix;
        initPos = vec4(position, 1.0);
        initNormal = vec4(vertexNormal, 0.0);
    }
```

We don’t support animations for instanced rendering to simplify the example, but this technique can be perfectly used for this.
Finally, the shader just set up appropriate values as usual.

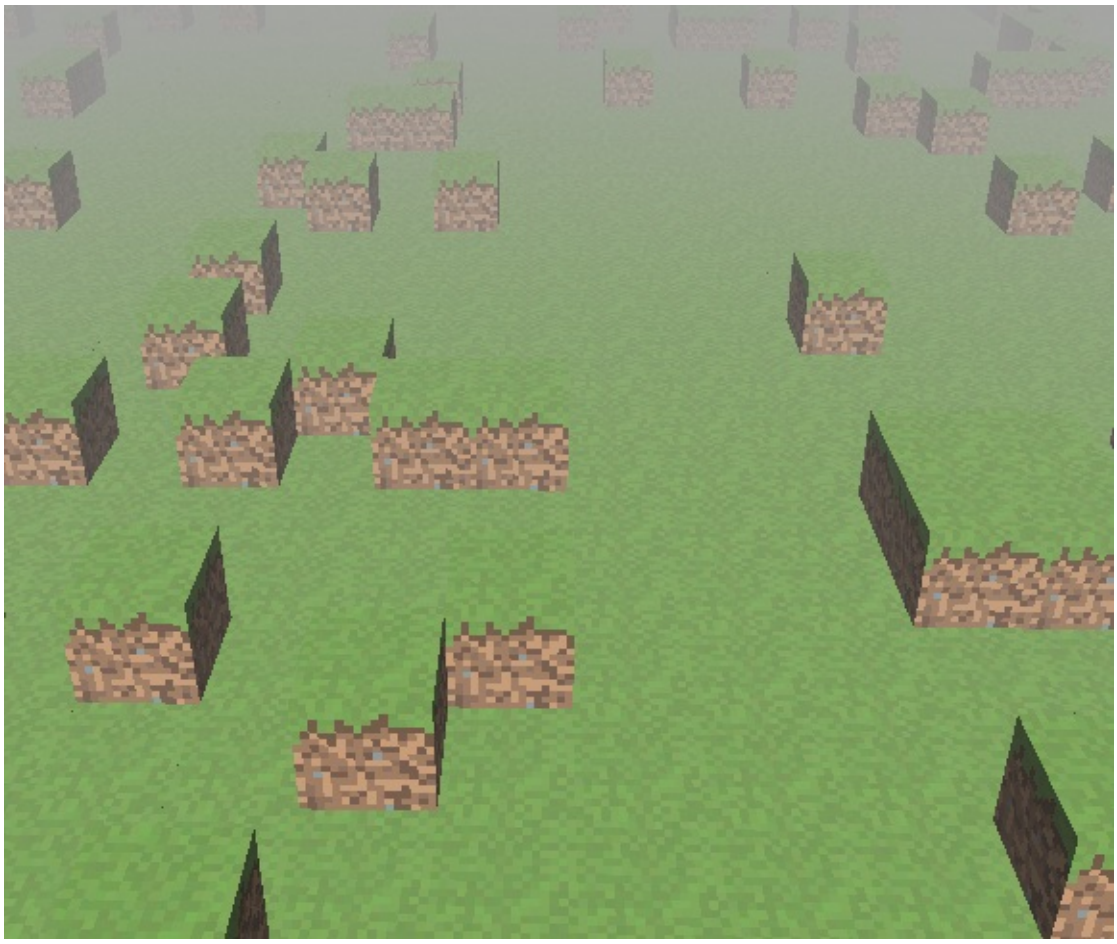
```
    vec4 mvPos = modelViewMatrix * initPos;
    gl_Position = projectionMatrix * mvPos;
    outTexCoord = texCoord;
    mvVertexNormal = normalize(modelViewMatrix * initNormal).xyz;
    mvVertexPos = mvPos.xyz;
    mlightviewVertexPos = orthoProjectionMatrix * lightViewMatrix * initPos;
    outModelViewMatrix = modelViewMatrix;
}
```

Of course, the `Renderer` has been modified to support the uniforms changes and to separate the rendering of non instanced meshes from the instanced ones. You can check the changes in the source code.

In addition to that, some optimizations have been added to the source code by the JOML author Kai Burjack. These optimizations have been applied to the `Transformation` class and are summarized in the following list:

- Removed redundant calls to set up matrices with identity values.

- Use quaternions for rotations which are more efficient.
- Use specific methods for rotating and translating matrices which are optimized for those operations.



Particles revisited

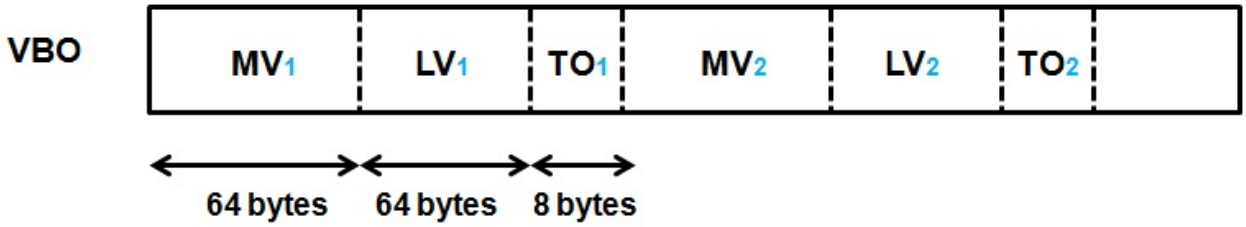
With the support of instanced rendering we can also improve the performance for the particles rendering. Particles are the best use case for this.

In order to apply instance rendering to particles we must provide support for texture atlas. This can be achieved by adding a new VBO with texture offsets for instanced rendering. The texture offsets can be modeled by a single vector of tow floats, so there's no need to split the definition as in the matrices case.

```
// Texture offsets
glVertexAttribPointer(start, 2, GL_FLOAT, false, INSTANCE_SIZE_BYTES, strideStart);
glVertexAttribDivisor(start, 1);
```


www.dbooks.org

But, instead of adding a new VBO we will set all the instance attributes inside a single VBO. The next figure shows the concept. We are packing up all the attributes inside a single VBO. The values will change per each instance.



In order to use a single VBO we need to modify the attribute size for all the attributes inside an instance. As you can see from the code above, the definition of the texture offsets uses a constant named `INSTANCE_SIZE_BYTES`. This constant is equal to the size in bytes of two matrices (one for the view model and the other one for the light view model) plus two floats (texture offesets), which in total is 136. The stride also needs to be modified properly.

You can check the modifications in the source code.

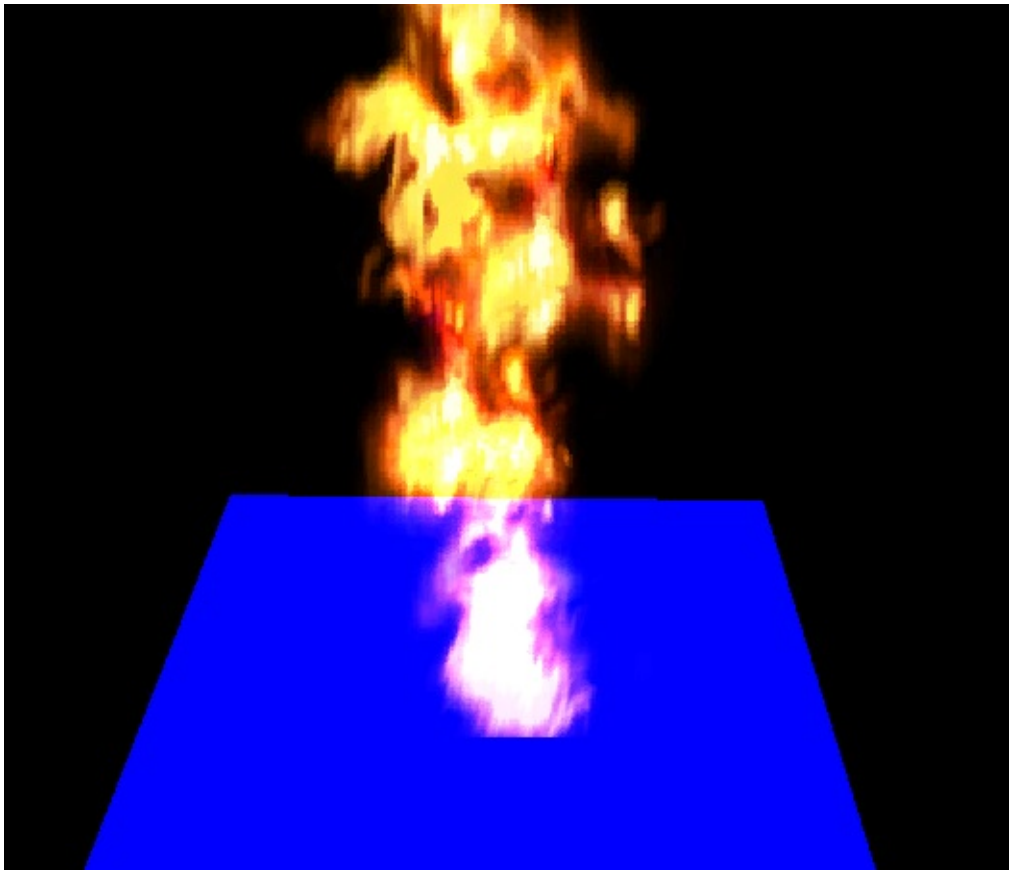
The `Renderer` class needs also to be modified to use instanced rendering for particles and support texture atlas in scene rendering. In this case, there's no sense in support both types of rendering (non instance and instanced), so the modifications are simpler.

The vertex shader for particles is also straight froward.

```
#version 330
layout (location=0) in vec3 position;
layout (location=1) in vec2 texCoord;
layout (location=2) in vec3 vertexNormal;
layout (location=5) in mat4 modelViewMatrix;
layout (location=13) in vec2 texOffset;
out vec2 outTexCoord;
uniform mat4 projectionMatrix;
uniform int numCols;
uniform int numRows;
void main()
{
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);

    // Support for texture atlas, update texture coordinates
    float x = (texCoord.x / numCols + texOffset.x);
    float y = (texCoord.y / numRows + texOffset.y);
    outTexCoord = vec2(x, y);}
```

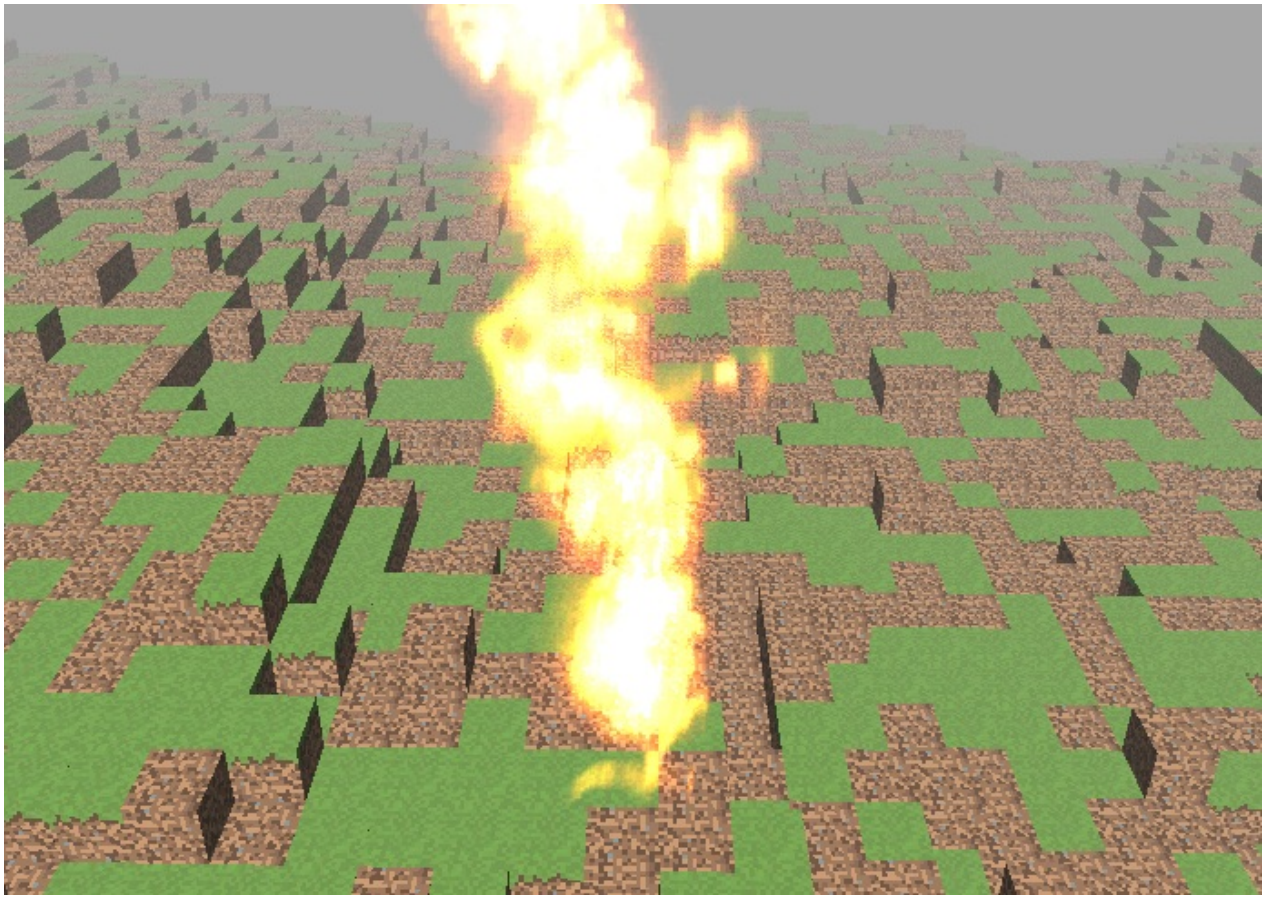
The results of this changes, look exactly the same as when rendering non instanced particles but the performance is much higher. A FPS counter has been added to the window title, as an option. You can play with instanced and non instanced rendering to see the improvements by yourself.



Extra bonus

With all the infrastructure that we have right now, I've modified the rendering cubes code to use a height map as a base, using also texture atlas to use different textures. It also combines particles rendering. It looks like this.

www.dbooks.org



Please keep in mind that there's still much room for optimization, but the aim of the book is guiding you in learning LWJGL and OpenGL concepts and techniques. The goal is not to create a full blown game engine (an definitely not a voxel engine, which require a different approach and more optimizations).

Audio

Until this moment we have been dealing with graphics, but another key aspect of every game is audio. This capability is going to be addressed in this chapter with the help of OpenAL (Open Audio Library). OpenAL is the OpenGL counterpart for audio, it allows us to play sounds through and abstraction layer. That layer isolates us from the underlying complexities of the audio subsystem. Besides that, it allows us to “render” sounds in a 3D scene, where sounds can be set up in specific locations, attenuated with the distance and modified according to their velocity (simulating Doppler effect)

LWJGL supports OpenAL without requiring any additional download, it’s just ready to use. But before start coding we need to present the main elements involved when dealing with OpenAL, which are:

- Buffers.
- Sources.
- Listener.

Buffers store audio data, that is, music or sound effects. They are similar to the textures in the OpenGL domain. OpenAL expects audio data to be in PCM (Pulse Coded Modulation) format (either in mono or in stereo), so we cannot just dump MP3 or OGG files without converting them first to PCM.

The next element are sources, which represent a location in a 3D space (a point) that emits sound. A source is associated to a buffer (only one at time) and can be defined by the following attributes:

- A position, the location of the source (x, y and z coordinates). By the way, OpenAL uses a right handed Cartesian coordinate system as OpenGL, so you can assume (to simplify things) that your world coordinates are equivalent to the ones in the sound space coordinate system.
- A velocity, which specifies how fast the source is moving. This is used to simulate Doppler effect.
- A gain, which is used to modify the intensity of the sound (it’s like an amplifier factor).

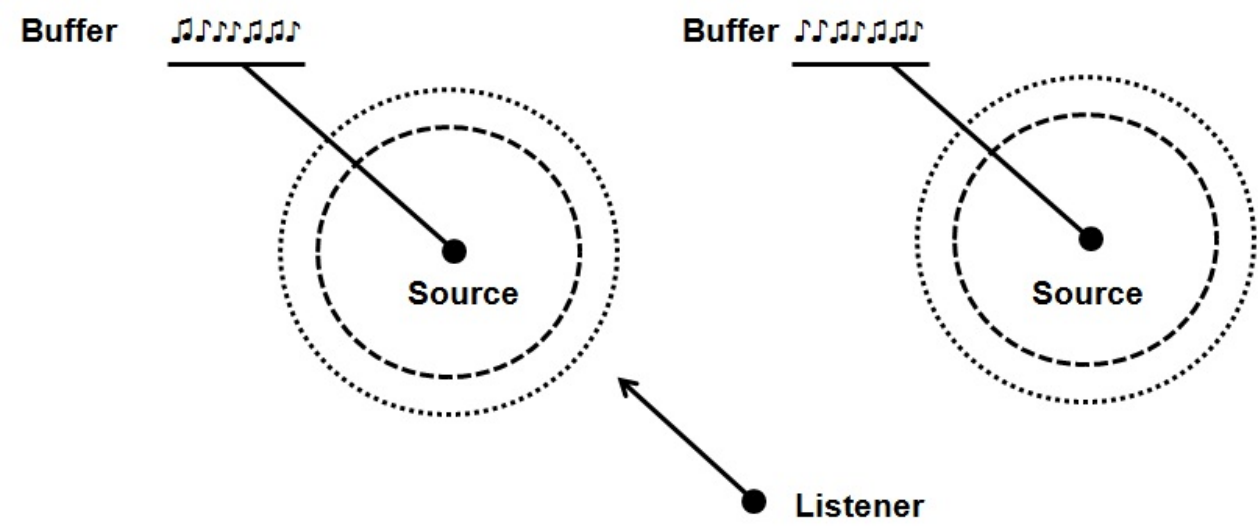
A source has additional attibutes which will be shown later when describing the source code.

And last, but no least, a listener which is where the generated sounds are supposed to be heard. The Listener represents were the microphone is set in a 3D audio scene to receive the sounds. There is only one listener. Thus, it’s often said that audio rendering is done form

www.dbooks.org

the listener's perspective. A listener shares some the attributes but it has some additional ones such as the orientation. The orientation represents where the listener is facing.

So an audio 3D scene is composed by a set of sound sources which emit sound and a listener that receives them. The final perceived sound will depend on the distance of the listener to the different sources, their relative speed and the selected propagation models. Sources can share buffers and play the same data. The following figure depicts a sample 3D scene with the different element types involved.



So, let's start coding, we will create a new package under the name `org.lwjglb.engine.sound` that will host all the classes responsible for handling audio. We will first start with a class, named `SoundBuffer` that will represent an OpenAL buffer. A fragment of the definition of that class is shown below.

```
package org.lwjglb.engine.sound;

// ... Some inports here

public class SoundBuffer {

    private final int bufferId;

    public SoundBuffer(String file) throws Exception {
        this.bufferId = alGenBuffers();
        try (STBVorbisInfo info = STBVorbisInfo.malloc()) {
            ShortBuffer pcm = readVorbis(file, 32 * 1024, info);

            // Copy to buffer
            alBufferData(buffer, info.channels() == 1 ? AL_FORMAT_MONO16 : AL_FORMAT_S
TEREO16, pcm, info.sample_rate());
        }
    }

    public int getBufferId() {
        return this.bufferId;
    }

    public void cleanup() {
        alDeleteBuffers(this.bufferId);
    }

    // ....
}
```

The constructor of the class expects a sound file (which may be in the classpath as the rest of resources) and creates a new buffer from it. The first thing that we do is create an OpenAL buffer with the call to `alGenBuffers`. At the end our sound buffer will be identified by an integer which is like a pointer to the data it holds. Once the buffer has been created we dump the audio data in it. The constructor expects a file in OGG format, so we need to transform it to PCM format. You can check how that's done in te source code, anyway, the source code has been extracted from the LWJGL OpenAL tests.

Previous versions of LWJGL had a helper class named `WaveData` which was used to load audio files in WAV format. This class is no longer present in LWJGL 3. Nevertheless, you may get the source code from that class and use it in your games (maybe without requiring any changes).

The `SoundBuffer` class also provides a `cleanup` method to free the resources when we are done with it.

Let's continue by modelling an OpenAl, which will be implemented by class named `SounSource`. The class is defined below.

www.dbooks.org

```
package org.lwjglb.engine.sound;

import org.joml.Vector3f;

import static org.lwjgl.openal.AL10.*;

public class SoundSource {

    private final int sourceId;

    public SoundSource(boolean loop, boolean relative) {
        this.sourceId = alGenSources();
        if (loop) {
            alSourcei(sourceId, AL_LOOPING, AL_TRUE);
        }
        if (relative) {
            alSourcei(sourceId, AL_SOURCE_RELATIVE, AL_TRUE);
        }
    }

    public void setBuffer(int bufferId) {
        stop();
        alSourcei(sourceId, AL_BUFFER, bufferId);
    }

    public void setPosition(Vector3f position) {
        alSource3f(sourceId, AL_POSITION, position.x, position.y, position.z);
    }

    public void setSpeed(Vector3f speed) {
        alSource3f(sourceId, AL_VELOCITY, speed.x, speed.y, speed.z);
    }

    public void setGain(float gain) {
        alSourcef(sourceId, AL_GAIN, gain);
    }

    public void setProperty(int param, float value) {
        alSourcef(sourceId, param, value);
    }

    public void play() {
        alSourcePlay(sourceId);
    }

    public boolean isPlaying() {
        return alGetSourcei(sourceId, AL_SOURCE_STATE) == AL_PLAYING;
    }

    public void pause() {
        alSourcePause(sourceId);
    }
```

```
    public void stop() {
        alSourceStop(sourceId);
    }

    public void cleanup() {
        stop();
        alDeleteSources(sourceId);
    }
}
```

The sound source class provides some methods to setup its position, the gain, and control methods for playing stopping and pausing it. Keep in mind that sound control actions are made over a source (not over the buffer), remember that several sources can share the same buffer. As in the `SoundBuffer` class, a `SoundSource` is identified by an identifier, which is used in each operation. This class also provides a `cleanup` method to free the reserved resources. But let's examine the constructor. The first thing that we do is to create the source with the `alGenSources` call. Then, we set up some interesting properties using the constructor parameters.

The first parameter, `loop`, indicates if the sound to be played should be in loop mode or not. By default, when a play action is invoked over a source the playing stops when the audio data is consumed. This is fine for some sounds, but some others, like background music, need to be played over and over again. Instead of manually controlling when the audio has stopped and re-launch the play process, we just simply set the looping property to true: "`alSourcei(sourceId, AL_LOOPING, AL_TRUE);`".

The other parameter, `relative`, controls if the position of the source is relative to the listener or not. In this case, when we set the position for a source, we basically are defining the distance (with a vector) to the listener, not the position in the OpenAL 3D scene, not the world position. This activated by the "`alSourcei(sourceId, AL_SOURCE_RELATIVE, AL_TRUE);`" call. But, What can we use this for? This property is interesting for instance for background sounds that should be affected (attenuated) by the distance to the listener. Think, for instance, in background music or sound effects related to player controls. If we set these sources as relative, and set their position to $(0, 0, 0)$ they will not be attenuated.

Now it's turn for the listener which, surprise, is modelled by a class named `SoundListener`. Here's the definition for that class.

www.dbooks.org

```
package org.lwjglb.engine.sound;

import org.joml.Vector3f;

import static org.lwjgl.openal.AL10.*;

public class SoundListener {

    public SoundListener() {
        this(new Vector3f(0, 0, 0));
    }

    public SoundListener(Vector3f position) {
        alListener3f(AL_POSITION, position.x, position.y, position.z);
        alListener3f(AL_VELOCITY, 0, 0, 0);

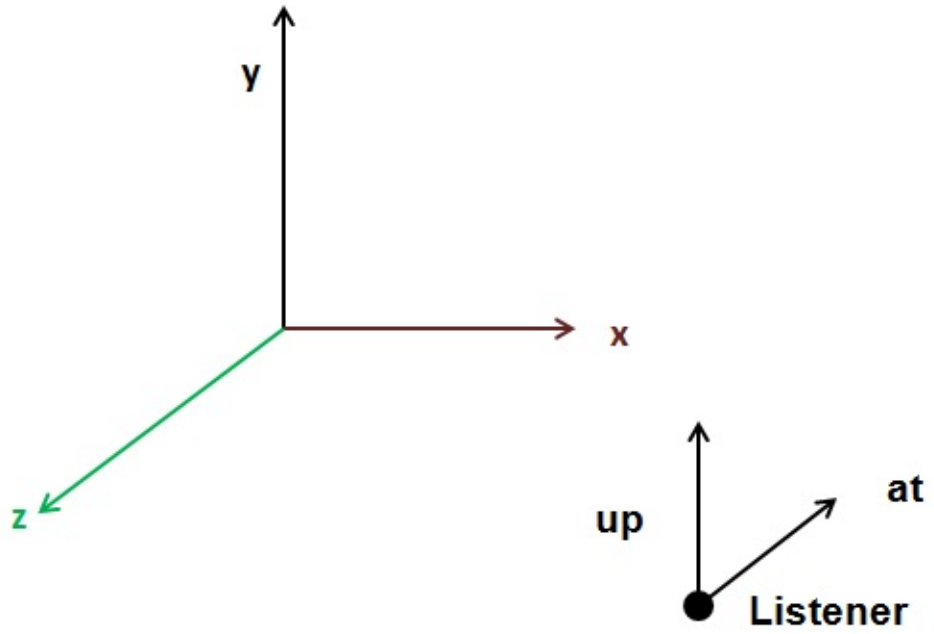
    }

    public void setSpeed(Vector3f speed) {
        alListener3f(AL_VELOCITY, speed.x, speed.y, speed.z);
    }

    public void setPosition(Vector3f position) {
        alListener3f(AL_POSITION, position.x, position.y, position.z);
    }

    public void setOrientation(Vector3f at, Vector3f up) {
        float[] data = new float[6];
        data[0] = at.x;
        data[1] = at.y;
        data[2] = at.z;
        data[3] = up.x;
        data[4] = up.y;
        data[5] = up.z;
        alListenerfv(AL_ORIENTATION, data);
    }
}
```

A difference you will notice from the previous classes is that there's no need to create a listener. There will always be one listener, so no need to create one, it's already there for us. Thus, in the constructor we just simply set its initial position. For the same reason there's no need for a `cleanup` method. The class has methods also for setting listener position and velocity, as in the `SoundSource` class, but we have an extra method for changing the listener orientation. Let's review what orientation is all about. Listener orientation is defined by two vectors, "at" vector and "up" one, which are shown in the next figure.



The “at” vector basically points where the listener is facing, by default its coordinates are $(0, 0, -1)$. The “up” vector determines which direction is up for the listener and, by default it points to $(0, 1, 0)$. So the tree components of each of those two vectors are what are set in the `alListenerfv` method call. This method is used to transfer a set of floats (a variable number of floats) to a property, in this case, the orientation.

Before continuing it's necessary to stress out some concepts in relation to source and listener speeds. The relative speed between sources and listener will cause OpenAL to simulate Doppler effect. In case you don’t know, Doppler effect is what causes that a moving object that is getting closer to you seems to emit in a higher frequency than it seems to emit when is walking away. The thing, is that, simply by setting a source or listener velocity, OpenAL will not update their position for you. It will use the relative velocity to calculate the Doppler effect, but the positions won’t be modified. So, if you want to simulate a moving source or listener you must take care of updating their positions in the game loop.

Now that we have modelled the key elements we can set them up to work, we need to initialize OpenAL library, so we will create a new class named `SoundManager` that will handle this. Here’s a fragment of the definition of this class.

www.dbooks.org

```
package org.lwjglb.engine.sound;

// Imports here

public class SoundManager {

    private long device;

    private long context;

    private SoundListener listener;

    private final List<SoundBuffer> soundBufferList;

    private final Map<String, SoundSource> soundSourceMap;

    private final Matrix4f cameraMatrix;

    public SoundManager() {
        soundBufferList = new ArrayList<>();
        soundSourceMap = new HashMap<>();
        cameraMatrix = new Matrix4f();
    }

    public void init() throws Exception {
        this.device = alcOpenDevice((ByteBuffer) null);
        if (device == NULL) {
            throw new IllegalStateException("Failed to open the default OpenAL device."
);
        }
        ALCCapabilities deviceCaps = ALC.createCapabilities(device);
        this.context = alcCreateContext(device, (IntBuffer) null);
        if (context == NULL) {
            throw new IllegalStateException("Failed to create OpenAL context.");
        }
        alcMakeContextCurrent(context);
        AL.createCapabilities(deviceCaps);
    }
```

This class holds references to the `SoundBuffer` and `SoundSource` instances to track and later cleanup them properly. SoundBuffers are stored in a List but SoundSources are stored in in a `Map` so they can be retrieved by a name. The `init` method initializes the OpenAL subsystem:

- Opens the default device.
- Create the capabilities for that device.
- Create a sound context, like the OpenGL one, and set it as the current one.

The `SoundManager` class also has a method to update the listener orientation given a camera position. In our case, the listener will be placed whenever the camera is. So, given camera position and rotation information, how do we calculate the “at” and “up” vectors? The answer is by using the view matrix associated to the camera. We need to transform the “at” $(0, 0, -1)$ and “up” $(0, 1, 0)$ vectors taking into consideration camera rotation. Let `cameraMatrix` be the view matrix associated to the camera. The code to accomplish that would be:

```
Matrix4f invCam = new Matrix4f(cameraMatrix).invert();
Vector3f at = new Vector3f(0, 0, -1);
invCam.transformDirection(at);
Vector3f up = new Vector3f(0, 1, 0);
invCam.transformDirection(up);
```

The first thing that we do is invert the camera view matrix. Why we do this? Think about it this way, the view matrix transforms from world space coordinates to view space. What we want is just the opposite, we want to transform from view space coordinates (the view matrix) to space coordinates, which is where the listener should be positioned. With matrices, the opposite usually means the inverse. Once we have that matrix we just transform the “default” “at” and “up” vectors using that matrix to calculate the new directions.

But, if you check the source code you will see that the implementation is slightly different, what we do is this:

```
Vector3f at = new Vector3f();
cameraMatrix.positiveZ(at).negate();
Vector3f up = new Vector3f();
cameraMatrix.positiveY(up);
listener.setOrientation(at, up);
```

The code above is equivalent to the first approach, it’s just a more efficient approach. It uses a faster method, available in JOML library, that just does not need to calculate the full inverse matrix but achieves the same results. This method was provided by the JOML author in a LWJGL forum, so you can check more details there. If you check the source code you will see that the `SoundManager` class calculates its own copy of the view matrix. This is already done in the `Renderer` class. In order to keep the code simple, and to avoid refactoring, I’ve preferred to keep this that way.

And that’s all. We have all the infrastructure we need in order to play sounds. You can check in the source code how all the pieces are used. You can see how music is played and the different effects sound (These files were obtained from Freesound, proper credits are in a

www.dbooks.org

file name CREDITS.txt). If you get some other files, you may notice that sound attenuation with distance or listener orientation will not work. Please check that the files are in mono, not in stereo. OpenAL can only perform those computations with mono sounds.

A final note. OpenAL also allows you to change the attenuation model by using the alDistanceModel and passing the model you want (``AL11.AL_EXPONENT_DISTANCE`, AL_EXPONENT_DISTANCE_CLAMP```, etc.). You can play with them and check the results.

3D Object Picking

Camera Selection

One of the key aspects of every game is the ability to interact with the environment. This capability requires to be able to select objects in the 3D scene. In this chapter we will explore how this can be achieved.

But, before we start talking about the steps to be performed to select objects, we need a way to represent selected objects. Thus, the first thing that we must do, is add another attribute to the GameItem class, which will allow us to tag selected objects:

```
private boolean selected;
```

Then, we need to be able to use that value in the scene shaders. Let's start with the fragment shader (`scene_fragment.fs`). In this case, we will assume that we will receive a flag, from the vertex shader, that will determine if the fragment to be rendered belongs to a selected object or not.

```
in float outSelected;
```

Then, at the end of the fragment shader, we will modify the final fragment colour, by setting the blue component to 1 if it's selected.

```
if ( outSelected > 0 ) {
    fragColor = vec4(fragColor.x, fragColor.y, 1, 1);
}
```

Then, we need to be able to set that value for each . If you recall from previous chapters we have two scenarios:

- Rendering of non instanced meshes.
- Rendering of instanced meshes.

In the first case, the data for each `GameItem` is passed through uniforms, so we just need to add a new uniform for that in the vertex shader. In the second case, we need to create a new instanced attribute. You can see bellow the additions that need to be integrated into the vertex shader for both cases.

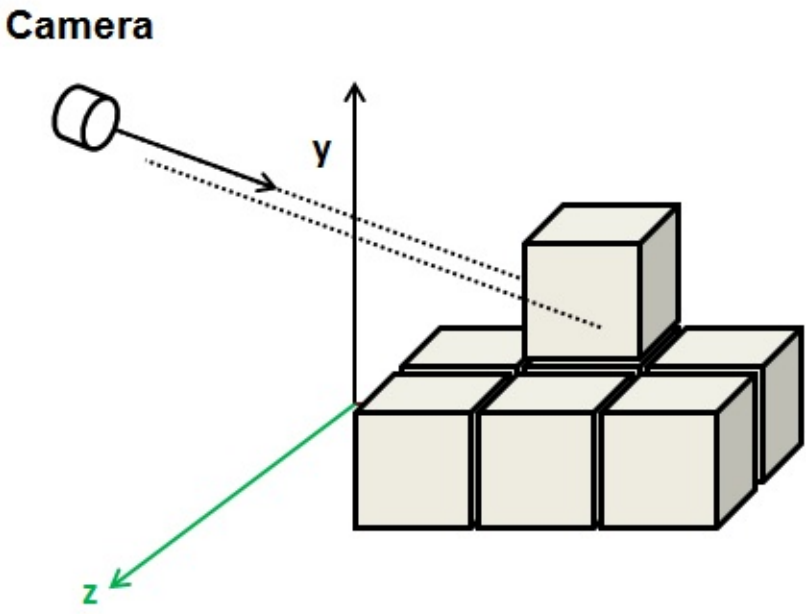
www.dbooks.org

```
layout (location=14) in float selectedInstanced;
...
uniform float selectedNonInstanced;
...
    if ( isInstanced > 0 )
    {
        outSelected = selectedInstanced;
...
    }
    else
    {
    outSelected = selectedNonInstanced;
...
```

Now that the infrastructure has been set-up we just need to define how objects will be selected. Before we continue you may notice, if you look at the source code, that the View matrix is now stored in the `Camera` class. This is due to the fact that we werrecalculating the view matrix sin several classes in the source code. Previously, it was stored in the `Transformation` and in the `SoundManager` classes. In order to calculate intersections we would need to cerate another replica. Instead of that, we centralize that in the `Camera` class. This change also, requires that the view matrix is updated in our main game loop.

Let's continue with the picking discussion. In this sample, we will follow a simple approach, selection will be done automatically using the camera. The closest object to where the camera is facing will be selected. Let's discuss how this can be done.

The following picture depicts the situation we need to solve.



We have the camera, placed in some coordinates in world-space, facing a specific direction. Any object that intersects with a ray cast from camera's position following camera's forward direction will be a candidate. Between all the candidates we just need to chose the closest one.

In our sample, game items are cubes, so we need to calculate the intersection of the camera's forward vector with cubes. It may seem to be a very specific case, but indeed is very frequent. In many games, the game items have associated what's called a bounding box. A bounding box is a rectangle box, that contains all the vertices for that object. This bounding box is used also, for instance, for collision detection. In fact, in the animation chapter, you saw that each animation frame defined a bounding box, that helps to set the boundaries at any given time.

So, let's start coding. We will create a new class named ``` CameraBoxSelectionDetector ```, which will have a method named selectGameItem``` which will receive a list of game items and a reference to the camera. The method is defined like this.

```
public void selectGameItem(GameItem[] gameItems, Camera camera) {
    GameItem selectedGameItem = null;
    float closestDistance = Float.POSITIVE_INFINITY;

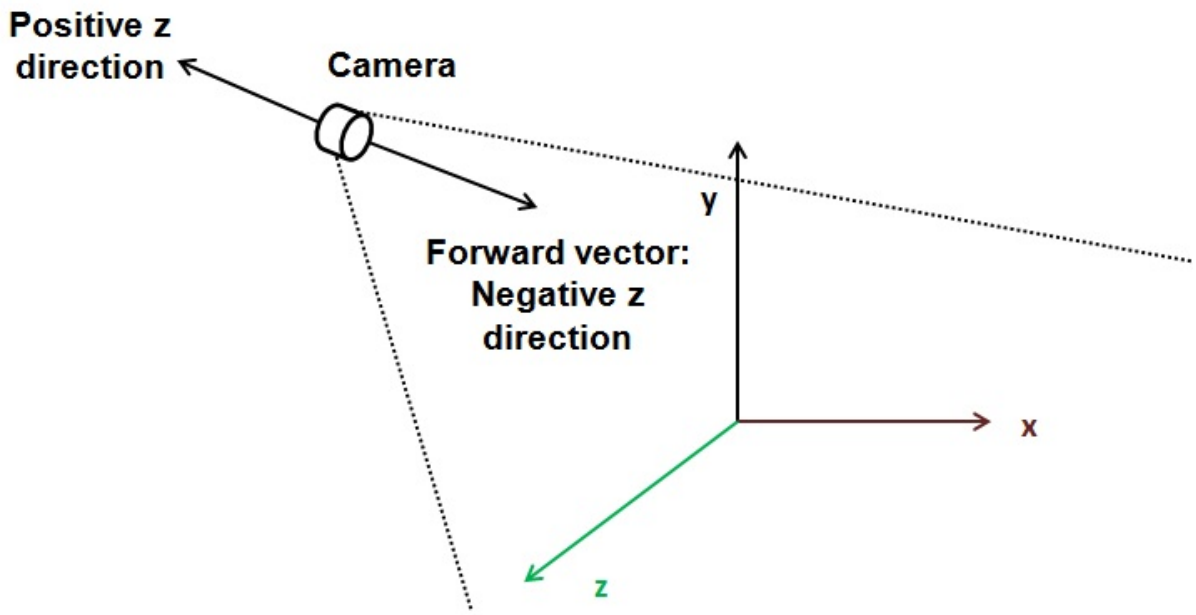
    dir = camera.getViewMatrix().positiveZ(dir).negate();
    for (GameItem gameItem : gameItems) {
        gameItem.setSelected(false);
        min.set(gameItem.getPosition());
        max.set(gameItem.getPosition());
        min.add(-gameItem.getScale(), -gameItem.getScale(), -gameItem.getScale());
        max.add(gameItem.getScale(), gameItem.getScale(), gameItem.getScale());
        if (Intersectionf.intersectRayAab(camera.getPosition(), dir, min, max, nearFar
) && nearFar.x < closestDistance) {
            closestDistance = nearFar.x;
            selectedGameItem = gameItem;
        }
    }

    if (selectedGameItem != null) {
        selectedGameItem.setSelected(true);
    }
}
```

The method, iterates over the game items trying to get the ones that interesect with the ray cast form the camera. It first defines a variable named `closestDistance`. This variable will hold the closest distance. For game items that intersect, the distance from the camera to the intersection point will be calculated, If it's lower than the value stored in `closestDistance`, then this item will be the new candidate.

www.dbooks.org

Before entering into the loop, we need to get the direction vector that points where the camera is facing. This is easy, just use the view matrix to get the z direction taking into consideration camera's rotation. Remember that positive z points out of the screen, so we need the opposite direction vector, this is why we negate it.



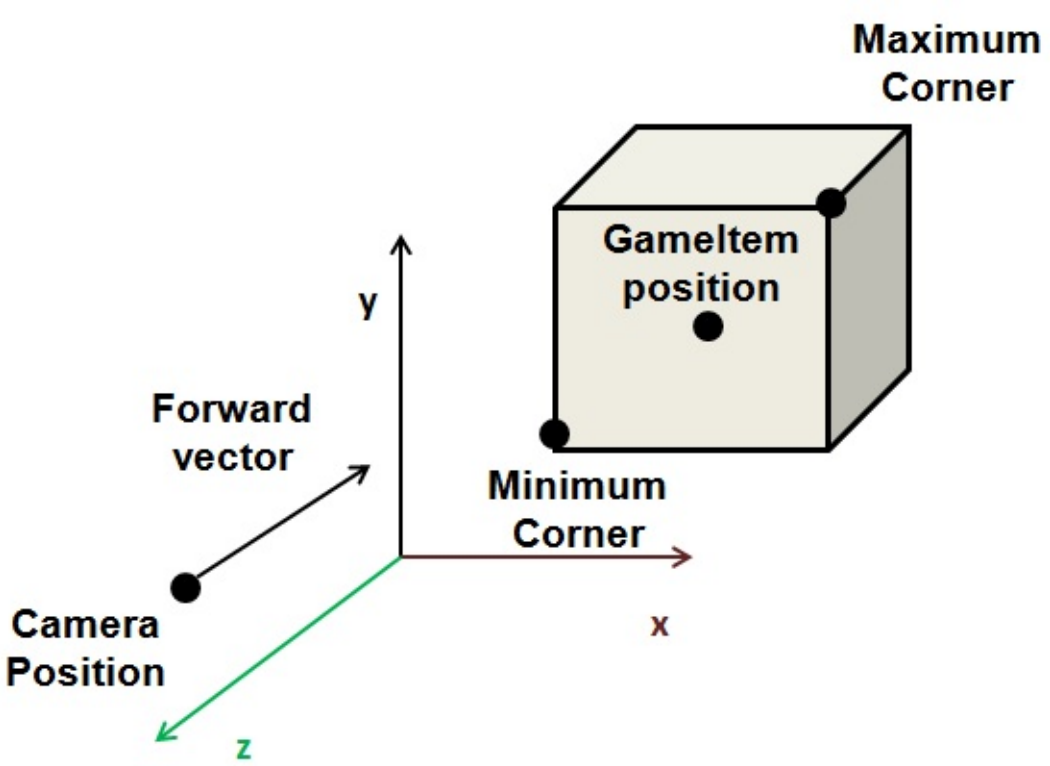
In the game loop intersection calculations are done per each `GameItem`. But, how do we do this? This is where the glorious JOML library comes to the rescue. We are using JOML's `Intersectionf` class, which provides several methods to calculate intersections in 2D and 3D. Specifically, we are using the `intersectRayAab` method.

This method implements the algorithm that test intersection for Axis Aligned Boxes. You can check the details, as pointed out in the JOML documentation, here.

The method tests if a ray, defined by an origin and a direction, intersects a box, defined by minimum and maximum corner. This algorithm is valid, because our cubes, are aligned with the axis, if they were rotated, this method would not work. Thus, the method receives the following parameters:

- An origin: In our case, this will be our camera position.
- A direction: This is where the camera is facing, the forward vector.
- The minimum corner of the box. In our case, the cubes are centered around the GameItem position, the minimum corner will be those coordinates minus the scale. (In its original size, cubes have a length of 2 and a sacle of 1).
- The maximum corner. Self explanatory.
- A result vector. This will contain the near and far distances of the intersection points.

The method will return true if there is an intersection. If true, we check the closes distance and update it if needed, and store a reference of the candidate selected `GameItem`. The next figure shows all the elements involved in this method.



Once the loop has finished, the candidate `GameItem` is marked as selected.

And that's, all. The `selectGameItem` will be invoked in the update method of the DummyGame class, along with the view matrix update.

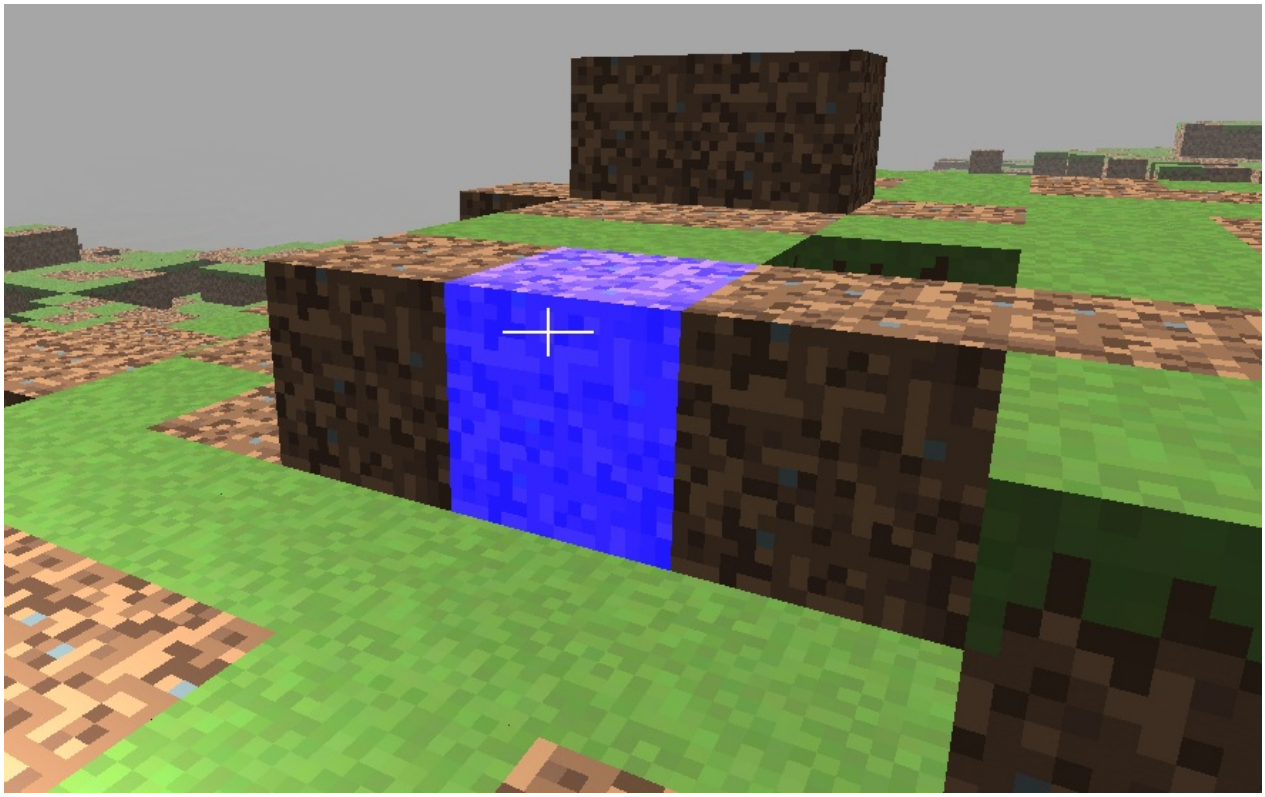
```
// Update view matrix
camera.updateViewMatrix();

// Update sound listener position;
soundMgr.updateListenerPosition(camera);

this.selectDetector.selectGameItem(gameItems, camera);
```

Besides that, a cross-hair has been added to the rendering process to check that everything is working properly. The result is shown in the next figure.

www.dbooks.org



Obviously, the method presented here is far from optimal but it will give you the basics to develop more sophisticated methods on your own. Some parts of the scene could be easily discarded, like objects behind the camera, since they are not going to be intersected. Besides that, you may want to order your items according to the distance to the camera to speed up calculations. In addition to that, calculations only need to be done if the camera has moved or. rotated from previous update.

Mouse Selection

Object picking with the camera is great, but what if we want to be able to freely select objects with the mouse? In this case, we want that, whenever the user clicks on the screen, the closest object is automatically selected.

The way to achieve this is similar to the method described above. In the previous method we had the camera position and generated rays from it using the “forward” direction according to camera’s current orientation. In this case, we still need to cast rays, but the direction points to a point far away from the camera, where the click has been made. In this case, we need to calculate that direction vector using the click coordinates.

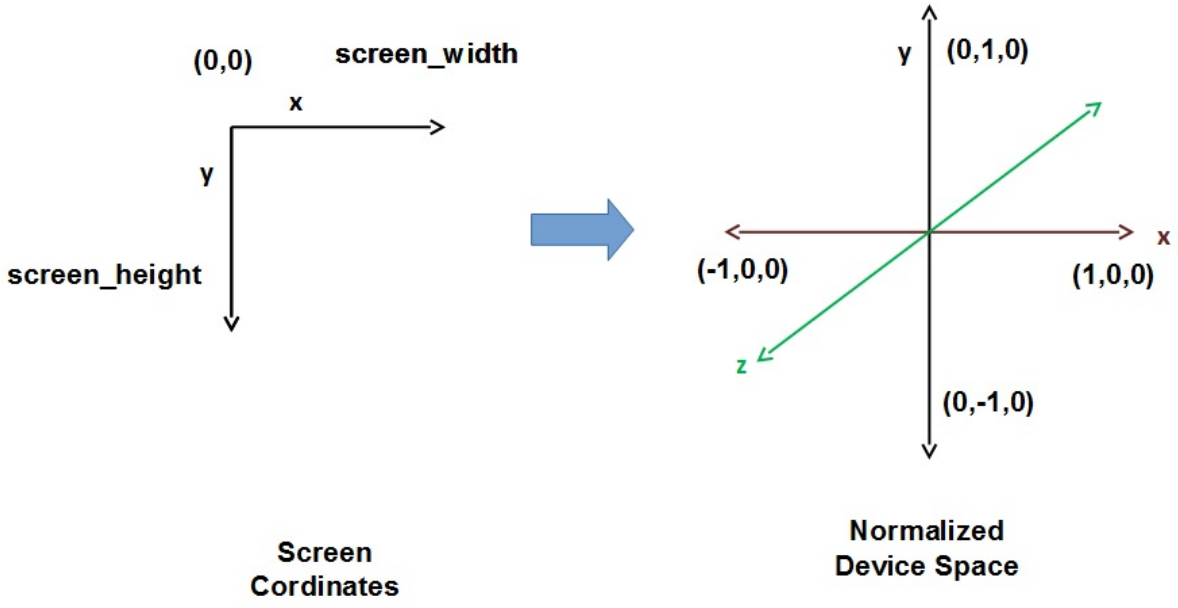
But, how do we pass from a (x, y) coordinates in viewport space to world space? Let’s review how we pass from model space coordinates to view space. The different coordinate transformations that are applied in order to achieve that are:

- We pass from model coordinates to world coordinates using the model matrix.

- We pass from world coordinates to view space coordinates using the view matrix (that provides the camera effect)-
- We pass from view coordinates to homogeneous clip space by applying the perspective projection matrix.
- Final screen coordinates are calculate automatically by OpenGL for us. Before doint that, it passes to normalized device space (by dividing the x, y, z coordinates by the w component) and then to x, y screen coordinates.

So we need just to perform the traverse the inevrse path to get from screen coordinates (x, y), to world coordinates.

The first step is to transform from screen coordinates to normalized device space. The (x, y) coordinates in the view port space are in the range $[0, screenwith]$ $[0, screenheight]$. The upper left corner of the screen has a value of $[0, 0]$. We need to transform that into coordinates in the range $[-1, 1]$.



The maths are simple:

$$x = 2 \cdot screen_x / screenwidth - 1$$

$$y = 1 - 2 * screen_y / screenheight$$

But, how do we calculate the z coordinate? The answer is simple, we simply assign it the -1 value, so that the ray points to the farthest visible distance (Remember that in OpenGL, -1 points to the screen). Now we have the coordinates in normalised device space.

In order to continue with the transformations we need to convert them to the homogeneous clip space. We need to have the w component, that is use homogeneous coordinates. Although this concept was presented in the previous chapters, let's get back to it. In order to

www.dbooks.org

represent a 3D point we just need the x, y and z coordinates, but we are continuously working with an additional coordinate, the w component. We need this extra component in order to use matrices to perform the different transformations. Some transformations do not need that extra component but other do. For instance, the translation matrix does not work if we only have x, y and z components. Thus, we have added the w component and assigned them a value of 1 so we can work with 4 by 4 matrices.

Besides that, most of transformations, or to be more precise, most of the transformation matrices do not alter the w component. An exception to this is the projection matrix. This matrix changes the w value to be proportional to the z component.

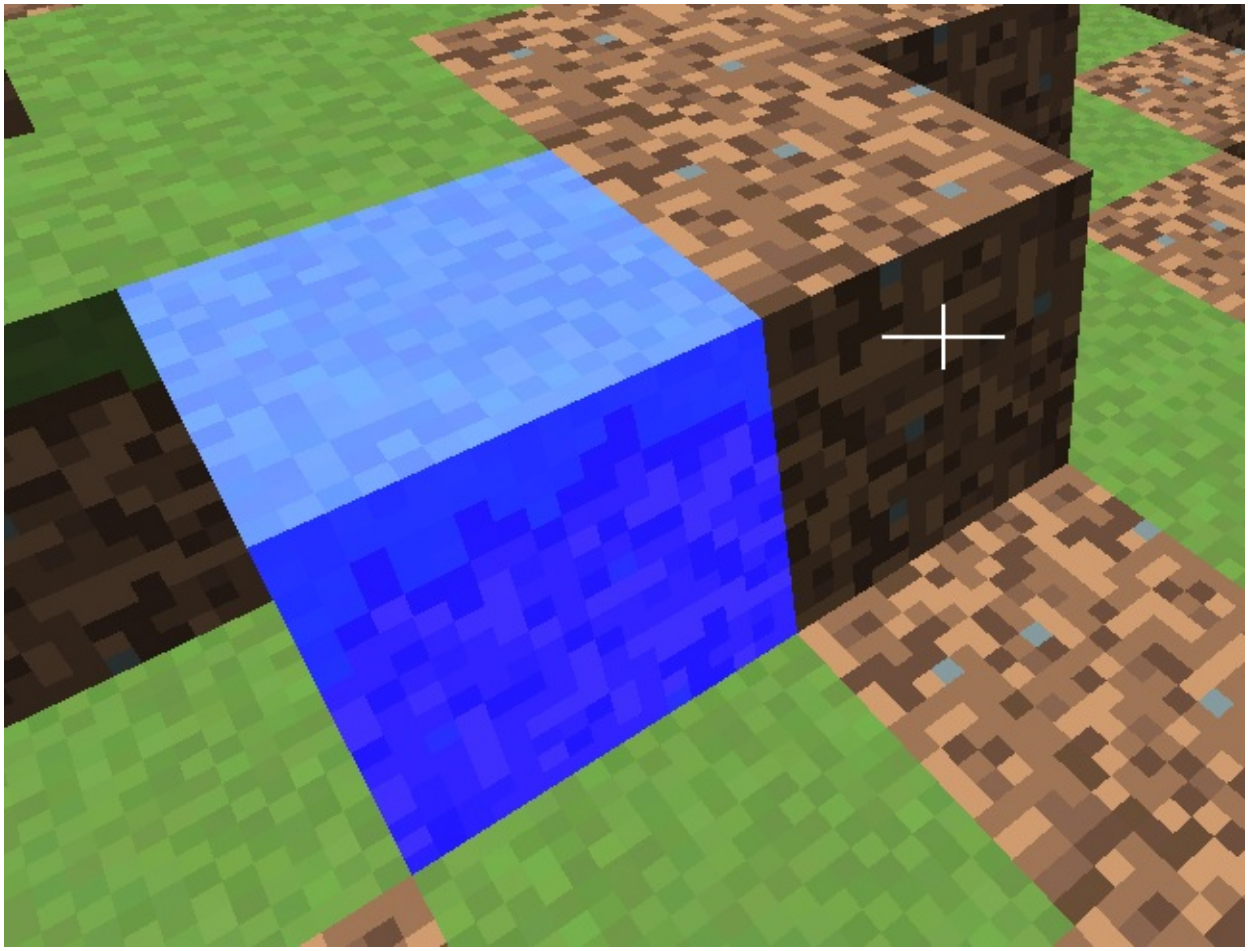
Transforming from homogeneous clip space to normalized device coordinates is achieved by dividing the x, y and z coordinates by w. As this component is proportional to the z component, this implies that distant objects are drawn smaller. In our case we need to do the reverse, we need to unproject, but since what we are calculating it's a ray we just simply can ignore that step, we just set the w component to 1 and leave the rest of the components with their original value.

Now we need to go back yo view space. This is easy, we just need to calculate the inverse of the projection matrix and multiply it by our 4 components vector. Once we have done that, we need to transform them to world space. Again, we just need to use the view matrix, calculate it's inverse and multiply it by our vector.

Remember that we are only interested in directions, so, in this case we set the w component to 0. Also we can set the z component again to -1, since we want it to point towards the screen. Once we have done that and applied the inverse view matrix we have our vector in world space. We have our ray calculated and can apply the same algorithm as in the case of the camera picking.

We have created a new class named `MouseBoxSelectionDetector` that implements the setps described above. Besides that, we have moved the projection matrix to the `Window` class so we can use them in several places of the source code and refactroed a little bit the `CameraBoxSelectionDetector` so the `MouseBoxSelectionDetector` can inheit from it and use the collision detection method. You can check the source code directly, since the implemenattion it's very simple.

The result now looks like this.



You just need to click over the block with the mouse left button to perform the selection.

In any case, if you can consult the details behind the steps explained here in an excellent article with very detailed sketechs of the different steps involved.

www.dbooks.org

HUD Revisited - NanoVG

In previous chapters we explained how a HUD can be created renderings shapes and textures over the top of the scene using an orthographic projection. In this chapter we will learn how to use the NanoVG library to be able to render antialiased vector graphics to construct more complex HUDs in an easy way.

There are many other libraries out there that you can use to accomplish this task, such as Nifty GUI, Nuklear, etc. In this chapter we will focus on Nanovg since it's very simple to use, but if you're looking for developing complex GUI interactions with buttons, menus and windows you should probably look for Nifty GUI.

The first step in order to start using NanoVG is adding the dependences in the `pom.xml` file (one for the dependencies required at compile time and the other one for the natives required at runtime).

```
...
<dependency>
    <groupId>org.lwjgl</groupId>
    <artifactId>lwjgl-nanovg</artifactId>
    <version>${lwjgl.version}</version>
</dependency>
...
<dependency>
    <groupId>org.lwjgl</groupId>
    <artifactId>lwjgl-nanovg</artifactId>
    <version>${lwjgl.version}</version>
    <classifier>${native.target}</classifier>
    <scope>runtime</scope>
</dependency>
```

Before we start using NanoVG we must set up some things in the OpenGL side so the samples can work correctly. We need to enable support for stencil buffer test. Until now we have talked about colour and depth buffers, but we have not mentioned the stencil buffer. This buffer stores a value (an integer) for every pixel which is used to control which pixels should be drawn. This buffer is used to mask or discard drawing areas according to the values it stores. It can be used, for instance, to cut out some parts of the scene in an easy way. We enable stencil buffer test by adding this line to the `Window` class (after we enable depth testing):

```
glEnable(GL_STENCIL_TEST);
```

Since we are using another buffer we must take care also of removing its values before each render call. Thus, we need to modify the clear method of the `Renderer` class:

```
public void clear() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
}
```

We will also add a new window option for activating antialiasing. Thus, in the Window class we will enable it by this way:

```
if (opts.antialiasing) {
    glfwWindowHint(GLFW_SAMPLES, 4);
}
```

Now we are ready to use the NanoVG library. The firs thing that we will do is get rid off the HUD artefacts we have created, that is the shaders the `IHud` interface, the hud rendering methods in the `Renderer` class, etc. Yo can check this out in the source code.

In this case, the new `Hud` class will take care of its rendering, so we do not need to delegate it to the `Renderer` class. Let's tart by defining that class, It will have an `init` method that sets up the library and the resources needed to build the HUD. The method is defined like this:

```
public void init(Window window) throws Exception {
    this.vg = window.getOptions().antialiasing ? nvgCreate(NVG_ANTIALIAS | NVG_STENCIL
_STROKES) : nvgCreate(NVG_STENCIL_STROKES);
    if (this.vg == NULL) {
        throw new Exception("Could not init nanovg");
    }

    fontBuffer = Utils.ioResourceToByteBuffer("/fonts/OpenSans-Bold.ttf", 150 * 1024);
    int font = nvgCreateFontMem(vg, FONT_NAME, fontBuffer, 0);
    if (font == -1) {
        throw new Exception("Could not add font");
    }
    colour = NVGColor.create();

    posx = MemoryUtil.memAllocDouble(1);
    posy = MemoryUtil.memAllocDouble(1);

    counter = 0;
}
```

The first thing we do is create a NanoVG context. In this case we are using an OpenGL 3.0 backend since we are referring to the `org.lwjgl.nanovg.NanoVGGL3` namespace. If antialiasing is activated we set up the flag `NVG_ANTIALIAS`.

Next, we create a font by using a True Type font previously loaded into a `ByteBuffer`. We assign it a name so we can later on use it while rendering text. One important thing about this is that the `ByteBuffer` used to load the font must be kept in memory while the font is used. That is, it cannot be garbage collected, otherwise you will get a nice core dump. This is why it is stored as a class attribute.

Then, we create a colour instance and some helpful variables that will be used while rendering. That method is called in the game init method, just before the rendered is initialized:

```
@Override
public void init(Window window) throws Exception {
    hud.init(window);
    renderer.init(window);
    ...
```

The `Hud` class also defines a render method, which should be called after the scene has been rendered so the HUD is drawn on top of it.

```
@Override
public void render(Window window) {
    renderer.render(window, camera, scene);
    hud.render(window);
}
```

The `render` method of the Hud class starts like this:

```
public void render(Window window) {
    nvgBeginFrame(vg, window.getWidth(), window.getHeight(), 1);
```

The first thing that we must do is call the `nvgBeginFrame`` method. All the NanoVG rendering operations must be enclosed between a` nvgBeginFrame `and` nvgEndFrame `calls. The` nvgBeginFrame``` accepts the following parameters:

- The NanoVG context.
- The size of the window to render (width an height).
- The pixel ratio. If you need to support Hi-DPI you can change this value. For this sample we just set it to 1.

Then we create several ribbons that occupy the whole screen with. The first one is drawn like this:

```
// Upper ribbon
nvgBeginPath(vg);
nvgRect(vg, 0, window.getHeight() - 100, window.getWidth(), 50);
nvgFillColor(vg, rgba(0x23, 0xa1, 0xf1, 200, colour));
nvgFill(vg);
```

While rendering a shape, the first method that shall be invoked is `nvgBeginPath`, that instructs NanoVG to start drawing a new shape. Then we define what to draw, a rect, the fill colour and by invoking the `nvgFill` we draw it.

You can check the rest of the source code to see how the rest of the shapes are drawn. When rendering text is not necessary to call nvgBeginPath before rendering it.

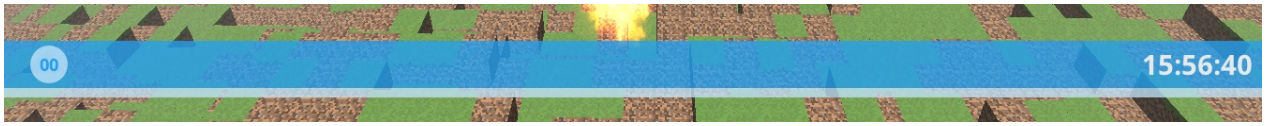
After we have finished drawing all the shapes, we just call the `nvgEndFrame` to end rendering, but there's one important thing to be done before leaving the method. We must restore the OpenGL state. NanoVG modifies OpenGL state in order to perform their operations, if the state is not correctly restored you may see that the scene is not correctly rendered or even that it's been wiped out. Thus, we need to restore the relevant OpenGL status that we need for our rendering. This is delegated in the `Window` class:

```
// Restore state
window.restoreState();
```

The method is defined like this:

```
public void restoreState() {
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_STENCIL_TEST);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    if (opts.cullFace) {
        glEnable(GL_CULL_FACE);
        glCullFace(GL_BACK);
    }
}
```

And that's all (besides some additional methods to clear things up), the code is completed. When you execute the sample you will get something like this:



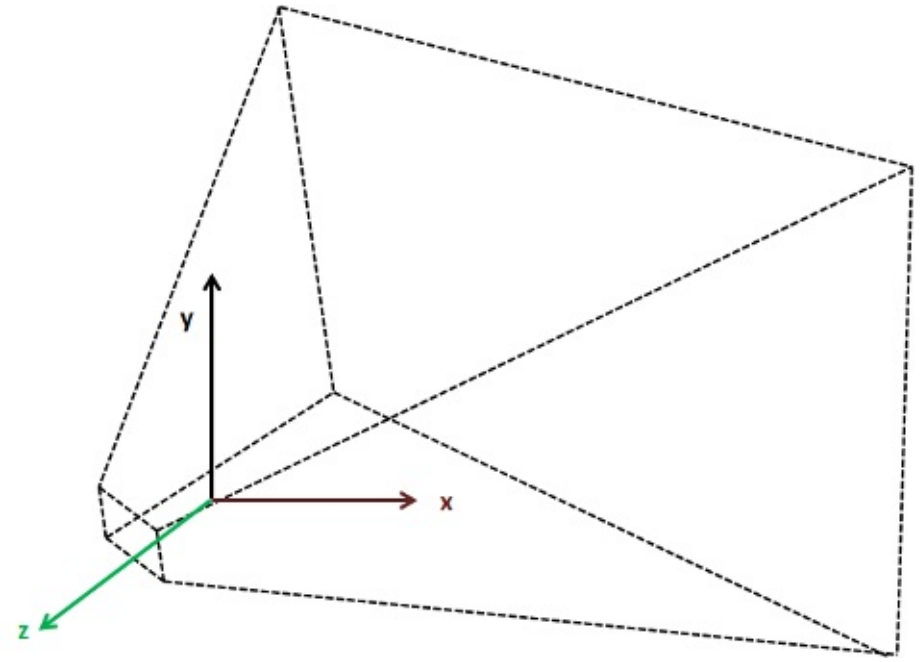
www.dbooks.org

Optimizations - Frustum Culling (I)

At this moment we are using many different graphic effects, such as lights, particles, etc. In addition to that, we have learned how to instanced rendering to reduce the overhead of drawing many similar objects. However, we still have plenty of room for applying simple optimization techniques that will increase the Frames Per Second (FPS) that we can achieve.

You may have wondered why are we drawing the whole list of GameItems every frame even if some of them will not be visible (because they are behind the camera or too far away). You may even think that this is handled automatically handled by OpenGL, and some way you are true. OpenGL will discard the rendering of vertices that fall off the visible area. This called clipping. The issue with clipping is that it's done per vertex, after the vertex shader has been executed. Hence, even this operation saves resources, we can be more efficient by not trying to render objects that will not be visible. We would not be wasting resources by sending the data to the GPU and by performing transformations for every vertex that is part of those objects. We need to remove the objects that do are not contained into the view frustum, that is, we need to perform frustum culling.

But, first let's review what is the view frustum. The view frustum is a volume that contains every object that may be visible taking into consideration the camera position and rotation and the projection that we are using. Typically, the view frustum is a rectangular pyramid like shown in the next figure.

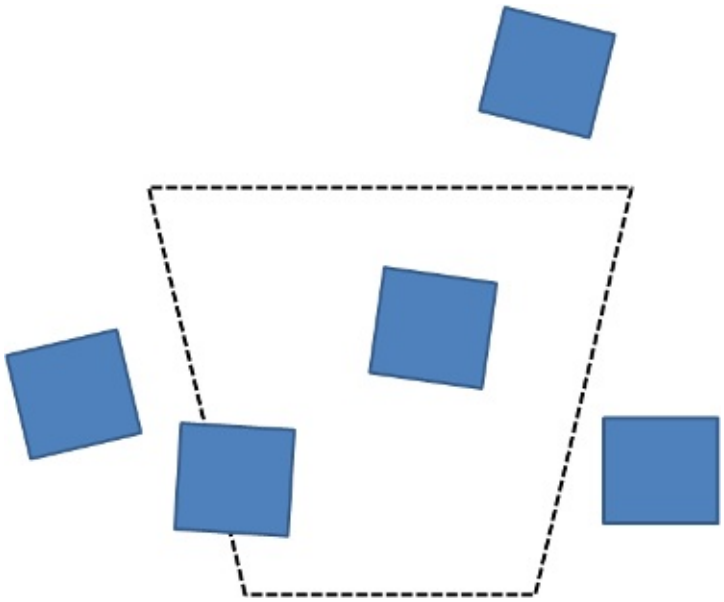


www.dbooks.org

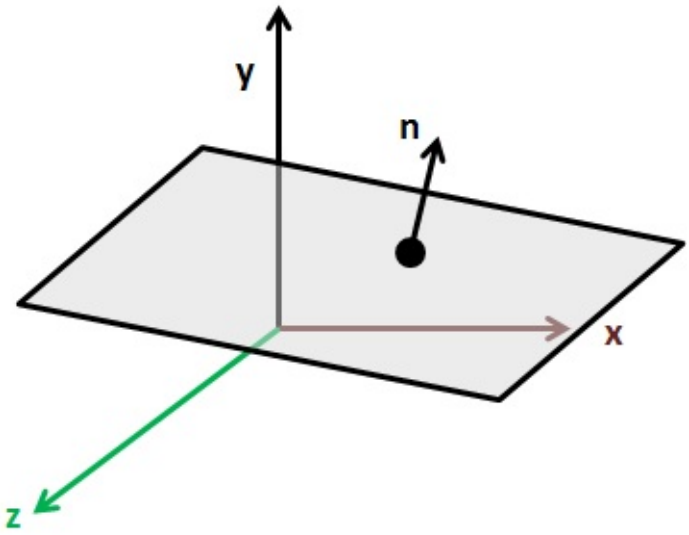
As you can see, the view frustum is defined by six planes, anything that lies outside the view frustum will not be rendering. So, frustum culling is the process of removing objects that are outside the view frustum.

Thus, in order to perform frustum culling we need to:

- Calculate frustum planes using the data contained in the view and projection matrices.
- For every GameItem, check if its contained inside that view frustum, that is, comatined between the size frustum planes, and eliminate the ones that lie outside from the rendering process.



So let's start by calculating the frustum planes. A plane, is defined by a point contained in it and a vector orthogonal to that plane, as shown in the next figure:



The equation of a plane is defined like this:

$$Ax + By + Cz + D = 0$$

Hence, we need to calculate the six plane equations for the six sides of our view frustum. In order to to that you basically have two options. You can perform tedious calculations that will get you the six plane equations, that this, the four constants (A, B, C and D) from the previous equation. The other option is to let JOML library to calculate this for you. In this case, we will chose the last option.

So let's start coding. We will create a new class named `FrustumCullingFilter` which will perform, as its name states, filtering operations according to the view frustum.

```
public class FrustumCullingFilter {

    private static final int NUM_PLANES = 6;

    private final Matrix4f prjViewMatrix;

    private final Vector4f[] frustumPlanes;

    public FrustumCullingFilter() {
        prjViewMatrix = new Matrix4f();
        frustumPlanes = new Vector4f[NUM_PLANES];
        for (int i = 0; i < NUM_PLANES; i++) {
            frustumPlanes[i] = new Vector4f();
        }
    }

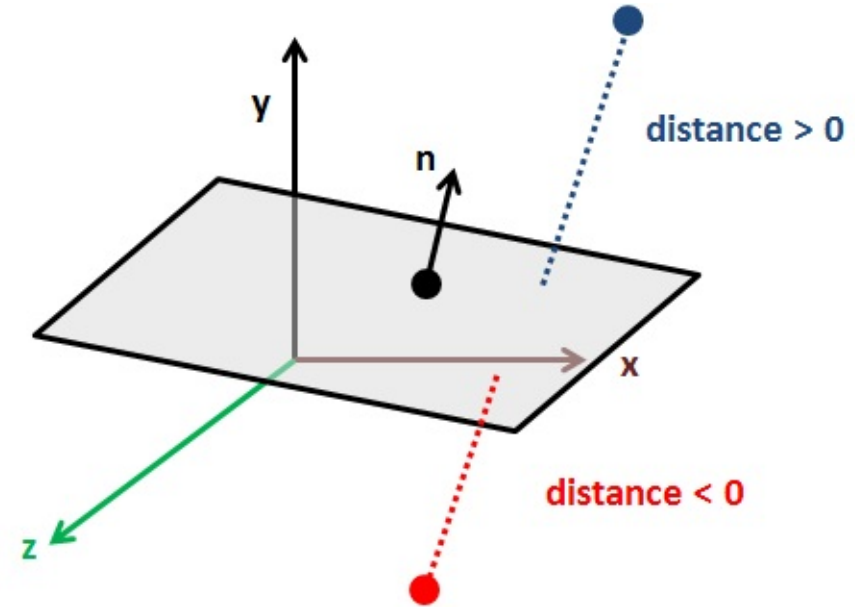
    public void updateFrustum(Matrix4f projMatrix, Matrix4f viewMatrix) {
        // Calculate projection view matrix
        prjViewMatrix.set(projMatrix);
        prjViewMatrix.mul(viewMatrix);
        // Get frustum planes
        for (int i = 0; i < NUM_PLANES; i++) {
            prjViewMatrix.frustumPlane(i, frustumPlanes[i]);
        }
    }
```

The `FrustumCullingFilter` class will also have a method to calculate the plane equations called `updateFrustum` which will be called before rendering. The method is defined like this:

```
public void updateFrustum(Matrix4f projMatrix, Matrix4f viewMatrix) {
    // Calculate projection view matrix
    prjViewMatrix.set(projMatrix);
    prjViewMatrix.mul(viewMatrix);
    // Get frustum planes
    for (int i = 0; i < NUM_PLANES; i++) {
        prjViewMatrix.frustumPlane(i, frustumPlanes[i]);
    }
}
```

First, we store a copy of the projection matrix and multiply it by the view matrix to get the projection view matrix. Then, with that transformation matrix we just simply need to invoke the `frustumPlane` method for each of the frustum planes. It's important to note that these plane equations are expressed in world coordinates, so all the calculations need to be done in that space.

Now that we have all the planes calculated we just need to check if the `GameItem` instances are inside the frustum or not. How can we do this ? Let's first examine how we can check if a point is inside the frustum. We can achieve that by calculating the signed distance of the point to each of the planes. If the distance of the point to the plane is positive, this means that the point is in front of the plane (according to its normal). If it's negative, this means that the point is behind the plane.



Therefore, a point will be inside the view frustum if the distance to all the planes of the frustum is positive. The distance of a point to the plane is defined like this:

$dist = Ax_0 + By_0 + Cz_0 + D$, where x_0, y_0 and z_0 are the coordinates of the point.

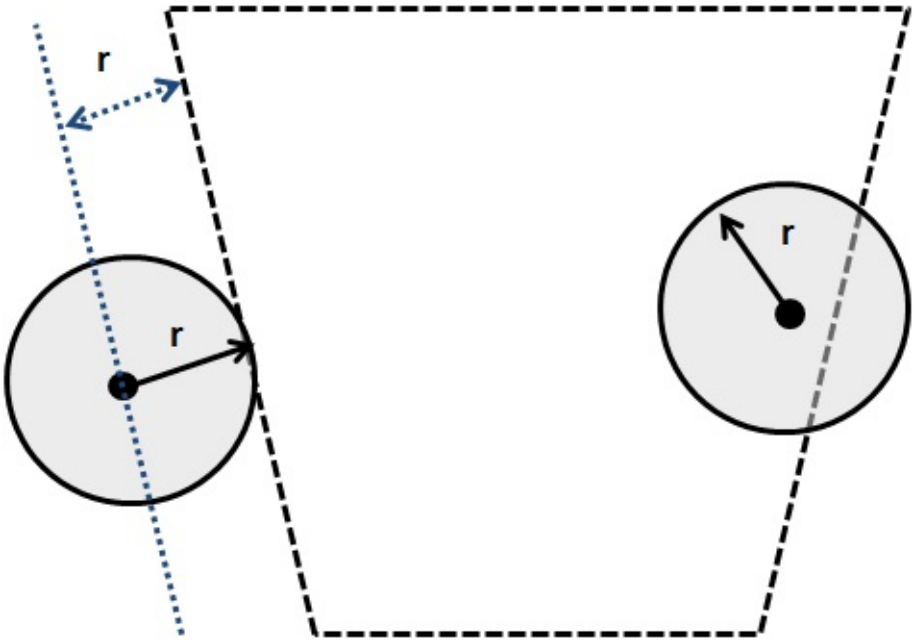
So, a point is behind the plane if $Ax_0 + By_0 + Cz_0 + D <= 0$.

But, we do not have points, we have complex meshes, we cannot just use a point to check if an object is inside a frustum or not. You may think in checking every vertex of the `GameItem` and see if it's inside the frustum or not. If any of the points is inside, the GameItem should be drawn. But this what OpenGL does in fact when clipping, this is what we are tying to avoid. Remember that frustum culling benefits will be more noticeable the more complex the meshes to be rendered are.

We need to enclsoe evey `GameItem` into a simple volume that is easy to check. Here we have basically two options:

- Bounding boxes.
- Bounding Spheres.

In this case, we will use spheres, since is the most simple approach. We will enclose every `GameItems` into a sphere and will check if the sphere is inside the view frustum or not. In order to do that, we just need the center and the radius of the sphere. The checks are almost equal to the point case, except that we need to take the radius into consideration. A sphere will be outside the frustim if it the following condition is met:
$dist = Ax0 + By0 + Cz0 <= -radius$.



So, we will add a new method to the `FrustumCullingFilter` class to check if a spphere is inside the frustum or not. The method is defined like this.

```
public boolean insideFrustum(float x0, float y0, float z0, float boundingRadius) {
    boolean result = true;
    for (int i = 0; i < NUM_PLANES; i++) {
        Vector4f plane = frustumPlanes[i];
        if (plane.x * x0 + plane.y * y0 + plane.z * z0 + plane.w <= -boundingRadius) {
            result = false; return result;
        }
    }
    return result;
}
```

Then, we will add method that filters the GameItems that outside the view frustum:

```
public void filter(List<GameItem> gameItems, float meshBoundingRadius) {
    float boundingRadius;
    Vector3f pos;
    for (GameItem gameItem : gameItems) {
        boundingRadius = gameItem.getScale() * meshBoundingRadius;
        pos = gameItem.getPosition();
        gameItem.setInsideFrustum(insideFrustum(pos.x, pos.y, pos.z, boundingRadius));
    }
}
```

We have added a new attribute, insideFrustum, to the `GameItem` class, to track the visibility. As you can see, the radius of the bounding sphere is passed as parameter This is due to the fact that the bounding sphere is associated to the `Mesh` , it's not a property of the `GameItem` . But, remember that we must operate in world coordinates, and the radios of the bounding sphere will be in model space. We will transform it to world space by applying the scale that has been set up for the `GameItem` , We are assumig also that the position of the `GameItem` is the centre of the spehere (in world space coordinates).

The last method, is just a utility one, that accepts the map of meshes and filters all the `GameItem` instances contained in it.

```
public void filter(Map<? extends Mesh, List<GameItem>> mapMesh) {
    for (Map.Entry<? extends Mesh, List<GameItem>> entry : mapMesh.entrySet()) {
        List<GameItem> gameItems = entry.getValue();
        filter(gameItems, entry.getKey().getBoundingRadius());
    }
}
```

And that's it. We can use that class inside the rendering process. We just need to update the frustum planes, calculate which GameItems are visible and filter them out when drawing instanced and non instanced meshes.

```
frustumFilter.updateFrustum(window.getProjectionMatrix(), camera.getViewMatrix());
frustumFilter.filter(scene.getGameMeshes());
frustumFilter.filter(scene.getGameInstancedMeshes());
```

You can play with activating and deactivating the filtering and can check the increase and decrease in the FPS that you can achieve. Particles are not considered in the filtering, but its trivial to add it. In any case, for particles, it may be better to just check the position of the emitter instead of checking every particle.

Optimizations - Frustum Culling (II)

Once the basis of frustum culling has been explained, we can get advatange of more refined methods that the JOMLlibrary provides. In particular, it provdies a class named `FrustumIntersection` which extracts the planes of the veiw frustum in a more efficient way as described in this paper. Besides that, this class also provides methods for testing bounding boxes, points and spheres.

So, let's change the `FrustumCullingFilter` class. The attributes and constructor are simplified like this:

```
public class FrustumCullingFilter {

    private final Matrix4f prjViewMatrix;

    private FrustumIntersection frustumInt;

    public FrustumCullingFilter() {
        prjViewMatrix = new Matrix4f();
        frustumInt = new FrustumIntersection();
    }
```

The updateFrustum method just delegates the plane extraction to the `FrustumIntersection` instance.

```
public void updateFrustum(Matrix4f projMatrix, Matrix4f viewMatrix) {
    // Calculate projection view matrix
    prjViewMatrix.set(projMatrix);
    prjViewMatrix.mul(viewMatrix);
    // Update frustum intersection class
    frustumInt.set(prjViewMatrix);
}
```

And the method that `insideFrustum` method is vene more simple:

```
public boolean insideFrustum(float x0, float y0, float z0, float boundingRadius) {
    return frustumInt.testSphere(x0, y0, z0, boundingRadius);
}
```

With this approach you will be able to vene get a few more FPS. Besides that, a global flag has been added to the `Window` class to enable / disbale frustum culling. The `GameItem` class also has a flag for enabling / disabling the filtering, because there may be some items for which frustum culling filtering does not make sense.

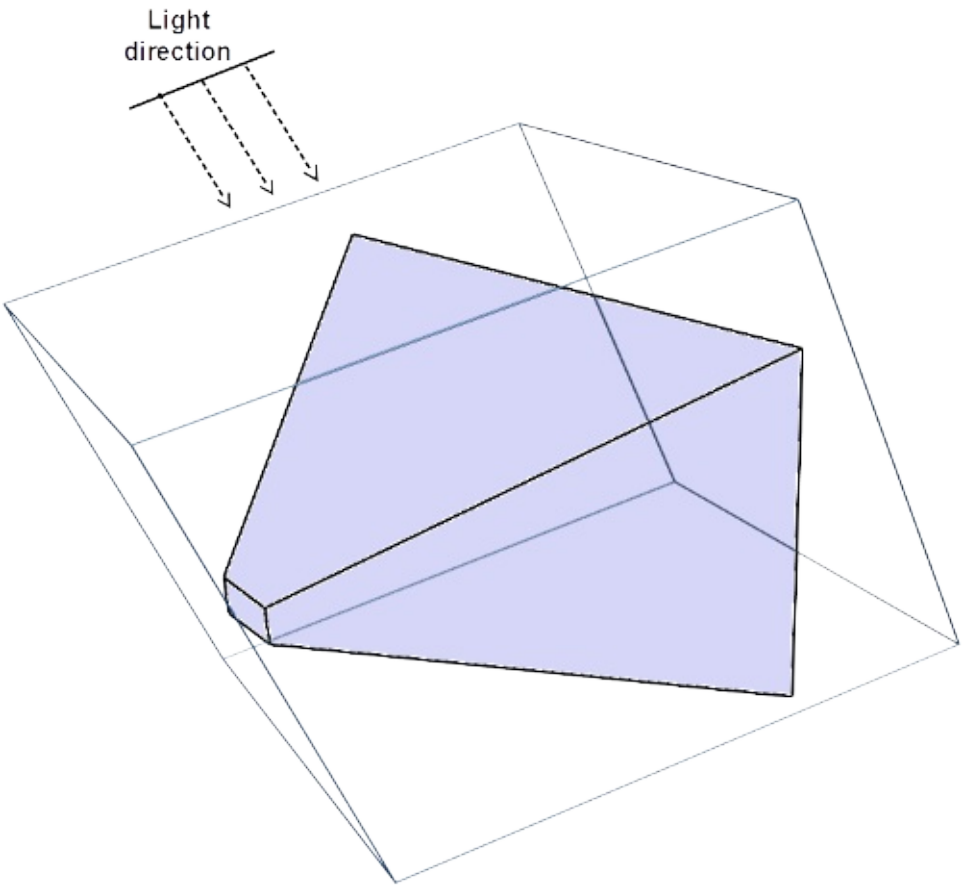
www.dbooks.org

Cascaded Shadow Maps

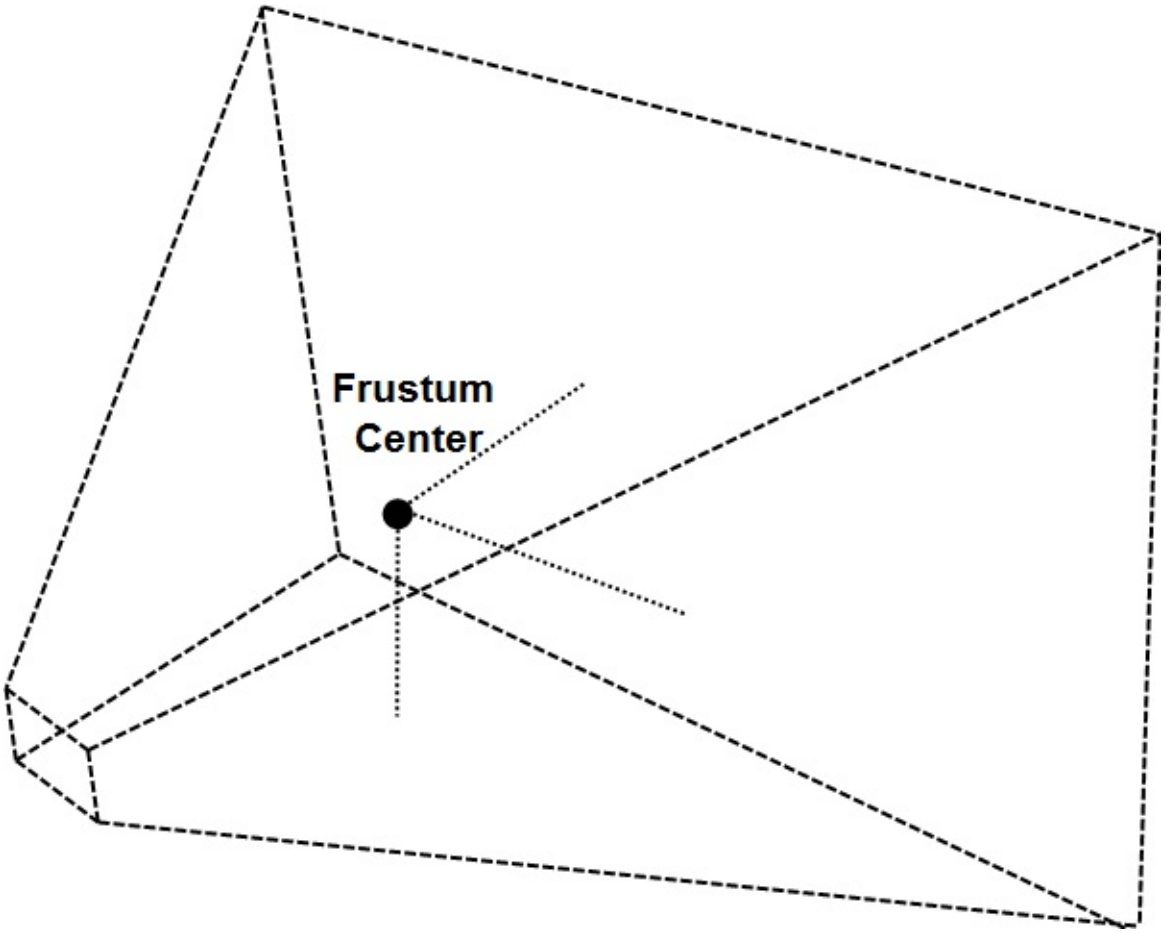
In the shadows chapter we presented the shadow map technique to be able to display shadows using directional lights when rendering a 3D scene. The solution presented there, required you to manually tweak some of the parameters in order to improve the results. In this chapter we are going to change that technique to automate all the process and to improve the r results for open spaces. In order to achieve that goal we are going to use a technique called Cascaded Shadow Maps (CSM).

Let's first start by examining how we can automate the construction of the light view matrix and the orthographic projection matrix used to render the shadows. If you recall from the shadows chapter, we need to draw the scene form the light's perspective. This implies the creation of a light view matrix, which acts like a camera for light and a projection matrix. Since light is directional, and is supposed to be located at the infinity, we chose an orthographic projection.

We want all the visible objects to fit into the light view projection matrix. Hence, we need to fit the view frustum into the light frustum. The following picture depicts what we want to achieve.

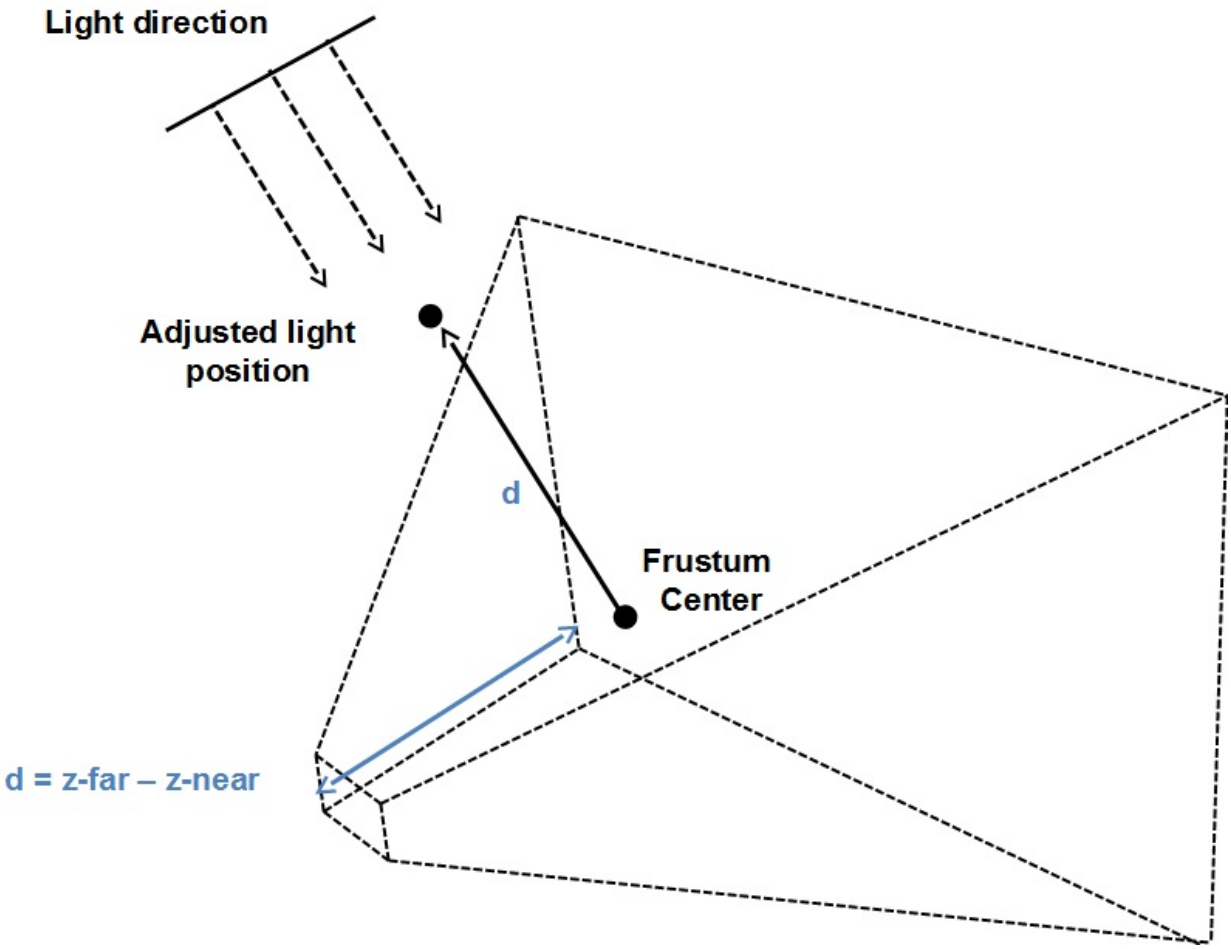


How can we construct that? The first step is to calculate the frustum corners of the view projection matrix. We get the coordinates in world space. Then we calculate the centre of that frustum. This can be calculating by adding the coordinates for all the corners and dividing the result by the number of corners.



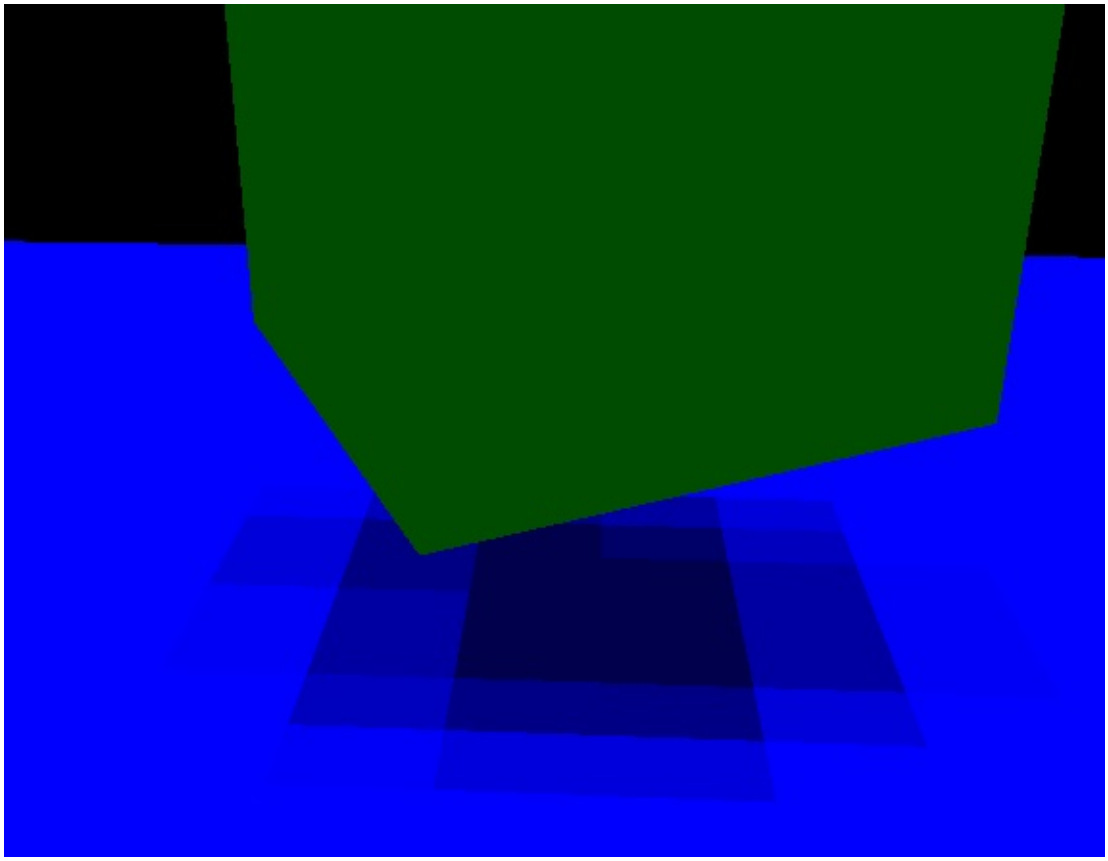
With that information we can set the position of the light. That position and its direction will be used to construct the light view matrix. In order to calculate the position, we start form the centre of the view frustum obtained before. We then go back to the direction of light an amount equal to the distance between the near and far z planes of the view frustum.

www.dbooks.org



Once we have constructed the light view matrix, we need to setup the orthographic projection matrix. In order to calculate them we transform the frustum corners to light view space, just by multiplying them by the light view matrix we have just constructed. The dimensions of that projection matrix will be minimum and maximum x and y values. The near z plane can be set up to the same value used by our standard projection matrices and the far value will be the distance between the maximum and minimum z values of the frustum corners in light view space.

However, if you implement the algorithm described above over the shadows sample, you may be disappointed by the shadows quality.

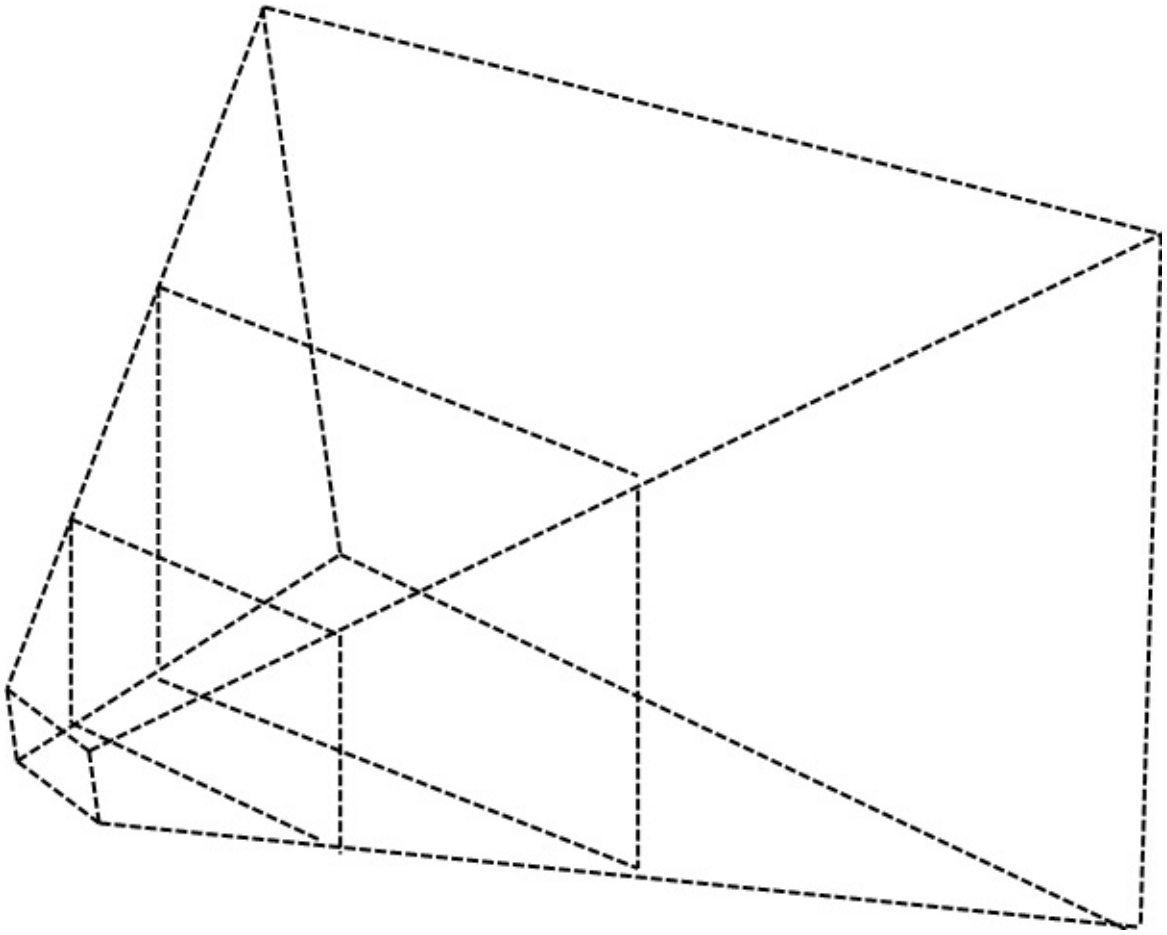


The reason for that is that shadows resolution is limited by the texture size. We are covering now a potentially huge area, and textures we are using to store depth information have not enough resolution in order to get good results. You may think that the solution is just to increase texture resolution, but this is not sufficient to completely fix the problem. You would need huge textures for that.

There's a smarter solution for that. The key concept is that, shadows of objects that are closer to the camera need to have a higher quality than shadows for distant objects. One approach could be to just render shadows for objects close to the camera, but this would cause shadows to appear / disappear as long as we move through the scene.

The approach that Cascaded Shadow Maps (CSMs) use is to divide the view frustum into several splits. Splits closer to the camera cover a smaller amount spaces whilst distant regions cover a much wider region of space. The next figure shows a view frustum divided into three splits.

www.dbooks.org



For each of these splits, the depth map is rendered, adjusting the light view and projection matrices to cover fit to each split. Thus, the texture that stores the depth map covers a reduced area of the view frustum. And, since the split closest to the camera covers less space, the depth resolution is increased.

As it can be deduced form explanation above, We will need as many depth textures as splits, and we will also change the light view and projection matrices for each of the, Hence, the steps to be done in order to apply CSMs are:

- Divide the view frustum into n splits.
- While rendering the depth map, for each split:
    - Calculate light view and projection matrices.
    - Render the scene from light’s perspective into a separate depth map
- While rendering the scene:
    - Use the depths maps calculated above.
    - Determine the split that the fragment to be drawn belongs to.
    - Calculate shadow factor as in shadow maps.

As you can see, the main drawback of CSMs is that we need to render the scene, from light's perspective, for each split. This is why is often only used for open spaces. Anyway, we will see how we can easily reduce that overhead.

So let's start examining the code, but before we continue a little warning, I will not include the full source code here since it would be very tedious to red. Instead, I will present the main classes their responsibilities and the fragments that may require further explanation in order to get a good understanding. All the shading related classes have been moved to a new package called `org.lwjglb.engine.graph.shadow`.

The code that renders shadows, that is, the scene from light's perspective has been moved to the `ShadowRenderer` class. (That code was previously contained in the `Renderer` class).

The class defines the following constants:

```
public static final int NUM_CASCADES = 3;
public static final float[] CASCADE_SPLITS = new float[]{Window.Z_FAR / 20.0f, Window.
Z_FAR / 10.0f, Window.Z_FAR};
```

The first one is the number of cascades or splits. The second one defines where the far z plane is located for each of these splits. As you can see they are not equally spaced. The split that is closer to the camera has the shortest distance in the z plane.

The class also stores the reference to the shader program used to render the depth map, a list with the information associated to each split, modelled by the `ShadowCascade` class, and a reference to the object that whill host the depth mapth information (tetxures), modelled by the `ShadowBuffer` class.

The `ShadowRenderer` class has methods for setting up the shaders and the required attributes and a render method . The `render` method is defined like this.

www.dbooks.org

```
public void render(Window window, Scene scene, Camera camera, Transformation transform
ation, Renderer renderer) {
    update(window, camera.getViewMatrix(), scene);

    // Setup view port to match the texture size
    glBindFramebuffer(GL_FRAMEBUFFER, shadowBuffer.getDepthMapFBO());
    glViewport(0, 0, ShadowBuffer.SHADOW_MAP_WIDTH, ShadowBuffer.SHADOW_MAP_HEIGHT);
    glClear(GL_DEPTH_BUFFER_BIT);

    depthShaderProgram.bind();

    // Render scene for each cascade map
    for (int i = 0; i < NUM_CASCADES; i++) {
        ShadowCascade shadowCascade = shadowCascades.get(i);

        depthShaderProgram.setUniform("orthoProjectionMatrix", shadowCascade.getOrthoP
rojMatrix());
        depthShaderProgram.setUniform("lightViewMatrix", shadowCascade.getLightViewMat
rix());

        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, sha
dowBuffer.getDepthMapTexture().getIds()[i], 0);
        glClear(GL_DEPTH_BUFFER_BIT);

        renderNonInstancedMeshes(scene, transformation);

        renderInstancedMeshes(scene, transformation);
    }

    // Unbind
    depthShaderProgram.unbind();
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

As you can see, I similar to the previous render method for shadow maps, except that we are performing several rendering passes, one per split. In each pass we change the light view matrix and the orthographic projection matrix with the information contained in the associated `ShadowCascade` instande.

Also, in each pass, we need to change the texture we are using. Each pass will render the depth information to a different texture. This information is stored in the `ShadowBuffer` class, and is setup to be used by the FBO with this line:

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, shadowBuffe
r.getDepthMapTexture().getIds()[i], 0);
```

As it's just have been mentioned, the `ShadowBuffer` class stores the information related to the textures used to store depth information. The code its very similar to the code used in the shadows chapter, excep t that we are using texture arrays. Thus, we have created a new class, `ArrTexture`, that creates an array of textures with the same attributes. This class also provides a `bind` method that binds all the texture arrays for using them in the scene shader. The method receives a parameter, with the texture unit to start with.

```
public void bindTextures(int start) {
    for (int i = 0; i < ShadowRenderer.NUM_CASCADES; i++) {
        glActiveTexture(start + i);
        glBindTexture(GL_TEXTURE_2D, depthMap.getIds()[i]);
    }
}
```

`ShadowCascade` class, stores the light view and orthographic projection matrices associated to one split. Each split is defined by a near and a z far plan distance, and with that information the matrices are calculated accordingly.

The class provided and update method which, taking as an input the view natrix and the light direction. The method calculates the view frustum corners in world space and then calculates the light position. That position is calculated going back, suing the light direction, from the frustum centre to a distance equal to the distance between the far and near z planes.

www.dbooks.org

```
public void update(Window window, Matrix4f viewMatrix, DirectionalLight light) {
    // Build projection view matrix for this cascade
    float aspectRatio = (float) window.getWidth() / (float) window.getHeight();
    projViewMatrix.setPerspective(Window.FOV, aspectRatio, zNear, zFar);
    projViewMatrix.mul(viewMatrix);

    // Calculate frustum corners in world space
    float maxZ = -Float.MAX_VALUE;
    float minZ =  Float.MAX_VALUE;
    for (int i = 0; i < FRUSTUM_CORNERS; i++) {
        Vector3f corner = frustumCorners[i];
        corner.set(0, 0, 0);
        projViewMatrix.frustumCorner(i, corner);
        centroid.add(corner);
        centroid.div(8.0f);
        minZ = Math.min(minZ, corner.z);
        maxZ = Math.max(maxZ, corner.z);
    }

    // Go back from the centroid up to max.z - min.z in the direction of light
    Vector3f lightDirection = light.getDirection();
    Vector3f lightPosInc = new Vector3f().set(lightDirection);
    float distance = maxZ - minZ;
    lightPosInc.mul(distance);
    Vector3f lightPosition = new Vector3f();
    lightPosition.set(centroid);
    lightPosition.add(lightPosInc);

    updateLightViewMatrix(lightDirection, lightPosition);

    updateLightProjectionMatrix();
}
```

With the light position and the light direction. we can construct the light view matrix. This is done in the `updateLightViewMatrix`:

```
private void updateLightViewMatrix(Vector3f lightDirection, Vector3f lightPosition) {
    float lightAngleX = (float) Math.toDegrees(Math.acos(lightDirection.z));
    float lightAngleY = (float) Math.toDegrees(Math.asin(lightDirection.x));
    float lightAngleZ = 0;
    Transformation.updateGenericViewMatrix(lightPosition, new Vector3f(lightAngleX, li
ghtAngleY, lightAngleZ), lightViewMatrix);
}
```

Finally, we need to construct the orthographic projection matrix. This is done in the `updateLightProjectionMatrix` method. The method is to transform the view frustum coordinates into light space. We then get the minimum and maximum values for the x, y

coordinates to construct the bounding box that encloses the view frustum. Near z plane can be set to 0 and the far z plane to the distance between the maximum and minimum value of the coordinates.

```
private void updateLightProjectionMatrix() {
    // Now calculate frustum dimensions in light space
    float minX =  Float.MAX_VALUE;
    float maxX = -Float.MAX_VALUE;
    float minY =  Float.MAX_VALUE;
    float maxY = -Float.MAX_VALUE;
    float minZ =  Float.MAX_VALUE;
    float maxZ = -Float.MAX_VALUE;
    for (int i = 0; i < FRUSTUM_CORNERS; i++) {
        Vector3f corner = frustumCorners[i];
        tmpVec.set(corner, 1);
        tmpVec.mul(lightViewMatrix);
        minX = Math.min(tmpVec.x, minX);
        maxX = Math.max(tmpVec.x, maxX);
        minY = Math.min(tmpVec.y, minY);
        maxY = Math.max(tmpVec.y, maxY);
        minZ = Math.min(tmpVec.z, minZ);
        maxZ = Math.max(tmpVec.z, maxZ);
    }
    float distz = maxZ - minZ;

    orthoProjMatrix.setOrtho(minX, maxX, minY, maxY, 0, distz);
}
```

Remember that the orthographic projection is like a bounding box that should enclose all the objects that will be rendered. That bounding box is expressed in light view coordinates space. Thus, what we are doing is calculate the minimum bounding box, axis aligned with the light position , hat encloses the view frustum.

The `Renderer` class has been modified to use the classes in the view package and also to modify the information that is passed to the renderers. In the renderer we need to deal with the model, the model view, and the model light matrices. In previous chapters we used the model–view / light–view matrices, to reduce the number of operations. In this case, we opted to simplify the number of elements to be passed and now we are passing just the model, view and light matrices to the shaders. Also, for particles, we need to preserve the scale, since we no longer pass the model view matrix, that information is lost now. We reuse the attribute used to mark selected items to set that scale information. In the particles shader we will use that value to set the scaling again.

In the scene vertex shader, we calculate model light view matrix for each split, and pass it as an output to the fragment shader.

www.dbooks.org

```
mvVertexPos = mvPos.xyz;
for (int i = 0 ; i < NUM_CASCADES ; i++) {
    mlightviewVertexPos[i] = orthoProjectionMatrix[i] * lightViewMatrix[i] * modelMatr
ix * vec4(position, 1.0);
}
```

In the fragment shader we use those values to query the appropriate depth map depending on the split that the fragment is. This needs to be done in the fragment shader since, for a specific item, their fragments may reside in different splits.

Also, in the fragment shader we must decide which split we are into. In order to do that, we use the z value of the fragment and compare it with the maximum z value for each split. That is, the z far plane value. That information is passed as a new uniform:

```
uniform float cascadeFarPlanes[NUM_CASCADES];
```

We calculate de split like this. The variable `idx` will have the split to be used:

```
int idx;
for (int i=0; i<NUM_CASCADES; i++)
{
    if ( abs(mvVertexPos.z) < cascadeFarPlanes[i] )
    {
        idx = i;
        break;
    }
}
```

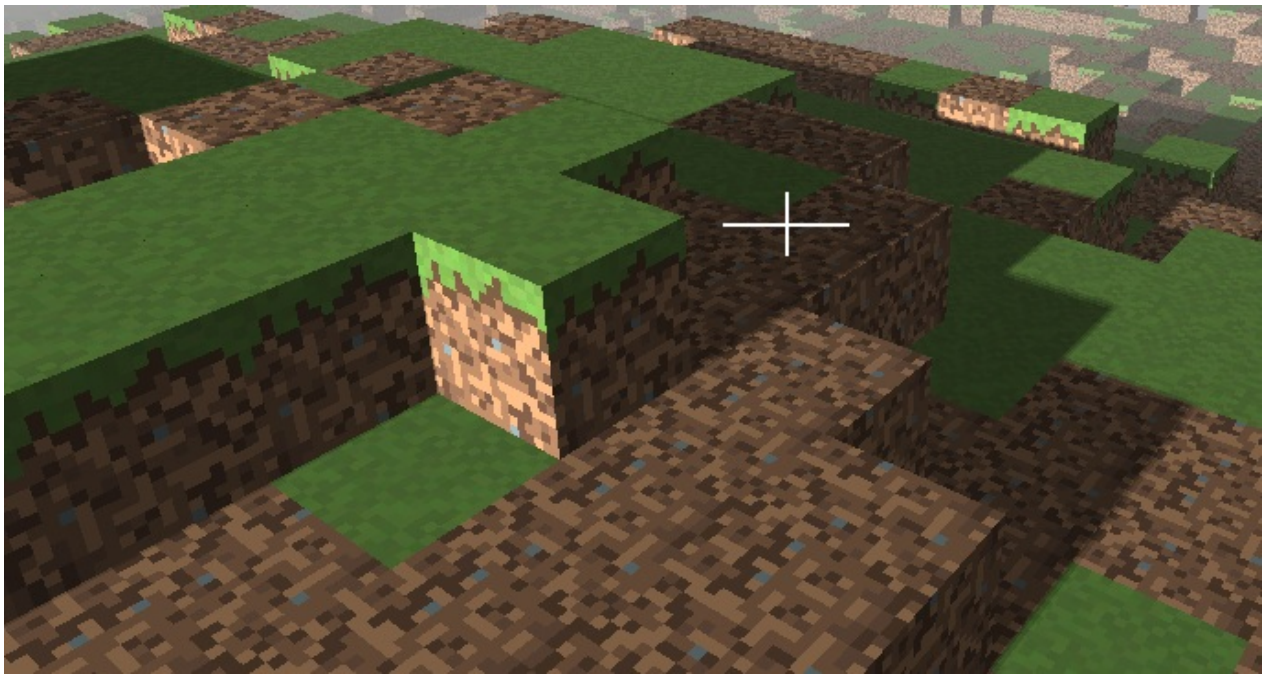
Also, in the scene shaders we need to pass an array of textures, an array of `sampler2D's`, to use the depth map, the texture, associated to the split we are into. The source code, instead of using an array uses a list of uniforms that will hold the texture unit that is used to refer to the depth map associated to each split.

```
uniform sampler2D normalMap;
uniform sampler2D shadowMap_0;
uniform sampler2D shadowMap_1;
uniform sampler2D shadowMap_2;
```

Changing it to an array of uniforms causes problems with other textures that are difficult to track for this sample. In any case, you can try changing it in your code.

The rest of the changes in the source code, and the shaders are just adaptations required by the changes described above. You can check it directly over the source code.

Finally, when introducing these changes you may see that performance has dropped. This is due to the fact that we are rendering three times the depth map. We can mitigate this effect by avoiding rendering at all when the scene has not changed. If the camera has not been moved or the scene items have not changed we do not need to render again and again the depth map. The depth maps are stored in textures, so they are not wiped out for each render call. Thus, we have added a new variable to the render method that indicates if this has changed, avoiding updating the depth maps it remains the same. This increases the FPS dramatically. At the end, you will get something like this:



www.dbooks.org

Assimp

Static Meshes

The capability of loading complex 3d models in different formats is crucial in order to write a game. The task of writing parsers for some of them would require lots of work. Even just supporting a single format can be time consuming. For instance, the wavefront loader described in chapter 9, only parses a small subset of the specification (materials are not handled at all).

Fortunately, the Assimp library already can be used to parse many common 3D formats. It's a C++ library which can load static and animated models in a variety of formats. LWJGL provides the bindings to use them from Java code. In this chapter, we will explain how it can be used.

The first thing is adding assimp maven dependencies to the project pom.xml. We need to add compile time and runtime dependencies.

```
<dependency>
    <groupId>org.lwjgl</groupId>
    <artifactId>lwjgl-assimp</artifactId>
    <version>${lwjgl.version}</version>
</dependency>
<dependency>
    <groupId>org.lwjgl</groupId>
    <artifactId>lwjgl-assimp</artifactId>
    <version>${lwjgl.version}</version>
    <classifier>${native.target}</classifier>
    <scope>runtime</scope>
</dependency>
```

Once the dependencies has been set, we will cerate a new class named StaticMeshesLoader that will be used to load meshes with no animations. The class defines two static public methods:

```
public static Mesh[] load(String resourcePath, String texturesDir) throws Exception {
    return load(resourcePath, texturesDir, aiProcess_JoinIdenticalVertices | aiProcess
_Triangulate | aiProcess_FixInfacingNormals);
}

public static Mesh[] load(String resourcePath, String texturesDir, int flags) throws E
xception {
    // ....
```

Both methods have the following arguments:

- `resourcePath` : The path to the file where the model file is located. This is an absolute path, because Assimp may need to load additional files and may use the same base path as the resource path (For instance, material files for wavefront, OBJ, files). If you embed your resources inside a JAR file, Assimp will not be able to import it, so uts must be a file system path.
- `texturesDir` : The path to the directory that will hold the textures for this model. This a CLASSPATH relative path. For instance, a wavefront material file may define several texture files. The code, expect this files to be located in the `texturesDir` directory. If you find texture loading errors you may need to manually tweak these paths in the model file.

The second method has an extra argument named `flags` . This parameter allows to tune the loading process. The firstmethodsjust invokes the secondoneand passes some values that are useful in most of the situations:

- `aiProcess_JoinIdenticalVertices` : This flag reduces the number of vertices that are used, identifiying those that can be reused between faces.
- `aiProcess_Triangulate` : The model may use quads or other geometries to define their elements. Since we are only dealing with triangles, we must use this flag to split all he faces into triangles (if needed).
- `aiProcess_FixInfacingNormals` : This flags try to reverse normals that may point inwards.

There are many other flags that can be used, you chan check them in the LWJGL Javadoc documentation.

Let’s go back to the second constructor. The first thing we do is invoke the `aiImportFile` method to load the model with the selectedflags.

www.dbooks.org

```
AIScene aiScene = aiImportFile(resourcePath, flags);
if (aiScene == null) {
    throw new Exception("Error loading model");
}
```

The rest of the code for the constructor is a as follows:

```
int numMaterials = aiScene.mNumMaterials();
PointerBuffer aiMaterials = aiScene.mMaterials();
List<Material> materials = new ArrayList<>();
for (int i = 0; i < numMaterials; i++) {
    AIMaterial aiMaterial = AIMaterial.create(aiMaterials.get(i));
    processMaterial(aiMaterial, materials, texturesDir);
}

int numMeshes = aiScene.mNumMeshes();
PointerBuffer aiMeshes = aiScene.mMeshes();
Mesh[] meshes = new Mesh[numMeshes];
for (int i = 0; i < numMeshes; i++) {
    AIMesh aiMesh = AIMesh.create(aiMeshes.get(i));
    Mesh mesh = processMesh(aiMesh, materials);
    meshes[i] = mesh;
}

return meshes;
```

We process the materials contained in the model. Materials define colour and textures to be used by the meshes that compose the model. Then we process the different meshes. A model can define several meshes and each of them can use one of the materials defined for the model.

If you examine the code above you may see that many of the calls to the Assimp library return `PointerBuffer` instances. You can think about them like C pointers, they just point to a memory region which contain data. You need to know in advance the type of data that they hold in order to process them. In the case of materials, we iterate over that buffer creating instances of theAIMaterialclass. In the second case, we iterate over the buffer that holds mesh data creating instance of the `AIMesh` class.

Let's examine the `processMaterial` method.

```
private static void processMaterial(AIMaterial aiMaterial, List<Material> materials, S
tring texturesDir) throws Exception {
    AIColor4D colour = AIColor4D.create();

    AIString path = AIString.calloc();
    Assimp.aiGetMaterialTexture(aiMaterial, aiTextureType_DIFFUSE, 0, path, (IntBuffer
) null, null, null, null, null, null);
    String textPath = path.dataString();
    Texture texture = null;
    if (textPath != null && textPath.length() > 0) {
        TextureCache textCache = TextureCache.getInstance();
        texture = textCache.getTexture(texturesDir + "/" + textPath);
    }

    Vector4f ambient = Material.DEFAULT_COLOUR;
    int result = aiGetMaterialColor(aiMaterial, AI_MATKEY_COLOR_AMBIENT, aiTextureType
_NONE, 0, colour);
    if (result == 0) {
        ambient = new Vector4f(colour.r(), colour.g(), colour.b(), colour.a());
    }

    Vector4f diffuse = Material.DEFAULT_COLOUR;
    result = aiGetMaterialColor(aiMaterial, AI_MATKEY_COLOR_DIFFUSE, aiTextureType_NON
E, 0, colour);
    if (result == 0) {
        diffuse = new Vector4f(colour.r(), colour.g(), colour.b(), colour.a());
    }

    Vector4f specular = Material.DEFAULT_COLOUR;
    result = aiGetMaterialColor(aiMaterial, AI_MATKEY_COLOR_SPECULAR, aiTextureType_NO
NE, 0, colour);
    if (result == 0) {
        specular = new Vector4f(colour.r(), colour.g(), colour.b(), colour.a());
    }

    Material material = new Material(ambient, diffuse, specular, 1.0f);
    material.setTexture(texture);
    materials.add(material);
}
```

We check if the material defines a texture or not. If so, we load the texture. We have created a new class named `TextureCache` which caches textures. This is due to the fact that several meshes may share the same texture and we do not want to waste space loading again and again the same data. Then we try to get the colours of the material for the ambient, diffuse and specular components. Fortunately, the definition that we had for a material already contained that information.

www.dbooks.org

The `TextureCache` definition is very simple is just a map that indexes the different textures by the path to the texture file (You can check directly in the source code). Due to the fact, that now textures may use different image formats (PNG, JPEG, etc.), we have modified the way that textures are loaded. Instead of using the PNG library, we now use the STB library to be able to load more formats.

Let's go back to the `StaticMeshesLoader` class. The `processMesh` is defined like this.

```
private static Mesh processMesh(AIMesh aiMesh, List<Material> materials) {
    List<Float> vertices = new ArrayList<>();
    List<Float> textures = new ArrayList<>();
    List<Float> normals = new ArrayList<>();
    List<Integer> indices = new ArrayList();

    processVertices(aiMesh, vertices);
    processNormals(aiMesh, normals);
    processTextCoords(aiMesh, textures);
    processIndices(aiMesh, indices);

    Mesh mesh = new Mesh(Utils.listToArray(vertices),
        Utils.listToArray(textures),
        Utils.listToArray(normals),
        Utils.listIntToArray(indices)
    );
    Material material;
    int materialIdx = aiMesh.mMaterialIndex();
    if (materialIdx >= 0 && materialIdx < materials.size()) {
        material = materials.get(materialIdx);
    } else {
        material = new Material();
    }
    mesh.setMaterial(material);

    return mesh;
}
```

A `Mesh` is defined by a set of vertices position, normals directions, texture coordinates and indices. Each of these elements are processed in the `processVertices`, `processNormals`, `processTextCoords` and `processIndices` method. A Mesh also may point to a material, using its index. If the index corresponds to the previously processed materials we just simply associate them to the `Mesh`.

The `processXXX` methods are very simple, they just invoke the corresponding method over the `AIMesh` instance that returns the desired data. For instance, the process processVertices is defined like this:

```
private static void processVertices(AIMesh aiMesh, List<Float> vertices) {
    AIVector3D.Buffer aiVertices = aiMesh.mVertices();
    while (aiVertices.remaining() > 0) {
        AIVector3D aiVertex = aiVertices.get();
        vertices.add(aiVertex.x());
        vertices.add(aiVertex.y());
        vertices.add(aiVertex.z());
    }
}
```

You can see that get get a buffer to the vertices by invoking the `mVertices` method. We just simply process them to create a `List` of floats that contain the vertices positions. Since, the method retyrns jusst a buffer you could pass that information directly to the OpenGL methods that create vertices. We do not do it that way for two reasons. The first one is try to reduce as much as possible the modifications over the code base. Second one is that by loading into an intermediate structure you may be able to perform some pros-processing tasks and even debug the loading process.

If you want a sample of the much more efficient approach, that is, directly passing the buffers to OpenGL, you can check this sample.

The `StaticMeshesLoader` makes the `OBJLoader` class obsolete, so it has been removed form the base source code. A more complex OBJ file is provided as a sample, if you run it you will see something like this:



Animations

www.dbooks.org

Now that we have used assimp for loading static meshes we can proceed with animations. If you recall form the animations chapter, the VAO associated to a mesh contains the vertices positions, the texture coordinates, the indices and a list of weights that should be applied to joint positions to modulate final vertex position.

VAO

0	positions
1	text coords
2	normals
3	indices
4	joint indices
5	weights

Each vertex position has associated a list of four weights that change the final position, referring the bones indices that will be combined to determine its final position. Each frame a list of transformation matrices are loaded, as uniforms, for each joint. With that information the final position is calculated.

In the animation chapter, we developed a MD5 parser to load animated meshes. In this chapter we will use assimp library. This will allow us to load many more formats besides MD5, such as COLLADA, FBX, etc.

Before we start coding let's clarify some terminology. In this chapter we will refer to bones and joints indistinguishably. A joint / bone is are just elements that affect vertices, and that have a parent forming a hierarchy. MD5 format uses the term joint, but assimp uses the term bone.

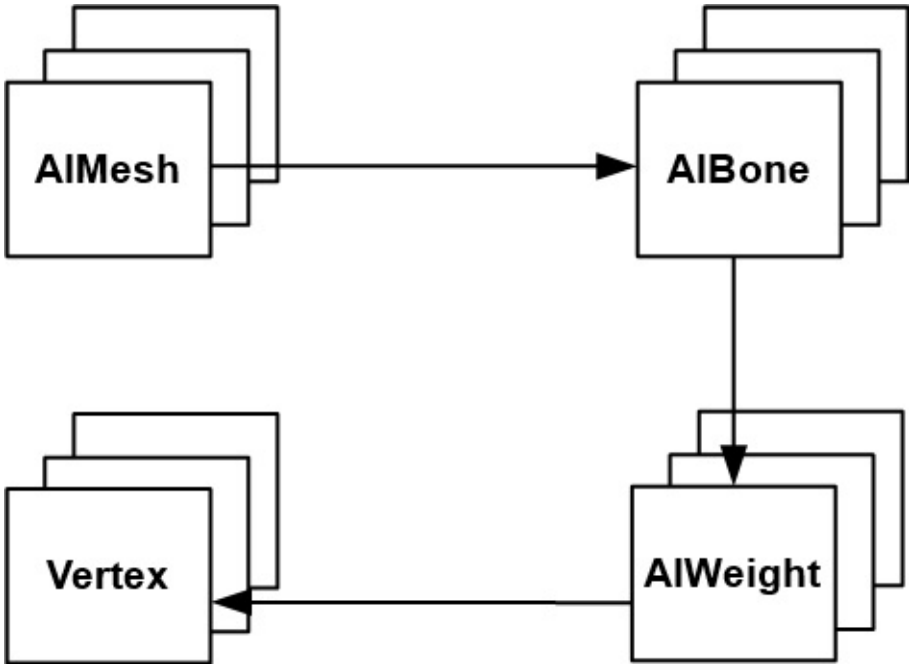
Let's review first the structures handled by assimp that contain animation information. We will start first with the bones and weights information. For each Mesh, we can access the vertices positions, texture coordinates and indices. Meshes store also a list of bones. Each bone is defined by the following attributes:

- A name.
- An offset matrix: This will used later to compute the final transformations that should be used by each bone.

Bones also point to a list of weights, each weights. Each weights is defined by the following attributes:

- A weight factor, that is, the number that will be used to modulate the influence of the bone's transformation associated to each vertex.
- A vertex identifier, that is, the vertex associated to the current bone.

The following picture shows the relationships between all these elements.



Hence, the first thing that we must do is to construct the list of vertices positions, the bones / joints / indices and the associated weights from the structure above. Once we have done that, we need to pre-calculate the transformation matrices for each bone / joint for all the animation frames defined in the model.

Assimp scene object defines a Node's hierarchy. Each Node is defined by a name a list of children node. Animations use these nodes to define the transformations that should be applied to. This hierarchy is defined indeed the bones' hierarchy. Every bone is a node, and has a parent, except the root node, and possible a set of children. There are special nodes that are not bones, they are used to group transformations, and should be handled when calculating the transformations. Another issue is that these Node's hierarchy is defined fro the whole model, we do not have separate hierarchies for each mesh.

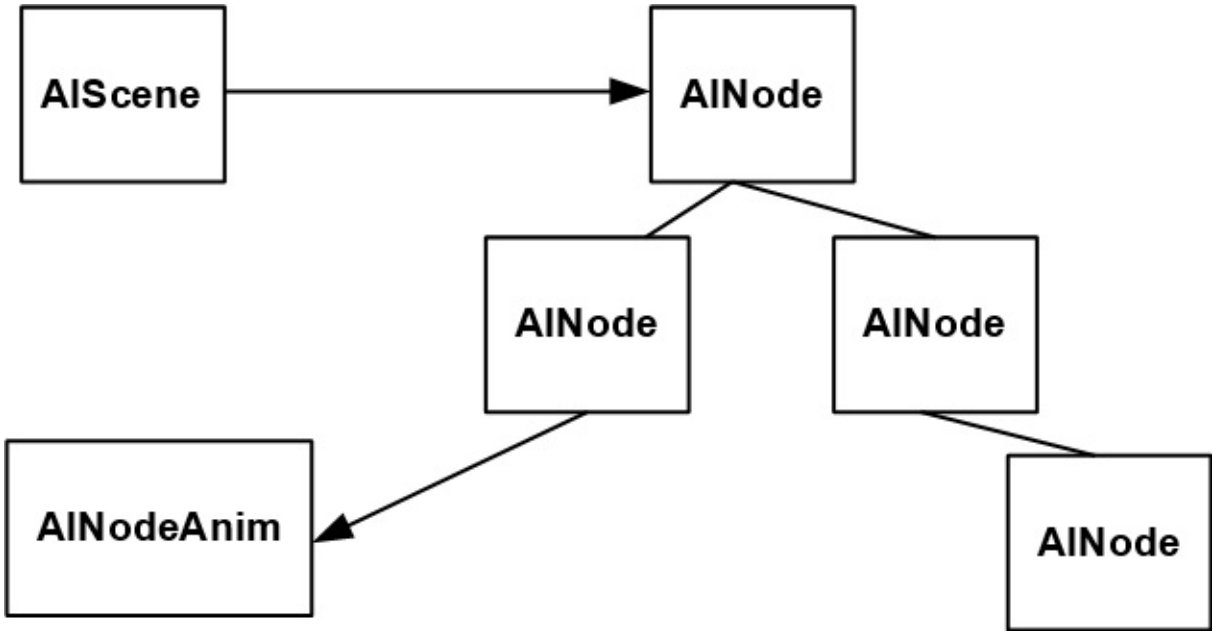
A scene also defines a set of animations. A single model can have more than one animation. You can have animations for a model to walk, run etc. Each of these animations define different transformations. An animation has the following attributes:

- A name.
- A duration. That is, the duration in time of the animation. name may seem confusing

www.dbooks.org

since an animation is the list of transformations that should be applied to each node for each different frame.
- A list of animation channels. An animation channel, contains, for a specific instant in time the translation, rotation and scaling informations that should be applied to each node. The class that models the data contained in the animation channels is the AINodeAnim.

The following figure shows the relationships between all the elements described above.



For a specific instant of time, for a frame, the transformation to be applied to a bone is the transformation defined in the animation channel for that instant, multiplied by the transformations of all the parent nodes up to the root node. Hence, we need to reorder the information stored in the scene, the process is as follows:

- Construct the node hierarchy.
- For each animation, iterate overeach animation channel (for each animation node): Construct the transformation matrices for all the frames. The transformation m matrix is the composition of the translation, rotation and scale matrix .
- Reorder that information for each frame: Construct the final transformations to be applied for each bone in the Mesh. This is achieved by multiplying the transformation matrix of the bone (of the associated node) by the transformation matrices of all the parent nodes up to the root node.

So let’s start coding. We will create firs a class named `AnimMeshesLoader` which extends from `StaticMeshesLoader`, but instead of returning an array of Meshes, it returns an `AnimGameItem` instance. It defines two public methods for that:

```
public static AnimGameItem loadAnimGameItem(String resourcePath, String texturesDir)
        throws Exception {
    return loadAnimGameItem(resourcePath, texturesDir,
            aiProcess_GenSmoothNormals | aiProcess_JoinIdenticalVertices | aiProcess_T
riangulate
            | aiProcess_FixInfacingNormals | aiProcess_LimitBoneWeights);
}

public static AnimGameItem loadAnimGameItem(String resourcePath, String texturesDir, i
nt flags)
        throws Exception {
    AIScene aiScene = aiImportFile(resourcePath, flags);
    if (aiScene == null) {
        throw new Exception("Error loading model");
    }

    int numMaterials = aiScene.mNumMaterials();
    PointerBuffer aiMaterials = aiScene.mMaterials();
    List<Material> materials = new ArrayList<>();
    for (int i = 0; i < numMaterials; i++) {
        AIMaterial aiMaterial = AIMaterial.create(aiMaterials.get(i));
        processMaterial(aiMaterial, materials, texturesDir);
    }

    List<Bone> boneList = new ArrayList<>();
    int numMeshes = aiScene.mNumMeshes();
    PointerBuffer aiMeshes = aiScene.mMeshes();
    Mesh[] meshes = new Mesh[numMeshes];
    for (int i = 0; i < numMeshes; i++) {
        AIMesh aiMesh = AIMesh.create(aiMeshes.get(i));
        Mesh mesh = processMesh(aiMesh, materials, boneList);
        meshes[i] = mesh;
    }

    AINode aiRootNode = aiScene.mRootNode();
    Matrix4f rootTransfromation = AnimMeshesLoader.toMatrix(aiRootNode.mTransformation
());
    Node rootNode = processNodesHierarchy(aiRootNode, null);
    Map<String, Animation> animations = processAnimations(aiScene, boneList, rootNode,
 rootTransfromation);
    AnimGameItem item = new AnimGameItem(meshes, animations);

    return item;
}
```

The methods are quite similar to the ones defined in the `StaticMeshesLoader` with the following differences:

- The method that passes a default set of loading flags, uses this new parameter: `aiProcess_LimitBoneWeights`. This will limit the maximum number of weights that affect a

www.dbooks.org

vertex to four (This is also the maximum value that we are currently supporting from the animations chapter).
- The method that actually loads the model just loads the different meshes but it first calculates the node hierarchy and then calls to the `processAnimations` at the end to build an `AnimGameItem` instance.

The `processMesh` method is quite similar to the one in the `StaticMeshesLoader` with the exception that it creates Meshes passing joint indices and weights as a parameter:

```
processBones(aiMesh, boneList, boneIds, weights);

Mesh mesh = new Mesh(Utils.listToArray(vertices), Utils.listToArray(textures),
    Utils.listToArray(normals), Utils.listIntToArray(indices),
    Utils.listIntToArray(boneIds), Utils.listToArray(weights));
```

The joint indices and weights are calculated in the `processBones` method:

```
private static void processBones(AIMesh aiMesh, List<Bone> boneList, List<Integer> bon
eIds,
        List<Float> weights) {
    Map<Integer, List<VertexWeight>> weightSet = new HashMap<>();
    int numBones = aiMesh.mNumBones();
    PointerBuffer aiBones = aiMesh.mBones();
    for (int i = 0; i < numBones; i++) {
        AIBone aiBone = AIBone.create(aiBones.get(i));
        int id = boneList.size();
        Bone bone = new Bone(id, aiBone.mName().dataString(), toMatrix(aiBone.mOffsetM
atrix()));
        boneList.add(bone);
        int numWeights = aiBone.mNumWeights();
        AIVertexWeight.Buffer aiWeights = aiBone.mWeights();
        for (int j = 0; j < numWeights; j++) {
            AIVertexWeight aiWeight = aiWeights.get(j);
            VertexWeight vw = new VertexWeight(bone.getBoneId(), aiWeight.mVertexId(),
                    aiWeight.mWeight());
            List<VertexWeight> vertexWeightList = weightSet.get(vw.getVertexId());
            if (vertexWeightList == null) {
                vertexWeightList = new ArrayList<>();
                weightSet.put(vw.getVertexId(), vertexWeightList);
            }
            vertexWeightList.add(vw);
        }
    }

    int numVertices = aiMesh.mNumVertices();
    for (int i = 0; i < numVertices; i++) {
        List<VertexWeight> vertexWeightList = weightSet.get(i);
        int size = vertexWeightList != null ? vertexWeightList.size() : 0;
        for (int j = 0; j < Mesh.MAX_WEIGHTS; j++) {
            if (j < size) {
                VertexWeight vw = vertexWeightList.get(j);
                weights.add(vw.getWeight());
                boneIds.add(vw.getBoneId());
            } else {
                weights.add(0.0f);
                boneIds.add(0);
            }
        }
    }
}
```

This method traverses the bone definition for a specific mesh, getting their weights and generating filling up three lists:

- `boneList`: It contains a list of nodes, with their offset matrices. It will uses later on to calculate nodes transformations. A new class named `Bone` has been created to hold that information. This list will contain the bones for all the meshes.

www.dbooks.org

- `boneIds` : It contains just the identifiers of the bones for each vertex of the `Mesh` . Bones are identified by its position when rendering. This list only contains the bones for a specific Mesh.
- `weights` : It contains the weights for each vertex of the `Mesh` to be applied for the associated bones.

The information contained in the `weights` and `boneIds` is used to construct the `Mesh` data. The information contained in the boneList will be used later when calculating animation data.

Let's go back to the `loadAnimGameItem` method. Once we have created the Meshes, we also get the transformation which is applied to the root node which will be used also to calculate the final transformatio. After that , we need to process the hierarchy of nodes, which is done in the `processNodesHierarchy` method. This method is quite simple, It just traverses the nodes hierarchy starting from the root node constructing a tree of nodes.

```
private static Node processNodesHierarchy(AINode aiNode, Node parentNode) {
    String nodeName = aiNode.mName().dataString();
    Node node = new Node(nodeName, parentNode);

    int numChildren = aiNode.mNumChildren();
    PointerBuffer aiChildren = aiNode.mChildren();
    for (int i = 0; i < numChildren; i++) {
        AINode aiChildNode = AINode.create(aiChildren.get(i));
        Node childNode = processNodesHierarchy(aiChildNode, node);
        node.addChild(childNode);
    }

    return node;
}
```

We have created a new `Node` class that will contain the relevant information of `AINode` instances, and provides find methods to locate the nodes hierarchy to find a node by its name. Back in the `loadAnimGameItem` method, we just use that information to calculate the animations in the `processAnimations` method. This method returns a `Map` of `Animation` instances. Remember that a model can have more than one animation, so they are stored indexed by their names. With that information we can finally construct an `AnimAgameItem` instance.

The `processAnimations` method is defined like this

```
private static Map<String, Animation> processAnimations(AIScene aiScene, List<Bone> bo
neList,
        Node rootNode, Matrix4f rootTransformation) {
    Map<String, Animation> animations = new HashMap<>();

    // Process all animations
    int numAnimations = aiScene.mNumAnimations();
    PointerBuffer aiAnimations = aiScene.mAnimations();
    for (int i = 0; i < numAnimations; i++) {
        AIAnimation aiAnimation = AIAnimation.create(aiAnimations.get(i));

        // Calculate transformation matrices for each node
        int numChanels = aiAnimation.mNumChannels();
        PointerBuffer aiChannels = aiAnimation.mChannels();
        for (int j = 0; j < numChanels; j++) {
            AINodeAnim aiNodeAnim = AINodeAnim.create(aiChannels.get(j));
            String nodeName = aiNodeAnim.mNodeName().dataString();
            Node node = rootNode.findByName(nodeName);
            buildTransFormationMatrices(aiNodeAnim, node);
        }

        List<AnimatedFrame> frames = buildAnimationFrames(boneList, rootNode, rootTran
sformation);
        Animation animation = new Animation(aiAnimation.mName().dataString(), frames,
aiAnimation.mDuration());
        animations.put(animation.getName(), animation);
    }
    return animations;
}
```

For each animation, animation channels are processed. Each channel defines the different transformations that should be applied over time for a node. The transformations defined for each node are defined in the `buildTransFormationMatricesmethod`. These matrices are store for each node. Once the nodes hierarchy is filled up with that information we can construct the animation frames.

Let’s first review the `buildTransFormationMatrices` method:

www.dbooks.org

```
private static void buildTransFormationMatrices(AINodeAnim aiNodeAnim, Node node) {
    int numFrames = aiNodeAnim.mNumPositionKeys();
    AIVectorKey.Buffer positionKeys = aiNodeAnim.mPositionKeys();
    AIVectorKey.Buffer scalingKeys = aiNodeAnim.mScalingKeys();
    AIQuatKey.Buffer rotationKeys = aiNodeAnim.mRotationKeys();

    for (int i = 0; i < numFrames; i++) {
        AIVectorKey aiVecKey = positionKeys.get(i);
        AIVector3D vec = aiVecKey.mValue();

        Matrix4f transfMat = new Matrix4f().translate(vec.x(), vec.y(), vec.z());

        AIQuatKey quatKey = rotationKeys.get(i);
        AIQuaternion aiQuat = quatKey.mValue();
        Quaternionf quat = new Quaternionf(aiQuat.x(), aiQuat.y(), aiQuat.z(), aiQuat.
w());
        transfMat.rotate(quat);

        if (i < aiNodeAnim.mNumScalingKeys()) {
            aiVecKey = scalingKeys.get(i);
            vec = aiVecKey.mValue();
            transfMat.scale(vec.x(), vec.y(), vec.z());
        }

        node.addTransformation(transfMat);
    }
}
```

As you can see, an `AINodeAnim` instance defines a set of keys that contain translation, rotation and scaling information. These keys are referred to specific instant of times. We assume that information is ordered in time, and construct a list of matrices that contain the transformation to be applied for each frame. That final calculation is done in the `buildAnimationFrames` method:

```
private static List<AnimatedFrame> buildAnimationFrames(List<Bone> boneList, Node root
Node,
        Matrix4f rootTransformation) {

    int numFrames = rootNode.getAnimationFrames();
    List<AnimatedFrame> frameList = new ArrayList<>();
    for (int i = 0; i < numFrames; i++) {
        AnimatedFrame frame = new AnimatedFrame();
        frameList.add(frame);

        int numBones = boneList.size();
        for (int j = 0; j < numBones; j++) {
            Bone bone = boneList.get(j);
            Node node = rootNode.findByName(bone.getBoneName());
            Matrix4f boneMatrix = Node.getParentTransforms(node, i);
            boneMatrix.mul(bone.getOffsetMatrix());
            boneMatrix = new Matrix4f(rootTransformation).mul(boneMatrix);
            frame.setMatrix(j, boneMatrix);
        }
    }

    return frameList;
}
```

This method returns a list of `AnimatedFrame` instances. Each `AnimatedFrame` instance will contain the list of transformations to be applied for each bone for a specific frame. This method just iterates over the list that contains all the bones. For each bone:

- Gets the associated node.
- Builds a transformation matrix by multiplying the transformation of the associated `Node` with all the transformations of their parents up to the root node. This is done in the `Node.getParentTransforms` method.
- It multiplies that matrix with the bone's offset matrix.
- The final transformation is calculated by multiplying the root's node transformation with the matrix calculated in the step above.

The rest of the changes in the source code are minor changes to adapt some structures. At the end you will be able to load animations like this one (you need yo press space par to change the frame).

www.dbooks.org



The complexity of this sample resides more in the adaptations of the assimp structures to adapt it to the engine used in the book and to pre-calculate the data for each frame. Beyond that, the concepts are similar to the ones presented in the animations chapter. You may try also to modify the source code to interpolate between frames to get smoother animations.

Deferred Shading

Up to now the way that we are rendering a 3D scene is called forward rendering. We first render the 3D objects and apply the texture and lightning effects in a fragment shader. This method is not very efficient if we have a complex fragment shader pass with many lights and complex effects. In addition to that we may end up applying these effects to fragments that may be later on discarded due to depth testing (although this is not exactly true if we enable early fragment testing).

In order to alleviate the problems described above we may change thy way that we render the scene by using a technical called deferred shading. With deferred shading we first render the geometry information that is required in later stages (in the fragment shader) to a buffer. The complex calculus required by the fragment shader are postponed, deferred, to a later stage when using the information stored in those buffers.

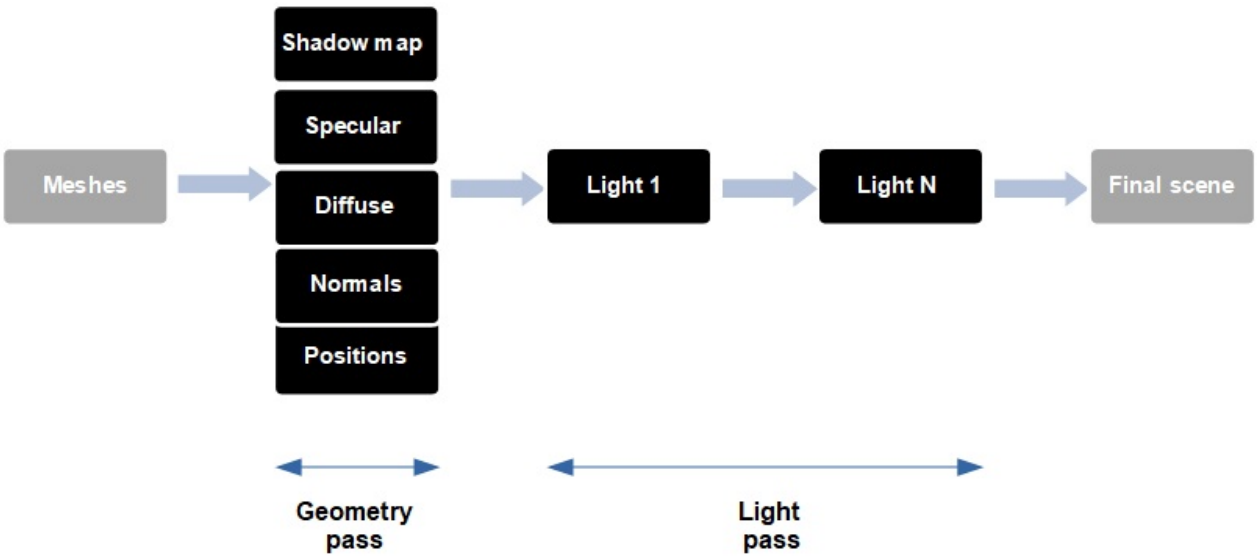
Hence, with deferred shading we perform two rendering passes. The first one, is the geometry pass, where we render the scene to a buffer that will contain the following information:

- The positions (in our case in light view coordinate system, although you may see other samples where world coordinates are used).
- The diffuse colours for each position.
- The specular component for each position.
- The normals at each position (also in light view coordinate system).
- Shadow map for the directional light (you may find that this step is done separately in other implementations).

All that information is stored in a buffer called G-Buffer.

The second pass, is called, the lightning pass. This pass takes a quad that fills up all the screen and generates the colour information for each fragment using the information contained in the G-Buffer. When we will be performing the lightning pass, the depth test will have already removed all the scene data that would not be seen. Hence, the number of operations to be done are restricted to what will be displayed on the screen.

www.dbooks.org



You may be asking if performing additional rendering passes will result in an increase of performance or not. The answer is that it depends. Deferred shading is usually used when you have many different light passes. In this case, the additional rendering steps are compensated by the reduction of operations that will be done in the fragment shader.

So let's start coding. The first task that we will be doing is create a new class for the G-Buffer. The class, named `GBuffer`, is defined like this:

```
package org.lwjglb.engine.graph;

import org.lwjgl.system.MemoryStack;
import org.lwjglb.engine.Window;
import java.nio.ByteBuffer;
import java.nio.IntBuffer;
import static org.lwjgl.opengl.GL11.*;
import static org.lwjgl.opengl.GL20.*;
import static org.lwjgl.opengl.GL30.*;

public class GBuffer {

    private static final int TOTAL_TEXTURES = 6;

    private int gBufferId;

    private int[] textureIds;

    private int width;

    private int height;
```

The class defines a constant that models the maximum number of buffers to be used. The identifier associated to the G-Buffer itself and an array for the individual buffers. The size of the screen is also stored.

Let’s review the constructor:

```
public GBuffer(Window window) throws Exception {
    // Create G-Buffer
    gBufferId = glGenFramebuffers();
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, gBufferId);

    textureIds = new int[TOTAL_TEXTURES];
    glGenTextures(textureIds);

    this.width = window.getWidth();
    this.height = window.getHeight();

    // Create textures for position, diffuse color, specular color, normal, shadow fac
tor and depth
    // All coordinates are in world coordinates system
    for(int i=0; i<TOTAL_TEXTURES; i++) {
        glBindTexture(GL_TEXTURE_2D, textureIds[i]);
        int attachmentType;
        switch(i) {
            case TOTAL_TEXTURES - 1:
                // Depth component
                glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32F, width, height, 0
, GL_DEPTH_COMPONENT, GL_FLOAT,
                        (ByteBuffer) null);
                attachmentType = GL_DEPTH_ATTACHMENT;
                break;
            default:
                glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB32F, width, height, 0, GL_RGB, GL
_FLOAT, (ByteBuffer) null);
                attachmentType = GL_COLOR_ATTACHMENT0 + i;
                break;
        }
        // For sampling
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

        // Attach the the texture to the G-Buffer
        glFramebufferTexture2D(GL_FRAMEBUFFER, attachmentType, GL_TEXTURE_2D, textureI
ds[i], 0);
    }

    try (MemoryStack stack = MemoryStack.stackPush()) {
        IntBuffer intBuff = stack.mallocInt(TOTAL_TEXTURES);
        int values[] = {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1, GL_COLOR_ATTACHMEN
T2, GL_COLOR_ATTACHMENT3, GL_COLOR_ATTACHMENT4, GL_COLOR_ATTACHMENT5};
        for(int i = 0; i < values.length; i++) {
            intBuff.put(values[i]);
        }
        intBuff.flip();
        glDrawBuffers(intBuff);
    }
```


www.dbooks.org

```
    // Unbind
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

The first thing that we do is create a frame buffer. Remember that a frame buffer is just an OpenGL objects that can be used to render operations instead of rendering to the screen. Then we generate a set of textures (6 textures), that will be associated to the frame buffer.

After that, we use a foor loop to initialize the textures. We have the following types:

- “Regular textures”, that will store positions, normals, the diffuse component, etc.
- A texture for storing the depth buffer. This will be our last texture.

Once the texture has been initialized, we enable sampling for them and attach them to the frame buffer. Each texture is attached using and identifier which starts at `GL_COLOR_ATTACHMENT0`. Each texture increments by one that id, so the positions are attached using `GL_COLOR_ATTACHMENT0`, the diffuse component uses `GL_COLOR_ATTACHMENT1` (which is `GL_COLOR_ATTACHMENT0 + 1`), and so on.

After all the textures have been created, we need to enable them to be used by the fargment shader for rendering. This is done with the `glDrawBuffers` call. We just pass the array with the idefintifiers of the colour attachments used (`GL_COLOR_ ATTACHMENT0` to `GL_COLOR_ATTACHMENT5`).

The rest of the class are just getter methods and the cleanup one.

```
public int getWidth() {
    return width;
}

public int getHeight() {
    return height;
}

public int getGBufferId() {
    return gBufferId;
}

public int[] getTextureIds() {
    return textureIds;
}

public int getPositionTexture() {
    return textureIds[0];
}

public int getDepthTexture() {
    return textureIds[TOTAL_TEXTURES-1];
}

public void cleanUp() {
    glDeleteFramebuffers(gBufferId);

    if (textureIds != null) {
        for (int i=0; i<TOTAL_TEXTURES; i++) {
            glDeleteTextures(textureIds[i]);
        }
    }
}
```

We will create a new class named `SceneBuffer` which is just another frame buffer. We will use it when performing the light pass. Instead of rendering directly to the screen we will render to this frame buffer. By doing it this way, we can apply the rest of the effects (such us fog, skybox, etc.). The class is defined like this.

```
package org.lwjglb.engine.graph;

import org.lwjglb.engine.Window;

import java.nio.ByteBuffer;

import static org.lwjgl.opengl.GL11.*;
import static org.lwjgl.opengl.GL30.*;

public class SceneBuffer {
```


www.dbooks.org

```
    private int bufferId;

    private int textureId;

    public SceneBuffer(Window window) throws Exception {
        // Create the buffer
        bufferId = glGenFramebuffers();
        glBindFramebuffer(GL_DRAW_FRAMEBUFFER, bufferId);

        // Create texture
        int[] textureIds = new int[1];
        glGenTextures(textureIds);
        textureId = textureIds[0];
        glBindTexture(GL_TEXTURE_2D, textureId);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB32F, window.getWidth(), window.getHeight(
), 0, GL_RGB, GL_FLOAT, (ByteBuffer) null);

        // For sampling
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

        // Attach the the texture to the G-Buffer
        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, te
xtureId, 0);

        // Unbind
        glBindFramebuffer(GL_FRAMEBUFFER, 0);
    }

    public int getBufferId() {
        return bufferId;
    }

    public int getTextureId() {
        return textureId;
    }

    public void cleanup() {
        glDeleteFramebuffers(bufferId);

        glDeleteTextures(textureId);
    }
}
```

As you can see, is similar to the `GBuffer` class, but here we will only use a single texture to store the resulting colours. Now that we have created these new classes, we can start using them. In the `Renderer` class, we will no longer be using the forward rendering shaders we were using for rendering the scene (named `“scene_vertex.vs”` and `“scene_fragment.fs”`).

In the `init` method of the `Renderer` class you may see that a `GBuffer` instance is created and that we initialize and another set of shaders for the geometry pass (by calling the `setupGeometryShader` method) and the light pass (by calling the `setupDirLightShader` and `setupPointLightShader` methods). You may see also tha we create a instance of the class `SceneBuffer` named `sceneBuffer`. This will be used when rendering lights as explained before. An utility matrix named `bufferPassModelMatrix` is also instantiated (it will be used when performing the geometry pass). You can see that we create a new `Mesh` at the end of the init method. This will be used in the light pass. More on this will be explained later.

```
public void init(Window window) throws Exception {
    shadowRenderer.init(window);
    gBuffer = new GBuffer(window);
    sceneBuffer = new SceneBuffer(window);
    setupSkyBoxShader();
    setupParticlesShader();
    setupGeometryShader();
    setupDirLightShader();
    setupPointLightShader();
    setupFogShader();

    bufferPassModelMatrix =  new Matrix4f();
    bufferPassMesh = StaticMeshesLoader.load("src/main/resources/models/buffer_pass_me
ss.obj", "src/main/resources/models")[0];
}
```

The shaders used in the geometry and light passes are defined like usual (you can check the source code directly). Let's focus in their content instead. Let's focus in their content instead. We will start with the geomety pass, here's the vertex shader code (`gbuffer_vertex.vs`):

```
#version 330

const int MAX_WEIGHTS = 4;
const int MAX_JOINTS = 150;
const int NUM_CASCADES = 3;

layout (location=0) in vec3 position;
layout (location=1) in vec2 texCoord;
layout (location=2) in vec3 vertexNormal;
layout (location=3) in vec4 jointWeights;
layout (location=4) in ivec4 jointIndices;
layout (location=5) in mat4 modelInstancedMatrix;
layout (location=9) in vec2 texOffset;
layout (location=10) in float selectedInstanced;

uniform int isInstanced;
uniform mat4 viewMatrix;
uniform mat4 projectionMatrix;
```


www.dbooks.org

```
uniform mat4 modelNonInstancedMatrix;
uniform mat4 jointsMatrix[MAX_JOINTS];
uniform mat4 lightViewMatrix[NUM_CASCADES];
uniform mat4 orthoProjectionMatrix[NUM_CASCADES];
uniform int numCols;
uniform int numRows;
uniform float selectedNonInstanced;

out vec2  vs_textcoord;
out vec3  vs_normal;
out vec4  vs_mvVertexPos;
out vec4  vs_mlightviewVertexPos[NUM_CASCADES];
out mat4  vs_modelMatrix;
out float vs_selected;

void main()
{
    vec4 initPos = vec4(0, 0, 0, 0);
    vec4 initNormal = vec4(0, 0, 0, 0);
    mat4 modelMatrix;
    if ( isInstanced > 0 )
    {
        vs_selected = selectedInstanced;
        modelMatrix = modelInstancedMatrix;

        initPos = vec4(position, 1.0);
        initNormal = vec4(vertexNormal, 0.0);
    }
    else
    {
        vs_selected = selectedNonInstanced;
        modelMatrix = modelNonInstancedMatrix;

        int count = 0;
        for(int i = 0; i < MAX_WEIGHTS; i++)
        {
            float weight = jointWeights[i];
            if(weight > 0) {
                count++;
                int jointIndex = jointIndices[i];
                vec4 tmpPos = jointsMatrix[jointIndex] * vec4(position, 1.0);
                initPos += weight * tmpPos;

                vec4 tmpNormal = jointsMatrix[jointIndex] * vec4(vertexNormal, 0.0);
                initNormal += weight * tmpNormal;
            }
        }
        if (count == 0)
        {
            initPos = vec4(position, 1.0);
            initNormal = vec4(vertexNormal, 0.0);
        }
    }
```

```
    mat4 modelViewMatrix = viewMatrix * modelMatrix;
    vs_mvVertexPos = modelViewMatrix * initPos;
    gl_Position = projectionMatrix * vs_mvVertexPos;

    // Support for texture atlas, update texture coordinates
    float x = (texCoord.x / numCols + texOffset.x);
    float y = (texCoord.y / numRows + texOffset.y);

    vs_textcoord = vec2(x, y);
    vs_normal = normalize(modelViewMatrix * initNormal).xyz;

    for (int i = 0 ; i < NUM_CASCADES ; i++) {
        vs_mlightviewVertexPos[i] = orthoProjectionMatrix[i] * lightViewMatrix[i] * mo
delMatrix * initPos;
    }

    vs_modelMatrix = modelMatrix;
}
```

This shader is very similar to the vertex shader used in previous chapters to render a scene. There are some changes in the name of the output variables but in essence is the same shader. Indeed, it should be almost the same, the way we render the vertices should not change, the major changes are done in the fragment shader, which is defined like this (`gbuffer_fragment.fs`):

```
#version 330

const int NUM_CASCADES = 3;

in vec2  vs_textcoord;
in vec3  vs_normal;
in vec4  vs_mvVertexPos;
in vec4  vs_mlightviewVertexPos[NUM_CASCADES];
in mat4  vs_modelMatrix;
in float vs_selected;

layout (location = 0) out vec3 fs_worldpos;
layout (location = 1) out vec3 fs_diffuse;
layout (location = 2) out vec3 fs_specular;
layout (location = 3) out vec3 fs_normal;
layout (location = 4) out vec2 fs_shadow;

uniform mat4 viewMatrix;

struct Material
{
    vec4 diffuse;
    vec4 specular;
    int hasTexture;
    int hasNormalMap;
```


www.dbooks.org

```
    float reflectance;
};

uniform sampler2D texture_sampler;
uniform sampler2D normalMap;
uniform Material  material;

uniform sampler2D shadowMap_0;
uniform sampler2D shadowMap_1;
uniform sampler2D shadowMap_2;
uniform float cascadeFarPlanes[NUM_CASCADES];
uniform mat4 orthoProjectionMatrix[NUM_CASCADES];
uniform int renderShadow;

vec4 diffuseC;
vec4 speculrC;

void getColour(Material material, vec2 textCoord)
{
    if (material.hasTexture == 1)
    {
        diffuseC = texture(texture_sampler, textCoord);
        speculrC = diffuseC;
    }
    else
    {
        diffuseC = material.diffuse;
        speculrC = material.specular;
    }
}

vec3 calcNormal(Material material, vec3 normal, vec2 text_coord, mat4 modelMatrix)
{
    vec3 newNormal = normal;
    if ( material.hasNormalMap == 1 )
    {
        newNormal = texture(normalMap, text_coord).rgb;
        newNormal = normalize(newNormal * 2 - 1);
        newNormal = normalize(viewMatrix * modelMatrix * vec4(newNormal, 0.0)).xyz;
    }
    return newNormal;
}

float calcShadow(vec4 position, int idx)
{
    if ( renderShadow == 0 )
    {
        return 1.0;
    }

    vec3 projCoords = position.xyz;
    // Transform from screen coordinates to texture coordinates
    projCoords = projCoords * 0.5 + 0.5;
```

```
    float bias = 0.005;

    float shadowFactor = 0.0;
    vec2 inc;
    if (idx == 0)
    {
        inc = 1.0 / textureSize(shadowMap_0, 0);
    }
    else if (idx == 1)
    {
        inc = 1.0 / textureSize(shadowMap_1, 0);
    }
    else
    {
        inc = 1.0 / textureSize(shadowMap_2, 0);
    }
    for(int row = -1; row <= 1; ++row)
    {
        for(int col = -1; col <= 1; ++col)
        {
            float textDepth;
            if (idx == 0)
            {
                textDepth = texture(shadowMap_0, projCoords.xy + vec2(row, col) * inc)
.r;
            }
            else if (idx == 1)
            {
                textDepth = texture(shadowMap_1, projCoords.xy + vec2(row, col) * inc)
.r;
            }
            else
            {
                textDepth = texture(shadowMap_2, projCoords.xy + vec2(row, col) * inc)
.r;
            }
            shadowFactor += projCoords.z - bias > textDepth ? 1.0 : 0.0;
        }
    }
    shadowFactor /= 9.0;

    if(projCoords.z > 1.0)
    {
        shadowFactor = 1.0;
    }

    return 1 - shadowFactor;
}

void main()
{
    getColour(material, vs_textcoord);
```


www.dbooks.org

```
    fs_worldpos   = vs_mvVertexPos.xyz;
    fs_diffuse    = diffuseC.xyz;
    fs_specular   = speculrC.xyz;
    fs_normal     = normalize(calcNormal(material, vs_normal, vs_textcoord, vs_modelMa
trix));

    int idx;
    for (int i=0; i<NUM_CASCADES; i++)
    {
        if ( abs(vs_mvVertexPos.z) < cascadeFarPlanes[i] )
        {
            idx = i;
            break;
        }
    }
    fs_shadow  = vec2(calcShadow(vs_mlightviewVertexPos[idx], idx), material.reflectan
ce);

    if ( vs_selected > 0 ) {
        fs_diffuse = vec3(fs_diffuse.x, fs_diffuse.y, 1);
    }
}
```

The most relevant lines areThe most relevant lines are:

```
layout (location = 0) out vec3 fs_worldpos;
layout (location = 1) out vec3 fs_diffuse;
layout (location = 2) out vec3 fs_specular;
layout (location = 3) out vec3 fs_normal;
layout (location = 4) out vec2 fs_shadow;
```

This is where we are referring to the textures that this fragment shader will write to. As you can see we just dump the position (in light view coordinates), the diffuse colour (which can be the colour of the associated texture of a component of the material), the specular component, the normal, and the depth values for the shadow map.

SIDE NOTE: We have simplified the `Material` class definition removing the ambient colour component.

Going back to the `Renderer` class, the render method is defined like this:

```
public void render(Window window, Camera camera, Scene scene, boolean sceneChanged) {
    clear();

    if (window.getOptions().frustumCulling) {
        frustumFilter.updateFrustum(window.getProjectionMatrix(), camera.getViewMatrix
());
        frustumFilter.filter(scene.getGameMeshes());
        frustumFilter.filter(scene.getGameInstancedMeshes());
    }

    // Render depth map before view ports has been set up
    if (scene.isRenderShadows() && sceneChanged) {
        shadowRenderer.render(window, scene, camera, transformation, this);
    }

    glViewport(0, 0, window.getWidth(), window.getHeight());

    // Update projection matrix once per render cycle
    window.updateProjectionMatrix();

    renderGeometry(window, camera, scene);

    initLightRendering();
    renderPointLights(window, camera, scene);
    renderDirectionalLight(window, camera, scene);
    endLightRendering();

    renderFog(window, camera, scene);
    renderSkyBox(window, camera, scene);
    renderParticles(window, camera, scene);
}
```

The geometry pass is done in the `renderGeometry` method (you can see that we no longer have a `renderScene`). The lightning pass is done in several steps, first we setup the buffer and other parameters to be used (`initLightRendering`), then we render point lights (`renderPointLights`) and the directional light (`renderDirectionalLight`)and finally the state is restored (`endLightRendering`).

Let's start with the gemeotry pass. The `renderGeometry` method is almost equivalent to the `renderScene` method used in previous chapters:

www.dbooks.org

```
private void renderGeometry(Window window, Camera camera, Scene scene) {
    // Render G-Buffer for writing
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, gBuffer.getGBufferId());

    clear();

    glDisable(GL_BLEND);

    gBufferShaderProgram.bind();

    Matrix4f viewMatrix = camera.getViewMatrix();
    Matrix4f projectionMatrix = window.getProjectionMatrix();
    gBufferShaderProgram.setUniform("viewMatrix", viewMatrix);
    gBufferShaderProgram.setUniform("projectionMatrix", projectionMatrix);

    gBufferShaderProgram.setUniform("texture_sampler", 0);
    gBufferShaderProgram.setUniform("normalMap", 1);

    List<ShadowCascade> shadowCascades = shadowRenderer.getShadowCascades();
    for (int i = 0; i < ShadowRenderer.NUM_CASCADES; i++) {
        ShadowCascade shadowCascade = shadowCascades.get(i);
        gBufferShaderProgram.setUniform("orthoProjectionMatrix", shadowCascade.getOrth
oProjMatrix(), i);
        gBufferShaderProgram.setUniform("cascadeFarPlanes", ShadowRenderer.CASCADE_SPL
ITS[i], i);
        gBufferShaderProgram.setUniform("lightViewMatrix", shadowCascade.getLightViewM
atrix(), i);
    }
    shadowRenderer.bindTextures(GL_TEXTURE2);
    int start = 2;
    for (int i = 0; i < ShadowRenderer.NUM_CASCADES; i++) {
        gBufferShaderProgram.setUniform("shadowMap_" + i, start + i);
    }
    gBufferShaderProgram.setUniform("renderShadow", scene.isRenderShadows() ? 1 : 0);

    renderNonInstancedMeshes(scene);

    renderInstancedMeshes(scene, viewMatrix);

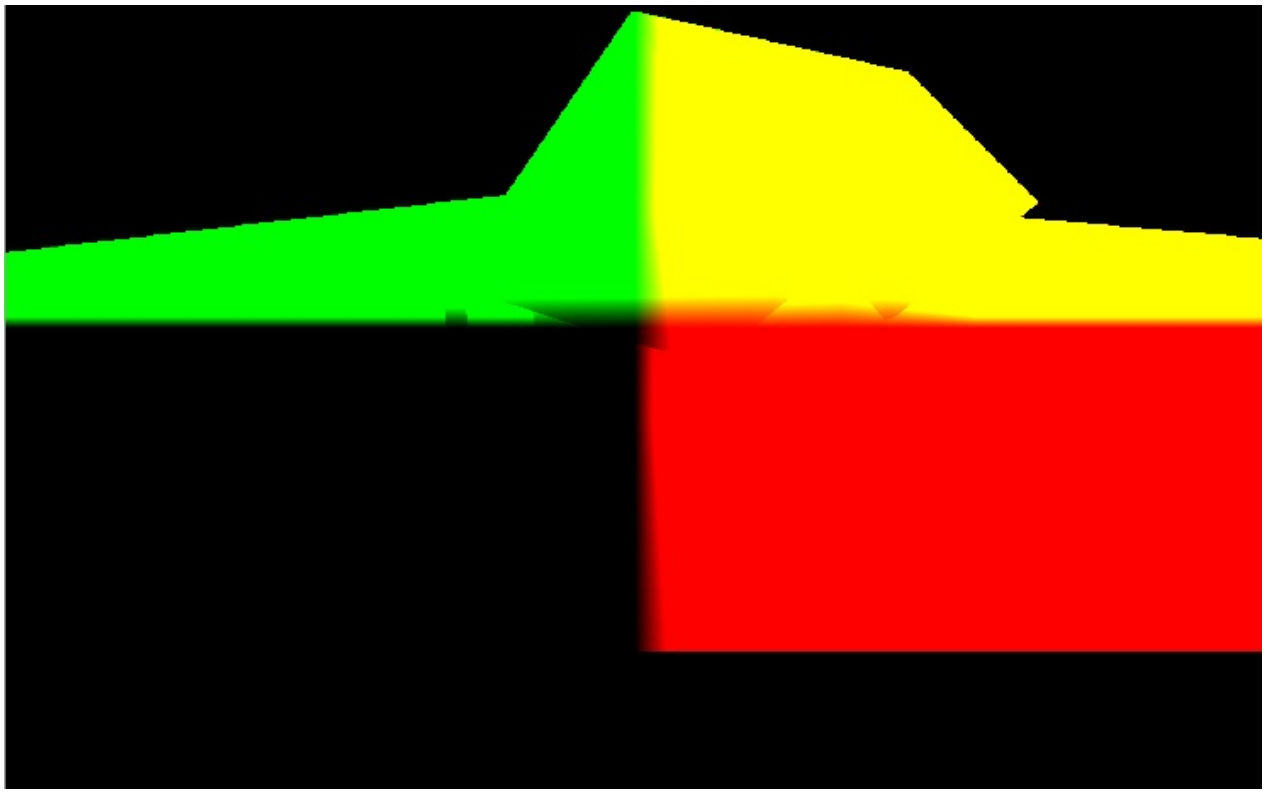
    gBufferShaderProgram.unbind();

    glEnable(GL_BLEND);
}
```

The only differences are:

- We bind to the G-Buffer instead of the screen.
- We disable blending. Since we just want to work with the values that are closest to the camera (the lowest depth values), we do not need blending.

If you debug the sample execution with an OpenGL debugger (such as RenderDoc), you can view the textures generated during the geometry pass. The positions texture will look like this:

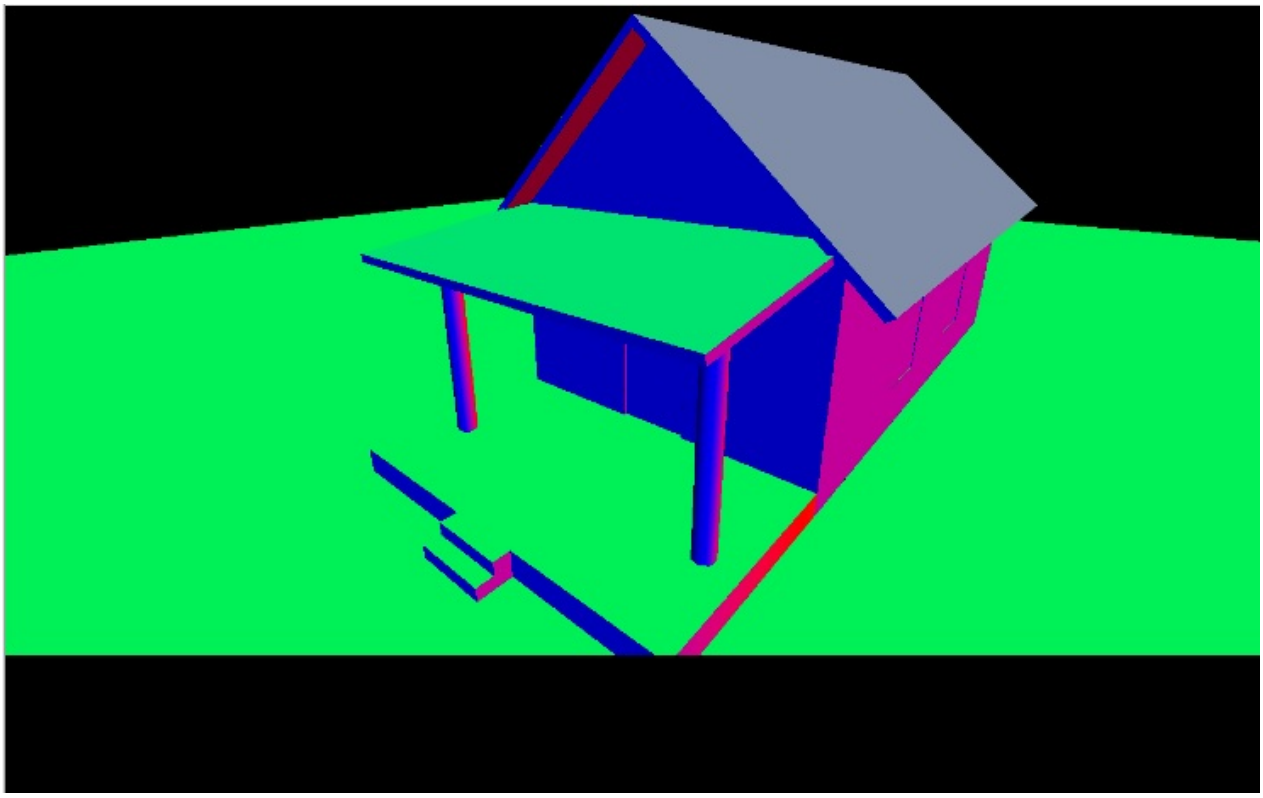


The texture that holds the values for the diffuse component will look like this:



The texture that holds the values for the normals will look like this:

www.dbooks.org



Now it’s the turn of the ligh pass. We first need to set up a few things before rendering, this is don in the `initLightRendering` method:

```
private void initLightRendering() {
    // Bind scene buffer
    glBindFramebuffer(GL_FRAMEBUFFER, sceneBuffer.getBufferId());

    // Clear G-Buffer
    clear();

    // Disable depth testing to allow the drawing of multiple layers with the same dep
th
    glDisable(GL_DEPTH_TEST);

    glEnable(GL_BLEND);
    glBlendEquation(GL_FUNC_ADD);
    glBlendFunc(GL_ONE, GL_ONE);

    // Bind GBuffer for reading
    glBindFramebuffer(GL_READ_FRAMEBUFFER, gBuffer.getGBufferId());
}
```

Since we won’t be rendering to the screen ,we need to first bind to the texture that will hold the results of the lightning pass. Then we clear that buffer and disable depth testing. This is not required any more, depth testing has been already done in the geometry pass. Another important step is to enable blending. The last action is to enable the G-Buffer for reading, it will be used during the light pass.

Before analyzing the render methods for the different types of light, let's think a little bit about how we will render the lights. We need to use the contents of the G-Buffer, but in order to use them, we need to first render something. But, we have already drawn the scene, what are we going to render. now? The answer is simple, we just need to render a quad that fills all the screen. For each fragment of that quad, we will use the data contained in the G-Buffer and generate the correct output colour. Do you remember the `Mesh` that we loaded in the init method of the `Renderer` class? It was named `bufferPassMesh`, and it just contains that, a quad that fills up the whole screen.

So, how the vertex shader for the light pass looks like?

```
#version 330

layout (location=0) in vec3 position;
uniform mat4 projectionMatrix;
uniform mat4 modelMatrix;

void main()
{
    gl_Position = projectionMatrix * modelMatrix * vec4(position, 1.0);
}
```

The code above is the vertex shader used when rendering point lights and directional light (`light_vertex.vs`). It just dumps the vertices using the model matrix and a projection matrix. There's no need to use a view matrix, we don't need a camera here.

The fragment shader for point lights (`point_light_fragment.fs`) is defined like this:

```
#version 330

out vec4 fragColor;

struct Attenuation
{
    float constant;
    float linear;
    float exponent;
};

struct PointLight
{
    vec3 colour;
    // Light position is assumed to be in view coordinates
    vec3 position;
    float intensity;
    Attenuation att;
};
```


www.dbooks.org

```
uniform sampler2D positionsText;
uniform sampler2D diffuseText;
uniform sampler2D specularText;
uniform sampler2D normalsText;
uniform sampler2D shadowText;
uniform sampler2D depthText;

uniform vec2 screenSize;

uniform float specularPower;
uniform PointLight pointLight;

vec2 getTextCoord()
{
    return gl_FragCoord.xy / screenSize;
}

vec4 calcLightColour(vec4 diffuseC, vec4 speculrC, float reflectance, vec3 light_colou
r, float light_intensity, vec3 position, vec3 to_light_dir, vec3 normal)
{
    vec4 diffuseColour = vec4(0, 0, 0, 1);
    vec4 specColour = vec4(0, 0, 0, 1);

    // Diffuse Light
    float diffuseFactor = max(dot(normal, to_light_dir), 0.0);
    diffuseColour = diffuseC * vec4(light_colour, 1.0) * light_intensity * diffuseFact
or;

    // Specular Light
    vec3 camera_direction = normalize(-position);
    vec3 from_light_dir = -to_light_dir;
    vec3 reflected_light = normalize(reflect(from_light_dir , normal));
    float specularFactor = max( dot(camera_direction, reflected_light), 0.0);
    specularFactor = pow(specularFactor, specularPower);
    specColour = speculrC * light_intensity  * specularFactor * reflectance * vec4(lig
ht_colour, 1.0);

    return (diffuseColour + specColour);
}

vec4 calcPointLight(vec4 diffuseC, vec4 speculrC, float reflectance, PointLight light,
 vec3 position, vec3 normal)
{
    vec3 light_direction = light.position - position;
    vec3 to_light_dir  = normalize(light_direction);
    vec4 light_colour = calcLightColour(diffuseC, speculrC, reflectance, light.colour,
 light.intensity, position, to_light_dir, normal);

    // Apply Attenuation
    float distance = length(light_direction);
    float attenuationInv = light.att.constant + light.att.linear * distance +
        light.att.exponent * distance * distance;
    return light_colour / attenuationInv;
```

```
}

void main()
{
    vec2 textCoord = getTextCoord();
    float depth = texture(depthText, textCoord).r;
    vec3 worldPos = texture(positionsText, textCoord).xyz;
    vec4 diffuseC = texture(diffuseText, textCoord);
    vec4 speculrC = texture(specularText, textCoord);
    vec3 normal  = texture(normalsText, textCoord).xyz;
    float shadowFactor = texture(shadowText, textCoord).r;
    float reflectance = texture(shadowText, textCoord).g;

    fragColor = calcPointLight(diffuseC, speculrC, reflectance, pointLight, worldPos.x
yz, normal.xyz) * shadowFactor;
}
```

As you can see it contains functions that sound familiar to you. They were used in previous chapters in the scene fragment shader. The important things here to note are the following lines:

```
uniform sampler2D positionsText;
uniform sampler2D diffuseText;
uniform sampler2D specularText;
uniform sampler2D normalsText;
uniform sampler2D shadowText;
uniform sampler2D depthText;
```

These uniforms model the different textures that compose the G-Buffer. We will use them to access the data. You may be asking now, how do we know which pixel to peek from those textures when we are rendering a fragment? The answer is by using the `gl_FragCoord` input variable. This variable contains the windows relative coordinates for the current fragment. To transform from that coordinates system to the textures one we use this function:

```
vec2 getTextCoord()
{
    return gl_FragCoord.xy / screenSize;
}
```

The fragment shader for the directional light is also quite similar, you can check the source code. Now that the shaders have been presented, let's go back to the `Renderer` class. For point lights we will do as many passes as lights are, we just bind the shaders used for this type of lights and draw the quad for each of them.

www.dbooks.org

```
private void renderPointLights(Window window, Camera camera, Scene scene) {
    pointLightShaderProgram.bind();

    Matrix4f viewMatrix = camera.getViewMatrix();
    Matrix4f projectionMatrix = window.getProjectionMatrix();
    pointLightShaderProgram.setUniform("modelMatrix", bufferPassModelMatrix);
    pointLightShaderProgram.setUniform("projectionMatrix", projectionMatrix);

    // Specular factor
    pointLightShaderProgram.setUniform("specularPower", specularPower);

    // Bind the G-Buffer textures
    int[] textureIds = this.gBuffer.getTextureIds();
    int numTextures = textureIds != null ? textureIds.length : 0;
    for (int i=0; i<numTextures; i++) {
        glActiveTexture(GL_TEXTURE0 + i);
        glBindTexture(GL_TEXTURE_2D, textureIds[i]);
    }

    pointLightShaderProgram.setUniform("positionsText", 0);
    pointLightShaderProgram.setUniform("diffuseText", 1);
    pointLightShaderProgram.setUniform("specularText", 2);
    pointLightShaderProgram.setUniform("normalsText", 3);
    pointLightShaderProgram.setUniform("shadowText", 4);

    pointLightShaderProgram.setUniform("screenSize", (float) gBuffer.getWidth(), (float
)gBuffer.getHeight());

    SceneLight sceneLight = scene.getSceneLight();
    PointLight[] pointLights = sceneLight.getPointLightList();
    int numPointLights = pointLights != null ? pointLights.length : 0;
    for(int i=0; i<numPointLights; i++) {
        // Get a copy of the point light object and transform its position to view coo
rdinates
        PointLight currPointLight = new PointLight(pointLights[i]);
        Vector3f lightPos = currPointLight.getPosition();
        tmpVec.set(lightPos, 1);
        tmpVec.mul(viewMatrix);
        lightPos.x = tmpVec.x;
        lightPos.y = tmpVec.y;
        lightPos.z = tmpVec.z;
        pointLightShaderProgram.setUniform("pointLight", currPointLight);

        bufferPassMesh.render();
    }

    pointLightShaderProgram.unbind();
}
```

The approach is quite similar for directional light. In this case, we just use do one pass:

```
private void renderDirectionalLight(Window window, Camera camera, Scene scene) {
    dirLightShaderProgram.bind();

    Matrix4f viewMatrix = camera.getViewMatrix();
    Matrix4f projectionMatrix = window.getProjectionMatrix();
    dirLightShaderProgram.setUniform("modelMatrix", bufferPassModelMatrix);
    dirLightShaderProgram.setUniform("projectionMatrix", projectionMatrix);

    // Specular factor
    dirLightShaderProgram.setUniform("specularPower", specularPower);

    // Bind the G-Buffer textures
    int[] textureIds = this.gBuffer.getTextureIds();
    int numTextures = textureIds != null ? textureIds.length : 0;
    for (int i=0; i<numTextures; i++) {
        glActiveTexture(GL_TEXTURE0 + i);
        glBindTexture(GL_TEXTURE_2D, textureIds[i]);
    }

    dirLightShaderProgram.setUniform("positionsText", 0);
    dirLightShaderProgram.setUniform("diffuseText", 1);
    dirLightShaderProgram.setUniform("specularText", 2);
    dirLightShaderProgram.setUniform("normalsText", 3);
    dirLightShaderProgram.setUniform("shadowText", 4);

    dirLightShaderProgram.setUniform("screenSize", (float) gBuffer.getWidth(), (float)
gBuffer.getHeight());

    // Ambient light
    SceneLight sceneLight = scene.getSceneLight();
    dirLightShaderProgram.setUniform("ambientLight", sceneLight.getAmbientLight());

    // Directional light
    // Get a copy of the directional light object and transform its position to view c
oordinates
    DirectionalLight currDirLight = new DirectionalLight(sceneLight.getDirectionalLigh
t());
    tmpVec.set(currDirLight.getDirection(), 0);
    tmpVec.mul(viewMatrix);
    currDirLight.setDirection(new Vector3f(tmpVec.x, tmpVec.y, tmpVec.z));
    dirLightShaderProgram.setUniform("directionalLight", currDirLight);

    bufferPassMesh.render();

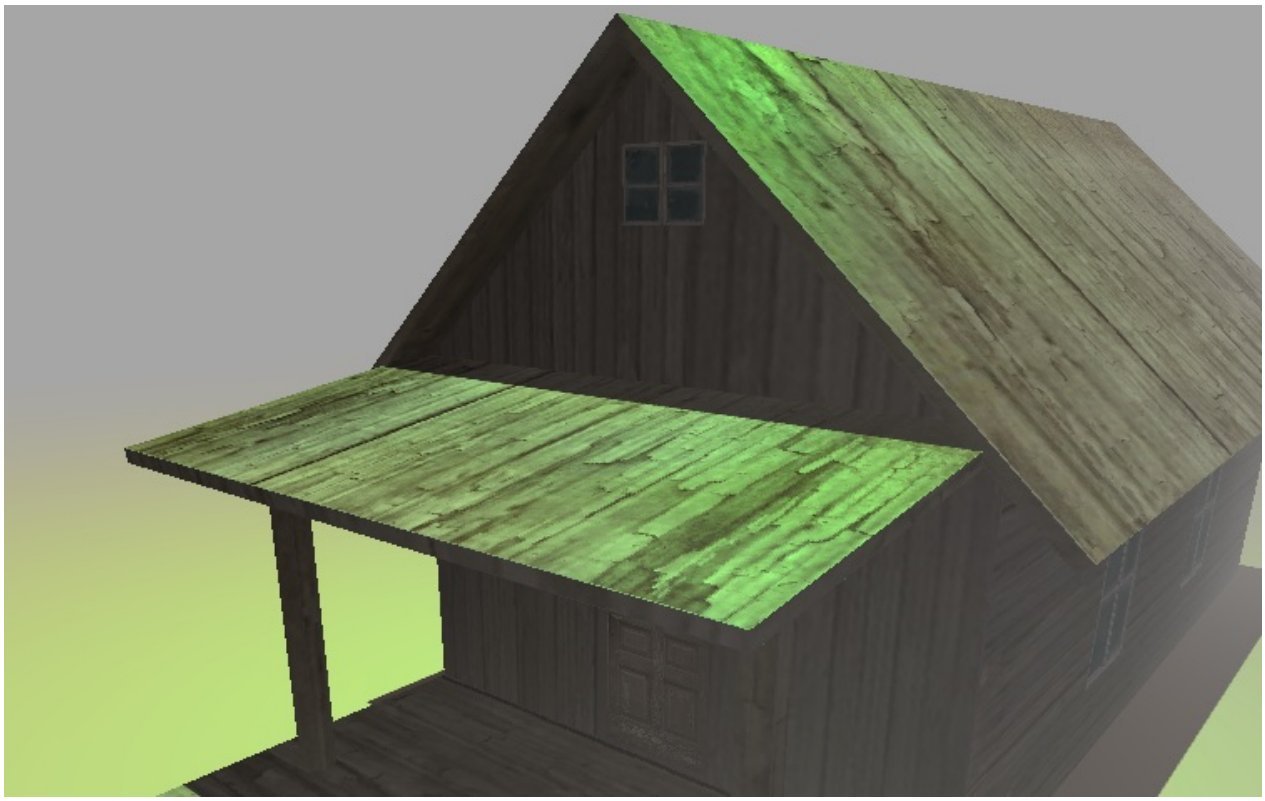
    dirLightShaderProgram.unbind();
}
```

The `endLightRendering` simply retsores the state.

www.dbooks.org

```
private void endLightRendering() {
    // Bind screen for writing
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glEnable(GL_DEPTH_TEST);
    glDisable(GL_BLEND);
}
```

If you execute the sample you will see something like this:



The chapter got longer that expected but there are a few key points that need to be clarified:

- Spot lights have been removed in order to simplify this chapter.
- A common technique used in deferred shading, for point lights, is just to calculate the area of the scene affected by that light. In this case, instead of rendering a quad that fills up the screen, you can use a smaller quad, a sphere, etc. Keep in mind that the best is enemy of the good. Performing complex calculus to determine the smallest shape required may be slower than using other coarse approaches.
- If you do not have many lights, this method will be slower than forward shading.

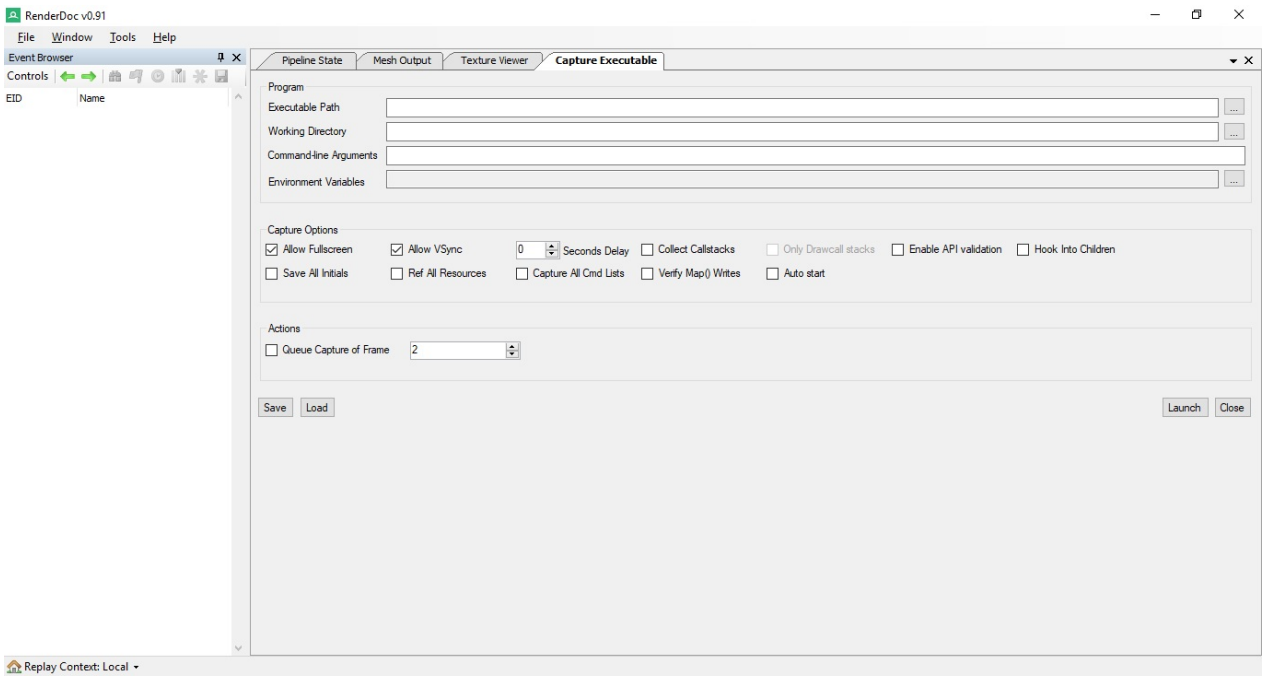
As a final note, if you want to see how these techniques are used in real world games, you can check this superb explanation about how a GTA V frame gets rendered.

Appendix A - OpenGL Debugging

Debugging an OpenGL program can be a daunting task. Most of the times you end up with a black screen and you have no means of knowing what's going on. In order to alleviate this problem we can use some existing tools that will provide more information about the rendering process.

In this annex we will describe how to use the RenderDoc tool to debug our LWJLG programs. RenderDoc is a graphics debugging tool that can be used with DIrect3D, Vulkan an OpenGL. In the case of OpenGL it only supports the core profile from 3.2 up to 4.5.

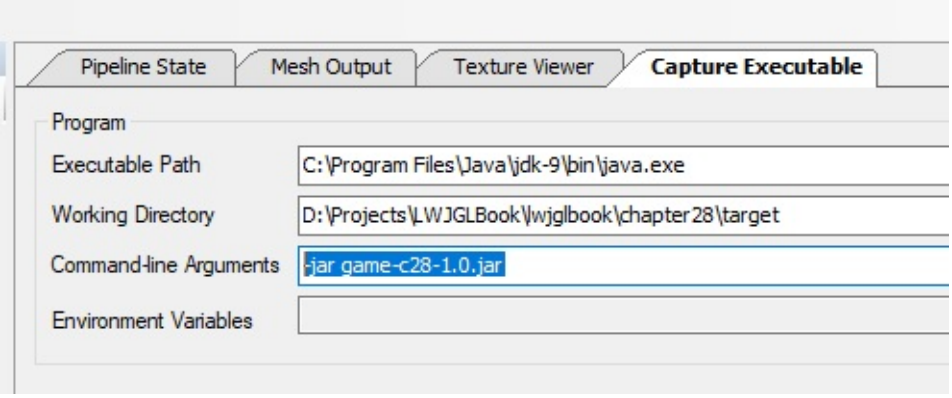
So let's get started. You need to downlaod and install the RenderDoc version for your OS. Once installed, when you launch it you will see something similar to this.



The first step is to configure RenderDoc to execute and monitor our samples. In the "Capture Executable" tab we need to setup the following parameters:

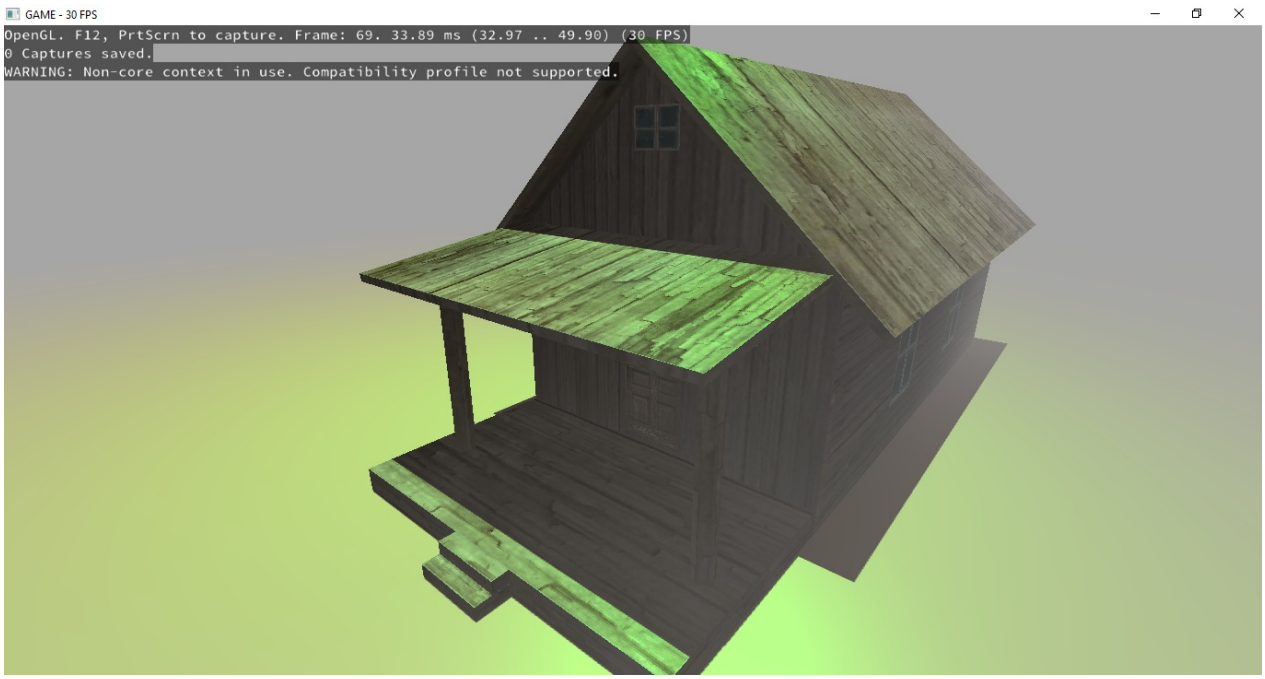
- **Executable path**: In our case this should point to the JVM launcher (For instance, "C:\Program Files\Java\jdk-9\bin\java.exe").
- **Working Directory**: This is the working directory that will be setup for your program. In our case it should be set to the target directory where maven dumps the result. By setting this way, the dependencies will be able to be found.
- **Command line arguments**: This will contain the arguments required by the JVM to execute our sample. In our case, just passing the jar to be executed (For instance, "-jar game-c28-1.0.jar").

www.dbooks.org

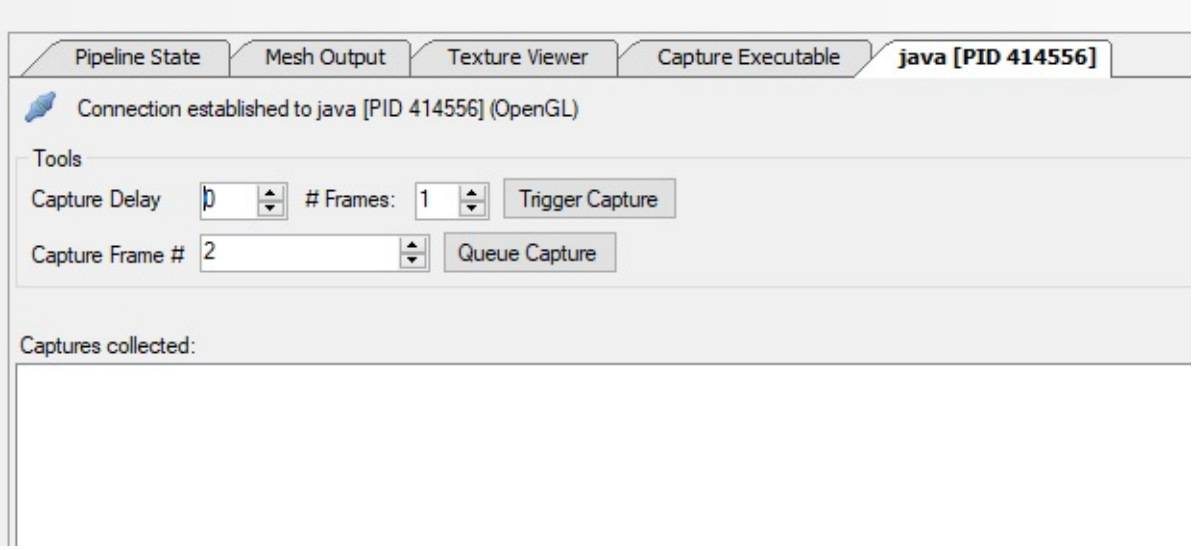


You must remember that 3D models are loaded now with Assimp, and we need real file path (no more `CLASSPATH` related paths), so you need to check the routes from the working directory you have set up. In this case, the easiest approach to quickly test is to copy the src folder into the target directory.

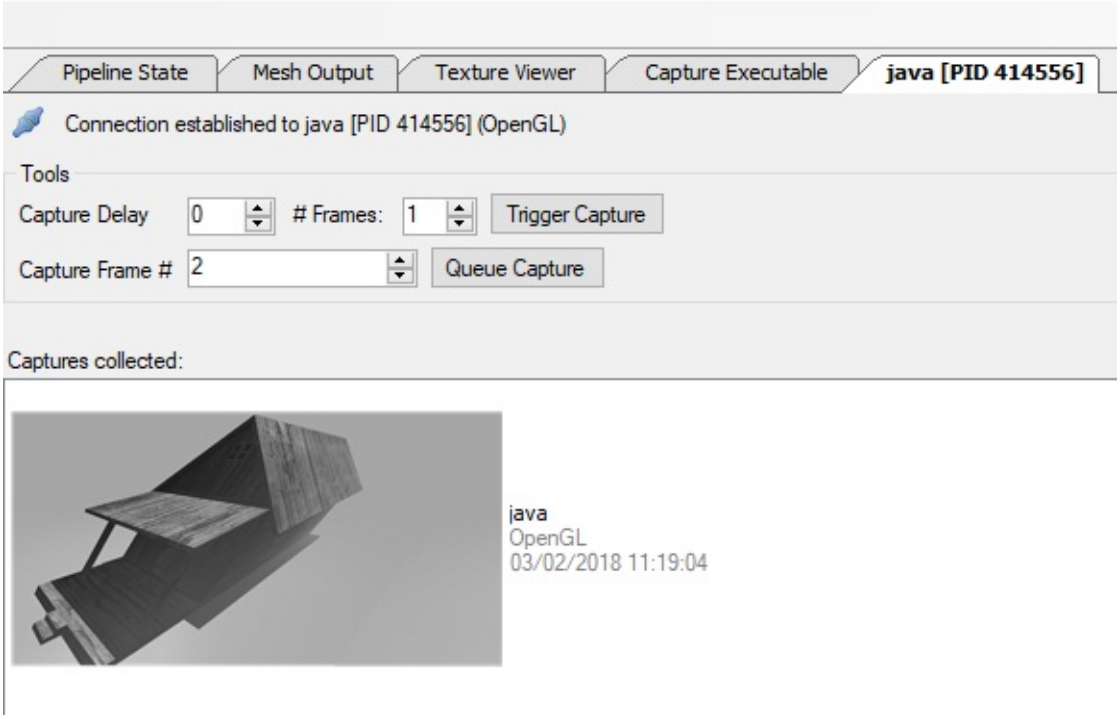
There are many other options int this tab to configure the capture options. You can consult their purpose in RenderDoc documentation. Once everything has been setup you can execute your program by clicking on the “Launch” button. You will see something like this:



You may see a Warning since RenderDoc can only work with OpenGL core profile. In the sample we’ve enabled compatibiñity profile, but it should work even with that warning. Once the program is being executed you can triigger snapshots of it. You will see that a new tab has been added which is named “java [PID XXXX]” (where the XXXX number represents the PID, the process identifier, of the java process).

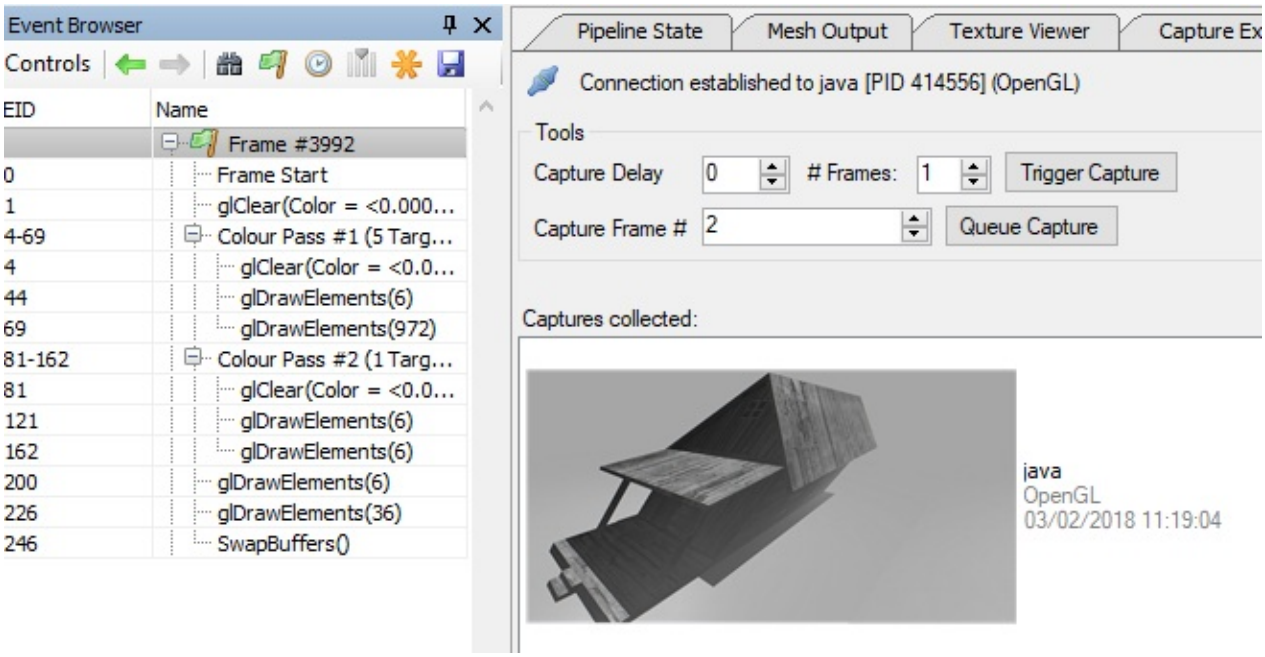


From that tab you can capture the state of your program by pressing the “Trigger capture” button. Once a capture has been generated, you will see a little snapshot in that same tab.



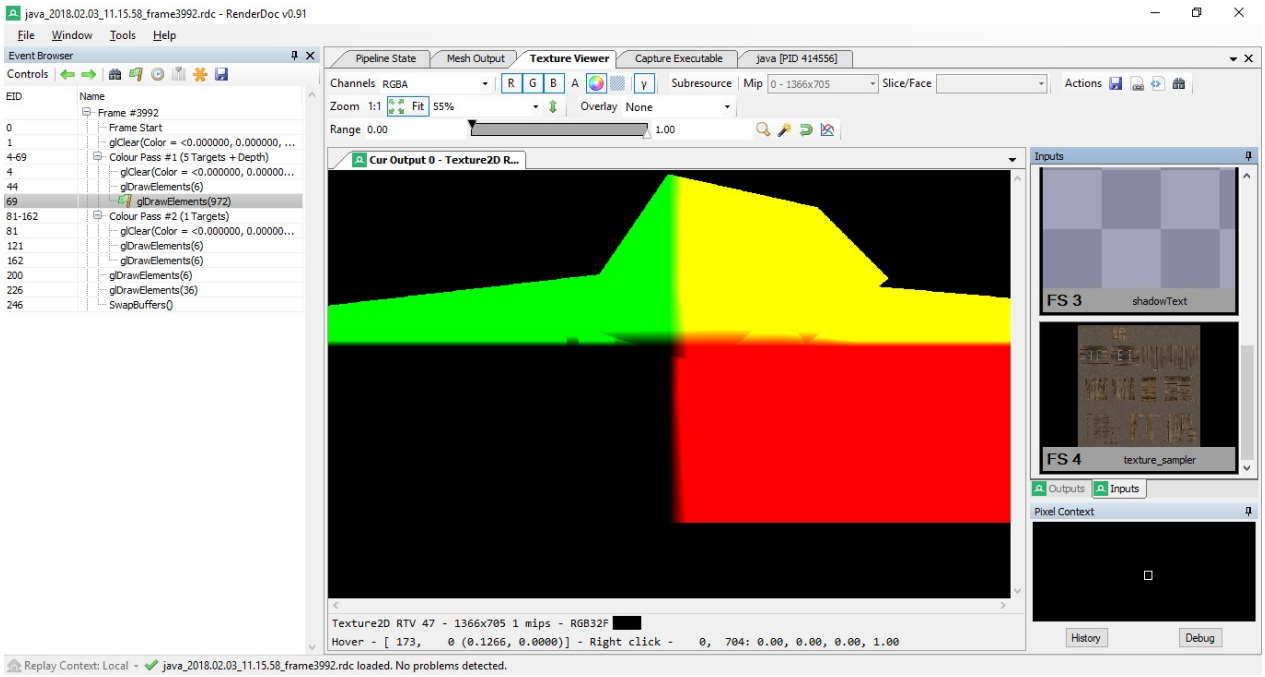
If you double click on that capture, all the data collected will be loaded and you can start inspecting it. The “Event Browser” panel will be populated will all the relevant OpenGL calls executed during one rendering cycle.

www.dbooks.org

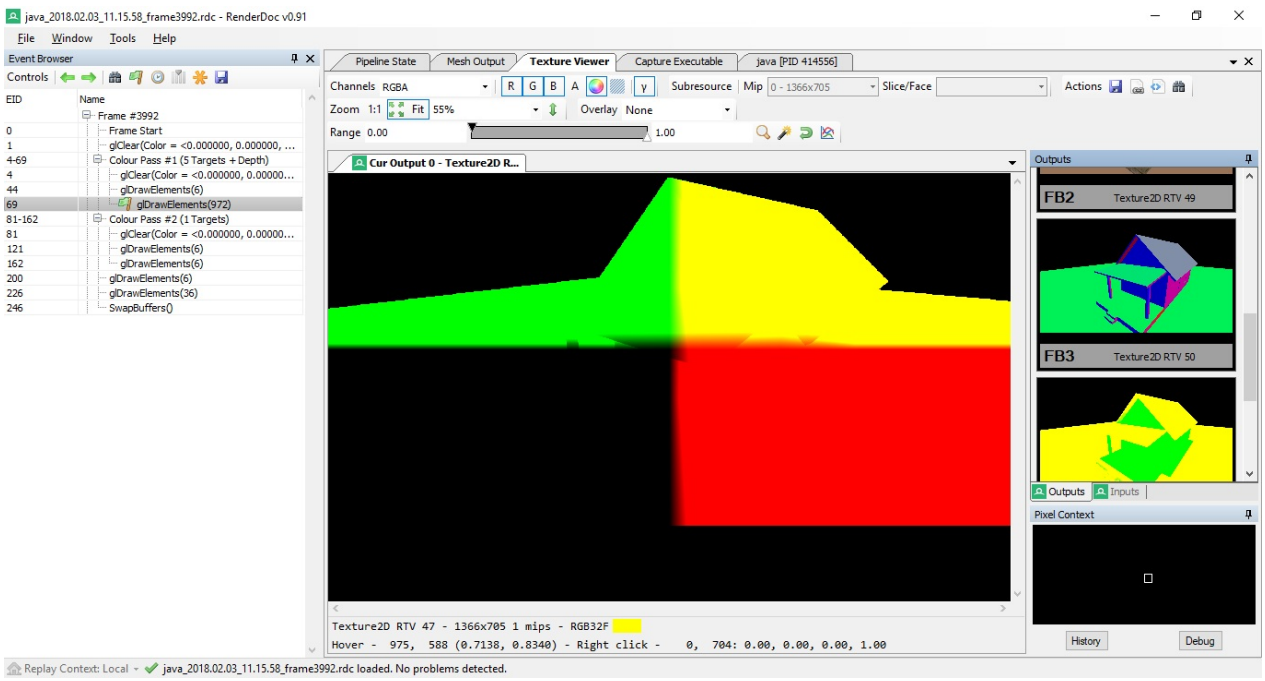


You can see, for the first rendering pass, how the floor is drawn and later on the mesh that models the house. If you click over a glDrawELements event, and select the “Mesh” tab you can event see the Mesh that was drawin, was the input for the vertex shader and its output.

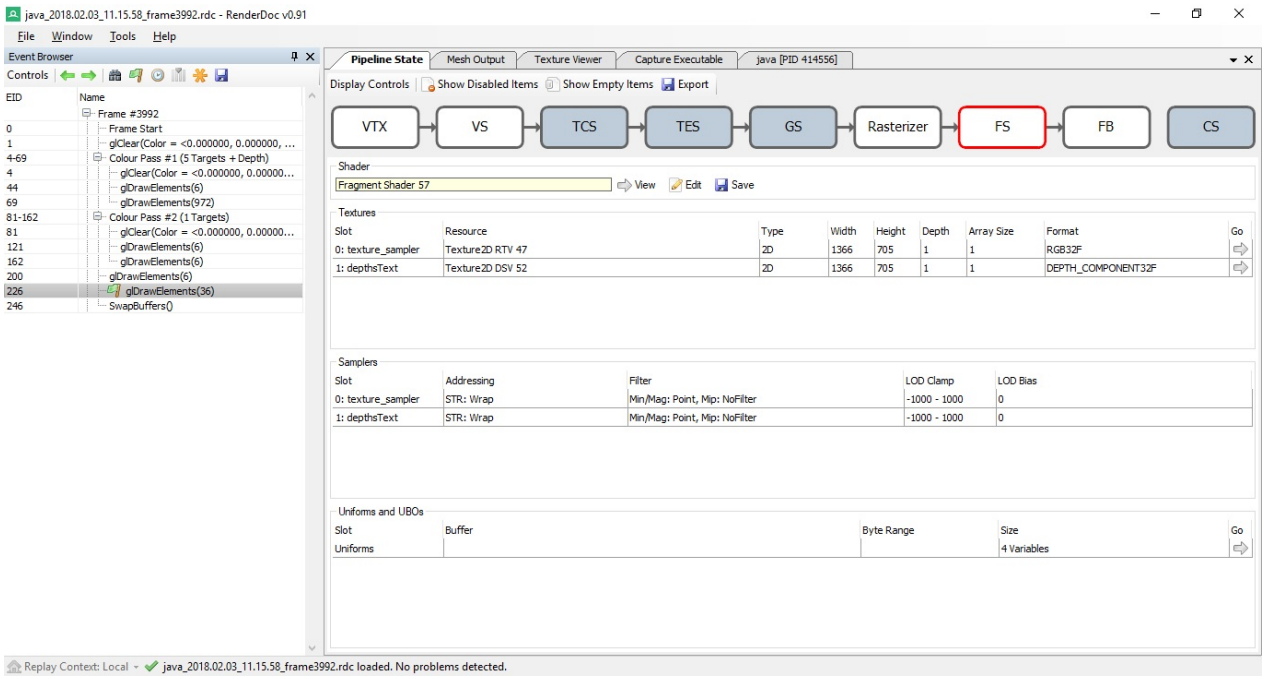
You can also view the input textures used for that drawing operation (by clicking the “Texture Viewer Tab”).



In the center panel, you can see the output, and on the right panel you can see the list of textures used as an input. You can also view the output textures one by one. This is very illustrative to show how deferred shading works.



As you can see, this tool provides valuable information about what's happening when rendering. It can save precious time while debugging rendering problems. It can even display information about the shaders used in the rendering pipeline.



www.dbooks.org