3D Game Development with LWJGL 3

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

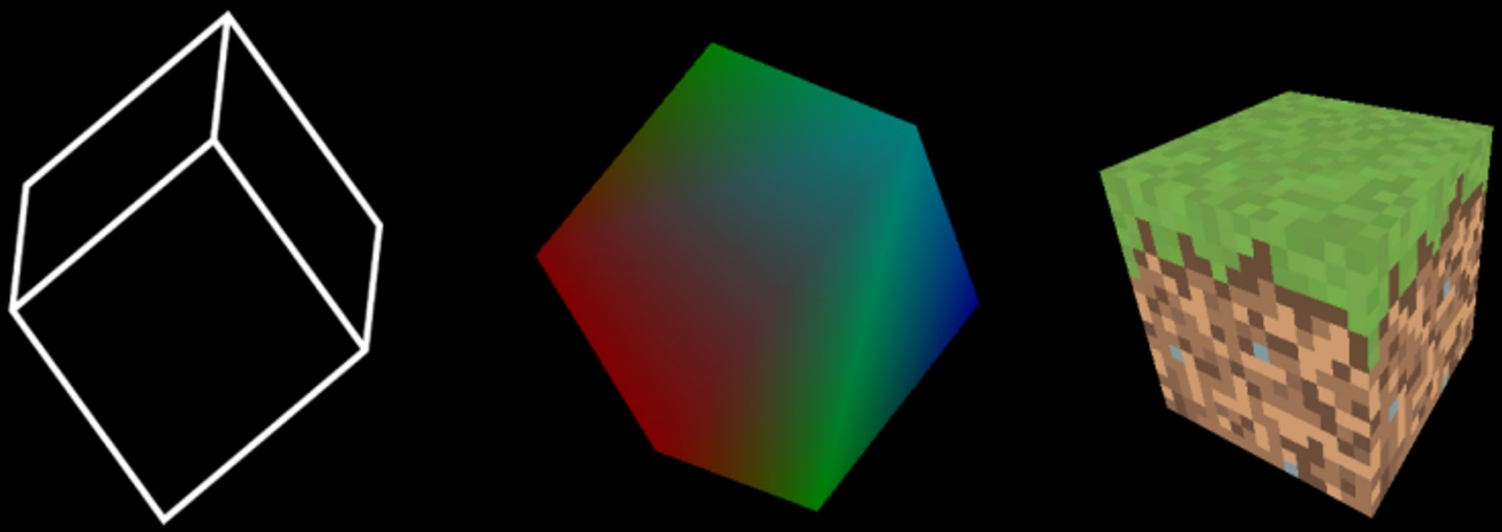


Table of Contents

3D Game Development with LWJGL 3

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 this book will be published in GitHub.

License

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 it here.

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 and make the corrections you consider in order to improve the book.

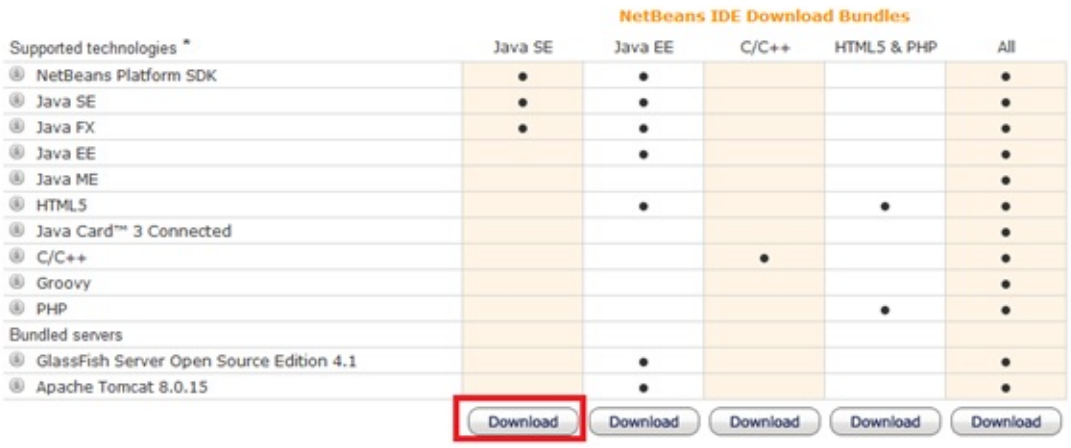
First steps

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 Java Lightweight Game Library (LWJGL). The LWJGL library enables the access to low-level APIs (Application Programming Interface) such as OpenGL.

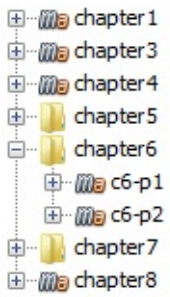
LWJGL is a low level API that acts like a wrapper around OpenGL. If your idea is to start creating 3D games in a short period of time maybe you should consider other alternatives like [JmonkeyEngine]. By using this low level API you will have to go through many concepts and create mane lines of code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

As said in the previous paragraphs we will be using Java for this book. We will be using Java 8, so you need to download Java SDK from Oracle's pages. Just choose the installer that suits your Operative System and install it. This book assumes that you have a moderate understanding of the Java language.

The source code that accompanies this book has been developed using the Netbeans IDE. You can download the latest version of that IDE from https://netbeans.org/. In order to execute Netbeans you only need the Java SE version but remember to download the version that suits with your JDK version (32 bits or 64 bits).



For building our samples we will be using Maven. Maven is already integrated in Netbeans and you can directly open the different samples from Netbeans, just open the folder that contains the chapter sample and Netbeans will detect that it is a maven project.



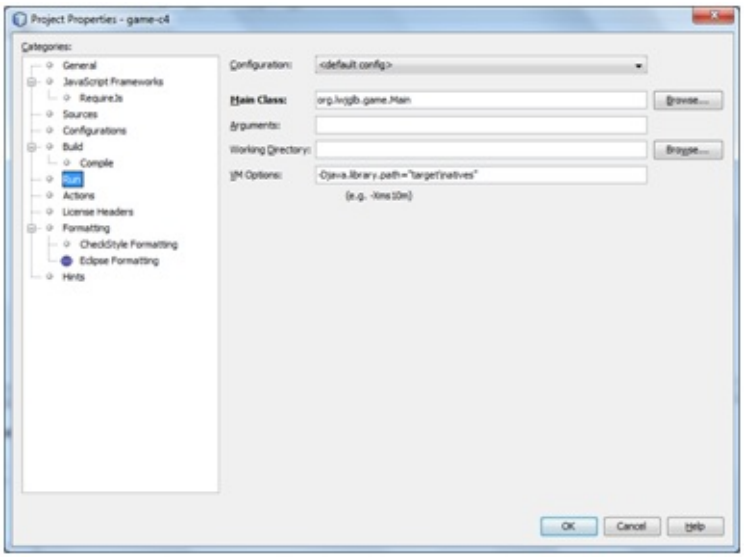
Maven builds projects based on an XML file named `pom.xml` (Project Object Model) which manages project dependencies (the libraries you need to use) and the steps to be performed during the build process. Maven follows the principle of convention over configuration, that is, if you stick to the standard 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.

We use a special plugin named `mavennatives` which unpacks the native libraries provided by LWJGL for your platform.

```
<plugin>
    <groupId>com.googlecode.mavennatives</groupId>
    <artifactId>maven-nativedependencies-plugin</artifactId>
    <version>${natives.version}</version>
    <executions>
        <execution>
            <id>unpacknatives</id>
            <phase>generate-resources</phase>
            <goals>
                <goal>copy</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Those libraries are placed under `target/natives` directory. When you execute the samples from Netbeans you need to specify the directory where the Java Virtual Machine will look for native libraries. This is done with the command line property: `“-Djava.library.path”` which sould be set to: `“-Djava.library.path="target\natives”` . This is done automatically for you in the `nbactions.xml` file. In case you want to change it or learn how to do it manually, right click in your project and select “Properties”. In the dialog that is shown select “Run” category and set the correct value for VM Options.

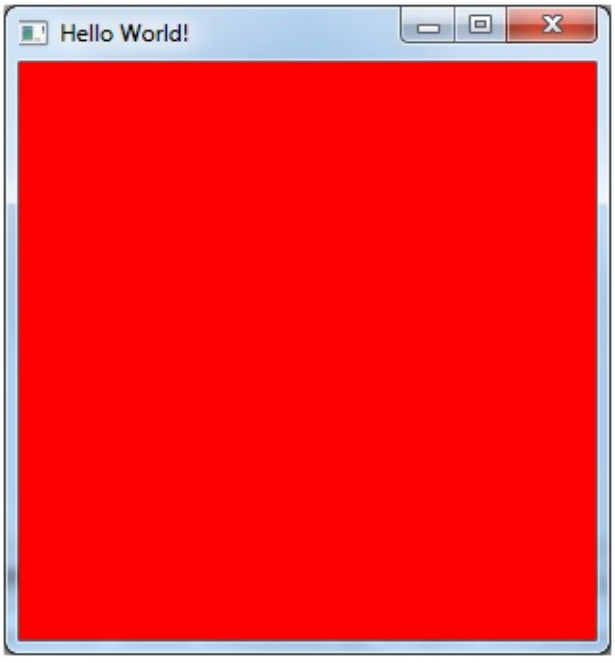


The source codee of this book is published in GitHub.

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 Open GL Context attached in a straight forward 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 straight forward so we won't repeat the comments here. **NOTE**: If you have been using alpha LWJGL 3 versions, you must be aware that some parts of the API have changed.

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



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 and endless loop which is responsible of periodically handling user input, updating game state and render to the screen.

The following snippet shows the structure of a game loop:

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

So, is that all? Have we finished with game loops? Well, not yet. The above snippet has many pitfalls. First of all the speed that the game loop runs will execute at different speeds 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 run 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 / 50;

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 equals to 20 ms.).

Besides that, our computer may be prioritizing another tasks that prevent our game loop to execute for 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 are no time. So, as you see the problem is not so simple.

In 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 her is usually called as Fixed Step Game Loop.

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

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

```
double secsPerUpdate = 1 / 30;
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 computer resources by rendering continuously? This is done in the sync method:

```
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 time taking into consideration the time we have 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 endTi**me).
2. Compare current time with that end time and wait just one second 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. 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.

```
// Setup resize callback
glfwSetWindowSizeCallback(windowHandle, windowSizeCallback = new GLFWWindowSizeCallback()
    @Override
    public void invoke(long window, int width, int height) {
        Window.this.width = width;
        Window.this.height = height;
        Window.this.setResized(true);
    }
});
```

We will also create a `Renderer` class which will do 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
}
```

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, IMageLogoc gameLogic) th
        gameLoopThread = new Thread(this, "GAME_LOOP_THREAD");
        window = new Window(windowTitle, width, height);
        this.gameLogic = gameLogic;
        //..[Removed code]..
    }
```

As 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 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 code 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 have 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 an 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.

```
public class Main {

    public static void main(String[] args) {
        try {
            IGameLogic gameLogic = new DummyGame();
            GameEngine gameEng = new GameEngine("GAME",
                600, 480, 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 update the 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 need to be ware if the window has been resized and update the view port to locate the center of the coordinates in the center of the window.

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 necessary at this moment we need to isolate generic tasks that every game will use from the state logic, artwork and resources of an 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.

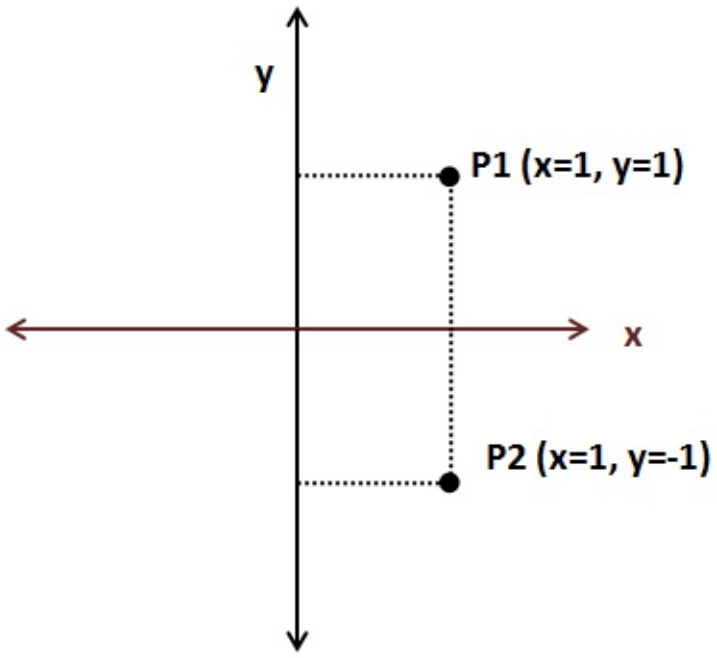
A brief about coordinates

In this chapter we will talk a little bit about coordinates and coordinates 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 in 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).

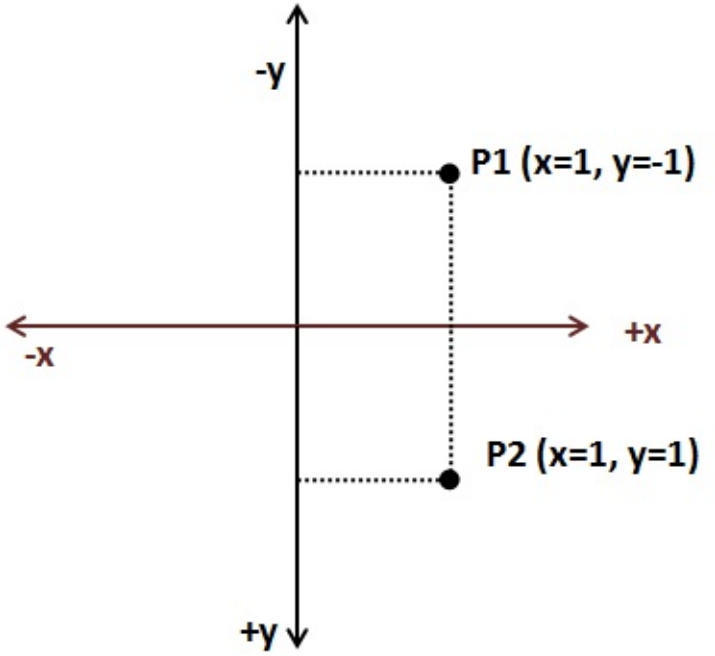
A coordinate system is a system which employs one or more numbers, that is, one or more coordinates to uniquely specify the position of a point. There are different coordinate systems (Cartesian, polar, etc.) and you can transform coordinates from one system to another. We will use the Cartesian coordinate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

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



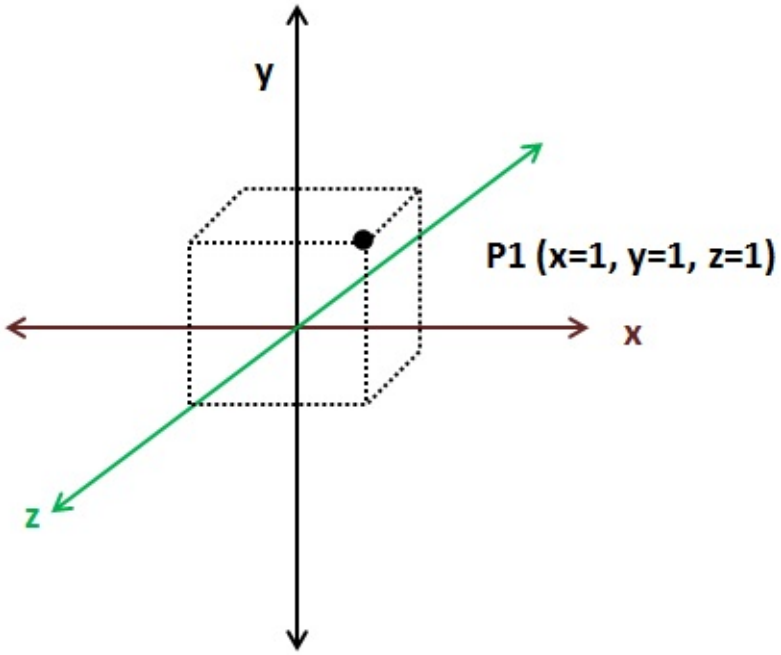
Continuing with the map analogy, coordinate systems define an origin. For geographic coordinates the origin is set in the point where the equator and the zero meridian cross. Depending on where we set the origin coordinates for a specific point are different. A coordinate system may also define the orientation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

coordinates increase as long as we move to the right and y coordinates increase 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.

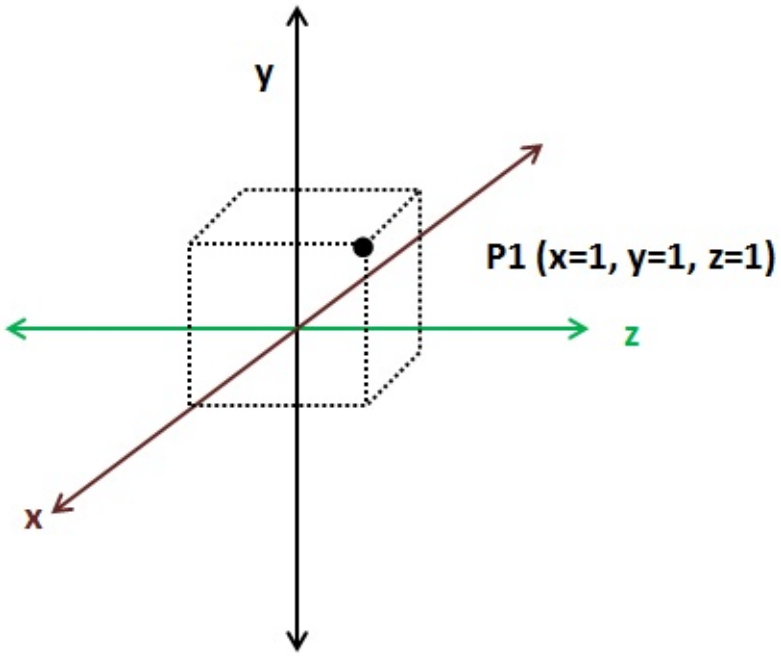


As you can see we need to define some arbitrary parameters, such as the origin and the axis orientation in order to give the appropriate meaning to the pair of numbers that constitute a coordinate. We will refer to that coordinate system with the set of arbitrary parameters as the coordinate space. In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

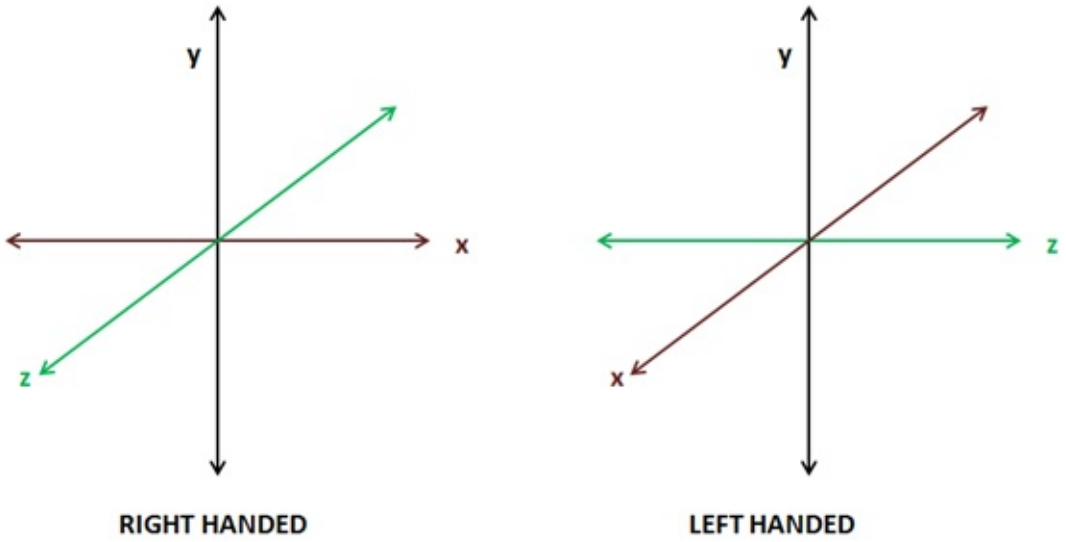
If we are dealing with 3D coordinates we need an additional axis, the z axis. 3D coordinates will be formed by a set of three numbers (x, y, z).



As in 2D Cartesian coordinate spaces we can change the orientation of the axis in 3D coordinate spaces as long as the axis 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.

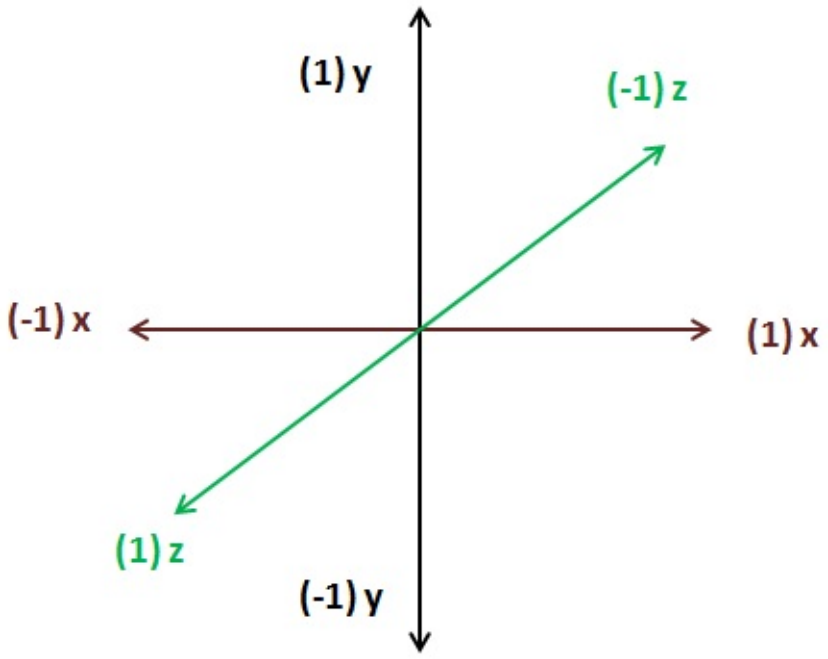
Now that we have define some basic topics let's talk about some common used terms that are used when dealing with 3D graphics. When we explain in later chapters how to render 3D models we will see that we use different 3D coordinate spaces, that is because each of those coordinate spaces have a context, a purpose. A set of coordinates are meaningless unless they are referred 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 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, so they can only the define the coordinates using a coordinate space that is only relevant for the model.

When we will be drawing a 3D scene we will refer all of our 3D objects to the so called world space coordinate space. We will need to transform from 3D object coordinate spaces coordinates 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.

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



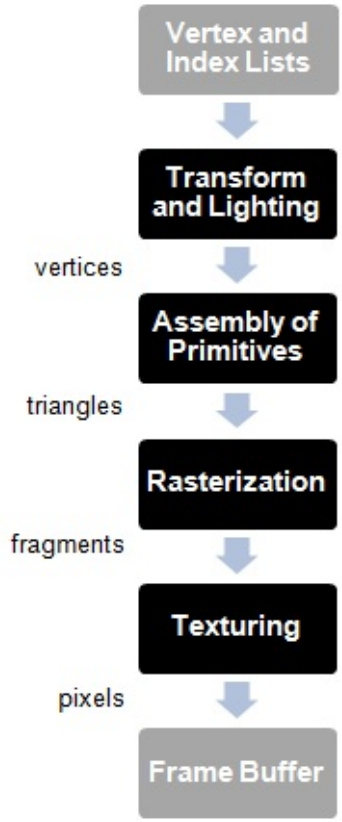
Don’t worry if you don’t have clear all the concepts, they will be revisited during next chapters with practical examples.

Rendering

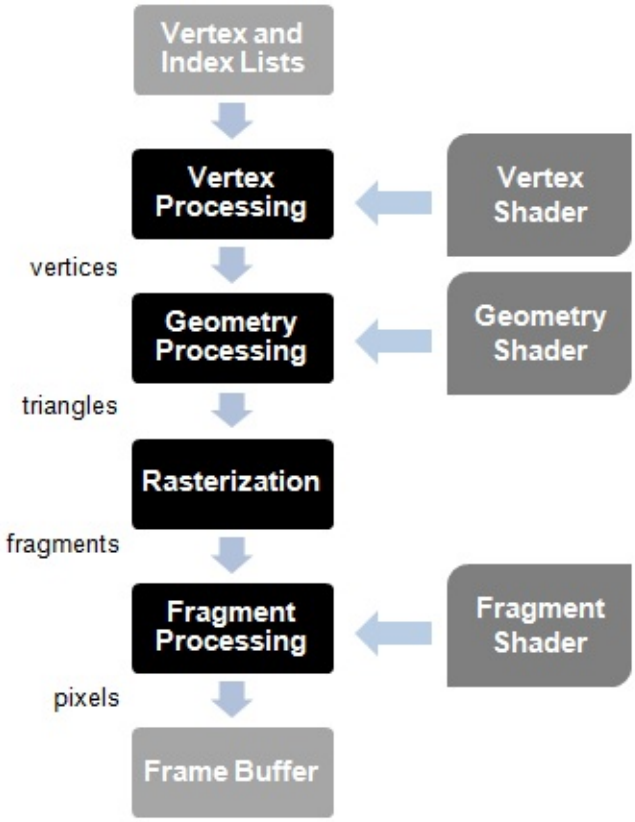
In this chapter we will learn the processes that takes place while rendering a scene using OpenGL. If you are used to older versions of OpenGL, that is fixed-function pipeline, you may end this chapter wondering why it needs to be so complex. You may end up thinking that drawing a simple shape to the screen should not require so many concepts and line of codes. 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 by these steps:



Open GL 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 it's coordinates x, y and z. 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 which 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 it's 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? Triangles 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 that get 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 uss 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 pos;

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).

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 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 by 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 f the operations we need to do are based on vector 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 lineal transformation, 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 about 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 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 shader code files.
3. For each shader, create a new shader program and specify its type (vertex, shader).
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 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");
        }
    }
```

```
    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. Code: " + shaderId);
        }

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

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

        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: " + glGetShaderInfoLog(progra
        }

        glValidateProgram(programId);
        if (glGetProgrami(programId, GL_VALIDATE_STATUS) == 0) {
            throw new Exception("Error validating Shader code: " + glGetShaderInfoLog(pro
        }

    }

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

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

    public void cleanup() {
        unbind();
        if (programId != 0) {
            if (vertexShaderId != 0) {
                glDetachShader(programId, vertexShaderId);
            }
```

```
            if (fragmentShaderId != 0) {
                glDetachShader(programId, fragmentShaderId);
            }
            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. `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:

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

Now we can use or 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:

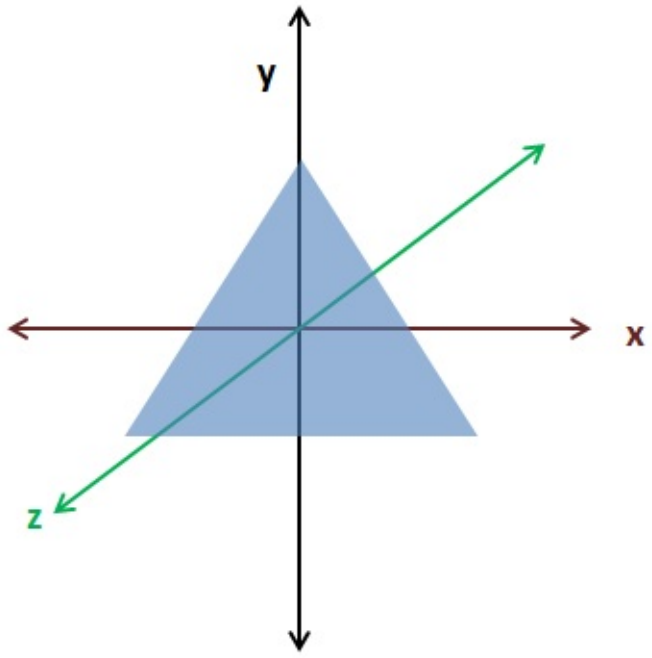
```
public void init() throws Exception {
    shaderProgram = new ShaderProgram();
    shaderProgram.createVertexShader(Utils.loadResource("/vertex.vs"));
    shaderProgram.createFragmentShader(Utils.loadResource("/fragment.fs"));
    shaderProgram.link();
}
```

We have created an 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 have 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). A VAO 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 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 with is to store our array of floats into a `FloatBuffer`. This is mainly due to the fact that we must interface with OpenGL library, which is C-bases, so we must transform our array of floats into something that can be managed by the library.

```
FloatBuffer verticesBuffer =
    BufferUtils.createFloatBuffer(vertices.length);
verticesBuffer.put(vertices).flip();
```

We use a utility class to create the buffer and 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 on it).

Now we need to create the VAO and bind to it.

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

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

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

Now it comes the most important part, we need to define the structure of our data and store 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 then 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 a offset of the first component of the first component in the array in the data store of the buffer.

After we have 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);
```

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

```
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 have 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 patience and continue reading.

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 org.lwjgl.BufferUtils;
import static org.lwjgl.opengl.GL11.*;
import static org.lwjgl.opengl.GL15.*;
import static org.lwjgl.opengl.GL20.*;
import static org.lwjgl.opengl.GL30.*;

public class Mesh {

    private final int vaoId;

    private final int vboId;

    private final int vertexCount;

    public Mesh(float[] positions) {
        vertexCount = positions.length / 3;
        FloatBuffer verticesBuffer = BufferUtils.createFloatBuffer(positions.length);
        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);
    }

    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);
    }
}
```

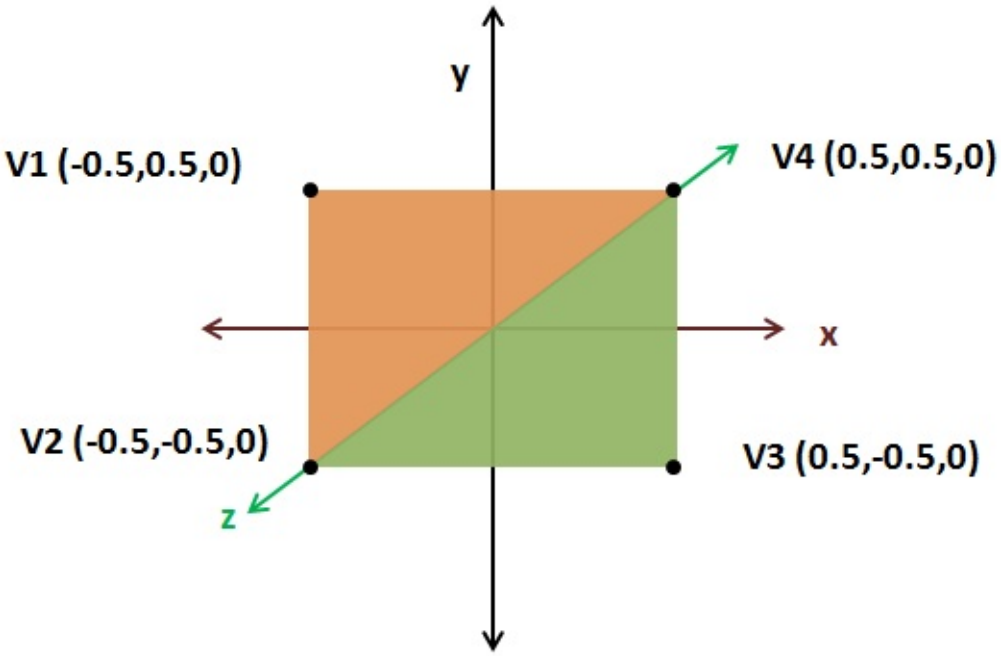
We will create our `Mesh` instance in our `DummyGame` class, removing the VAO and VBO code from `Renderer` `init` method. Our render method in the `Renderer` class will accept also a Mesh instance to render. The `cleanup` method will also be simplified since the `Mesh` class already provides one for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

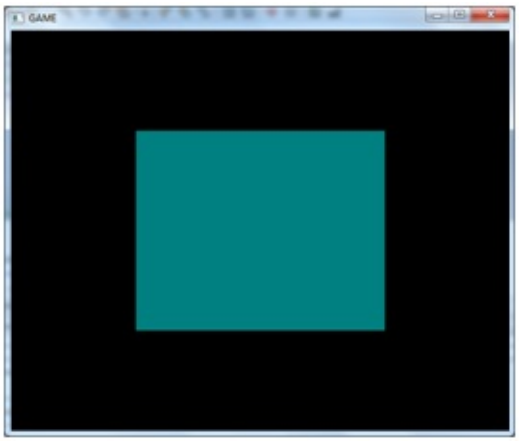
Our `Mesh` counts the number of vertices by dividing the position array by 3 (since we are passing X, Y and Z coordinates)). Now that we can render more complex shapes, let us try to render a more complex shape, let us render a quad. A quad can be constructed by using two triangles as shown in the next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



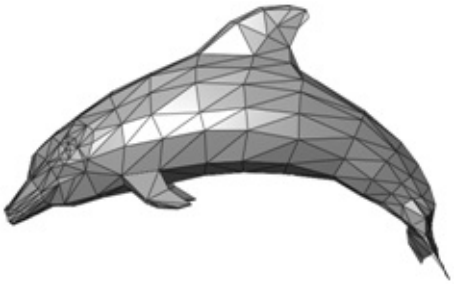
As you can each of the two triangles is composed by three vertices, the first one formed by the vertices: V1, V2 and V4 (the orange one) and the second one formed by the vertices V4, V2, V3 (the green one). Vertices are specified in a counter clockwise order, so the float array to be passed will be [V1,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



Are we done yet ? Unfortunately no, 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 image 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 cane 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 into which those vertices should be drawn by referring to their position:

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

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();
IntBuffer indicesBuffer = BufferUtils.createIntBuffer(indices.length);
indicesBuffer.put(indices).flip();
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, idxVboId);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indicesBuffer, GL_STATIC_DRAW);
```

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

And that's, 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:

- 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.

An 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 hold 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 = BufferUtils.createFloatBuffer(colours.length);
colourBuffer.put(colours).flip();
glBindBuffer(GL_ARRAY_BUFFER, colourVboId);
glBufferData(GL_ARRAY_BUFFER, colourBuffer, GL_STATIC_DRAW);
glVertexAttribPointer(1, 3, GL_FLOAT, false, 0, 0);
```

Please notice that in the `glVertexAttribPointer` call, the first parameter is now a `"1"`, this 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 processes 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);
}
```

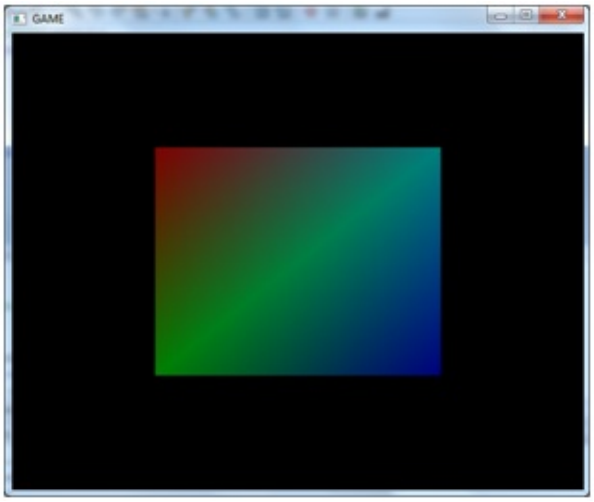
The last important thing to do is to modify our rendering code to use that second array of data:

```
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.

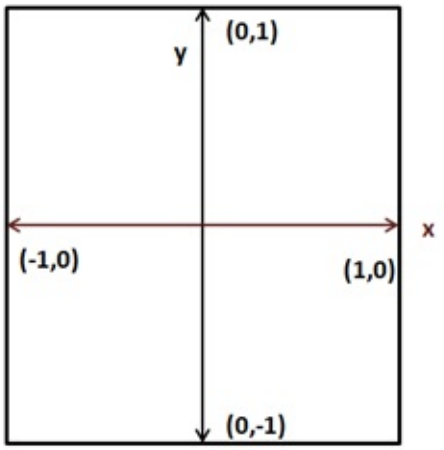


Transformations

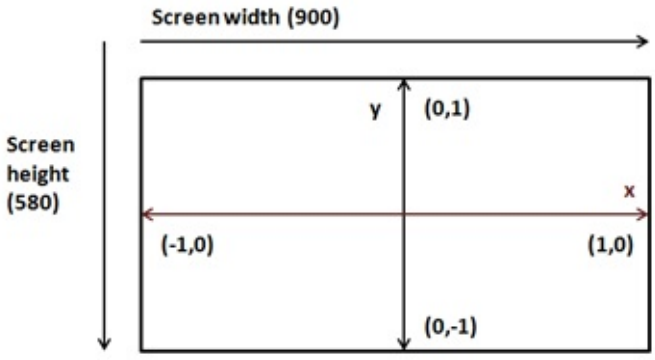
Projecting

Let’s get back to our nice coloured quad we created in previous chapter. If you look carefully at it, it resembles more to 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 in x position 0.5 in our screen. The following figure shows OpenGL coordinates (just for x and y axis).



Those coordinates are mapped, considering our window size, to window coordinates (which have the origin at top-left corner of t he 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.



But, the problem is more serious than that. Modify the z cords of our quad from 0.0 to 1.0 and to -1.0. What do see ? The quad is exactly drawn in the same place no matter if it's displaced along the z axis. Why is happening this ? 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 not be this handled by the z-coord? The answer is yes an now, The z coordinate tells OpenGL that an object is closer or far away, but OpenGL does not know nothing 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 into the screen with the same size (those would have same x and y coordinates and different z). OpenGL just uses the coordinates we are passing, so we must take care of this, we need to correctly project our coordinates.

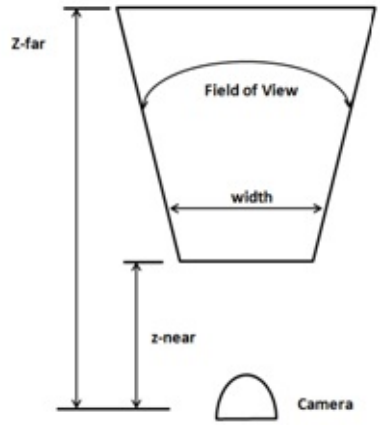
Now that we have diagnosed the problem, how do we do this ? The answer is by using a projection matrix or frustrum. The projection matrix will take care of the aspect ratio (the relation between size and height) of our drawing area so objects won't be distorted. It also will handle the distance so objects far away from us will be drawn smaller. The projection matrix will also consider our field of view and how far is the maximum distance 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 into 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 to 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 between them. We could chose to warite all that code by our own from scracth or use an already existing library. We will choose the easy path and use a specific library for dealing with math operations sin LWJGL which is called JOML (Java OpenGL Math Library). In order to use that librray 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.

```
    <properties>
        [...]
        <joml.version>1.6.5</joml.version>
        [...]
    </properties>
```

Now that everithing has been set up let's define our projection matrix. We will create a instance of the class `Matrix4f` (provided by the JOML library) in our `Renderer` class. The `Matrix4f` provides a method to set up a projection matrix named `perspective`. This methods need 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 and 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 in 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 on 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 matrix of 4x4 elements 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 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 indexing by its 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<>();
```

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 porjection 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 en 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 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
    FloatBuffer fb = BufferUtils.createFloatBuffer(16);
    value.get(fb);
    glUniformMatrix4fv(uniforms.get(uniformName), false, fb);
}
```

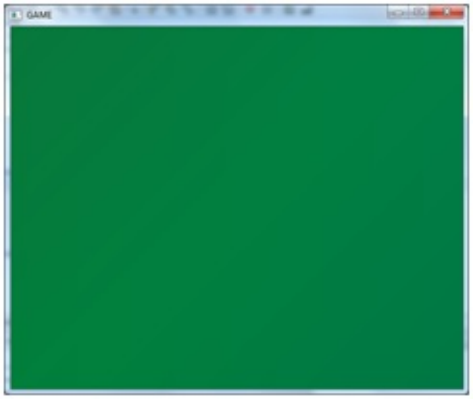
Now we can use that method in the `Renderer` class in the `render` method, after the shader program has been binded:

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

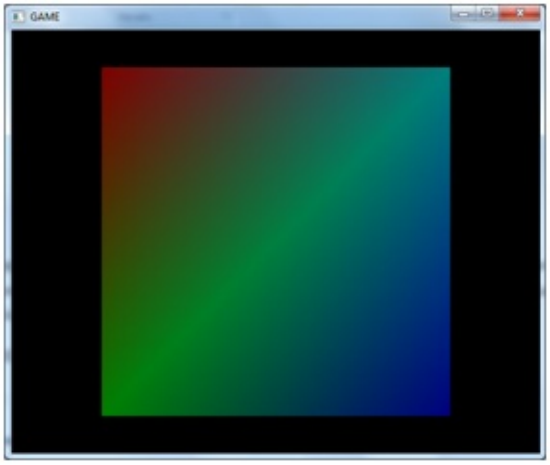
We are almost over, we can now show the quad correctly rendered, so you can now launch your program and will obtain a.... black background without any coloured quad. What's happening? Did we break something? Well, actually no, remember that we are now simulating the effect of camera looking at our scene, and that we provided to distances, one to the farthest plane (equal to 1.000f) 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 to 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 see now our coloured quad.



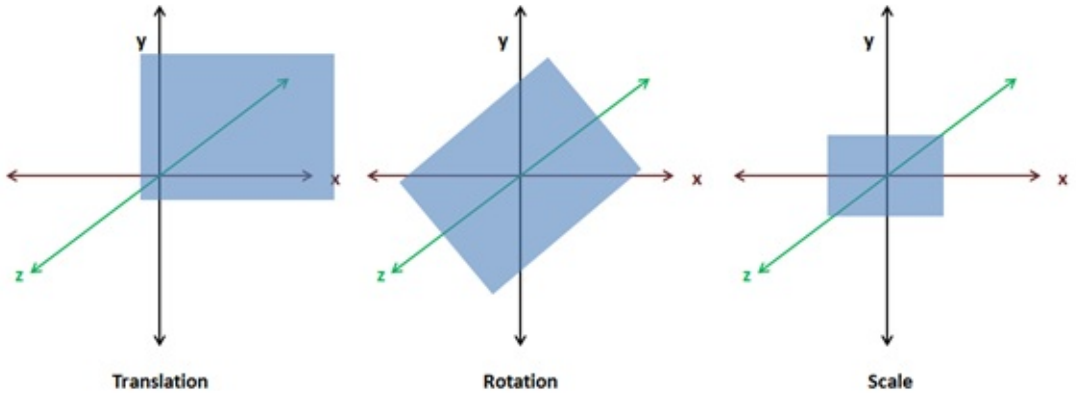
If we continue pushing backwards the quad we will see it smaller. Notice also that our quad does not resemble to a rectangle anymore.

Applying Transformations

Let’s recall what we’ve done so far. We have learned how to pass data in an efficient format to our graphic card. 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 an represent it in our 3D space in a specific position, with the appropriate size and the required rotation.

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

- Translation: Move an object by some amount in any of the three axis.
- Rotation: Rotate an object by some amount of degrees over any of the three axis.
- 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 we 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 using it’s 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 systems (the model one) to another coordinate systems (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):

$$WorldMatrix\left[TranslationMatrix\right]\left[RotationMatrix\right]\left[ScaleMatrix\right]$$

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

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

The translation matrix is defined like this:

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

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.

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 in 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 draw it independently according to its position, size and rotation. We need to transform those models when we are rendering it.

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.

```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, float
        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 matrix. 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, based on the Field Of View, the aspect ratio and the near and far distance gets the projection matrix.

An important thing notice is that the the `mul` method of the `Matrix4f` class modifies the matrix instance that applies to. So if we directly multiply the projection matrix with the transformation matrix we will be modifying the projection matrix itself. This is why 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. The uniform name has been renamed to transformation to better match its purpose.

```
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 receive now 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.getWidth()
    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", tMatrix);
        // Render the mes for this game item
        gameItem.getMesh().render();
    }

    shaderProgram.unbind();
}
```

We update the projection matrix once per `render` call. By doing 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 it's a separate variable in our Transformation class.

We have 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 simply adds a new the called `worldMatrix` and 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 frustrum, position, scale and rotation information.

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 will be doing many more matrices multiplication. Think about a model with tons of vertices and that's a lot of extra operations.

But you may now think, that if the world matrix does not change per `GameItem` instance, why we don't do the matrix multiplication in our Java class. We would be by multiplying the projection matrix and the world matrix just once per GameItem and we would send it as single uniform. In this case we would be saving many more operations. The answer is that this 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 so it's better to handlen in and independet way those two matrices.

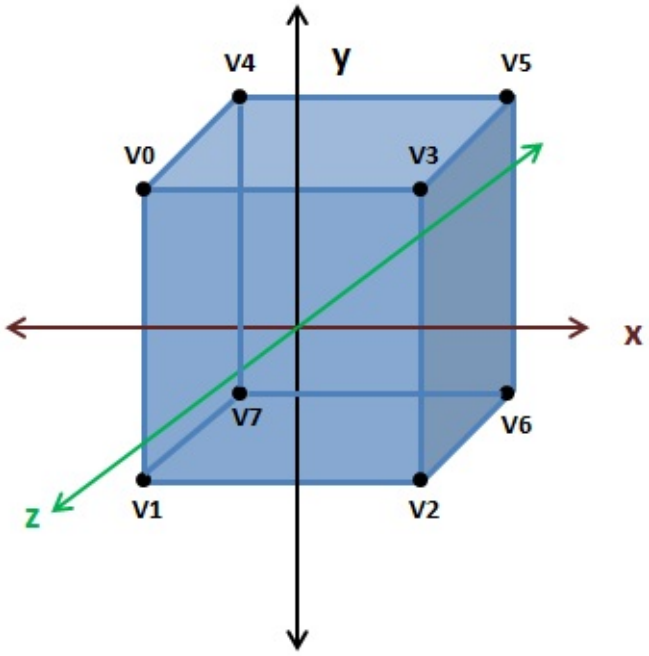
Finally we only need to change the `DummyGame` class to create a instance of `GameItem` with its associated `Mesh` and add some logic to translate, rotate and scale our quad. Since it's only a test example and does not add too much you can find it in the source code that accompanies this book.

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:

```
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,
};
```

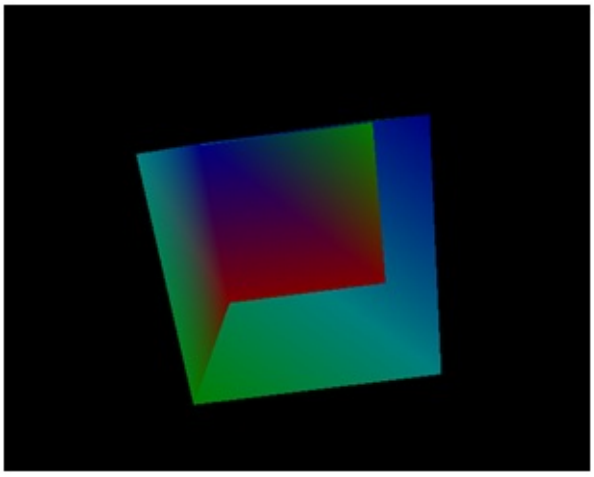
Finally, since a cube is made of six faces we need to draw twelve triangles (two per face), so we need to update the indices array. Remember that triangles must be defined in counter clock wise order.

```
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
    0, 1, 6, 4, 0, 6,
    // Bottom face
    6, 1, 2, 7, 6, 2,
    // Back face
    4, 6, 7, 5, 4, 7,
};
```

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

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

An that's all, we are now able to display a spining 3D cube. You can now compile an run your example and you will obtain something like this.

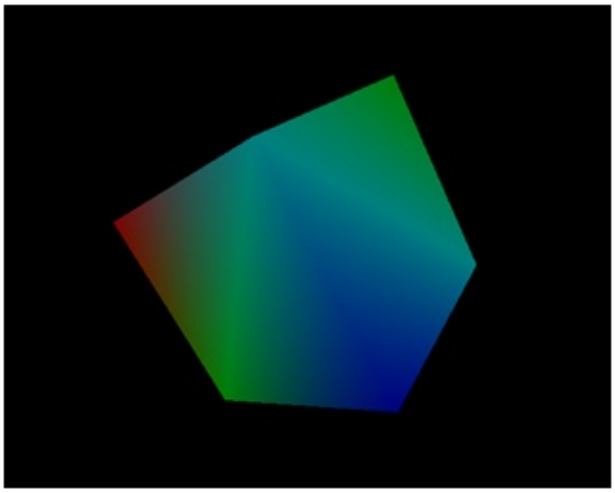


There is something weird with this cube, some faces are not being painted correctly. What is happening? The reason why the cube has this aspect is that the triangles that compose the cube are being drawn in a sort of random order. The pixels that are far away should be drawn before pixels that are closer.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

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

```
glEnable(GL_DEPTH_TEST);
```

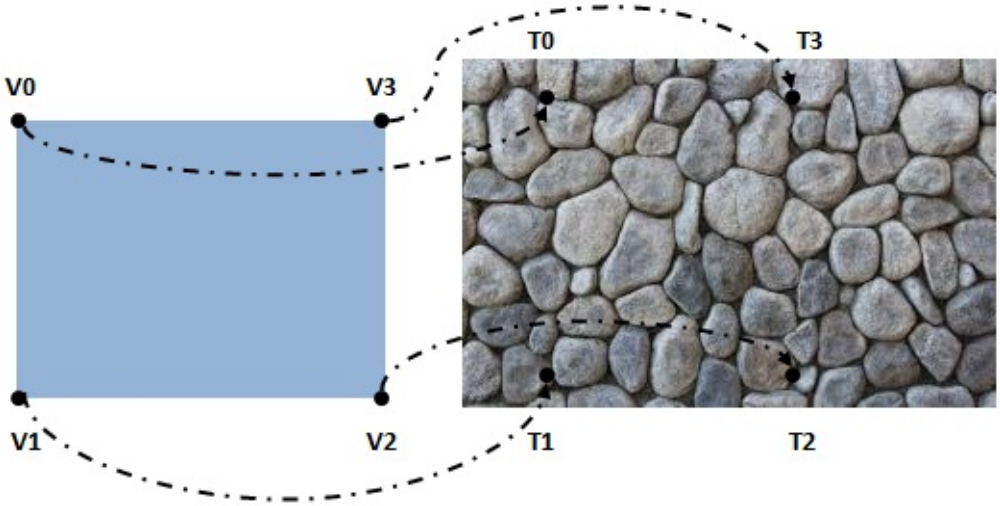
Now our cube is being rendered correctly!.



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

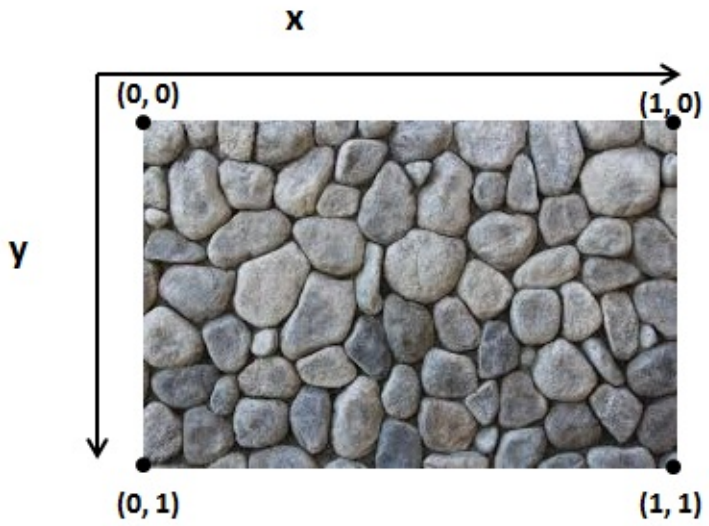
Now we are going to apply a texture to our cube. A texture is an image which is used to draw the colour of the pixels of a certain model. You can think about a texture like a skin that is wrapped around your 3D model. What you do is assign points in the image texture to the vertices in your model. With 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 textura to a model, is assign texture coordinates to each of our vertices. Texture coordinates system are a bit different than the coordinates 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 equal to 1.



How do we relate texture coordinates with our position coordinates? Easy, the same way as 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 tasks, 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.

```
<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 fixed pipepline, since we are using GLSL shader 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 two 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 in bytes 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 to 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 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, before calling the `glTextImage2D` method, filtering parameters are set up. 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.

By this moment we will not set up those parameters, instead of that 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 have 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);
FloatBuffer textCoordsBuffer = BufferUtils.createFloatBuffer(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 graphic 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 evolve it more 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:

```
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.

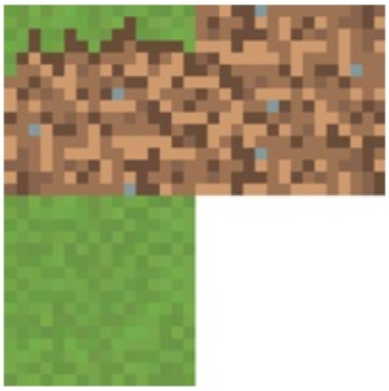
```
shaderProgram.setUniform("texture_sampler", 0);
```

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:

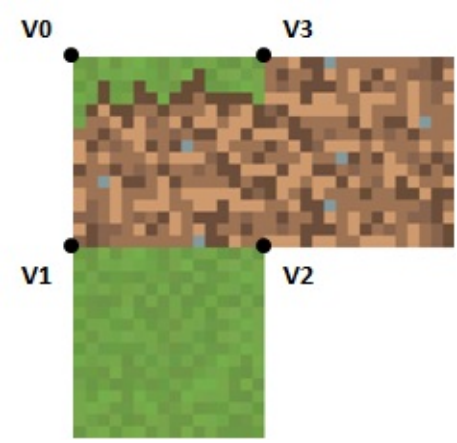
```
// Activate first texture unit
glActiveTexture(GL_TEXTURE0);
// Bind the texture
glBindTexture(GL_TEXTURE_2D, texture.getId());
```

We basically are binding to the texture identified by `texture.getId()` in 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:

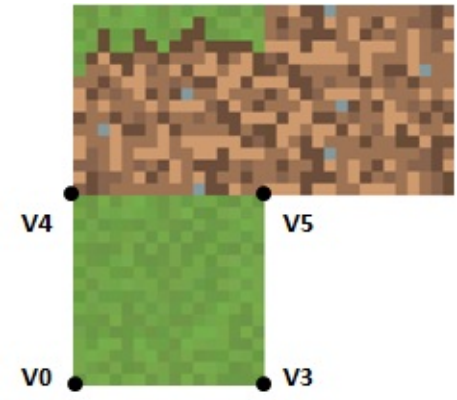


In our 3D model we have eight vertices. Let's see how this can be done. Let's first define the front face texture coordinates for each vertex.



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

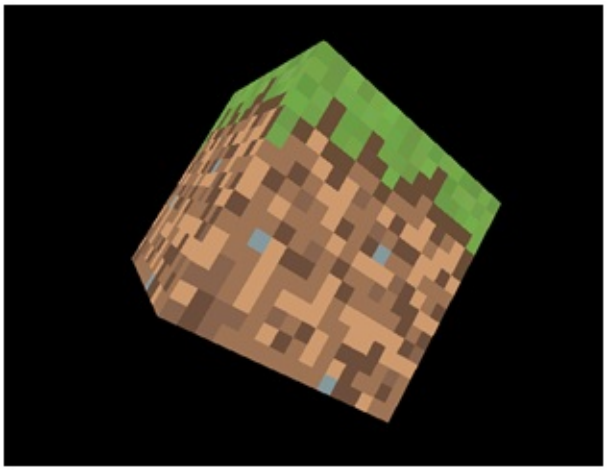
Now, let's define the texture mapping of the top face.



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.

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 pass 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).

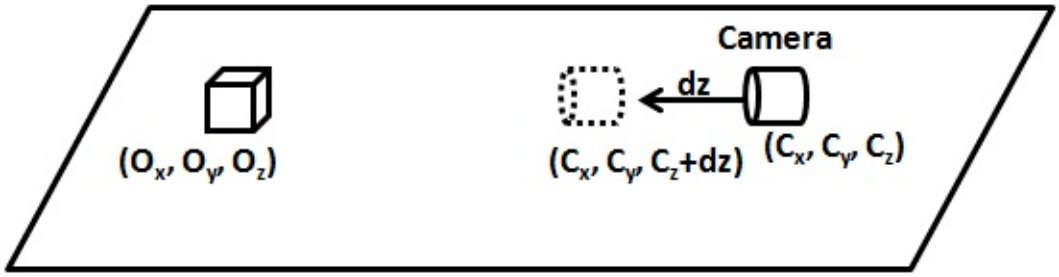
Camera

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.

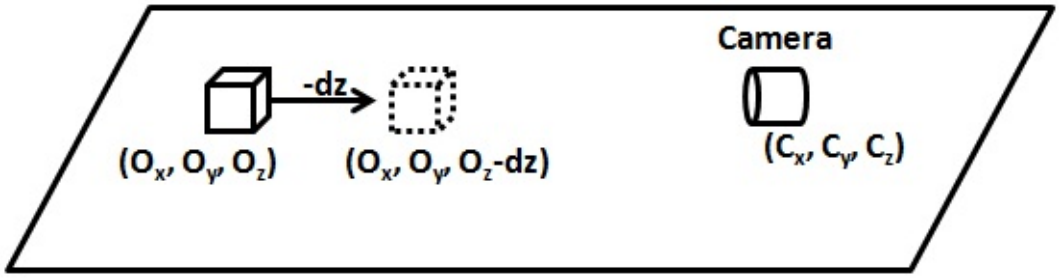
But if you try to search for specific camera functions in OpenGL you will discover that there is no camera concept, or in other words the camera is always fixed, centered in the (0, 0, 0) position at the center of the screen.

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.

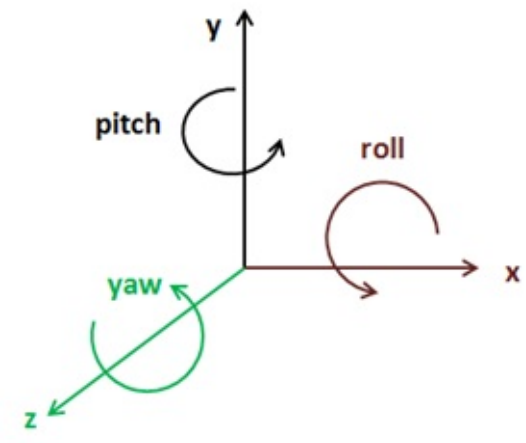
So, suppose that we would like to move the camera position along the z axis from a starting position (Cx, Cy, Cz) to a position (Cx, Cy, Cz+dz) to get closer to the object which is placed at the coordinates (Ox, Oy, Oz).



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).



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:

$$Transf = [Proj\ Matrix] \cdot [Translation\ Matrix] \cdot [Rotation\ Matrix] \cdot [Scale\ Matrix] = [Proj\ Matrix] \cdot [World\ Matrix]$$

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

$$Transf = [Proj\ Matrix] \cdot [View\ Matrix] \cdot [Translation\ Matrix] \cdot [Rotation\ Matrix] \cdot [Scale\ Matrix] = [Proj\ Matrix] \cdot [World\ Matrix] = [Proj\ Matrix] \cdot [View\ Matrix] \cdot [World\ Matrix]$$

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 * offsetZ;
            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 * offs
            position.z += (float)Math.cos(Math.toRadians(rotation.y - 90)) * offsetX;
        }
        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.y += 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(), wi
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();
}
```

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.*;
import org.lwjgl.glfw.GLFWCursorPosCallback;
import org.lwjgl.glfw.GLFWCursorEnterCallback;
import org.lwjgl.glfw.GLFWMouseButtonCallback;

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;

    private GLFWCursorPosCallback cursorPosCallback;

    private GLFWCursorEnterCallback cursorEnterCallback;

    private GLFWMouseButtonCallback mouseButtonCallback;

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

    public void init(Window window) {
        glfwSetCursorPosCallback(window.getWindowHandle(), cursorPosCallback = new GLFWCu
            @Override
            public void invoke(long window, double xpos, double ypos) {
                currentPos.x = xpos;
                currentPos.y = ypos;
            }
        });
        glfwSetCursorEnterCallback(window.getWindowHandle(), cursorEnterCallback = new GL
            @Override
            public void invoke(long window, int entered) {
                inWindow = entered == 1;
            }
        });
        glfwSetMouseButtonCallback(window.getWindowHandle(), mouseButtonCallback = new GL
            @Override
            public void invoke(long window, int button, int action, int mods) {
                leftButtonPressed = button == GLFW_MOUSE_BUTTON_1 && action == GLFW_PRESS
                rightButtonPressed = button == GLFW_MOUSE_BUTTON_2 && action == GLFW_PRES
            }
        });
    }

    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 received mouse evevents 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.

One important thing related to callbacks and GLFW is that we need to keep a reference to the callback implementation into our Java class. You see that we have one attribute per callback. This is because callbacks are implemented in native code and the Java part of GLFW des not hold any reference to them. If we don't hold a reference they will be garbage collected and you will see an exception like this:

```
Exception in thread "GAME_LOOP_THREAD" org.lwjgl.system.libffi.ClosureError: Callback fai
```

The `MouseInput` class provides an input method which should be 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;
    }
}
```

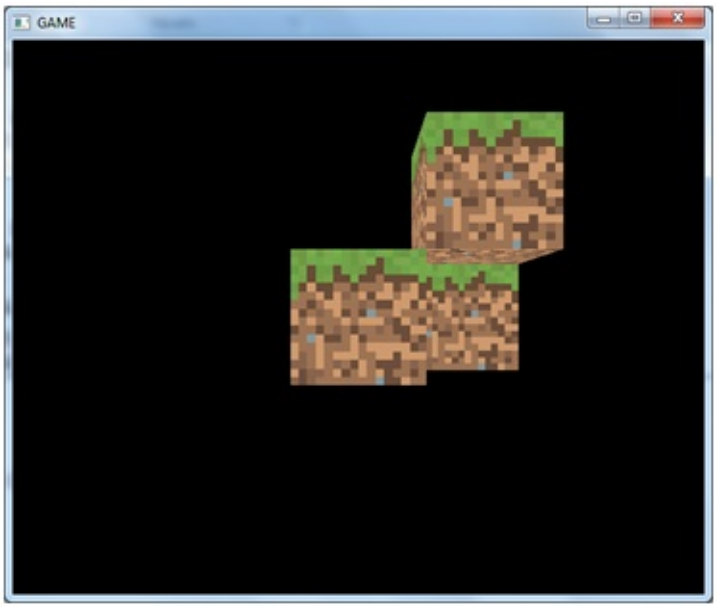
It just updates a `Vector3f` variable named `cameraInc` which holds the camera displacement that should be applied. The update method of the `DummyGame` class modifies the camera position and rotation according to the processes key and mouse events.

```
@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.


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.

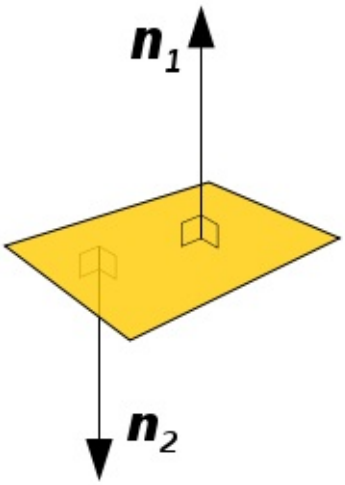
OBJ (or .OBJ) is a geometry definition open file format developed by Wavefront Technologies which has been widely adopted. An OBJ file defines the vertices, texture coordinates and polygons that compose a 3D model. It's a relative easy format to parse since is text based and each line defines an element (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 and texture coordinates (f v1/t1 v2/t2 V3/t3). Example: f 6/4 3/5 7/6.
    - It can define vertex positions and texture coordinates (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 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.

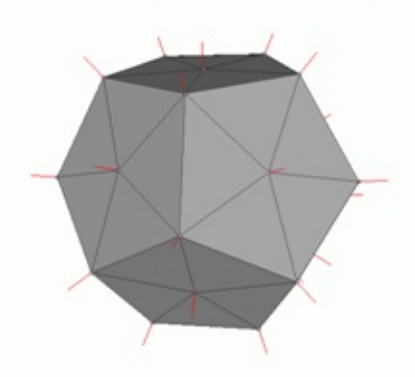
But what is anormal ? Let's define it first. When you have a plane its normal is a vector perpendicular to that plane which has a length equal to one.



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);
FloatBuffer vecNormalsBuffer = BufferUtils.createFloatBuffer(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:

```
#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 = Files.readAllLines(Paths.get(OBJLoader.class.getResource(fileName).t

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.

```
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.

```
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(textCoord)
            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.

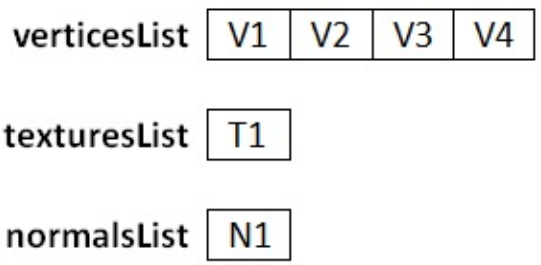
Now we can talk about how to reorder the information we have. Finally we need to reorder that information. Our `Mesh` class expects four arrays, one for position coordinates, other for texture coordinates, other for vector normals and another one for the indices. The first three arrays shall have the 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).

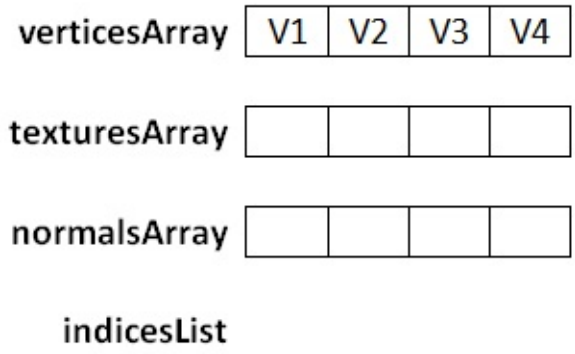
```
v 0 0 0
v 1 0 0
v 1 1 0
v 0 1 0

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)

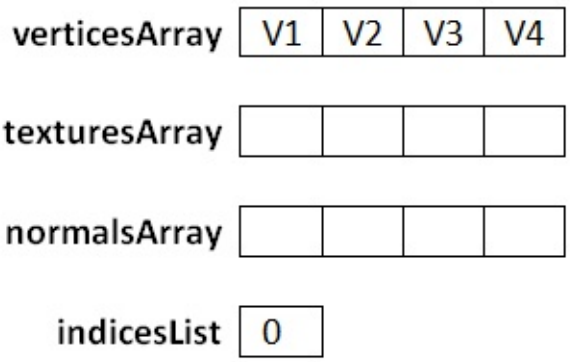


Now we will use the face definitions to construct the final arrays including the indices. A thing to take into consideration is that the order in which textures coordinates and vector normals are defined does not correspond to the orders in which vertices are defined. If the size of the lists would be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

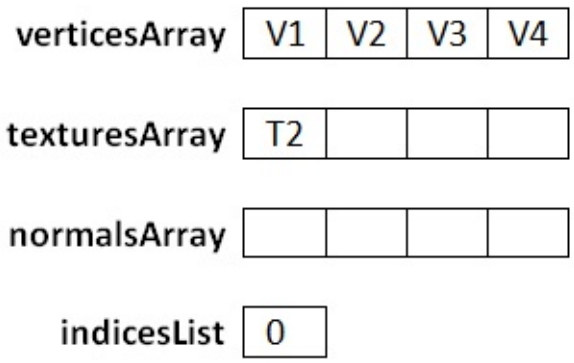
So we need to order the data and setup accordingly to our needs. The first thing that we must do is create three arrays and one list, one for the vertices, other for the texture coordinates, other for the normals and the list for the indices. As we have said before the three arrays will have the same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



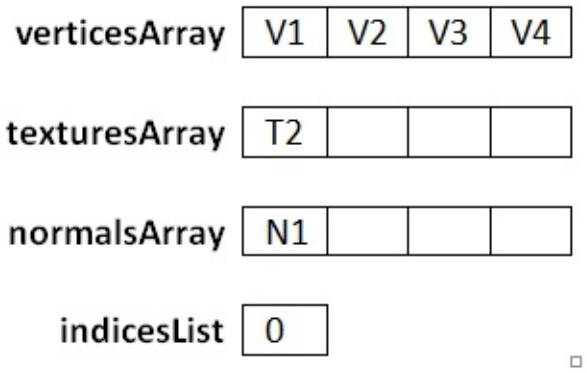
Now we start processing the faces. The first index group of the first face is 1/2/1. We use the first index in the index group, the one that defines the geometric vertex to construct the index list. Let's name it as `posIndex`. Our face is specifiying that the we should add the index of the element that 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).



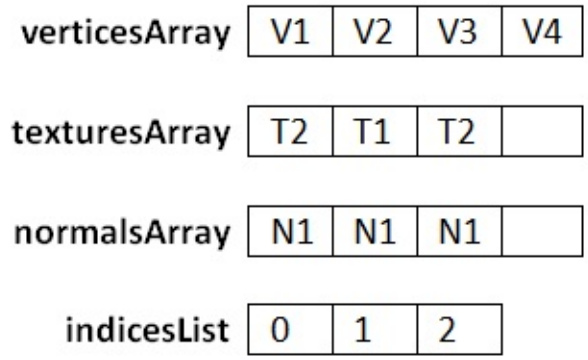
Then we use the rest of the indices of the index group to set up the `texturesArray` and `normalsArray`. The second index, in the index group, is 2, so what we must do is put the second texture coordinate in the same position as the one that occupies the vertex designated posIndex (V1).



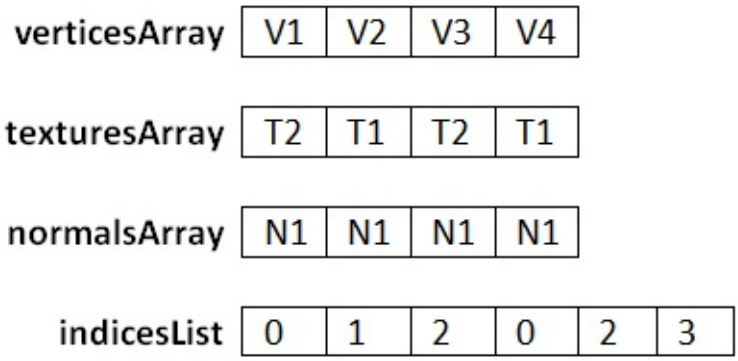
Then we pick the third index, which is 1, so what we must do is put the first vector normal coordinate in the same position as the one that occupies the vertex designated `posIndex` (V1).



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

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.

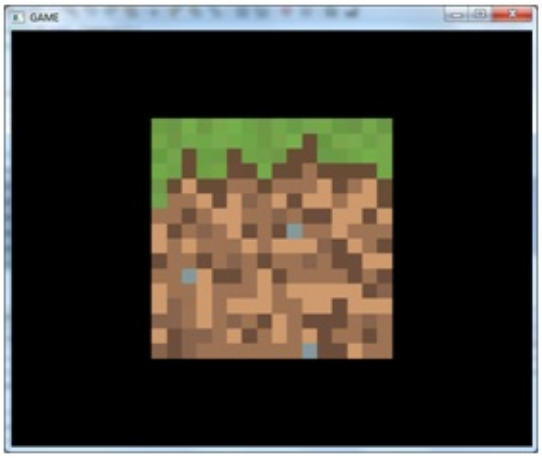
```
private static Mesh reorderLists(List<Vector3f> posList, List<Vector2f> textCoordList,
    List<Vector3f> normList, List<Face> facesList) {

    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;
    }
}
```

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.

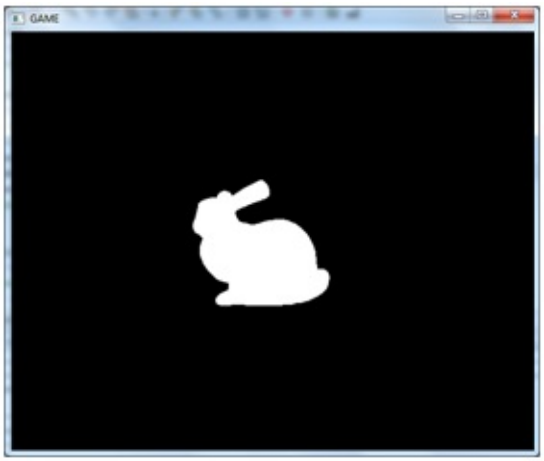
```
Texture texture = new Texture("/textures/grassblock.png");
mesh.setTexture(texture);
GameItem gameItem = new GameItem(mesh);
gameItem.setScale(0.5f);
gameItem.setPosition(0, 0, -2);
gameItems = new GameItem[]{gameItem};
```

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 us it this way.

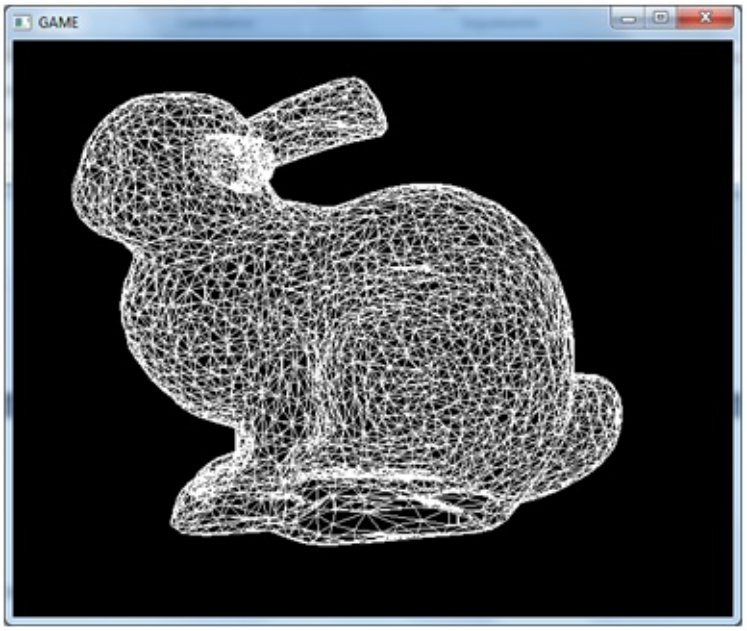
```
Mesh mesh = OBJLoader.loadMesh("/models/bunny.obj");
GameItem gameItem = new GameItem(mesh);
gameItem.setScale(1.5f);
gameItem.setPosition(0, 0, -2);
gameItems = new GameItem[]{gameItem};
```



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.

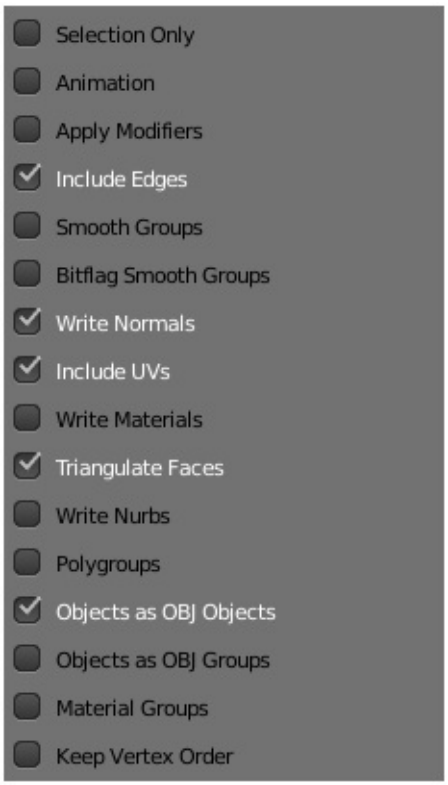
```
glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );
```

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

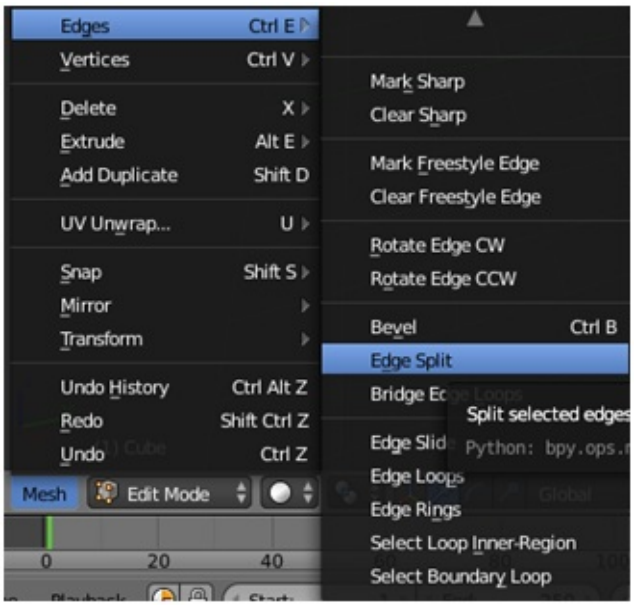


Now you can now see all the triangles that compose the model.

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.



Also if you are applying textures please remember to split edges since we cannot assign several texture coordinates to the same vertex,

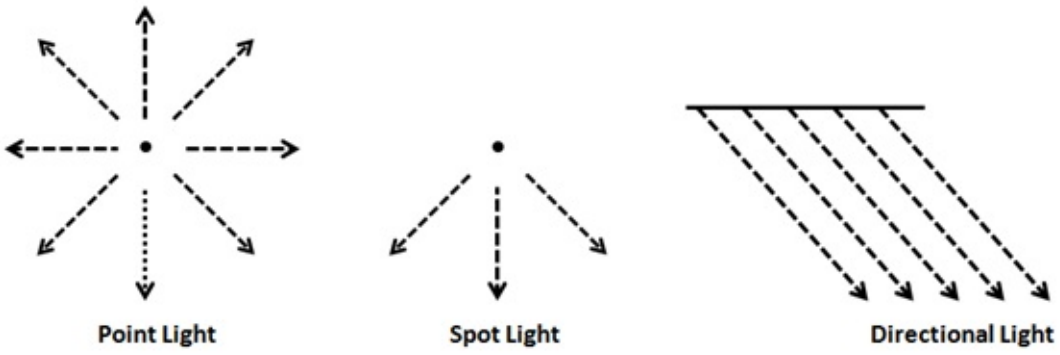


Let there be light

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

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). So the final colour will be: $L * basecolour$.

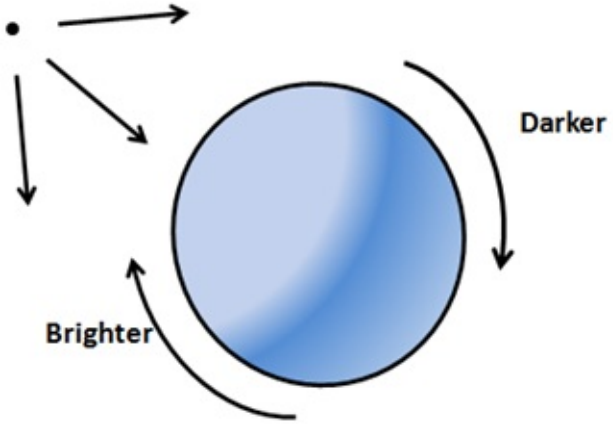
Ambient Light component

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.

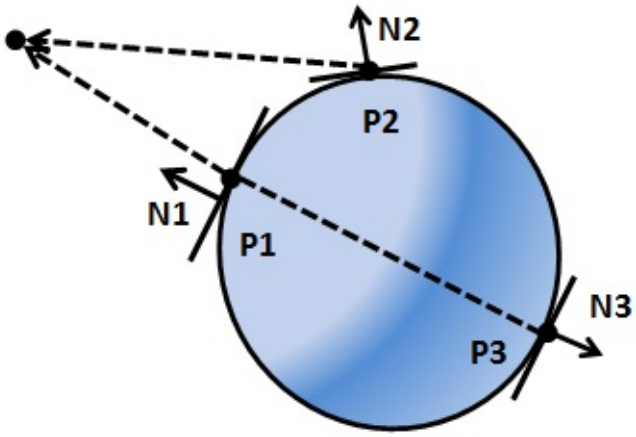
Ambient light is the easiest component to calculate, we just need to pass a colour, since it will be multiplied by our base colour it just modulates that base colour. Imagine that we have determined that a colour for a fragment is $(1.0, 0.0, 0.0)$, that is red colour. Without ambient light it will be 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).



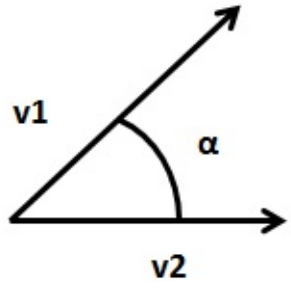
As you can see, the normal associated to $P1$, named $N1$, is parallel to the vector that points to the light source, which models the opposite of the light ray ($N1$ has been sketched displaced so you can see it, but it’s equivalent mathematically). $P1$ has an angle equal to 0 with the vector that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

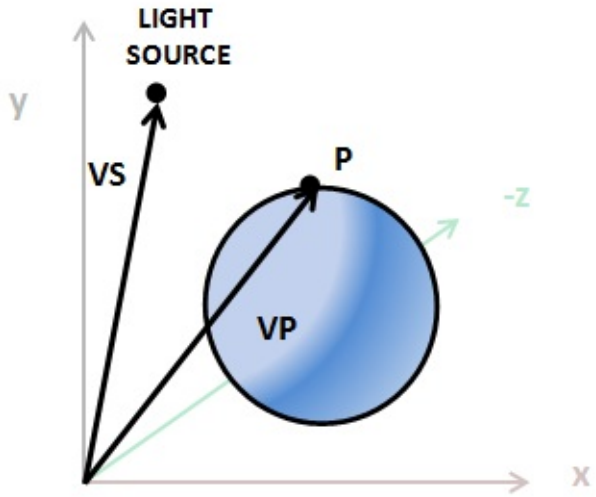
Let's define two vectors, $v1$ and $v2$, and let $alpha$ be the angle between them. The dot product is defined by the following formula.



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

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.
- $intensity$: 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

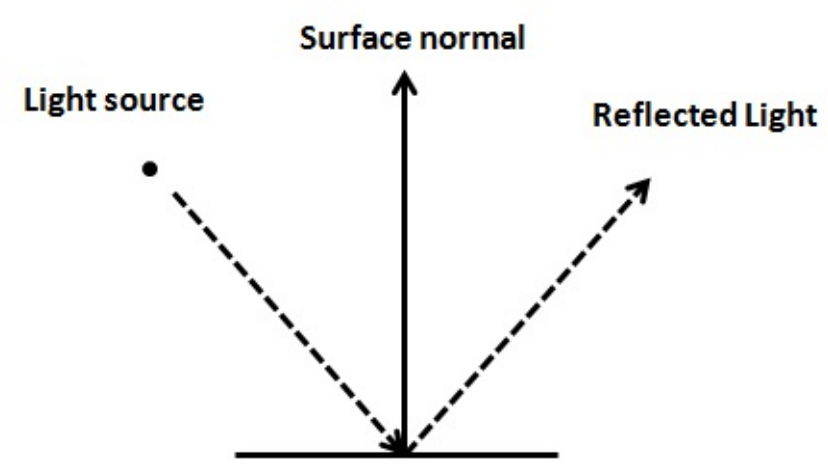
Then we need to calculate the diffuse factor (an scalar): $diffuseFactor = normal \cdot toLightDirection$. It's calculated as dot product between two vectors, since we want it to be between -1 and 1 both vectors need to be normalized. Colours need to be between 0 and 1 so if a value it's lower than 0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

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

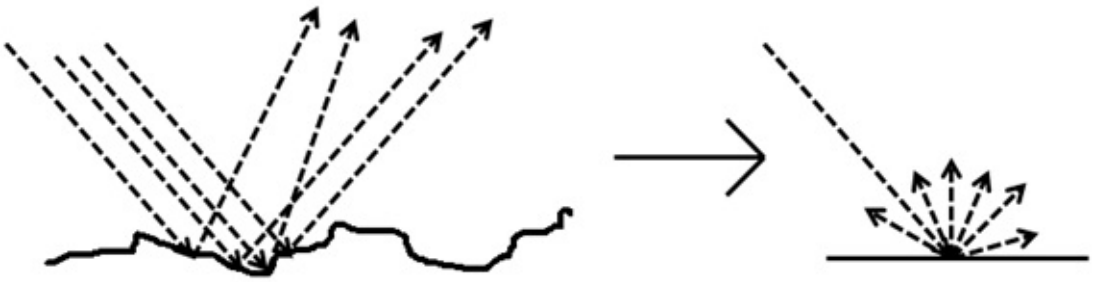
$$colour = lColour * diffuseFactor * intensity$$

Specular component

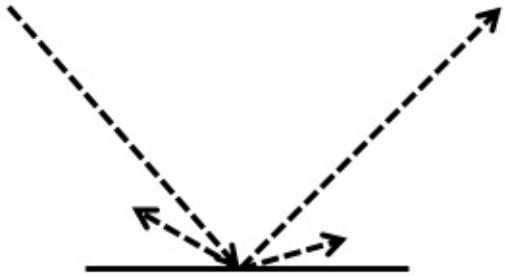
Let's view now the specular component, but first we need to examine how light is reflected. When light hits a surface some part of it is absorbed and the other part is reflected, if you remember from your physics class, reflection is when light bounces off an object.



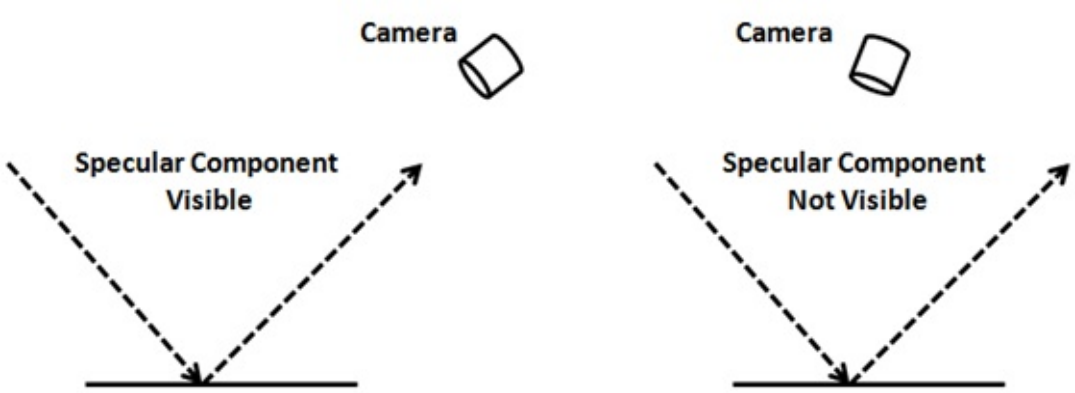
Of course, surfaces are not totally polished, and if you look at closer distance you will see a lot of imperfections. Besides that, you have many ray lights (photons in fact), that impact that surface, and that get reflected in a wide range of angles. Thus, what we see is like a beam of light being reflected 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.



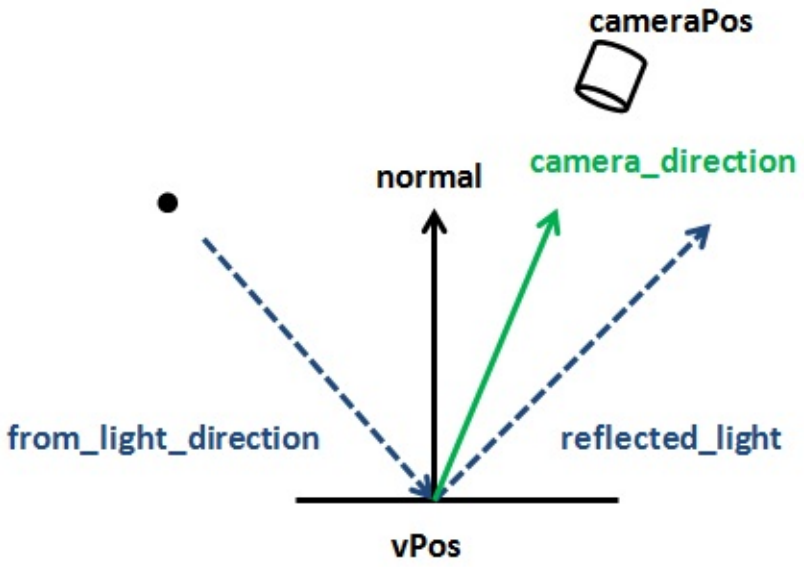
This is what the specular component models, and it depends on the material characteristics. Regarding specular reflectance, it's important to note that the reflected light will only be visible if the camera is in a proper position, that is, if it's in the area of where the reflected light is emitted.



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, $reflectedLight = reflect(fromLightSource, normal)$.

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:
$lColour * reflectance * specularFactor * intensity$.

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.

```
#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
{
    vec3 colour;
    int useColour;
    float reflectance;
};
```

A material is defined by a base colour (if we don't use texture to colour the fragments), a flag that controls that behaviour 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:

- 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.

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 = vec4(light.colour, 1.0) * light.intensity * diffuseFactor;

    // Specular Light
    vec3 camera_direction = normalize(camera_pos - 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 = 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. With that function, the main function of the vertex function is very simple.

```
void main()
{
    vec4 baseColour;
    if ( material.useColour == 1 )
    {
        baseColour = vec4(material.colour, 1);
    }
    else
    {
        baseColour = texture(texture_sampler, outTexCoord);
    }
    vec4 lightColour = calcPointLight(pointLight, mvVertexPos, mvVertexNormal);

    vec4 totalLight = vec4(ambientLight, 1.0);
    totalLight += lightColour;

    fragColor = baseColour * totalLight;
}
```

The first part of the function is the same as the one used in previous chapter, we calculate the fragment colour either by using a fixed colour or by calculating it from texture coordinates. Final colour is calculated by multiplying that colour and the summation of the ambient light, diffuse and colour components. As you can see ambient light is not affected by attenuation.

We have introduced some new concepts into our shader, 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 + ".colour");
    createUniform(uniformName + ".useColour");
    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.

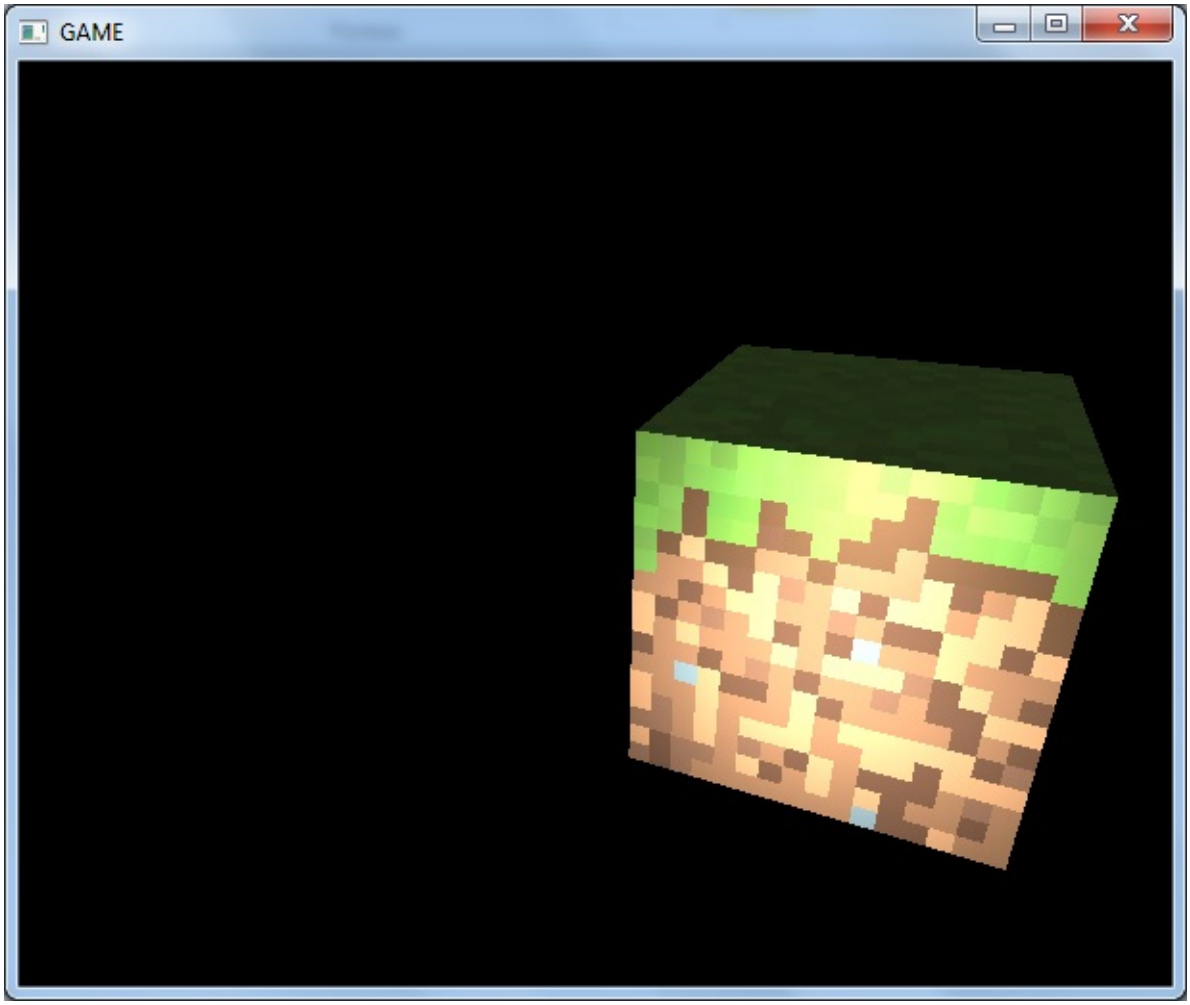
```
public void setUniform(String uniformName, PointLight pointLight) {
    setUniform(uniformName + ".colour", pointLight.getColor() );
    setUniform(uniformName + ".position", pointLight.getPosition());
    setUniform(uniformName + ".intensity", pointLight.getIntensity());
    PointLight.Attenuation 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 + ".colour", material.getColour() );
    setUniform(uniformName + ".useColour", material.isTextured() ? 0 : 1);
    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);
```

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.

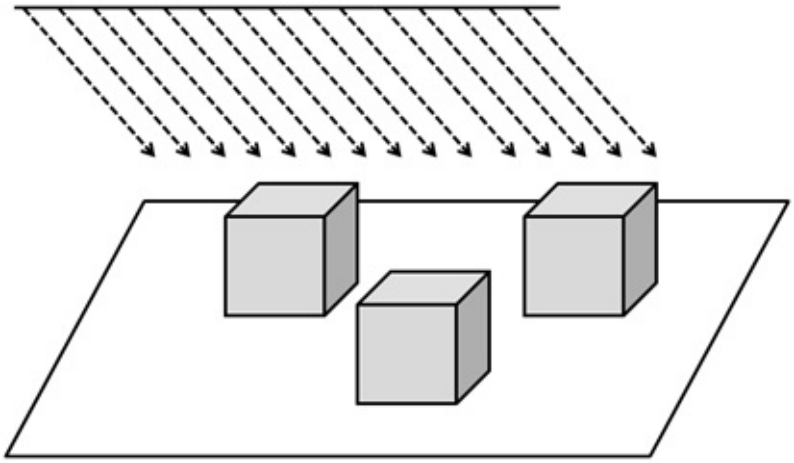

GAME

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.

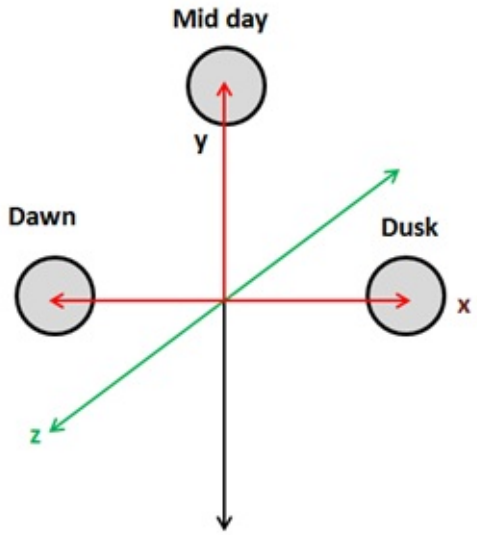
Directional Light

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.



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.ge
    }

    // 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.

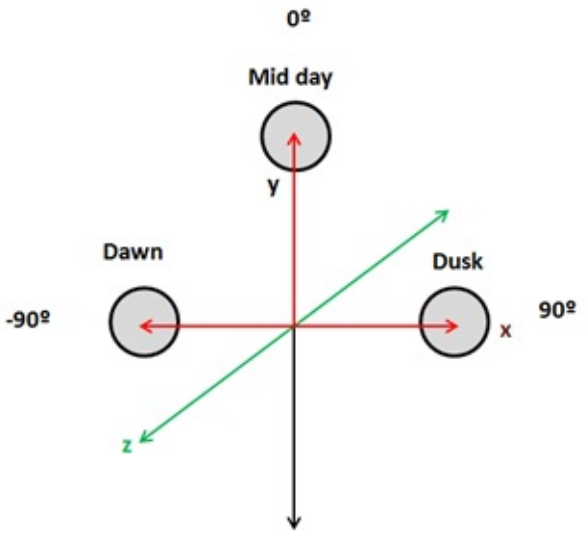
In our fragment shader we will define a structure that models a directional light.

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

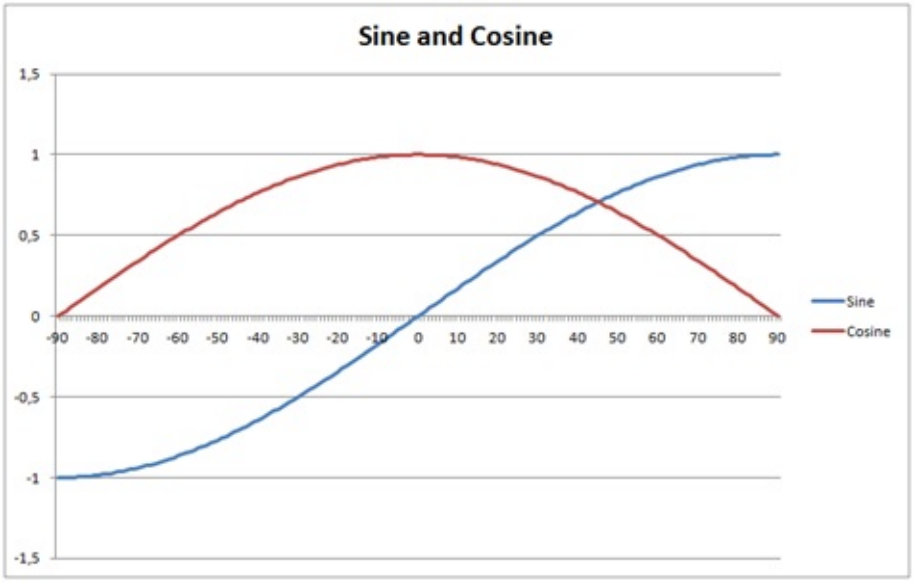
With that definition the new methods in the `ShaderProgram` class are straight forward.

```
// ...
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.

```
// Get a copy of the directional light object and transform its position to view coordina
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_lig
{
    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 = vec4(light_colour, 1.0) * light_intensity * diffuseFactor;

    // 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 = light_intensity  * specularFactor * material.reflectance * vec4(light_co

    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_light

    // 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;
}
```

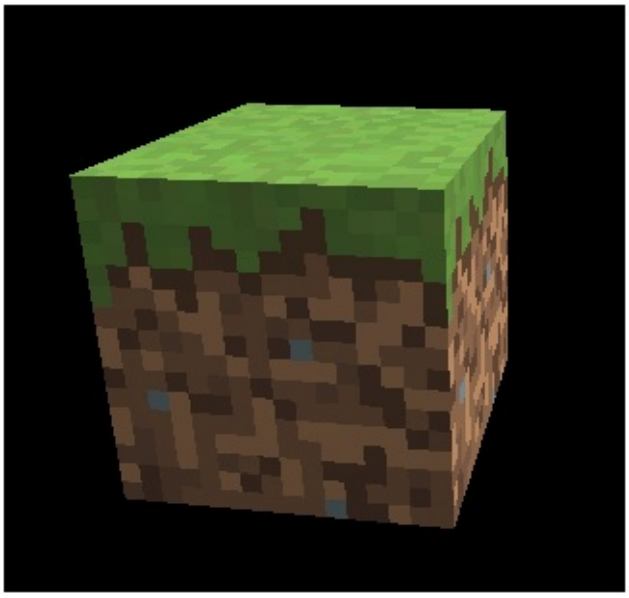
We will create also a new function to calculate the effect of a directional light which just invokes the `calcLightColour` function with the light direction.

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

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

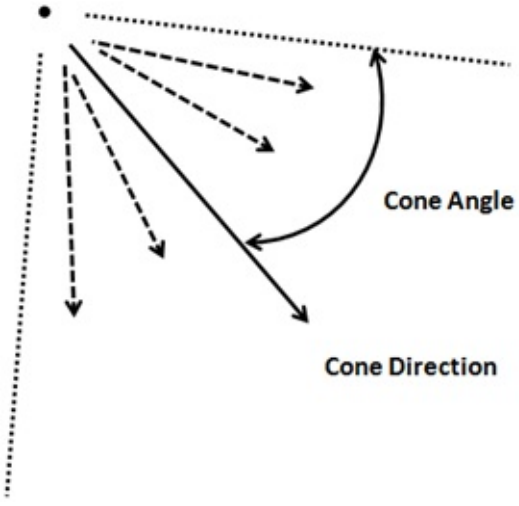
```
void main()
{
    vec4 baseColour;
    if ( material.useColour == 1 )
    {
        baseColour = vec4(material.colour, 1);
    }
    else
    {
        baseColour = texture(texture_sampler, outTexCoord);
    }
    vec4 totalLight = vec4(ambientLight, 1.0);
    totalLight += calcDirectionalLight(directionalLight, mvVertexPos, mvVertexNormal);
    totalLight += calcPointLight(pointLight, mvVertexPos, mvVertexNormal);

    fragColor = baseColour * totalLight;
}
```

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).

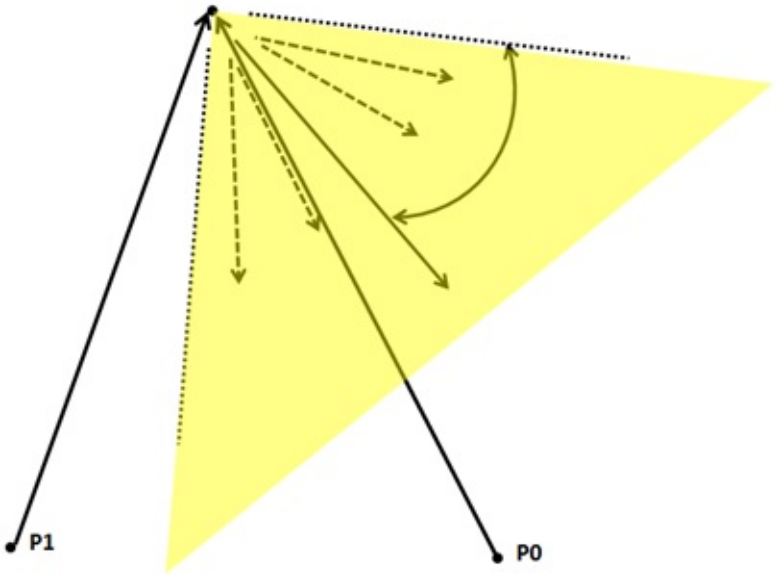


Spot Light

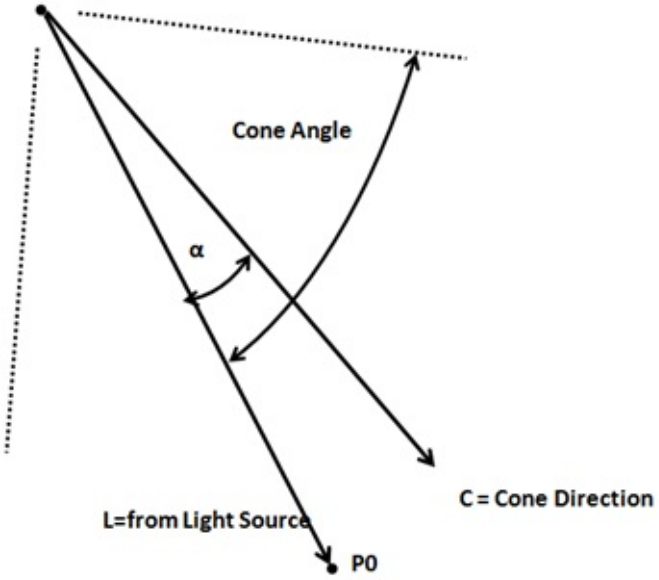
Now we will implement a spot light which are very similar to a point light but the emitted light is restricted to a 3D cone. It models the light that comes out from focuses or any other light source that does not emit in all directions. A spot light has the same attributes as a point light but adds two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



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 now 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).

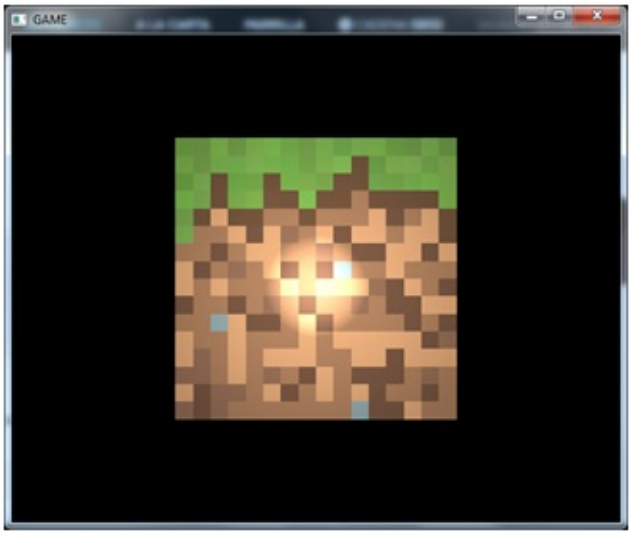
The second difference is that the points that are far away from the cone vector will receive less light, that is, the attenuation will be higher. There are several ways of calculate this, we will chose a simple approach by multiplying the attenuation by the following factor:

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

(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).

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

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

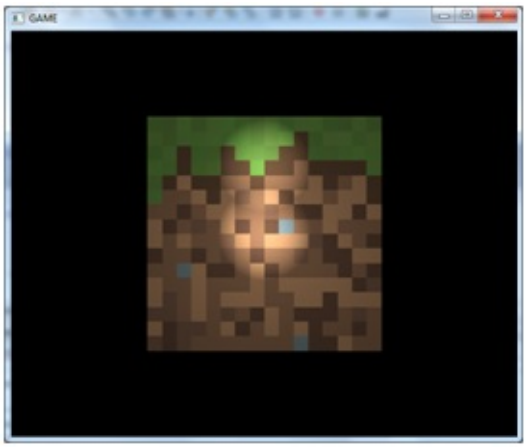
Now we need to create those uniforms in the `Render` class. When we are using arrays we need to create a uniform for each element of the list. So, for instance, for the $pointLights$ array we need to create a uniform named `pointLights[0]`, `pointLights[1]`, etc. And of ocurse, this translates also to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

Game HUD

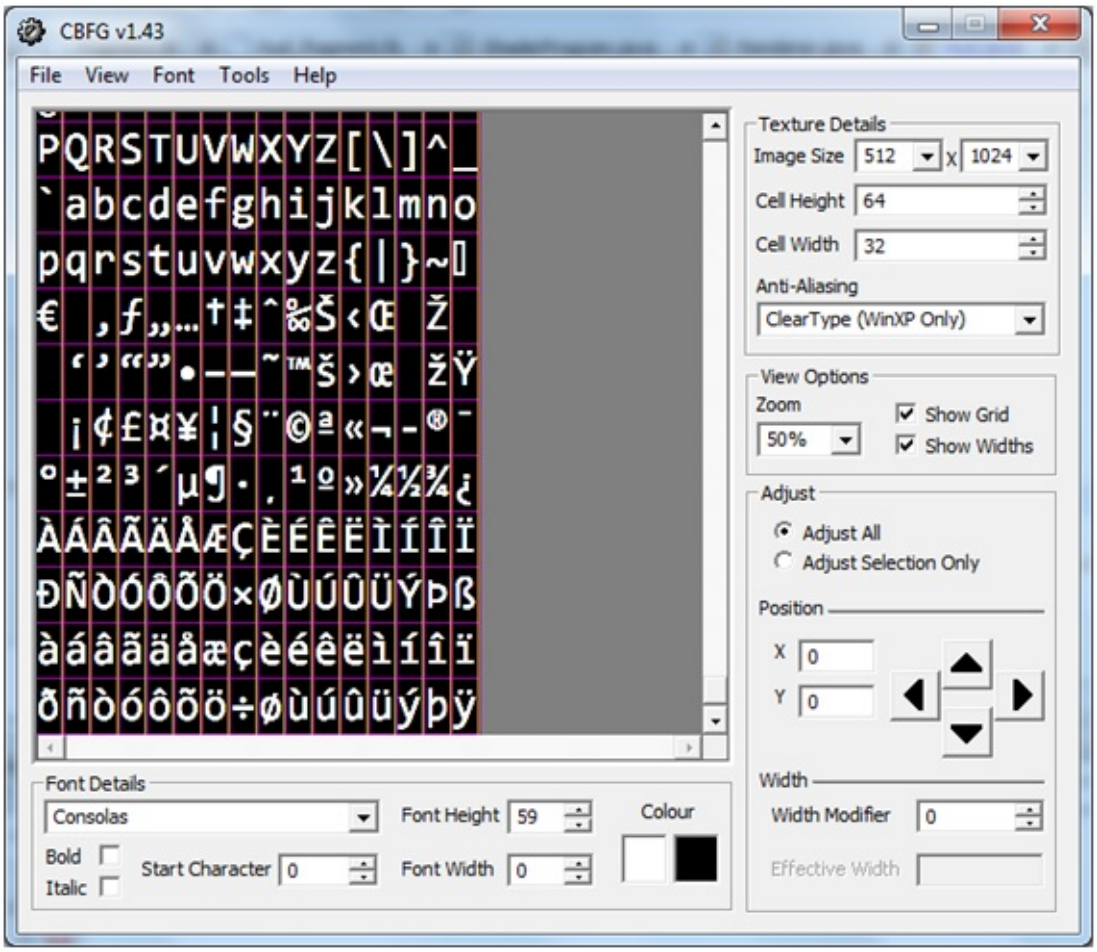
In this chapter we will create a HUD (Heads-Up Display) for our game. That is, a set of 2D shapes and text that is displayed at any time over the 3D scene to show relevant information. We will create a simple HUD that will serve us to show some basic techniques for representing that information.

You will see also that some little refactoring has been applied to the source code, especially in the `Renderer` class to prepare it for the separation of the 3D scene and the HUD rendering.

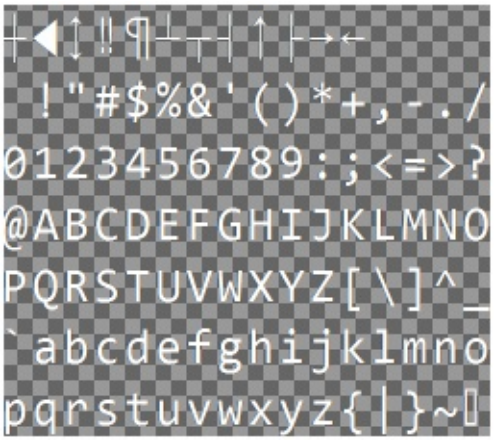
Text rendering

The first thing that we will do is render text. in order to do so what we are going to do is to map a texture that contains alphabet characters into a quad which is formed by a set of tiles, each of them representing a single letter. So to start we must create the texture that contains the alphabet, there are many programs out there that can do this task, such as, CBG, F2IBuilder, etc.

We will use Codehead's Bitmap Font Generator (CBFG). This tool 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 our 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 you can export the result to several image formats. In this case we will export it as a BMP file and later on we will 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 equals to 0 (You can use GIMP to do that). We will have something similar as the following picture.



We have 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 column is

calculated as follows: $column = code \ mod \ numberOfColumns$. Where mod is the module operation. The row is 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.

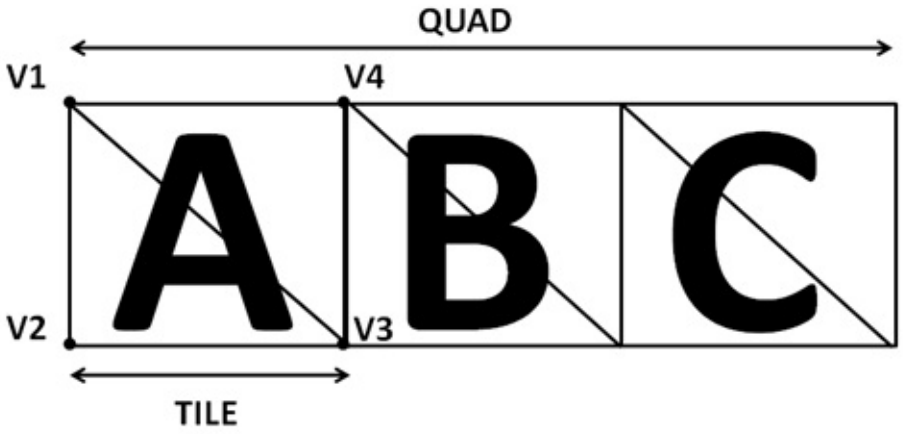
```
package org.lwjglb.engine;

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 Ex
        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 itself and the relevant data of the texture 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. 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 one (the upper one). Texture coordinates are calculated based on the column and the row associated to each character, 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.

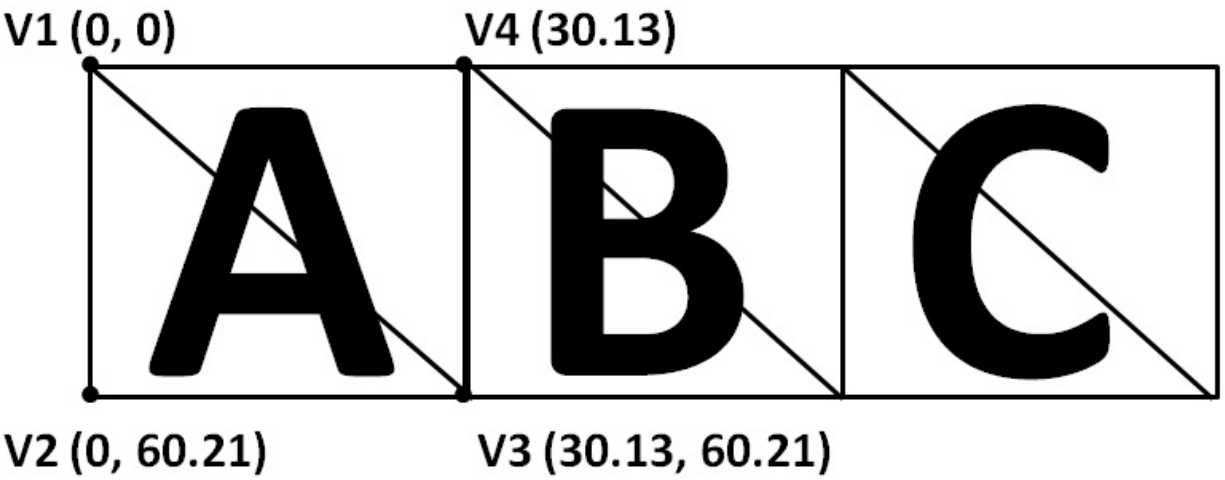
```
for(int i=0; i<numChars; i++) {
    byte currChar = chars[i];
    int col = currChar % numCols;
    int row = currChar / numCols;

    // 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.

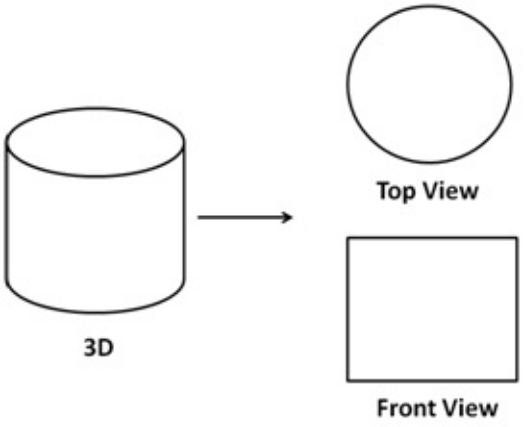


Why do we use screen coordinates ? First of all, because we will be rendering 2D objects in our HUD and often is more handy to use them, and secondly because we will use an orthographic projection in order to draw them. We will explain what is an orthographic projection later on.

The `TextItem` class is completed with other methods to get the text and to change it. 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));
}
```

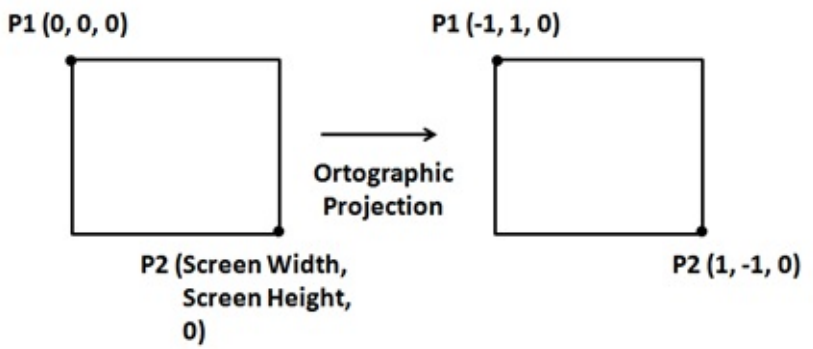
As it has been said in the beginning of this chapter, we need first to render our 3D scene and the render our 2D HUD. The HUD is composed by 2D objects, texts, shapes and we won't apply any lightning effects. Besides that we will use an orthographic projection (also named orthogonal projection) in order to render all those objects. An Orthographic projection is a 2D representation of a 3D object, you may 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 z coordinate, the distance to the view, so the size of the objects does 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 you can see 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 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 transforms screen coordinates to 3D space coordinates, the following picture shows how this mapping is done.



This will allow us to use screen coordinates.

So we will need another set of shaders, one vertex and one fragment shader in order to draw the objects in our HUD. The vertex shader is 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 combines the orthographic projection matrix and the model matrix associated to each element. 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 the java code to avoid doing it for each vertex, we will just do it once per item. Remember that our vertices should be expressed in screen coordinates.

The fragment shader is also very simple.

```
#version 330

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

uniform sampler2D texture_sampler;
uniform vec3 colour;

void main()
{
    fragColor = vec4(colour, 1) * 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. Now that we have created two more shaders we can use them in the Renderer class, but before that we will create an interface named `IHud` that will contain all the objects that are to be displayed in the HUD. The interface also provides 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 it's generic enough to not be dependent on the specific implementation of each game. In the `Renderer` class we 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 firs 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:

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

    hudShaderProgram.bind();

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

        // 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, flo
    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.Vector3f;
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_ROWS
        this.statusTextItem.getMesh().getMaterial().setColour(new Vector3f(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.

In the Texture class we need to modify the way textures are interpolated in order for the text to be more clear (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);
```

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. This is due to the fact that we must explicitly enable support for blending so the alpha component has any effect. 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 complete our 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 our fragment shader for the HUD a little bit to detect if we have a texture or not. We will do this by using a uniform.

```
#version 330

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

uniform sampler2D texture_sampler;
uniform vec3 colour;
uniform int hasTexture;

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

In the `Hud` class we will create a new `GameItem` that loads the compass and add 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 often you will need to increase its size.

```
// Create compass
Mesh mesh = OBJLoader.loadMesh("/models/compass.obj");
Material material = new Material();
material.setColour(new Vector3f(1, 0, 0));
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 like coordinates, and 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 concepts let's go back to the text rendering approach we have presented here. The solution is very simple and useful to present some of 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 size is either to scale it, which may result in a poor quality rendered text, or to generate another texture file.
- The most important one, characters in most of the fonts do not occupy the same size and we 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 to generate 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 a texture by drawing each letter for a specified font family and size. That texture will be used in the same way as described above, 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;
        }
    }
```

Then we will create an image that will contain all the available characters. 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_ANTIALIAS_
    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 (yes, 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 cards and you could always achieve that adding some extra empty space). Actually, if after that block of code, you put a line like this:

```
ImageIO.write(img, IMAGE_FORMAT, new java.io.File("Temp.png"));
```

You will be able to view the image, which will be a long strip with all the available characters, drawn in white over transparent background using anti aliasing.

Then we just need to create a texture 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);
    }
```

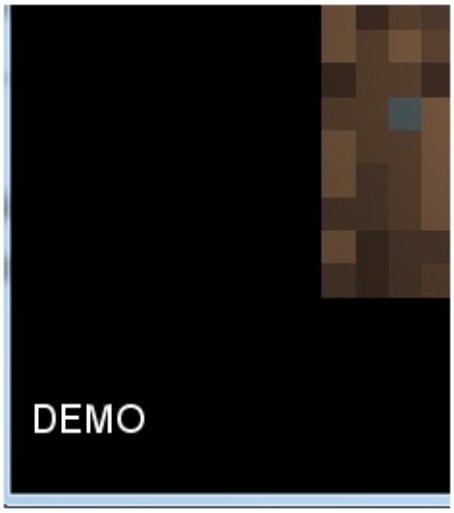
We have modified a little bit the `Texture` class to have another constructor that receives an `InputStream`. Now we just need to change the TextItem class to receive a `FontTexture` instance in its constructor.

```
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. What we will wget (for Arial font with a size of 20) is this:



As you can see the quality of the rendered text has been increased 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.

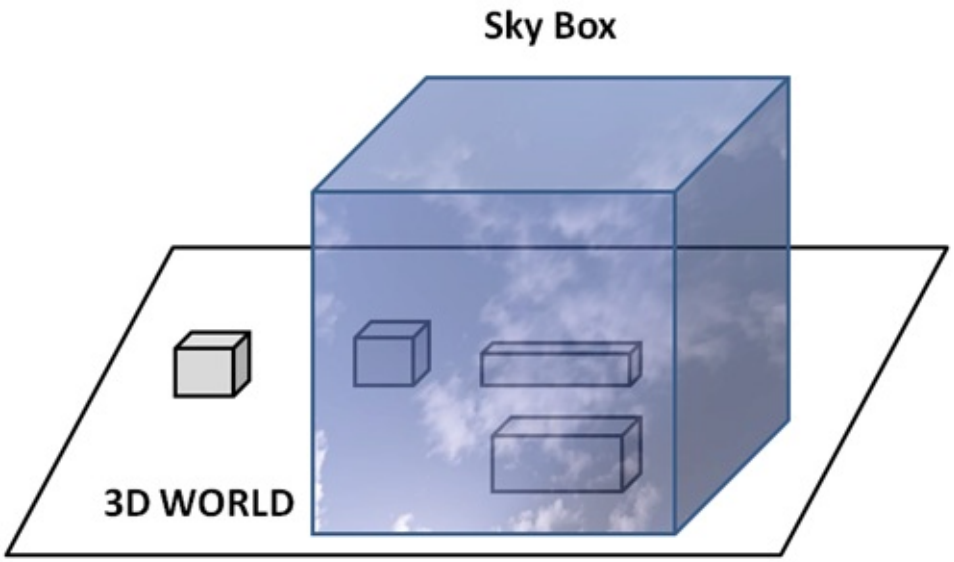
We have set up all the infrastructure needed in order to create a HUD for our games. Now it is just a matter of creating all the elements that represent relevant information to the user and give them a professional look and feel.

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 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.



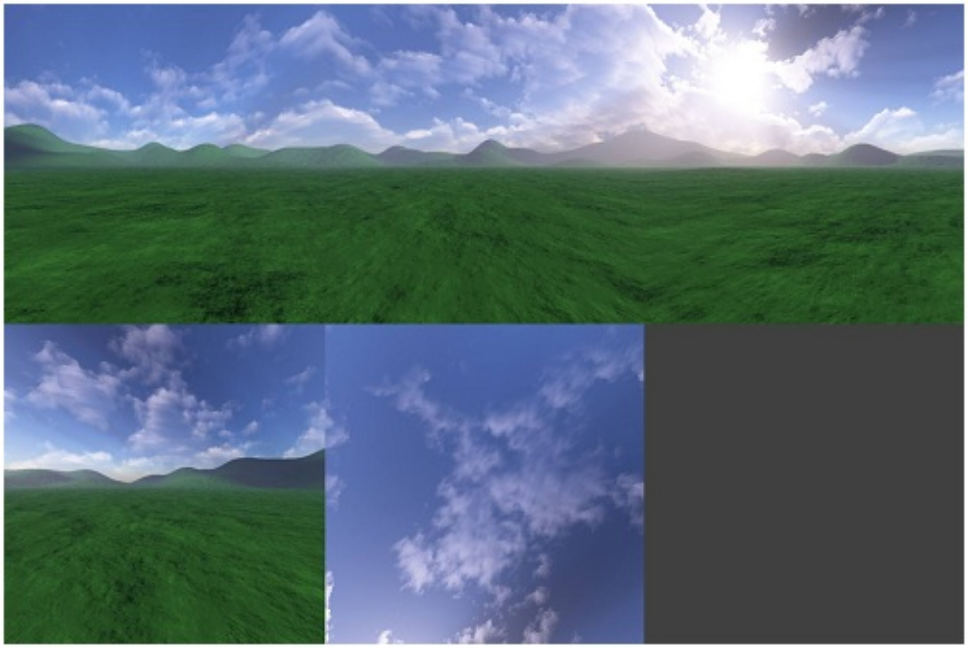
So the process of creating a sky box is basically as follows:

- 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 with the origin located at the centre of the camera.

Then, let's start with the texture. You will find that there are lots of textures pre-generated for you to use in the internet. The one used in the sample has been downloaded from here: http://www.custommapmakers.org/skyboxes.php. The concrete sample that we 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 those texture method but they will be explained in later chapters. The result image is something like this.



Then 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).

We will create 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 the definition and the attributes of the `Scene` class, that contains an instance of the `SkyBox` class.

```
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 a simplified version of those shaders, we will not be applying lights to the light box (or to be more precise, we don'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. 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 allow s 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 texture to simulate day and night (If not, the skybox will look like if were at 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.

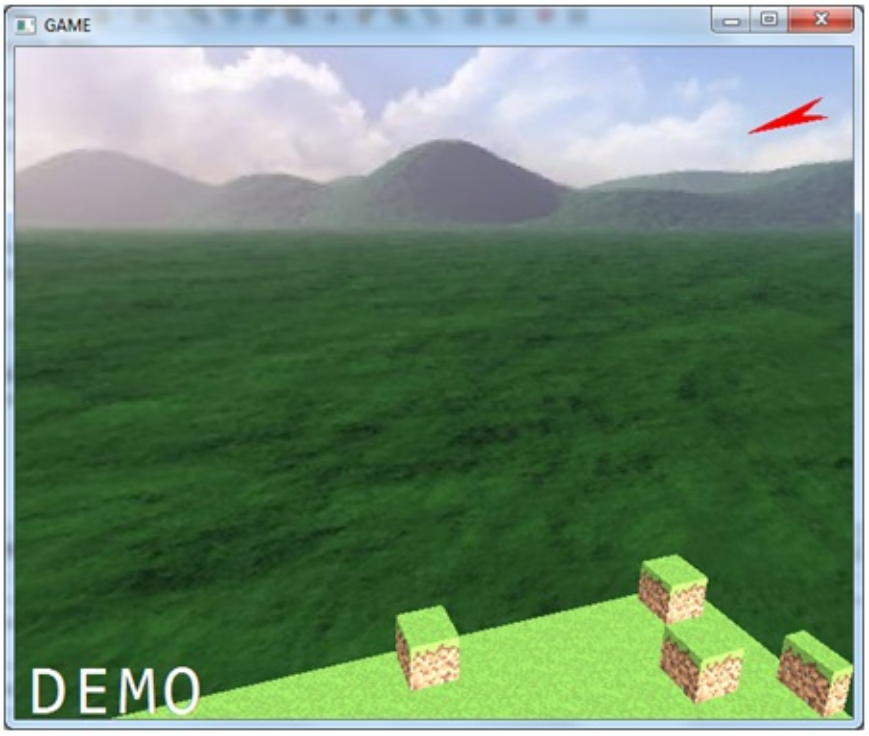
```
private void renderSkyBox(Window window, Camera camera, Scene scene) {
    skyBoxShaderProgram.bind();

    skyBoxShaderProgram.setUniform("texture_sampler", 0);

    // Update projection Matrix
    Matrix4f projectionMatrix = transformation.getProjectionMatrix(FOV, window.getWidth()
    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().getAmbientLight(

    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 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. But we do not want this, we want to stick it at the coordinates origin at (0, 0, 0). This achieved by removing 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 we get is something like this.

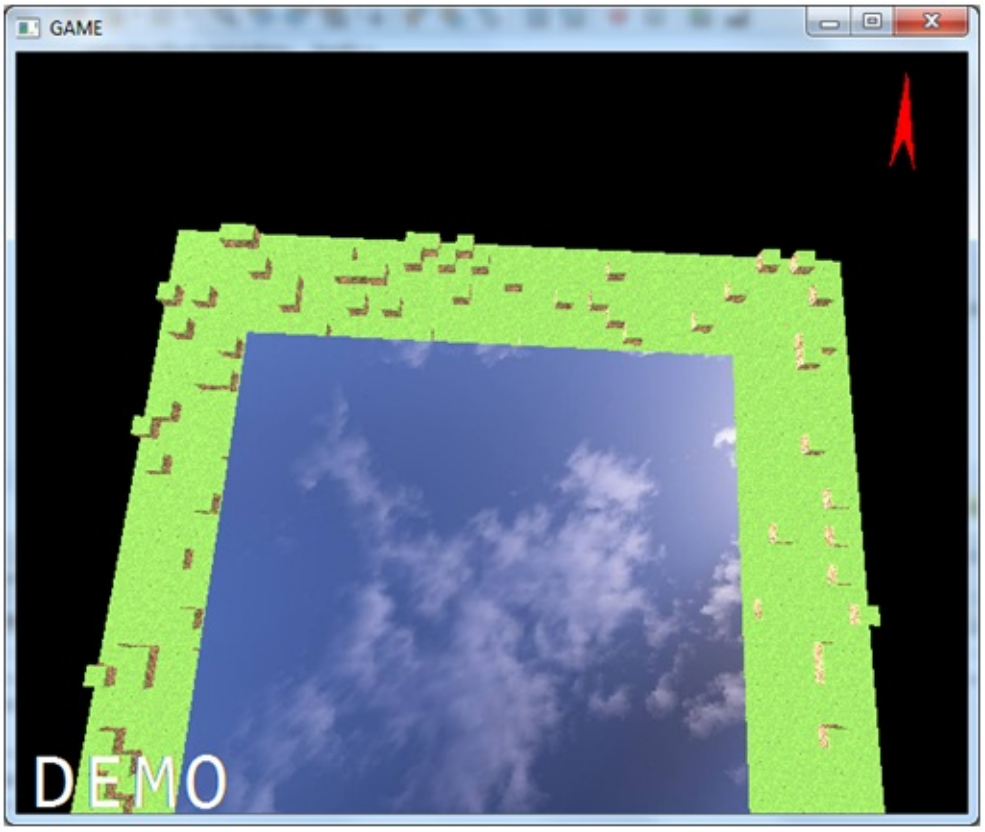


The sky box is a small one so can easily see the effect of moving through the world (in a real game it should be much bigger). You can see also that the world space objects, the blocks that form the terrain are larger than the skybox, so as you move through it you will see block

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 smooth 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 moving. Then you will be able to get out of the box and see something like this.



Although it is not what a sky box should do it can help you out to understand the sky box technique. This is a simple example, we will need to add other effects such as a sun moving through the sky or moving clouds. Also, in order to create bigger worlds we will need to split our world into fragments and only load the ones that are contiguous to the fragment where the player is currently in. We will try to introduce all these techniques later on.

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 from the hills). So, ok, let's increase the skybox size and thus let's increase the size of our world. Let's scale the size of our skybox by a factor of 50 so our 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 focus a little 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, we will explain face culling. In our examples we are rendered 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 in to a cube, you will see its interior like this.

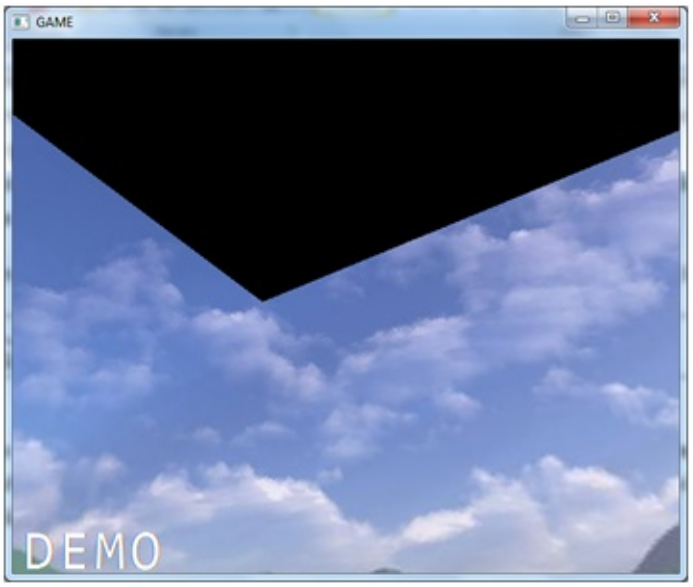
Faces that cannot be seen should be discarded immediately and this is what face culling does. In fact, for a cube you can only see 3 faces at the same time, so we can just discard half of the faces (40,000 *3* 2 triangles) just by applying face culling (this will only be valid if your game does not require 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 face 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.

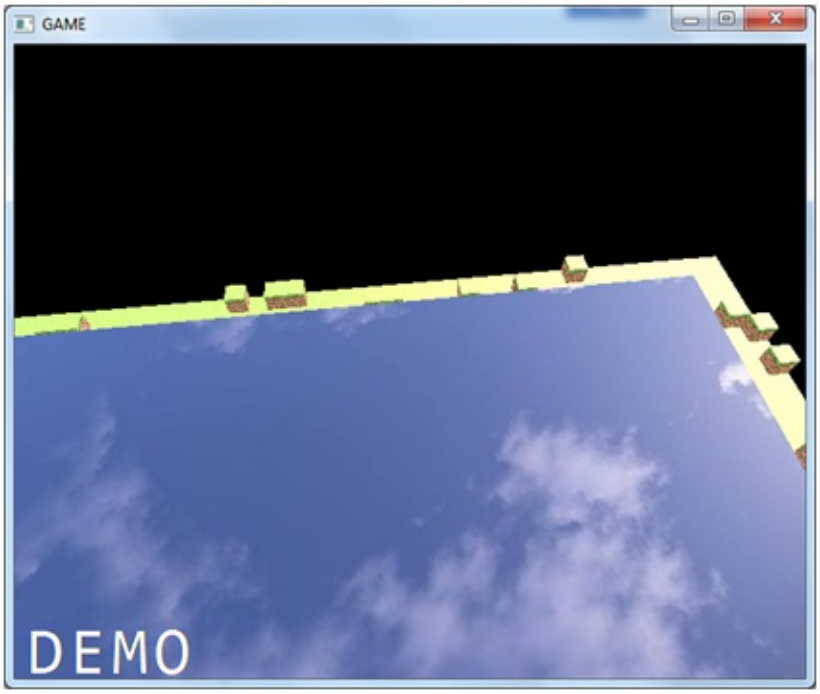
Remember from the first chapters that we may define of vertices in clockwise or counter-clockwise order. In OpenGL, by default, triangles that are in counter-clockwise order are facing towards the viewer and triangles that are in clockwise order are facing backwards. The key thing here, is that this order is checked while rendering and taking into consideration the point of view. So a triangle that has been defined in counter-clock wise order can be seen, at rendering, as clockwise 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);
```

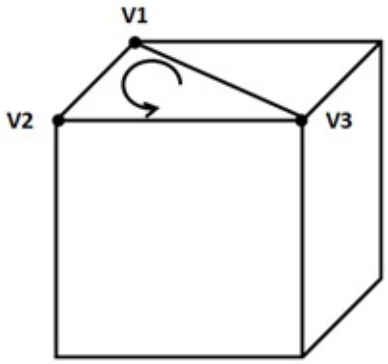
The first line will enable face culling and the second line states that faces that are facing backwards should be culled (removed). With that line if you look upwards you will see something like this.



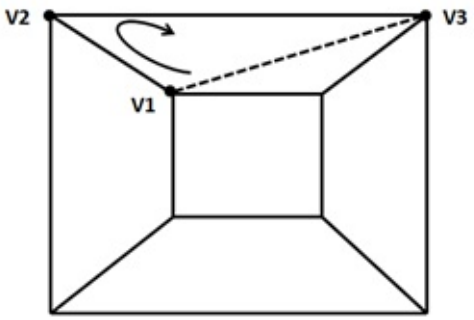
What's happening ? if you review the vertices order for the top face you will see that is has been defined in counter-clockwise order. Well, it was, but remember that the winding refers to the point of view. In fact, if you apply translation also to the skybox so you are able to see it form the upside 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 on 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.



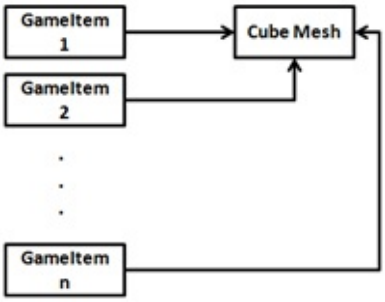
This is because, the skybox was defined to be looked from the outside. So we need to flip the definition for some of the faces in order to be viewed correctly and we will have face culling working properly.An if you get inside a cube you will see that inner sides are not shown.

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 `Gametem` array and render the associated Mesh. For each `GameItem` we do the following:

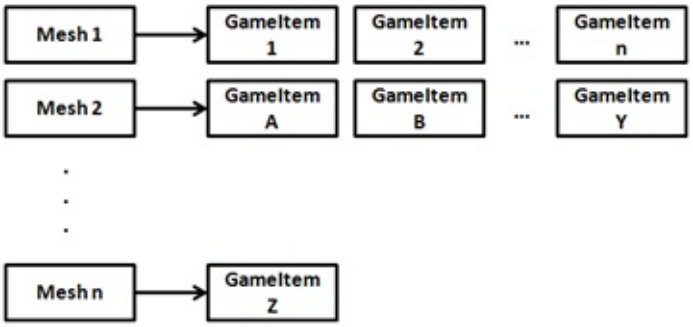
1. Set up the model view matrix (unique per `GameItem`).
2. Get the `Mesh` associated to the `GameItem` and activate the texture, bind the VAO and enable its attributes.
3. Perform a call to draw the triangles.
4. Disable the texture and the VAO elements.

But, in our current game, we reuse the same `Mesh` for the 40,000 GameItems, we are repeating some operations that have the same effect again and again. This is not very efficient, keep in mind that each call to a OpenGL function is 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 GamItems and we have split the activating and deactivating code into separate methods.

```
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 a the old method that renders the a `Mesh` taking into consideration that we have only one GameItem (this will may be used in the HUD, etc.). The new method renders a list of GameItems and receives as a paremeter a `Consumer` (a

function, this 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 this 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, viewMatr
        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 are 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.

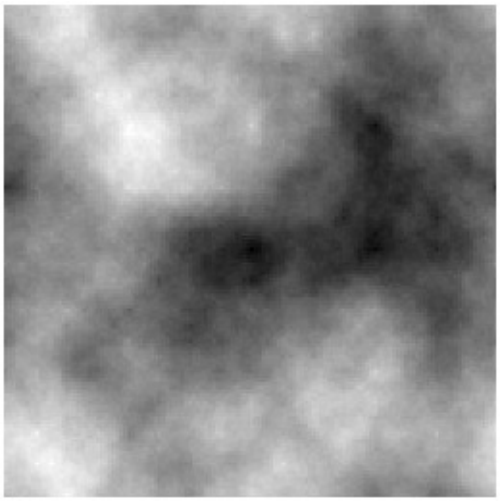
We have also avoided the construction of new `FloatBuffer` instances each time we set a uniform for a Matrix and removed some other useless instantiations. With all that in place you can see now that the rendering is smoother and more agile.

You can check all the details in the source code.

Height Maps

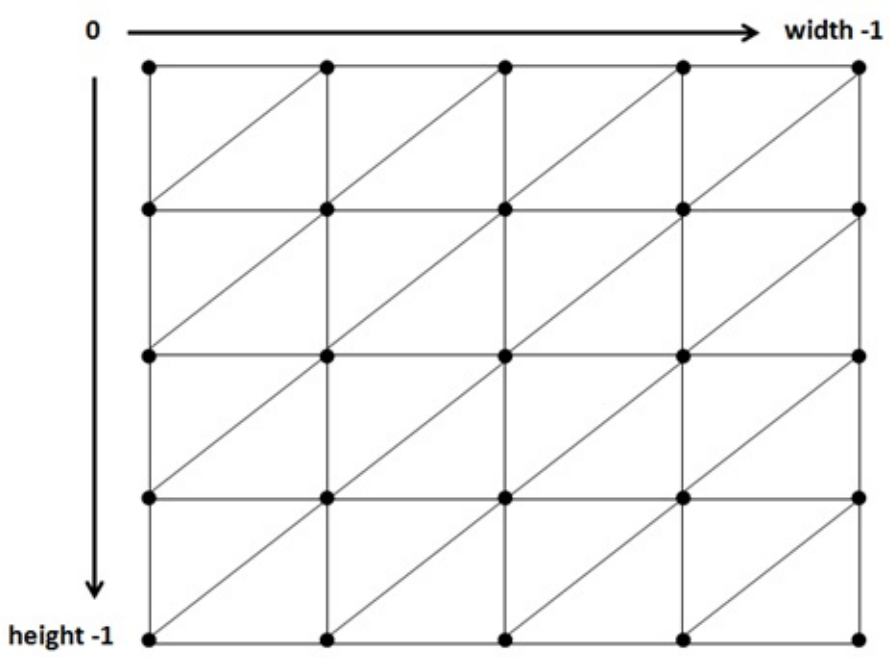
In this chapter we will learn how to create complex terrain using height maps. Before we start, we have done some refactoring, we added a few packages and moved some of the classes to better organize them. You can check the changes in the source code.

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.

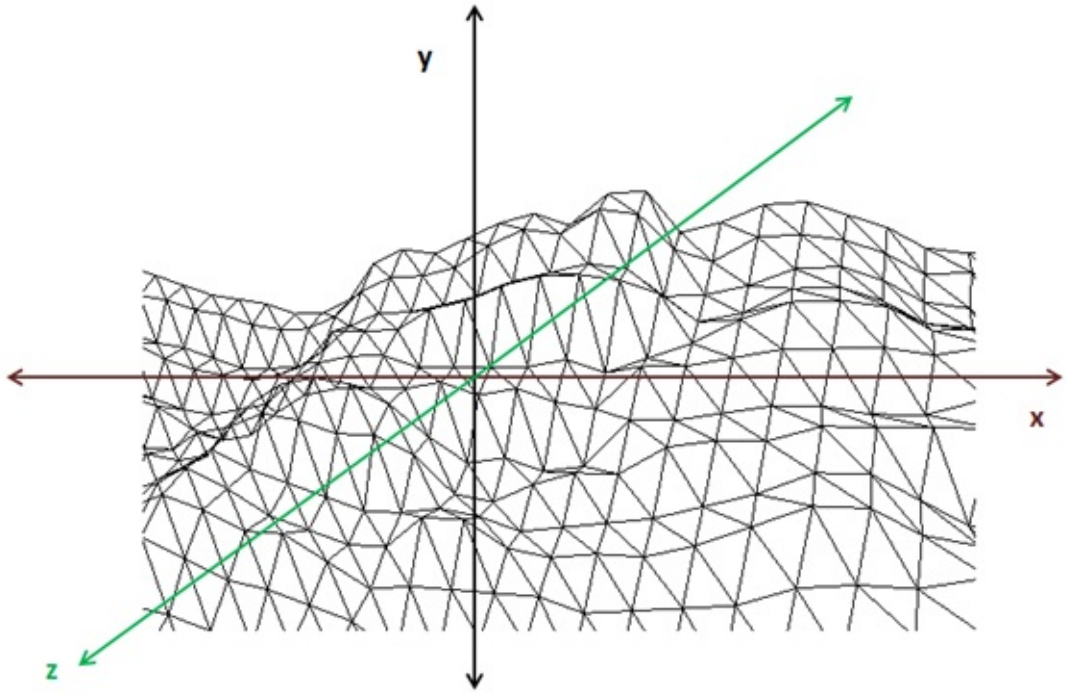


The image above it's like if you were looking at a fragment of land from above. With that image we will build a mesh composed by triangles formed by vertices. The altitude of each vertex will be calculated depending on the colour of each of the image pixels. Black colour will represent the lowest value 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.



That mesh will form a giant quad that will be rendered across x and z axis using the data contained in the height map to change the elevation in the y axis.



The process of creating a 3D terrain from a height map will be 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 which heights will be 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.

We will create a class named `HeightMapMesh` that will create a `Mesh` based on a height map 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 or not, 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. In this case, (we will not be limited to use 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) which x and z coordinates will be in the range of -0.5 (STARTX) to 0.5 for x axis and -0.5 (STARTZ) to 0.5 for z axis. 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. 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, in
```

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`.

```
this.minY = minY;
this.maxY = maxY;

BufferedImage buffImage = ImageIO.read(getClass().getResourceAsStream(heightMapFile));
int height = buffImage.getHeight();
int width = buffImage.getWidth();
```

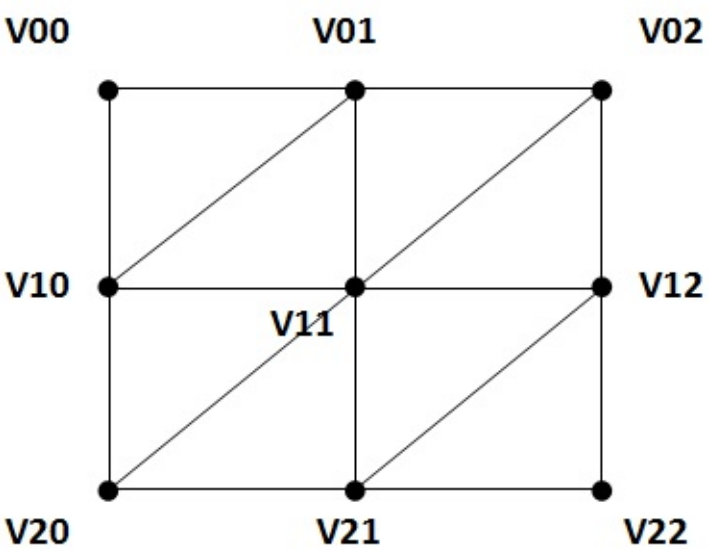
Then we load the texture file and setup the variables that we will need to construct the `Mesh`. The `incx` and `incz` variables will have the increment to be applied to each vertex in the x and z coordinates so the Mesh covers the range [STARTX, -STARTX] and [STARTZ, -STARTZ].

```
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();
```

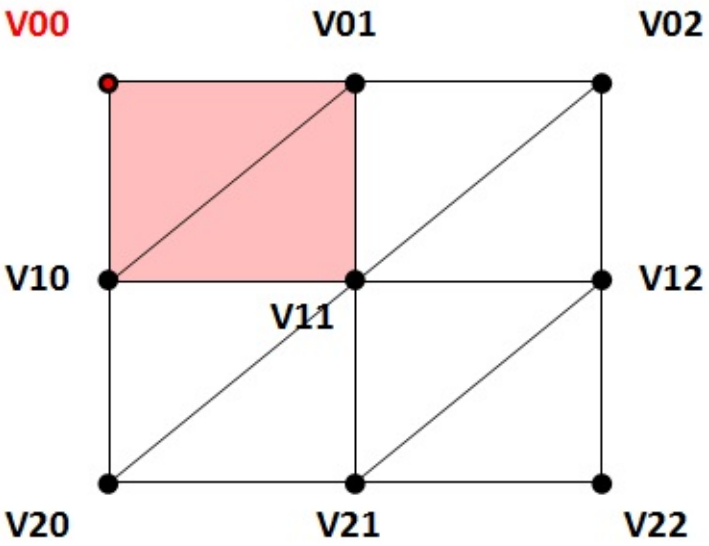
Then we are ready to iterate over the image, creating a vertex per each pixel, setting its texture coordinates and setting up the indices to define correctly the triangles that compose the `Mesh`.

```
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, buffImage)); //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);
        }
    }
}
```

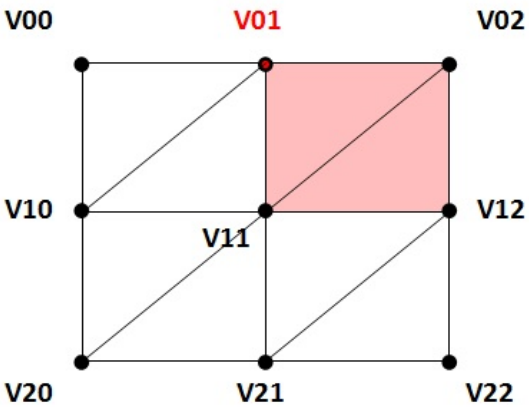
The process of creating the vertex coordinates is self explanatory, let's ignore at this moment why we multiply the texture coordinates by a number and how the height is calculated. You can see that for each vertex we define the indices of two triangles except if we are in the last row or column. Let's review with a 3x3 image to visualize how they are constructed. A 3x3 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).

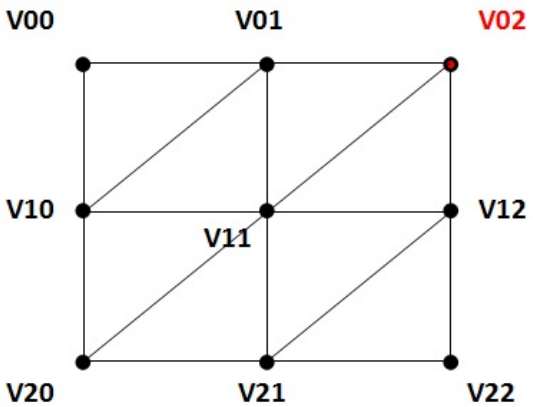


When we are processing the first vertex (V00), we define the indices of the two triangles shaded in red.



When we define the second vertex (V01),), we define the indices of the two triangles shaded in red, but when we define the third vertex (V02) we do not need to define more indices, the triangles have already been defined.





You can easily see how the process continues for the rest of vertices. Now, once we have created all the vertex 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);
```

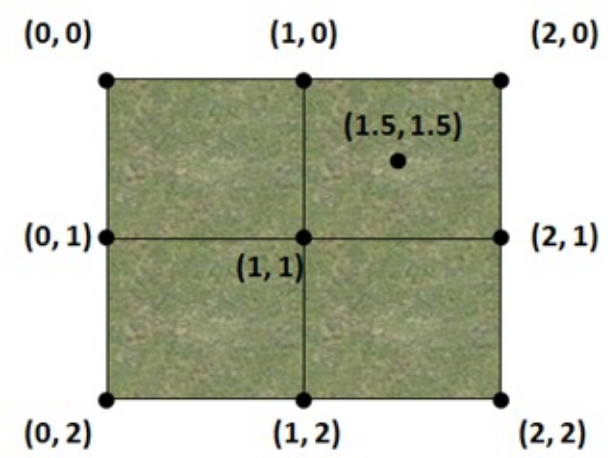
You can see that we calculate the normals taking as an input the vertex positions. Before we see how normals can be calculated, let's see how heights are obtained. We have created a method named getHeight which calculates the height for a vertex.

```
private float getHeight(int x, int z, BufferedImage buffImage) {
    float result = 0;
    if (x >= 0 && x < buffImage.getWidth() && z >= 0 && z < buffImage.getHeight()) {
        int rgb = buffImage.getRGB(x, z);
        result = this.minY + Math.abs(this.maxY - this.minY) * ((float) rgb / (float) MAX
    }
    return result;
}
```

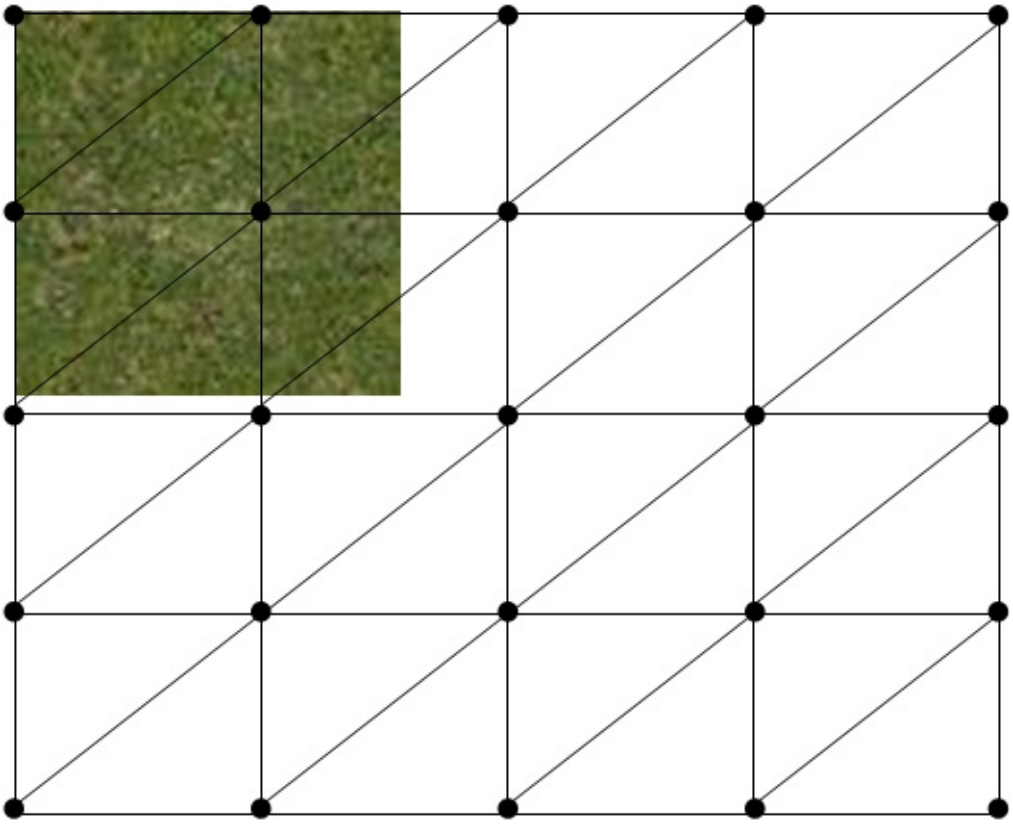
The method receives the x an z coordinates for a pixel, gets the RGB colour (the sum of the individual R, G and B components) and assigns a value contained between `minY` and `maxY` (`minY` for black colour and `maxY` for white colour).

Let's review 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 e vary efficient technique that is based that if we set texture coordinates that are beyond (1,1), we get back to origin and start counting again. The following picture shows this behavior.

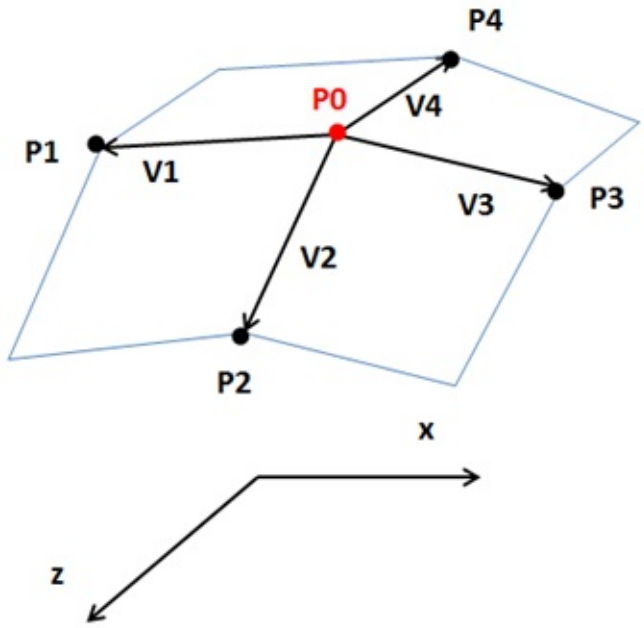


This is what we are doing when calculating the texture coordinates, we are multiplying the texture coordinates (calculated as if the texture just was wrapped covering the whole mesh) by a factor, the `textInc` parameter, to increase the number of pixels of the texture to be used between adjacent vertices.

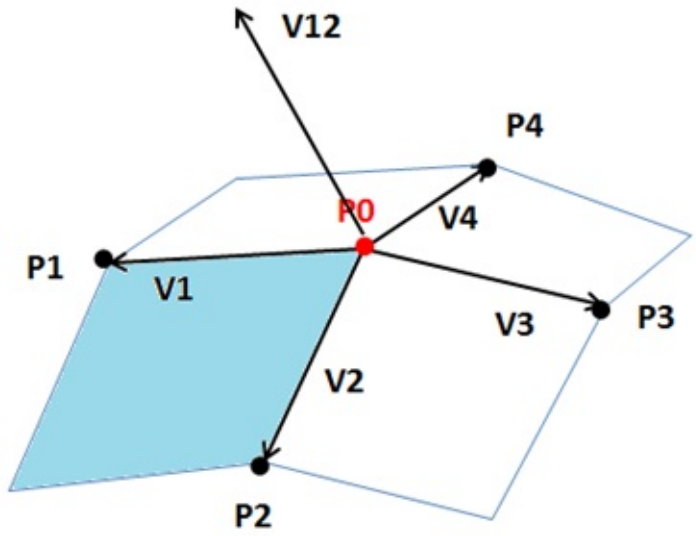


The only thing that's pending is how to calculate normals. Remember that we need normals so light can be applied to the terrain. Without normals our terrain will be rendered with the same colour no matter how light hits each point. The method that we will use here may not be the most efficient for height maps but it will help you understand how normals can be auto-calculated. If you search for other solutions you may find other approaches that only use the heights of adjacent points without performing cross product operations and are more efficient. Nevertheless since this will only be done at startup, the method resented here will not hurt performance so much.

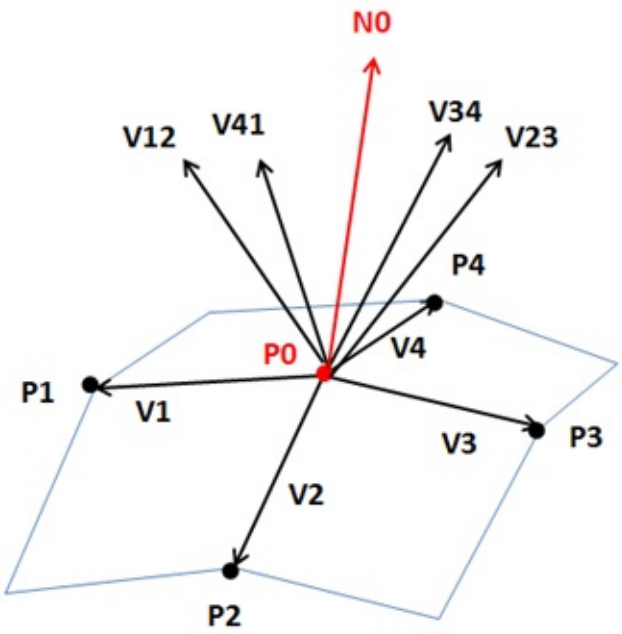
Let's graphically explain how the normal will be calculated. Imagine that we have a vertex named $P0$. We first calculate for each of the surrounding vertices ($P1$, $P02$, $P3$ and $P4$) the vectors that it's tangent to the surface that connects those points. These vectors ($V1$, $V2$, $V3$ and $V4$) 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 vector. For instance, the normal of the surface that connects $P1$ and $P2$ (shaded in blue) is calculated as the dot product between $P1$ and $P2$, $V12 = P1 \times P2$.



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();
    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:

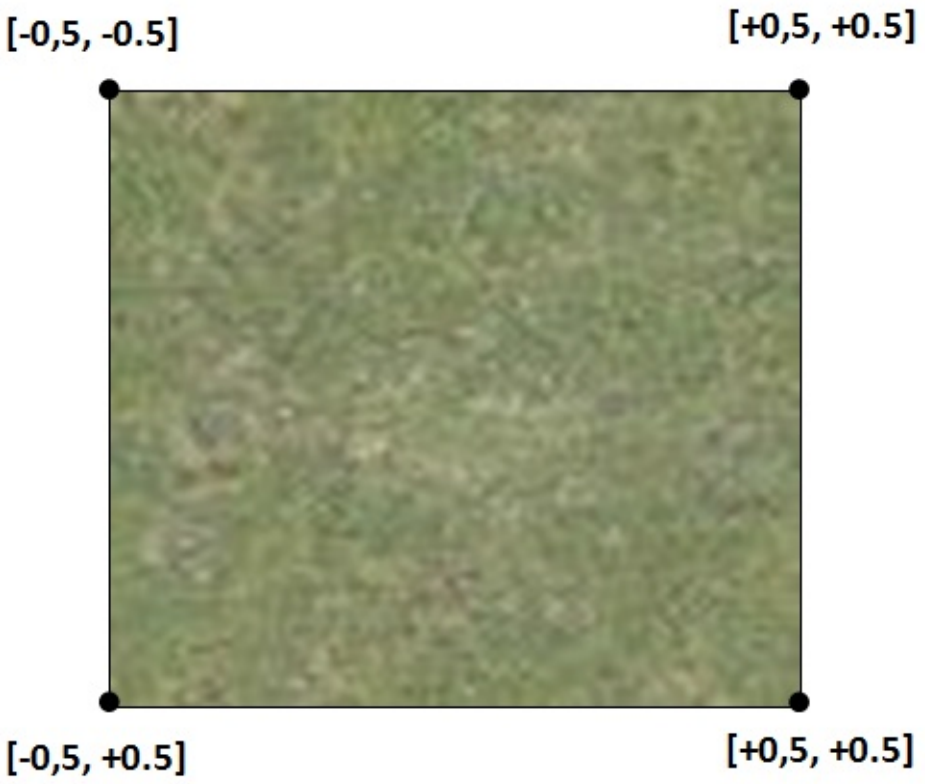
- Create a larger height map.
- Reuse a height map and tile it through the 3D space. The height map will be like a

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). We will create a class named `Terrain` that will create a square of height map tiles, defined like this.

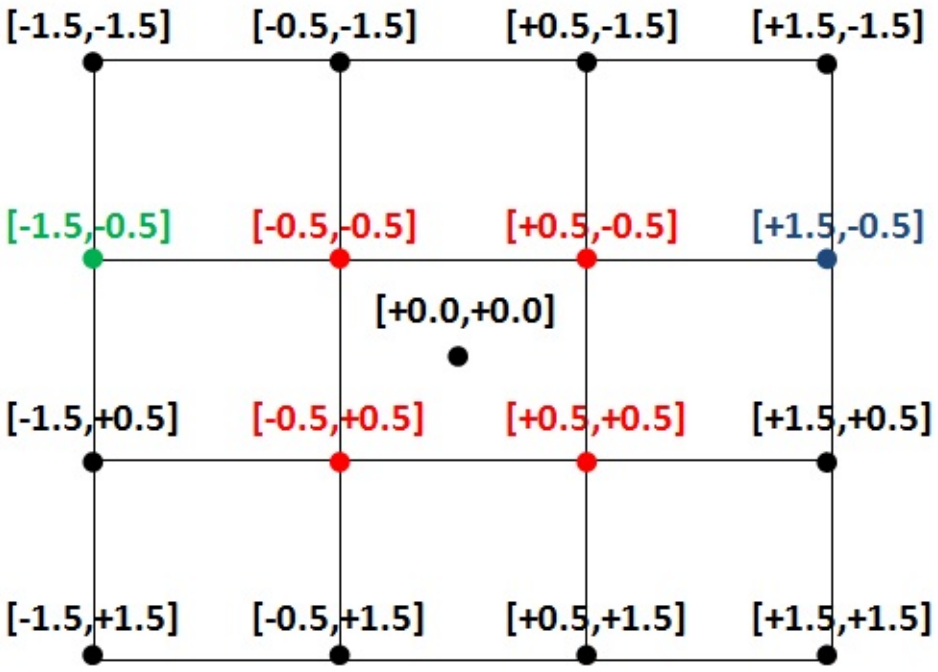
```
package org.lwjglb.engine.items;

import org.lwjglb.engine.graph.HeightMapMesh;

public class Terrain {

    private final GameItem[] gameItems;

    public Terrain(int blocksPerRow, float scale, float minY, float maxY, String heightMa
        gameItems = new GameItem[blocksPerRow * blocksPerRow];
        HeightMapMesh heightMapMesh = new HeightMapMesh(minY, maxY, heightMap, textureFil
        for (int row = 0; row < blocksPerRow; row++) {
            for (int col = 0; col < blocksPerRow; col++) {
                float xDisplacement = (col - ((float) blocksPerRow - 1) / (float) 2) * sc
                float zDisplacement = (row - ((float) blocksPerRow - 1) / (float) 2) * sc

                GameItem terrainBlock = new GameItem(heightMapMesh.getMesh());
                terrainBlock.setScale(scale);
                terrainBlock.setPosition(xDisplacement, 0, zDisplacement);
                gameItems[row * blocksPerRow + col] = terrainBlock;
            }
        }
    }

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

We will explain the overall process, we have blocks that have the following coordinates (for x and z and with the constants defined above).



Let's create a terrain formed by a 3x3 grid and that we wont' scale the terrain blocks (taht is, the variable `blocksPerRow` will have a 3 and the variable `scale` will have a 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 done by calling setPosition, buy remember 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 adquate coordinates. The vertex

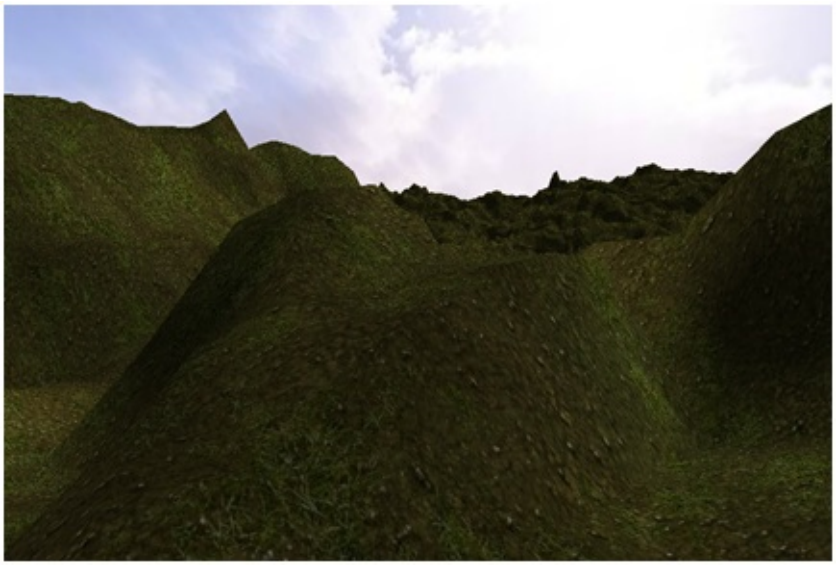
drain in green needs a displacement, for the x coordinate, of -1 and the vertex drawn in blue needs a displacement of $+1$. The formula to calculate the x displacement, taking into consideration the scale and the block width, is this one:

$$xDisplacement = (col - (blocksPerRow - 1)/2) \times scale \times width$$

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 above.

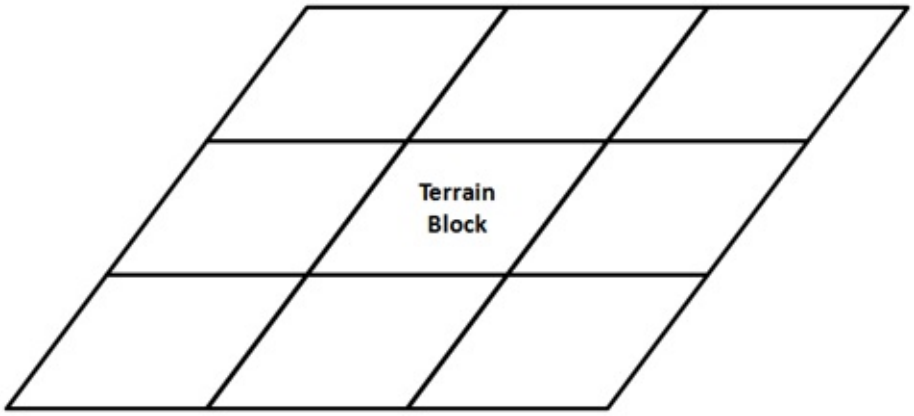
Terrain Collisions

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 what our current position is. Since we do not have a player concept yet 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 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 do those calculations in the `Terrain` class constructor. By doing this way we access them later at any time without repeating those operations in eacha game loop cycle.

We will create a new method that calculates the bounding box of a terrain block, named `getBoundingBox`.

```
private Rectangle2D.Float 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;
    Rectangle2D.Float boundingBox = new Rectangle2D.Float(topLeftX, topLeftZ, width, heig
    return boundingBox;
}
```

We need to calculate the world coordinates of our terrain block. In the previous chapter you saw that all of our terrain meshes were created inside a quad with its origin set to [STARTX, STARTZ], so we need to transform that 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 is done in the Terrain class constructor, so we need to add a new attribute which will hold the bounding boxes:

```
private final Rectangle2D.Float[][] 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 heightMapFile
    this.terrainSize = terrainSize;
    gameItems = new GameItem[terrainSize * terrainSize];

    BufferedImage heightMapImage = ImageIO.read(getClass().getResourceAsStream(heightMapF
    // The number of vertices per column and row
    verticesPerCol = heightMapImage.getWidth();
    verticesPerRow = heightMapImage.getHeight();

    heightMapMesh = new HeightMapMesh(minY, maxY, heightMapImage, textureFile, textInc);
    boundingBoxes = new Rectangle2D.Float[terrainSize][terrainSize];
    for (int row = 0; row < terrainSize; row++) {
        for (int col = 0; col < terrainSize; col++) {
            float xDisplacement = (col - ((float) terrainSize - 1) / (float) 2) * scale *
            float zDisplacement = (row - ((float) terrainSize - 1) / (float) 2) * scale *

            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.

```
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
    Rectangle2D.Float 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, pos
    }

    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 `Rectangle2D` already 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, cinluding their heights. But, we need the height of apoint that is not located at any of those vertices but in a point 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 (Altogh 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

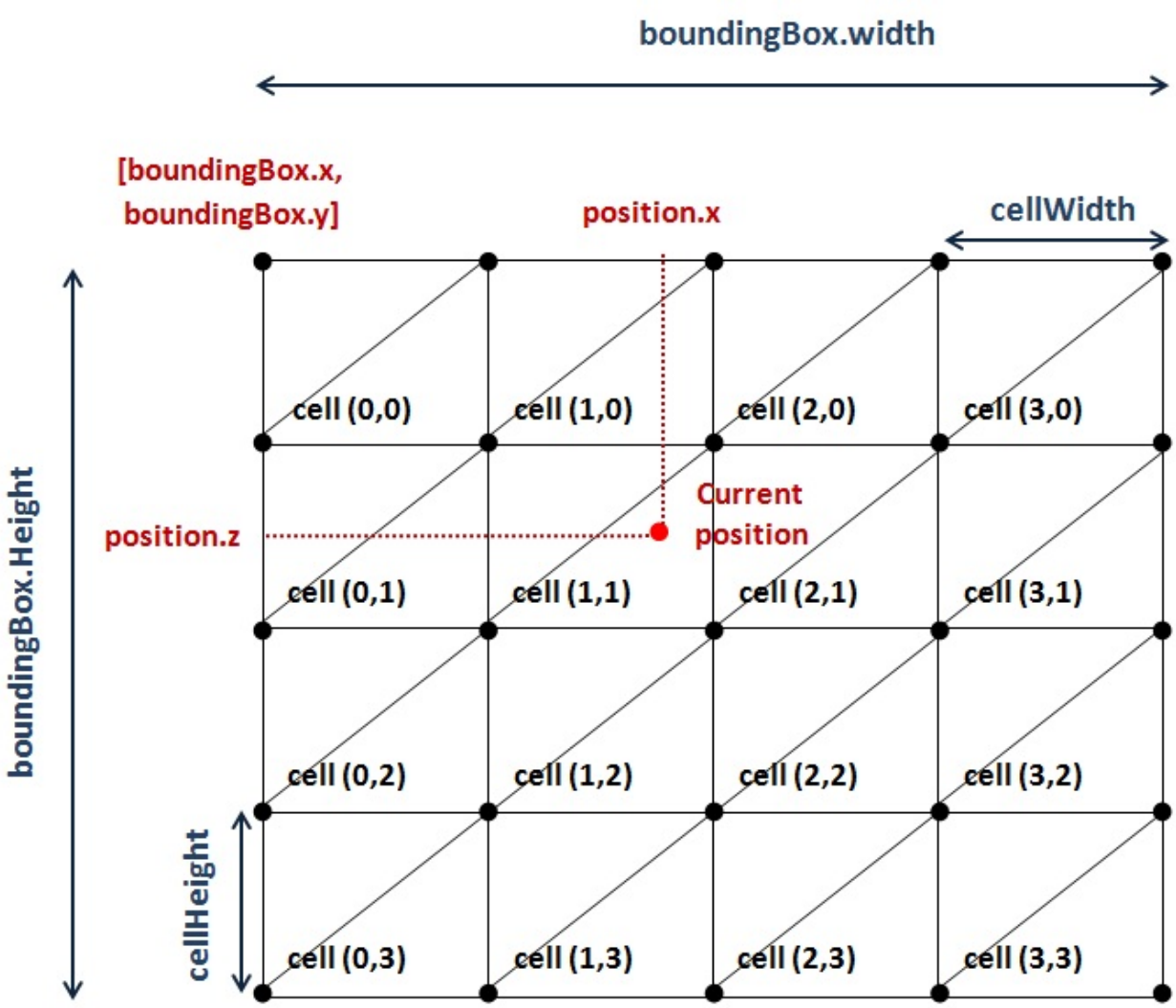
$$cellHeight = \frac{boundingBox.height}{verticesPerRow}$$

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:

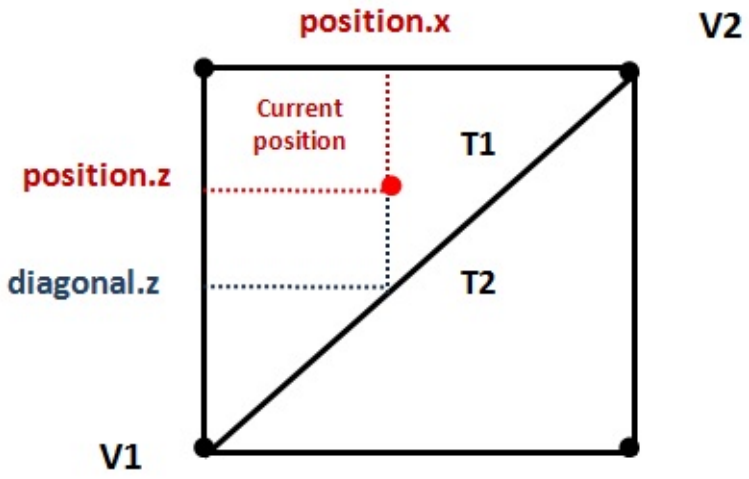
$$col = \frac{position.x - boundingBox.x}{boundingBox.width}$$

$$row = \frac{position.z - boundingBox.y}{boundingBox.height}$$

The following picture shows all the variables described above for a sample terrain block.



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 its greater than that we are in T2.

We can determine that by calculating the line equation that matches the diagonal.

If you rememeber 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 coodrinates 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:

```
protected Vector3f[] getTriangle(Vector3f position, Rectangle2D.Float boundingBox, GameIt
    // 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, trian
        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 precalculated 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 hieght 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:

```
// Update camera position
Vector3f prevPos = new Vector3f(camera.getPosition());
camera.movePosition(cameraInc.x * CAMERA_POS_STEP, cameraInc.y * CAMERA_POS_STEP, cameraI
// 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 undesrtand 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 avñue can get so hight 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, that added will sum up the desired final position displacement.

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 firs examine what are the attributes that define the fog effect. 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 next one is the density of the fog.

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, 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.

So the first thing that we need to do is to model how it affects the objects contained in our 3D scene. We know that objects closer to the camera will not be affected and that objects that are too far away will not be distinguishable. We need to calculate a factor that is dependent on the distance and that can be used to calculate the final colour.

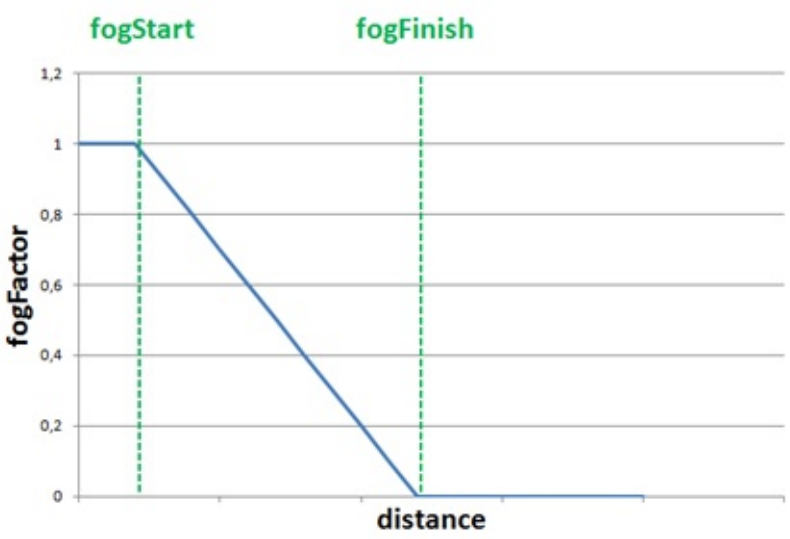
The first option is to chose a linear model. In this model we should define the following parameters:

- $fogStart$: 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:

$$fogFactor = \frac{(fogFinish - distance)}{(fogFinish - fogStart)}$$

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.



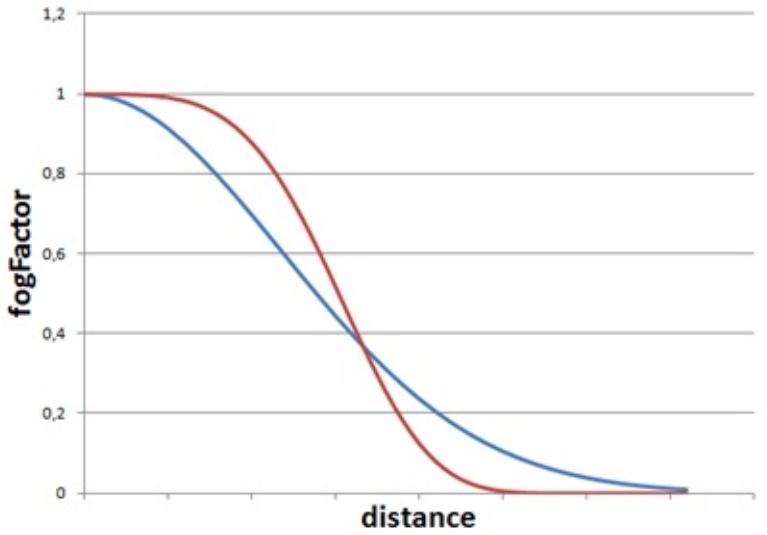
The previous model is easy to calculate but it is not very realistic and it does not take into consideration the fog density. In reality fog tends to grow in smoother way. So the next suitable model is a exponential one. The equation for that model is as follows:

$$focFactor = e^{-(distance \cdot fogDensity)^{exponent}} = \frac{1}{e^{(distance \cdot fogDensity)^{exponent}}}$$

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 that will be handled in the Java code, which is another POJO which contaings the fog attributes.

```
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 defines 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 + ".active", 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 calFog 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, 1);
}
```

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 o and 1 and use the mix function in GLSL which is used to blend the fog colour and the fragment colour (defined by variable colour). It's equivalent to apply this equation:

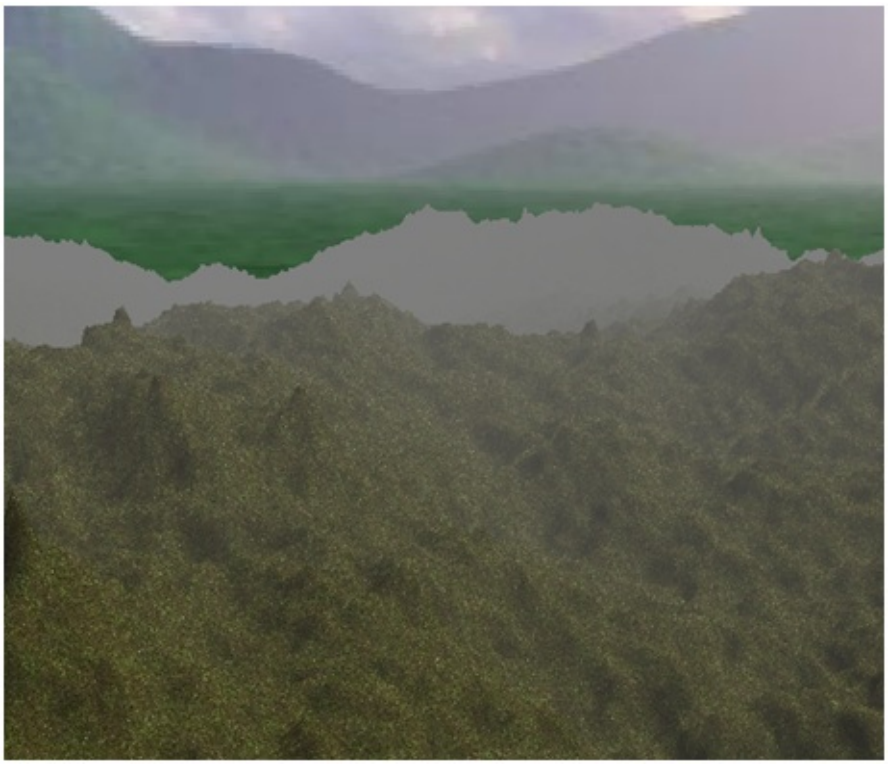
$$resultColour = (1 - fogFactor) \cdot fog.colour + fogFactor \cdot colour$$

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.active == 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));

And we will get an effect like this:

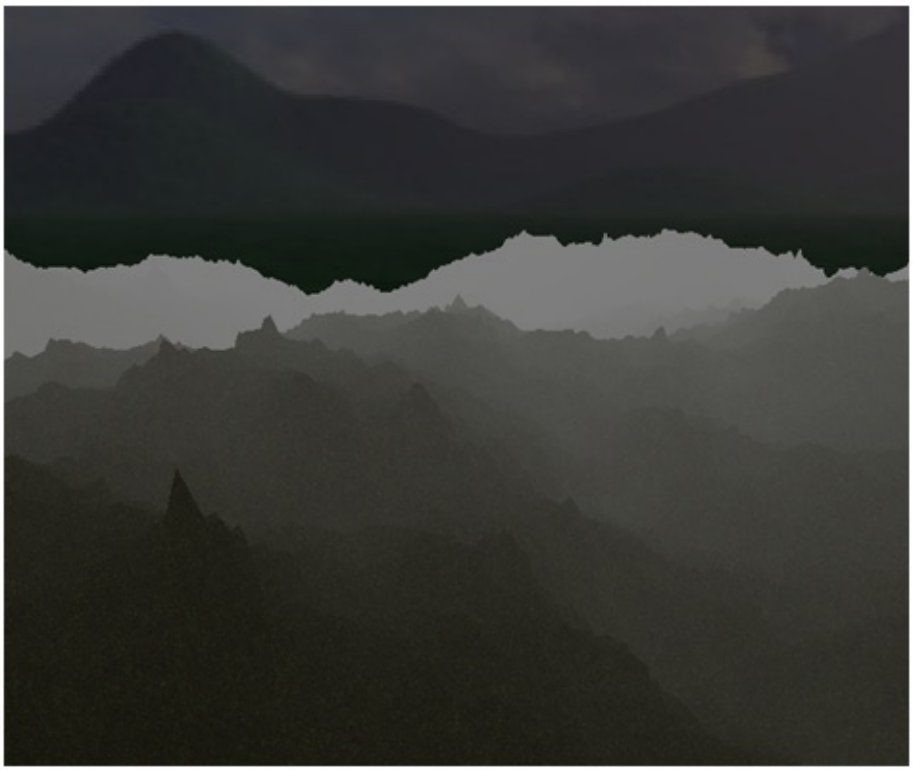


You will see that distant objects get faded in the distance and that fog starts to disappear when you approach to them. There’s a problem, though with the skybox, it looks a little bit weird that the horizon is not affected by the fog. There are several ways to solve this:

- 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.



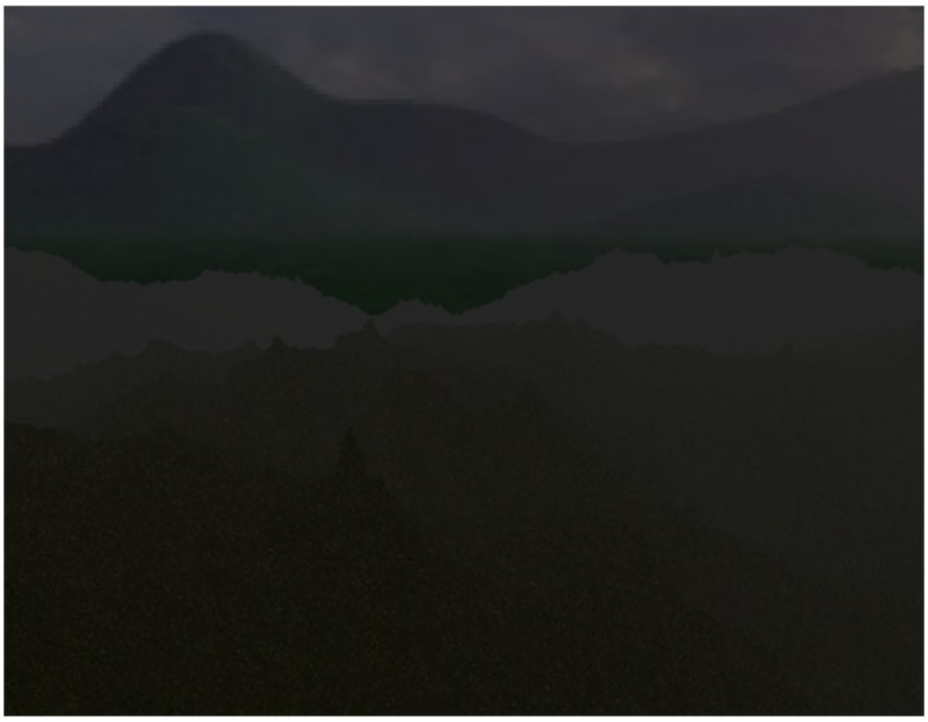
Distant objects are set to the fig colour which is a constants and it produces like a glowing in the dark effect (which may be ok for you or not). We need to change the funcion that calculates the fog to take into consideration the light. The function will receive the ambient light and the directional light to modulate the fog colour.

```
vec4 calcFog(vec3 pos, vec4 colour, Fog fog, vec3 ambientLight, DirectionalLight dirLight
{
    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);
}
```

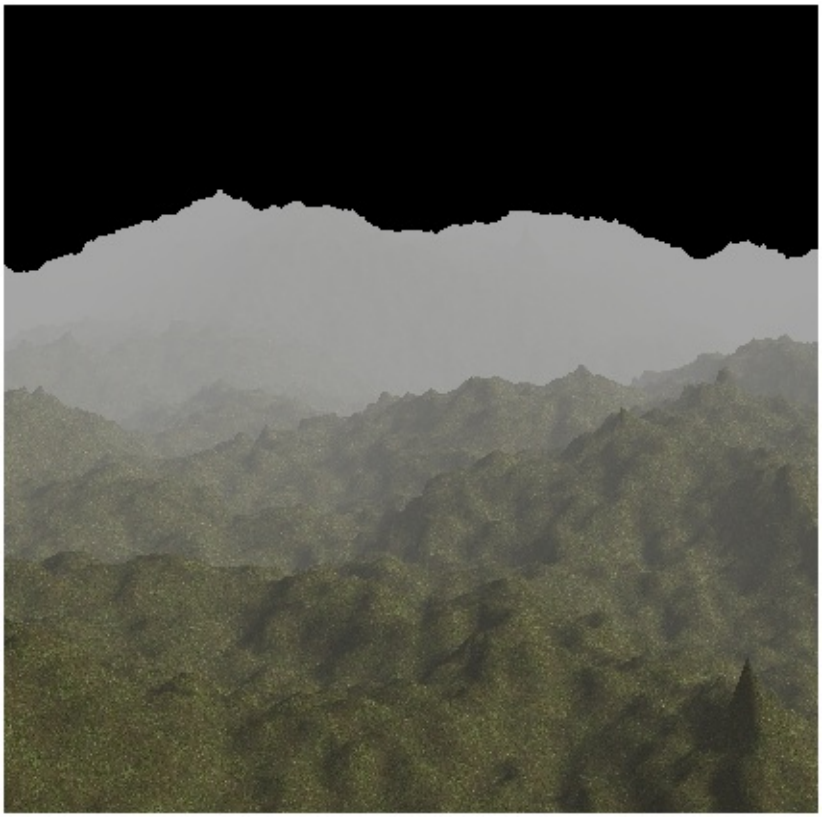
As you can see with the directional light we just use the colour and the intensity, we are not interested in the direction. With that modification we just need to slightly modify the call to the function like this:

```
if ( fog.active == 1 )
{
    fragColor = calcFog(mvVertexPos, fragColor, fog, ambientLight, directionalLight);
}
```

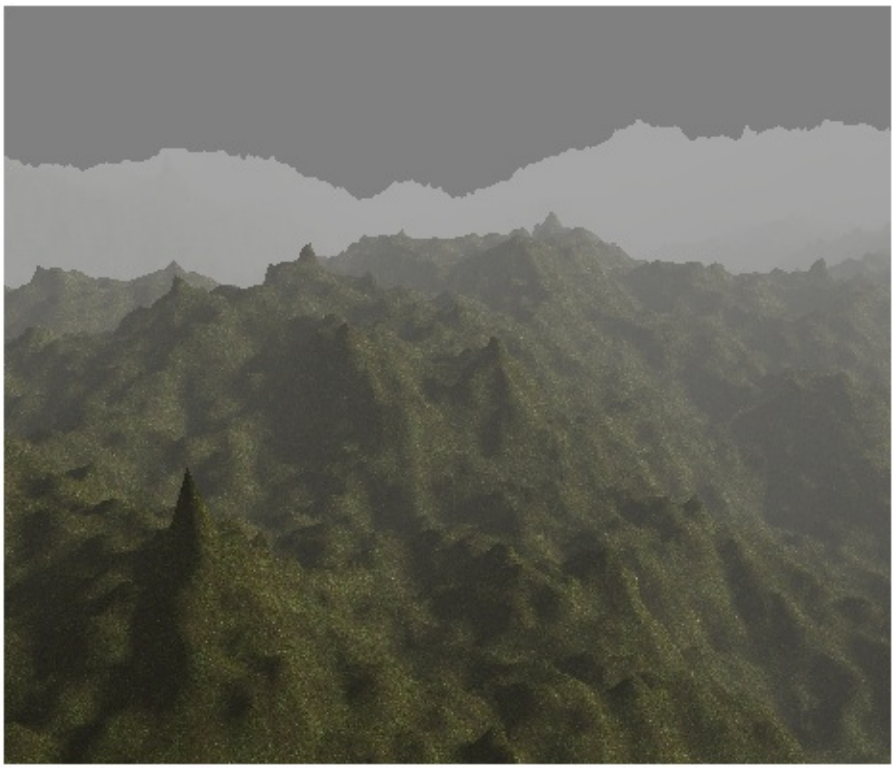
And we will get something like this when the night falls.



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.



Shadows

Animations

Next to come (in a near future)