Report: computer graphics

Introduction

The framework used to leverage OpenGL in our Java prjoect is LWJGL. This stands for Ligt-Weight Java OpenGL-Library. Lwjgl is java wrapper for 3 native libraries namely, OpengGL, OpenAL (Audio) and GLFW (windowing with OpenGL context, input and events). Supported platforms are Linux, Windows and Mac all with their respective native libraries located in the folder “native” in the class path. Those are set by the use of:

```
private static void setNatives() {
        if (System.getProperty("org.lwjgl.librarypath") == null) {
            Path path = Paths.get("native");
            String librarypath = path.toAbsolutePath().toString();
            System.out.println(librarypath);
            System.setProperty("org.lwjgl.librarypath", librarypath);
        }
    }
```

To run the project on Mac OSX the VM option *-XstartOnFirstThread* has to be added. See the website of LWJGL for more details. Remove this flag if you want to run it on any other platform.

Initialization

GLFW

Once the native libraries are referenced, calls can be made from the Java code to the several libraries. As most (if not all) of those libraries are compiled c(++) code the interaction is heavily based on pointers, a concept rarely used explicitly in Java. The first things we do is setting up the callbacks for different events like errors, window events, keyboard events, …

```
// Setup an error callback. The default implementation
// will print the error message in System.err.
glfwSetErrorCallback(errorCallback = errorCallbackPrint(System.err));
…

// Setup a key callback. It will be called every time a key is pressed, repeated or released.
glfwSetKeyCallback(window, keyCallback = new GLFWKeyCallback() {
            @Override
            public void invoke(long window, int key, int scancode, int action, int mods) {
                if (key == GLFW_KEY_ESCAPE && action == GLFW_RELEASE) {
                    glfwSetWindowShouldClose(window, GL_TRUE);
            // We will detect this in our rendering loop
```

```
        }
        boolean on;
        if (action == GLFW_PRESS) {
            on = true;
...
```

All those callbacks have a designated Java wrapper class that can subclassed or overridden to handle the events properly. Another part is the setting of environment parameters, done by using constant int values.

```
glfwDefaultWindowHints(); // optional, the current window hints are already the default
glfwWindowHint(GLFW_VISIBLE, GL_FALSE); // the window will stay hidden after creation
glfwWindowHint(GLFW_RESIZABLE, GL_TRUE); // the window will be resizable
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
```

The last 4 parameters set are specifically for OpenGL. They determine the version and compatibility of the running OpenGL. In our case OpenGL 3.2 supporting the core profile and forward compatible with higher (newer) versions.

Once everything is set correctly we take the pointer of our window

```
// Create the window
int window = glfwCreateWindow(windowWidth, windowHeight, "Hello World!", NULL, NULL);
```

and “bind” the pointer to our OpenGL context.

```
glfwMakeContextCurrent(window);
```

Now we have a GLFW window. The next part of initialization is for rendering our 3D objects in this window using OpenGL. The last thing we will discuss from our main class Doom, is the mandatory destruction of all our allocated memory in the running native code (callbacks and window). This is handled in 2 ways by LWJGL. One is implicitly by calling a release method on the callback that implements the freeing of the memory, another is an explicit destroy call with (integer) pointer to GLFW. Throughout our code (especially in *Game.exit()*) this will have to be done for every piece of allocated memory saved as an integer pointer.

```
glfwDestroyWindow(window);
keyCallback.release();
```

OpenGL

Initialization of OpenGL starts with GLContext.createFromCurrent(). Which makes the bindings of OpenGL available in the current thread and is critical for the interoperation between GLFW and OpenGL. We create a vertex array and vertex buffer objects to store our data that we'll render later on. These objects are allocated memory and again we use pointers to reference them, in our project we use a custom wrapper class for both the array and the buffer containing the respective allocation,

generation and data upload calls.

```
//memory alloc and pointer generation
int vertexPointer = glGenVertexArrays();
int bufferPointer = glGenBuffers();
```

The data is edited, uploaded, from the Java context to the pointer location in the binaries through the use of buffer objects (float, int or byte – buffer).

```
glBufferData(pointer, data, usage);
```

Once all necessary memory has been allocated, bound to OpenGL and the pointers saved, we can start to fill the memory with our data. Our data is abstracted on a first level as a list of cubes. All that is rendered is this list of cubes called “scene”. A cubes itself is nothing more than an array of vertices. In OpenGL everything is rendered by triangles, uniquely defined by 3 vertices. Thus our cube consists of 36 (6 square faces, 2 triangles for a face) virtual vertices (elements). The reason it's 36 elements is because there are only 24 unique vertices, as some vertices are shared between the 2 triangles forming a square face and are thus, for memory and performance reasons, not duplicated but appointed multiple times. In the class Cube the vertices are generated based on position, size and color. Each vertex contains 3 floats for position (x,y,z), 3 floats for color (R,G,B), 2 floats for the texture coordinate (u,v) and finally 3 floats for the normal of the face (used for lighting).

The rendering order mentioned earlier (elements) of the triangle's vertices in a cube is given by the static variable vertexIndices. Now is clear that there are only 4 unique vertices needed to render a square face, which makes sense.

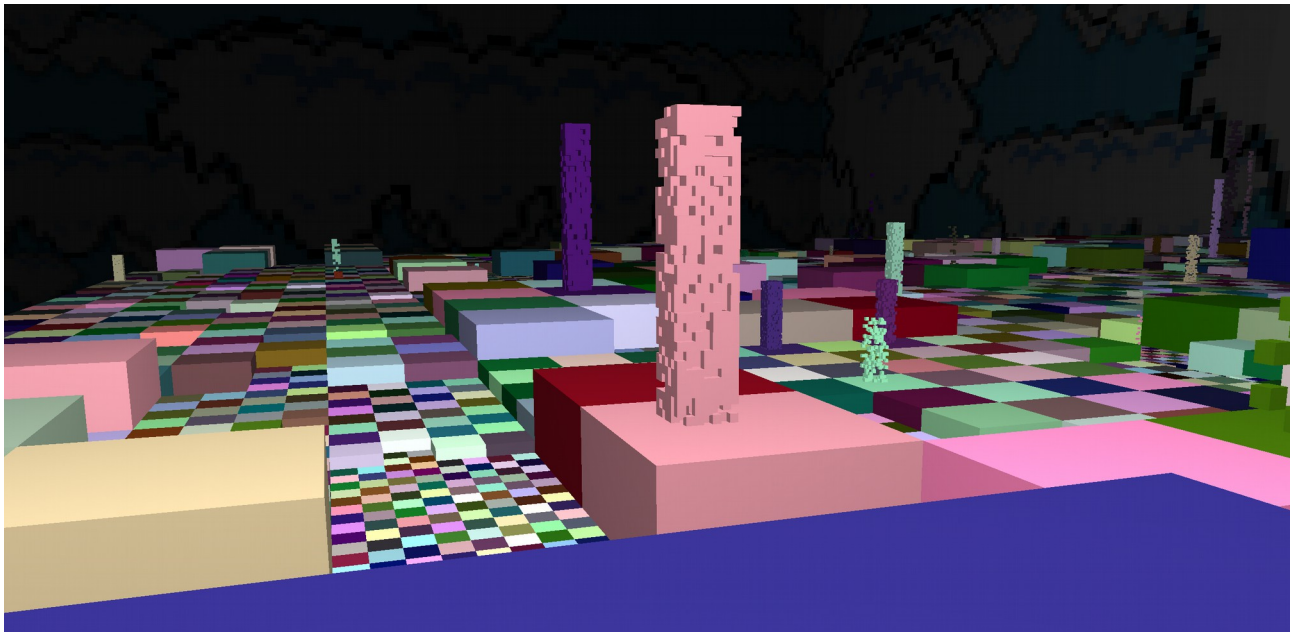
```
 public static int[] vertexIndices = {
        // front
        0, 1, 2,
        2, 3, 0,
        // top
        4, 5, 6,
        6, 7, 4,
        // back
...
```

Shaders and lighting

```
/* Load shaders */
vertexShader = Shader.loadShader(GL_VERTEX_SHADER, "resources/vertex.glsl");
fragmentShader = Shader.loadShader(GL_FRAGMENT_SHADER, "resources/fragment.glsl");
```

To render all this data correctly we use shaders. We have written 2 shaders, as is customary in OpenGL but this is not mandatory. One to process the vertices (vertex.glsl) and one to process the resulting fragments (fragment.glsl). Our shaders are quite similar to the standard shaders often provided in tutorials, calculating transformation of vertices (3D) into on-screen positioned (2D) fragments with a texture, color and light. Some particularities are; We use a single matrix for model and view transformations in our vertex shader (instead of 2 seperate ones) and we have 2 light structs in our fragment shader (hardcoded light). One light is a directional light, closely simulating a light source that emits light in a single direction (like the sun, called moodlight in our shader calculations). The other light is a point light, emitting light in all directions but which loses intensity over distance (called orblight). The intensity of the point light is partly constant, partly linear and

partly quadratically dependent of the distance.



In the picture can be seen that because of the point light that hoovers above our position, the scene (and especially the texture on the wall and ceiling) becomes gradually less visible the further you look (less light applied). Also if you look at any of the cubes you'll see that some face are lighter than others, this is because of the directional light. Every face of a cube, faces a different direction (has a different normal) which results in less or more reflected light dependent of it's normal relative to the angle of the directional light.

Rendering

Before we start to render we have to set all shader parameters correctly. This is done through the use of uniform variables that can be set from Java code using the LWJGL library and a wrapper class hiding all the different type of set uniform OpenGL/LWJGL calls into a single method *ShaderProgram.setUniform()*.

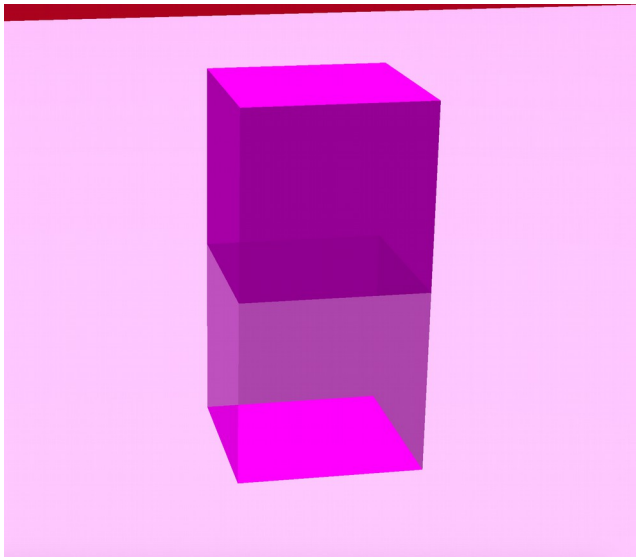
```
/* Set texture uniform */
int uniTex = program.getUniformLocation("texImage");
program.setUniform(uniTex, 0);
```

We also have to specify the vertex attributes, these are the positions of the “in” variables of the shader in the vertex data that is processed.

```
/* Specify Vertex Pointer */
int posAttrib = program.getAttributeLocation("position");
program.enableVertexAttribute(posAttrib);
program.pointVertexAttribute(posAttrib, 3, 11 * Float.BYTES, 0);
...
```

Once everything is set, the rendering can begin. First we clear the window to a single color to delete all data rendered from the previous frame. We also clear the depth buffer that aids OpenGL in identifying which vertices to draw first corresponding to the transformation matrix. We bind all

necessary data and enable the shaders that have been loaded to our GPU. We iterate our elements located in our vertex element buffer with *glDrawElements()*, which are integer pointers to our vertices in the vertex array buffer. To render some cubes with different textures we've saved the position of those cubes in the list and switch textures once when we've reached those cubes' corresponding position in the iteration. The final step in rendering is that of the reflected cubes. We flip those cubes around the reflection axis by using a different transform matrix and enable blending to mimic reflection on the surface. In this step the depth test is disabled as the reflected cubes are located under other rendered cubes that serve as reflecting surface and otherwise wouldn't be rendered. Finally we redraw the original reflected cube (not the reflection) such that it is always seen in front of the reflection. This method has some problems which haven't been solved in the scope of the project. One is that the reflection is drawn for all faces of the cubes and not only those visible in a reflection, this is because we had to disable the depth test. Two solutions will be proposed.



One is to render all cubes excluding the reflection and reflected cubes to a texture, and than render the reflection and reflected cubes both with depth test over this texture. This would result in a drop in performance (frame-rate), but not with much. Another solution is manually sorting the faces of the cubes according to it's relative position to the camera and only drawing those closest to it.

Game logic

The Player class is responsible for generating the modelview matrix based on it's position, rotation and camera properties. The players position can change in 4 directions relative to it's rotation. The rotation only changes if the mouse leaves the bounds of the central square area of the window, as long as the mouse pointer stays inside these bounds only the camera pitch and yaw changes. The player contains an update method that is called every frame to change the position and rotation according to the elapsed time between 2 frames to ensure a smooth and consistent change. Also in the Player class is the ray cast that calculates which tile of the board the mouse is pointing to. In short; 2 vectors are calculated to project the screen mouse position to world coordinates. Once this ray is calculated all cubes of the board are iterated and checked for intersection with this ray. This is done by checking if the ray intersects with any of the 6 faces of the cube. Once all intersecting cubes are found the closest in front of the camera is returned. For more detail check the method calls in order: *Player.shoot()->Board.getClosestCubeInFrontByRay()->Cube.intersectsWithLine()-*

>Cube.interstectFaceWithLine().

The Board, Tile and CubeCloud classes are used to generate a semi-random collection of cubes to build a scene. The most important method for the generation of the board is *divide()* in Tile. This method fractions a tile in 4,16 or 64 equal smaller tiles. By repeating this several times on random tiles, a random landscape is generated as seen on the first picture.

The Timer class is used exclusively in the *gameloop()* method of Game where the window and game state (more accurately player state) are continuously updated. The class keeps track of the number of updates that can be executed per second and prints them out to the title. Internally the Timer uses the *glfwGetTime()* method that returns the time passed in seconds since the initialization of the GLFW context.

Content (summary)

The scene consists of a board made up of thousands of cubes of different sizes all aligned in a single landscape. On the horizon and above we have a skybox that switches texture every so frame to give the impression of a loop of moving clouds. On our board we have several “Cubeclouds” that are also randomly generated. It is possible to move and jump on a single “tile” on the board and navigation between tiles is done by pointing and clicking on them. In the center of the currently occupied tile is the only (poorly) reflected cube of the game.

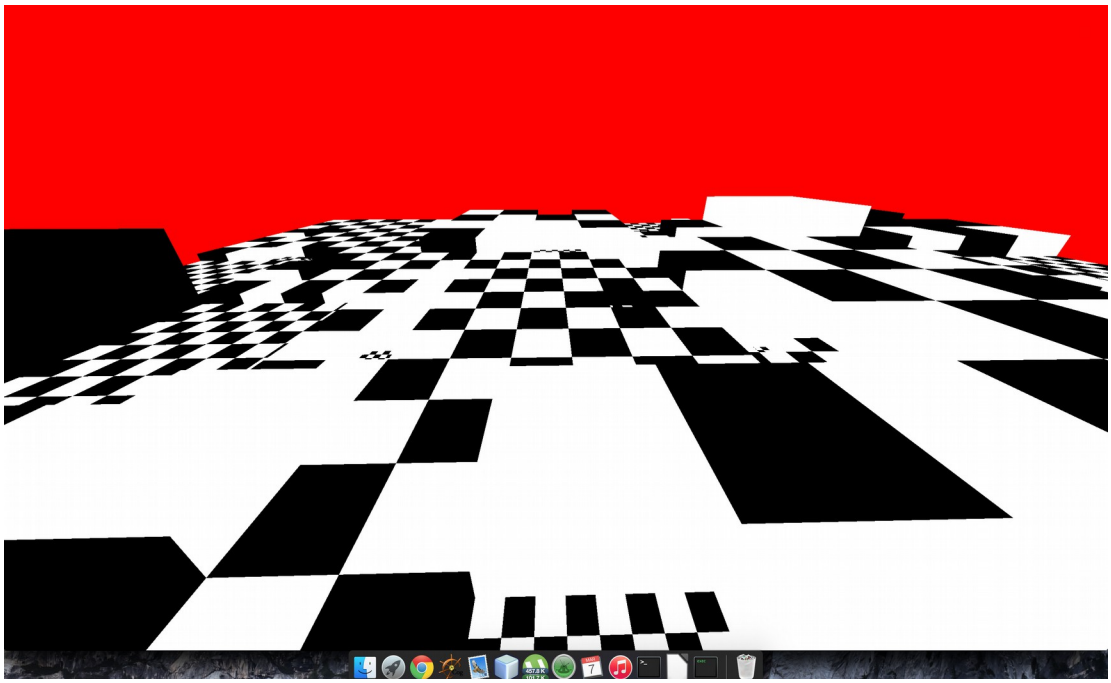
Demo

3 fat jars have been generated for linux, macosx and windows (x64) that run as is (double-click). please only run the application on the primary monitor.
Press escape to exit, “wasd” to move, space to jump and the left mouse button to change tile position.

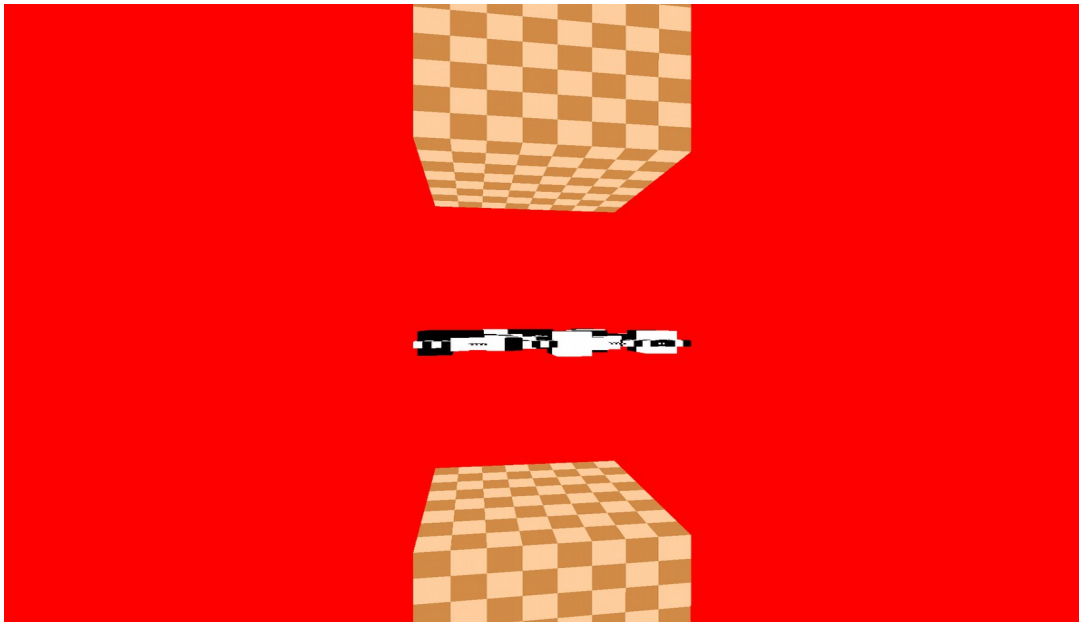
Libraries

pngdecoder: to load .png files used in texturing.
lwjgl: mentioned in intro.
math (package): a collection of classes implementing all necessary vector and matrix transformations for rendering 3D objects.

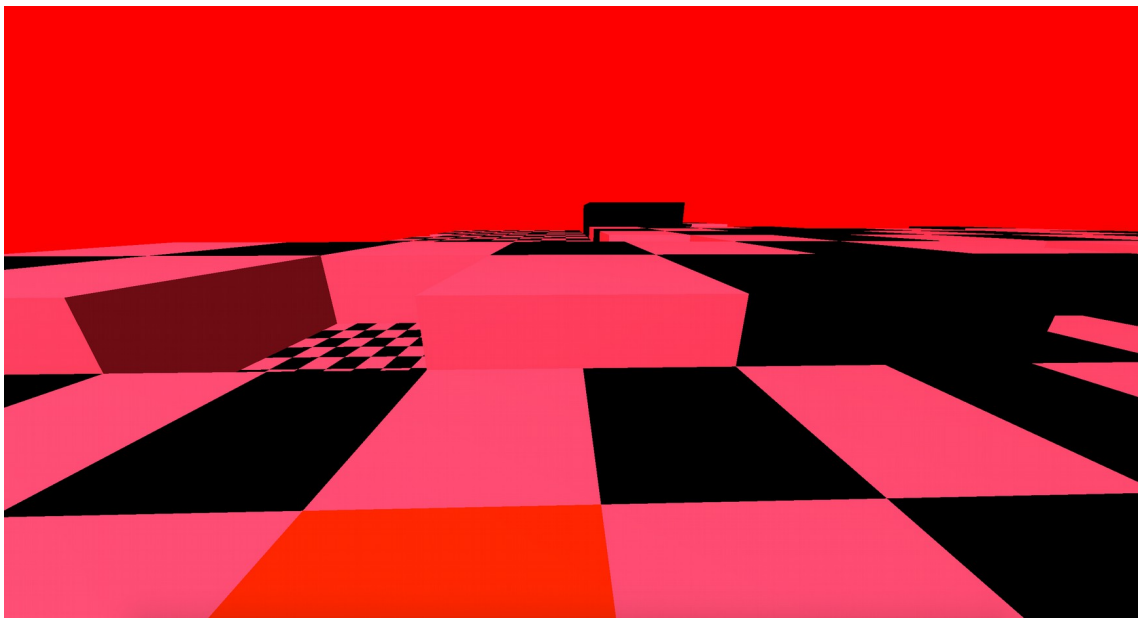
Some older screenshots of the implementation process



The first successful render of the board in black and white.



The first successful render of textures on giga cubes above and under the board.



The first result of the initial implementation of the directional light.