Programming Studio 2 – Project – 3D Viewer (own topic)

Erika Marttinen 885665
1st year CS student (Tietotekniikka) **27.4.2021**

GENERAL DESCRIPTION

The program is a simple 3D viewer. It shows a 3D object and has simple configurations for the lights. It is possible for the user to upload his/her own OBJ file.

Upon opening, the program shows the default rotating cube object, with a light gray directional light from the right. Another light, this time a point light, is located to the left of the objet. This light is off by default. A third, ambient, light makes sure that the base color of the object differs from the black background, when no other light hits a surface.

The user can upload another object from a Wavefront OBJ file. The program parses the file and turns it into a 3D object that can be viewed in the program. In the program, the user can toggle the rotation and adjust the colors of the directional and point lights. By pressing keys and clicking and dragging the mouse the user can roam the virtual space freely.

The program uses matrices, vectors and quaternions to calculate the location of each point in the three-dimensional space in any given point in time. A 3D object consists of points and triangles connecting the points and creating surfaces. These triangles are projected and rendered onto a 2D surface, making the scene visible to the user.

USER INTERFACE

The program shows a default scene when started. The user can choose another file to view by clicking the “Select OBJ file” -button. This opens a popup to the default directory, which is a resource folder in the program. The user can also upload a file from any other location on his/her computer.

The rotation of the object can be toggled from the “Toggle rotation” button. The colors of the directional and point lights can be adjusted using RGB color sliders and the colors blend additively. The program shows the intensity of each color element on a scale of 0 to 1. A demo patch of the color is located next to the sliders.

Using the keys W, S, A, D, Q and E, it is possible for the user to move the camera in the virtual space. WSAD being front, back, left and right, and EQ up and down, respectively. These directions are applied in relation to the camera. E. g. pressing “W” while the camera points straight up causes the camera to move upwards. By clicking and dragging the user can change the direction of the camera. Moving in the virtual scene is possible after the user clicks the scene once, if the user clicks something on the configuration bar.

To close the program, the user simply presses the x-button.

PROGRAM STRUCTURE

The Matrix4, Vec4 and Quaternion classes and their companion objects handle all the math behind the scenes. Almost all classes use these in some way. (A note regarding the UML: The arrows from these classes have been left out, as they would have made the rest of the UML completely unreadable.)

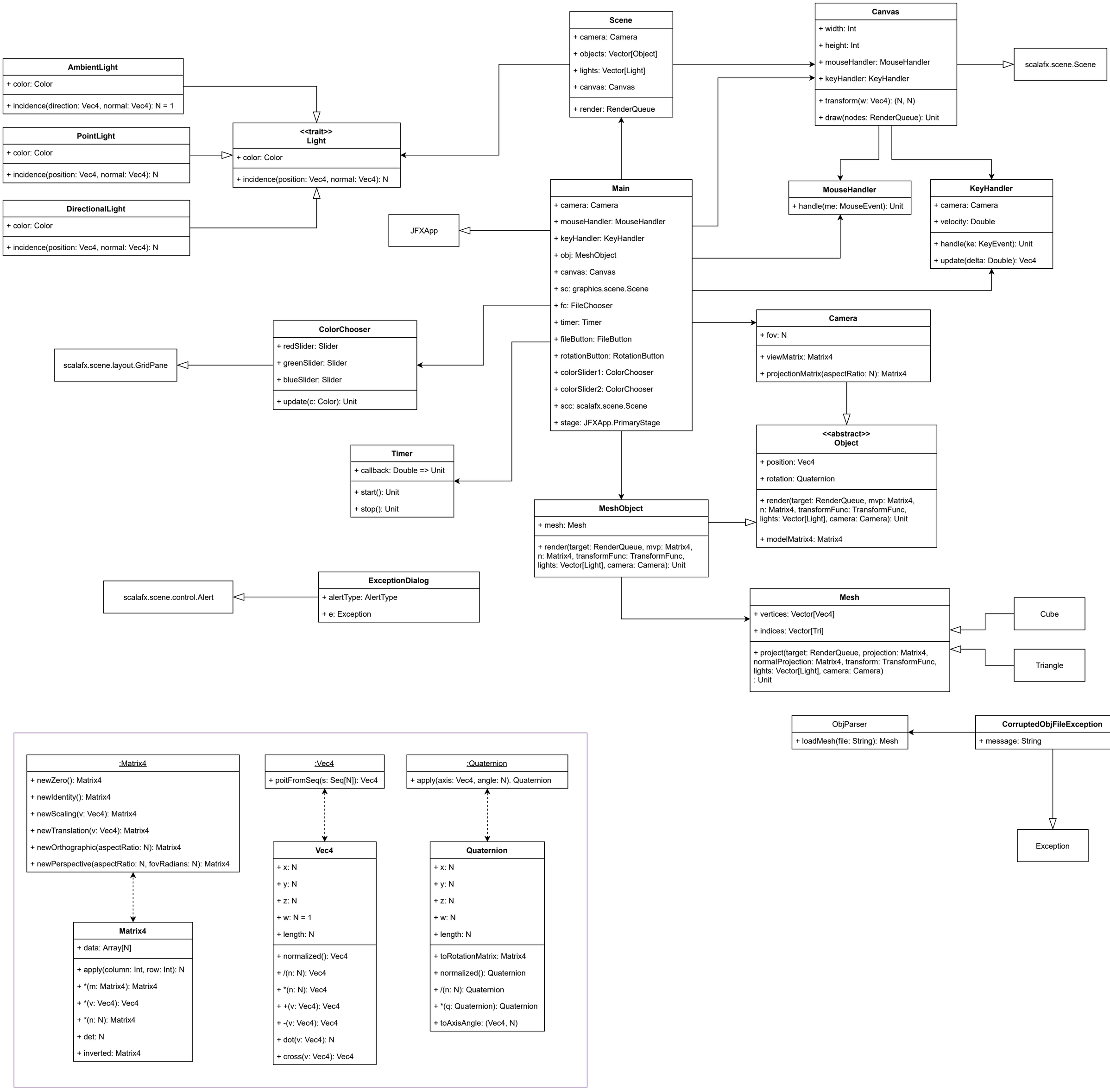
Things in the virtual scene are called objects (abstract class). Both the camera and the parsed meshes extend this object class. This has been done, because both the camera and the mesh objects have a position and rotation, meaning they both have model matrices. A mesh object has it's corresponding mesh saved as a parameter. 3D objects can also be hard-coded as meshes in the program, e. g. there is the default cube in the program. An obj parser -object does the reading and parsing of a file.

A scene itself contains the objects, the lights, the camera and the canvas, onto which the scene is projected. All lights share a common trait. The canvas draws the final scene and keeps track of the mouse movements and key actions. The mouse and key events are then processed by the mouse and key handlers respectively.

The programs GUI is another (ScalaFX) scene that is written into the main class. It contains two buttons and two color choosers. A timer keeps track of the animation. Error popups are handled by an exception dialog.

The final program structure very similar to the plan's structure. Some parts were expanded to contain more classes and some classes were removed/combined.

3D VIEWER UML



DATA STRUCTURES AND ALGORITHMS

The core math behind the program is based on 4-by-4 matrices (Matrix4, basically an array of size 16), vectors of size 4 (Vec4) and quaternions. These classes contain the basic arithmetic operations. Vec4 handles both three-dimensional points (the fourth component, w = 1) and directions (w = 0). The matrix class' companion object contains all necessary affine transformations and projections (translation, scaling, perspective), except for rotation, as this is handled through quaternions. The transformations are then applied in a specific order to create a desired output.

A buffered reader reads an OBJ file and turns each read line into a point (Vec4) or triangle. A buffered reader does the reading. The triangle itself is a Seq containing the three indices. These contain the information about which points will form a triangle's surface. Both vertices (points) and indices will be stored in collection vectors, as these need not be mutable. An array buffer is used during the parsing of a file, but this is changed to a vector after the parsing finishes for performance reasons. The points are given in a specific order, which is used together with a cross product to calculate the surfaces' normal vector.

The lights in the scene determine the color of a triangle. The angle between a triangle's normal and a ray of light is calculated through a dot product. Once normalized, the dot product is equal to the angle between the vectors in radians. By additively mixing the effect of all lights in the scene, the program calculates the final color of each triangle.

Not all triangles of an object are rendered at once. Any triangles pointing away from the camera are not rendered. If a triangle's every coordinate is outside the camera's viewing frustum, it is also not rendered. There is also a clipping plane right in front of the camera.

A rendering queue (PriorityQueue) is used in the order of rendering. The triangles furthest away from the camera are rendered first and those closest last. This ensures that there is no incorrect overlapping of triangles in the final render. The mesh keeps track of which triangles are being rendered and in which order.

The scene's camera is always located at the origin and points towards the negative z-axis. To create the illusion of flying around the object, the object is moved to the opposite direction of the camera. E. g. if the user presses "E" to move up, the camera actually stays in the origin and the object is moved to the opposite direction, in this case down. Rotating the camera is handled similarly by simply moving the object.

The scene has a collection vector for its objects and its lights again for performance reasons. If the objects in a scene change, a new vector will simply be created to replace the old vector.

The final scene is then rendered onto a 2D surface, the canvas. The whole process of applying the correct matrix to the object, applying the color of each triangle from the lights, choosing which triangles to render and rendering them in the correct order is then repeated for each frame.

The program contains a package object with type definitions used in the program.

FILES

The program uses Wavefront OBJ files. This file format is simple and provides all necessary information to render a simple 3D object. Vertices (points) are given in lines starting with “v” and indices in lines starting with “f”. This program ignores all other lines (e. g. the object’s name). The direction a triangle is pointing to can be determined from the order of the indices. Vertices are given as doubles and indices as integers. The numbers are separated with spaces.

TESTING

Unit tests were used for matrix and vector arithmetic operations. The correct solutions for each operation were imported from an external calculator. The thrown errors were also tested using unit tests.

Because this is a very graphically heavy program, a lot of testing happened through trial and error. For example the mouse handling and camera movement were implemented by writing something and starting the program. This was then repeated until the feature worked correctly.

Some temporary features were implemented to make this kind of debugging and testing easier. One example of this is the now deleted list of all JavaFX library’s colors. Using the index of a triangle, a color was chosen from the list. The colors were the applied with 50% opacity, so that the whole object was visible at once. Another temporary feature were green dots in the centers of the triangles. These were used to position the normal vectors correctly. A third temporary feature was an orthographic perspective to test the lighting and which triangles should be rendered. These temporary features were missing from the initial plan, as at the time of planning I had no idea which temporary features exactly were needed.

The order of testing followed the plan except for the swapped order of the lights and the object parser. This means the program was built step by step and then tested in between each small change. For example to test ScalaFX in general, a simple animated window was created to test how the animation timer works before any matrices/vectors were used.

KNOWN BUGS AND MISSING FEATURES

Creating a well functioning and simple program on an expandable base was more important to me than having every possible feature available to the user. E. g. being able to add and remove lights or move the objects themselves around the virtual scene would not have provided anything crucial, and would only make the GUI’s usability unnecessarily complicated. The base still supports the implementation of these elements and they can be done by coding them into the Main class. It would be possible to add these features to the GUI, if I’ll expand this project further in the future.

There are no shadow maps on the render. The color of a triangle is purely based on the direction it’s facing and the lights in the scene. This will cause some areas to be unnaturally bright where a shadow would normally be.

3 BEST SIDES AND 3 WEAKNESSES

- Best sides:
  - The base of the program is built to support further development.
  - It is possible for the user to move around the virtual scene freely. This feature turned out way better than expected.
  - The rendering happens in the CPU. This means that the program is easily accessible for every computer.

- Weaknesses:
  - Not all possible features are adjustable from the GUI (yet). These include adding/removing lights, adding/removing objects and changing the objects' animations.
  - The program uses the CPU for rendering instead of the GPU. This means that too detailed models will cause the program to lag.
  - The color of a triangle is purely based on the lights and the direction the triangle is facing. This means that there is no shadow mapping and some objects will have weird bright areas, where normally a shadow would be.

DEVIATIONS FROM THE PLAN, REALIZED PROCESS AND SCHEDULE

The first two weeks of the project followed the plan almost exactly. This was mainly due to the fact that I was most sure about what to do in this part. The only deviation came in testing. The implementation of a rendered object with points and lines (no surfaces) proved unnecessary, as it didn't provide anything useful and was mainly additional work. Instead, to view the full object, I used randomly selected colors at 50% opacity. This can be seen in action in "progress_demo.gif".

I ended up implementing the file parser before the lights, unlike originally planned. While working on the project this new order seemed more natural. Testing the file parser was easier with my 50% opacity random colors, than what it would have been with the full light implementation. Here I also expanded the Object-Mesh implementation. "Object" ended up being "a thing in the virtual space" (including the camera) and a rendered OBJ file was a "MeshObject". I also added the hard-coded meshes "Cube" and "Triangle" for testing and removed the "Point" class, as Vec4 was basically the same thing. The removal of the "Point" class also influenced this change in the order, as there was no need to think about points on the triangle, which I would have used in the lighting.

Except for the light's and file parser's implementation order switch up, the order of implementation roughly followed the plan. When it comes to the features of the program, changes also took part. While planning, I did not have a detailed idea of what exactly the final program should be able to do. Thus some features were stripped and others added. A removed feature is for example the ability to change the animation of the viewed object. Now the object has a default rotation that can be toggled from a button. More complex animations are possible with the current code, but since the program is mainly a 3D VIEWER and it is possible for the user to fly around in the virtual scene freely, there was no need for the user to be able to change the default animation. Adding the feature would only hurt the usability of the program.

Speaking of flying around the scene, this was also an additional feature that was not included in the original plan. Implementing this proved easier than initially expected. Moving in the scene can be viewed in

“progress_demo_2.gif”. There the initial lighting can also be seen. I tested the outlook of the lighting by placing a directional light in the same direction as the camera is pointing to. This was later expanded to a directional, point and ambient light, thus making the final light implementation more complex than initially planned.

The scene graph was also expanded. A scene can include any number of lights and objects. Even though the current final viewer can only have one object at a time and has the three default lights (again, this is only a 3D viewer), the program is built so that it supports more objects/lights at once. This is also something that was added later and was not originally planned. Two objects can bee seen in “monkey_tower_demo.mp4” and a light demo in “color_demo.mp4”.

The Matrix4, Vec4 and Quaternion -classes ended up being more complex than initially planned, though this was something I expected to happen. Matrices originally had multiplication with matrices and vectors, and vectors only had multiplication and division with values. There came a point in the project, where quaternion multiplication was needed. Thus a private constructor was created for it. Vec4 now has all arithmetic operations, including the dot and cross products. Matrix4 was changed from a hard-coded version (using parameters aa, ab, ac, …, dd) to using an array of size 16. This change was necessary after I needed to invert a matrix (for the camera movement). Inverting a matrix using hard coded parameters was simply too difficult and would decrease the quality of the code.

I had no clear vision of the GUI’s visuals when I started working on the program. Initially I thought of maybe having a start window before the main scene opens. When I started working on the GUI, I realized that this would not be a good match for the mechanics I had in mind. The GUI’s task was mainly to provide a “Load OBJ” -button and some light adjustments, the latter also being optional in the plan. Thus a completely separate window was not necessary. Instead a configuration bar was added to the bottom of the project.

I spent slightly more time on the project than I initially planned. The first six weeks took roughly 1.2-1.5 times the estimated time. On the last two weeks my time spent on this project was over twice the estimated amount. Finalizing every detail took longer than expected and I hadn’t planned in the time it took to write this document. The overall time spent on this project was still spread out quite evenly over the eight weeks working on the project.

This project has been one of the most valuable programming assignments, in terms of how much I’ve learned. It is the first project I’ve coded entirely from scratch. Even things like discussing a program’s architecture with friends broadened my horizon greatly. Using an external library (ScalaFX) took some getting used to, especially because there aren’t infinitely many resources on the internet for it. By the end I’d say that I finally got the gist of how JavaFX elements should be translated into Scala.

FINAL EVALUATION

The program is a simple 3D viewer. The user is able to fly anywhere in the virtual scene, can upload his/her own Wavefront OBJ file and can adjust the colors of the lights in the scene. The program's strength is that is built on an easily expandable and flexible base. The user may not be able to infinitely many things, but the current features demonstrate just how much the potential the program has.

There are many features that could be added to the program in the future. For example it could be possible to allow the user to add multiple objects and manually configure the location and direction of the lights. Another feature would be to be able to adjust an object's rotation and maybe move it around the scene. It could also be possible to upload 3D objects from another file format, such as STL and glTF. All the aforementioned features are cool, yet they do not provide any core elements of a 3D viewer and have thus not been implemented yet.

If I were to start this program anew, I would redesign the mesh-object structure. Their functions "render" and "project" take in way too many parameters. There is no quick fix for this, the whole structure would have to be redesigned. The matrices and vectors would stay roughly the same. The main class would probably be split into smaller parts and the GUI would be expanded.

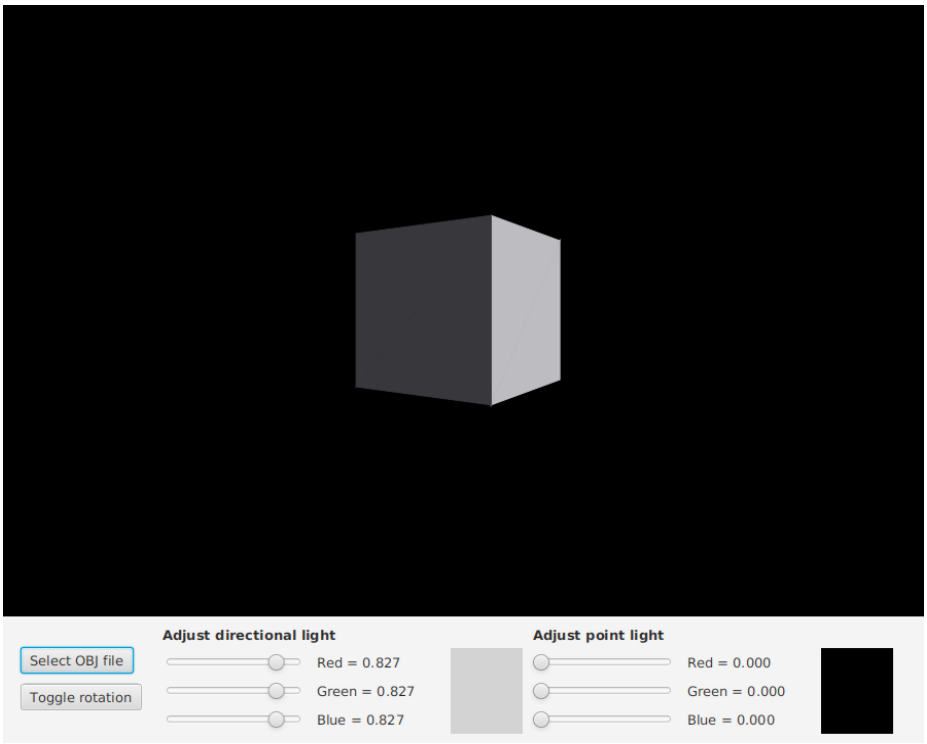
Overall I am very happy with the outcome of this project. Especially the ability to move around the scene and how well the lights turned out are some elements I'm very proud of. I am seriously considering developing this project further.

REFERENCES/LINKS

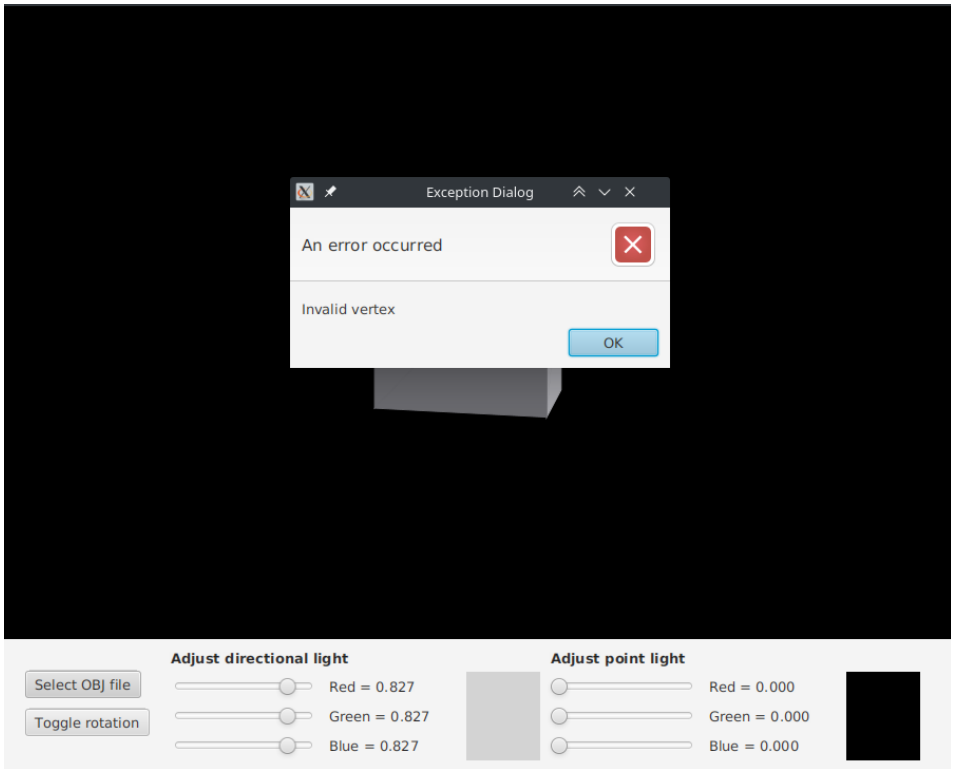
- Scala Standard Library: https://www.scala-lang.org/api/current/
- Sample .obj files https://groups.csail.mit.edu/graphics/classes/6.837/F03/models/
- ScalaFX:
  - ScalaFX API: http://www.scalafx.org/api/8.0/#scalafx.package
  - JavaFX API: https://docs.oracle.com/javase/8/javafx/api/overview-summary.html
  - Tutorials: https://github.com/scalafx/ScalaFX-Tutorials
  - YouTube tutorials: https://www.youtube.com/channel/UCEvjiWkK2BoIH819T-buioQ
- Wikipedia
- Stack Overflow

APPENDIXES

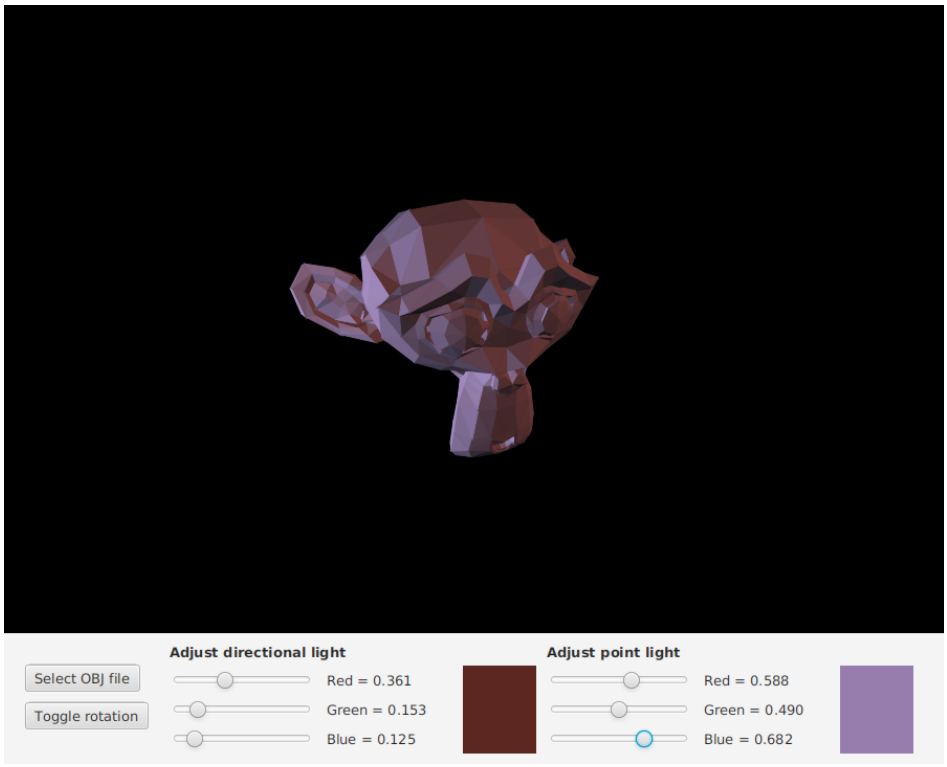
Default starting screen:



Error popup after loading an invalid file:



After adjusting the lighting and changing the object:



The ambient light: