1. Project Document 5/2019

threeD

1. Abstract

The project is a from-scratch 3D engine running on Scala FX, written in Scala. Scala FX is mostly a re-skin for Java FX and the two are both functional inside the app. I have taken some functionality from Java FX, regarding the parts in which Scala FX documentation was sparse. The engine is fully scalable with different 3D objects, as long as the data is convertible to the format of three 3d points on a CSV row, each consisting of (x,z,y)-coordinates forming a triangle in a clock-wise order. The project has a bug on "Projection.scala", on rows 145, and 146: A plane intersection edge case problem, where the values scale too big and get drawn on the wrong corner of the screen.

2. User manual

The program is started by launching "Front.scala" as a Scala Application. This opens up the starting menu as a window, in which you can operate by using the mouse or keyboard. Arrow keys allow movement between buttons, same as tab, and with space you can press any button. Remember that if you are using arrow keys and enter a slider, it is not possible to get out of it by left and right arrow keys, so use instead tab/up-down arrow keys. To enter fullscreen mode you can press "F" any time, same to exit it. This is especially preferable on the Keyboard shortcuts scene, to get a good view of the whole picture.

When you are done tinkering with settings, get back to the first scene, from which you can choose "Start" to enter the main program. One final note about settings is that the intersection math is rather heavy so if your hardware is having trouble, you can turn of clipping under "Show more" in settings, but then you have to remember to always have the objects in the world in front of you, as the clipping is integral to filtering out objects on the backside of camera.

The controls are as follows. WASD to move around, mouse to turn around. Shift to run, control to crouch. While crouched, Q and E to lean to the sides. Plus, to zoom in, minus to zoom out. Z to reset zoom level to the original. To exit/enter full-screen, press F, and to exit the program Esc. Space to jump around/ fly.

The program is best used as a visualization tool, so use of CSVWriter to make on shapes is encouraged.

3. Program Structure

The main window is Front.scala which handles everything related to ScalaFx excluding styling, which resides inside src/stylesheets. Front is responsible for the whole visualization of the program, including both the starting menus and the main program. The substructure consists of scenes, each handling one view of the program, and is as follows:

- bootstrap: The opening screen with navigation to the main screen, options, and quit. Quitting is always possible with Esc, but here is a fallback incase that fails.
- options: All the options handling “game mechanics”, which includes everything that affects the player directly. This includes things such as player speed, as well as all the “cheats”. From here it is possible to navigate “Back” -> back to the opening screen, “Controls” -> to view the keyboard navigation tips, and “Show more” -> to show the rest of the options.
- more:
  - This includes setting the clipping of the main screen. This clips all the to-be-drawn triangles against the plane to the sides of the player, with normal vector to the “viewing direction”, in other words where the camera is pointing. Disabling it increases performance greatly but then it is required that all the objects reside in front of player all times.
  - Here are also settings for adjusting screen size. A good thing to note is that the window is always at max the width or height of the screen, so if the screen is already full with 16:9 “small”, the result will be exactly the same with 16:9 “medium”. The aspect ratio affects mostly the windowed mode, as fullscreen always fills the entire screen, but if the image remains distorted, changing the aspect ratio to the right one might help. Also pressing “Shift+F” when exiting fullscreen changes it to full-screen with the screen only filling the size a corresponding windowed mode would fill.
  - Last in the scene is file loading. Only csv works. Obj-files load as well but are corrupted, if the ordering of the points in the triangles are not the exact same as the one used in the program. This is mainly manifested in the clock-wise order of the points of the triangle viewed from the side the triangle should be visible from e.g. a wall that is only visible from either inside or outside. So, if you want a house to have walls visible from inside and

outside, create two tringles facing opposite ways. Press the "Load" to load a preferred file. If the program did not receive the string, it will ask for one, likewise it will inform in the window if the file was found. If the file does not exist it the resources-folder, the program will close. In the console printout on the top row there is the custom exception which will tell the tried and failed path. Usually you must scroll up in the console to find the exception.

- threeD: As the name implies, this is the core of the product. All the referrals before to the "main screen" mean this one. This is where all the game mechanics are handled. You can see all the flags established at the beginning, as they are prepared when the whole program launches for the first time, so they will not be reset when the game is on-going. The dynamic part of all the scenes are the event handlers like "onKeyPressed" which can be used to alter the scene content while in the scene. The recurring operation, which happens approxiamately once every 0.01 seconds, is the AnimationTimer, and everything inside it. Even though the program only executes the loop multiple times, all the event listeners of threeD work parallel to its running. The exact order of execution is unknown to me, but effectively they work parallel. From inside threeD there is calls to Projector to prepare the data for visualization, and to Camera, to handle the rotational aspects as well as moving around. The call to projector gives first of all the 2D converted shapes, but also the final 3D shapes to allow for finding out the right draw order. The timer also hosts all the listeners to the flags established in the scene/stage, that need dynamic functionality. At the end of the timer, everything is drawn on the screen

A lot of the other program structure is just a varying degree of helper functions for the main app, which is better handled in the following chapter 5.

4. Algorithms

The key algorithms of the project are the matrix calculations made in VectorVer and Matrix classes, the 3D planar projection, as well as intersection math, and the sorting, filtering and converting of the drawable shapes.

Algorithms under following classes:

Camera

Camera handles the rotation and the moving of the player in the world. The rotation is achieved with Tait-Bryan angles which allow as to look left-right , up-down as well as lean to the sides. As well all the other algorithms my version uses (x,z,y) sequence with behore-mentioned order of rotation.

Located under Front is the amount of turning, which I've decided on to be fractions of 2 times Pi to achieve regularity, and of course times sensitivity to allow user customization without touching the code.

Camera also handles moving in conjuction with Front, and to achieve precision the moving is inversely proportional to the tickrate of the AnimationTimer, because if the distance moved was always the same, the player would move more slowly when the tickrate is lower. Now the algorithm adjust the player to move more when the framerate is lower and compensate by lowering the speed when it is higher.

Front

A lot of required functionality is achieved with the help of the ScalaFX and JavaFX toolboxes, but some aspects of the program were not easily handled by them, and I'll mention some of them here. First of all the program can be run in full-screen mode, or in any customisable aspect ratio the user wishes. The big problem was to make everything scala according to the adjustable window, as the packages did not contain much in the way of this. The way I've achieved the desired aspect ratio on the window is as follows. I'll get the size of the screen, and if the base exceeds the screensize, I'll crop it to that and get the ratio in which it is reduced and reduce height in the same way. Then I'll repeat the the method with height, and the window is limited to the screen. If the width does not exceed the screensize, then I'll do the process in the opposite direction with height adjustments first and witdth after height has been cropped and it has had some kind of effect on width as well. All the drawable buttons, grids, menus and the main program adjust to screensize changes.

Because user losing control is a strictly allowed concept in JavaFX, if you want to turn around in the world, the mouse will go over the sides and escape the window. So as the AnimationTimer is always running, I'll center the mouse using Java's Robot, and get the distance the mouse has had time to move between ticks, and react to that accordingly. As the Robot moves the mouse as well, I cannot react straight to the distance moved between ticks because otherwise it would just do zigzag between the user and Robot input, so I'll always move the mouse in regard to distance from the center of the screen, such that when Robot moves the mouse back, the distance will be 0. Moving the mouse around resulted in it racing around the center 50 pixels of the screen, so I hid it. This had the unfortunate outcome between the Cursor and the JavaFX app, that I couldn't get the cursor back when switching back to options, so I've made it so that the options acts as a bootstrap and is non-returnable.

JavaFX app also has the design functionality that switching between them or stages, which is the main module of the app but in itself does not do much, is not doable. Because I wanted to implement smooth transitions between the options menus and the game itself, I found a way to switch the scenes. It is not perfect approach because of two things. First all the scenes have to be assigned to pointers, so that I can either do them in lazy vals or the normals vals. Using lazy vals disabled the intended fullscreen transition functionality so that I couldn't use them. If using them I could not enable fullscreen before the main screen, as in, in the options, even thought I've managed to implement a stage-wise fullscreen handler function using private[package name]. So that I had to use normal vals, and that meant all the content in each scene is initiated on boot, which means the AnimationTimer starts to run for example. This does not affect the performance noticably as all the options scenes are light-weight. JavaFX apps handle a lot of the content change dynamically but for example scene sizes have to be defined as constructors, and scene size is not a mutable value inside a scene. This I've countered using a default size 4:3 windowed mode on all the options scenes, with option to go fullscreen, and the user customisable aspect ratio and window size only affect the main window, in which I can give the switching sizes all the time to the canvas inside AnimationTimer and the scene scales to fit the canvas. Another problem with the switching scenes approach is that it has poor JavaFX fullscreen support. The exit hint displays every time, when user switches a scene in fullscreen mode, and stays in the middle of the screen for a good while. This is good design in part of giving user control, but fits my design poorly. Because of that and the fact that there is virtually no support for centering content, I decided to start the program as windowed with option to enter fullscreen (for example when viewing the image of key inputs, which does not fit nicely in the small window), and the program enters fullscreen automatically when going to the main screen. Another thing I couldn't do anything about is that if switching between scenes in fullscreen mode, there is a noticiable time between the scenes when the app is windowed, before entering fullscreen again in next scene. This I could not combat.

Another problem arose as the user input only mostly allows a key at a time, so I established a flag system to enable full multi-key support. Each key input triggers a flag, and the AnimationTimer which runs continuously reacts to the flags and not the keyinputs. To limit extra flags I've limited this to all the player controls, so the feature of moving the screen around in windowed mode is handles by straight up key inputs. Speaking of it, I've introduced to different ways to move the screen around to combat the fact that the mouse is always in the middle of the screen per Robot in the paragraph couple notches up. Firstly, they are only possible in windowed mode because doing it in full-screen does not really make sense. Using left-right arrow keys you can move the window between the left and right sides of the screen, as well as centring it. Then up-down I gave it more mobility to fine tune for desired height, with the functionality that if the windows Windows-control bar is on the top of the screen, it hops straight out on the next press, and back down on a press down.

Then regarding the actual game mechanics. The design problem was to make it feel as real as possible with not a lot of resources. I introduced approximate vertical Earth kinetic energy in the game, with 9.81 meters of down-ward acceleration each second you are in the air. If I can get the obj. file loading to work, this is scalable to create different environments like the moon etc. In addition to this, the player has stamina, which depletes when running or jumping, and to capture that moment when a person has stopped running and needs just a second or two to adjust before being able to continue, I introduced a replenishing stage. So, if you fall under half the stamina, as in did run a fair bit or jumped, there is an interval before you can start running again, or the stamina starts replenishing. Then when this period is over the stamina starts the recovery process where it first slowly increases and after being over the halfway-point, it starts increasing rapidly. Because it feels kind of annoying to have these real-world constraints, I then added all kinds of cheat modes to entertain the user and allow more mobility. Flying is just removing the constraints on jumping but the double jump known from by example Sly Cooper games was niftier. For it I introduced constraints that it has to be after a certain time from the first jump. This is because the tick rate in AnimationTimer is so fast that the one space press actually registers as many, and the player would use the second jump immediately after the first.

Continuing on the game mechanics, crouching was the odd ball out because it is not a one of time event like jumping or a continuous one like sprinting, but kind of like a transition with an animation. So, pressing control starts the crouching phase and releasing it starts the getting up phase. And while crouching, thanks to Tait-Bryan angles, I made it so that you can lean of the sides with similar in and out transitions as crouching, and so that releasing control which is the crouching button, also starts the leaning up process. Then wrapping up all the mobility, there is a “if else”-structure in priority order to determine how fast is the player. To keep the maximum customizability in mind, there is two values for this. One is holder for the user input from the settings, and other takes this and the current mobility phase in count and feeds it to the Camera to move around.

VectorVer

I’ve decided to implement all the Vector and Matrix calculations by myself to fit my (x,z,y)-coordinate system, and to reduce dependencies. In VectorVer (name represents the structure of the vector, vertical) there is an algorithm to check the validity of each Vector to ensure that the loaded data is correct. There is also addition, multiplication with another 1 time 3 vector, or 3 times 3 matrix, multiplication with scalar, normalization, sign switching, absolute values, equality, cross product, dot product and finally taking the length of the vector. I only implemented the ones I needed, and most of this is for the intersection math, and if it is turned off, I think only minus and multiplication is needed for the planar projection.

Matrix

In matrix class I've got just the multiplication which is enough for the projection.

Projection

Home to the conversion of CSV data, and the main planar projection, as well as updating the data to draw each cycle.

First of all, to ensure the integrity of triangles, the three corner points of a triangle must be drawn in clock-wise order. It could be other way around as well, but the thing that matters it that every single one is the same. This ensures the normal vectors always point in the same way in relation to the triangle. This is a major way to save computing power as I'll only draw the triangles whose normal vector points to the direction of the player. Basically, the back side of any structure remains undrawn. Here is also the clipping against the y-plane of the player (going to the sides with normal vector to the direction which the camera is pointing), which was so long it got its own file.

The following is the planar projection which I ended up using.
https://en.wikipedia.org/wiki/3D_projection#Perspective_projection *link can also be found at the end.*

In the end I got it to work for the exact reasons I chose it from: Camera position, Tait-Bryan angles, and certain efficiency. As rotation and shearing can be achieved with 3x3 matrices, but transformation cannot, all the above is just 3x3 matrices, and the transformation can be handled by the formula below, so that I don't have to go through 4x4 matrices. The problem with this approach is that the e does not fit very well together with my clipping against the camera plane, which results in unfortunate edge cases of triangles glitching across the screen. If it has a value over zero, every object is upside down, and if it is over zero everything is inverted against the y-z plane, which is manageable. It was not possible to set the Camera plane so that nothing glitches out, so I just empirically went with a relatively good value.

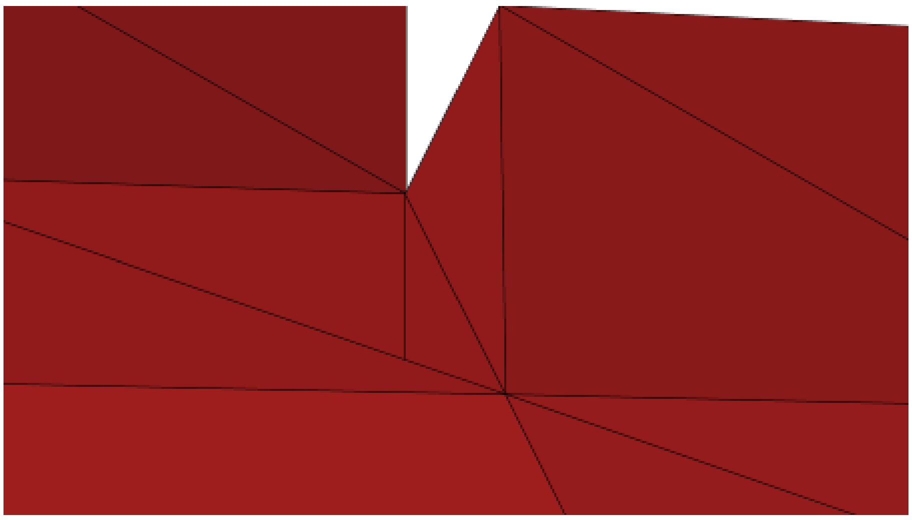
$$\begin{bmatrix}\mathbf{d}_x\\ \mathbf{d}_y\\ \mathbf{d}_z\end{bmatrix} = \begin{bmatrix}1 & 0 & 0\\ 0 & \cos(\theta_x) & \sin(\theta_x)\\ 0 & -\sin(\theta_x) & \cos(\theta_x)\end{bmatrix}\begin{bmatrix}\cos(\theta_y) & 0 & -\sin(\theta_y)\\ 0 & 1 & 0\\ \sin(\theta_y) & 0 & \cos(\theta_y)\end{bmatrix}\begin{bmatrix}\cos(\theta_z) & \sin(\theta_z) & 0\\ -\sin(\theta_z) & \cos(\theta_z) & 0\\ 0 & 0 & 1\end{bmatrix}\left(\begin{bmatrix}\mathbf{a}_x\\ \mathbf{a}_y\\ \mathbf{a}_z\end{bmatrix} - \begin{bmatrix}\mathbf{c}_x\\ \mathbf{c}_y\\ \mathbf{c}_z\end{bmatrix}\right)$$

$$\mathbf{b}_x = \frac{\mathbf{e}_z}{\mathbf{d}_z}\mathbf{d}_x + \mathbf{e}_x\,,$$

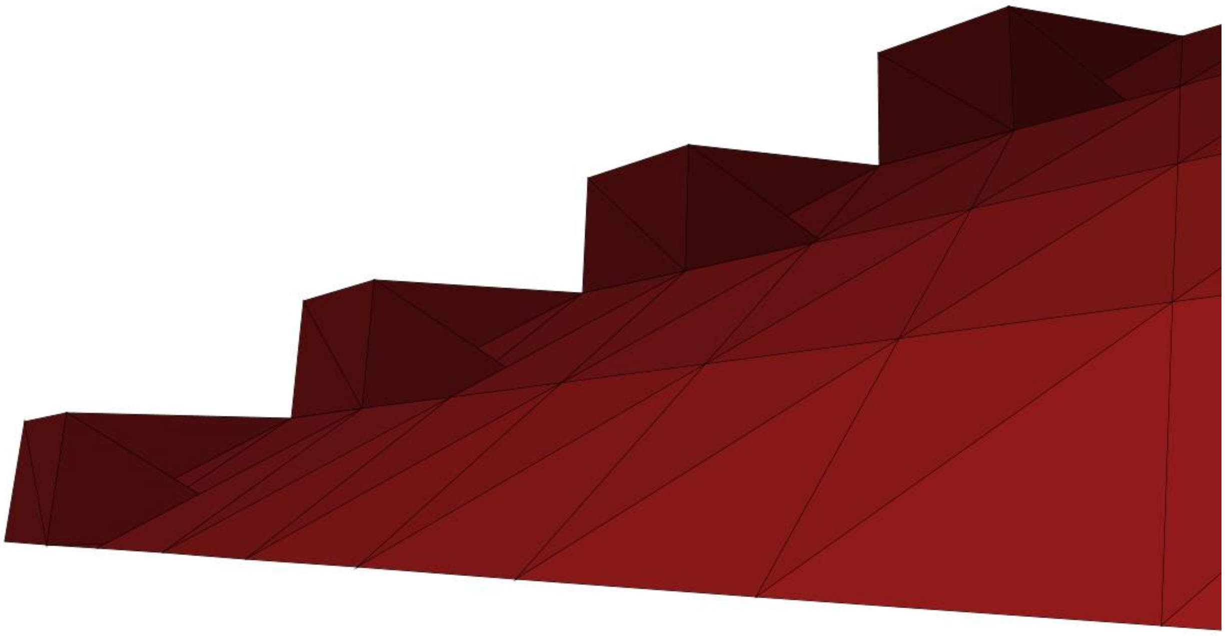
$$\mathbf{b}_y = \frac{\mathbf{e}_z}{\mathbf{d}_z}\mathbf{d}_y + \mathbf{e}_y\,.$$

- $\mathbf{a}_{x,y,z}$ – the 3D position of a point *A* that is to be projected.
- $\mathbf{c}_{x,y,z}$ – the 3D position of a point *C* representing the camera.
- $\theta_{x,y,z}$ – The orientation of the camera (represented by Tait–Bryan angles).
- $\mathbf{e}_{x,y,z}$ - the display surface's position relative to the camera pinhole C.[8] Most conventions use positive z values (the plane being in front of the pinhole), however negative z values are physically more correct, but the image will be inverted both horizontally and vertically.

The projection always gives the triangles in order of the distance to the player, and they are drawn accordingly. They are sorted in the conversion process according to the combined distance of their end points to the camera.

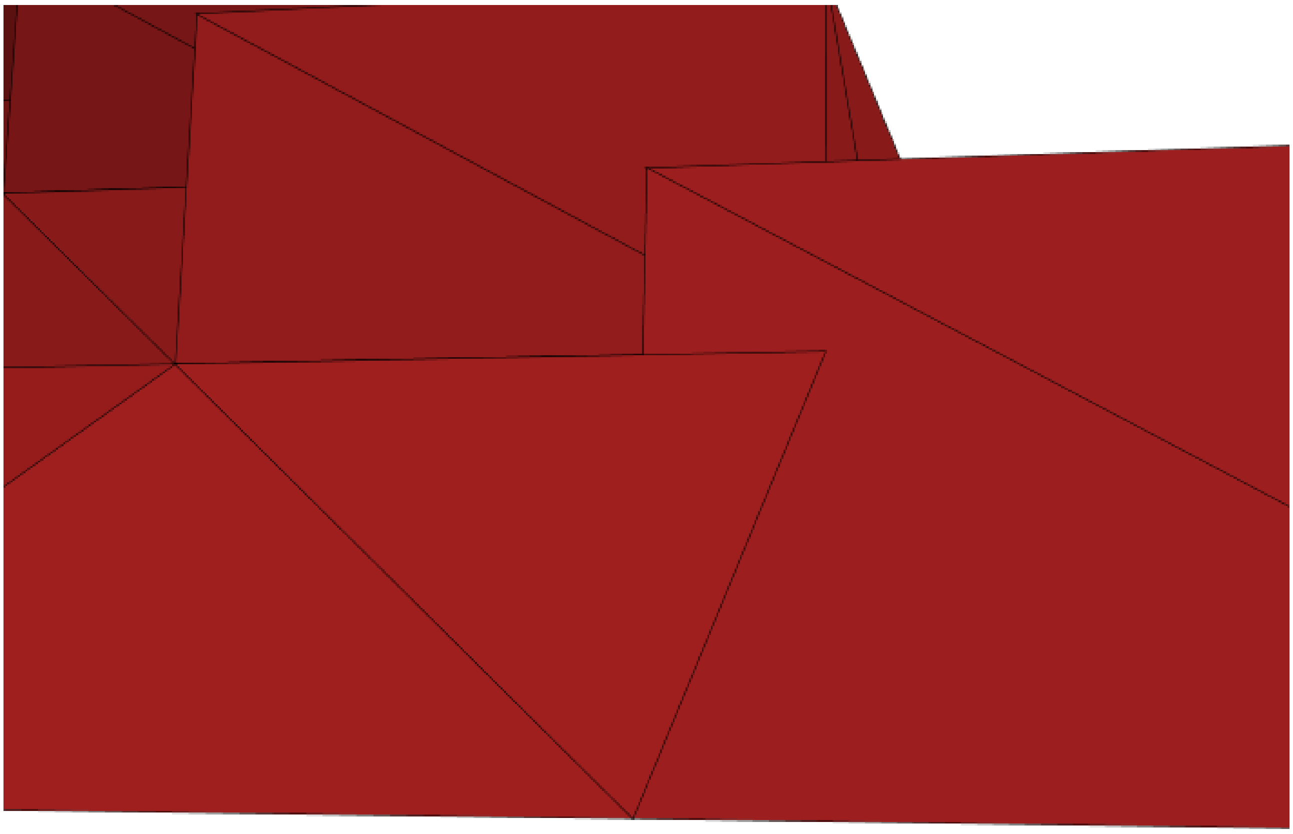


It works most of the time and for the projects I made it only messed it up very rarely. I've gone through alternate methods. If I emphazise the furthest away point the result is pretty much the same but with more computational resources wasted. If I only take the furthest away points:



The result is more immerse-breaking.

Same with taking the closest one.



5. Data Structures

The space is constructed using double values for precision. This is necessary for example because finding intersection point with the rounding of integers would be inprecise, and would need workarounds to function properly. But handling a lot of doubles around is resource-heavy so I created the VectorVer class to store them and this object is handed around. One of the problems of the projection is that for some cases there is an indeterminate amount of results from each operations (e.g. filtering), so for those I'm using Buffers for the ability to append just the right amount. Each time such a process is ready, I'll convert it to Array for the efficient index-fetching and for a degree of immutability for safety as Array is in many cases the fastest for exactly those. Buffers could be replaced by var Lists and appending to those, but Buffer is less memory overhead and all the operations are done on the same one, not a copy, which is something I like. Allows clearing for example.

6. Files

My default file system in CSV for it's simplicity, ease-of-use and overall popularity. It means people can very easily write the coordinates they wish in their preferred way. CSV is all-together a pleasure to work with. On the side I also tried to implement OBJ files with only the bare minumum information: the coordinates and the references to coordinates. When there is a program that can write in this way and make use of the common shared points between triangles, it is significantly faster to work with and smaller file. Unfortunately as I downloaded Blender just for this, I'm still beginner in the program and the points are unfortunately loaded in wrong order, so triangles are having wrong endpoints, and normal vectors being in the wrong direction, which results in the triangles visibility being from the wrong side.

I also introduced a lot of methods to help write the CSV in CSVWriter. Each line has the points needed for a triangle, in a clockwise order. That is if you are looking at the triangle, the points are in your eyes in clockwise order. Each point has (x,z,y)-coordinates. So when the camera rotation is zero, positive x is right, z is up and y is forward. As I mentioned in the projection paragraph, the world is projected across the z,y-plane, which results in the negative x being on the right. This is handled in the moving, but good thing to keep in mind is that you can use all the tools to create any shape in the world but it just will be on the other side from where you expected. I think this is quite negligible fault.

7. Testing

All the math has been tested on implementation with confirmed values from apt references. I also made great use of the continuing running of the AnimationTimer, which provides with changing realtime values, which made it in most cases rather easy to find out the problem. I also made the file readers throw out useful exceptions when the input data is not suitable. Otherwise as there is very limited user input, I do not think making Unit Tests for the user is very useful. I did the testing I thought, more of it than I thought, but mostly in the form I thought.

8. Known faults and short-comings

The clipping of objects on the sides and behind the camera. As seen from the difference on how it looks with clipping of, it is certainly doing a lot, but just a couple of triangles glitching over to the wrong side of camera is a program-breaking bug in the amount of disruption it causes to the player. I couldn't fix it and neither did the main assistant, not to any fault of either in my honest opinion. In the program using – and + keys it is possible to alter the "e" value of the projection which corresponds to a certain kind of plane. This in my testing has a surprisingly limited effect on the amount of unprompted clipping. I've found the exact problem point in the code: Under Projection.scala, on the rows 145 and 146. The values should lie within (-1,1), but as the points get closer to the player's y-plane, they grow in order of magnitude. This could be salvaged by min/maxing the values, but the problem lies somewhere deeper, as right on the edge case, values get flipped around. So when for example a point gets to the limit of Double value on the bottom hand corner, it stays there arbitariry, and then might flip to any other corner of the screen. I tried to get around this by not allowing the value get close to that, or only adding the required 1.0 in the conversion to screen coordinates, if the values are under a certain value, but these did not work. When I added the extra clipping with a FOV-cone, the problem was lessened to a degree, but I could not get around it. This affects the integral parts of drawing process, and is the main reason you cannot go inside a structure without everything glithcing out. I've tried approxiamately five different ways of doing the intersection math for object collision and drawing process, but I think I would have to change the projection in itself to get the program up and running smoothly.

9. The best and the worst

Worst:

The above-mentioned clipping problems. Please refer to Chapter 9 for info.

Best:

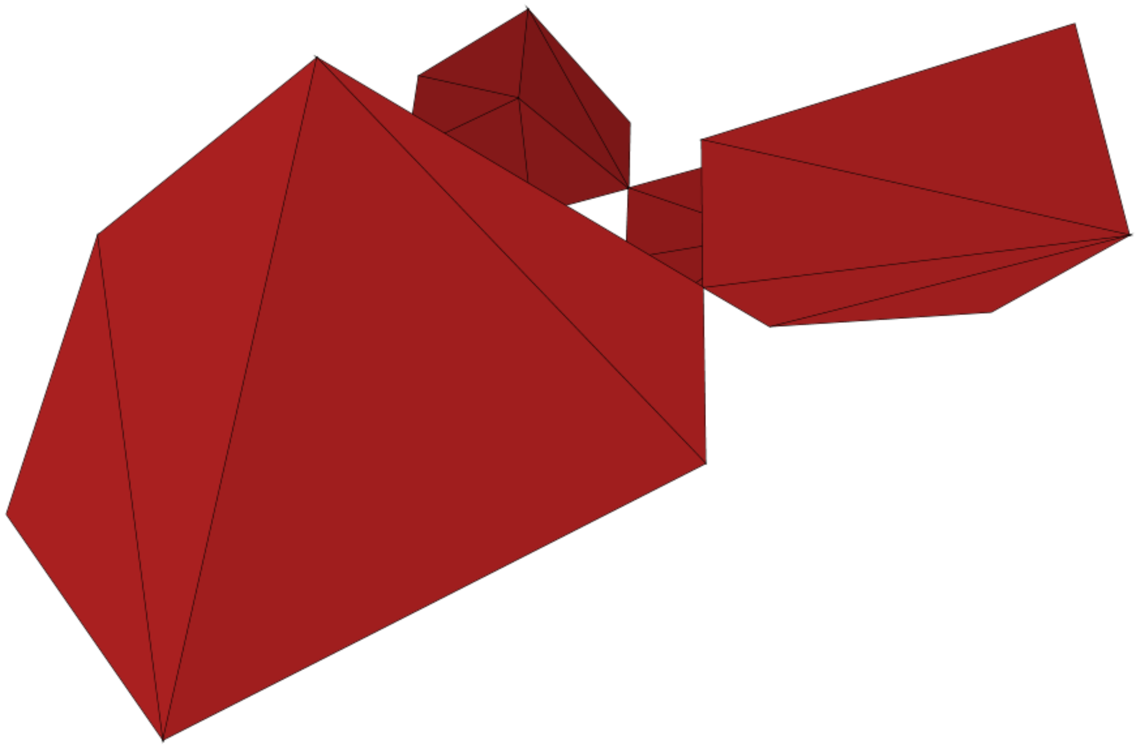
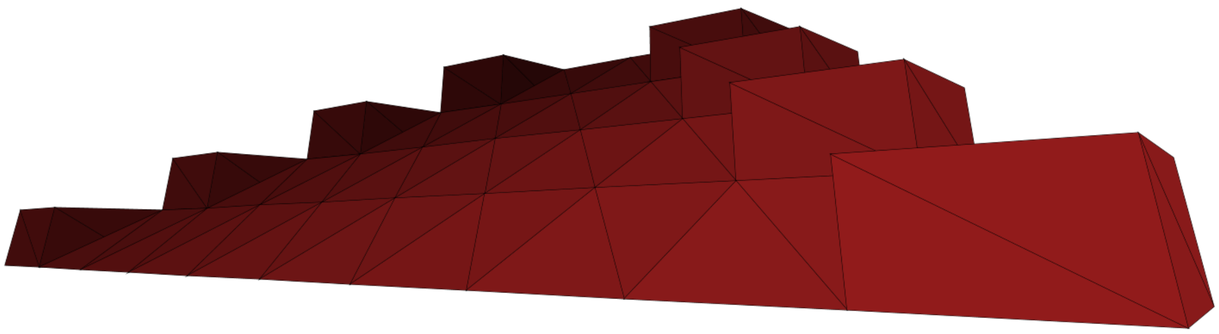
The use of ScalaFx. Both the wide usage of different aspects of the app, as well as finding work-arounds to fit just the intended functionality. For all the options I've included the best possible type of controls, and figured out how to react accordingly. For the limited number of the controls where it is possible to use this.selected.onChange, I tried to use that. It's good to know that for most of the cases it's the best option, as if you have a group of buttons, of which you can only select one, when a button loses the selection it won't fire anything with just onAction because that's for user inputs only. I would say this good understanding of the underlaying platform and making the best of it is one of the best parts of the project. Also the use of -fx- styling in the app, which is kind of like css-lite, and like ScalaFx not the easiest to work with.

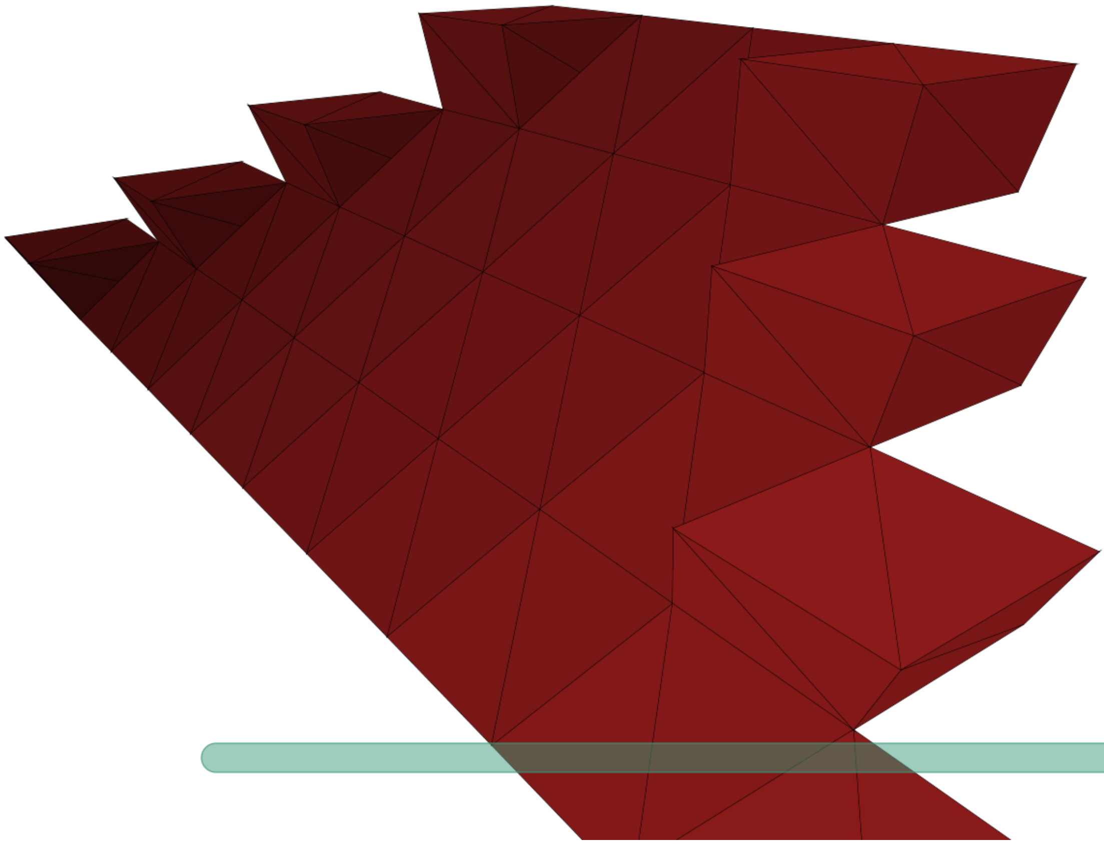
The amount of customization options the user has. For example the support for windowed, "windowed top bar hidden", and full-screen with all the possible aspect ratios for all. The windowed top bar hidden is a personal favourite as the bar just pops up quite satisfyingly. All-together I have tried to realize all the relatively easy improvements, and implement those. Some include: all the cheats, zooming in/out and perhaps the most of all, screen size. The last was a result of good thinking ahead of time. In the beginning I decided to just set a base of 250 pixels which would be multiplied with the aspect ratios for the corresponding width and height. Then when I had done all the aspect ratio manipulation, I realized that I could just change the base to allow different sizes of windows. Created a button for it with a couple of sizes to fill all needs, and that was it. Because I had not been using some arbitrary numbers for the screen width and height but a multiplication of a common base, both introducing the full aspect ratio support with screen clipping, and size manipulation were relatively easy fixes. In this way I've tried to keep all options open and build the whole program around future development and scalability.

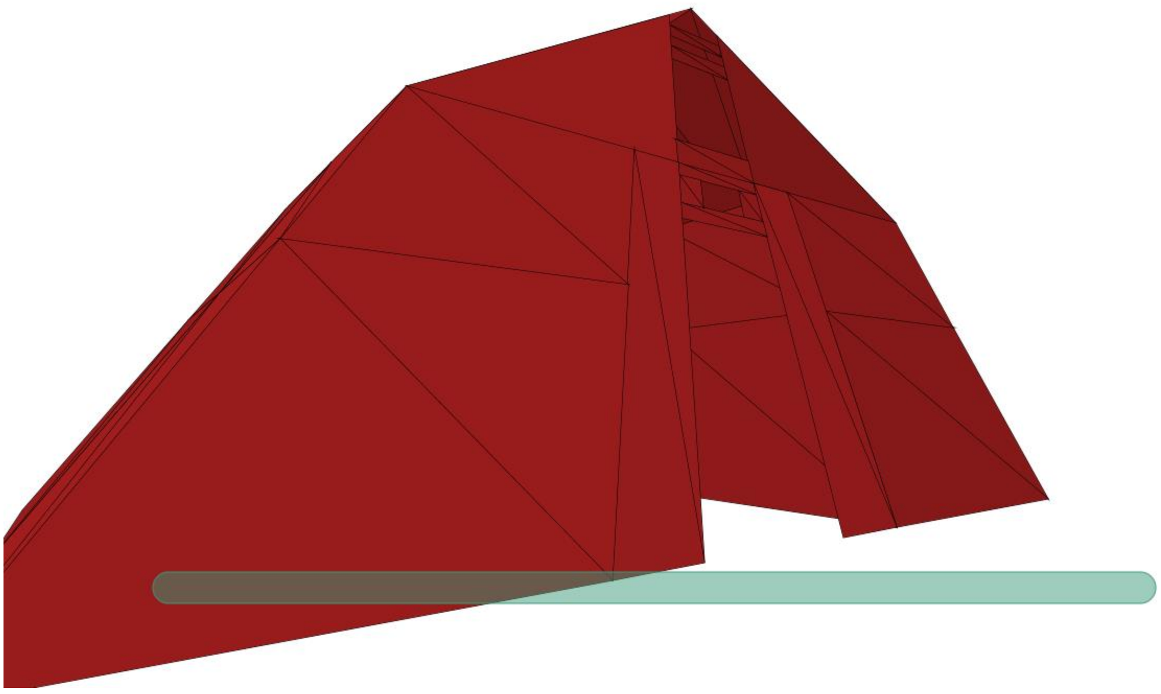
10. Conclusion

I think the project goes above and beyond the scope present by the project definition. The class structure is well defined with a minimal amount of anything not requiring ScalaFx done in the main file. Otherwise the different objects each serve their purpose and are well confined in their integral scope. The only failure on the project is the intersection math. To problem lies deep within the projection and its “e” plane, which I could not alter in any meaningful way to curtain the clipping. If I’d do this project all over again, I still wouldn’t do it just according to the scope with limited 2D coordinates, but I would do it on a different projection. Otherwise I would do everything the same. Using ScalaFx is the best option available, when the language must be Scala. Without this limitation I would have probably first looked at the possibility of just making it web-based, which of course I would have had to learn as well. CSV is the perfect file for this simple 3D application, but if I would be doing a much bigger world, I would format it to make use of duplicate points. In future there is no options for continuing development because of the problems with the projection and obj file loading, but if those got to work, I would try to optimise the running, to do less iterations over the triangles before they are ready to be displayed.

Following is some screenshots which can also be found under “progress” :

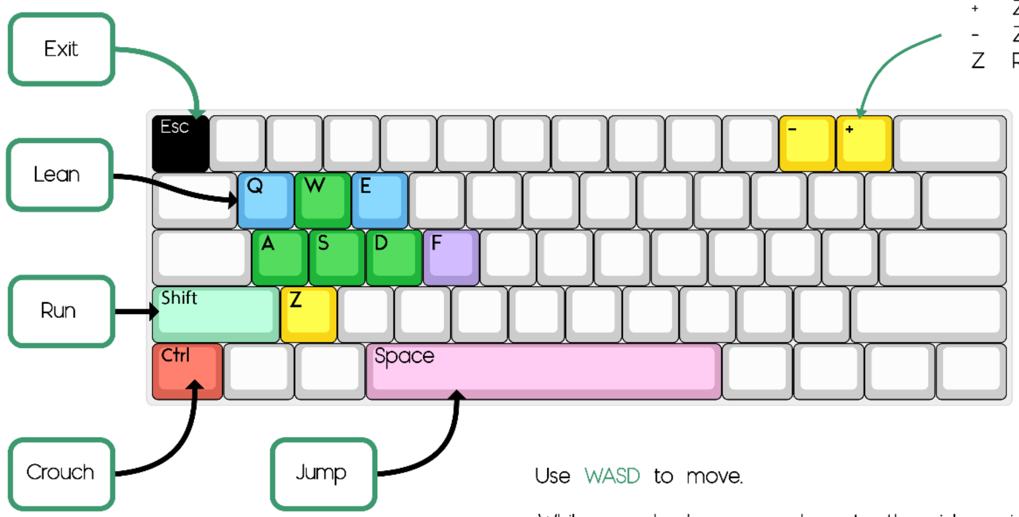







F to exit/enter fullscreen mode.

+	Zoom in
-	Zoom out
Z	Reset zoom level



Use WASD to move.

While crouched you can lean to the sides using Q and E.

13. Sources

https://stackoverflow.com/questions/5666222/3d-line-plane-intersection

https://stackoverflow.com/questions/328107/how-can-you-determine-a-point-is-between-two-other-points-on-a-line-segment

http://www.ambrsoft.com/TrigoCalc/Plan3D/PlaneLineIntersection_.htm

http://www.songho.ca/math/line/line.html#intersect_lineline

https://stackoverflow.com/questions/5666222/3d-line-plane-intersection

http://eguruchela.com/math/Calculator/shortest-distance-between-point-plane

https://stackoverflow.com/questions/6615002/given-an-rgb-value-how-do-i-create-a-tint-or-shade

https://www.youtube.com/channel/UC-yuWVUplUJZvieEligKBkA

https://en.wikipedia.org/wiki/3D_projection#Perspective_projection

http://www.lihaoyi.com/post/BenchmarkingScalaCollections.html

https://docs.scala-lang.org/overviews/collections/performance-characteristics.html

http://www.scalafx.org/api/8.0/#package

https://docs.oracle.com/javase/8/javafx/api/toc.htm

https://docs.oracle.com/javafx/2/api/

https://www.scala-lang.org/api/2.12.3/index.html

https://www.youtube.com/user/DrMarkCLewis

https://www.youtube.com/watch?v=LzhNsd9ut_4

https://www.youtube.com/watch?v=ktpeW2m4rtU&t=407s

https://www.youtube.com/user/thenewboston