Project Manual. 4/2019

threeD

Teemu Saravirta

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 was in my opinion executed on hard-difficulty as the scalability is virtually limitless, and the engine is able to handle all kinds of different shapes.

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>

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. I’ll tell about the re-scaling and re-sizing of content in [--------------------], but 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 tickrate 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.

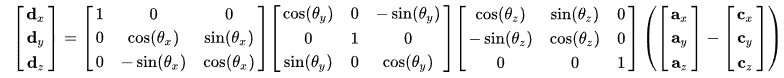
Projection

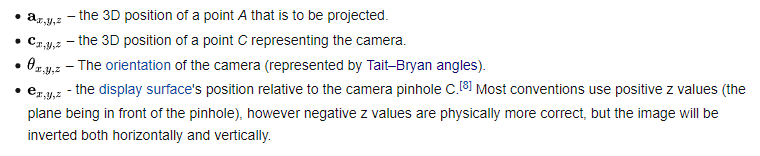
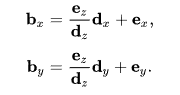
Home to the conversion of CSV data, and the main planar projection, as well as updating the data to draw each cycle.

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.

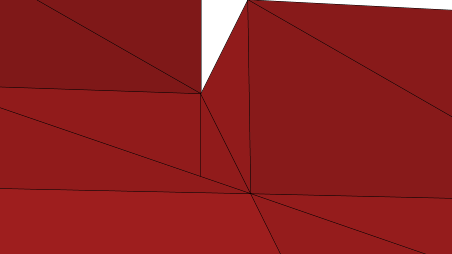
The following is the planar projection which I ended up using. <https://en.wikipedia.org/wiki/3D_projection#Perspective_projection> *link can also be found at the end.*

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.

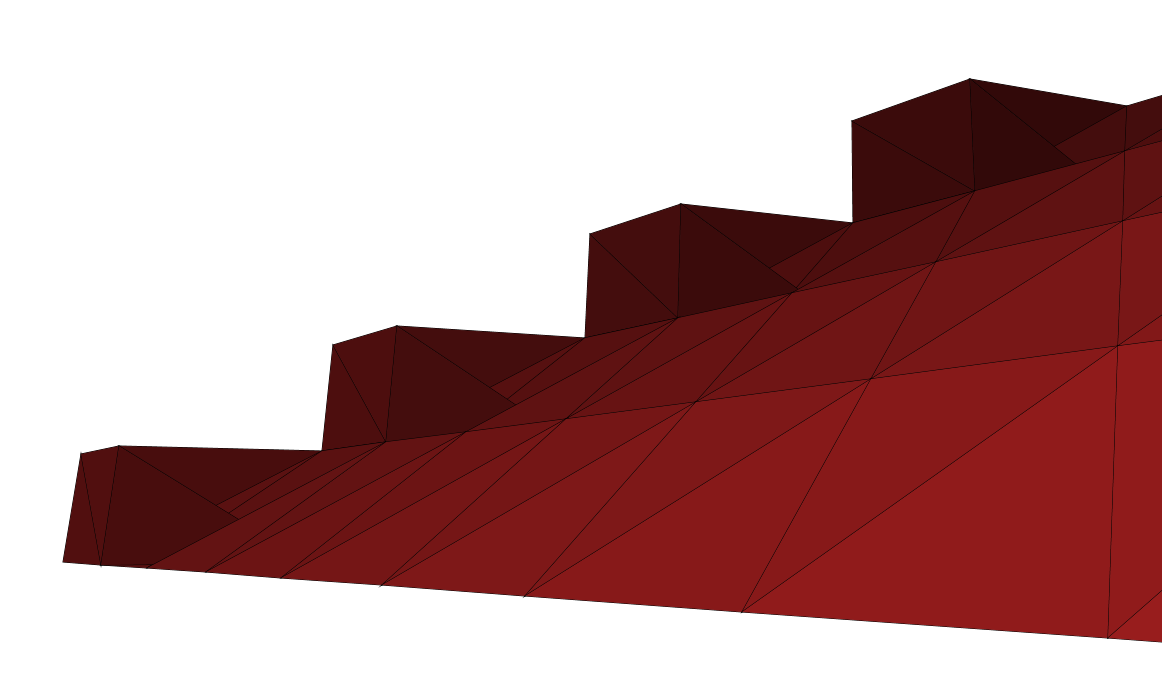




The projection always gives the triangles in order of the distance to the player, and they are drawn accordingly. They are sorted in the conversion process according to the combined distance of their end points to the camera.

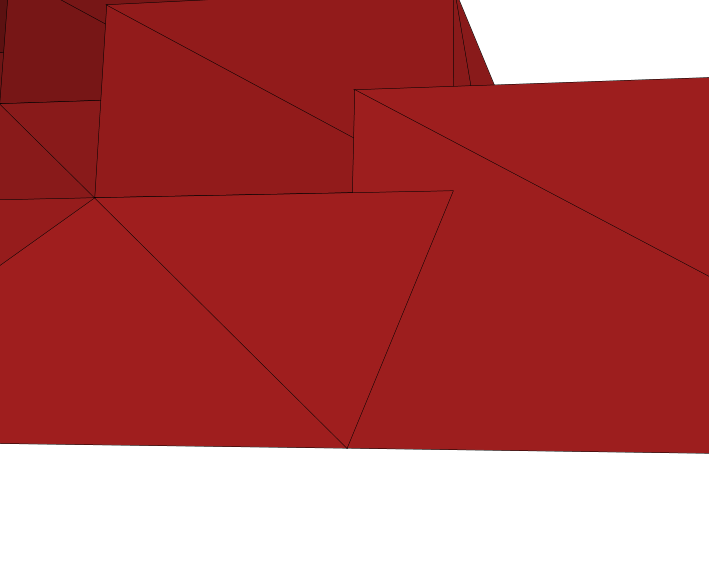


It works most of the time and for the projects I made it only messed it up very rarely. I’ve gone through alternate methods. If I emphazise the furthest away point the result is pretty much the same but with more computational resources wasted. If I only take the furthest away points:



The result is more immerse-breaking.

Same with taking the closest one.



6. 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.

7. 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.