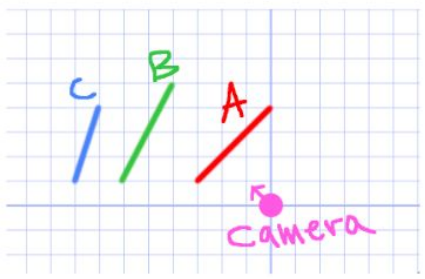
Andrew Weller, Trey Nevitt

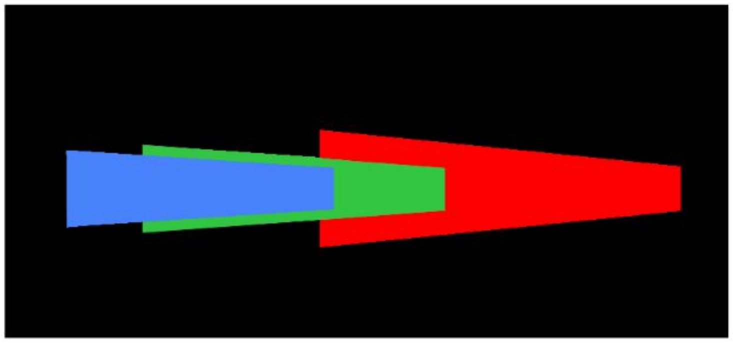
# Demise - Data Structures Final Project

Before reading this, It would be beneficial to watch a short video which showcases the project's abilities. This can be found either in the sources files for this project (Titled Project_Showcase.mp4) or at this youtube link.

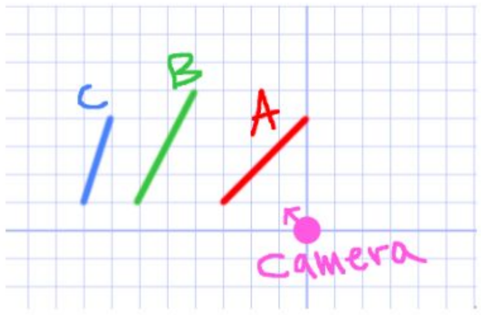
When designing a 3d application, whether that is a game or CAD software one of the most important things to do is to make sure that objects can successfully be rendered in 3 dimensions. Getting one polygon to render in 3d is a pretty easy task actually. If you have access to a window of sorts to draw on, all you have to do is what's called a "homogeneous transform" on each vertex of the polygon. Where is the vertex is [x,y,z], this becomes [x/z, y/z, 1].

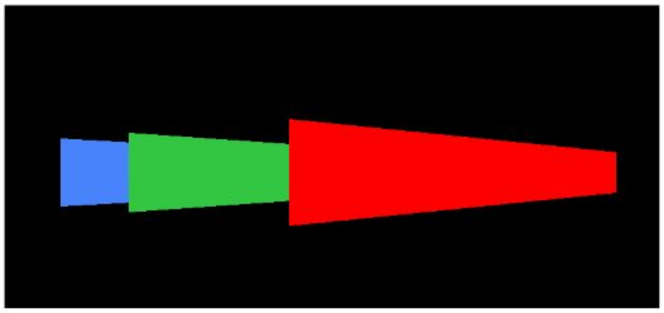
But have you ever seen a game where there is only one polygon on screen? Sounds really boring. Or a cgi effect that involves just 1 triangle on the screen? This is not going to cut it. We have to add MORE polygons to our 3d scene in order to make anything actually worth while. This causes a problem though. We know how to draw 1 polygon on-screen, but when there are two, we now have to know which order to draw them in. We could end up with scenarios where an object behind another object is actually drawn in front of it. This is unacceptable and can be hard to look at for our brains. This is an example of that happening.





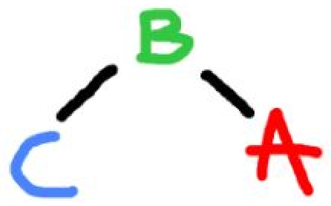
This is no good. We want to accomplish this.





As you can see in the second example the walls are rendered in the correct order. Not only is this better to look at, it is a more factual representation of how the world is perceived through the human eye. This is an important concept for 3d software which can often be jarring to look at for the untrained eye.

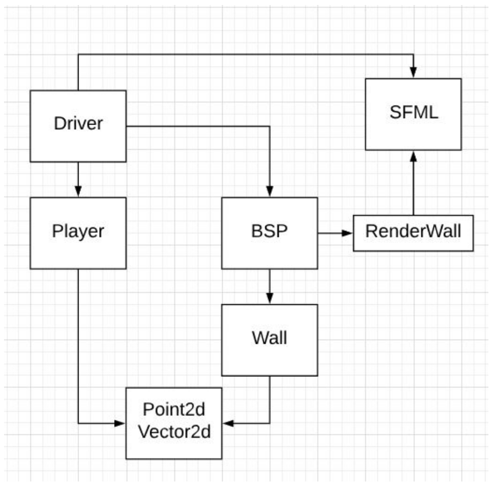
This is where the data structures come in. We can use a little help from our friend the binary search tree to help us achieve the correct rendering order. All we have to do is store walls inside of a binary search tree which is sorted **front** to **back.** There is a special name for this kind of tree in 3d graphics, the binary space partition.



Here is what a tree of the scene in the last page might look like. It could be in the reverse order as well, with A as the left child of B and C as the right child. How we define front and back for any individual wall doesn't matter as long as we are consistent throughout the entire construction of the tree.

Then, in order to draw the walls in the correct order on screen, it simply becomes an in-order traversal of the tree. We start at the root, and if the observer or camera is in front of the root, we first draw everything behind the root, then we draw the root itself, then we draw everything behind the root. This is done recursively on the whole tree and it works extremely well. It allows us to render the whole scene in the correct order in O(n) time. This is much faster than other naive approaches such as sorting the scene by distance from the camera.

We applied these principles in order to make a 3d maze crawling game. Where the player traverses through a maze in a first-person perspective in 3d.



That's how the project works. The project actually has a very simple structure though. The heavy hitters of the program are the BSP class (Binary Space Partition) which stores the walls, and the RenderWall function which is responsible for rendering a single wall onto the display.

The Wall and the Player (which is the camera/observer) classes use utility classes to represent points and vectors. These classes are mathematical objects which represent the game on a 2d plane.

This is all pulled together in the driver file which is where the BSP is constructed from a map file and then the game is rendered and played in a small game loop.

For results of the project you can see pictures in the screenshots folder of the project, or you can look at the video linked at the start of this report.