From One Dimension to Another: Raycasting

Project Description

I have designed a two-dimensional raycaster in Haskell using the Simple DirectMedia Layer library. Raycasters are a popular yet somewhat antiquated way of rendering pseudo-three-dimensional environments. This is not to be confused with raytracing: a more modern technique to render reflections and real-time graphics. Raycasting follows a much simpler algorithm, and was designed to give depth to games such as Id Software's *Wolfenstein 3D* in 1992. In general, raycasting is extremely fast, which was necessary for the limited processing power in home computers at that time. My approach, however, is different from Wolfenstein's; they use a grid-like layout to cast rays and determine points of intersection with objects in the plane. My algorithm, on the other hand, is to compute the closest intersection point of all objects in the two-dimensional plane, then cast a ray out to that point.

The player has a position pair (x, y), at which a ray r is fired at some degree α. Along the line segment generated by r, up to a maximum distance ρ, we compute the intersection points among each line segment of all objects in the plane. Then we take the minimum of these, say p_m, and set the ending coordinate pair of r to p_m. This process is repeated for n rays, fired from a field-of-view θ spanning from $[\alpha - \theta/2, \alpha + \theta/2]$. After generating the list of rays R comes the pseudo-3D projection. Every ray $r \in R$ has a distance from the player δ_r, and this distance determines the height of the projected column of pixels. Namely, each column of pixels corresponds to one ray, and δ_r determines how far or close we want to simulate the rectangle to be from the player. In summary, the farther away the point of collision is from the player, the smaller the rectangle. In this project we chose to map textures onto the rectangles rather than just solid colors, which are assets taken directly from Wolfenstein 3D, since it is an open-source project. Mapping textures onto the rectangles induces a performance penalty, since we must splice the textures, one column of pixels at a time, onto the rectangle. Though, the use of textures goes a long way in making the environment more "life-like."

The primary goal of this project was to create something that students who may be learning Haskell for the first time could design with relative ease. The original, non-textured variant of this project was only around 200 lines of code, making this an extremely worthwhile project, since it introduces I/O actions, monadic operations, mapping, filters, reduction, and more. Including textures, file I/O, comments, and other documentation, we counted 359 lines of code.

Project Features

Users can create a map with numeric codes that correspond to projected textures in the plane. For example, the following text can be saved as a file, say, `m2.map`, and it will draw a boxed environment with a "plus" in the center. We include this as an example in the `assets` directory. Each numeric code, from 1 to 5, represents a different texture under the `assets/sprites` directory, all of which originate from Id Software's (open-sourced) *Wolfenstein 3D*.

```
1111111111111
1000000000001
1000002000001
1000002000001
1000222220001
1000002000001
1000002000001
1000000000001
1111111111111
```

Players can move through the map using the arrow keys, and can change the facing angle via the 'A' and 'D' keys. To stop moving, press the 'P' key. Note that only one key can be pressed at a time with the current SDL key bindings setup. The direction that the player moves in respects the top-down perspective rather than the projection. This means that there is no notion of moving "forwards" or "backwards", but rather moving up, down, left, and right, which may lead to some initial confusion.

Source Code

The source code is available at `https://github.com/joshuacrotts/raycaster-haskell`. It includes a relevant README and a GIF of some recorded "gameplay".

Haskell Development

The use of Haskell made this project, at least initially, harder. As I continued development, however, and I grew to appreciate the more complex higher-order functions that Haskell provides, the easier it was to develop. I still believe that designing this type of project benefits more from an object-oriented language, but writing it in Haskell truly showed that I can write a complex project with only a few hundred lines of code. Lines of code may not be the best metric for measuring the complexity of a project, but they are absolutely a factor!