New Mexico

Different 3D Rendering Methods

Final Report

April 6th, 2021

Team 28

Los Alamos Middle School

Team Members

Andrew Morgan

Teacher(s)

Nathaniel Morgan

Project Mentor

Nathaniel Morgan

Overview

There are many ways to render a 3D world onto a 2D computer screen. Some of these methods include, but are not limited to, perspective projection, forwards ray tracing, backwards ray tracing, backwards ray marching, forwards ray marching, path tracing, ray casting, isometric view, forwards ray interception, backwards ray interception. Of these, I have created codes with the perspective projection, backwards ray tracing, backwards ray marching, isometric view, backwards ray interception, and ray casting methods, in python3 and the backwards ray marching method in c++ (using shadertoy.com).

Vocabulary

Mesh: A bunch of vertices that create faces that represent shapes
Height Map: A 2D (or 3D) image/texture which represents height (black is low terrain and white is high terrain)
Signed Distance Function: A mathematical function which returns the distance from a given point (x, y, z) to the surface of a shape.

Perspective Projection (Python3)

Perspective can be created by projecting the points of a mesh of triangles onto a 2D screen, and filling in those triangles using a method called rasterizing. To project those points, simple linear equations can be used. A projection matrix can also be used, which all the vectors (positions) of the triangles are multiplied by. The lighting effect in the image of figure 1 was created by altering the color of each triangle based on its vertical change. I generated the mesh by creating a detailed plain and displacing the height of the different points on it based on a height map, which was generated using my implementation of the perlin noise algorithm. My code for the perlin noise can be found at https://github.com/AndrewDMorgan/PyVectors-and-PyMarching/tree/main/PyVectors. The perspective projection was about 250 lines (for the version using basic linear equations), and the perlin noise/PyVectors code is 2,500 lines. The perspective projection (using a 4 by 4 projection matrix) code took about 700 lines. The number of lines includes the comments and spaces (this applies on all the rest). The picture in Figure 1 was created using my code for perspective projection. You can also find this projected on https://replit.com/talk/share/3D-Game-Engine/57403. References [1] - [4] are helpful for learning more about perspective projection.

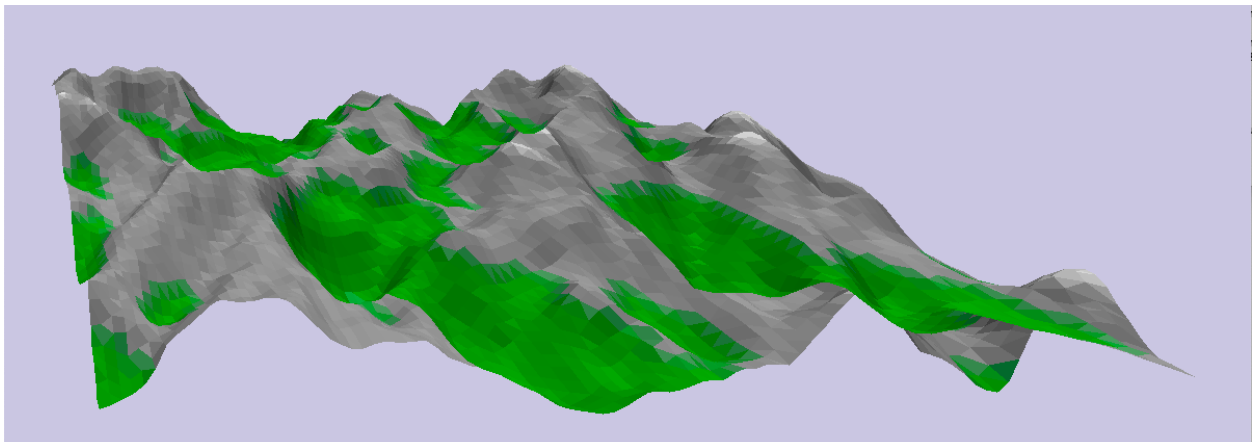

Figure 1 - A rendering of a mountainous terrain using the perspective projection method in my code is shown.

Backwards Ray Tracing (Python3)

Backwards ray tracing is a method where you cast a ray which travels through the scene and interacts with objects to determine its color. Each ray steps forward with a constant amount until a collision occurs where it is stopped (absorption), passes through and refracts, or reflects. Each of the ray angles is offset away from the center to create perspective, which is the effect that makes far away objects smaller and closer ones bigger. This effect also makes faraway objects move slowly when walking to the side, and closer ones move more quickly, and is called parallaxing. Also, perspective is why an infinitely straight wall will appear to shrink inwards. When perspective is not present, and the rays travel straight away from the screen, that is called an orthographic view, and when using the perspective projection method, it's called an orthographic projection. In the image shown in Figure 2, I have multiple solid colored planes, multiple mirrors, and a pane of blue colored glass. The code to render this image was about 150 lines long. My code for this implementation can be found on https://replit.com/talk/share/Ray-Tracing-Bata/57953.

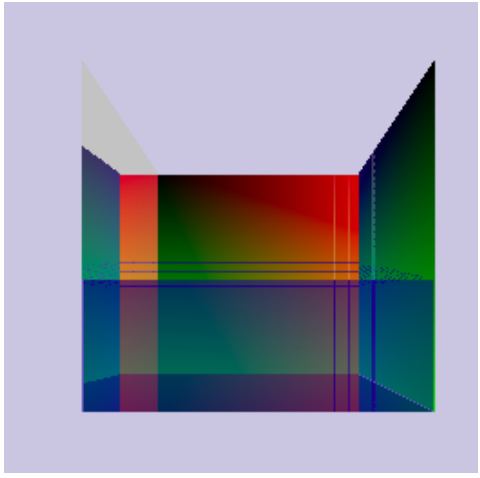

Figure 2 - A rendering of mirrors, solid planes, and tinted glass using the backwards ray tracing method implemented using my code.

Backwards Ray Marching (c++)

Backwards ray marching is the same as backwards ray tracing but that rays don't step along by a constant number but rather the distance to the closest object. The downside to this is that all shapes require a signed distance function. The upside to this is that instead of stepping hundreds of times per ray, you may only have to step 30 times. An optimization for backwards ray marching is to instead of waiting till the distance to the closest object is zero, waiting till it's less than or equal to 0.1. This code was about 450 lines. My code for the implementation of this method can be found on https://www.shadertoy.com/view/tdtfW4. Note that it is running slowly because of the heavy use of perlin noise. Also, that perlin noise code was not created by me. The image of Figure 3 shows the ray marcher with shadows and light based on surface normals. References [6] - [7] are helpful for learning more about ray marching.

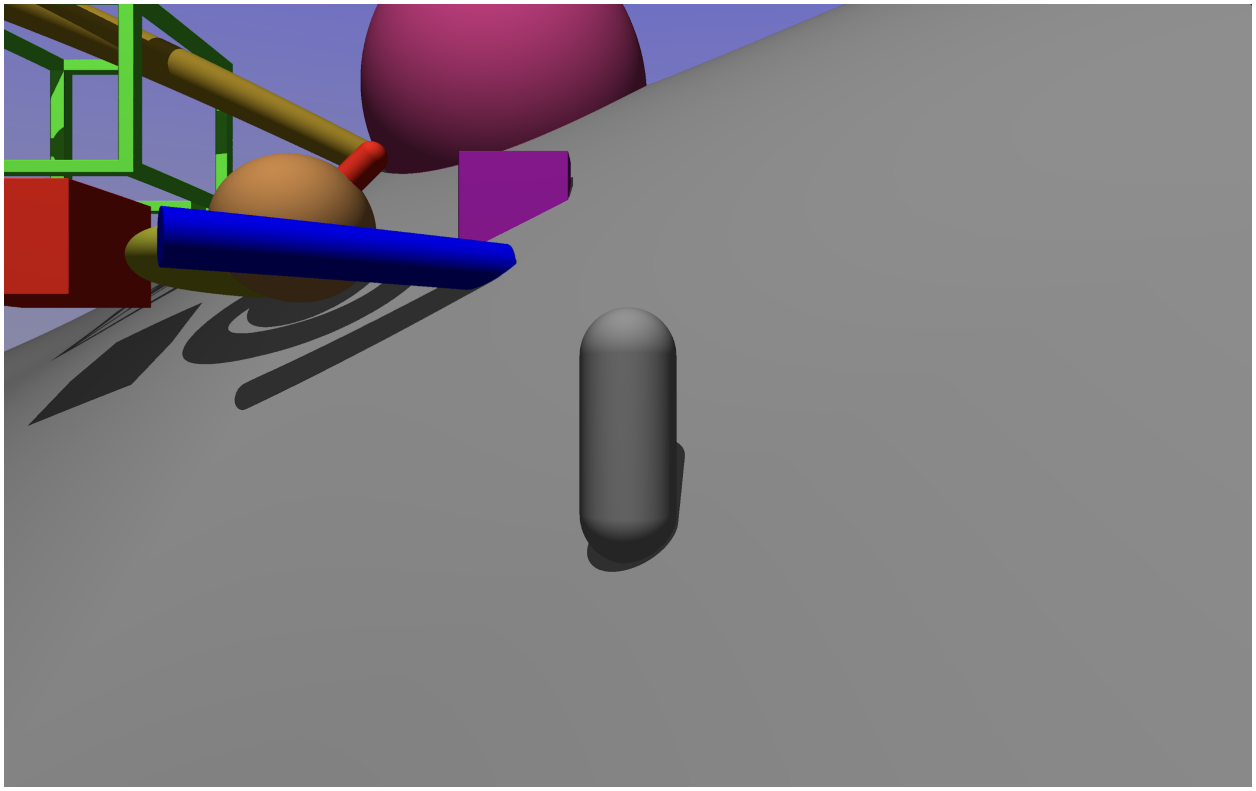

Figure 3 - An image from my code created using the backwards ray marching method. The only part of the code not written by me was the code for the perlin noise. This version was created in the c++ programming language.

Backwards Ray Marching (Python3)

Backwards ray marching in python3 is the same method and implementation as in c++ version, but it runs more slowly than python3. Python3 is compiled while running whereas c++ is compiled and then run. Note that the image of Figure 4 is much more basic and this is because python3 is much slower making it harder to add some of the features seen in Figure 3. The code

I created for this rendering method is about 100 lines. It also used my PyVectors code, which is 2,500 lines long. References [6] - [7] are helpful for learning more about ray marching.

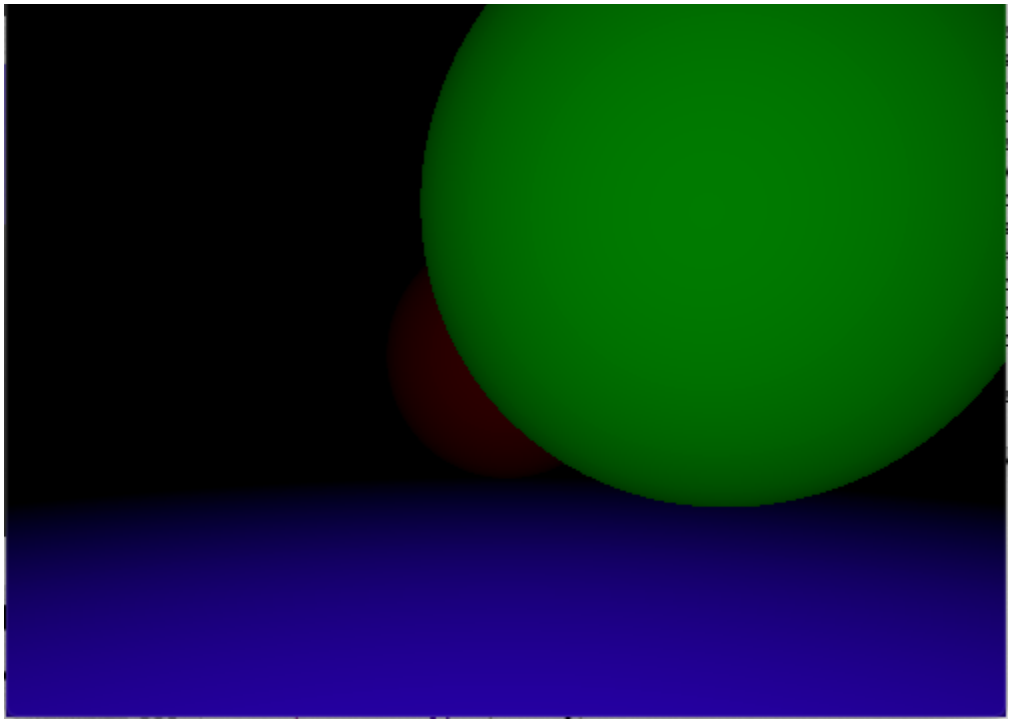

Figure 4 - An image from my code created using the backwards ray marching method. This version was created using the python3 programming language.

Isometric View (Python3)

An isometric view isn't created through complex math or code but rather by creating art that appears to be rotated 45 degrees, with depth being incorporated within the art. The images are rendered at an offset that varies to allow the tiles to seamlessly repeat. The process of rendering the images (tiles) is sped up by only rendering blocks that have a transparent block to one of their sides. The image of Figure 5 shows what an isometric view looks like. The tiles in Figure 5 were created by me and can be found on my piskel (a free online pixel art software) account (https://www.piskelapp.com/p/agxzfnBpc2tlbC1hcHByEwsSBlBpc2tlbBiAgOCAjYLPCQw/view). The code I created for this rendering method is about 550 lines (over multiple files). It also uses my PyVectors code (for perlin noise for shaping the land and caves) which is 2,500 lines long.

Figure 5 - An image created by my code using an isometric view. The terrain (including caves) was created using my version of the perlin noise algorithm.

Backwards Ray Interception (Python3)

Backwards ray interception is a method similar to backwards ray tracing or marching but faster. It is also much more limited. In this method, the rays that are cast from the camera are represented by linear equations. Then, for each of the rays, it uses linear interception equations on each of the shapes to see if/where the ray hits something. This is incredibly fast because the computer doesn't have to do many calculations. The downside is that very few shapes can be represented by linear interception equations. The image in Figure 6 shows the results from my code for the backwards ray interception method rendering an image in real time (meaning the render is shown live and happens very quickly) in python3. The code I created for this rendering method is about 100 lines long.

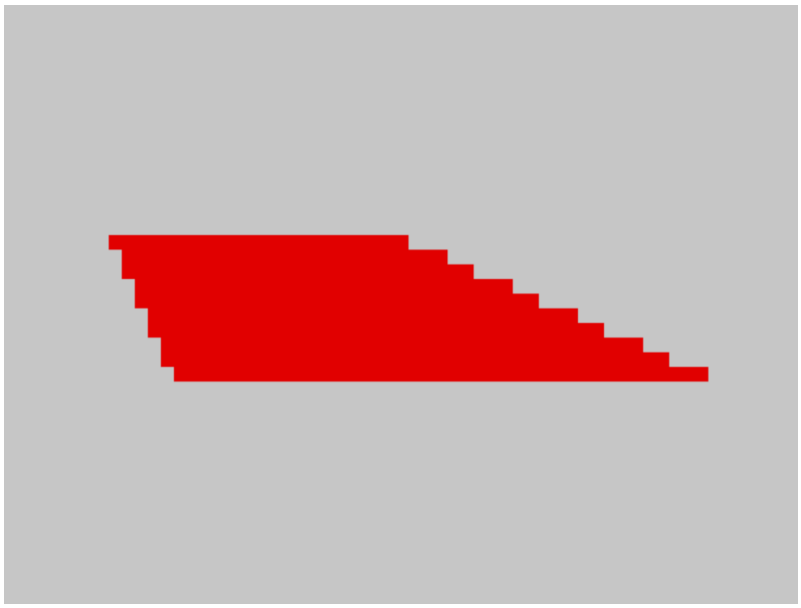

Figure 6 - An image created by my code using the backwards ray interception. The only obstacle in the scene is a flat plane. Note that the render was low resolution to allow for the render to run in real time and allow it to run at an average of 1/58 of a second per frame.

Ray Casting (Python3)

Ray casting is the original method for rendering 3D graphics. It was used in old arcade games. The first game to use the method was Wolfenstein 3D. The method only allows for 2D levels to be rendered and allows for only left and right movement (the player and camera direction) without major modification (which also slows the render down some). This method is the same as ray tracing but rays are only cast horizontally (reducing the number of rays increasing render speed). Since the scene is made up of solid square tiles, the rays can step along one tile at a time reducing the number of steps and therefore increasing the render speed. The textures for the walls can be found on my piskel account https://www.piskelapp.com/p/agxzfnBpc2tlbC1hcHByEwsSBlBpc2tlbBiAgOCEycP-CQw/view. The result from this code is shown in the image of Figure 7. The code I created for this rendering method is about 800 lines and uses my PyVectors code which is 2,500 lines long. Reference [5] is good for learning more about ray casting.

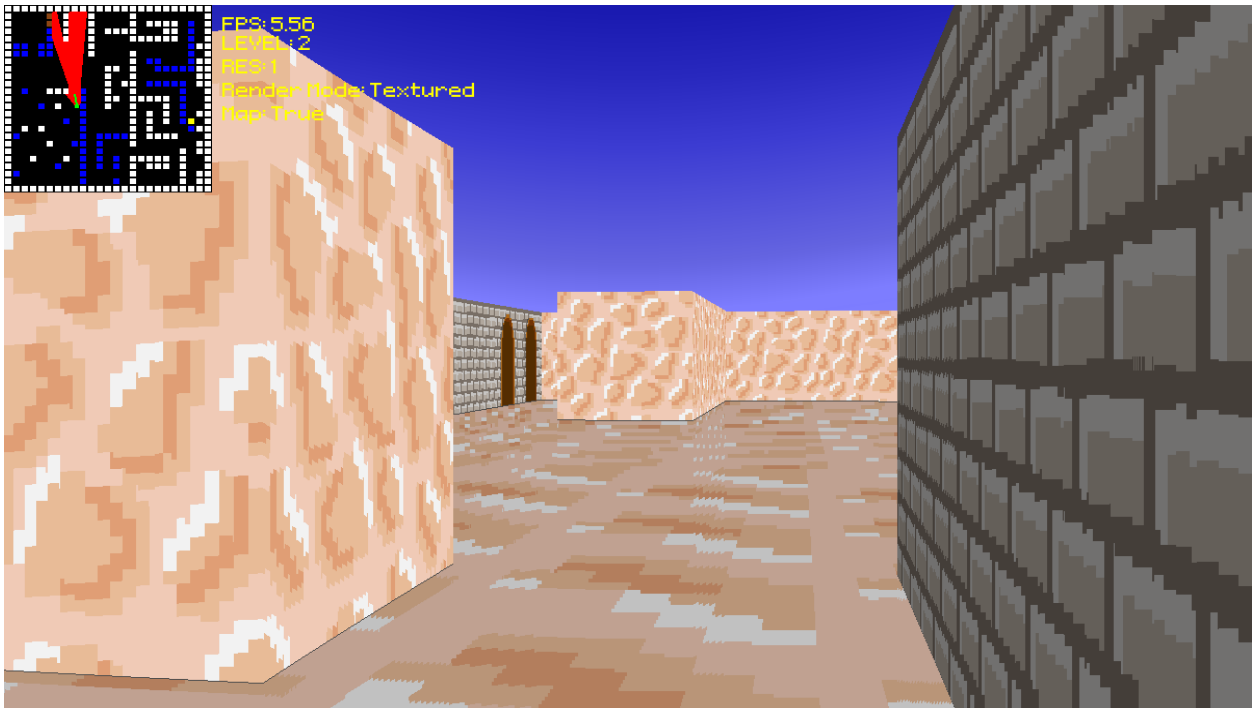

Figure 7 - An image created by my code using the ray casting method. This image was rendered in real time meaning the results can be seen within a very short time (like 1/30 of a second).

Results

Method	Perspective Projection	Backwards Ray Tracing	Backwards Ray Marching (c++)	Backwards Ray Marching (Py3)	Ray Casting (Unoptimized)	Isometric View
Time	5-20 FPS (depending on the scene)	1000 seconds	60 FPS (Could Be Faster But Is Clamped)	120 Seconds	10-20 FPS	5-40 FPS (Depending On Blocks Rendered)
Lighting	Based On Steepness	None	Based On Normals And An Extra Ray Towards The Sun	None	Based on the render face's orientation	Only Shadows Within The Tile Textures (Baked Lighting)

Method	Backwards Ray Interception
Time	58 FPS
Lighting	None (Could Add Advanced Lighting Easily)

Conclusion

In conclusion, the fastest method that allows for most/all shapes to be rendered is perspective projection. Ray tracing is an alternate solution, it is slower but can give more visually pleasing and realistic results. The fastest method overall when not caring about the number of supported shapes is backwards ray interception. Ray marching comes after that. Ray casting is one of the fastest if you don't care about up and down movement for the camera and player (and all levels are limited to 2D). An isometric view isn't very good for 3D games as it lacks a sense of depth due to the tiles and tile renderer having an orthographic style. Despite that, an isometric view is great for adding a sense of depth to a 2D game.

References

1. https://stackoverflow.com/questions/724219/how-to-convert-a-3d-point-into-2d-perspective-projection#:~:text=Here's%20a%20very%20general%20answer,(X'%2C%20Y')
2. https://www.cse.unr.edu/~bebis/CS791E/Notes/PerspectiveProjection.pdf
3. https://www.scratchapixel.com/lessons/3d-basic-rendering/computing-pixel-coordinates-of-3d-point/perspective-projection
4. https://www.youtube.com/watch?v=ih20l3pJoeU&t=6s
5. https://www.youtube.com/watch?v=gYRrGTC7GtA&t=904s
6. https://www.youtube.com/watch?v=Cp5WWtMoeKg
7. https://www.youtube.com/watch?v=PGtv-dBi2wE&t=0s

Code

The code for perspective projection (using marticies), ray casting, and isometric views can be found at https://github.com/AndrewDMorgan/Super-Computing-Rendering-Method-Files and those and others on my replit account: https://replit.com/@TurtleAndrew. The ray marching code can be found on my shader toy account: https://www.shadertoy.com/profile?show=shaders. Finally, the PyVectors code can be found on my github.