Interior Mapping (Nick Forester)

1 Introduction

Buildings are a staple in a lot of video games. From first person shooters all the way to city building simulations, buildings are most likely going to play a pivotal role in the world and level design. In order to cut back on computational overhead, a lot of these buildings are rendered as empty shells that have no interior. There is nothing more immersion breaking in a game like Grand Theft Auto to see sprawling skyscrapers with blacked out windows. Interior Mapping is a technique thought up by Joost van Dongen in 2008 which simulates building interiors from a distance completely through the rendering process. This eliminates the spatial overhead of using geometry for the buildings, and also removes the burden from the CPU by offloading the responsibility onto the GPU.

2 Getting Started

The first requirement is to know where the camera is facing in world space. Since the camera's world position and the pixel's world position are known, getting the camera's direction is a simple as subtracting the pixel's position from the camera's position. The next step is to simulate planes along the xy-axis, the xz-axis, and the yz-axis. These planes represent the floors and walls. Some arbitrary value can be chosen as the distance between each wall and each floor. A plane can be represented as a normal vector and point on the plane. Since each of these planes are axis aligned, their normal vectors are easy to find (floor's normal is (0, 1, 0) which is up). To find a point on the plane, it requires knowledge about which plane is of interest for the current pixel. For example, to find the relevant plane for the ceiling above the current pixel, the following equation can be used.

ceil(pixel_y_position / distance_between_floors) * distance_between_floors.

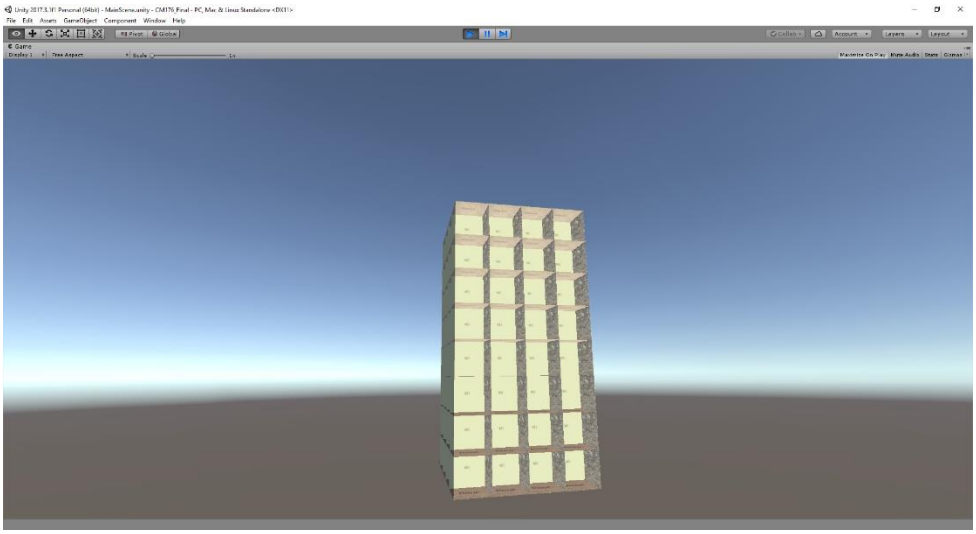
So for the current pixel, a point on the ceiling plane is (0, ceil(y / d) * d, 0). Now that a point on the plane and the normal of the plane is known, the distance between the pixel and the plane along the camera's direction vector can be found using the following equation.

dot(point_on_plane - pixel_position, plane_normal) / dot(plane_normal, camera_direction_vector).

The previous calculations should be performed on every surface within the "room" the pixel represents, and whichever surface has the shortest distance should be the surface used to color that pixel. With all of the information that has already been discovered, it is fairly trivial to find the intersection point with the ray from the camera and the plane. Since the distance is known, it can be found with the following equation.

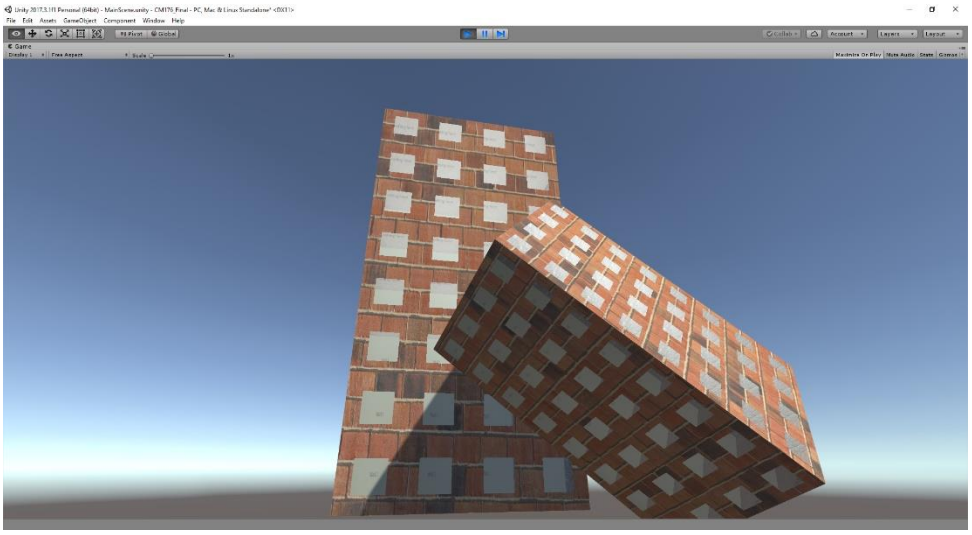
Intersection_point = pixel_position + (camera_direction_vector * distance_from_pixel_to_plane).

The last step is to use the intersection point as UV coordinates for a texture lookup. If the plane is on the xy-axis, then only the x and y values are needed for the UV. The xz-axis would use x and z values, and the yz-axis would use the y and z values. The color retrieved from the texture lookup should be used as the color for the current pixel.



3 Adding More Detail

In order to render the exterior of the building, a texture detailing the window positions is required. This texture should be black everywhere except where windows belong, which should be white. Then this texture can be sampled using the UV of the building, and if the color is black then an exterior texture should be rendered instead. If the color is white, then the interior texture should be found. In addition, some kind of glass texture can be mixed with the interior texture to result in a window effect. This window effect can be amplified by adding lighting effects such as reflections.



Teleportation Shader (Samuel Barish)

Keywords:

Teleportation, Shader, Teleport

1 Introduction

Teleportation has been used in various mediums from cinema to video games. Computer graphics plays a strong role in making these effects possible. Many video games use these effects to make games more immersive and fantasy oriented to let players experience dynamic and interesting gameplay. Teleportation obviously isn't possible in real life, but through computer graphics, engineers could experiment with what teleportation might look like. Shows like Star Trek show teleportation with a beaming process and a futuristic aspect and I wanted to replicate that effect. I originally was going to do my project on smoke but decided that I wanted to venture more into the idea of teleportation. So, to keep part of my project still relevant to smoke I decided I used a smoke styled noise function to make the object disappear in a smoke shaped pattern then reappear somewhere else.

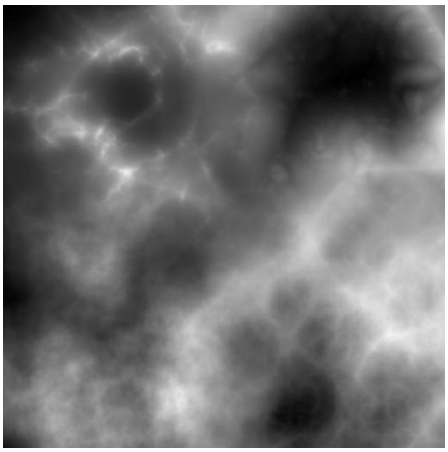
2 Disappearing Act

The first step to make the object teleport is making it disappear. To start with this "disappearing act" we add a noise function texture. I used a noise function texture that resembled smoke (fig.1). Use this texture to determine where we want the object to start disappearing first. I made the object disappear starting at the darkest points of the noise texture then move to the lightest parts until the whole object has disappeared, but you could change that around. To do this, first read from the textures red values and add that to the speed you want your object to disappear at. I also added time to this speed to make the object continuously disappear and reappear to continue showing teleportation. Then set that value to the alpha. The next thing to add is the gradient that goes from top to bottom to make the object disappear from top to bottom as if teleporting to a new location. To make this effect, a direction vector must be made to control the disappearing effect. Then make a vertex shader that calculates a new alpha value that is based on this effect's progression through the object. Create a gradient vector using the lerp function in unity using the gradients start position, the gradients end position and the speed of the effect. Subtract the gradient vector from the current vertex and dot product that with dissolve direction and finally multiply that by the reciprocal of the size of the scan size. Send that value to the fragment shader and combine it with the texture factor and set this new value to be the new alpha value. Finally, to finish this disappearing effect I added the effect to predict what parts of the object were going to disappear next and highlight that with a color. First shift the alpha value based on the factor of this prediction size and multiply it by how intense you want this effect, scale from 1 to 0. Change the color interpolation by the gradients factor. Finally, multiply shifted and modified alpha value by a gradient vector of new color interpolation, the end prediction number, and the color. Set this vector to the emission value and the effect is complete.

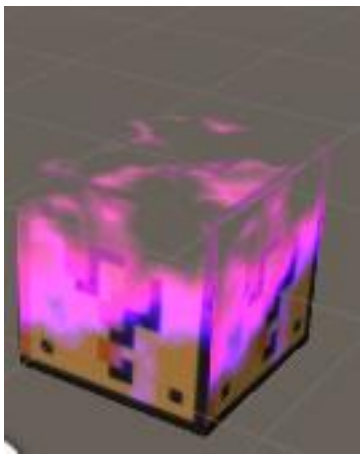
3 Teleportation

The teleportation is incomplete without the actual movement of the object though. To complete this in Unity some C# code must be added. Fist add a counter to the beginning of the class. In the update

function change this counter by 1 every frame. Then create if statements that change the position of the geometry using transform.position. If the counter is equal to some number of frames, then change its position then after another series of frames change the position back. These numbers should be set to a frame when the object is invisible to “teleport” the object to another position.



(Fig 1 smoke-like noise texture used for making the object disappear)



(Fig 2 a screenshot of the effect in action)

Volumetric Clouds (Eduardo Gaona)

1 Introduction

Everyone has seen clouds at least once in their lifetime. That being said for every game, video, or scene outside we need to make sure that our environment looks realistic. One of the main features are clouds. The main challenge I faced in tackling this task is that I had never used Unity. I came to learn and understand the powerful features that Unity has to offer because without it my volumetric clouds would not be possible.

2 Getting Started

The way I tackled this problem was by using Assets. I imported a couple of packages including, Particle Systems and Volumetric Clouds. With the particle systems asset I included a dust storm into my scene. In the Volumetric clouds I created a material and shader that was going to help texturize the dust storm. Once the material of the cloud I wanted was ready I implemented it to the dust storm. From then the dust storm turned into fluffy white clouds in the sky.

The technique that was used in the shader is raymarching. With the raymarching approach I was able to create 3D volumetric clouds in the scene. As you are able to see from the figure below I was able to create clouds with a black outline to portray realistic clouds in the sky.

In the shader we take into account many different attributes for the clouds. We are controlling the shape, color, animations, dimensions and a raymarcher in order to make the clouds. Within the shape aspect we are controlling the density of the clouds. We have to make sure as well that the clouds are spread out across the scene with raymarching.

Distance Fog (Shawn Edmond)

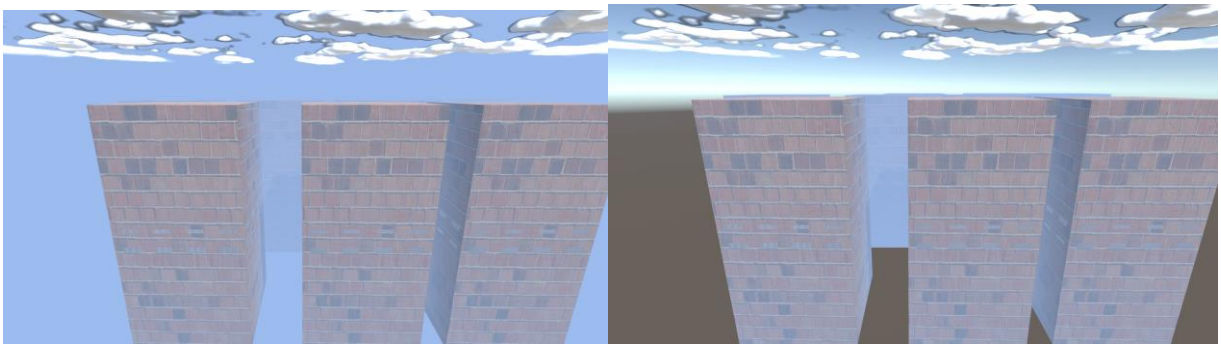
1. **Introduction**

Fog is used to great effect in games, serving varying purposes. In 3d games fog can be used to hide draw distance problems such as pop-in or clipping, such as in early Superman or Spiderman games. Fog can also directly enhance the game's atmosphere or directly serve as part of the gameplay, such as in Silent Hill.

2. **Getting Started**

Distance fog works by getting the world position of each texel, relative to the camera, and applying a gradient. The game for my 170 series makes use of this same effect in the Unreal Engine, and I initially assumed that I could use the same logic to add a fog effect in Unity. Unity works in a vastly different way so I was unable to do that. My next roadblock was getting the fog to work on the sky, as it was only working on meshes. After some thinking, the solution was simple. If I want the fogged meshes to disappear, I make the sky a compatible color. (In this instance I simply made them an identical color.)

(Fig 2 Without)



(Fig 1 With similar sky color)