P3DPGE Render Pipeline

As of commit 7e79421 (11/24/2020)

Written up by sushi

Structure

I believe it would be best to skim over how entity and mesh are structured and go into some detail about the god-awful structuring of Triangle.

Entity and Mesh:

Entity is the parent class of all drawn objects to screen including 3D and 2D objects. Entity contains basic information such as an objects position, rotation, scale, etc. Entity also holds a pointer to a Mesh object, which is currently (as I see it) only used for objects drawn in 3D .Mesh is a collection of Triangles and is the basis of all 3D objects. Mesh contains what is every 3D object’s Draw function and this is discussed in detail below.

Triangle:

Triangle is currently the least structured object as of right now. Each triangle holds an array of 3 points defining the triangles position in WorldSpace and another two arrays depicted the triangles position in ScreenSpace. It may not be necessary to have two arrays for this, but the second is meant to hold the triangles ScreenSpace position throughout each frame since the original ScreenSpace array would reset back to WorldSpace at the beginning of each frame.

Triangle also holds an array of 3 Edges and this may not be necessary since Edges are meant to only be used for calculations and aren’t to be seen as actual objects in the scene. I’m thinking I could just create Edge objects whenever necessary, such as in the *contains\_point* function.

The Flow

Render:

The render pipeline beings when *Render::Update* is called. *Render::Update* currently does three things.

First, Render makes a view matrix based off the camera stored in *Render’s* namespace. This view matrix orients the camera in the world and is used to tell the rest of the objects in the world how they’re oriented with respect to the camera.

In a for loop going over all entities in the scene, *Render* proceeds to call *Update* on each *Entities’ mesh*. The purpose of this Update function is to feed each mesh with the newest 3D information for that frame. The two main things that are being updated here are the camera’s position and the view matrix associated with it. The third thing is the Projection Matrix which could probably be updated only when it actually changes, but for now it is updated every frame.

In the same for loop, each *Entities’ Draw* function is called. The *Draw* function on each *Entity* is a virtual function that is only overridden when there is a special case such as a 2D object, in fact this is the only special case so far. Otherwise, it calls the default *Draw* which just calls *Draw* on the object’s mesh. Entity::Draw takes in two arguments; a pointer to olc::PixelGameEngine and a boolean telling it if wireframe is enable or not. Then we move into *Mesh’s* flow.

Mesh:

Mesh’s Draw function is currently comprised of all the steps required to draw the Mesh’s triangles and should be abstracted later. I will reference line numbers to guide you through the function.

Beginning at line 165 in Mesh.h the first thing Draw does is make a *visibleTriangles* vector which holds all Triangles who are deemed visible by the first visibility check. Then *drawnTriangles* is cleared and the temporarily lighting set up is declared.

At line 174 is the first visibility check. All it does is compute the dot product between a normal on the Triangle and a ray between the camera’s position and that normal. If the dot product is greater than 0 (meaning that the angle between the normal and the ray is less than 90 degrees) then the triangle is deemed visible and pushed into *visibleTriangles*. Before this, the triangles world points array is copied to its projected points array. This may be uneccesary and I’ll explain why later. Each triangles color is also set according to the dot product between the defined light direction and the triangles normal.

At line 184 we begin to clip Triangles if necessary and then project them to the screen. The first thing done of course is the clipping. I’ll will explain this algorithm more later, but in essence what it is doing is taking in a triangle, checking if it intersects with a plane and cutting it into new triangles if it does. After this, we loop through the clipped triangles and project them to the screen. We push each projected triangle into the *drawnTriangles* vector.

At line 202 we do our sorting of each triangle in *drawnTriangle*. This way of drawing triangles over each other is known as the Painter’s algorithm and simply sorts the triangles in drawnTriangles based off the z coordinate of their midpoint.

At line 208 we get to clipping triangles off the planes forming the fulcrum of the camera’s view. This whole clipping method was pretty much copied from Javid’s video and I have yet to get into it and actually understand why each part works, but as far as I know it just checks if a triangle clips with any of the four planes extruding from the camera and has to actually check each new triangle it makes against all planes which is why it does a whole bunch of weird stuff. At the top of this for loops I copy another set of points in Triangle called proj\_persistent and I’ll explain why and why this may not be necessary later.

At line 237 we finally get to actually drawing the triangles in the complete *listTriangles* vector. This vector just holds all the previous *drawnTriangles* as well as all the new ones formed by the clipped done previously.

Current Issues

The majority of issues currently lies within interacting with Triangles. When we want to select a Triangle in ScreenSpace we have to do this before we actually calculate anything to do with this, as *Input::Update* is called before all else. My solution to this was to create a new *proj\_persistent* array that holds a points position in ScreenSpace throughout each frame. The reason I have to do this is because of how we are passing the triangles to several different vectors throughout the *Draw* function in Mesh. I can not set the *proj\_points* in our original collection of Triangles because of this. This is also the reason I moved *drawnTriangles* outside of the *Draw* function because *proj\_points* is actually set properly here and this way I could access this vector from Input. Another issue I ran into with this was setting the colors and other properties of Triangles. Since *drawnTriangles* is the only vector that contains the right info and can be interacted with, it is where I currently have to set properties for Triangles. But of course this info does not carry over because *drawnTriangles* is reset every time the *Draw* function is called.

This entire problem also causes there to be a huge issue with actually interacting with Triangles, and in the future, interacting with objects in general. Because there is such a separation between the actual triangles and the ones we see in ScreenSpace (the Triangles currently get passed through 3 different vectors), interacting with a certain triangle and then trying to carry over what we did to it doesn’t work. Maybe it’s solved by actually checking every Triangle and not just the drawn ones? I’ll have to check this later.

There is a solution I have in mind for this, but I’m not entirely sure it would work and it would mean redoing how we do calculations on each point of a triangle. Currently, it’s set up so that every vertice (Vector3) does its own transformation on itself meaning that once that Triangle’s points all transform themselves, that object’s points are modified. This is why I had to make a new array for projected points and then do all the calculations on them. The solution I have in mind is instead of storing that info on Triangle and modifying it through there, I change all the transformation functions to be in *Math’s* namespace and have them return the modified point instead. This way it will be set up more like Javid’s video where he just makes a new variable for each step he does so it doesn’t keep modifying the triangles actual position. I’ll have to look more into this.

Another 2 issues that are sort of beyond me at the moment is Triangles being drawn in the wrong order at some points and entire faces of the cube or parts of it not being drawn. When coloring each triangle different colors is it obvious that at certain rotations of the cube the top or bottom face will begin to overlap the left or right faces to an extent, then go back to normal at some angle. The only place I think could be causing this is when we do the Painter’s algorithm sort. For some reason, the z-value of the midpoint of the triangle is being seen as closer than the one that is actually closer. I haven’t actually looked too far into it yet, but I will.

When it comes to entire faces or part of them not being drawn (when they definitely should be), I believe there is an error somewhere with how far away a triangle must be before it is clipped. The only place I know that defines this however doesn’t fix anything when I change it and I can’t think of anywhere else this would be an issue. It is obvious this is the problem because if you move the camera away from the triangle it will reappear. This problem is beyond me currently.