CMP203 Coursework Report

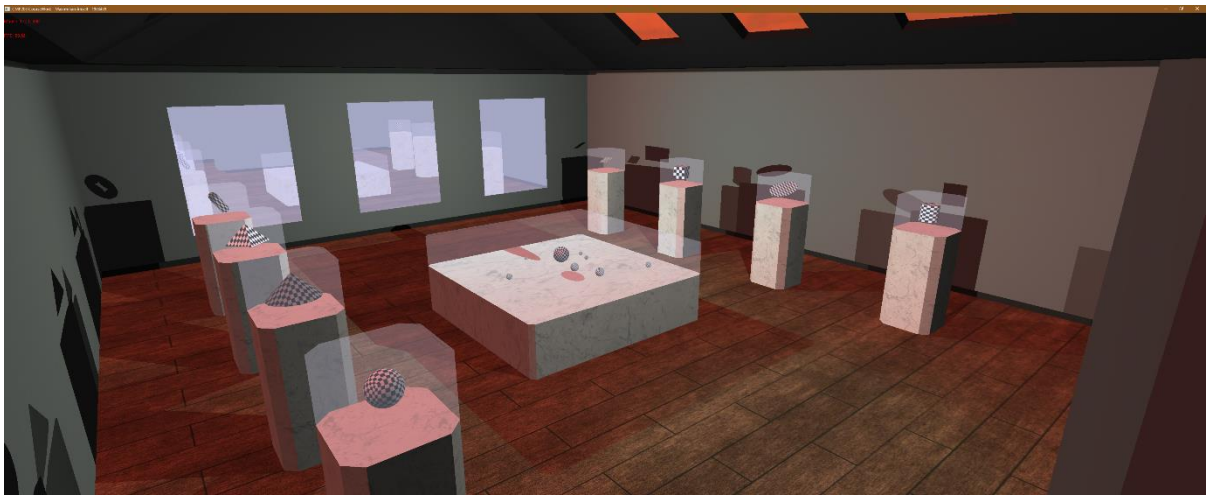
Maximilian (max) Ansell

1903439

Git Username: Wolfy1938

Introduction

Using OpenGL, I've created a depiction of a museum. The layout of the room has been designed specifically to help demonstrate and display my work, a key example of this is the use of display plinths, not only do they fit within the scene thematically but create an excellent medium to represent my primitive generation. With my experience with 3D modelling I could have made a more cluttered scene however I decided to direct my time towards making more complex programmatic qualities within my scene such as shadow volumes. Another example of prioritising the coursework brief over visuals is that each primitive has been left with a default checkerboard texture to help demonstrate the UV unwrapping of the primitives. Each mesh and texture in my scene are of my own making with the exceptions being the sky box texture and the checkerboard texture.



Geometry

There are two main types of geometry within my solution, 'Primitives' that are procedurally generated, and 'Loaded Models' which are obj files. To have both types of geometry handled identically a 'Mesh' base class has 'protected' vertex arrays which both derived classes use.

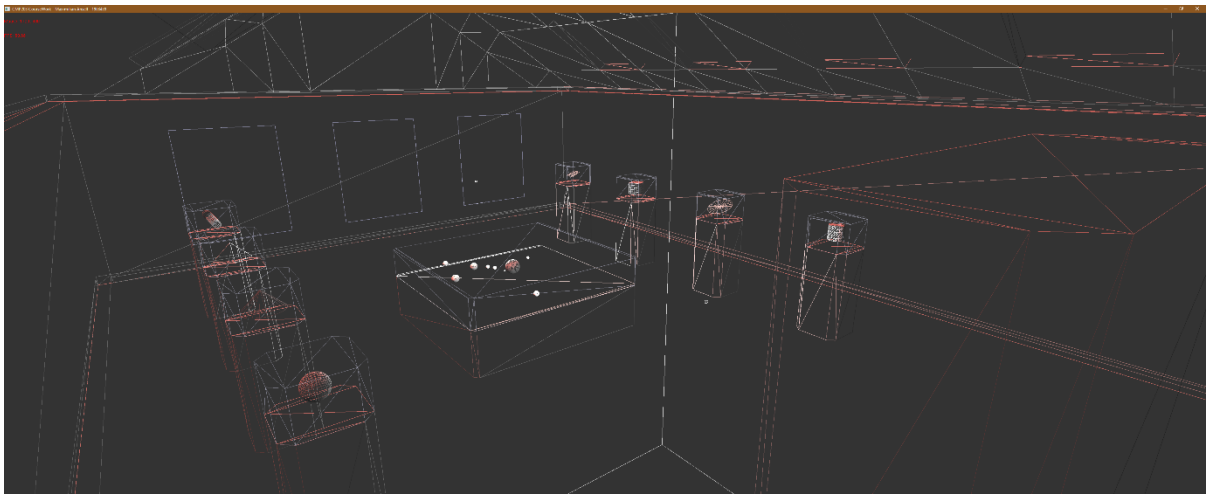
Loaded Models

My method for model loading is essentially built off the lab solution. Reading the file and parsing the data is handled within a while loop that was programmed by Dr Paul Robertson. To convert this data requires looping through each 'Face' that was read and adding it's verts, texture coordinates and normals into their respective vertex arrays. This solution only handles loaded tri models, if I was to take this project further, I would add support for loading quad models as I already support quad primitives.

Primitives

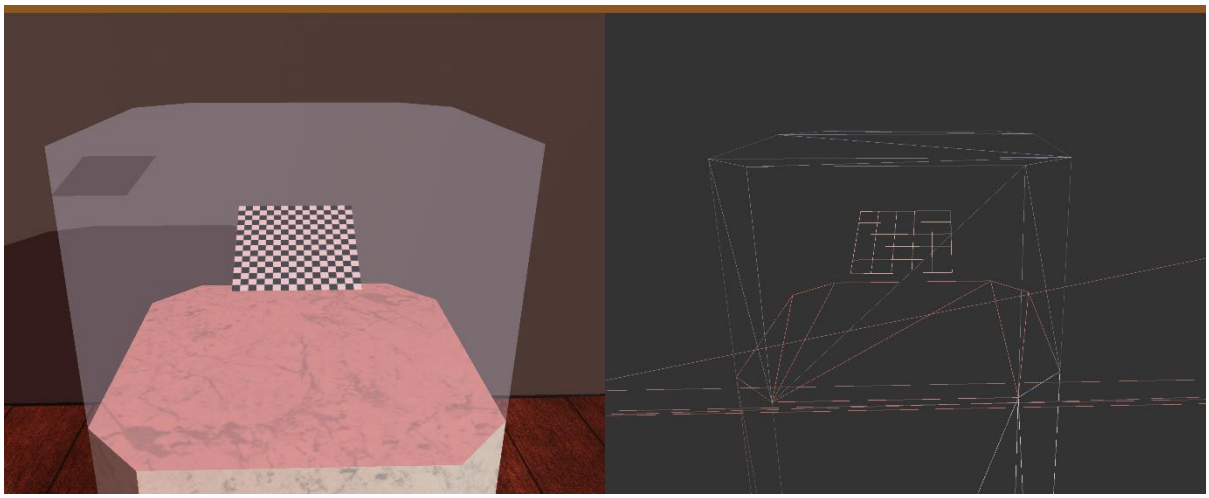
The 'Primitive' class is an empty base class that inherits from 'Mesh'; it contains a sub class for each

of the primitives I've programmed: Plane, Cube, Sphere, Disk, Cylinder, Cone, Pyramid, and Torus. Each 'Primitive' sub class just has a constructor that populates the vertex arrays using the cartesian equations of each shape to calculate the vertices. Each 'Primitive' also takes values for segments or horizontal and/or vertical subdivisions so you can control the resolution of the wire mesh. 2D primitives such as the disk and plane have duplicated back faces with reversed normals so they can cast shadows



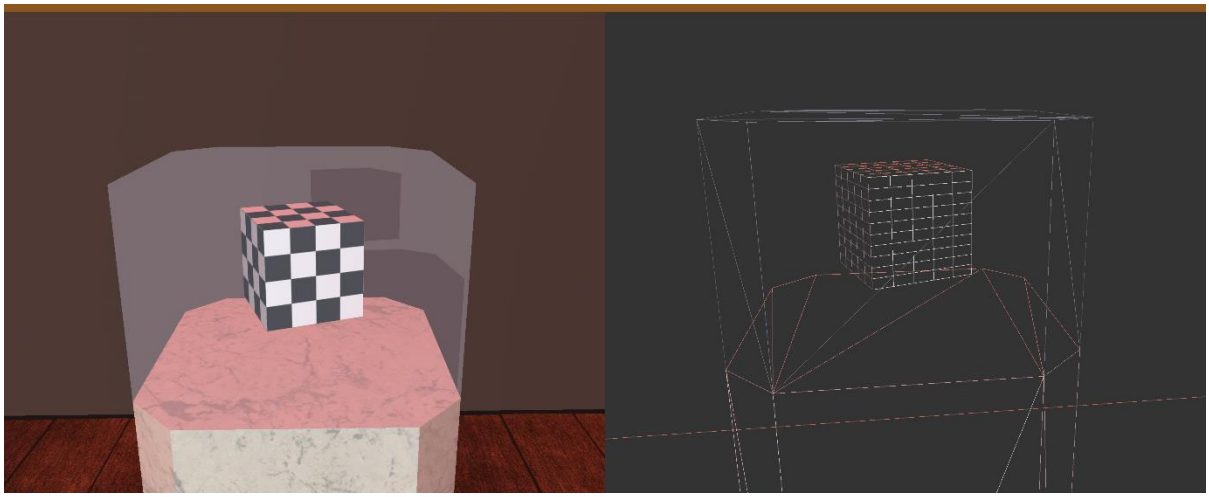
Plane

A normal plane would be trivial to generate; however, my implementation allows for different amounts of subdivisions. I make a unit plane centred around the origin, or wherever the matrix stack happens to be, so each point is offset by -0.5. Rotation is handled by the matrix stack; I can fill out each normal pointing in the positive Y. As it's a unit plane the UV coordinates are the same as the vertex coordinates (without the offset).



Cube

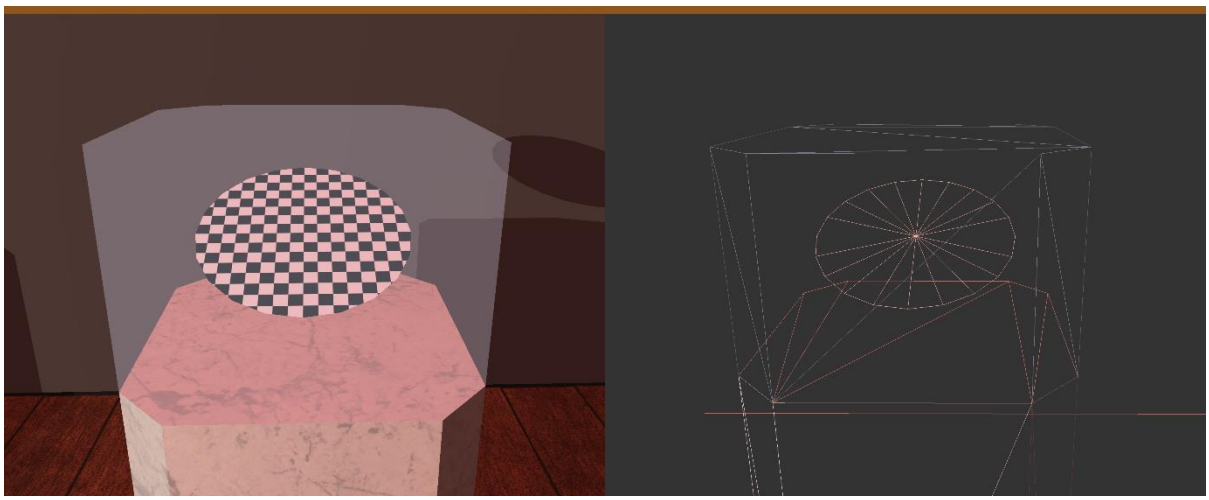
Originally my cube would just have 6 planes as children each representing a side of the cube, unfortunately my final solution doesn't support children casting shadows so the same logic from the plane generation is used 6 times for the different sides with one slight alteration, the top and bottom sides will have their horizontal and vertical subdivision lines swapped so that they line of with the other sides.



Disk

The best way to generate a disk would be to use a triangle fan, my shadow volume calculations don't support triangle fans, only tris and quads so the disk uses tris. The angle between segments, theta, is calculated using 2π over the number of segments. The normals work the same as a plane as it's technically a 2D object whereas the UV coordinates are more complicated, essentially the disk UVs are a circle draw in the centre of the texture file.

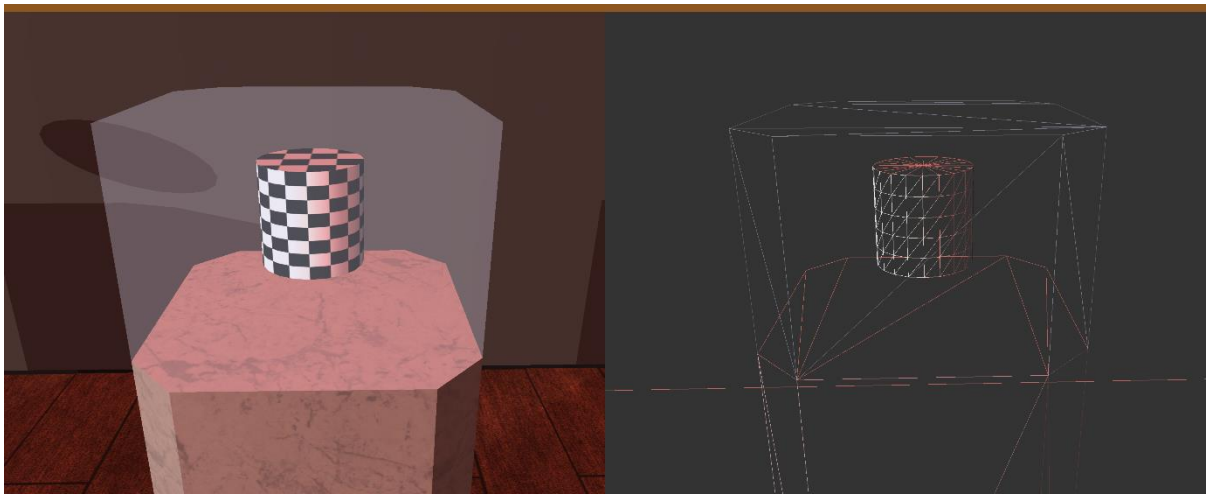
```
theta = 2π / segments;

for each segment
  Vert 1 (0,  0,  0)
  Vert 2 (sin(theta * segmentNumber),  0,  cos(theta * segmentNumber))
  Vert 2 (sin(theta * (segmentNumber + 1)),  0,  cos(theta * (segmentNumber + 1)))

  Normal (0,  1,  0)

  TextureCoord 1 (0.5,  0.5)
  TextureCoord 2 (sin(-theta * segmentNumber) / 2) + 0.5,         cos(-theta * segmentNumber) / 2) + 0.5)
  TextureCoord 3 (sin(-theta * (segmentNumber + 1)) / 2) + 0.5),  cos(-theta * (segmentNumber + 1)) / 2) + 0.5)
```



Cylinder

The logic of generating a disk can be used to calculate a ring of verts on the cylinder, the number of rings and their distance between them need to be added a nested loop. The normals of the verts can be calculated as the position on the cylinder disregarding the y value. To make the cylinder fit onto one texture map the vertical part is cylindrically unwrapped from the positons (0,0.25) to (1,0.75)

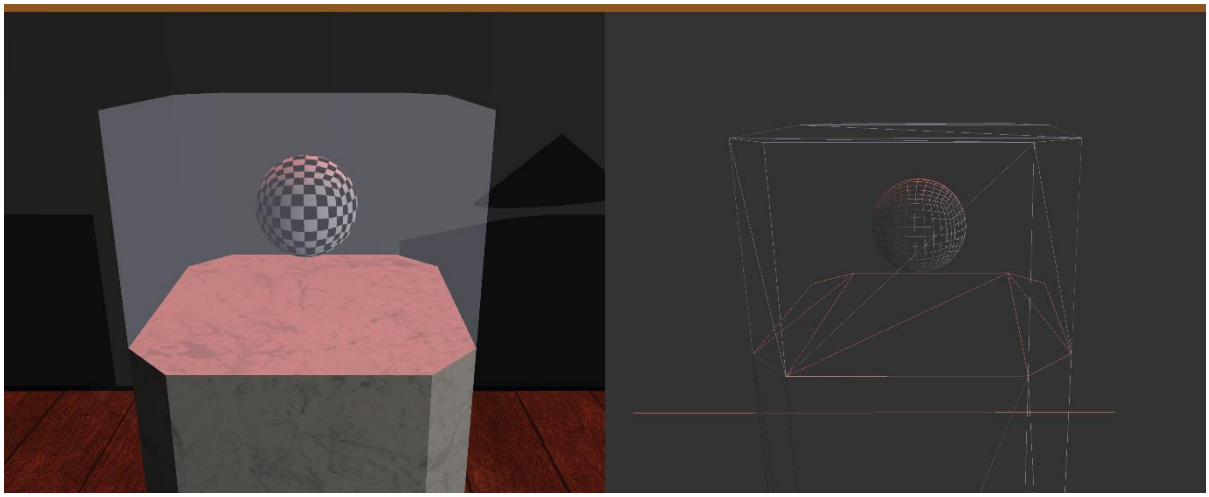
and the two disks are unwrapped as above and below this.



Sphere

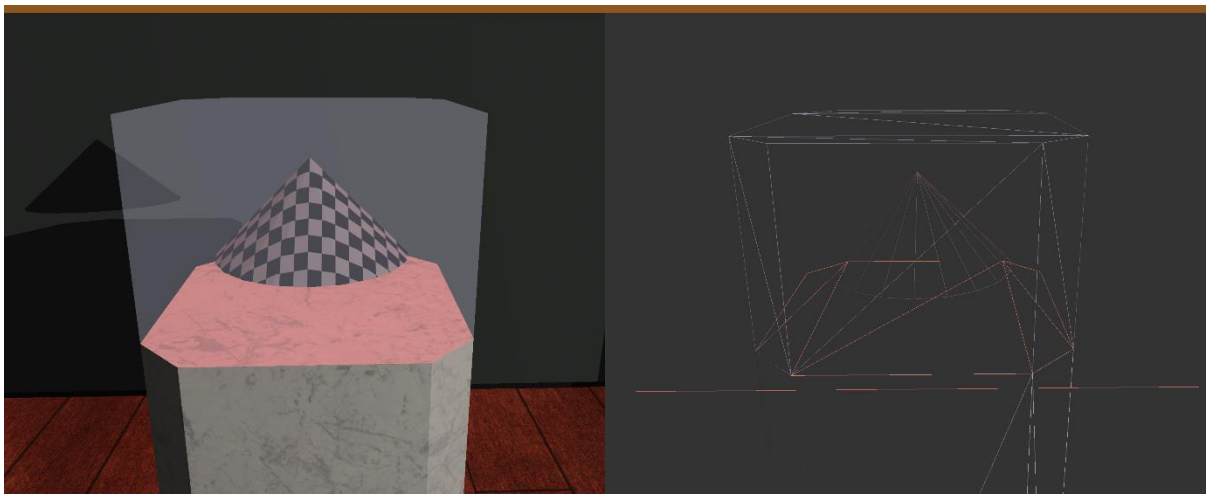
One method of generating a sphere is to use it's cartesian coordinates, however this method will leave pinching at the top and bottom of the sphere that will hinder the shadow volume edge detection as it will interpret those edges as having 3 faces. To combat this a cubic sphere is generated instead, this is achieved by generating and unwrapping a cube as normal but an extra step it to normalise the positions of each vertex so that each vert is now equidistant from the centre, then each vertex normal is equal to it's position. This results in a better distribution of faces, not as accurate, however, as a truncated icosahedron.

```
for (unsigned int vert = 0; vert < verticies.size(); vert += 3)
{
    Vector3 vertex = Vector3(verticies[vert], verticies[vert + 1], verticies[vert + 2]);
    vertex.normalise();

    normals.insert(normals.end(), { vertex.getX(), vertex.getY(), vertex.getZ() });

    // Make sure it's a unit sphere
    vertex = vertex / 2.0f;

    verticies[vert] = vertex.getX();
    verticies[vert + 1] = vertex.getY();
    verticies[vert + 2] = vertex.getZ();
}
```



Cone + Pyramid

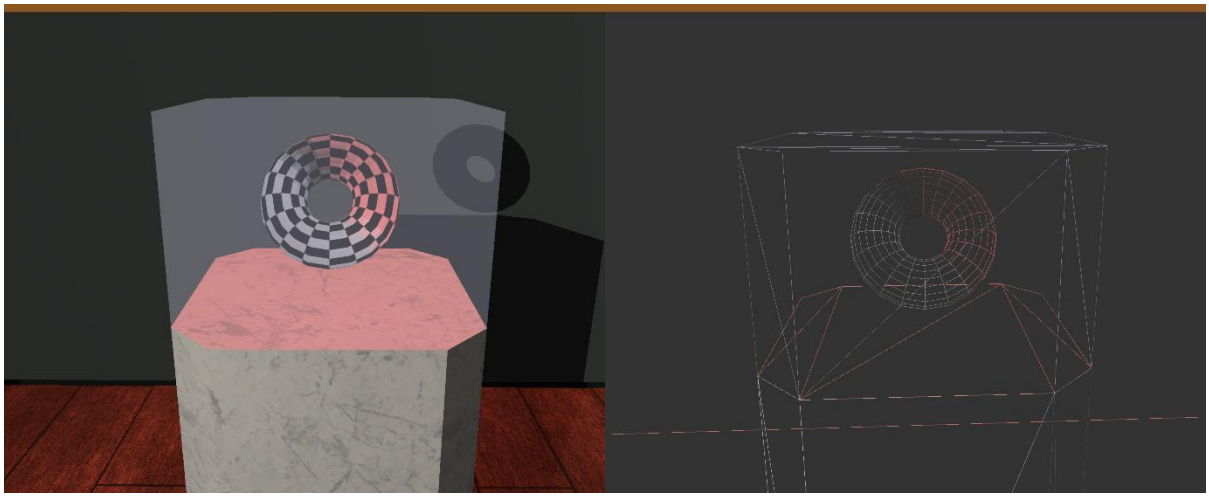
I've combined cone and pyramid into one section because my pyramid is just a cone with four segments. Adapting a disk into a cone is trivial because the only difference is that the centre vertex is offset on the world Y axis.



Torus

Initially generating a torus seemed a daunting task as they seem to be a complex shape, however, they are essentially a ring of disks generated around a disk and connected. The normals of each vertex are equal to their position on the second disk normalised. Finally, the UV unwrapping of a torus can be done cylindrically using the same method as the cylinder.

```
outerRadius = 0.5
innerRadius = 0.25
theta = 2π / vSubdivisions
float delta =  2π / hSubdivisions

for each vSubdivision
    point 1 = (sin(theta * y),       cos(theta * y),       0)
    point 2 = (sin(theta * (y + 1)), cos(theta * (y + 1)), 0)

    for each hSubdivisions
        vert 1 =
            (outerRadius + innerRadius * cos(x * delta)) * cos(theta * y),
            (outerRadius + innerRadius * cos(x * delta)) * sin(theta * y),
            innerRadius * sin(x * delta)

        vert 2 =
            (outerRadius + innerRadius * cos(x * delta)) * cos(theta * (y + 1)),
            (outerRadius + innerRadius * cos(x * delta)) * sin(theta * (y + 1)),
            innerRadius * sin(x * delta)

        vert 3 =
            (outerRadius + innerRadius * cos((x + 1) * delta)) * cos(theta * (y + 1)),
            (outerRadius + innerRadius * cos((x + 1) * delta)) * sin(theta * (y + 1)),
            innerRadius * sin((x + 1) * delta)

        vert 4 =
            (outerRadius + innerRadius * cos((x + 1) * delta)) * cos(theta * y),
            (outerRadius + innerRadius * cos((x + 1) * delta)) * sin(theta * y),
            innerRadius * sin((x + 1) * delta)

        normal 1 = vert 1 - point 1) normalised
        normal 2 = vert 2 - point 1) normalised
        normal 3 = vert 3 - point 2) normalised
        normal 4 = vert 4 - point 2) normalised
```



Transparency

Transparent meshes are stored in their own vector to be rendered last and separate from the normal meshes. This is because they neither cast nor receive shadows and should be excluded from such calculations, they are also rendered last to ensure that all the geometry on the far side of it is fully rendered to the frame buffer before the transparent mesh so that the blend calculation is successful.

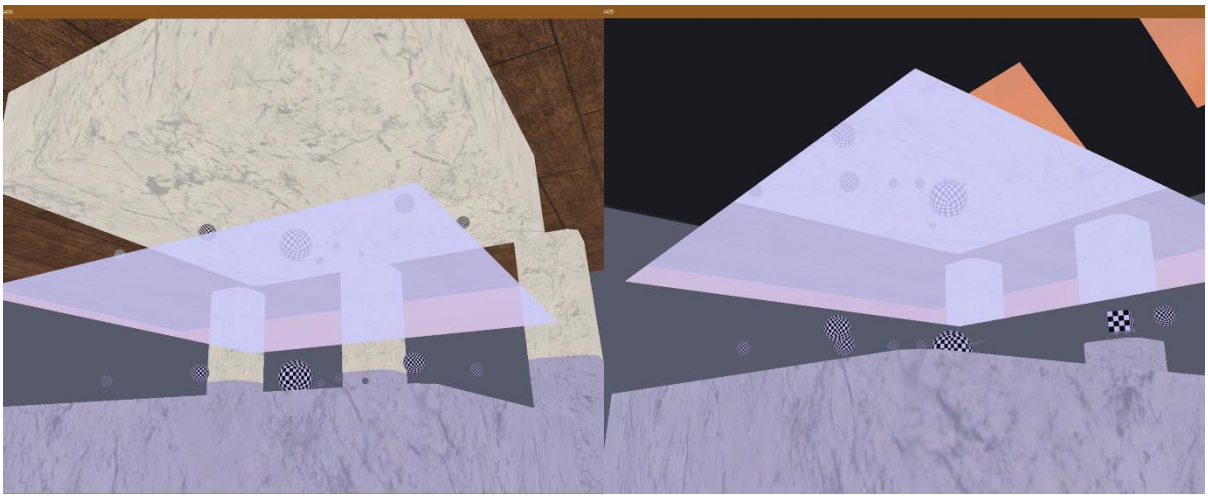
Depth Sorting

A key use of depth sorting within my solution is the skybox. The skybox is a textured unit cube that

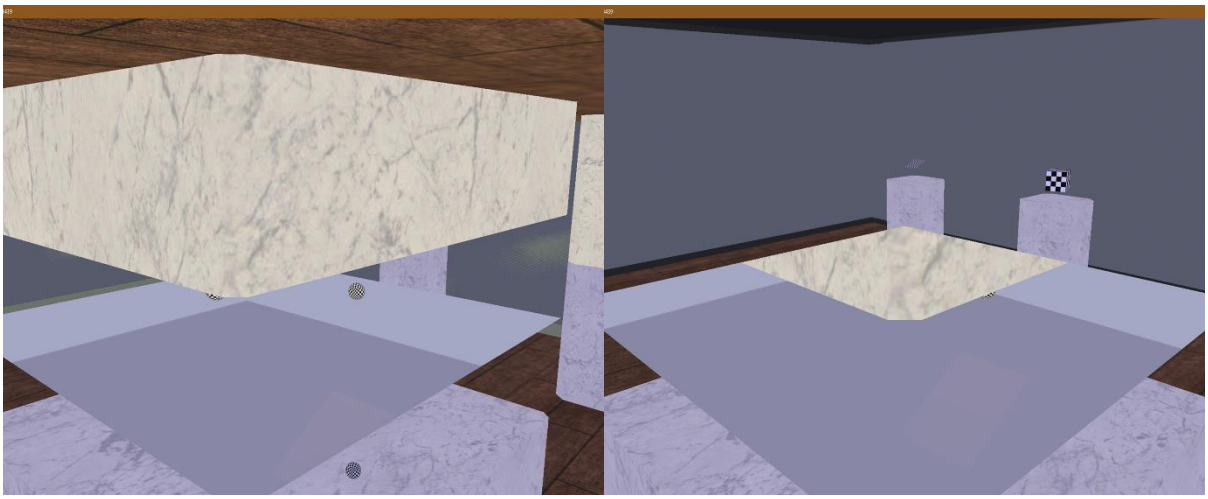
updates its position to be the cameras position each frame. By disabling the depth test before rendering the sky box I can guarantee all geometry will be rendered on top of the skybox despite it being closer to the camera. Front face culling used instead of back face culling as we're technically looking at the back faces from within the cube.

Reflectivity

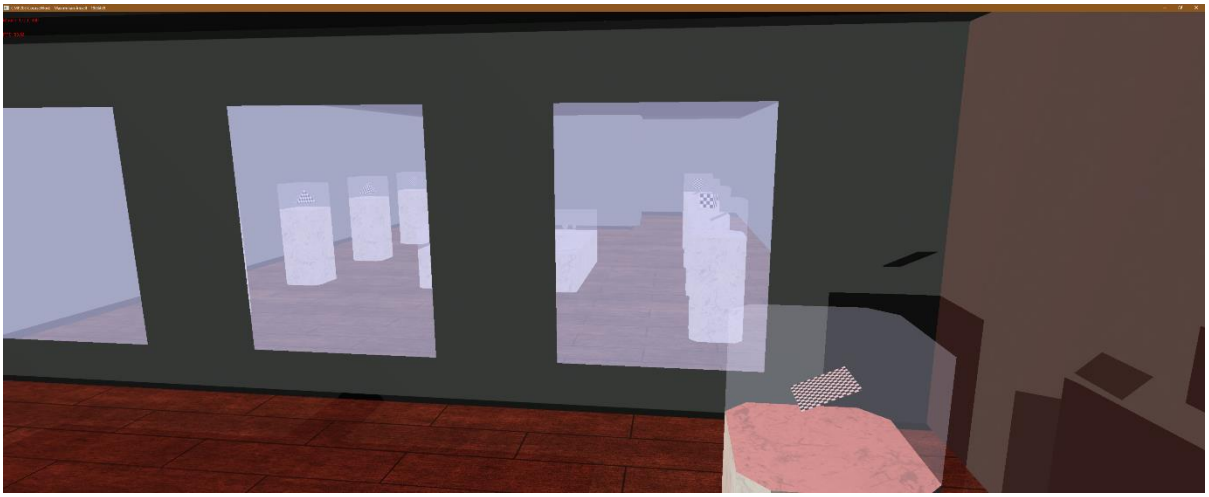
Originally, I had planned to make the transparent surface reflective however my method of reflection did not support this. To make the reflection I render the scene after flipping it in the reflective axis, then use the stencil buffer to cull. This works fine when the reflection occurs on the extremity of the room. If the reflective surface is within the room this causes issues. For example, with a reflection about the XZ plane if the Y axis is 0 then there is no issue as the flipped geometry is on one side and the actual geometry is on the other. When the Y is in the middle of the room then there's flipped geometry on both sides of the reflection plane this flipped geometry will pass the stencil test as it logically should but semantically you wouldn't want it to. An attempted solution was to only flip geometry that's position was on the same side of the reflective surface as the camera but this wouldn't work with objects like the walls that were one object that had bits both above and below the reflective surface.



Reflecting from below



Reflecting from above

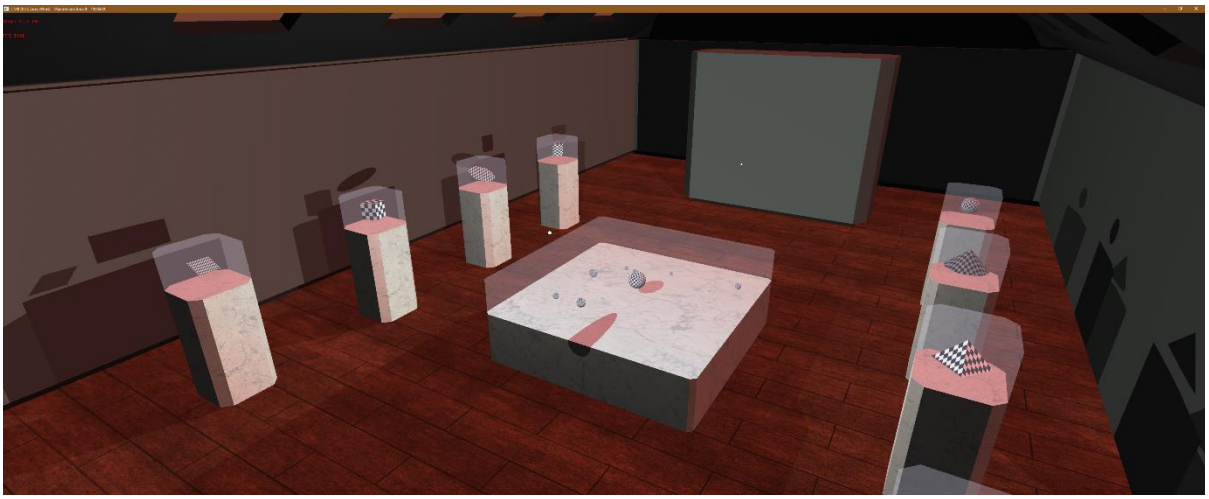


Lighting

My scene has support for three different types of lighting: ‘Directional’, ‘Point’, and ‘Spot’. Each light is handled within the ‘Light’ class that stores the necessary ‘GLfloat’ arrays, using an ‘enum’ to track the light type. Each light has support for attenuation and a pointer to a ‘Primitive’ cube that is placed where the light is for ease of use. Most of the lights use a white light whereas the direction light is a warm orange to reflect the colour of the skybox.

One of the point lights is controllable to the user, it can move in all 3 directions using ‘I’ ’J’ ’K’ ’L’ and ‘U’ ’O’ but its movements are capped within the room. Only one light is not static because the shadow calculations take a lot of compute power. To help combat this the light’s shadows are only calculated when the light is moved.

Point lights are not included in the final solution as they require lots of normals to display smooth attenuation and this would negatively impact the shadow volume calculation as the edge detection calculations are sensitive to the face count of the shadow caster.



Shadows

Shadow generation is where I spent most of my time, and while the current implementation is not perfect, I'm very satisfied with my result. A 'ShadowVolume' class inherits from the base 'Mesh' class, this is so it can be rendered to the buffers. To calculate the shadow volume the light's position and the mesh is needed, this caused an issue as the mesh is stored as vertex arrays, not an easily human read format. Solving this issue involved creating a container that stores the face normal, face position, and the vertices making up the face, this container works with both quads and tris. Whenever the 'Mesh' is translated, rotated, or scaled that transformation needs to be applied to the list of faces, this is achieved by using a custom matrix application method ensuring that the list of faces is a direct representation of the world position of the mesh. Children not casting shadow is because of this current implementation as the child's exact world position world be needed to then calculate the faces for the shadow volume. Given more time I would alter how my solution transforms vertex arrays to support child world position.

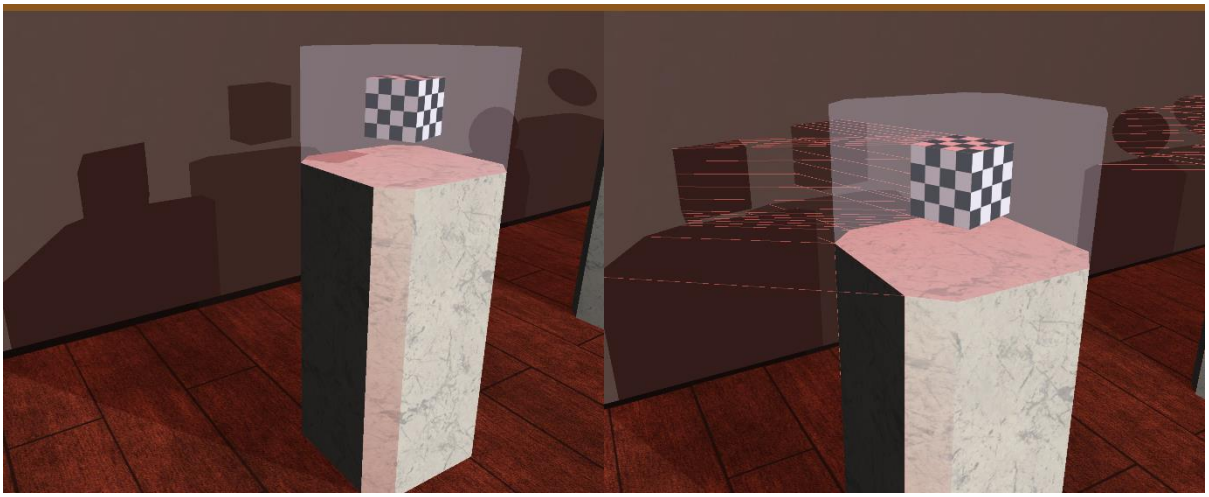
Using the light's position and a representation of the 'Mesh' vertex arrays a shadow volume can now be calculated. The crux of the calculation is getting the contour edges, essentially the edge between the faces pointing towards and the faces pointing away from the light source. This is achieved by looping through each face, calculate the dot product of the light's direction and the face normal to see if face is pointing towards or away from the light source. If the face is pointing away from the light source, then that face is in shadow. Then for each face in shadow get each of it's edges and add them to a data structure, if the edge is already in the data structure then remove it, the result of this is that you get each edge that only appears once. These are the contour edges. This works because each edge has two faces. If the edge is never added it's because both faces are in the light. If the edge is added twice then both faces are in shadow. However, if the edge is only added once then one face is in the light and one of the faces is in shadow, thus depicting it a contour edge.

Once you have the contour edges then you extrude them out into a shadow volume the same way that you would with a simple plane as the shadow caster. One thing to note is that these shadow volumes are not capped at either end the openGL wiki mentions that for a depth fail test the shadow volumes need to be capped so my solution only uses the depth pass method and as such the shadows will not be rendered if the camera is within one of the shadow volumes.

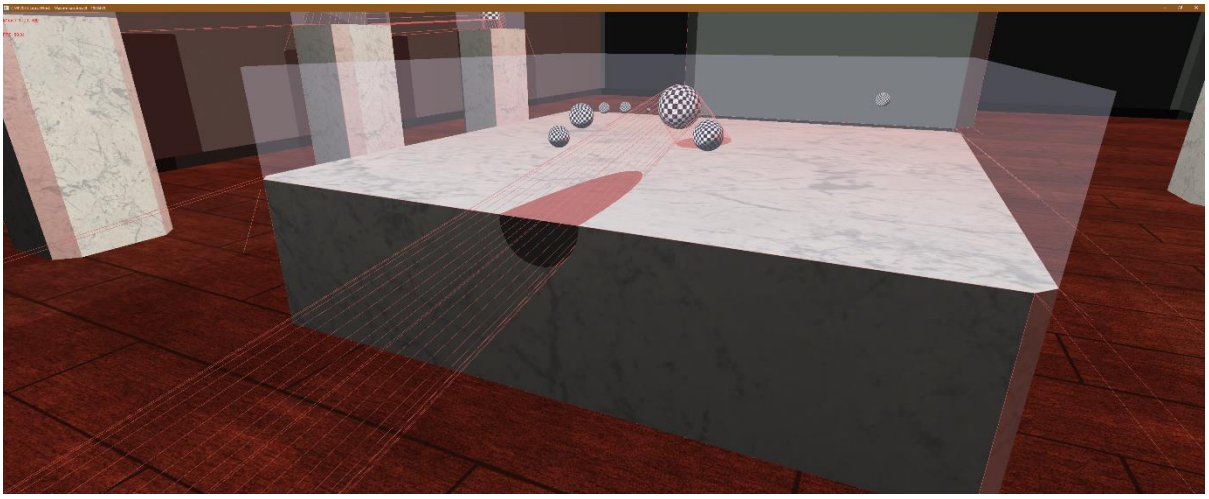
To handle multiple lights that are casting shadows each 'Light' has a list of pointers to the shadow volumes they are casting, for each volume a stencil is drawn using that shadow volume then each mesh is rendered. The 'ShadowVolume' has a pointer to the 'Mesh' that casts it as it that mesh

needs to be excluded from the render when the volume is rendered, this is to prevent a mesh being in its own shadow volume. Each volume needs to rendered separately with its own stencil because if all the volumes are combined on one stencil it will lead to object in an even number of shadows be incremented and decremented so won't actually be rendered in shadow.

As previously mentioned, the solution is not perfect, there's two main flaws with my implementation the shadows are 'stacked' instead of being blended and shadows are cast onto faces with normals that point away from the light source. To solve these, I would probably go down a shader implementation route as opposed to a shadow volume one.



Shadows 'stacking'



Shadows 'spilling over'

Camera

Three different types of cameras have been implemented into the solution: default, orbiting, and procedural. The 'Camera' base class inherits from the 'Transform' class and adds 'lookAt' functionality that converts the cameras 'Vector3's into the openGL 'gluLookAt'. The 'DefaultCamera' class simply overrides the 'handleImput' function and translate the camera based on the user inputs. The 'OrbitingCamera' class is essentially the same but its 'lookAt' function uses a target position instead of the camera forward.

Hierarchical Modelling

To handle hierarchical modelling each 'Mesh' has a 'vector' of pointers to children meshes then each of those children mesh's positions, rotations, and scalars are relative to the parents. Then within each 'Mesh' 'render' function each child's 'render' function is called before the matrix is popped

```
Push Matix

Translate Mesh
Rotate Mesh
Scale Mesh

Render Mesh

for each Child
    Call Child Render

Pop Matrix
```

Conclusion

To conclude I feel that working with OpenGL has really helped build my understanding of rendering engines and pipelines and I'll be able to apply this knowledge in the future to help optimise my other projects. Starting this module, I had no interest in graphical programming however I've developed a great interest in shadows, reflections and shaders and plan to research these areas in more detail. I feel that over the course of this module I've improved my c++ memory usage, however even writing this report there are still some things I would want to change, such as using a set or a map in the edge detection as opposed to a vector with my own custom 'contains' function.

If I were to complete this task again, I would use my new knowledge and create a more graphically complex scene with the inclusion of OpenGL shaders whereas for the main structure of my code I wouldn't change much. I'm proud of my current solution that I modelled around the unity system, I found it very seamless to use and to expand on it.

References

"Mr.Who" (2017). "Pack skybox texture #1" [online]
Available at: https://gamebanana.com/textures/5227 [Accessed 14 December 2020].

"John Tsiombikas" (Unknown). "Volume Shadows Tutorial" [online]
Available at: http://nuclear.mutantstargoat.com/articles/volume_shadows_tutorial_nuclear.pdf
Accessed 08 December 2020].

OpenGL (unkown) 'Multiple Light Sources'
Available at:
https://www.opengl.org/archives/resources/code/samples/advanced/advanced97/notes/node103.html
[Accessed 18 December 2020].