CMP301

Application Report and Project Summary

Janek Uchman

Scene Overview

Scene Objects

The scene at its base contains three objects, a planemesh that serves as the floor, a plasma sphere floating in the centre of the room, with a particle effect, and a textured box to help demonstrate shadows and lighting. In the scene, but not visible, are three lights, two directional and one point light based inside the plasma sphere.

Figure 1: Scene Overview

The sphere is a tessellated Icosahedron, that has also been displacement mapped and textured. This demonstrates the vertex manipulation aspect of the brief, including properly calculated normals (although it is not lit, the normals are used for displacement), and texture coordinates. It is also an example of non-trivial tessellation, as it is based on projecting the newly tessellated polygons into world space, along with calculating new normals and texture coordinates.

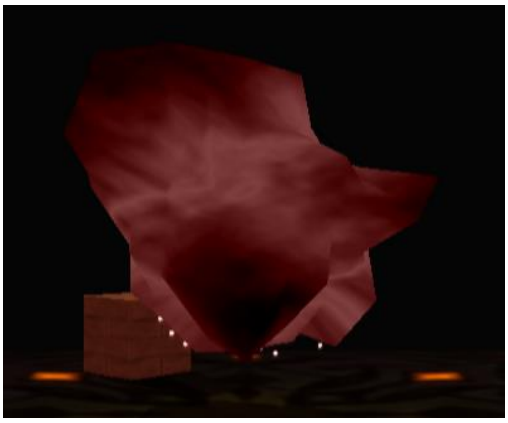

Figure 2: Displacement map increased sphere with low tessellation

Figure 3: Displacement map increased sphere with high tessellation

Shadows can be seen being cast from both the box and the sphere; the sphere shadows match the displacement map being modified in real time. The directional lights can be seen lighting the box from separate angles, as the lights are facing towards the scene at different angles. There is a pulsing point light with attenuation coming from the sphere, this changes its colour to reflect that of the sphere.

Figure 4: An example of shadows being cast

Below the sphere, making use of the geometry shader, are particles. These move in real time and also change colour to reflect that of the sphere. The direction and quantity of these can be changed.

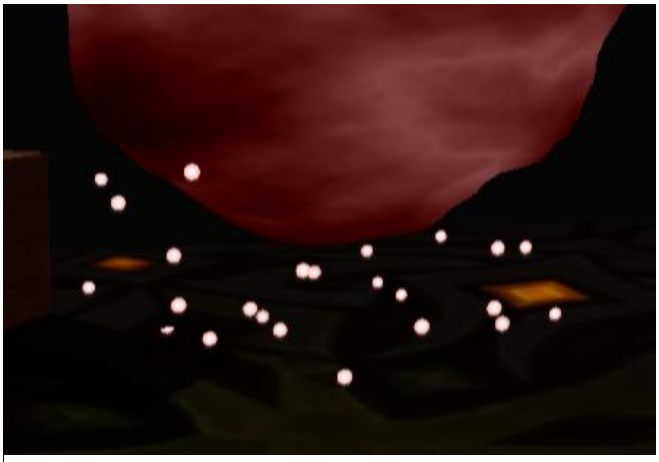

Figure 5: Particle effects

Also demonstrated in the program is depth of field, or “bokeh”, as a post processing effect. This uses a box blur to make images that are close to the camera seem clear, while those that are further away are blurry.

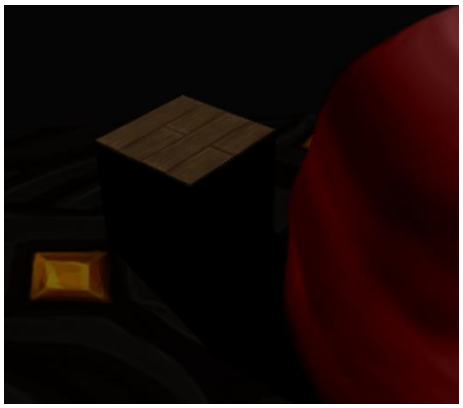

Figure 6: Depth of Field with the box in the centre of the camera

Figure 7: Depth of Field with the sphere in the centre of the camera

Additional Controls:

- **Plasma flow rate:** The speed and direction that the displacement map and texture moves
- **Invert plasma colour:** Makes a negative of the current colour
- **Plasma colour:** Allows changing the plasma ball colour, point light colour, and particle colour
- **Sphere displacement:** How far along the normal the displacement map will affect the sphere
- **Sphere tessellation:** The amount of edges and insides to be tessellated into the sphere
- **Render box:** Toggles the box on or off
- **Particle directional speed:** The speed and direction of the particles
- **Amount of particles:** The amount of particles rendered

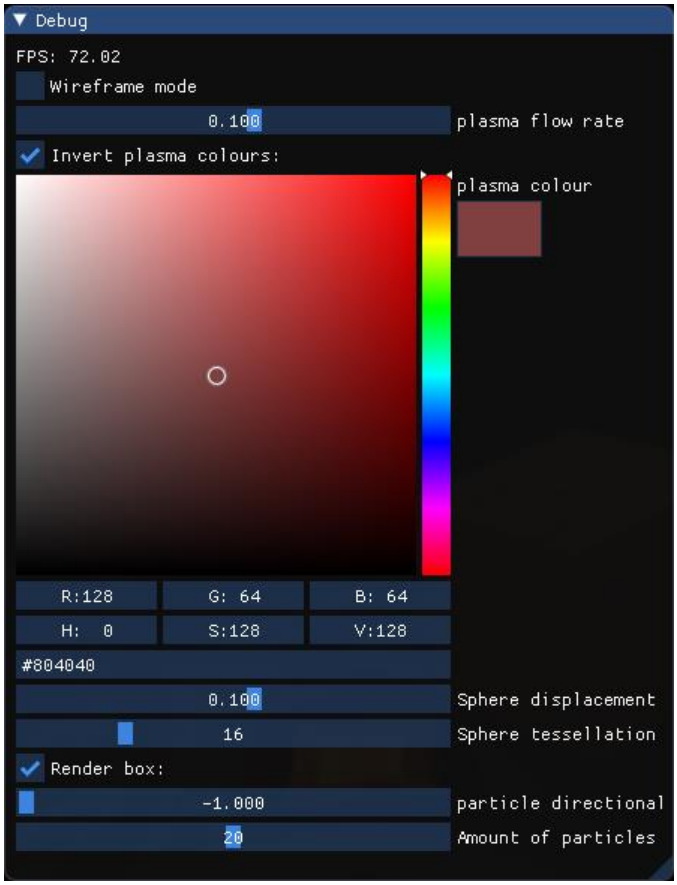

Figure 8: ImGui settings

Algorithms and Data Structures

Plasma Sphere

The plasma sphere is generated first of all through a set of indices (Luna, 2016) that would create a icosahedron, then they could be subdivided to create an icosasphere. Instead these vertices and indices were left without projection to be passed into the tessellation shader. Once tessellated the new vertices (and the original set) are projected onto a sphere by converting Euclidean coordinates to spherical coordinates, we can work this out by using the normal of the calculated vertex, and multiplying it by pi.

The texture coordinates are then wrapped around the sphere (Game Dev, 2015), and the values received from the displacement map are projected along the newly calculated normals, along with a change in time and displacement modifier, to give the plasma sphere a flowing texture. For the pixel shader the grey scale value is taken, then passed in colour values are applied in either a negative or positive depending on the colour inversion passed in as a float.

```
float radius = 1;

// Determine the position of the new vertex by interpolating from the parent coordinates
vertexPosition = uvwCoord.x * patch[0].position + uvwCoord.y * patch[1].position + uvwCoord.z * patch[2].position;
// Project onto unit sphere.
float3 n =normalize(vertexPosition);
// Project onto sphere.
output.position.xyz = radius * n;
output.normal = n;

//Wrap the texture around the sphere
output.tex.x = asin(output.normal.x) / (2.0 * 3.14159265359f) + 0.5 ;
output.tex.y = asin(output.normal.y) / 3.14159265359f + 0.5 + time;

//Calculate how much we shoud displace based on the darkness of the displacement map
float4 textureColour = texture0.SampleLevel(sampler0, output.tex, 0);
float averageDisplacement = (textureColour.x + textureColour.y + textureColour.z) * displacement;

//Move the vertex along the normal
output.position.xyz += averageDisplacement * output.normal;

output.position.w = 1;
// Calculate the position of the new vertex against the world, view, and projection matrices.
output.position = mul(float4(output.position), worldMatrix);
output.normal = mul(float4(output.position.xyz, 0.0f), worldMatrix);
output.position = mul(output.position, viewMatrix);
output.position = mul(output.position, projectionMatrix);

//Store the depth position for the shadow shader
output.depthPosition.xyz = output.position;

return output;
```

Figure 9: Domain shader code for vertex projection, texture wrapping, normal calculation, and displacement

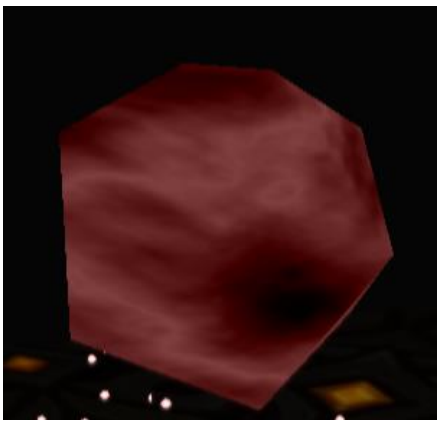

Figure 10: Icosahedron with no tessellation (displacement map set to 0)

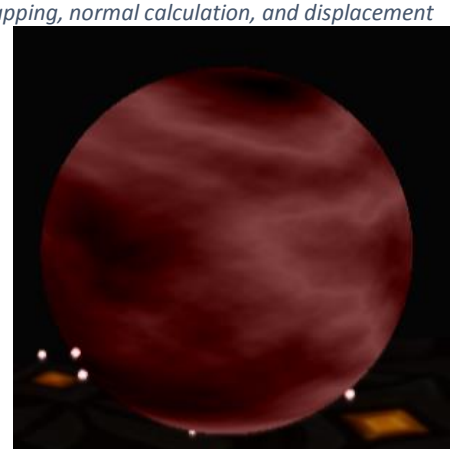

Figure 11: Icosahedron with 64 inside/edge tessellation (displacement map set to 0)

Lighting and Shadows

There are three lights in the scene, the first two are directional and work out how much a face should be light based on the angle from the pixel to the camera, the last light is a point light and calculates attenuation based on distance from the light.

The first two lights are set to cast shadows, however only one of them does (see Critical Reflection), they do this by gathering depth data from the scene in two separate passes. One is calculated for each light, using the orthographic matrix of the light acting as if the light's a camera. When these render textures are passed into the pixel shader their depth is found and if that area should be in shadow then it won't be lit by that light. The colour still takes into consideration the other two lights. Shadows are also manually darkened further to create a more imposing contrast.

The shadow for the plasma sphere is calculated using a different depth map shader. This one takes into consideration the displacement of a regular, untessellated sphere, then projects that onto the depth map instead (see Figure 9). This doesn't give a true one-to-one recreation of the vertices from the sphere as it won't match tessellation, but it appears close enough and computationally is much cheaper.

Particles

Particles are created using the geometry shader. They're a simple point mesh that is then rendered as a "sprite". The particles billboard towards the camera to appear as if they're real 3D objects (BraynzarSoft, 2015). This is done by calculating the plane normal using the camera's position passed in from the shader and the point's position. Using this we calculate the vector right by normalizing the cross product of the plane normal (the forward) and the up vector, in this case (0,1,0).

```
float2 g_tex[4] =
{
    float2(0, 0),
    float2(0, 1),
    float2(1, 0),
    float2(1, 1)
};

float3 planeNormal = input[0].position - cameraPos;
planeNormal = normalize(planeNormal);

float3 upVector = float3(0.0f, 1.0f, 0.0f);
float3 rightVector = normalize(cross(planeNormal, upVector));

rightVector = rightVector * 0.5f;
upVector = float3(0, -1.f, 0);

float3 positions[4] =
{
    input[0].position - rightVector,
    input[0].position + rightVector,
    input[0].position - rightVector + upVector,
    input[0].position + rightVector + upVector
};

float4 g_positions[4] =
{
    float4(positions[0].x, positions[0].y, positions[0].z, input[0].position.w),
    float4(positions[1].x, positions[1].y, positions[1].z, input[0].position.w),
    float4(positions[2].x, positions[2].y, positions[2].z, input[0].position.w),
    float4(positions[3].x, positions[3].y, positions[3].z, input[0].position.w)
};


for (int i = 0; i < 4; i++)
{
    SetPoint(input, triStream, output, g_positions[i], g_tex[i], -planeNormal);
}
```

Figure 12: Code snippet demonstrating how the vectors are calculated and used to rotate the particle towards the camera

Depth of Field

Depth of Field is created using box blur, and an additional depth texture for the scene. Box blur is created by taking samples of texels in the X and Y of the scene texture. The sampled texels are then added into the current pixel's colour with certain weights applied to allow for greater or lesser blur.

Using the depth shader created for the lights, a depth texture of the scene as seen from the camera can be taken. This, along with the final blurred render of the scene, are passed into the depth of field shader. The depth of field pixel shader looks at the centre of the screen and takes the depth of this point. Then the image is sampled between the scene's render and the blur texture, based on the depth distance to this point, along with some weighting to reduce/increase the blur difference in the focus and out of focus sections.

```
float4 main(InputType input) : SV_TARGET
{
    float weight = 0.5f;
    float4 sceneSample = sceneTexture.Sample(Sampler0, input.tex);
    float4 blurSample = blurTexture.Sample(Sampler0, input.tex);
    // Distance to the center of the screen
    float centerDist = sceneTexture.Sample(Sampler0, float2(0.5f, 0.5f)).w;

    float depthDifference = abs(centerDist - sceneSample.w) * weight;

    //Get a value based on the difference
    return saturate(lerp(sceneSample, blurSample, depthDifference));

}
```

Figure 13: Depth shader focus calculations

Critical Reflection

Shadows

After much troubleshooting I failed to get two shadows to render at the same time. Initially, after consultation in the labs, I thought this was simply a problem with passing arrays into the constant buffers in the shadow shaders. After taking these out it still did not work. Setting the lights to be the exact same in every section of the program still wouldn't allow for the second shadow to be rendered. Swapping around the order seemed to have no impact at all on this either. After further help in the labs we found the view projection matrix of the second light to be the problem, as keeping everything the same but using the view projection matrix of the first light allowed for two projected shadows (however they would not be aligned properly with where the light should be). After this discovery I looked through all stages where this could go wrong, removed more arrays just to be sure, checked the values were being input correctly into the shaders, etc. but still could not get the second shadow to render. I'm not sure where exactly this went wrong.

Particles

I feel that the way I render particles could be improved on. Currently I'm rendering single point meshes and adjusting their world matrix. It may be computationally better to work this out within the shader pipeline.

Overall Program

I believe I could have made better use of inheritance and extension to create a cleaner base program, instead of storing closely coupled values in the main App1 file.

My debugging process could be improved on, for example when I came across a problem with my icosahedron I believed the problem to be with the vertices and indices I was passing in. I assumed because my tessellation shader had no problem rendering a polygon that it was not an issue. It was only after transferring the icosahedron over and rendering it normally I realised the points were correct. Double checking all stages of the pipeline could have saved me a lot of time.

Learning and making use of a frame capture debugger could have helped me find out where exactly in the graphics pipeline my program was going wrong earlier, and speed up reaching the solution.

References

BraynzarSoft. (2015). Retrieved from https://www.braynzarsoft.net/viewtutorial/q16390-36-billboarding-geometry-shader

Game Dev, S. E. (2015). *GameDev Stack Exchange*. Retrieved from https://gamedev.stackexchange.com/questions/98068/how-do-i-wrap-an-image-around-a-sphere

Luna, F. D. (2016). *Introduction to 3D Game Programming with DirectX12.*

Resources

http://www.coolgrannyflats.com/mountain-heightmap-generator/mountain-heightmap-generator-getting-height-maps-from-qgis/ - Height map

https://schneide.blog/2016/07/15/generating-an-icosphere-in-c/

http://pngimg.com/download/14422 - Particle sprite

https://www.artstation.com/artwork/lr01J - Floor + Box

Additional Screenshots