CMP301

Graphics Programming with Shaders

1600490 Rowan Stockton

Contents

Overview

The scene consists of a central sphere in with a particle system surrounding it, with a controllable wave system that can be sent through to the sphere. In front of the sphere is a plane where a post processing technique (kuwahara filter) can be demonstrated on a texture. The post processing technique can also be applied to the whole scene and the effect can be controlled in the GUI as well. To the left the sphere is another plane demonstrating dynamic tessellation on a manipulated height map. The tessellation and manipulation values of this plane can also be controlled with in the GUI. Two point lights that can be toggled into spot lights are rotating around the middle casting a shadow onto a flat plane below.

UI

The UI uses collapsible headers each handling the variables that affect different parts of the scene. In Enable there are check boxes that let you enable and disable objects around the scene. In Wave options you can edit values, such as: the wave like wave origin, speed, frequency and amplitude. For the Tessellation options you can change the maximum and minimum tessellation values and the maximum distance tessellation will occur at. You can also affect the wave being passed through the tessellated mesh, in the same way as the previous wave options. Light options allows you to change the colour, attenuation values, and whether the light is a spotlight or not and the spotlights exponent. The Post processing and Kuwahara options are identical, allowing you to change the radius of the subregions of the filter samples.

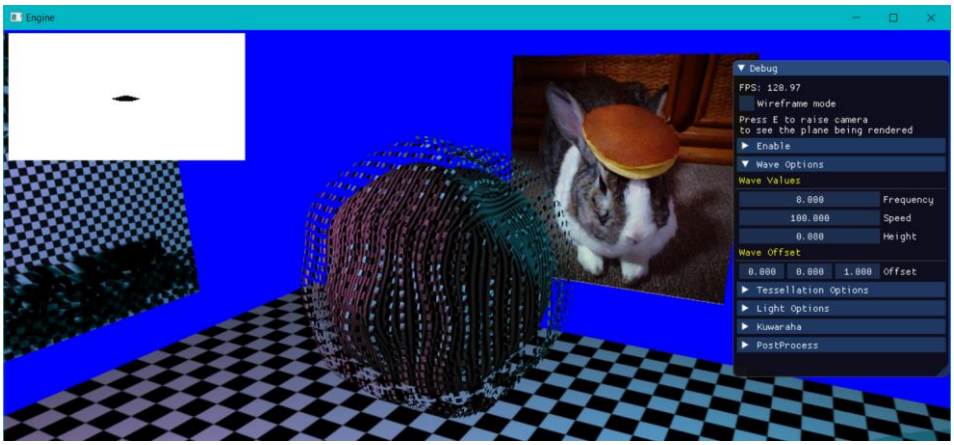
Algorithms and Data Structures

Vertex Manipulation

A wave ripple effect is applied to a sphere in the scene using a vertex shader. To create the ripple effect on the sphere, a 2D ripple effect equation from Josh Marinacci on his blog (marinacci, 2018) was adapted to be applied to a 3D surface. This worked through taking the equation to calculate the Y displacement from the X and Z positions and applying it to the displacement of every axis in respect of the two other components' positions. To ensure the right displacements occurred in each axis, their displacements were multiplied by their corresponding normal components giving a clean ripple effect through a sphere. The wave's height, amplitude, frequency, speed and the ripples origin can be changed in real time. The same vertex manipulation is applied to the particles.

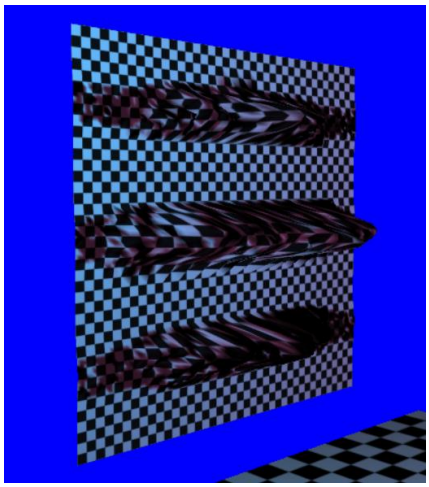
```
float4 shiftPosition(float3 position, float4 displace ,float3 normal)
{
    float dx = position.x + displace.x;
    float dy = position.y + displace.y;
    float dz = position.z + displace.z;

    //shift in Z
    float dx2 = dx * dx, dy2 = dy * dy, dz2 = dz * dz;
    float angleZ = sqrt(dx2 + dy2);
    angleZ = -time * speed + angleZ * frequency;
    position.z = sin(angleZ) * height * normal.z;

    //shift in y
    float angleY = sqrt(dx2 + dz2);
    angleY = -time * speed + angleY * frequency;
    position.y = sin(angleY) * height * normal.y;

    //shift in x
    float angleX = sqrt(dz2 + dy2);
    angleX = -time * speed + angleX * frequency;
    position.x = sin(angleX) * height * normal.x;



    return float4(position.x, position.y, position.z, 1.0);
}
```

Code snippet of the vertex 3D wave ripple



Wave manipulation being applied to sphere and particles

For the tessellated plane, a height map is passed through to the domain shader and using the value of the texture is what gives the height displacement. The height map's displacement is also dynamically changed by using a sine wave to generate a value with which to multiply the height map's displacement by. If the value is less than zero, a value of zero is returned instead, thus applying no displacement at that vertex and creating a flat plane.



Tessellated displaced terrain

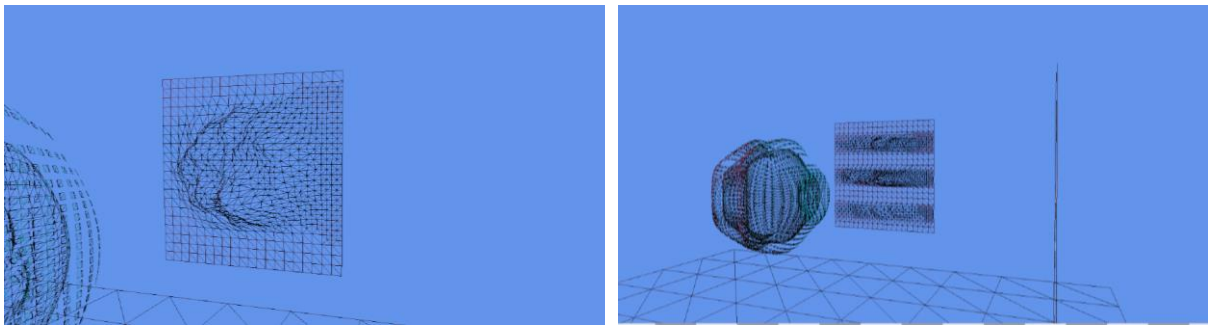
Like the sphere's wave all the same parameters can be changed apart from wave's direction/ origin. To pass these wave values to the different shader stages a standardised struct was created to allow ease of passing to different shaders that may use the same values to pass to a buffer.

Tessellation

Tessellation is applied to a custom control patch plane to which a height map is being applied. This was to be accomplished by dynamically tessellating the mesh, depending on the distance from the camera. Because we created the mesh, we know the order in which the vertices are to be read, so the midpoint of the edge of each quad can be calculated by interpolating between the shared vertices of that edge. From that midpoint the distance from the camera to that edge can be calculated and tessellated accordingly depending on that value. For the middle tessellation factor, the mid point of two of the opposite edge mid points is calculated. Since the edge is shared between quads the tessellation for that edge is consistent between the two quads causing no popping. The flat plane is now being dynamically tessellated by distance, but the plane is to be moved, scaled and the vertices to be displaced. The distance to each vertex needs to be accurate for the dynamic tessellation to appear effective. To fix this, the same displacement which is being applied in the domain shader needs to be applied to the interpolated midpoint to give an accurate distance. While applying this change the point is also multiplied by the world matrix. This allows for matrix manipulations to be applied and for the distance to remain accurate. Finally, if no displacement is being applied it returns a tessellation factor of one for that edge as no detail is required as the terrain is flat.

```
int distanceMultiplier(float3 camPos, float3 vertPos, float2 texPos)
{
    //Sample height map
    float displacementVal = texture0.SampleLevel(Sampler0, texPos, 0);
    //Multiple height value by wave value
    displacementVal *= max(0, height * sin((vertPos.x + (speed*time))*frequency));
    vertPos.y += displacementVal;
    //transform to world space
    vertPos = mul(float4(vertPos, 1.0f), worldMatrix).xyz;
    //Find the distance between camera and the current position
    int dist = distance(camPos,vertPos);
    //Calculate the tesselation value depending on the distance
    dist = distFactor - dist;
    dist /= distFactor/maxTessValue;
    //make sure the tesselation is between the max and min
    int tessellationFactor = max(minTessValue, dist);
    tessellationFactor = min(maxTessValue, tessellationFactor);
    //If no tesselation don't do any displacement other wise do
    displacementVal = ceil(saturate(displacementVal));
    tessellationFactor = tessellationFactor * displacementVal + (1 - displacementVal);

    return tessellationFactor;
}
```

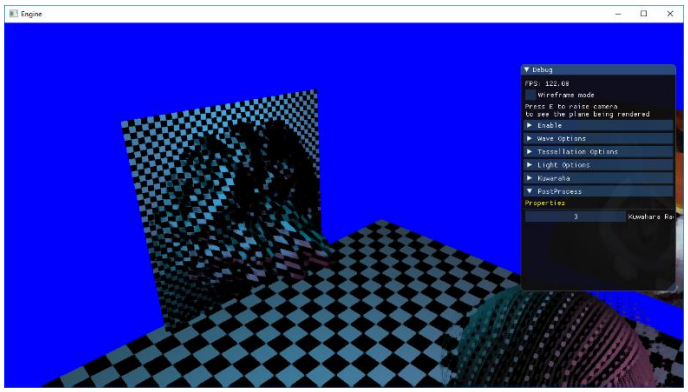
Dynamic Tessellation



Dynamic tessellation being applied based on distance and current height displacement

For the normals a Sobel operator (Jason Zink, 2011, p. 430) is used for quick efficient normals based off the height map. Instead of calculating vectors between surrounding vertices and using the cross product to calculate the normal, the Sobel operator uses the surrounding pixels and calculates the normal based on the displacement of each of the surrounding pixels. Using this method lets normals

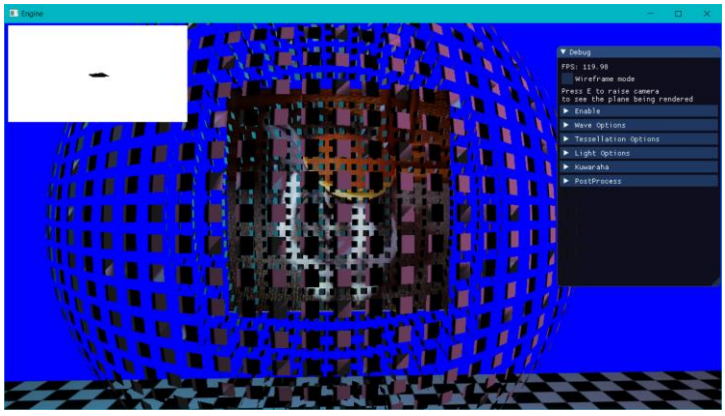
be created without needing to know anything about the other vertices. The only problem lies with the fact that the height is being dynamically offset so when flat the normals are still being calculated for the height map even though the mesh is now flat. To account for this if the offset value is zero or less a default normal / unit vector in the y axis is given, since the plane is generated in the z and the x axes, is the normals would be up. Although not fully accurate normals, they are sufficiently close and efficient enough that the trade-off is acceptable.



Tessellated height map with normals being calculated through the Sobel operator and Kuwahara filter being applied

Geometry Shader

The geometry shader is used for two things. The first is to calculate normals on the manipulated sphere. The method that is used takes vectors between all the vertices to one another and uses the cross product to generate a normal that should be roughly correct. The reason for doing this in the geometry shader is because all the vertices for that triangle is read in and already manipulated, so there'd be no need to try calculating the predicted position of an adjacent vertex. As for the particles a similar process is done except for each vertex a quad is drawn instead to creating a sphere of quads. The quads generated also face outwards in the direction of the normals this is achieved through finding the angle between its forward vector and the x component of the normal it's being rotated to. The same is done with the y component, then a rotation matrix for rotation around the x and y axis is generated and applied to the points of the quad to get them to face outwards along that normal. A offset needed to be applied to the rotation around the x axis however since the back face would be rotated by 180° in the x axis and the y the overall rotation would cancel out. For any y angle that is less that -90° or greater than 90° the y angle is shifted to not cancel out the rotation with the x axis.



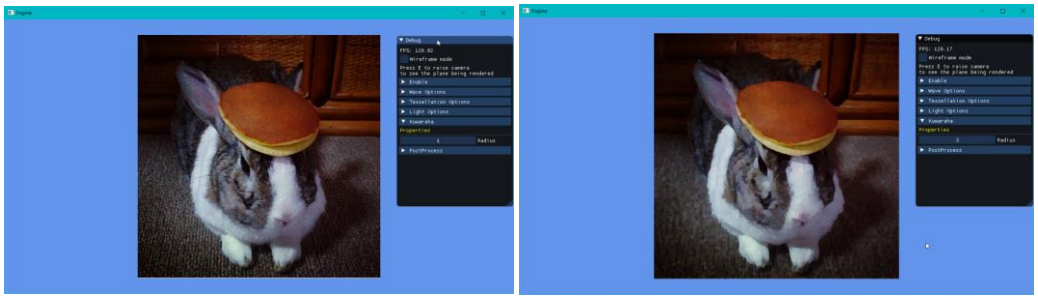
Particles rotation counter cancelling out on the back side so the particles end up rotating the same way

Pixel Shader

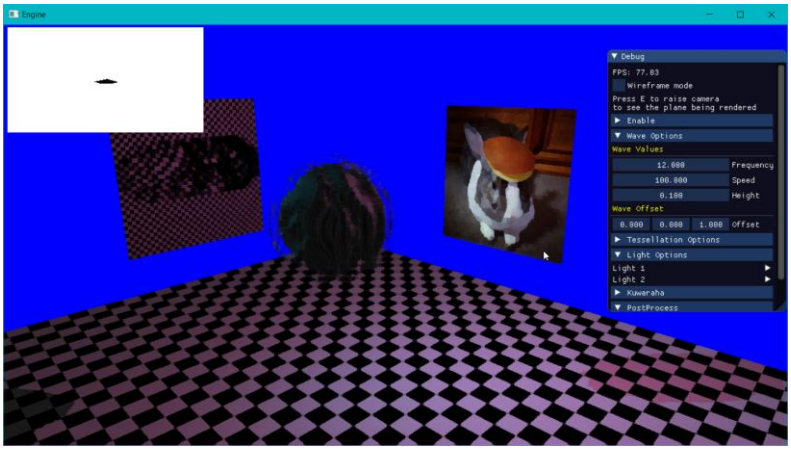
The scene uses two point lights rotating around the centre sphere to light the scene. The light information is stored in a custom struct to be easily passed to the shaders just like the wave information for the manipulation. The information is then passed to a buffer where in the w component of the position and direction the light type and exponent is packed in as to simplify the padding of the buffer. Additionally, a separate buffer is used for attenuation values. In the application one main pixel shader is used for lighting. This pixel shader deals with two lights being a point or a spotlight. To switch between a point light and a spot light the spot factor is multiplied by a value or either one or zero depending on the type and then adding one minus the type so as not to multiply the lighting by zero. The spot factor is calculated using the equation found in 3d graphics programming for beginners. (Lengyel, 2012, p. 160)

For the post processing an oil painting effect called the Kuwahara filter was implemented. It's an edge preserving filter used to remove unnecessary detail in high contrast areas but keeps shapes boundaries even in low contrast areas. In the application a Kuwahara filter is implemented GPU Pro (Engel, 2010, pp. 247-250) . The filter was applied using a pixel shader. It works by sampling four subregions surrounding the pixel in the texture you're currently on. The size of the subregions is decided by the radius which can be dynamically set. The next step is to go through each pixel in the subregion finding the average colour of that sub region and the overall variance in colour of that sub region. After which each sub regions variance is compared with the others and the sub region with the least variance's average colour is set to be the colour of that pixel. This effect is applied to the overall screen and a mesh with a texture on it, so it can be viewed independent of the rest of the scene. To apply this effect to the scene two passes are done, one where all the meshes are generated and all relevant effects are applied which is then rendered to a texture. That texture is then passed through the Kuwahara filter pixel shader and rendered to the screen.

Kuwahara implementation code snippet



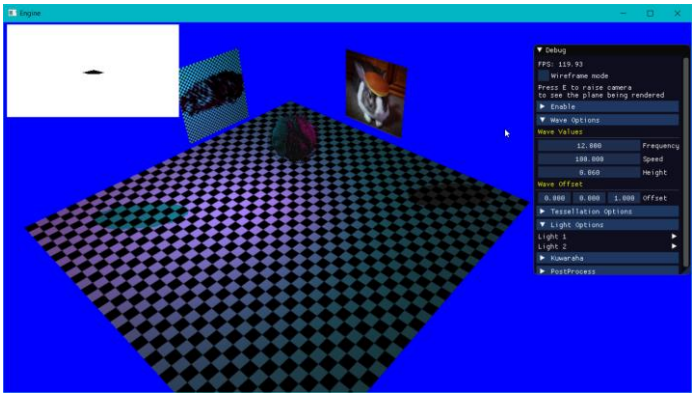
Same Picture with and without the Kuwahara filter applied



Kuwahara filter applied to the scene as a whole

Shadow Mapping

Using the two lights previously mentioned two shadow maps can be generated. It is important to note that although being point lights the direction for the light is just a normal vector going towards the centre. To generate the depth maps two depth passes are done where only the geometry that we wish to shadow is rendered and a depth value from the light is rendered onto a render texture. Repeating this for a second pass with the second light gives us the two shadow maps required. To make sure the shadows are accurate the same manipulation needs to be applied to the sphere during the depth passes as is being done in the final render. Passing in the same wave information and using the same calculation allows this to be achieved. Since the shadows are only appearing on the ground plane a shadow shader only needs to be used for the ground mesh. To achieve this however with two lights both lights view and ortho matrices need to be passed through the shader stages. These are passed within an array to the vertex shader where two positions are generated for the vertex from the lights point of view. Now in the pixel shader we can loop through comparing the depth value from the shadow map to the depth value of the point to determine whether lighting should be done. Since this is done for both lights the overall lighting value for that pixel in time is a blend of the two lights including shadows giving the following result.



One Spot light and One Point light generating shadows onto the plane

Critical Analysis

Vertex Manipulation

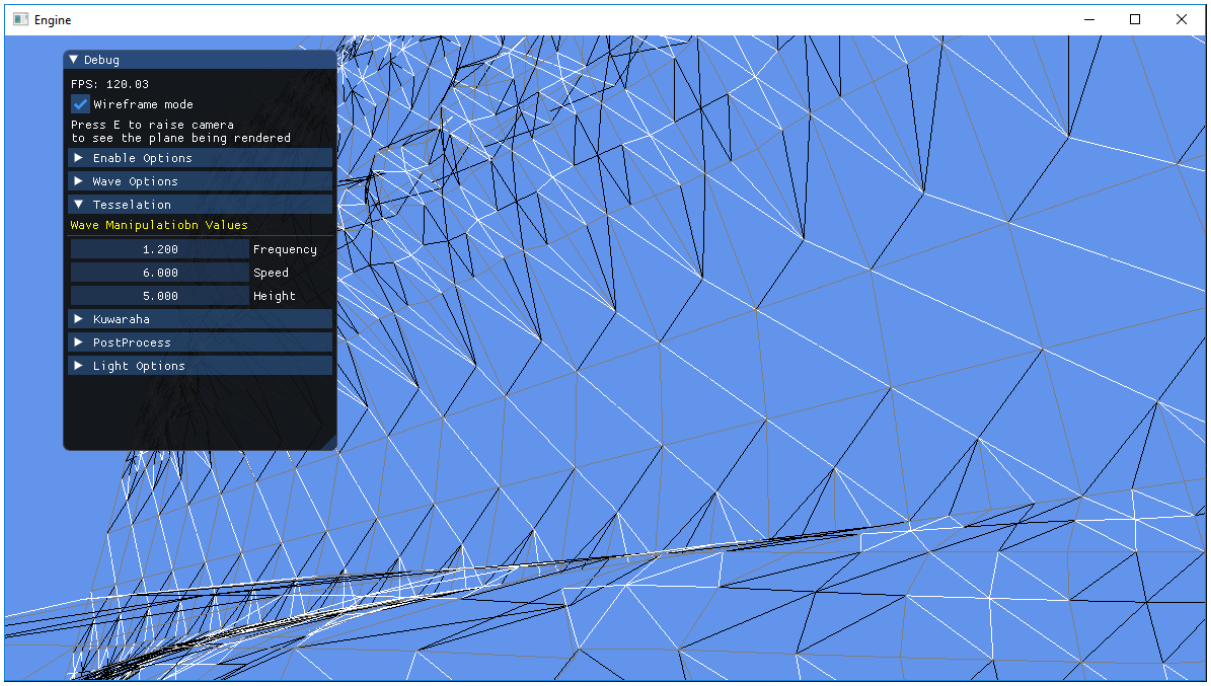
While implementing the sphere's manipulations another method was theorized but then scrapped. The original method however would be to apply the 2D wave manipulation to a rectangular mesh

and then manipulating it into a sphere much like a 2D map of the world onto a globe. It wasn't used in the end as it would take more time to create a mesh for it and then accurate displace a wave through it and then map it into a sphere when the other solution had already been theorised and would require less steps to achieve the same effect. The aim of the manipulation was to create an effect similar to that of an effect in a music video used on a sphere. The waves in the music video have variance instead of each being uniform like the solution presented. Given more time the solution presented on the book of shaders (Patricio Gonzalez Vivo, 2018) would be implemented in with the 3D ripple wave by applying several sine waves of different frequencies, speeds, amplitudes and offsets. Several attempts for generating normals for the manipulated sphere were attempted. Originally the same manipulation was applied to the surrounding four vertices to find their displaced positions. Then taking vectors from each of those to the centre point and getting the cross products of those vectors and averaging to get a normal was attempted. But applying the manipulation exceeded the instruction count on vertex shader however at the time the project was not using shader model 5.0. Next was to try calculating the differential equation of the equation at that point to get two tangent lines from which you could acquire vectors from to cross product to a normal. This proved too difficult to implement when having to take to differentiate three components instead of two. Instead as tessellation wasn't being applied the geometry shader could be used to get the positions of the surrounding displaced vertexes and subsequently get the vectors and then cross product for the normal.

Tessellation

When researching dynamic tessellation, a problem that came about is popping would occur between adjacent quads due to the shared edges being tessellated by different amounts. The solution in Practical Rendering (Jason Zink, 2011) samples the surrounding four edges to the current quad being tessellated and calculates how much they will be tessellated. The lower of the two tessellated values would be applied to that edge. The current method implemented also achieves this effect the problem lies with taking dynamic tessellation further. Ideally if there is little detail in the region there would be no reason to tessellate it by a major amount, and vice versa with an area with lots of detail, you should tessellate it more than usual but with the current method distance to midpoint method it'd be more difficult to implement that sort of solution. However, with the interlocking tessellation method provided in Practical Rendering (Jason Zink, 2011, pp. 432-449) popping can be avoided while also tessellating high detail and low detail areas accordingly. In a simplified explanation using a compute shader a height map would be split into sections and those sections variance in height would be sampled and a value generated letting the hull shader know which patches to tessellate more and less.

Applying no tessellation when the mesh wasn't being displaced came with imperfections. Since the midpoint of the two side edges and the middle aren't being displaced yet but one edge is, these ridge-like disruptions appear where the tessellation hasn't been applied yet.

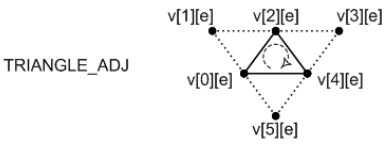


Issue in the dynamic displacement where one edge is being displaced but the midpoint isn't yet so no tessellation is being applied.

To fix this problem a check would be made when tessellation the midpoint to see if either of the vertices it had been calculated from was being displaced and if so to apply tessellation instead of not because the midpoint isn't yet.

Geometry Shader

Since the Sphere isn't being tessellated triangle adjacency should be able to be used to generate more accurate normals. It was attempted however the mesh would not render even after getting the order in which the vertices are passed in and trying multiple combinations. Due to time constraints it was scrapped and instead just using the three vertices of the triangle was used. Although not giving perfectly smooth normals the effect generated was enough but given more time using the triangle adjacency more accurate normals would be implemented through the same technique but also using vectors between the current vertices and the adjacent ones.



Cropped image of the vertex read order for the geometry shader from Microsoft docs (Mircosoft Docs, 2018)

As for the particle rotation the temporary fix of shifting the y angle depending on when it's greater or less than 90° and – 90° respectively would be replaced through applying a similar method to billboarding quads to face a camera. Instead of using a camera up right and forward vector to shift the quad the normals components would be used instead. As well when the quads are generated some quads are being redrawn as some vertices are shared between triangles and since a quad is drawn at each vertex it doesn't check if a quad has already been drawn at that position. With further implementation a different sphere mesh would be used for the particles constructed using a control point mesh instead. Only problem generating normals as there'd be no adjacent vertices to calculate them off of. At this point the differential equation of the vertex manipulation equation would have

to be implemented to calculate normals from the tangents. However, if calculating the tangents to calculate normals, a normal map may as well be used since the normal map just stores necessary vectors to calculate normals in tangent space. Since we are already calculating tangents with differentiation we may as well apply a normal map since it would provide higher quality normals. Normal maps uses the rgb values in a texture to give the xyz values of the normals. The important thing to note though is that the normals are given in tangent space. This is so the normal map can be applied to any shape in any rotation. If the vectors were stored in world space a normal facing out in the z direction when applied to a plane facing in the x direction would be wrong. But if we have normals in tangent space and then transform them using the same vertex manipulation and the world matrix. (Learn OpenGL Normal Mapping, 2018)

Pixel Shader

This version of the Kuwahara filter implemented is the most basic implementation that doesn't attempt to combat its downfalls where it cannot handle noise effectively or block artifacts. If to be continued the more complicated versions of the Kuwahara filter would be implemented using a generalized Kuwahara filter or the anisotropic Kuwahara filter from GPU Pro. The generalized Kuwahara filter instead of taking four square sub regions a disc split up into smaller subregions would be used as well as instead of taking the subregion with the smallest variance the result would be determined by a weighted sum of the means of the subregions with the smallest variance providing smoother region boundaries and fewer artifacts (Engel, 2010, pp. 251-254). The Anisotropic Kuwahara filter takes the improvements of the generalized filter further by manipulating the shape of the subregions to match the shape of boundaries to remove clustering artifacts from regions with directional features (Engel, 2010, pp. 255-261).

Lighting

In the current model lighting isn't very scalable having to recreate shaders for different amounts of lights. Now, the amount if lights in the shader are hard coded to two and if more were wanting to be implemented new shaders would need to be generated. This is wildly inefficient, so another method would need to be implemented. One method would be to create a shader that handles a large amount of lights and pass a variable to the shader saying how many lights are being passed and to do the normal light calculations in a loop that just ends at the amount of lights it's been told to do. Another method that was theorized was to do a render pass for each individual light and blend the final textures together and although very scalable in terms of code the performance drop of having to render the scene multiple times and do all the tessellation and manipulation per pass would be very inefficient just to generate a light. However, to continue on this train of thought deferred shading was come to which seems like the most efficient method for a project of this scale. The idea is to hold off on the lighting for as long as possible and when rendering the scene to store all the geometric information into a G-buffer (all the light relevant textures generated from the first pass) that can be passed into the lighting pass. In the lighting pass all the lighting can be applied to the geometry as needed. A bonus to this method is vertex data doesn't need to be passed all the way from the vertex shader down to the pixel shader all that needs to be passed is the texture information generated. Since the scene has been rendered already the information that is outputted to the G-buffer is actually ready for screen space it ensures that all the pixels we calculate lighting

for are the only pixels that needed to be lit anyway since any other geometry would have been hidden. One of the disadvantages is the amount of data generated from generating all the relevant high-resolution lighting maps. (Learn OpenGL Deffered Shading, 2018)

Shadow mapping

One of the problems with the current setup of shadow mapping as currently two point can lights be used. However, a shadow map is generated using the direction of the light but a point light shines in ever direction. In the current implementation we have it, so the direction just points towards the object we want shadowed but if this was to be fully implemented into a scene with multiple objects all around the light it wouldn't work. We would need to create a shadow map for each of the axes to get accurate lighting. This could be achieved through cube mapping the environment and then generating a depth map for each face of the cube. (Learn OpenGl Point Shadows, 2018) This would give us shadows in all directions as a shadow map has been generated in every axis direction. A similar problem lies in the current solution that was present with the lighting and that is the shadow shader is hardcoded to handle only two shadow maps and two lights. A new shader would need to be created for a different amount of lights. A similar solution as suggested before could be attempted in using a shader that can take in a bunch of lights / shadow maps and then passing a value that indicates how many lights and shadows have been passed in and to loop through and calculate accordingly due to that value.

References

Engel, W. (2010). *GPU Pro Advanced Rendering Techniques.* New York: CRC Press.

Jason Zink, M. P. (2011). *Practical Rendering and Computation with Direct3D 11.* Boca Raton: CRC Press.

Learn OpenGL Deffered Shading. (2018, December). Retrieved from Learn OpenGL: https://learnopengl.com/Advanced-Lighting/Deferred-Shading

Learn OpenGL Normal Mapping. (2018, December). Retrieved from Learn OpenGL: https://learnopengl.com/Advanced-Lighting/Normal-Mapping

Learn OpenGl Point Shadows. (2018, December). Retrieved from Learn OpenGL: https://learnopengl.com/Advanced-Lighting/Shadows/Point-Shadows

Lengyel, E. (2012). *Mathematics for 3D game programming and computer graphics.* Course Technology/Cengage Learning.

marinacci, J. (2018, November). *Water Ripples with Vertex Shader*. Retrieved from Medium: https://medium.com/@joshmarinacci/water-ripples-with-vertex-shaders-6a9ecbdf091f

Mircosoft Docs. (2018, December). Retrieved from https://docs.microsoft.com/en-us/windows/desktop/direct3dhlsl/dx-graphics-hlsl-geometry-shader

Patricio Gonzalez Vivo, J. L. (2018, November). *The Book of Shaders Fractal Brownian Motion*. Retrieved from The Book of Shaders: https://thebookofshaders.com/13/