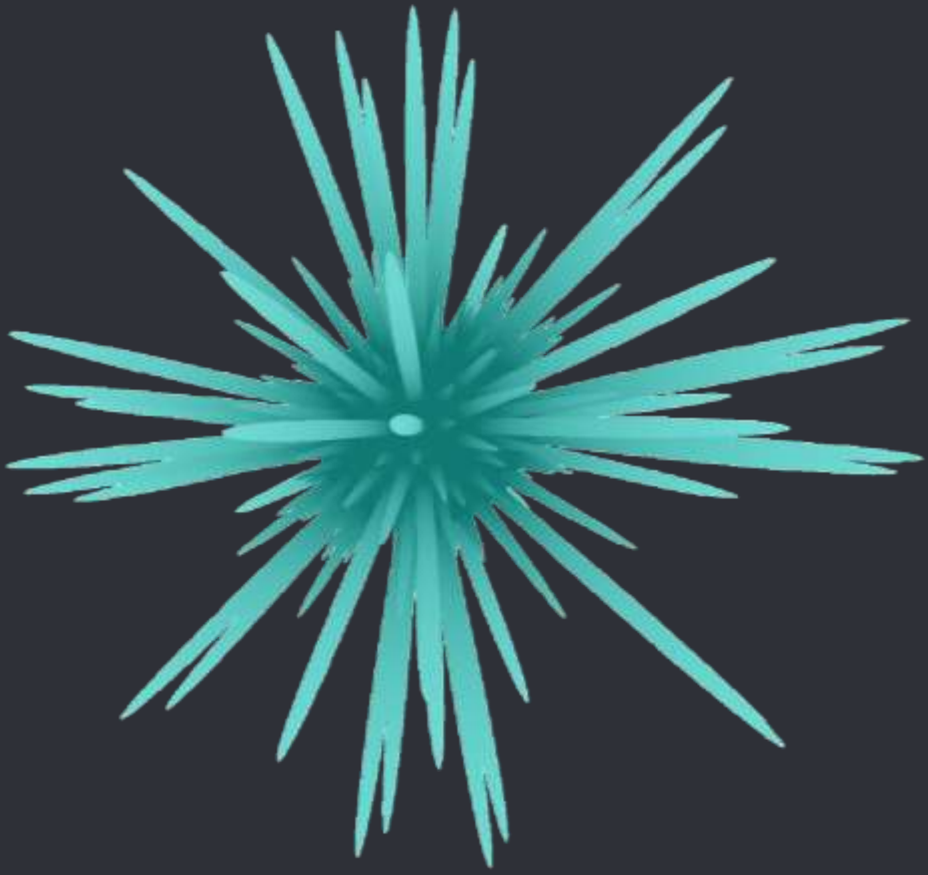


Graphics Programming with Shaders

CMP301

Joshua Hale
1402439

Introduction

An application was created to demonstrate the application of shader programs in DirectX11. The application uses a simple sphere (using a cube-sphere algorithm for vertex placement) and passes this sphere to all stages of the programmable pipeline to create the displayed effect.

The application was created using a machine with the AMD R9 Fury graphics card with the following specifications:

- Memory 4096 MB
- Memory Clock 500 MHz
- Core Clock 1000 MHz
- Total Memory Bandwidth 512 GByte/s

In addition to this, the machine had AMD FreeSync enabled synchronising monitor and GPU refresh rate.

The window renders natively at 1920x1080, however the application can be maximised or the user may adjust the window to the desired size. This currently causes distortion rather than adjusting the viewport and therefore interacting with imgui becomes quite difficult as the recorded mouse position is different to that displayed.

User Controls

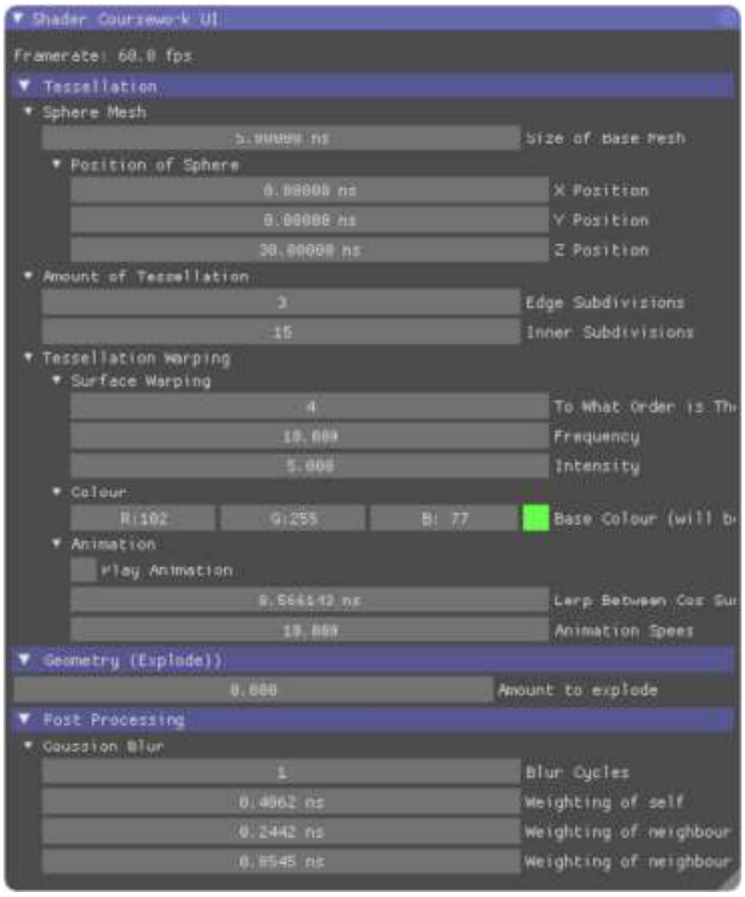
Input

W	Move forwards
A	Move left
S	Move backwards
D	Move right
Q	Move up
E	Move Down

Arrow Up	Look Up
Arrow Down	Look Down
Arrow Left	Turn Left
Arrow Right	Turn Right
Esc	Exit Application
LMB	Interact with GUI

imgui

A graphical user interface was implemented using the imgui implementation. This allows various properties that are used by the application or its shaders to be adjusted at runtime. All numbers are adjustable by dragging and are clamped at reasonable values.



```
▼ Shader Coursework UI
Framerate: 60.0 fps
▼ Tessellation
  ▾ Sphere Mesh
     5.00000 ns            Size of Base Mesh
    ▾ Position of Sphere
       0.00000 ns          X Position
       0.00000 ns          Y Position
       30.00000 ns         Z Position
  ▾ Amount of Tessellation
     3                     Edge Subdivisions
     15                    Inner Subdivisions
  ▾ Tessellation Warping
    ▾ Surface Warping
       4                   To What Order is Th
       10.000              Frequency
       5.000               Intensity
    ▾ Colour
       R:102  G:255  B: 77  🟩 Base Colour (will b
    ▾ Animation
       ☐ Play Animation
       0.566142 ns         Lerp Between Cos Su
       10.000              Animation Speed
▼ Geometry (Explode))
   0.000                   Amount to explode
▼ Post Processing
  ▾ Gaussian Blur
     1                     Blur Cycles
     0.4062 ns             Weighting of self
     0.2442 ns             Weighting of neighbour
     0.0545 ns             Weighting of neighbour
```

Framerate
The current framerate is displayed for user reference.

Sphere Mesh
The size and position of the base shape can be adjusted using the click drag input.

Amount of Tessellation
The number of inner or outer subdivisions on for each control point patch can be adjusted.

Tessellation Warping – Surface
The parameters for the equation which distorts the sphere can be adjusted here.

Colour
The base colour for the shape (this is inverted before rendering).

Animation
A checkbox for whether the shape is static or animated, also shown is its stage in the animation (between 0 and 1), which is adjustable when static. The speed of the animation can also be adjusted.

Geometry Shader Explode
The amount to explode the shape out from the centre.

Gaussian Blur
The number of times the blurred texture is re-blurred before rendering can be adjusted, as can the weightings of neighbour colour contribution used in the blur algorithm.

C++ Classes and Implementation

DXFramework

The application is built upon the DXFramework which was provided for this module. Some edits have been made to the framework for the purposes of this implementation. Firstly, the imgui implementation was integrated with the `BaseApplication` and `D3D` classes. `BaseApplication` is where the gui receives its input information, and `D3D` is where the gui is rendered so that it is the last possible item drawn to the screen. A further modification to the framework was the inclusion on the `TessellationSphere` class. This works in very much the same way as the `SphereMesh` class however it saves the primitive topology information as `D3D11_PRIMITIVE_TOPOLOGY_3_CONTROL_POINT_PATCHLIST` which is required for passing to the hull shader for tessellation.

App1

The application class is used for the running of the implementation. It creates all objects upon creation, handles all logic frame by frame and then specifies how the scene is rendered.

UiManager

This class handles the drawing of the custom imgui window as well as all input data received from the gui. This data is all public as it is required by the application class for logic or sending to any shader classes.

TessellationShader

This class is responsible for directing data to all of the stages of the pipeline which are involved in rendering the warped sphere shape. In addition to the required Hull and Domain shaders for tessellation, the class also uses a Vertex shader as an intermediary for the Hull shader and a Pixel shader to output its received data. A Geometry shader is also managed by this class as it receives vertex information directly form the Domain shader in order to perform work on the tessellated vertices before passing resultant vertex information to the Pixel shader.

TextureShader

Sends texture information to the texture Vertex and Pixel shaders.

HorizontalBlurShader

Used as part of the Gaussian Blur algorithm, this class sends a down-scaled texture of the screen to the horizontal blur Pixel and Vertex shaders.

VerticalBlurShader

Also used as part of the Gaussian Blur algorithm, this sends the horizontally blurred texture to the vertical blur Pixel and Vertex shaders.

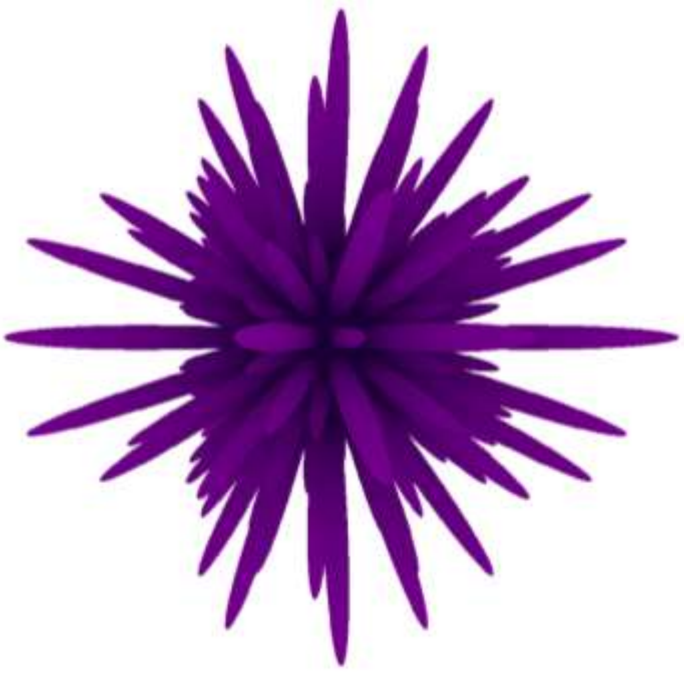
Shader Implementation

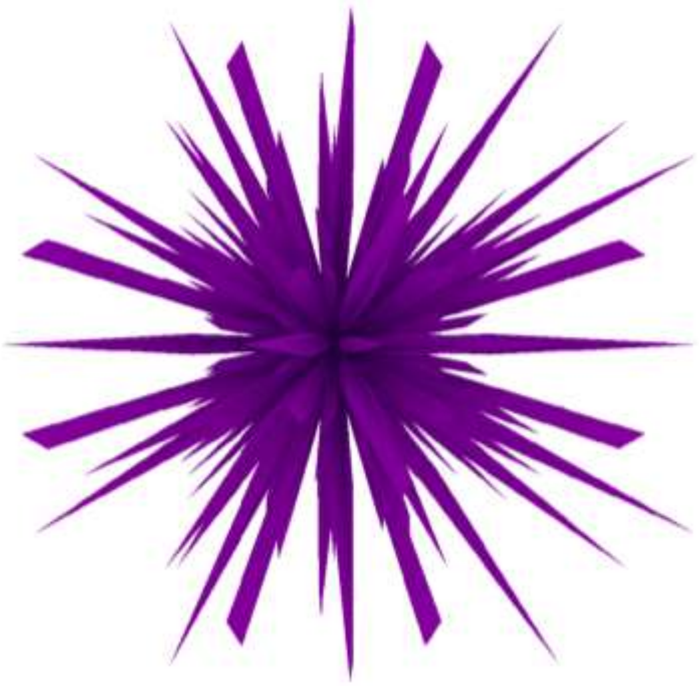
TessellationShader

Tessellation_vs.hlsl
Receives a vertex and multiplies it by the world matrix to get its position in 3D space.

Tessellation_hs.hlsl
Prepares a control point patch to hand to the tessellation stage. Receives vertex information from the Vertex shader. Also receives input from the application using a constant buffer. This information is used to determine how many times each edge of the triangular patch will be split and what the tessellation factor is for

```
cbuffer TessellationSetupBuffer : register(b1)
{
    int3 edgeSplitting;
    int innerSplitting;
}
```

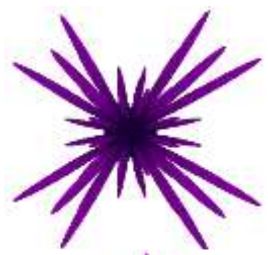

1 High Tessellation


2 Low Tessellation

Tessellation_ds.hlsl
This stage receives all vertices after tessellation has occurred, it then warps them according to an equation and generates a colour tone based on the vertex's new position relative to the original radius of the sphere. The shader does this for two similar equations and then interpolates between them to create an animated effect. The major information being sent to this stage from the application is contained in a

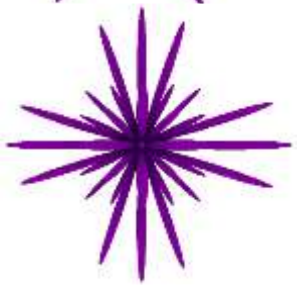
```
cbuffer TessellationWarpBuffer : register(b2)
{
        int powers;
        float repeats;
        float severity;
        float lerpAmount;
        float3 baseColour;
        bool targetSin;
}
```

constant buffer. The top three variables come from the gui and are the parameters for the equation which generates the shape. `baseColour` also comes from the gui and controls the colour which is inverted for the final outputted colour of the shape with tone. `lerpAmount` is used by the application class to calculate where in the animation the shader should be at, this can also be adjusted by the user in the gui when the animation is not playing. Finally, `targetSin` is a bool sent by the application to determine in which direction the animation is going: toward the sin configuration or toward the cos configuration. The set of parametric equations which describes the sin configuration is as follows:



$$x_1 = x_0 + (x_0 severity(\sin(x_0 repeats)\sin(y_0 repeats)\sin(z_0 repeats))^{powers})$$
$$y_1 = y_0 + (y_0 severity(\sin(x_0 repeats)\sin(y_0 repeats)\sin(z_0 repeats))^{powers})$$
$$z_1 = z_0 + (z_0 severity(\sin(x_0 repeats)\sin(y_0 repeats)\sin(z_0 repeats))^{powers})$$

Additionally, the cos configuration is:



$$x_1 = x_0 + (x_0 severity(\cos(x_0 repeats)\cos(y_0 repeats)\cos(z_0 repeats))^{powers})$$
$$y_1 = y_0 + (y_0 severity(\cos(x_0 repeats)\cos(y_0 repeats)\cos(z_0 repeats))^{powers})$$
$$z_1 = z_0 + (z_0 severity(\cos(x_0 repeats)\cos(y_0 repeats)\cos(z_0 repeats))^{powers})$$

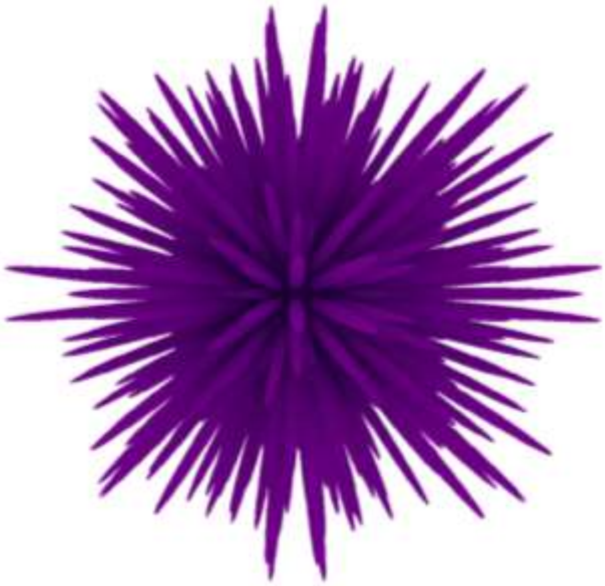
It would be extremely inefficient to do these six calculations in their entirety for each vertex every frame, especially due to the presence of multiple trigonometric functions. Therefore, a series of intermediates are used in the shader to perform the calculations. Displayed are the calculations performed for the sin configuration, the same process is followed for the cos calculations.

```
float vxr = vertexPosition.x * repeats;
float vyr = vertexPosition.y * repeats;
float vzr = vertexPosition.z * repeats;

float sinWarp = 1.0f;
float sinvx = sin(vxr);
float sinvy = sin(vyr);
float sinvz = sin(vzr);
float sinvxyz = sinvx * sinvy * sinvz;

for (int i = 0; i < powers; i++)
{
        sinWarp *= sinvxyz;
}

float sinSev = sinWarp * severity;

sinVertexPosition.x += radialVector.x * sinSev;
sinVertexPosition.y += radialVector.y * sinSev;
sinVertexPosition.z += radialVector.z * sinSev;
```

For animation the shader generates the final vertex position based upon an interpolation between the `sinVertexPosition` and the `cosVertexPosition`.

The position is output in this state (no multiplication against vertices) as the 3D space position is required by the Geometry shader in the next stage.

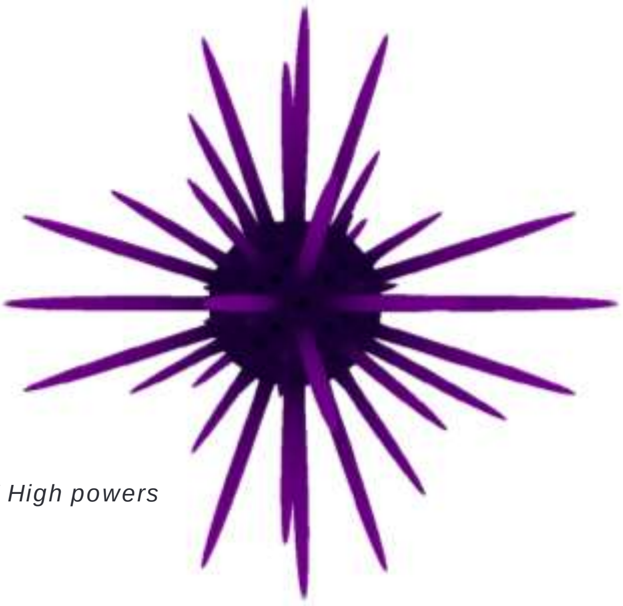
The following page displays the effects each of the parameters has on the equation of the warped shape.


3 High repeats

4 Low repeats

5 Low severity

Tessellation
Sphere Mesh
Amount of Tessellation
Edge Subdivisions
Inner Subdivisions
Tessellation Warping
Surface Warping
Frequency
Severity
Colour
Animation
Play Animation
Animation Speed
Geometry (Explode)
Amount to Explode
Post Processing
Gaussian Blur
Blur Cycles
6 high severity

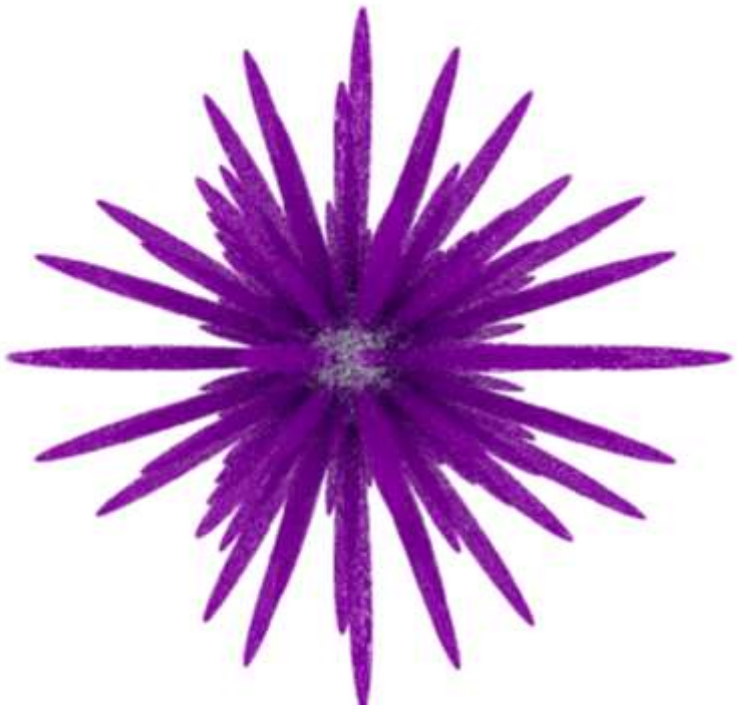

7 High powers

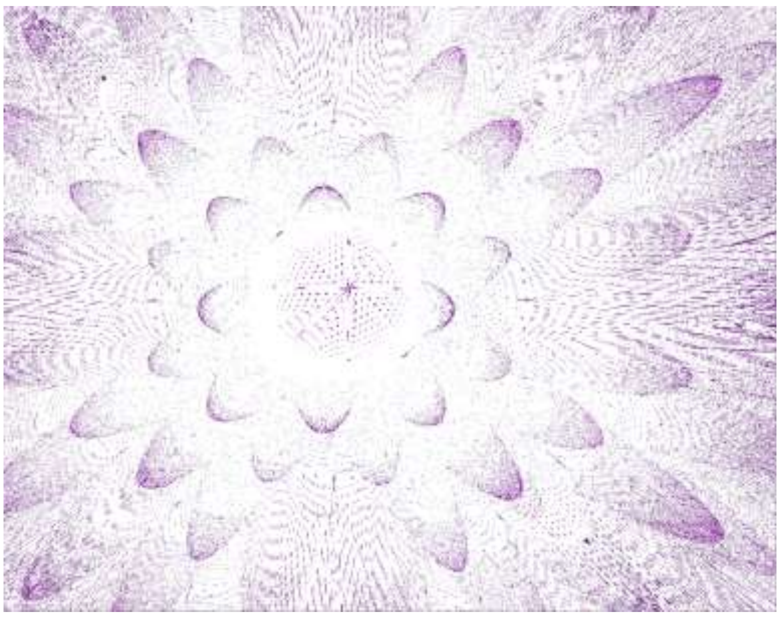
8 Low powers

`Triangle_gs.hlsl`

```
cbuffer GeometryBuffer : register(b1)
{
    float explode;
    float3 padding;
}
```

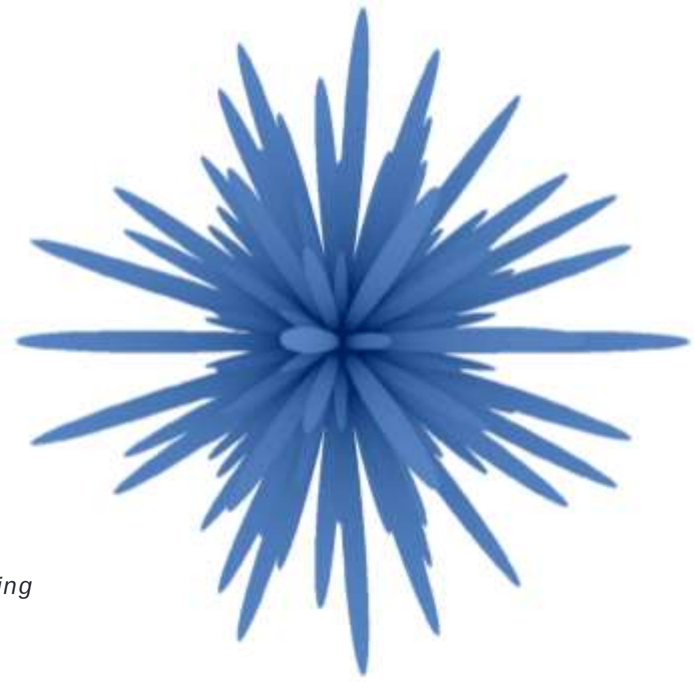
This stage receives the vertex positions from the previous stage and explodes them as triangles a given distance along the vector from the centre of the sphere to the first vertex in the triangle's position. The amount that the triangle is displaced along this vector is passed into the shader from the application (set in the gui) via a constant buffer.


9 Low Explosion on Geometry Shader


10 High Explosion on Geometry Shader

`Tessellation_ps.hlsl`

Finally, the fragment shader outputs the colour of the pixel as defined by the tonal calculation in the domain shader.


11 Base Colour Selection is Orange, Resulting in Blue Shading

Gaussian Blur

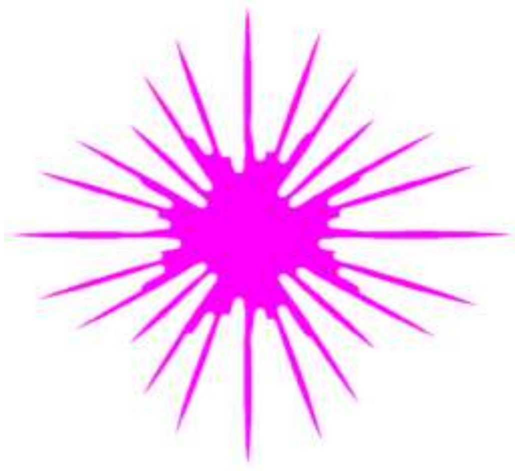
For post processing, the display can be blurred using a Gaussian Blur algorithm. The process works as follows:

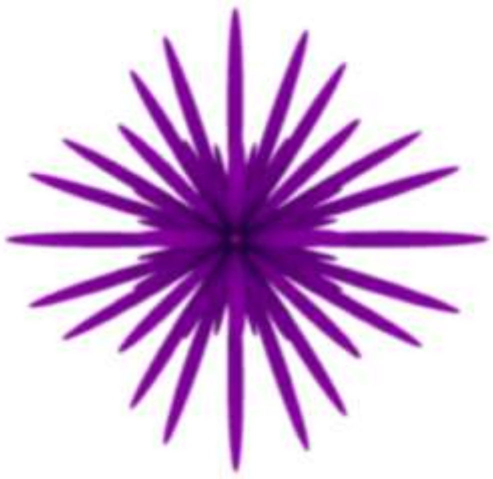
- Render entire scene to texture
- If user has selected at least one blur loop
    - Down scale texture to one half the size
    - Blur each pixel by combining its colour with that of 2 neighbours either side
    - Blur each pixel by combining its colour with two neighbours above and below
    - Render completed blur to a texture double the size
    - Repeat if required
- Render texture to screen

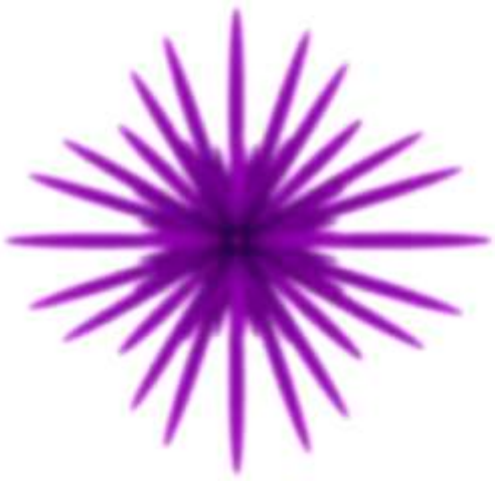
`horizontalBlur_vs.hlsl & verticalBlur_vs.hlsl`
Both vertex shaders get a floating point uv coordinate for the neighbouring pixels either horizontally or vertically depending on the shader. The screen width is passed in using a constant buffer. This constant buffer also contains the weightings used for the next stage which are sent via the `OutputType`

`horizontalBlur_ps.hlsl & verticalBlur_ps.hlsl`
Both pixel shaders create the pixel colour based on the algorithm:

$$pixelColour += neighbour.colour * neighbour.weight$$


12 High Weightings for Neighbours


13 Low Number of Blur Loops


14 High Number of Blur Loops

Critical Analysis

Original Intentions

The original design of this application also included a point light with a specular component which would reflect of the surface of the created shape, expanding on this the shape would self-shadow. The essential requirement for correct lighting is that the normal of the surface are correct. As the surface is completely generated in the Domain shader, the normal would also have to be calculated here. I tried at length to implement this, however was ultimately unable to, due to the complexity of the formula I chose.

Finding the normal to a surface required first reducing the equation's variables into a u and a v component, by replacing all xyz's with their spherical coordinate counterparts:

$$x = radius * \cos(u) * \sin(v)$$
$$y = radius * \sin(u) * \sin(v)$$
$$z = radius * \cos(v)$$

This would result in the calculated xyz components being generated by two generated variables (treating the user-inputted variables as constants). By adding the x, y and z parametric equations we get the parametric equation of the surface. The normal can then be calculated by taking the cross product of the surface partially differentiated with respect to u and partially differentiated with respect to v.

$$\underline{n} = \frac{\partial S}{\partial u} \times \frac{\partial S}{\partial v}$$

Even with the aid of wolframalpha.com to assist with the differentiating legwork, this process proved as nothing more than a time-sink. The sheer number of terms in the resultant differential equations made the margin for error colossal (not to mention the combined error of so much floating point arithmetic).

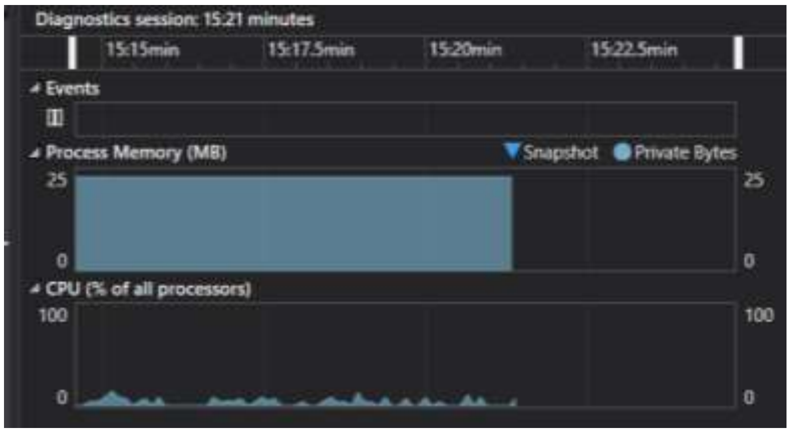
In order to work around this problem, I added a tonal calculation which gives the effect that the shape is evenly lit from every angle, even though this is not the case. I would very much like to expand on this application and this would be the first problem I would solve. The formula used for the shape warping absolutely does not need to be as complex at is it is, especially when making normal calculations close to impossible is entirely self-defeating.

Other Issues

The window resizing issue is a pain and something which again will be high on my priority list in the continuing development of this application, although I have been informed by my colleagues that imgui's mouse position detection is a widespread issue with an annoying work-around.

Conclusion

The main purpose of this application was to demonstrate the abilities of using shaders for geometry rendering and calculations which would not be possible on the CPU. Using the Visual Studio built-in profiler, it is obvious that this has significant advantages, as the CPU usage was exceptionally low, and the memory being used for a very complex shape was low as well.



References

All code/algorithms used is either that provided in the module, from imgui or created by myself.

Tools which were useful in the development of this application:

imgui
https://github.com/ocornut/imgui

WolframAlpha
http://www.wolframalpha.com/

HLSL Intrinsic Functions API
https://msdn.microsoft.com/en-us/library/windows/desktop/ff471376(v=vs.85).aspx

HLSL Tools for Visual Studio
https://marketplace.visualstudio.com/items?itemName=TimGJones.HLSLToolsforVisualStudio