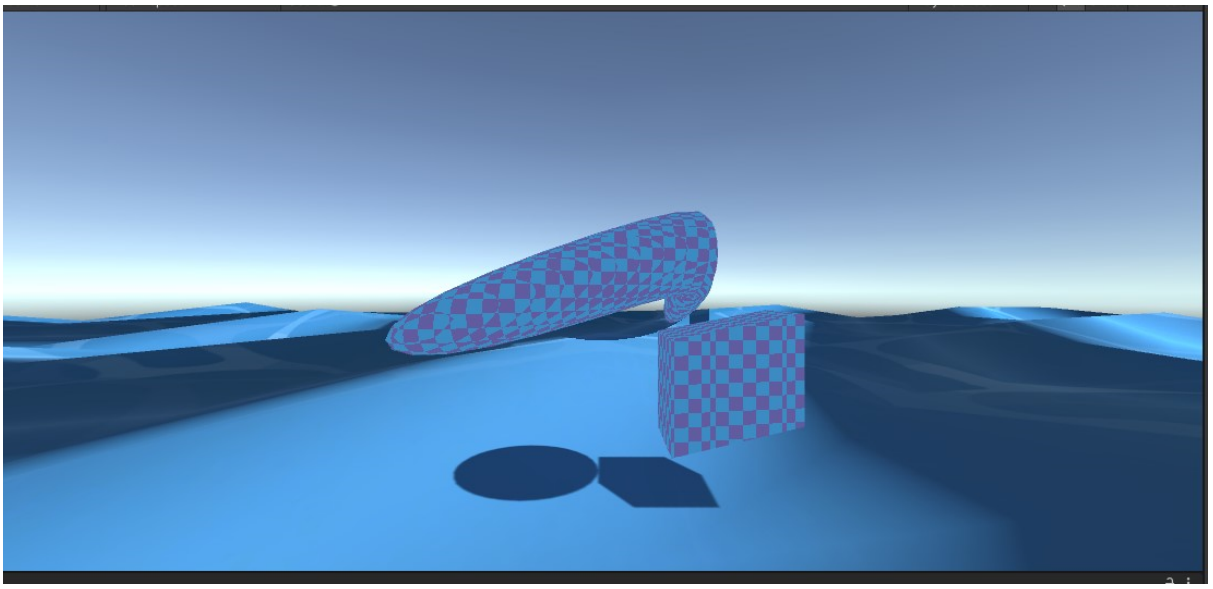
GPU Shader



a) The *graphics pipeline* – also referred to as the *rendering pipeline* – is a process that converts a 3D scene into a 2D image. It involves transformations between spaces and conversion from one data type to another.

A *vertex* is a data structure in computer graphics that describes attributes such as the position of a point in 2D or 3D space or multiple points – these points represent meshes; it is the endpoint where two lines or points come together. A vertex is stored as a 3D vector, where its three values represent the 3D coordinates of the vertex.

A vertex shader manipulates the positions of vertices by defining new ones. More specifically, a vertex shader is a programmable function that transforms attributes of vertices such as colour, texture, position and direction from the model space to the view space with the camera at the centre (matrices represent these view spaces). It also allows for the original objects to be distorted or reshaped.

For the peer review, a *PatternShader* renders any object that utilises it distorted by applying a *sin* function – based on the trigonometric sine function used in right-angled triangles – to the vertex's x position and then adding the output of that function to all the coordinates of the vertex.

```
// sine angle = opposite/hypotenuse of a right-angled triangle
vertexData.vertex.xyz += sin(vertexData.vertex.x);
```

b) A sine wave, an s-shaped, smooth wave oscillating above and below zero, ensures that a deformed object can move. The sine wave can be implemented by using non-built-in uniform values such as a *frequency* (i.e. the number of cycles), *amplitude* (i.e., the maximum distance between the horizontal axis and the vertical position of the waveform, and the time value of the *y* coordinate.

```
vertexData.vertex.xyz += sin(vertexData.vertex.x * _Frequency + _Time.y) * _Amplitude;
```

c) A fragment shader – also known as a pixel shader – handles how pixels between vertices look; these pixels are interpolated between the defined vertices following specific rules. The first extension uses the interpolation of the vertex function's output of the *PatternShader* to draw a checkerboard colour pattern. The pattern is drawn from the interpolated vertex coordinates and then modified using several multiplications before the colours are displayed using a *lerp* function [1].

```
// Vertices are in local/model space, so they are relative to the camera and local to the object
// Vectors are used as an [x, y, z] position to describe the displacement of a vertex relative to the origin.
// Matrices are used to represent coordinate spaces, describing the origin and orientation of a space in which // // all positions in that space are // placed relative to
v2f output;

// sine angle = opposite/hypotenuse of a right-angled triangle
// the frequency and amplitude are customisable uniform variables
vertexData.vertex.xyz += sin(vertexData.vertex.x * _Frequency + _Time.y) * _Amplitude;

// Transforms a vertex point from model/local space to a camera's world space, then to a view space.
```

```
// The model is used to transform, scale or rotate a model/object to its place in the world (the world space).
// The view space is the space seen from the camera's point of view.
// Objects can be transformed to the view space by combining transformations and rotations using a view matrix.
// The clip space discarded any coordinates not to be used, turning others into fragments.
// This is done using a projection matrix, then translated to a screen view.
// The past function for one used below in Unity was mul(UNITY_MATRIX_MVP, v.vertex)
output.position = UnityObjectToClipPos(vertexData.vertex);

// The customisable density value allows for the vertices to be altered, i.e. changing the density
// worldPos below is the result of calculating the vertex's position in the world
output.worldPos = mul(unity_ObjectToWorld, vertexData.vertex) * _Density;
return output;
// The pattern is determined by the interpolated output from the vertex's processing function
// The pattern is determined by taking the floor of the vertex's coordinates in the world position, which are added together
 float pattern = floor(i.worldPos.x) + floor(i.worldPos.y) + floor(i.worldPos.z);
 pattern = frac(pattern * 0.5); // Returns the fractional part of a float value, which is then multiplied by 0.5
 pattern *= 2; // The pattern is multiplied again by two

 // A lerp function creates the customisable colour that is drawn on the shader
 float4 colour = lerp(_FirstColour, _SecondColour, pattern); // FirstColour and Second colours are Inspector values
return colour;
```

The other extension involves a *WavesShader* that uses Gerstner waves to extrude a plane-like object by manipulating its *vertex normals*, which are used to replace the true geometric normal of a surface. In graphics, manipulations of a normal vector are often used to simulate geometrical detail on otherwise planar surfaces [2].

Gerstner waves allow the water to move backwards and forwards, achieving a more realistic effect. Without going into the inner workings of how such a wave is calculated, the following code snippet shows how the coordinates of the vertex are passed into a GerstnerWave function and how its normals are modified by normalising the result of a cross-product [2].

```
float3 gridPoint = vertexData.vertex.xyz;
float3 tangent = 0;
float3 binormal = 0;float3 p = gridPoint;

// Creating three waves, passed into the GerstnerWave function that returns a 'GerstnerWave'
// These are calculated using the x, y, z coordinates from the vertex
p += GerstnerWave(_WaveA, gridPoint, tangent, binormal);
p += GerstnerWave(_WaveB, gridPoint, tangent, binormal);
p += GerstnerWave(_WaveC, gridPoint, tangent, binormal);

// A vertex normal at a vertex of a polyhedron is a directional vector associated with a vertex
// It is intended as a replacement for the true geometric normal of the surface
// To create the waves, the surface normal is changed by normalising the cross product from the modified // binormal and tangent vectors
 float3 normal = normalize(cross(binormal, tangent));
 vertexData.vertex.xyz = p
 vertexData.normal = normal;
```

References

[1] https://www.ronja-tutorials.com/post/011-chessboard/
[2] https://catlikecoding.com/unity/tutorials/flow/waves/