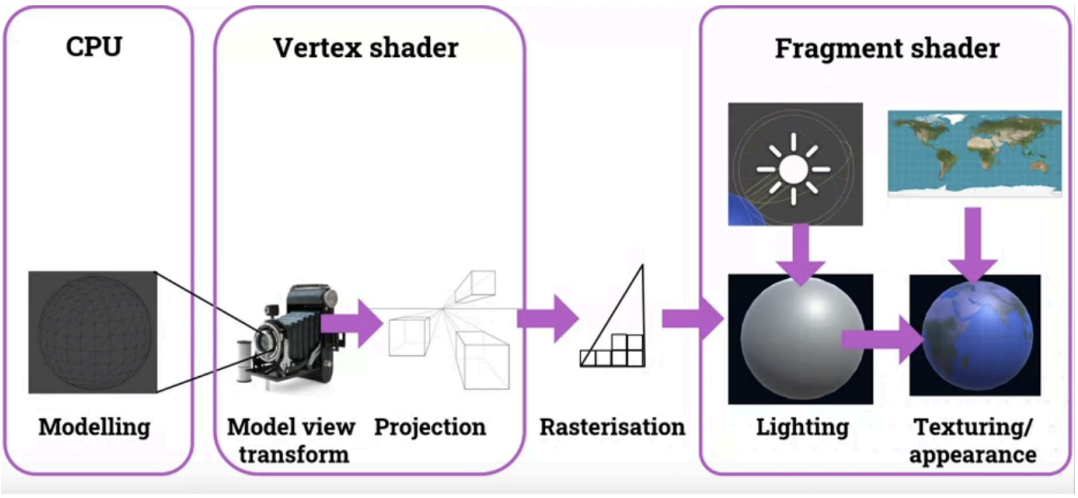
Topic 7 - GPU Programming

Graphics hardware and graphics processing units, also called GPUs, what have enabled the amazing quality of graphics we can see now in modern films and games

Vertex Shader: This is a small program that runs on the GPU that processes individual vertexes, after that the vertexes are combined into polygons and rasterized.



Vertex shader happens to vertexes and fragment shaders happen to fragments, those little patches of an object that will become pixels.

CPU Processor

CPU is a **serial** processor, what does that mean? It means that it performs instructions one at a time, in particular, if you're processing polygons or vertexes, it will process each polygon or vertex one at a time.

GPU Processor

GPU is a parallel processor so that multiple instructions can be happening at the same time. And that means we can process multiple vertexes, multiple fragments simultaneously, massively speeding up the process of rendering.

Why is the GPU so different that it can be so parallel in comparison to a CPU?

- Each vertex that you're processing is **independent of the other vertexes**, so there's no dependencies or data sharing between vertexes or fragments.
- So the only data that's shared between them doesn't change, it's the same across all vertexes

- It **doesn't matter what order you do them** in and in particular means you can do them in any order and at the same time in parallel.
- Dedicated Language GLSL, HLSL
- The open GL graphics standard has a language called GLSL for the GL shader language, and Microsoft developed the high level shader language, HLSL, which is what is used by Unity.

Q What is the relationship between GPUs and programming languages

The GPU and CPU code are in different programmes and GPUs use different languages from CPU programmes

Q: Think of an example of something that might be improved in a graphics scene by having a better GPU, and something that will not.

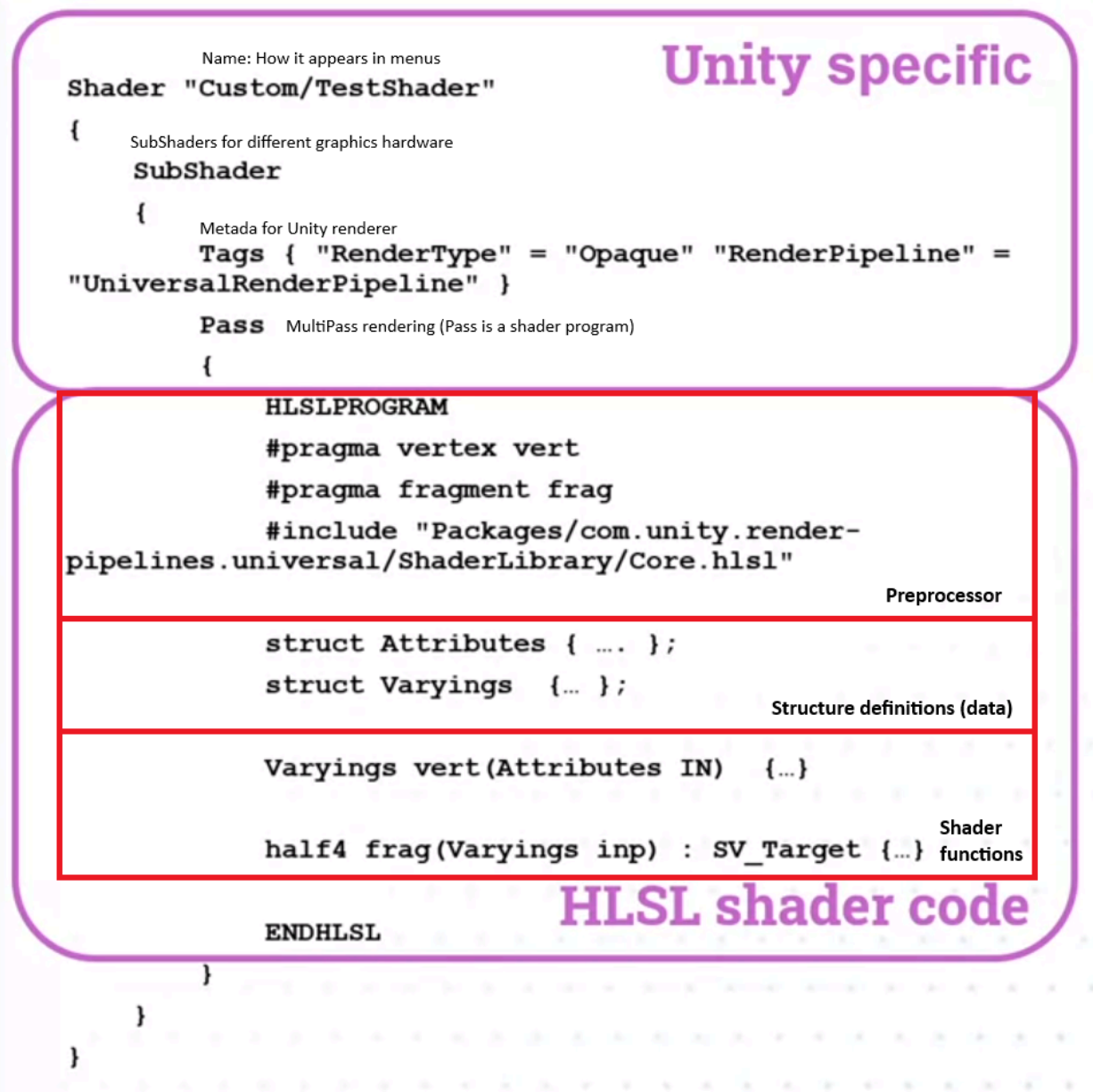
Having a better GPU will almost always increase the framerate. With a better GPU you can also have more polygons, higher resolution textures, better lighting effects and more complex surface properties.

The GPU won't have much effect on things like animation, physics or AI.

How Vertex Shaders Work

- First part of the GPU pipeline
- They Transform and project vertices
- Vertex Shaders **Inputs:** if you're doing things like modeling, transforms, and projections, you need those **transform matrices**, **modelview matrix**, **projection matrix**
- These inputs don't change between vertices; they're the same for all vertices and objects. So we call these **uniform variables**.
- Other types of data like **vertex position,** which is different for each vertex, but is specific only to that vertex so that you don't share the vertex position between vertices and we call this a **vertex attribute.**
- The Vertex Shader provides **output** that is **rasterized** and used by **fragment shaders.** This output is called the **Varying variable** (EG Transformed vertex position)
- Varyings are **interpolated** in the rasterization face for the Fragment Shaders, .

Shader Program



Vertex Shader Example

```
Varyngs vert(Attributes IN)
{
    Varyngs OUT;
    OUT.pos = mul(UNITY_MATRIX_M, IN.pos);  // model view transform
    OUT.pos = mul(UNITY_MATRIX_V, OUT.pos); // model view transform
    OUT.pos = mul(UNITY_MATRIX_P, OUT.pos); // Projection
    return OUT;
}
```

UNITY_MATRIX variables are the **Uniform** variables

Shortcut for doing the same as avobe

```
Varyngs vert(Attributes IN)
{
    Varyngs OUT;
    OUT.pos = mul(UNITY_MATRIX_MVP, IN.pos);
    return OUT;
}
```

The one you should use by Unity

```
Varyngs vert(Attributes IN)
{
    Varyngs OUT;
    OUT.pos = TransformObjectToHClip(IN.pos);
    return OUT;
}
```

An example of creating an effect

```
Varyngs vert(Attributes IN)
{
    Varyngs OUT;
    OUT.pos = IN.pos;
    OUT.pos = 0.5sin(IN.pos.y) + 1.0;
    OUT.pos = TransformObjectToHClip(OUT.pos);
    return OUT;
}
```

Fragment Shaders

- These are the shaders that act on fragments, these are the tiny little bits of an object that will become **pixels** if they eventually get drawn to screen
- **What is a fragment**: A part of a polygon
- In Unity, what is the most normal way of setting non-built in uniform variables? Through the inspector
- The **varying variables** are the **input** the fragment shader but this is interpolated
- The **output** of the fragment shader is usually the **color of a pixel,** sometimes the **z-value**
- **half4:** outputs or the return value of the function is a **pixel color**. It's a half the space of a float number

Simple Fragment Shader function

```
half4 frag (Varyings inp): SV_Target
{
     return half4(1.0,0,0,1);
}
```

Simple Fragment Shader function

```
half4 _BaseColor
half4 frag (Varyings inp): SV_Target
{
     return _BaseColor;
}
```

Uniform variable for Fragment Shader

```
Shader "Custom/TestShader"
{
    Properties
    {                          Make the property appear in unity inspector
        _BaseColor("Base Color", Color) = (1, 1, 1, 1)
    }
    SubShader
    {
        Pass
        {
            struct Attributes { …. };
            struct Varyings  {… };
            CBUFFER_START(UnityPerMaterial)
            half4 _BaseColor;
            CBUFFER_END          Declare property in HLSL Code
            Varyings vert(Attributes IN)  {…}
            half4 frag(Varyings inp) : SV_Target {
                    return _BaseColor;
                        Use property in Vert or Frag shader
            }
            ENDHLSL
        }
    }
```

Example Unity Fragment Shader

```
_BaseColor("Base Color", Color) = (1, 1, 1, 1)
 _Frequency("Freqency", Float) = 1



CBUFFER_START(UnityPerMaterial)
    half4 _BaseColour;
    float _Frequency;
CBUFFER_END

half4 frag(Varyings inp) : SV_Target
{
      float intensity
            = 0.5*sin(_Frequency*_Time.y)+1.0;
      return _BaseColour*intensity;
}
```

In my videos I have said that transforms and projection typically happen in vertex shaders and lighting and texturing in fragment shaders. Why do you think that is?

Transformation and projection are applied to vertices not fragments. The vertices need to be projected into 2D before we can rasterize them and generate the 2D fragments which are input to the fragment shader.

Texturing adds fine detail to a surface. The main benefit is that the detail can be at a smaller scale than that of vertices and polygons. That can only be achieved if textures are applied to fragments, not vertices.

Lighting can actually be done in both vertex and fragment shaders, but doing it in fragment

shaders gives better quality because the lighting can be calculated correctly for each individual fragment. Otherwise they would be calculated at the vertex and interpolated, which would give less accurate values. (we will also see later that we can use texture mapping to influence lighting)