Materials and Textures

Pixels, colours, texturing and the Fragment Shader Stage

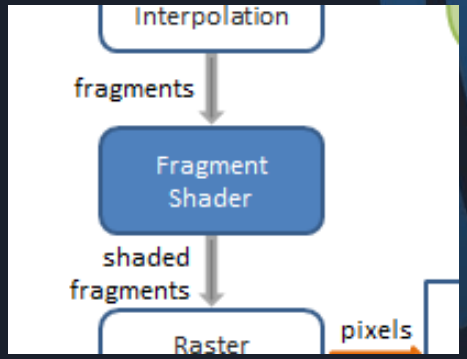
Programming – Computer Graphics

Contents

- Fragment Shader Stage
- Materials
- Textures
  - Texture Coordinates
  - Texture Addressing
  - Texture Filtering

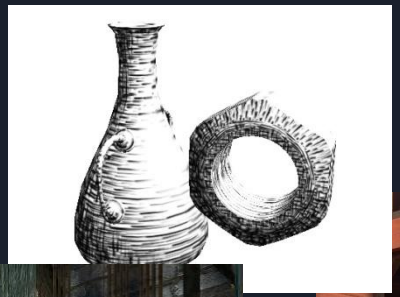
Fragment Shader Stage

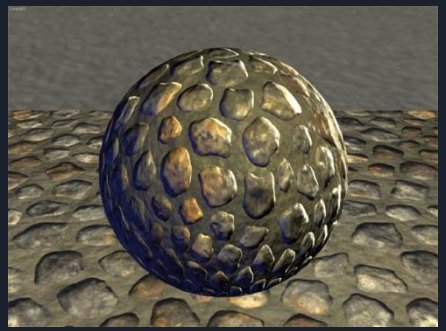
- This programmable stage is the final programmable stage
- Also called the Pixel Shader, as it typically is used to output a pixel colour to the screen
- Its job usually is to output a desired colour at the specified pixel
  - Based on the input data from the Rasteriser Stage
  - By sampling other buffers containing texture information
  - By sampling shader uniform variables

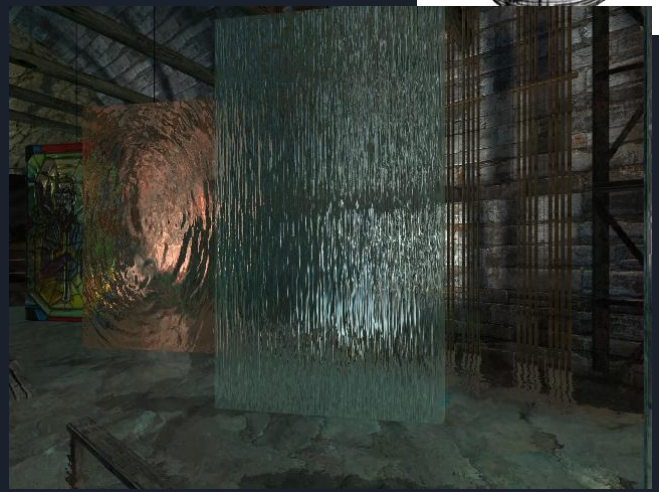


Fragment Shader Uses

- As fragment shaders are used to define the final pixel colour they can be used to achieve all kinds of effects!
  - Per-pixel lighting
  - Reflection / Refraction / Blur
  - Non-Photorealistic Shading
  - Many more!







Writing Fragment Shaders

- Fragment Shaders are written much like Vertex Shaders
- Their input comes from the Rasteriser Stage and they have access to…
  - Any data that was output from the last used programmable stage
  - Built-in math functions
  - Variables / Flow Control / Functions
  - Texture and Image Buffers bound to the device
- Their output is usually one or more pixel colours
  - We will talk about multiple output in a future session

Simple Fragment Shader

```
#version 410

in vec4 position; // input from the vertex buffer, not passed to other stages by default
in vec4 colour; // input from the vertex shader that we will pass as a custom output

out vec4 vColour; // output that goes to the next used programmable stage

uniform mat4 projectionViewWorldMatrix;

void main( ) {
	vColour = colour;
	gl_Position = projectionViewWorldMatrix * position;
}
```

Vertex Shader

```
#version 410

in vec4 vColour; // input from the last used programmable stage

out vec4 pixelColour; // output pixel colour (default if the only out)

void main( ) {
	pixelColour = vColour;
}
```

Fragment Shader

Materials

- A Material consists of the properties that define the surface of an object
  - Colour
  - Shine / Gloss
  - Texture / Roughness
  - Transparency
  - Reflectivity / Refraction

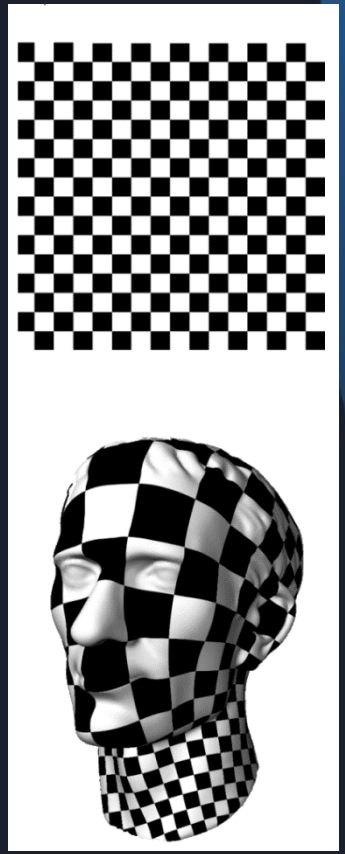
- When rendering an object we take the surface properties into account when calculating the pixel colours of the object
  - Despite all of the different properties and ways to define an object's colour, the most common way is by using a Texture buffer

Texturing

- One of the most common uses of a Fragment Shader is to sample Texture buffers
- A texture is simply a big image buffer of data bound to the GPU
  - 1-dimensional, 2-dimensional, and even 3-dimensional buffers are available!
  - Data contained in a texture isn't called a Pixel, it is called a Texel (Texture Element or Texture Pixel)
  - Hardware requires textures have power of 2 dimensions, ie 512x512, 1024x2048
- Sampling a texture has a few different rules to how they work
  - More on sampling soon

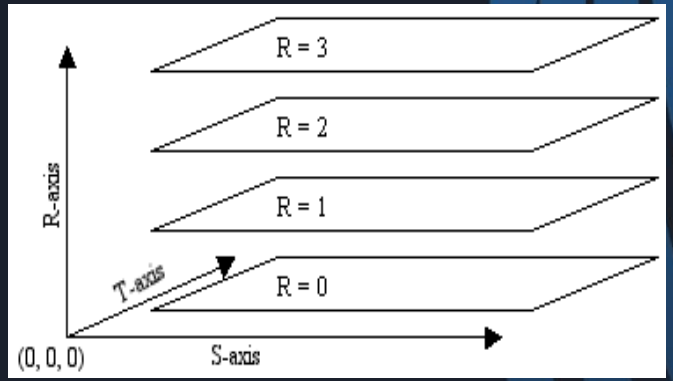
Texture Coordinates

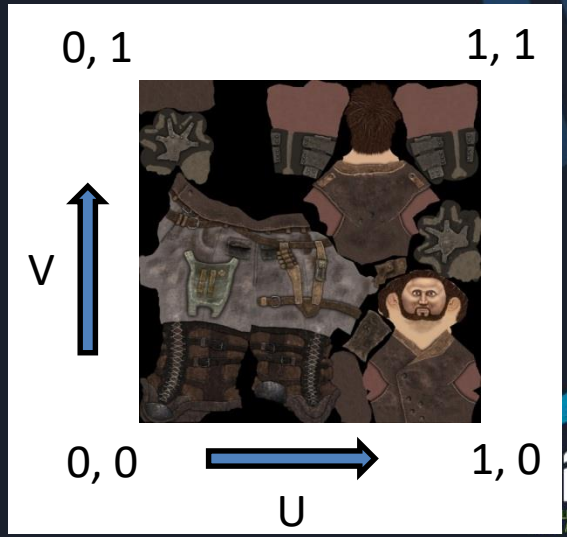
- Applying a texture to geometry is called texture mapping
- To be able to map a texture to geometry we need to know which parts of the texture apply to which parts of the mesh
  - To do this we need texture coordinates
- Texture coordinates are simply elements in a vertex buffer
  - For a 1-dimensional texture we need a float
  - For 2-dimensional we need two floats
  - For 3-dimensional we need three floats
  - 3-dimensional textures can be thought of as 2-dimensional ones layered on top of each other



Texture Coordinates

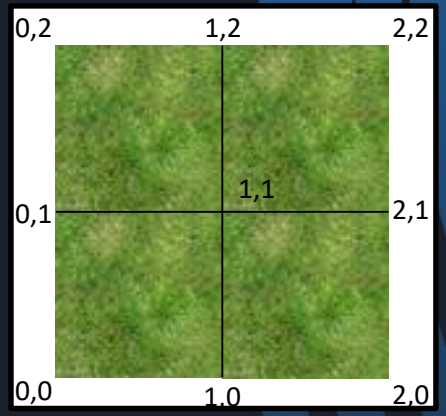
- Texture coordinates, although typically a vector, usually are referred as UVW or STR, rather than XYZ
  - In GLSL you can us stpq in addition to xyzw and rgba when accessing vectors, but not uvw!
  - p and q are used instead of r because r is already used in rgba!
- Texture coordinates are in the range 0.0 to 1.0
  - The U represents left to right
  - The V represents bottom to top
  - The W represents depth bottom to top
- Coordinates can be outside the [0,1] range, and there are different options that affect what happens when these ranges are used





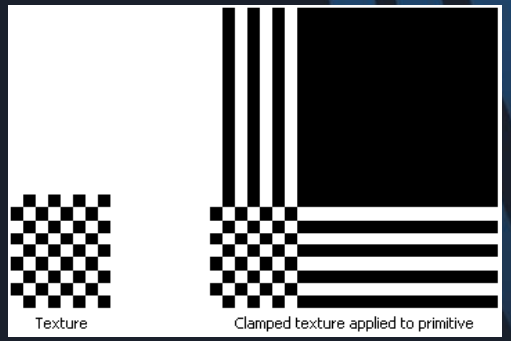
Texture Addressing

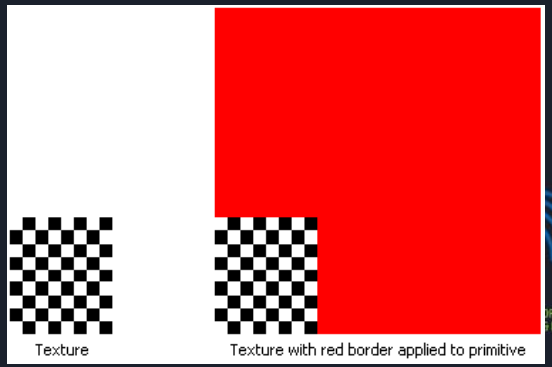
- Texture Addressing refers to how textures are sampled outside the standard [0,1] range
  - They are set in the application when creating a texture
- There are a few options
  - Wrap
  - Mirror
  - Edge Clamp
  - Border Clamp
- Wrap is the most common
  - Coordinates outside [0,1] simply wrap around
  - A coordinate of [1.5,-2.5] simply becomes [0.5,0.5]
  - Useful for mapping repeatable textures across a large surface, like grass across terrain



Texture Addressing

- Mirror simply inverts a coordinate every other integer range
  - For example 0-1 is normal, 1-2 is mirrored, 2-3 is normal, etc
- Edge Clamp means the coordinate is clamped to [0,1] range and samples the edge texels of the texture
- Border Clamp means all coordinates outside the [0,1] range instead sample a solid constant colour rather than the texture
  - This colour is set in the application





Sampling Textures

- Within a shader we are able to sample texel information at specific coordinates within a texture
  - The coordinate is part of the vertex buffer, and the vertex shader typically just passes through the coordinate to be used in later stages, like the fragment stage
  - Sampling takes addressing modes into account to determine the colour returned
- To be able to sample a texture in a shader we need to have bound a uniform variable for a sampler type
  - sampler1D, sampler2D, and sample3D are all built-in types
- We then call a built-in function, using the sampler and a coordinate, that returns the texel data
  - texture()

Sampling Textures

```
#version 410
in vec4 position;
in vec2 texCoord; // input from the vertex shader that we will pass as a custom output

out vec2 vTexCoord; // output that goes to the next used programmable stage

uniform mat4 projectionViewWorldMatrix;

void main( ) {
	vTexCoord = texCoord; // simply pass through the variable
	gl_Position = projectionViewWorldMatrix * position;
}
```

Vertex Shader

```
#version 410
in vec2 vTexCoord; // input from the last used programmable stage

uniform sampler2D texture; // the bound texture, 2D in this example

out vec4 pixelColour; // output pixel colour

void main( ) {
	// sample the 2D texture, which returns a vec4 with rgba colour data
	pixelColour = texture( texture, vTexCoord );
}
```

Fragment Shader

Sampling Textures

- Just because a texture() call returns a vec4 colour doesn't mean we have to use it as the output or that we can't modify the texture coordinate first!

```
#version 410
in vec2 vTexCoord;
uniform sampler2D texture;

out vec4 pixelColour;

void main( ) {
	// only use the red colour channel from the texture!
	pixelColour = texture( texture, vTexCoord ).rrra;
}
```

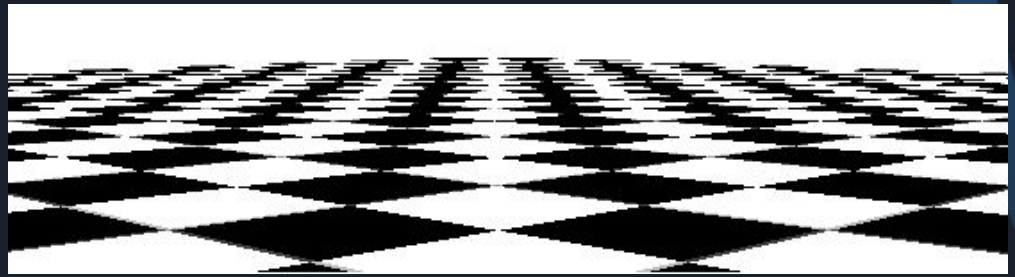
```
#version 410
in vec2 vTexCoord;
uniform sampler2D texture;

out vec4 pixelColour;

void main( ) {
	// invert the texture!
	pixelColour = vec4(1,1,1,1);
	pixelColour.rgb -= texture( texture, vTexCoord ).rgb;
}
```

```
#version 410
in vec2 vTexCoord;
uniform float time;
uniform sampler2D texture;

out vec4 pixelColour;

void main( ) {
	vec2 warpedST = vTexCoord + vec2( sin( time ), 0 ); // warp the S coordinate by a sin wave
	pixelColour = texture( texture, warpedST );
}
```

Texture Filtering

- One thing to bring up is that when sampling a texture the coordinate doesn't always map exactly to a texel
  - The coordinate could be in the corner of a texel, or right on the edge of two texels

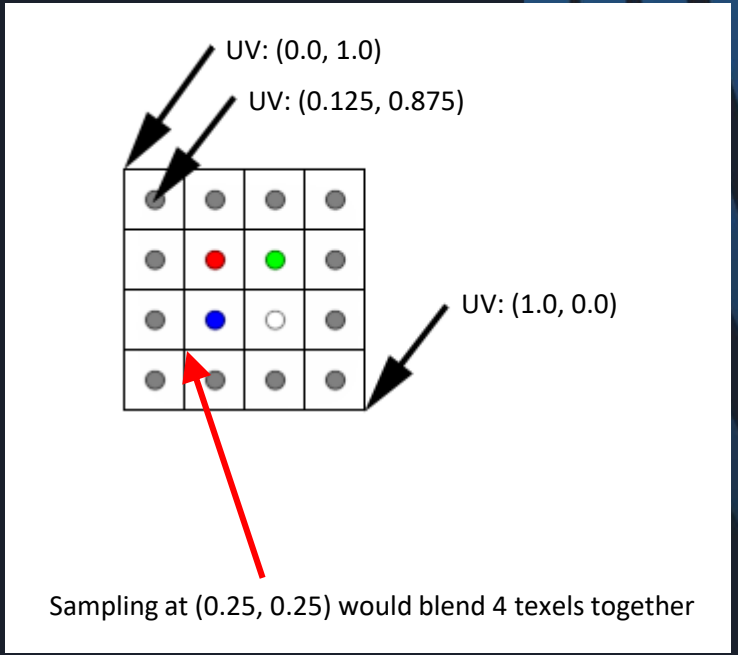
- This can cause visible artefacts when textures are applied to geometry of all shapes and sizes, called aliasing



- Filtering is a method of smoothing the texel colour, blending it with neighbouring texels based on certain settings

Texture Filtering

- A texel is not a little square!
  - The colour is at the centre of the square
  - The edge of the square would be in between two texel colours
- For 3-D models created by artists this isn't an issue as an artist arranges the texture how they want
  - For other meshes, such as 2D polygons, we may need to offset a texture coordinate by half a texel!
- Sampling a texture will apply filtering
  - Sampling at coordinate (0,0) would blend between 4 texels; the first, the last, the last on the first row and the last on the first column!



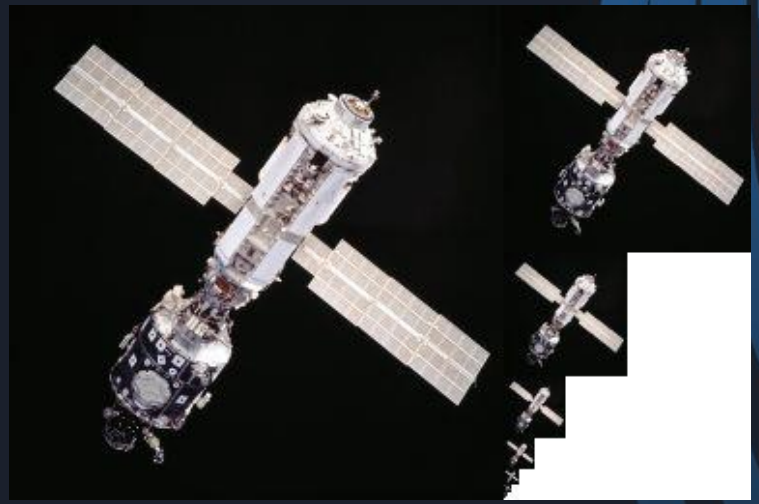
Texture Filtering

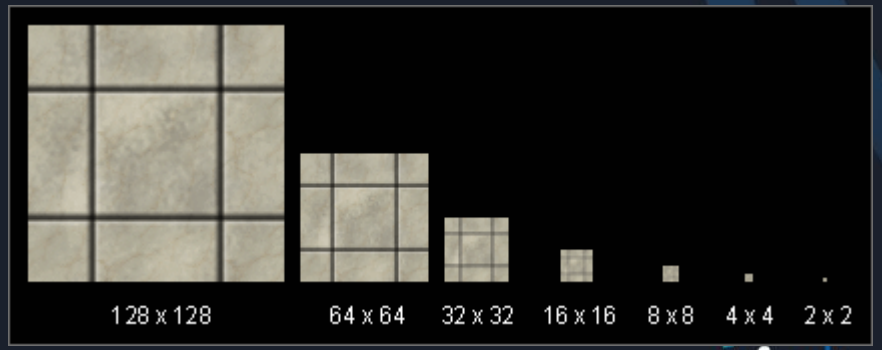
- There are various filtering techniques that can be set in the application
  - In shader code we don't have to worry about the filtering as it is automatically calculated when we call texture()
  - Nearest: simply samples the texel the pixel is on (no smoothing)
  - Linear: samples neighbour texels and performs a weighted blend between them
- Anisotropic filter is an advanced method of filtering that is dependent on hardware
  - Simply performs more samples and blends them
  - Number of samples can be set in the application (2+)

Texture Filtering

Mip-Mapping

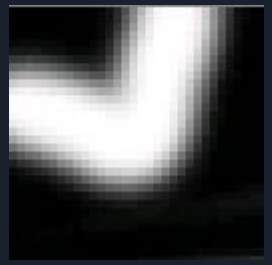
- Mip-Mapping is another form of pre-filtering
  - It creates a set of lower resolution copies of a texture down to 1x1
- When sampling a texture it will try and find a mip layer that matches the pixel size 1-to-1 for the texel
  - This stops it having to filter the texels when sampled at runtime
- Mip-Mapping does increase memory usage about 33%!





Multiple Textures and Texture Blending

- Lastly you are also able to bind multiple textures to a shader and achieve a myriad range of effects!


X

=


```
#version 410
in vec2 vTexCoord;

uniform sampler2D grassTexture;
uniform sampler2D tintTexture;

out vec4 pixelColour;

void main( ) {
      pixelColour = texture( grassTxture, vTexCoord ) * texture( tintTxture, vTexCoord );
}
```

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Summary

- Fragment Shaders define the final pixel colour output from the Render Pipeline
  - There are many great effects that can be achieved through the Fragment Shader
- Textures contain texel data that we can sample from within our shaders
  - Modern hardware allows texture sampling at any programmable
    stage in the pipeline

Further Reading

- Akenine-Möller, T, Haines, E, 2008, *Real-Time Rendering*, 3rd Edition, A.K. Peters

- Wolff, D, 2013, *OpenGL 4 Shading Language Cookbook*, 2nd Edition, PACKT Publishing