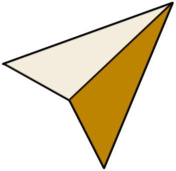


COMP4610/COMP6461

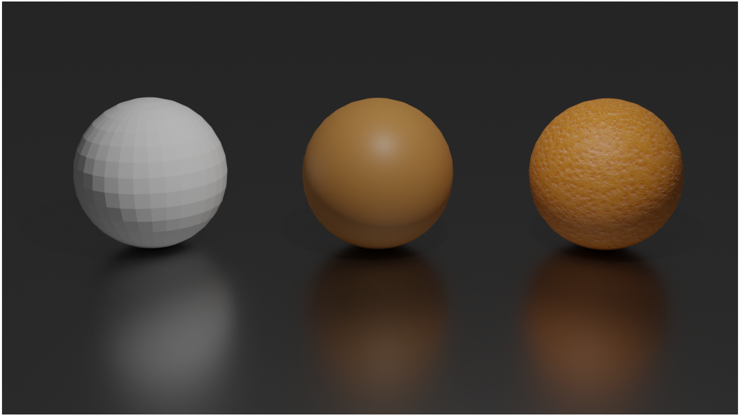
Week 7 - Textures and Shaders

<Print version>

Admin

Textures

What is a Texture?

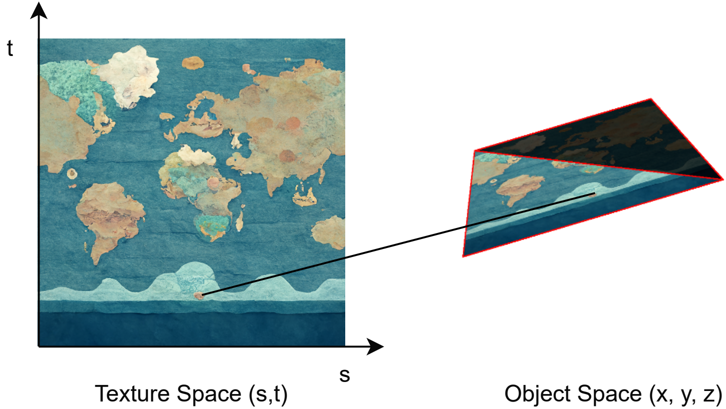


Textures allow us to add fine detail to 3D objects.

What is a Texture?

- Textures can be defined either by
  - **image textures**: an array of colours.
  - **procedural textures** a function which maps texture coordinates to colours.
- Texture can be 1D, 2D or 3D.
- Texture coordinates range from 0.0 to 1.0
- Color can be grayscale, RGB, RGBA, and others.
- Texture coordinates are usually named u,v,w, but OpenGL uses s, t, r (this is because w was already used...).
- To use 2D Textures on 3D objects some kind of mapping is required.
- We refer to the ‘pixels’ of the texture as **texels**.

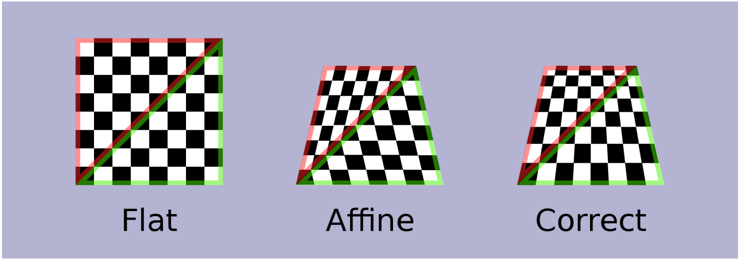
UV Space



X, Y, Z object coordinates must be mapped to S,T (UV) coordinates.

Affine Mapping

Linear interpolation of texture coordinates over pixels produces incorrect results. Therefore perspective correction needs to be applied. This involves multiplying by $\frac{1}{z}$ (hyperbolic interpolation), which was not practical on early hardware. Blinn outlines how to do this using homogeneous coordinates in his 1992 paper [1].



Credit https://en.wikipedia.org/wiki/Texture_mapping

GLSL still supports the affine interpolation via

```
noperspective out vec4 uvCoord;
```

Textures in OpenGL

During initialization

To render using textures you first need to **generate** a texture id (handle), then **load** it, and **enable** texturing.

```
gl.glGenTextures(1, texID);
gl.glBindTexture(gl.GL_TEXTURE_2D, texID);
gl.glTexImage2D(gl.GL_TEXTURE_2D, 0,
    gl.GL_RGB, width, height, 0,
    gl.GL_RGB, gl.GL_UNSIGNED_BYTE,
    ByteBuffer.wrap(textureData));
gl.glEnable(gl.GL_TEXTURE_2D);
```

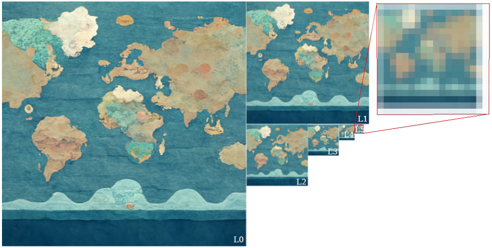
When drawing

You need to **bind** the texture, then set UV coords at the vertices.

```
gl.glBindTexture(gl.GL_TEXTURE_2D, texID);
gl.glBegin(gl.GL_POLYGON);
gl.glTexCoord2d(0.0, 0.0);
gl.glVertex3d(0.0, 0.0, 0.0);
...
gl.glEnd();
```

JOGL provides a class TextureIO that does some of this work for us.

MIP Maps



- Textures have aliasing artifacts when zoomed out too far.
- Solution: Average over texels within a region... too slow.
- Better solution: Create a reduced resolution texture with high frequencies filtered out.
- Sampling a single texel from the level-5 mipmap is effectively averaging over 32 samples in the original texture.
- Can increase performance due to localised memory access patterns.

Texture Filtering

In OpenGL you can configure the filtering mode for both **magnification** and **minification**.

OpenGL allows configuration of interpolation between texels, as well as interpolation between mipmaps via

```
gl.glTexParameteri( GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MIN_FILTER, SETTING);
```

where SETTING is taken from the following table.

OpenGL	Common Name
GL_NEAREST	No filtering (without mipmaps)
GL_NEAREST_MIPMAP_NEAREST	No filtering (with mipmaps)
GL_LINEAR	Bilinear (without mipmaps)
GL_LINEAR_MIPMAP_NEAREST	Bilinear (with mipmaps)
GL_LINEAR_MIPMAP_LINEAR	Trilinear

Wrapping

In **OpenGL** we can control what happens texture coordanates outside of [0, 1] are given, using

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, *);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, *);
```

GL_REPEAT

GL_MIRRORED_REPEAT

GL_CLAMP_TO_EDGE

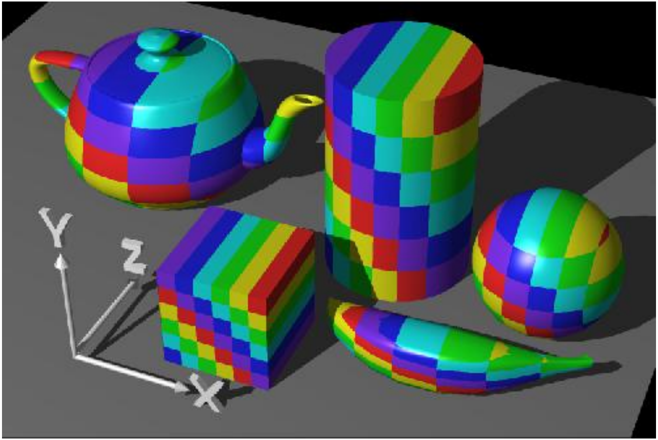
GL_CLAMP_TO_BORDER

Credit https://open.gl/textures

UV Mapping

Planar Mapping

Simplest method. Just throw away one of the dimensions.



Credit: https://education.siggraph.org/static/HyperGraph/

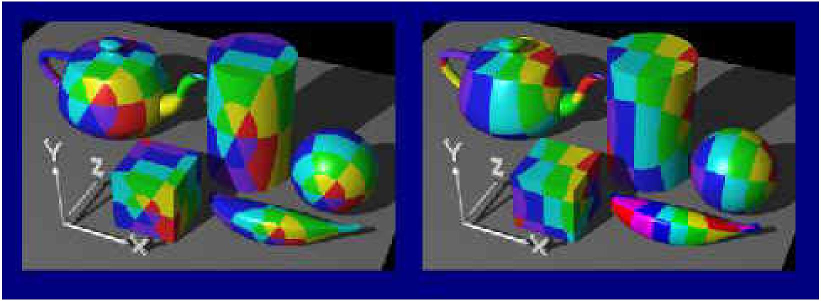
Spherical Mapping

Calculate longitude and latitude (or just use the normals...)

$$r = \sqrt{(x^2 + y^2 + z^2)} \tag{1}$$

$$u = \text{atan2}(y, x)/(2\pi) + 0.5 \tag{2}$$

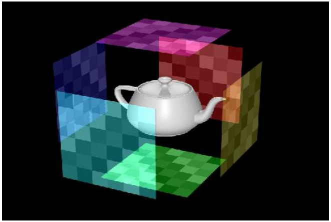
$$v = \text{asin}(z/r)/\pi + 0.5 \tag{3}$$

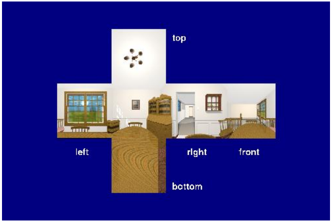


Credit: https://education.siggraph.org/static/HyperGraph/

Cube Maps

Use 6 planar maps. Quite useful for reflections, and skyboxes.

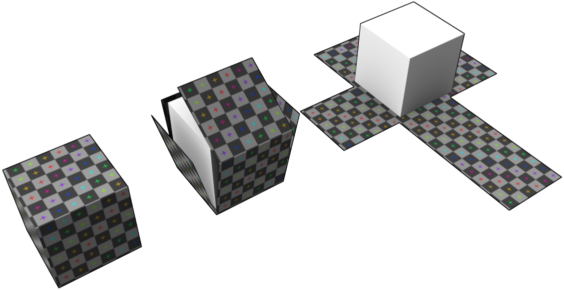




Credit: `https://education.siggraph.org/static/HyperGraph/`

UV Unwrapping

Probably the most common method. Often requires human input to define the seams. Might not make best use of the texture space.



Credit: https://en.wikipedia.org/wiki/UV_mapping

Shaders

[Why Shaders?]

- Early on hardware for 3D graphics was very expensive and limited.
- Features were typically implemented in hardware, and gave only limited control over the lighting calculations.
- As more and more features were added, it became efficient for hardware to implement very general operations in hardware, then ‘build’ the predefined lighting algorithms out of these basic operations, in either firmware or at the driver level.
- It became increasingly obvious that giving access to these low-level operations would be a good idea. However, how could this be done without locking into specific hardware design choices?
- **Shaders** were the solution to this problem.

OpenGL Rendering Pipeline

- OpenGL provides a processing pipeline to produce real time 3D rendered images.
- OpenGL 1.1 uses a fixed pipeline. The GPU cards effectively implemented this in fixed hardware. To provide more rendering flexibility the computational intensive parts of the pipeline became programmable, this was done via **shaders**.

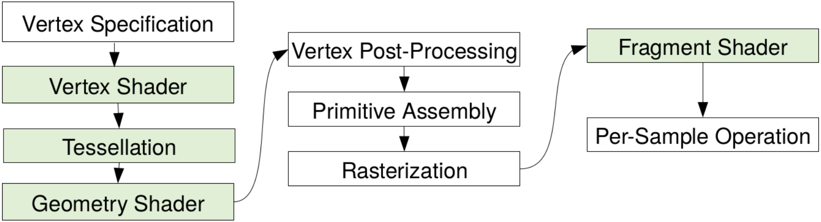


Diagram based on information from
https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

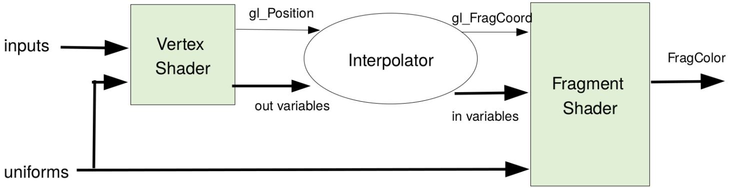
Vertex and Fragment Shaders

Shaders are short bits of ‘c like’ code that have mostly defined inputs and outputs that can be run on the GPU in parallel. The two main types of shaders are:

- **Vertex** - which transform individual vertices which make up the scene to be renders. For the most part this involves applying the model-view and perspective matrix transformations. However it often will also involve transformations of colours at these vertices and also texture coordinates.
- **Fragment** - each triangle is divided up into fragments. Which are basically potential pixels. The fragment shader will work out the colour of that potential pixel, this will involve lighting calculations and/or looking up texture values.

Basic Shader Processing

The GLSL (OpenGL Shading Language) has evolved over the years. Initially it had more defined and constrained inputs and outputs between shaders. Newer versions moving to more general and flexible connections.



This diagram shows how information is provided to the vertex and fragment shaders for GLSL 3.3.

Example Vertex Shader

The below vertex shader transforms points based on provided model-view and perspective transformation matricies. It just passes through to colour value for each vertex.

```
#version 330 core
in vec3 aPos;
in vec3 color;
uniform mat4 mvMat, pMat;
out vec4 vertex_color;
vec4 mc;
void main() {
    vertex_color = vec4(color,1.0);
    mc = vec4(aPos.x,aPos.y,aPos.z,1.0);
    gl_Position = (pMat * mvMat) * mc;
}
```

Example Fragment Shader

The below fragment shader uses the interpolated colour values to determine the final colour of the fragment.

```
#version 330 core
out vec4 FragColor;
in vec4 vertex_color;
void main() {
    FragColor = vec4(vertex_color.x,vertex_color.y,vertex_color.z,1.0);
}
```

Handling Uniforms

Uniforms need to be passed to OpenGL via **gl.glUniform...**

```
int mvMatrixID = gl.glGetUniformLocation(
    shaderProgram, "mvMat");
gl.glUniformMatrix4fv(mvMatrixID, 1, false
    , matrix.glGetMvMatrixf());
```

GLSL

GLSL is for the most part standard c code. The 2 key exception are that pointers and recursion are not permitted. Although there are a number of extra but built in functions and operators that simplify graphics calculations. These include:

- Vector and matrix types and their associated operators (multiplication, inverse, determinate, transpose)
- Geometric functions (length, distance, dot, cross, normalize, reflect, refract).
- Texture lookup functions.
- Handy maths functions (max, min, clamp).
- Trigonometry and exponential functions.

Getting it Going

To get a shader approach working in OpenGL you need to:

- Load, compile, link the shader program,
- setup the buffers for providing the data for the shader,
- setup any uniforms you are using, setting there values before you draw,
- set the input attributes to point to buffers you are using, and
- do the drawing.

In my experience this is usually about 120 lines of code, and if you get it 95% correct it won't work at all. For this reason we have provided a very basic working example as part of Lab-3.

Shaders and Textures

Using Textures with Shaders

We can access our texture in the shader using (note we usually modulate the texture color with the vertex color)

```
in vec3 VertexColor;
in vec3 TexCoord
uniform sampler2D ourTexture;
void main()
{
    FragColor = texture(ourTexture, TexCoord) * vec4(VertexColor, 1.0);
}
```

By default this uses the first **texture unit**, GL_TEXTURE0, we can bind textures to other units using

```
glActiveTexture(GL_TEXTURE2);
glBindTexture(GL_TEXTURE_2D, ourTextureID);
```

The number of available texture units is hardware dependant, but for OpenGL 4.0 it will always be ≥ 16. If UV coordinates are not shared between textures use **glMultiTexCoord2d**.

References

[1] James F Blinn. "Hyperbolic interpolation". In: IEEE Computer Graphics and Applications 12.4 (1992), pp. 89–94.

Aside: AI Generated Graphics