Texturing

Computer Graphics - SET08116

EDINBURGH NAPIER UNIVERSITY

Outline

What is Texturing?

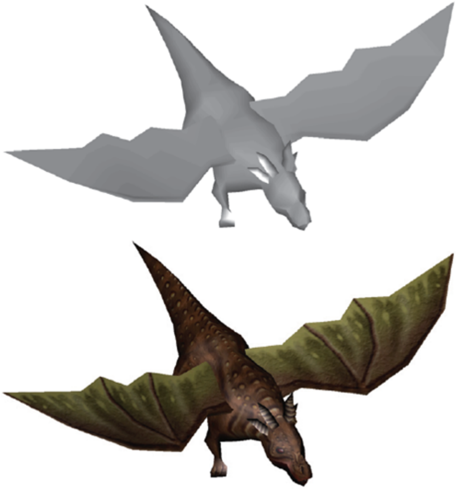
Texturing is the process of applying image data to a surface in our rendered scene.
For example, we can create a wall using a simple plane geometry. We can change the appearance of this wall just by changing the texture:

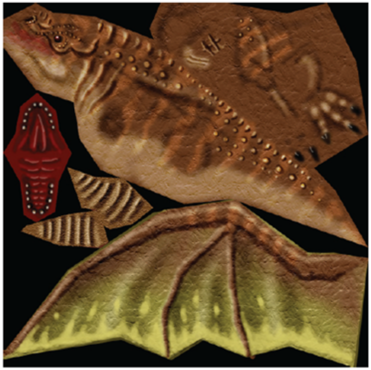
- If we use an image of a brick wall, we can make the wall look like a brick wall
- If we use an image of a wooden wall, we can make the wall look like a wooden wall

Texturing can be considered as the third main part of what we consider the core technologies of 3D rendering:

- Geometry
- Lighting
- Texturing

Example





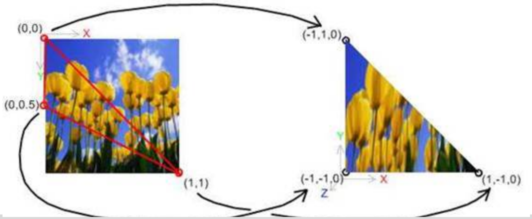
How Do We Texture?

So far we have just been dealing with position data for our vertexes.

We can also attach texture coordinate information to our vertexes. These are 2D coordinates.

These coordinates are used to create triangles that can be attached to the geometry.

The fragment shader uses these coordinates to determine pixel colour.



Paper Wrapping

Texturing can be though of as wrapping the texture around the defined geometry.

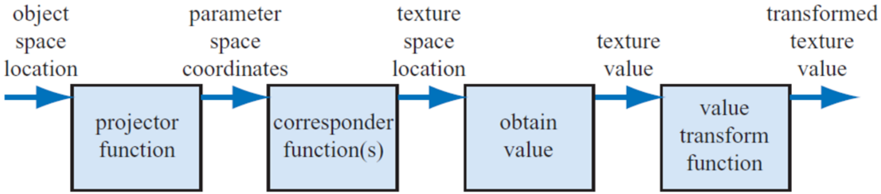
- Not strictly true - more mapping the texture to the geometry

Each defined piece of geometry has a part of the image stuck to it

The GPU can easily take the required image and place it on the geometry

- However, this operation is expensive relative to transfomations done in the vertex shader

Texturing Pipeline

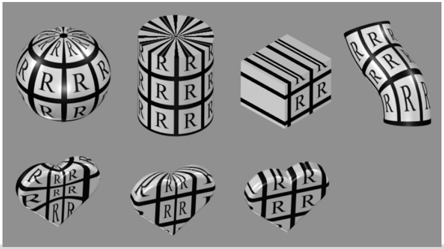


Projector Function

The projector function determines how the texture should be mapped to the geometry. It therefore takes into account the 3D position.

There are four types of projector function:

- Spherical
- Cylindrical
- Planar
- Natural



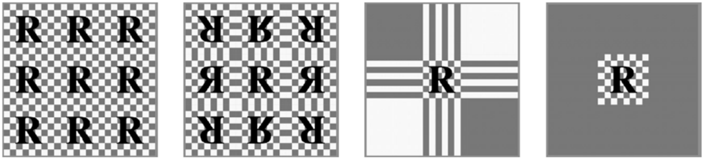
Corresponder Function

The corresponder function takes generated values from the projector function and applies them to a texture space.

There are a few options here, such as sub-image selection, transformation, etc.

Wrapping is another option:

- Repeat
- Mirror
- Clamp
- Border



Texture Values

Once we have passed through a projector function, and *n* corresponder functions, we are ready to map the texture to the geometry.

This is known as determining the texel information for the gometry. The texel is the image data to apply.

Typically, this works out as an RGBA value that is to be applied to a specific pixel

- Although the fragment shader may modify the colour further.

Triangle Sections

Basically, when using images to texture objects we are cutting out triangles and applying these to our triangle geometry.

We wont usually have to worry about where texture coordinates come from for basic work

- This is the artist's job

We are more likely to work with texture coordinates for advanced techniques:

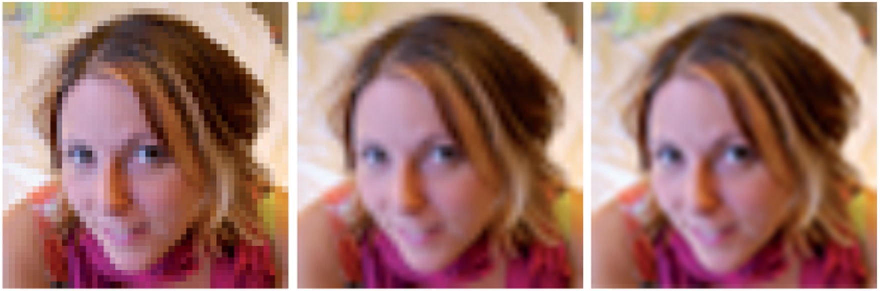
- Blending
- Alpha mapping
- Bump mapping
- Post-process
- etc.

Magnification

Sometimes our textures are too low a resolution for our render.

- For example, see Oblivion

In these circumstances, the GPU has to magnify our image to compensate:



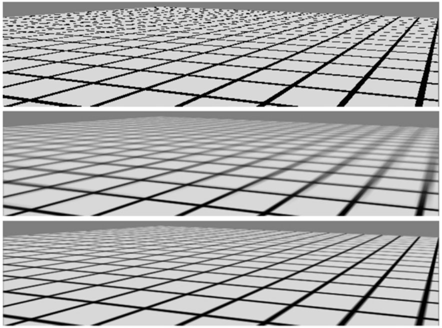
Minification

It is far more likely that we will want to perform minification on our textures due to distance.

- We aim to have high resolution textures for near objects

There are four core techniques to improve distant texture quality:

- Point sampling
- Mipmapping
- Summed area tables
- Anisotropic filtering



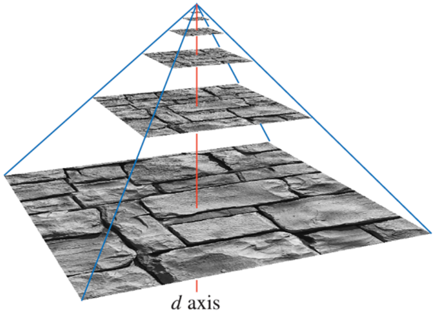
Mipmaps

Mipmapping is an important concept in texturing.

An image has a number of smaller images created from it:

- Divide by two
- This is why the GPU expects power of 2 texture sizes

Based on the distance to the pixel, a different image is used for texture mapping



Using Texture Coordinates

Using texture coordinates is easy

- For each vertex declaration, declare a glTexCoord2f
- Same as working with colour

This data can be used directly by OpenGL

- Textures using the currently mapped texture

Or can be done with a shader

```
glBegin(GL_QUADS);
glTexCoord2f(0.0, 0.0);
glVertex3f(0.0, 0.0, 0.0);
glTexCoord2f(1.0, 0.0);
glVertex3f(10.0, 0.0, 0.0);
glTexCoord2f(1.0, 1.0);
glVertex3f(10.0, 10.0, 0.0);
glTexCoord2f(0.0, 1.0);
glVertex3f(0.0, 10.0, 0.0);
glEnd();
```

Using Textures in Shaders

Texture mapping can be performed directly in the fragment shader

We use the texture2D function to get a colour from the texture

- sampler2D

Samplers are how we access texture data in GLSL

```
uniform sampler2D tex;

in vec2 texCoord;
out vec4 colour;

void main()
{
  vec4 colour =
        texture2D(tex,
                  texCoord);
}
```

Blending Textures

As texture data eventually just becomes a colour value, we can blend textures together easily.

A simple algorithm for this is:

- Take in n textures
- Apply a blend weight for each vertex for each of the n textures
- Add all the colours together to determine the final pixel colour

Example Shader

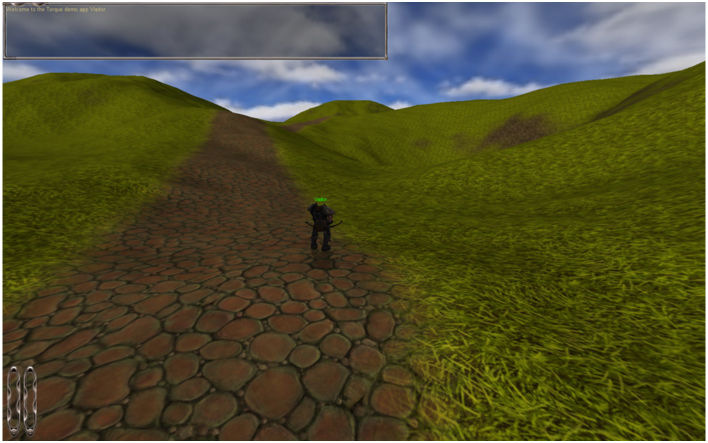
```
uniform sampler2D tex1;
uniform sampler2D tex2;

in vec2 texCoord;
in float blend1;
in float blend2;

out vec4 colour;

void main()
{
  vec4 col1 = blend1 * texture2D(tex1, texCoord);
  vec4 col2 = blend2 * texture2D(tex2, texCoord);
  colour = col1 + col2;
}
```

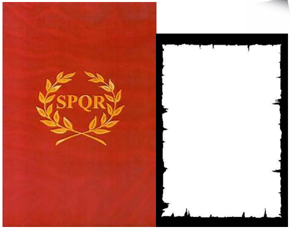
Example Output

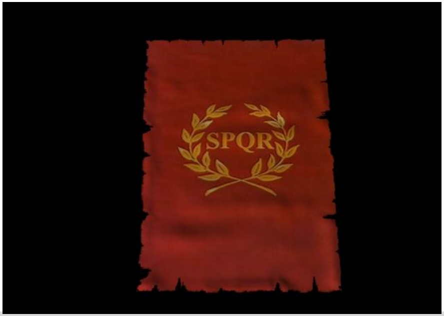


Alpha Mapping

We can take the blending concept further and apply an alpha map

For the second texture, any point that has no alpha value (black) is transparent in the final render





Light Maps

The blending idea can be taken even further and can be used to apply light maps to areas.

Instead of determining lighting colour from sources, use a static texture for much of the light

If light direction is fixed, and geometry is fixed, then ambient and diffuse light will be the same

- Why calculate per vertex?

Provides a more per-pixel look to the final render, but in a cheaper manner than standard phong shading

Summary

We have now covered the three core parts of 3D rendering

- Geometry
- Lighting
- Texturing

Texturing is the process of applying images to our geometry

- Provides better detail

Texturing can be used to create better effects

- Blending, etc.
- Bump mapping and post-process (still to come)

Recommended Reading

Interactive Computer Graphics, chapter 7

Real-Time Rendering, chapter 6