



COMP220: Graphics & Simulation

5: Textures and models

Learning outcomes

- ▶ **Explain** how a 2D texture image can be wrapped onto a 3D model
- ▶ **Explain** how a complex 3D model is represented in memory
- ▶ **Write** programs which draw textured meshes to the screen

Basic texture mapping



Loading textures from a file

- ▶ The **SDL_Image library** lets us load images from JPG, PNG, BMP etc.

Loading textures from a file

- ▶ The **SDL_Image library** lets us load images from JPG, PNG, BMP etc.
- ▶ Steps:

Loading textures from a file

- ▶ The **SDL_Image library** lets us load images from JPG, PNG, BMP etc.
- ▶ Steps:
 - ▶ Load the image with `IMG_Load`

Loading textures from a file

- ▶ The **SDL_Image library** lets us load images from JPG, PNG, BMP etc.
- ▶ Steps:
 - ▶ Load the image with `IMG_Load`
 - ▶ Create a texture with `glGenTextures`

Loading textures from a file

- ▶ The **SDL_Image library** lets us load images from JPG, PNG, BMP etc.
- ▶ Steps:
 - ▶ Load the image with `IMG_Load`
 - ▶ Create a texture with `glGenTextures`
 - ▶ Bind the texture with `glBindTexture`

Loading textures from a file

- ▶ The **SDL_Image library** lets us load images from JPG, PNG, BMP etc.
- ▶ Steps:
 - ▶ Load the image with `IMG_Load`
 - ▶ Create a texture with `glGenTextures`
 - ▶ Bind the texture with `glBindTexture`
 - ▶ Load the pixel data into the new texture with `glTexImage2D`

Loading textures from a file

- ▶ The **SDL_Image library** lets us load images from JPG, PNG, BMP etc.
- ▶ Steps:
 - ▶ Load the image with `IMG_Load`
 - ▶ Create a texture with `glGenTextures`
 - ▶ Bind the texture with `glBindTexture`
 - ▶ Load the pixel data into the new texture with `glTexImage2D`
 - ▶ Set the texture filtering modes with `glTexParameter` (more on this later)

Texture coordinates

- ▶ We use **UV coordinates** to refer to points in a texture

Texture coordinates

- ▶ We use **UV coordinates** to refer to points in a texture
- ▶ u axis is horizontal and ranges from 0 (left) to 1 (right)

Texture coordinates

- ▶ We use **UV coordinates** to refer to points in a texture
- ▶ u axis is horizontal and ranges from 0 (left) to 1 (right)
- ▶ v axis is vertical and ranges from 0 (bottom) to 1 (top)

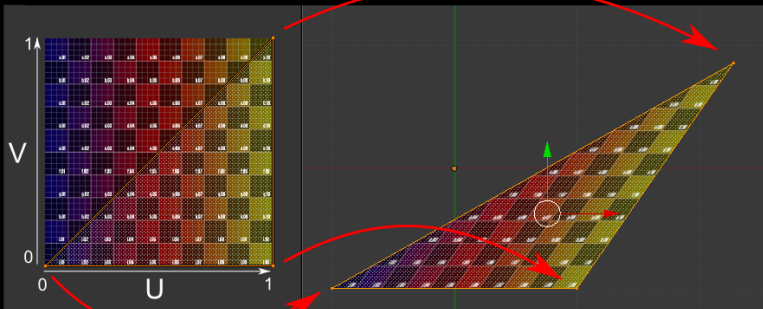
Texture coordinates

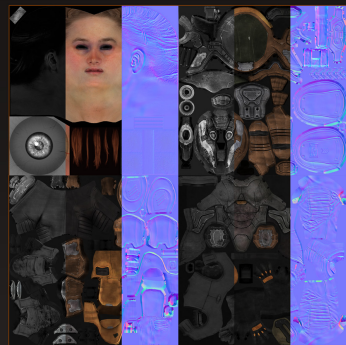
- ▶ We use **UV coordinates** to refer to points in a texture
- ▶ u axis is horizontal and ranges from 0 (left) to 1 (right)
- ▶ v axis is vertical and ranges from 0 (bottom) to 1 (top)
- ▶ (So really just another name for xy coordinates in texture space)

Texture coordinates

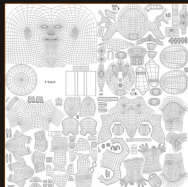
- ▶ We use **UV coordinates** to refer to points in a texture
- ▶ u axis is horizontal and ranges from 0 (left) to 1 (right)
- ▶ v axis is vertical and ranges from 0 (bottom) to 1 (top)
- ▶ (So really just another name for xy coordinates in texture space)
- ▶ Basic idea of texture mapping: give each vertex a uv coordinate, and interpolate across the triangle

UV coordinates





Specular/Diffuse/Normal



UV layout/Masking maps

Textures in GLSL

Fragment shader:

```
in vec2 textureCoords;  
uniform sampler2D textureSampler;  
  
void main()  
{  
    fragmentColour = texture(textureSampler, ←  
        textureCoords);  
}
```

Texture filtering

Texture filtering

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

- **Linear interpolation** (`GL_LINEAR`) smooths between pixels

Texture filtering

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

- ▶ **Linear interpolation** (`GL_LINEAR`) smooths between pixels
- ▶ **Nearest neighbour** (`GL_NEAREST`) is pixelated but may be slightly faster

Texture filtering

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

- ▶ **Linear interpolation** (`GL_LINEAR`) smooths between pixels
- ▶ **Nearest neighbour** (`GL_NEAREST`) is pixelated but may be slightly faster
- ▶ **Anisotropic filtering** improves the quality of linear interpolation but is slower

Texture filtering

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

- ▶ **Linear interpolation** (`GL_LINEAR`) smooths between pixels
- ▶ **Nearest neighbour** (`GL_NEAREST`) is pixelated but may be slightly faster
- ▶ **Anisotropic filtering** improves the quality of linear interpolation but is slower
- ▶ **Mip-mapping** pre-calculates scaled down versions of the texture — improves quality but costs memory

Texture dimensions

- ▶ In the old days, OpenGL required textures to have **power of two** dimensions

Texture dimensions

- ▶ In the old days, OpenGL required textures to have **power of two** dimensions
 - ▶ 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, ...

Texture dimensions

- ▶ In the old days, OpenGL required textures to have **power of two** dimensions
 - ▶ 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, ...
- ▶ Nowadays **non-power of two (NPOT)** textures are widely supported

Texture dimensions

- ▶ In the old days, OpenGL required textures to have **power of two** dimensions
 - ▶ 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, ...
- ▶ Nowadays **non-power of two (NPOT)** textures are widely supported
- ▶ Still better to stick to powers of two as some things work better (e.g. mipmapping)

Texture dimensions

- ▶ In the old days, OpenGL required textures to have **power of two** dimensions
 - ▶ 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, ...
- ▶ Nowadays **non-power of two (NPOT)** textures are widely supported
- ▶ Still better to stick to powers of two as some things work better (e.g. mipmapping)
- ▶ NB: **rectangular** textures are fine, but **square** textures make UV coordinates saner

Texture Mapping Example

Transparency



Alpha

- ▶ We are used to working with colours in **RGB** space

Alpha

- ▶ We are used to working with colours in **RGB** space
- ▶ We can also work in **RGBA** space, where A = alpha = transparency

Alpha

- ▶ We are used to working with colours in **RGB** space
- ▶ We can also work in **RGBA** space, where A = alpha = transparency
- ▶ $A = 0 \implies$ fully transparent

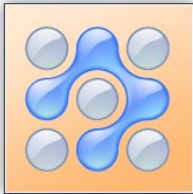
Alpha

- ▶ We are used to working with colours in **RGB** space
- ▶ We can also work in **RGBA** space, where A = alpha = transparency
- ▶ $A = 0 \implies$ fully transparent
- ▶ $A = 1$ (or $A = 255$) \implies fully opaque

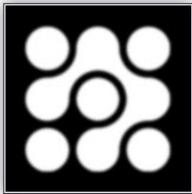
Alpha

- ▶ We are used to working with colours in **RGB** space
- ▶ We can also work in **RGBA** space, where A = alpha = transparency
- ▶ $A = 0 \implies$ fully transparent
- ▶ $A = 1$ (or $A = 255$) \implies fully opaque

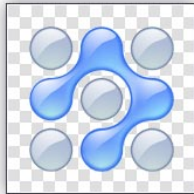
Use of Alpha Channel to create Transparent Image



Original Image
RGB - 24 bpp



Alpha Channel
A - 8 bpp



Transparent Image
RGBA - 32 bpp

Alpha in OpenGL

- ▶ Use `vec4` instead of `vec3` for colours

Alpha in OpenGL

- ▶ Use **vec4** instead of **vec3** for colours
- ▶ Textures can have an **alpha channel**

Alpha in OpenGL

- ▶ Use **vec4** instead of **vec3** for colours
- ▶ Textures can have an **alpha channel**
 - ▶ PNG supports alpha channels, JPG and BMP do not

Alpha in OpenGL

- ▶ Use **vec4** instead of **vec3** for colours
- ▶ Textures can have an **alpha channel**
 - ▶ PNG supports alpha channels, JPG and BMP do not
- ▶ Need to enable **alpha blending**

Alpha in OpenGL

- ▶ Use **vec4** instead of **vec3** for colours
- ▶ Textures can have an **alpha channel**
 - ▶ PNG supports alpha channels, JPG and BMP do not
- ▶ Need to enable **alpha blending**

```
glEnable(GL_BLEND);  
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

- ▶ Other values can be passed to `glBlendFunc` for special effects (e.g. **additive blending** is often used for particle effects simulating light, fire, explosions etc.)

Transparency and depth testing

- ▶ Recall we are using **depth testing**

Transparency and depth testing

- ▶ Recall we are using **depth testing**
 - ▶ Each fragment on screen remembers its **depth** (distance from the camera)

Transparency and depth testing

- ▶ Recall we are using **depth testing**
 - ▶ Each fragment on screen remembers its **depth** (distance from the camera)
 - ▶ A new fragment is drawn **only if** its depth value is **less** than the current depth value

Transparency and depth testing

- ▶ Recall we are using **depth testing**
 - ▶ Each fragment on screen remembers its **depth** (distance from the camera)
 - ▶ A new fragment is drawn **only if** its depth value is **less** than the current depth value
 - ▶ I.e. don't draw objects that should be behind something that was already drawn

Transparency and depth testing

- ▶ Recall we are using **depth testing**
 - ▶ Each fragment on screen remembers its **depth** (distance from the camera)
 - ▶ A new fragment is drawn **only if** its depth value is **less** than the current depth value
 - ▶ I.e. don't draw objects that should be behind something that was already drawn
- ▶ But if the object in front is (semi-)transparent, we want to see the object behind it!

Transparency and depth testing

- ▶ Recall we are using **depth testing**
 - ▶ Each fragment on screen remembers its **depth** (distance from the camera)
 - ▶ A new fragment is drawn **only if** its depth value is **less** than the current depth value
 - ▶ I.e. don't draw objects that should be behind something that was already drawn
- ▶ But if the object in front is (semi-)transparent, we want to see the object behind it!
- ▶ Solution: draw semi-transparent objects **after** opaque objects, and in **back to front** order

Transparency and depth testing

- ▶ Recall we are using **depth testing**
 - ▶ Each fragment on screen remembers its **depth** (distance from the camera)
 - ▶ A new fragment is drawn **only if** its depth value is **less** than the current depth value
 - ▶ I.e. don't draw objects that should be behind something that was already drawn
- ▶ But if the object in front is (semi-)transparent, we want to see the object behind it!
- ▶ Solution: draw semi-transparent objects **after** opaque objects, and in **back to front** order
- ▶ Further discussion: <http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-10-transparency/>

Texturing Example

More meshes



Mesh Formats

- ▶ Typically we invent our own mesh format (see Doom's MD6, Valve's smd formats)

Mesh Formats

- ▶ Typically we invent our own mesh format (see Doom's MD6, Valve's smd formats)
- ▶ These formats are optimised for realtime rendering and are very efficient

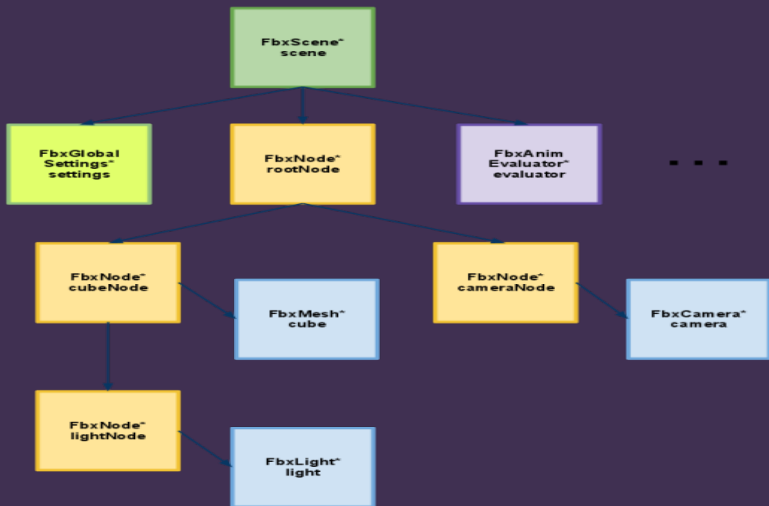
Mesh Formats

- ▶ Typically we invent our own mesh format (see Doom's MD6, Valve's smd formats)
- ▶ These formats are optimised for realtime rendering and are very efficient
- ▶ Usually developers write exporters for Maya or 3DSMax to support their format

Mesh Formats

- ▶ Typically we invent our own mesh format (see Doom's MD6, Valve's smd formats)
- ▶ These formats are optimised for realtime rendering and are very efficient
- ▶ Usually developers write exporters for Maya or 3DSMax to support their format
- ▶ We are going to use FBX as our model format, this known as an 'interchange' format

Quick Tour of the FBX Format



Open Asset Import Library

- ▶ There is an FBX SDK published by Autodesk, this can be used to load FBX files

Open Asset Import Library

- ▶ There is an FBX SDK published by Autodesk, this can be used to load FBX files
- ▶ We will use Asset Import Library to load FBX files

Open Asset Import Library

- ▶ There is an FBX SDK published by Autodesk, this can be used to load FBX files
- ▶ We will use Asset Import Library to load FBX files
- ▶ This allows us to support multiple file formats include

Open Asset Import Library

- ▶ There is an FBX SDK published by Autodesk, this can be used to load FBX files
- ▶ We will use Asset Import Library to load FBX files
- ▶ This allows us to support multiple file formats include
 - ▶ FBX

Open Asset Import Library

- ▶ There is an FBX SDK published by Autodesk, this can be used to load FBX files
- ▶ We will use Asset Import Library to load FBX files
- ▶ This allows us to support multiple file formats include
 - ▶ FBX
 - ▶ OBJ

Open Asset Import Library

- ▶ There is an FBX SDK published by Autodesk, this can be used to load FBX files
- ▶ We will use Asset Import Library to load FBX files
- ▶ This allows us to support multiple file formats include
 - ▶ FBX
 - ▶ OBJ
 - ▶ DAE (aka Collada)

Open Asset Import Library

- ▶ There is an FBX SDK published by Autodesk, this can be used to load FBX files
- ▶ We will use Asset Import Library to load FBX files
- ▶ This allows us to support multiple file formats include
 - ▶ FBX
 - ▶ OBJ
 - ▶ DAE (aka Collada)
 - ▶ MD5 (DOOM3)

Open Asset Import Library

- ▶ There is an FBX SDK published by Autodesk, this can be used to load FBX files
- ▶ We will use Asset Import Library to load FBX files
- ▶ This allows us to support multiple file formats include
 - ▶ FBX
 - ▶ OBJ
 - ▶ DAE (aka Collada)
 - ▶ MD5 (DOOM3)
 - ▶ SMD (Half Life 2, Portal etc)

Open Asset Import Library Example

Exercises



Exercise 1 - Texturing

- ▶ Load in a image using SDL Image
- ▶ Copy this image into a OpenGL Texture
- ▶ Add Texture Coordinates to your Cube or Square
- ▶ Map this texture onto the Cube or Square
- ▶ Finally change the texture to a transparent texture

Exercise 2 - Model Loading

- ▶ Create the following NFF models and load each one to the screen
 - ▶ Tetrahedron
 - ▶ Cube
 - ▶ Sphere
 - ▶ Cylinder
- ▶ `http://assimp.sourceforge.net/howtoBasicShapes.html`
- ▶ `https://github.com/assimp/assimp/tree/master/test/models/NFF/NFF`

Exercise 3 - More Complex Scene

- ▶ Create a GameObject class which contains the following as member variables
 - ▶ Vertex Buffer
 - ▶ Element Buffer
 - ▶ Vertex Array Object
 - ▶ Position, Scale, Rotation Vectors
 - ▶ Position, Scale, Rotation, Model Matrices
 - ▶ Open GL Texture
 - ▶ Number of vertices and Indices
- ▶ Add in functions to initialise and get each of these values
- ▶ Add in functions to update (calculate the model matrix) and render
- ▶ Create an instance of this Game Object and display it on the screen