

# COMP3201 – Computer Graphics

## Module 3: Realism and Performance

### 3.4 Texturing

#### 3.4.1 Introduction

At a basic level there are two ways to increase the details (and hence the realism) in a scene. The first is to increase the polygon count by modelling finer details. The second is to use texture mapping. Put simply, texture mapping is the process of sticking an image onto a polygon. This can reduce the scene complexity as one texture could replace hundreds of polygons. This is easily demonstrated by comparing the process of drawing a tree firstly by constructing every leaf as a polygon and alternatively by pasting a photo of a tree onto a rectangular polygon.

*Textures* are simply arrays of colour data, i.e., an image. Each individual component in the array is called a *texel*. Depending on the transformations applied to the underlying polygon, one texel could be mapped to many fragments (proto-pixels) or many texels could be mapped to one fragment. This means that texture calculations can be computationally expensive. Because of this, most graphics subsystems implement hardware support for texturing.

In order to map the required piece of the texture to a given vertex the programmer specifies what are called *texture coordinates*. That is, the texture coordinates are simply an index into the texture.

Textures can be 1, 2 or 3 dimensional. In this course we will only examine 2 dimensional textures.

To use texture mapping (in its simplest form) in an OpenGL application the following steps are followed

1. Specify the image to be used for the texture. (Section 3.4.2)
2. Specify the functions for mapping the texture onto polygons. (Section 3.4.3)
3. Enable texture mapping by `glEnable( GL_TEXTURE_2D );`
4. Draw the scene specifying texture coordinates along with the geometry. (Section 3.4.4).

Note that for interpreting the example that follows, the first row of the texture reads

R,	G,	B,	A,	R,	G,	B,	A,
R,	G,	B,	A,	R,	G,	B,	A,

with values “green, white, white, white”.

## Example

```
/* tex1.c */

#include <GL/glut.h> /* includes gl.h */

/* The image for the texture */
static GLubyte pImage[] = {
/* bottom left – row 0 */
0x00, 0xff, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
/* row 1 */
0xff, 0x00, 0x00, 0x00, 0xff, 0x00, 0x00, 0x00,
0xff, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
/* row 2 */
0xff, 0x00, 0x00, 0x00, 0xff, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
/* row 3 */
0xff, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
/* top right */
};

GLvoid display( GLvoid )
{
    glClear( GL_COLOR_BUFFER_BIT );

    glBegin( GL_QUADS );
    {
        /* A square with texture coords */
        glTexCoord2f( 0.0, 0.0 ); glVertex2f( -0.5, -0.5 );
        glTexCoord2f( 1.0, 0.0 ); glVertex2f( 0.5, -0.5 );
        glTexCoord2f( 1.0, 1.0 ); glVertex2f( 0.5, 0.5 );
        glTexCoord2f( 0.0, 1.0 ); glVertex2f( -0.5, 0.5 );
    }
    glEnd();
    glFlush();
}

void init( void )
{
    /* Put the image into texture memory */
    glTexImage2D( GL_TEXTURE_2D, /* texture dim and whether or not a proxy */
        0, /* texture level. Always 0 unless mipmapping */
        GL_RGBA, /* suggested internal format */
        4, 4, /* width, height of the source image */
        0, /* border. Not covered in this course */
        GL_RGBA, /* source image format */
        GL_UNSIGNED_BYTE, /* source image data type */
        pImage /* pointer to source image */ );

    /* Specify the magnification/minification method */
}
```

```

glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );

/* Replace the current polygon colours with the texture */
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE );

/* Enable texturing */
glEnable( GL_TEXTURE_2D );
}

int main( int argc, char *argv[] )
{
    glutInit( &argc, argv );
    glutCreateWindow( argv[0] );

    glutDisplayFunc( display );
    init();
    glutMainLoop();
    return 0;
}

```

### 3.4.2 Texture Specification

Let us now examine the above code in light of the previously mentioned steps to texturing. Step 1 (specifying the texture) is achieved by a call to

```

void glTexImage2D( GLenum target,
                  GLint level,
                  GLint internalFormat,
                  GLsizei width,
                  GLsizei height,
                  GLint border,
                  GLenum format,
                  GLenum type,
                  const GLvoid* pTexels );

```

where

- *target* is one of `GL_TEXTURE_2D` or `GL_PROXY_TEXTURE_2D`. If `GL_TEXTURE_2D` is used then the image is loaded into texture memory. If `GL_PROXY_TEXTURE_2D` is used then OpenGL does a calculation to determine whether or not it is theoretically possible for the requested texture to fit into texture memory. The success or failure of the proxy is determined by calls to `glGetTexLevelParameter*()`. Note that even if it is theoretically possible to fit the texture into texture memory, it may not be practical if other textures are currently filling the texture memory.
- *level* is the magnification/minification level. If only one resolution of the texture is being supplied then this should be zero. We will examine this further in the mipmapping section.

- *internalFormat* is the `_suggested_` format for OpenGL to use when storing the texture in texture memory. The specific implementation of OpenGL is free to substitute a close approximation. Note that this internal format does not have to be the same as the other format parameter. There are 38 possible internal formats. A few are `GL_ALPHA8`, `GL_INTENSITY12`, `GL_RGB`, `GL_R3_G3_B2`, `GL_RGBA`, `GL_RGB5_A1`, etc.
- *width* and *height* are the width and height of the source image. These dimensions `_must_` be a power of two.
- *border* is the thickness of the border of the image over and above the width and height specified previously. We will not be covering the use of borders in this course and so this parameter will always be set to zero.
- *format* is the format that the image is currently in. *format* can be one of `{GL_COLOR_INDEX, GL_RGB, GL_RGBA, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_LUMINANCE, GL_LUMINANCE_ALPHA}`.
- *type* is the type of the data that the image is currently in. *type* can be one of `{GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_BITMAP, or one of the packed pixel data types}`.

Note that the data in the array starts at the bottom left of the image and is row-wise until the top right of the image is reached.

### 3.4.3 Texture Mapping Parameters

Step 2 of Section 3.4.1 is implemented by using two functions: `glTexParameter*()` and `glTexEnv*()`. This step was only necessary in the example because the defaults were inappropriate (in fact the defaults cause nothing to appear on the screen for reasons we will go into later). The purpose of `glTexParameter*()` is to specify how the texture will be treated as it is applied to a fragment or stored in a texture object (these will be discussed later). For example, if we "run out of texture" do we repeat the texture again or clamp to the edge value? `glTexParameter*()` is specified by

```
void glTexParameter{if} ( GLenum target, GLenum paramname, TYPE param );
void glTexParameter{if}v( GLenum target, GLenum paramname, TYPE* param );
```

where

- *target* is one of `{GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D}`. We will only be using `GL_TEXTURE_2D` in this course.
- *paramname* and *param* (or a pointer to *param*) are chosen from the following table

Parameter Name	Value
GL_TEXTURE_WRAP_S	GL_CLAMP, GL_CLAMP_TO_EDGE, GL_REPEAT (default)
GL_TEXTURE_WRAP_T	GL_CLAMP, GL_CLAMP_TO_EDGE, GL_REPEAT (default)
GL_TEXTURE_WRAP_R	GL_CLAMP, GL_CLAMP_TO_EDGE, GL_REPEAT (default)
GL_TEXTURE_MAG_FILTER	GL_NEAREST, GL_LINEAR (default)
GL_TEXTURE_MIN_FILTER	GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR (default), GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR
GL_TEXTURE_BORDER_COLOR	a vector of four colours (default = {0,0,0,0})
GL_TEXTURE_PRIORITY	GLfloat in [0, 1] (default = 1)
GL_TEXTURE_MIN_LOD	GLfloat (default = -1000)
GL_TEXTURE_MAX_LOD	GLfloat (default = 1000)
GL_TEXTURE_BASE_LEVEL	non-negative GLint (default = 0)
GL_TEXTURE_MAX_LEVEL	non-negative GLint (default = 1000)

We will elaborate on the meaning of these parameters later.

The purpose of `glTexEnv*()` is to specify how the values in the texture image are mapped to a fragment. For example the texture colours could simply replace the fragment's current colour or the two colours could be blended. There are four possible texturing functions to choose from. `glTexEnv*` is specified by

```
void glTexEnv{if} ( GLenum target, GLenum paramname, TYPE param );
void glTexEnv{if}v( GLenum target, GLenum paramname, TYPE* param );
```

where

- *target* (OpenGL v1.2) must be set to `GL_TEXTURE_ENV`,
- *paramname* can be `GL_TEXTURE_ENV_MODE` or `GL_TEXTURE_ENV_COLOR`,
- *param* is one of {`GL_DECAL`, `GL_REPLACE`, `GL_MODULATE`, `GL_BLEND`} if *paramname* is `GL_TEXTURE_ENV_MODE`. When *paramname* is `GL_TEXTURE_ENV_COLOR` then *param* must be a vector of RGBA values. This colour is only used when the texture function is `GL_BLEND`.

The exact application of the texture to the fragment depends on both the texturing function and the internal format of the texture image. The following table details the calculation used for each case.

Internal Format	Texture Function			
	Replace	Modulate	Decal	Blend
GL_ALPHA	$C = C_f$ $A = A_t$	$C = C_f$ $A = A_f A_t$	undefined	$C = C_f$ $A = A_f A_t$
GL_LUMINANCE	$C = L_t$ $A = A_f$	$C = C_f L_t$ $A = A_f$	undefined	$C = C_f(1-L_t) + C_c L_t$ $A = A_f$
GL_LUMINANCE_ALPHA	$C = L_t$ $A = A_t$	$C = C_f L_t$ $A = A_f A_t$	undefined	$C = C_f(1-L_t) + C_c L_t$ $A = A_f A_t$
GL_INTENSITY	$C = I_t$ $A = I_t$	$C = C_f I_t$ $A = A_f I_t$	undefined	$C = C_f(1-I_t) + C_c I_t$ $A = A_f(1-I_t) + A_c I_t$
GL_RGB	$C = C_t$ $A = A_f$	$C = C_f C_t$ $A = A_f$	$C = C_t$ $A = A_f$	$C = C_f(1-C_t) + C_c C_t$ $A = A_f$
GL_RGBA	$C = C_t$ $A = A_t$	$C = C_f C_t$ $A = A_f A_t$	$C = C_f(1-A_t) + C_t A_t$ $A = A_f$	$C = C_f(1-C_t) + C_c C_t$ $A = A_f A_t$

In the above table C means colour and the attached subscript means f = fragment, t = texture, c = colour (specified by GL\_TEXTURE\_ENV\_COLOR), none = final colour. A = alpha, I = intensity, L = luminance, with the same subscripts as the colour.

The replace function simply replaces the colour of the fragment with that of the texture. Modulate is mostly used with lighting because the fragment colour can attenuate the texture. Decal is mostly used in RGBA mode so that alpha mapped textures apply as you would intuitively expect. Blend is mostly used to achieve an effect similar to decal but for the modes where decal is undefined.

### 3.4.4 Texture Coordinates

The implementation of step three is rather straight forward. This only leaves step 4, the specification of texture coordinates. Texture coordinates specify which part of the texture is assigned to a particular vertex. The coordinates are then interpolated between vertices in much the same way that colour is interpolated between vertices in the smooth shading mode. Texture coordinates are specified by using

```
void glTexCoord{1234}{sifd} ( TYPE coords );
void glTexCoord{1234}{sifd}v( TYPE* coords );
```

Texture coordinates are traditionally called the s, t, r, and q coordinates to distinguish them from object coordinates. If any of the coordinates are not specified (except s; s must always be specified) then the defaults are t = 0, r = 0, q = 1. As with the object coordinates, texture coordinates are homogeneous coordinates, with the fourth coordinate, q, being the scaling factor. The coordinate (0.0, 0.0, 0.0, 1.0) refers to the bottom left of the texture and (1.0, 1.0, 0.0, 1.0) refers to the top right (of a 2D texture). Coordinate values greater than 1.0 are allowable and the texel that they refer to is determined by the wrapping mode (covered in a following section).

The question then arises about how to choose the texture coordinates for your polygon. For geometrically flat objects (planar polygons, e.g. triangles, cylinders, cones) this is easy. In these cases you just apply the natural mapping from the object

to the correct part of the texture. For non-flat objects the application of texture coordinates is more of an art.

Another issue that needs to be thought about is distortion of the texture. If you map a square texture onto a rectangular object, the image will distort (notice that the aspect ratio has been changed) unless you only use part of the texture. Part of the texture can be used by specifying texture coordinates like (0.0, 0.0), (0.5, 0.0), (0.5, 0.3), (0.0, 0.3). If you are mapping onto an object like a sphere or a torus (doughnut) then you can't avoid distortion.

### **3.4.5 Texture Wrapping Modes**

Let us now examine texture coordinates greater than 1.0. What does 2.4 mean when 1.0 describes the top and right extremes. The answer is that it depends on how the `GL_TEXTURE_WRAP*` parameters (see `glTexParameter`) are set. If `GL_TEXTURE_WRAP_S` is set to `GL_REPEAT` then the image is repeated 2.4 times in the x-direction, and similarly for the other `GL_TEXTURE_WRAP*` directions. If `GL_TEXTURE_WRAP_S` is set to `GL_CLAMP_TO_EDGE` then the image occupies the lower left corner of the polygon until the texture coordinate reaches 1.0 then the colour is fixed for the rest of the x-direction until the edge of the polygon is reached. This results in a smearing of the edge of the texture across the remainder of the polygon. If `GL_TEXTURE_WRAP_S` is set to `GL_CLAMP` then the border or border colour in combination with the texture filtering are used to decide on what colour will be smeared over the rest of the polygon. We do not go into the details of this wrapping mode in this course.

### **3.4.6 Minification and Magnification Filters**

It is a rare occurrence when the texels of a texture are mapped one to one onto pixels of the screen. Usually the texture image is too small and one texel is being mapped to many pixels (magnification) or the texture image is too large and many texels are being mapped to one pixel (minification). OpenGL allows the specification of the functions that do the magnification and minification. The functions for magnification and minification are allowed to be different.

OpenGL only applies one magnification or minification operation during the mapping of the texture onto the fragment. This has the consequence that if the polygon has been stretched in one direction and squeezed in another direction (so that both magnification and minification seem necessary) OpenGL will try and choose the best operation but it will not do both.

From above you can see that (excluding mipmapping which is covered in the next section) there are two different mag/min functions to choose from. These are `GL_NEAREST` and `GL_LINEAR`. `GL_NEAREST` means that the pixel which is closest to the centre of the pixel will be used for the min/mag process. This operation is very fast but can lead to bad aliasing. `GL_LINEAR` means that (for 2D textures) the weighted average of a 2 by 2 texel array is used for the mag/min process. The weighted average near the edge of the texture uses pixels depending on the texture

wrapping mode and any border colours specified. GL\_LINEAR produces smoother images but is computationally more complex.

The pixels chosen for the mag/min by GL\_NEAREST and GL\_LINEAR are always from the base level texture image (level zero) even if other levels are specified (see mipmapping for more details).

### 3.4.7 Mipmapping

Mipmapping is a technique which provides improved visual appearance of textures under minification.

The standard minification filters specified above can result in artefacts such as shimmering, flashing, and scintillation due to the algorithms used (think about what happens under GL\_NEAREST if the object is getting smaller and a different texel is now "nearest"). To avoid these artefacts OpenGL provides a facility to specify predetermined minification textures - called mipmaps. Use of mipmaps involves specifying all "power of two sizes" from the base level texture down to the 1x1 texture map. For example, if the base level (level 0) is a 32x16 texel image then you must specify the 16x8 (level 1), 8x4(level 2), 4x2(level 3), 2x1(level 4), and 1x1(level 5) images. These mipmaps are specified by using glTexImage2D.

#### EXAMPLE

```
float baseimage[] = { ... };  
float level1image[] = { ... };
```

```
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA, 32, 16, 0, GL_RGBA, GL_UNSIGNED_BYTE, baseimage );  
glTexImage2D( GL_TEXTURE_2D, 1, GL_RGBA, 16, 8, 0, GL_RGBA, GL_UNSIGNED_BYTE, level1image );
```

To make OpenGL use the mipmap images you must specify that the minification function is one of the mipmap minification function. Note that mipmaps are only useful in minification and not magnification. The mipmap specific minification functions are specified by GL\_NEAREST\_MIPMAP\_NEAREST, GL\_NEAREST\_MIPMAP\_LINEAR(default), GL\_LINEAR\_MIPMAP\_NEAREST, and GL\_LINEAR\_MIPMAP\_LINEAR.

The GL\_\*\_MIPMAP\_NEAREST function chooses the mipmap closest to the correct magnification. Then if \* = NEAREST the closest texel to the centre of the pixel is chosen, else if \* = LINEAR a 2x2 linear average is chosen for the pixel colour. These GL\_\*\_MIPMAP\_NEAREST functions are quick but there is a sudden transition from one mipmap to the next.

The GL\_\*\_MIPMAP\_LINEAR function chooses the two closest mipmaps and then does a linear interpolation between the texel colours. If \* = NEAREST then the interpolation is between the closest texels to the centre of the pixel else if \* = LINEAR then the interpolation is between the 2x2 linear average of the texels. As

expected, `GL_LINEAR_MIPMAP_LINEAR` produces the best quality images but at a computational cost.

This presentation of mipmaps is a simplified version of what OpenGL really does. This course has not covered details such as `GL_TEXTURE_MIN_LOD`, `GL_TEXTURE_MAX_LOD`, `GL_TEXTURE_BASE_LEVEL`, and `GL_TEXTURE_MAX_LEVEL`.

### **3.4.8 Troubleshooting**

A typical problem with textures is that a texture doesn't appear on the polygon when you thought that it should. The problem is usually to do with mipmapping. If you have set the minification filter to any of the mipmapping functions (e.g. the default is `GL_NEAREST_MIPMAP_LINEAR`) then all the levels of mipmaps from the base level to the maximum level (by default a 1x1 texel image) must have been specified. If you missed any level, or specified an illegal texture, then texturing will be disabled.

### **3.4.9 Automated MIPMAP Generation**

The OpenGL Utility library has a function to help with the creation of the mipmap images. Given an image the function `gluBuild2DMipmaps` constructs the pyramid of mipmaps from level 0 down to level N (where level N is a 1x1 texel image). If the given image is not a "power of 2 size" then a copy is created of the correct power size. The function then generates a proxy to check that the texture will fit into memory and, given the texture will fit, calls `glTexImage2D` for you. This occurs until the 1x1 texel image has been created. `gluBuild2DMipmaps` is specified by

```
int gluBuild2DMipmaps( GLenum target,
                      GLint internalFormat,
                      GLsizei width,
                      GLsizei height,
                      GLenum format,
                      GLenum type,
                      const GLvoid* pTexels );
```

where the parameters are the same as `glTexImage2D()`. An error code is returned, with 0 if all is well.

### **3.4.10 Texture Objects**

So far we have shown you how to create textures as they are needed. Consider now an application that uses more than one texture. Furthermore assume that the drawing order specifies that the first texture is used, then the second texture, and then back to the first. Under the process outlined so far you would have to load the first texture (using `glTexImage2D` or `gluBuild2DMipmaps`) then load the second texture, and then reload the first. This reloading can impinge upon the performance of a program. It is (usually) possible to avoid this performance hit by using texture objects.

Texture objects contain the image, including any mipmaps, and associated data like height, width and min/mag filters.

A texture object is named and used via the same command.

```
void glBindTexture( GLenum target, GLuint textureName );
```

where

- *target* is one of { GL\_TEXTURE\_1D, GL\_TEXTURE\_2D, GL\_TEXTURE\_3D } and
- *textureName* is an unsigned integer.

When choosing the texture name for a new texture, do not use zero. Zero is the name of the default texture. While it is useful to bind to the default texture to use it, all new textures must have a non-zero unsigned integer as their name.

`glBindTexture` creates a new texture object whenever a new texture name is used. This new texture is initialised to the default texture settings so any settings used by a previous texture are now no longer active. Reread the previous sentence, as not understanding that point is a common cause of unexpected behaviour. Whenever an existing texture name is used, the effect is to activate that object's settings.

### Example

In an initialisation function we set up the texture object via

```
glBindTexture( GL_TEXTURE_2D, 1 );  
  
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA, 4, 4, 0, GL_RGBA, GL_UNSIGNED_BYTE, pImage );  
  
/* set up more parameters here */
```

and in a rendering function the texture is used via

```
glBindTexture( GL_TEXTURE_2D, 1 );  
/* now drawing commands with texture coords */
```

When a texture is no longer needed in a program, the texture resources tied up with the texture object can be freed by a call to

```
void glDeleteTextures( GLsizei n, const GLuint* textureNames );
```

where *n* is the number of texture objects to clean up in the array pointed to by *textureNames*.

To avoid accidentally reusing names, OpenGL provides a mechanism for generating texture names.

```
void glGenTextures( GLsizei n, GLuint* textureNames );
```

where  $n$  is the number of names to generate and *textureNames* is an array of GLuints (at least  $n$  in size).

It should be noted that the numbers that are given back for the texture names are not necessarily contiguous.

#### EXAMPLE

```
#define MAXTEXNAMES 2
GLuint texNames[MAXTEXNAMES];
glGenTextures( MAXTEXNAMES, texNames );
glBindTexture( GL_TEXTURE_2D, texNames[0] );
/* set up the first texture */
glBindTexture( GL_TEXTURE_2D, texNames[1] );
/* set up the second texture */
and so on
```

### **3.4.11 Lighting and Textures**

Lighting is a per vertex operation and texturing is a per fragment operation. Thus lighting occurs first in the rendering pipeline. To still have lighting effects it is thus necessary to use an environment mode such as `GL_MODULATE` or `GL_DECAL`. Because this can still mute specular highlights OpenGL provides a mode of operation so that in the original lighting calculation the specular component is kept separate and not combined with the other lighting calculations. After the texturing operation this specular component is then combined into the fragment.

The separation of the specular component calculations is achieved with a call to

```
glLightModeli( GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR );
```

### **3.4.12 Perspective Correction Hint**

Both colour and texture coordinates are normally interpolated linearly between vertices. For many applications this provides a visually acceptable image even if it is not technically correct. However, when it does matter the interpolation can be set to include a perspective correction. This is achieved with a call to

```
glHint( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
```

As usual the trade-off is lower performance for higher visual acceptability.