CA417 - Computer Graphics

OpenGL - Part 3

David Sinclair

Translation

The **model view mode** affects the positioning of objects relative to the camera. The **projection mode** specifies the projection, clipping and mapping to normalised projection coordinates.

Translation

The **model view mode** affects the positioning of objects relative to the camera. The **projection mode** specifies the projection, clipping and mapping to normalised projection coordinates.

Objects are typically modelled with respect to the origin in object coordinates. The default camera position in OpenGL is at the origin, oriented along the $-z$ axis.

Translation

The **model view mode** affects the positioning of objects relative to the camera. The **projection mode** specifies the projection, clipping and mapping to normalised projection coordinates.

Objects are typically modelled with respect to the origin in object coordinates. The default camera position in OpenGL is at the origin, oriented along the $-z$ axis.

Translation is applied to the current matrix (usually the projection matrix) using the `glTranslate` function.

```
void glTranslatef (dx, dy, dz) or
void glTranslated (dx, dy, dz)
```

where (`dx`, `dy`, `dz`) is the displacement applied to the object is either of type `GLfloat` or `GLdouble` respectively.

Translation (2)

The following code fragment moves the current scene 2 world coordinate units away from the camera along the z-axis.

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glTranslate (0.0, 0.0, -2.0);
```

Composing Transformations

Transformation can be concatenated. Compare the following 2 code fragments.

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glTranslatef (0.0, 0.0, -2.0);
glutWireCube ();
glLoadIdentity ();
glTranslatef (0.0, 0.0, -3.0);
glutWireSphere ();
```

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glTranslatef (0.0, 0.0, -2.0);
glutWireCube ();
glTranslatef (0.0, 0.0, -3.0);
glutWireSphere ();
```

These produce different outputs. Why?

Composing Transformations

Transformation can be concatenated. Compare the following 2 code fragments.

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glTranslatef (0.0, 0.0, -2.0);
glutWireCube ();
glLoadIdentity ();
glTranslatef (0.0, 0.0, -3.0);
glutWireSphere ();
```

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glTranslatef (0.0, 0.0, -2.0);
glutWireCube ();
glTranslatef (0.0, 0.0, -3.0);
glutWireSphere ();
```

These produce different outputs. Why?

The first fragment resets the model view matrix (the cube at -2 and sphere at -3), while the second fragment composes the translations (the cube at -2 and sphere at -5).

Rotation

In OpenGL we can rotate an object around an arbitrary axis passing throughout the origin by using the `glRotate` function.

`void glRotatef (angle, dx, dy, dz)` or
`void glRotated (angle, dx, dy, dz)`

where `angle` is the counterclockwise rotation about the axis defined by the vector (`dx`, `dy`, `dz`).

Rotation

In OpenGL we can rotate an object around an arbitrary axis passing throughout the origin by using the `glRotate` function.

`void glRotatef (angle, dx, dy, dz)` or
`void glRotated (angle, dx, dy, dz)`

where `angle` is the counterclockwise rotation about the axis defined by the vector (`dx`, `dy`, `dz`).

To rotate around an axis defined by (`dx`, `dy`, `dz`) through an arbitrary point (`x`, `y`, `z`) we need to compose 2 translations and a rotation in the **right** order.

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glTranslatef (x, y, z);
glRotatef (angle, dx, dy, dz);
glTranslatef (-x, -y, -z);
```

Remember that the last transform specified is the first applied!

Scaling

Scaling in OpenGL is with respect to the origin. The OpenGL scaling functions are:

`void glScalef (sx, sy, sz)` or
`void glScaled (sx, sy, sz)`

where `sx`, `sy` and `sz`) are the scaling factors along the x-axis, y-axis and z-axis respectively.

Setting the Matrix Directly

The **model view matrix** or the **projection matrix** can be set directly using `glLoadMatrix`.

```
void glLoadMatrixf (GLfloat *m) or
void glLoadMatrixd (GLdouble *m)
```

where `m` points to a 16-element array in column order.

Setting the Matrix Directly

The **model view matrix** or the **projection matrix** can be set directly using `glLoadMatrix`.

```
void glLoadMatrixf (GLfloat *m) or
void glLoadMatrixd (GLdouble *m)
```

where `m` points to a 16-element array in column order.

```
void glMultMatrixf (GLfloat *m) or
void glMultMatrixd (GLdouble *m)
```

post-multiplies the current matrix by `m`.

Setting the Matrix Directly (2)

To shear along the x-axis we need the following matrix.

$$\begin{bmatrix} 1 & cot(\theta) & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Setting the Matrix Directly (2)

To shear along the x-axis we need the following matrix.

$$\begin{bmatrix} 1 & cot(\theta) & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

With this we can generate an oblique projection.

```
GLfloat shear[16];
for (i = 0; i < 16, i++)
  shear[i] = 0.0;
shear[0] = shear[5] = shear[10] = shear[15] = 1.0;
shear[4] = 1 / tan(DEG_TO_RAD*theta);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(left, right, bottom, top, near, far);
glMultMatrix(shear);
glMatrixMode(GL_MODELVIEW);
```

Hierarchical Modelling

Consider a simple manufacturing robot that consists of a base and an arm. The arm consists of a lower arm, connected to the base and an upper arm that is connected to the lower arm. If the base rotates in the $x - z$ plane, then the lower and upper arms will also rotate.

Hierarchical Modelling

Consider a simple manufacturing robot that consists of a base and an arm. The arm consists of a lower arm, connected to the base and an upper arm that is connected to the lower arm. If the base rotates in the $x - z$ plane, then the lower and upper arms will also rotate.

The arms can rotation the $x - y$ plane. If the lower arm is rotated, then the position of the upper arm will be effected.

Hierarchical Modelling

Consider a simple manufacturing robot that consists of a base and an arm. The arm consists of a lower arm, connected to the base and an upper arm that is connected to the lower arm. If the base rotates in the $x - z$ plane, then the lower and upper arms will also rotate.

The arms can rotation the $x - y$ plane. If the lower arm is rotated, then the position of the upper arm will be effected.

The following code illustrates how this could be done in a hierarchical fashion.

Hierarchical Modelling (2)

```
GLUquadricObj *obj;
GLfloat theta[3];

void base ()
{
  glPushMatrix ();

  \\ align cylinder with y-axis
  glRotatef (-90.0, 1.0, 0.0, 0.0);

  gluCylinder (p, BASE_RADIUS, BASE_RADIUS, BASE_HEIGHT, 5, 5);
  glPopMatrix ();
}

void lower_arm ()
{
  glPushMatrix ();
  glTranslatef (0.0, 0.5*LOWER_ARM_HEIGHT, 0.0);
  glScalef (LOWER_ARM_WIDTH, LOWER_ARM_HEIGHT, LOWER_ARM_WIDTH);
  glutWireCube (1.0);
  glPopMatrix ();
}
```

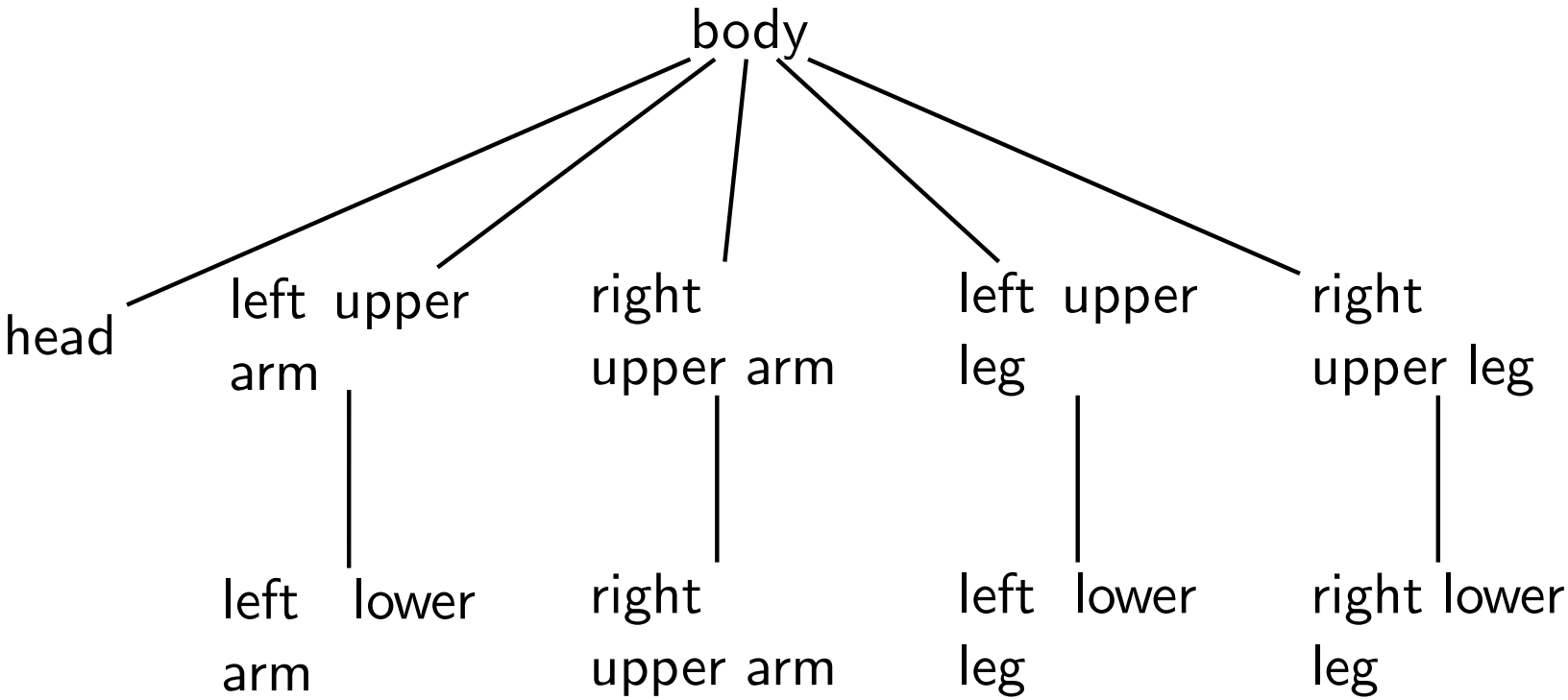
Hierarchical Modelling (3)

```
void upper_arm ()
{
  glPushMatrix ();
  glTranslatef (0.0, 0.5*UPPER_ARM_HEIGHT, 0.0);
  glScalef (UPPER_ARM_WIDTH, UPPER_ARM_HEIGHT, UPPER_ARM_WIDTH);
  glutWireCube (1.0);
  glPopMatrix ();
}

void display ()
{
  glClear (GL_COLOR_BUFFER_BIT);
  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity ();
  glColor3f (1.0, 0.0, 0.0);
  glRotatef (theta[0], 0.0, 1.0, 0.0);
  base ();
  glTranslatef (0.0, BASE_HEIGHT, 0.0):
  glRotatef (theta[1], 0.0,0.0, 1.0);
  lower_arm ();
  glTranslatef (0.0, LOWER_ARM_HEIGHT, 0.0):
  glRotatef (theta[2], 0.0,0.0, 1.0);
  upper_arm ();
  glutSwapBuffers ();
}
```

Hierarchical Modelling (4)

A more complicated object where some component objects are independent of other component objects, but dependent on others, can be modelled as a tree.



Hierarchical Modelling (5)

By using a "left-child right-sibling" data structure we can traverse the tree as follows.

```
typedef struct treenode
{
  GLfloat m[16];
  void (*f) ();
  struct treenode *sibling;
  struct tree node *child;
} treenode;

void traverse (treenode *root)
{
  if (root == NULL) return;
  glPushNatrix ();
  glMultMatrix (root ->m);
  root -> f ();
  if (root.child != NULL)
    traverse (root->child);
  glPopMatrix ();
  if (root->sibling != NULL)
    traverse (root->sibling);
}
```

Hierarchical Modelling (6)

```
void display ()
{
   glClear (GL_COLOR_BUFFER_BIT)|
   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity ();
   traverse (&body_node);
   glutSwapBuffers ();
}
```

and the body node could be defined in an `init` function as:

```
tree node body_node;
glLoadIdentity ()
glRotatef (theta[0], 0.0, 1.0, 0.0);
glGetFloatv (GL_MODELVIEW, body_node.m);
body_node.f = body;
body_node.sibling = NULL;
body_node.child = & head_node;
```

where `body()` is the function that draws the body.

The Phong Model

OpenGL uses the **Phong Model** to determine the shade at each point on an object. The **Phong Model** considers 4 types of contributions to the shade at each point.

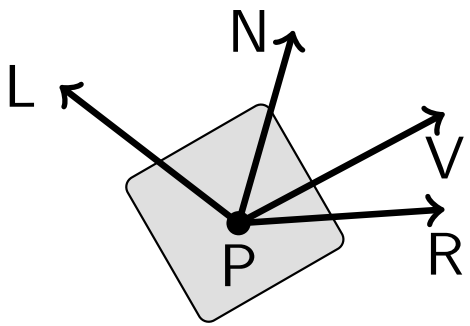
- diffuse reflection
- specular reflection
- ambient reflection
- emissive light

The Phong Model

OpenGL uses the **Phong Model** to determine the shade at each point on an object. The **Phong Model** considers 4 types of contributions to the shade at each point.

- diffuse reflection
- specular reflection
- ambient reflection
- emissive light

To understand these contributions we need to understand the vectors used in the **Phong Model**.



N The normal vector at the point P.

L The vector from point P to the light source.

R The vector of perfect reflection at the point P.

V The vector from point P to the viewer.

The Phong Model (2)

Light that is incident on a surface is partially reflected and partially absorbed. In **diffuse reflection** the reflected light is scattered equally in all directions and hence is not dependent on **V**. It is dependent on the angle between **L** and **N**, being at a maximum when **L** and **N** are colinear. The strength of the scattering may be different for each component of the RGB light. Diffuse surfaces tend to look dull, like plastic.

The Phong Model (2)

Light that is incident on a surface is partially reflected and partially absorbed. In **diffuse reflection** the reflected light is scattered equally in all directions and hence is not dependent on **V**. It is dependent on the angle between **L** and **N**, being at a maximum when **L** and **N** are colinear. The strength of the scattering may be different for each component of the RGB light. Diffuse surfaces tend to look dull, like plastic.

In **specular reflection** the light is concentrated in the direction **R** and hence the amount of specular reflection a viewer sees is dependent on the angle between **R** and **V**. The smoother the surface is, the stronger is the reflected light. The **Phong Model** captures this using a **shininess coefficient**. **Specular reflection** models smooth highly polished surfaces, like metals, and the amount of **specular reflection** may be different for each component of the RGB light. **Specular reflection** is responsible for the highlights in an image.

The Phong Model (3)

Ambient Reflection models the reflection of the ambient light which does not seem to come from any specific point. Hence the **ambient reflection** is solely dependent on the intensity of the ambient light and the fraction of this light that the surface reflects. The fraction of the reflected ambient light may be different for each component of the RGB light.

The Phong Model (3)

Ambient Reflection models the reflection of the ambient light which does not seem to come from any specific point. Hence the **ambient reflection** is solely dependent on the intensity of the ambient light and the fraction of this light that the surface reflects. The fraction of the reflected ambient light may be different for each component of the RGB light.

A surface may emit its own light, **emissive light**. The **Phong Model** does not use **emissive light** to calculate the shading at a point. It is unaffected by other light sources and the position of the viewer, **V**.

Lighting

In OpenGL we can have **point sources**, **spotlights** and **ambient sources**, all of which can be located at a finite distance from the objects or can be looted at infinity. Each light can have separate diffuse, specular and ambient properties for each of its RGB(A) components.

Lighting

In OpenGL we can have **point sources**, **spotlights** and **ambient sources**, all of which can be located at a finite distance from the objects or can be looted at infinity. Each light can have separate diffuse, specular and ambient properties for each of its RGB(A) components.

If we want to model a white light hitting a red wall and resulting in red reflections, we could do this with a light with white diffuse and specular properties and a red ambient property.

Lighting

In OpenGL we can have **point sources**, **spotlights** and **ambient sources**, all of which can be located at a finite distance from the objects or can be looted at infinity. Each light can have separate diffuse, specular and ambient properties for each of its RGB(A) components.

If we want to model a white light hitting a red wall and resulting in red reflections, we could do this with a light with white diffuse and specular properties and a red ambient property.

Lighting calculations and individual light sources must the enables by `glEmable`.

```
glEnable (GL_LIGHTING);
glEnable(GL_LIGHT0);
```

Once lighting is enabled, the colours set by `glColor*` are no longer used. The colour of an object is defined by its material properties and the properties of the lights.

Materials Properties

Like lights, a material has ambient, diffuse, specular and shininess properties. Unlike lights where each light has its own Phong properties, there is only one set of material properties. There are set by:

```
void glMateriali (GLenum face, GLenum name,
                  Glint value)
void glMaterialf (GLenum face, GLenum name,
                  GLfloat value)
void glMaterialiv (GLenum face, GLenum name,
                   GLint *value)
void glMaterialfv (GLenum face, GLenum name,
                   GLfloat *value)
```

where `face` is either `GL_FRONT`, `GL_BACK` to `GL_FRONT_AND_BACK`; and `name` is either `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, `GL_EMISSION` or `GL_SHININESS`.

Materials Properties (2)

```
typedef struct material
{
  GLfloat ambient[4];
  GLfloat diffuse[4];
  GLfloat specular[4];
  GLfloat shininess
} material;

material brass =
{
  {0.33, 0.22, 0.03, 1.0},
  {0.78, 0.57, 0.11, 1.0},
  {0.99, 0.91, 0.81, 1.0},
  27.8
};

material red_plastic =
{
  {0.3, 0.0, 0.0, 1.0},
  {0.6, 0.0, 0.0, 1.0},
  {0.8, 0.6, 0.6, 1.0},
  32.0
};
```

Materials Properties (3)

```
void materials (material *m)
{
  glMaterialfv (GL_FRONT_AND_BACK, GL_AMBIENT, m->ambient);
  glMaterialfv (GL_FRONT_AND_BACK, GL_DIFFUSE, m->diffuse);
  glMaterialfv (GL_FRONT_AND_BACK, GL_SPECULAR, m->specular);
  glMaterialfv (GL_FRONT_AND_BACK, GL_SHININESS, m->shininess);
}
```

so we can set the material properties by a single call

```
materials (&red_plastic);
```

Normals

Both Lighting and Material properties require normals to be defined. Some built-in functions (e.g.`glutSolidTeapot`) will provide normals, but in general you need to provide the normals, one for each vertex or one for each polygon.

`glNormal* (type dx, type dy, type dz)` or
`glNormal*v (type dx, type dy, type dz)`

where * is from [bsidf], type corresponds to * and (`dx`, `dy`, `dz`) is the normal vector.

Specifying a Light Source

A light source can be specified by the following functions.

```
void glLighti (GLenum light, GLenum param,
               Glint value)
void glLightf (GLenum light, GLenum param,
               GLfloat value)
void glLightiv (GLenum light, GLenum name,
                GLint *value)
void glLightfv (GLenum light, GLenum param,
                GLfloat *value)
```

where `light` specifies the lighting source and `value` is assigned to the named parameter `param`.

Specifying a Light Source

A light source can be specified by the following functions.

```
void glLighti (GLenum light, GLenum param,
               Glint value)
void glLightf (GLenum light, GLenum param,
               GLfloat value)
void glLightiv (GLenum light, GLenum name,
                GLint *value)
void glLightfv (GLenum light, GLenum param,
                GLfloat *value)
```

where `light` specifies the lighting source and `value` is assigned to the named parameter `param`.

The default values for `GL_LIGHT0` white diffuse and specular components located at (0.0, 0.0, 1.0, 0.0) in eye coordinates (at infinity, w=0.0, looking along the -ve z axis). All other components and lights are set to black.

Specifying a Light Source (2)

For a **point light** the parameters are `GL_POSITION`, `GL_DIFFUSE`, `GL_SPECULAR` and `GL_AMBIENT`. **Directional lights** are point lights placed at infinity (w=0.0) directed along the specified (x, y, z) vector.

Specifying a Light Source (2)

For a **point light** the parameters are `GL_POSITION`, `GL_DIFFUSE`, `GL_SPECULAR` and `GL_AMBIENT`. **Directional lights** are point lights placed at infinity (w=0.0) directed along the specified (x, y, z) vector.

Spotlights have 2 additional parameters, their direction (`GL_SPOT_DIRECTION`) and the angle of the cone (`GL_SPOT_CUTOFF`). The default value is 180 degrees. The amount by which the spotlight intensity falls off from its centre is specified by `GL_SPOT_EXPONENTIAL`.

Specifying a Light Source (2)

For a **point light** the parameters are `GL_POSITION`, `GL_DIFFUSE`, `GL_SPECULAR` and `GL_AMBIENT`. **Directional lights** are point lights placed at infinity (w=0.0) directed along the specified (x, y, z) vector.

Spotlights have 2 additional parameters, their direction (`GL_SPOT_DIRECTION`) and the angle of the cone (`GL_SPOT_CUTOFF`). The default value is 180 degrees. The amount by which the spotlight intensity falls off from its centre is specified by `GL_SPOT_EXPONENTIAL`.

Light intensity attenuates with the distance from the light sources. The model used for attenuation is $1/(a + bd + cd^2)$ where d is the distance from the light source. The values a, b and c can be set using `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION` and `GL_QUADRATIC_ATTENUATION` respectively. The default is no attenuation (a=1.0, b=0.0, c= 0.0)

Lighting Example

```
GLfloat light_pos[] = {1.0, 2.0, 3.0, 1.0};
GLfloat dx = 1.0; GLfloat dy = 1.0; GLfloat dz = 1.0;
GLfloat theta = 0.0;

void init ()
{
  // rest of init

  glEnable (GL_LIGHTING);
  glEnable (GL_LIGHT0);
  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity ();
  glLightfv (GL_LIGHT0, GL_POSITION, light_pos);
}

void idle ()
{
  angle += 2.0;
  if (angle > 360.0)
    angle -= 360.0;
}
```

Lighting Example (2)

```
void display ()
{
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity ();
  glLookAt (1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

  glPushMatrix ()
  glRotatef (angle, dx, dy, dz);
  glLightfv (GL_LIGHT0, GL_POSITION, light_pos);
  glPopMatrix ();

  // draw objects

  glutSwapBuffers ();
}
```

Texels and Pixels

Textures in OpenGL are implemented by "wallpapering" a 2-D images across the surface of a 3-D surface. Hence for every projected element of the surface (**pixel**) we need to associate it with an element of the **texture image** (**texel**).

Texels and Pixels

Textures in OpenGL are implemented by "wallpapering" a 2-D images across the surface of a 3-D surface. Hence for every projected element of the surface (**pixel**) we need to associate it with an element of the **texture image** (**texel**).

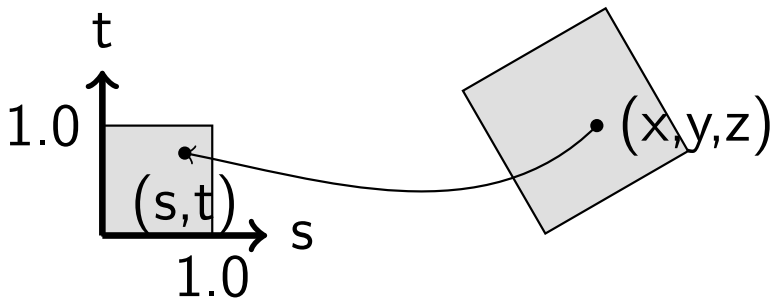
OpenGL models the **texture image** is a continuous image with s and t dimensions that range from (0,0), at the lower-left corner of the image, to (1,1), at the upper-right corner of the image.

Texels and Pixels

Textures in OpenGL are implemented by "wallpapering" a 2-D images across the surface of a 3-D surface. Hence for every projected element of the surface (**pixel**) we need to associate it with an element of the **texture image** (**texel**).

OpenGL models the **texture image** is a continuous image with s and t dimensions that range from (0,0), at the lower-left corner of the image, to (1,1), at the upper-right corner of the image.

Hence we need 2 functions that map each **pixel** to a **texel**.



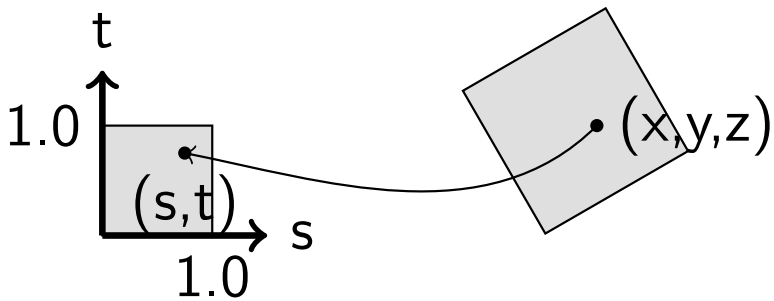
$$s = f(x, y, z)$$
$$t = g(x, y, z)$$

Texels and Pixels

Textures in OpenGL are implemented by "wallpapering" a 2-D images across the surface of a 3-D surface. Hence for every projected element of the surface (**pixel**) we need to associate it with an element of the **texture image** (**texel**).

OpenGL models the **texture image** is a continuous image with s and t dimensions that range from (0,0), at the lower-left corner of the image, to (1,1), at the upper-right corner of the image.

Hence we need 2 functions that map each **pixel** to a **texel**.



$$s = f(x, y, z)$$
$$t = g(x, y, z)$$

We do not explicitly specify the functions f and g. Instead we give each vertex a texture coordinate and OpenGL infers the required texture coordinates by interpolation.

Texture Maps

OpenGL supports 1-D, 2-D and 3-D texture mappings. We will focus on 2-D texture maps. There are similar OpenGL functions for 1-D and 3-D texture maps.

Texture Maps

OpenGL supports 1-D, 2-D and 3-D texture mappings. We will focus on 2-D texture maps. There are similar OpenGL functions for 1-D and 3-D texture maps.

```
GLubyte image [64][64][3];
glEnable (GL_TEXTURE_2D);

glTexImage2D (GL_TEXTURE_2D, 0, GL_RGB, 64, 64, 0,
              GL_RGB, GL_UNSIGNED_BYTE, image);
```

Texture Maps

OpenGL supports 1-D, 2-D and 3-D texture mappings. We will focus on 2-D texture maps. There are similar OpenGL functions for 1-D and 3-D texture maps.

```
GLubyte image [64][64][3];
glEnable (GL_TEXTURE_2D);

glTexImage2D (GL_TEXTURE_2D, 0, GL_RGB, 64, 64, 0,
              GL_RGB, GL_UNSIGNED_BYTE, image);
```

- Texturing has to be enabled.

Texture Maps

OpenGL supports 1-D, 2-D and 3-D texture mappings. We will focus on 2-D texture maps. There are similar OpenGL functions for 1-D and 3-D texture maps.

```
GLubyte image [64][64][3];
glEnable (GL_TEXTURE_2D);

glTexImage2D (GL_TEXTURE_2D, 0, GL_RGB, 64, 64, 0,
              GL_RGB, GL_UNSIGNED_BYTE, image);
```

- Texturing has to be enabled.
- We are storing the image in a 64x64x3 array as the image is in RGB format.

Texture Maps

OpenGL supports 1-D, 2-D and 3-D texture mappings. We will focus on 2-D texture maps. There are similar OpenGL functions for 1-D and 3-D texture maps.

```
GLubyte image [64][64][3];
glEnable (GL_TEXTURE_2D);

glTexImage2D (GL_TEXTURE_2D, 0, GL_RGB, 64, 64, 0,
              GL_RGB, GL_UNSIGNED_BYTE, image);
```

- Texturing has to be enabled.
- We are storing the image in a 64x64x3 array as the image is in RGB format.
- The 1^{st} parameter is either `GL_TEXTURE_2D` or `GL_PROXY_TEXTURE_2D`.
- The 2^{nd} parameter set the **mipmapping** level. **Mipmapping** allows to use a sequence of texture images at different resolutions for efficiency reasons. Level 0 is the highest level and we use this value when we do not want to use **mipmapping**.

Texture Maps (2)

- The 3^{rd} parameter specifies the format of the texture image. There are several image formats, the most common being `GL_RGB`, `GL_RGBA` and `GL_LUMINANCE`.

Texture Maps (2)

- The 3^{rd} parameter specifies the format of the texture image. There are several image formats, the most common being `GL_RGB`, `GL_RGBA` and `GL_LUMINANCE`.
- The 4^{th} and 5^{th} parameters are the number of rows and columns respectively in the texture image. These **must** be a power of 2.

Texture Maps (2)

- The 3^{rd} parameter specifies the format of the texture image. There are several image formats, the most common being `GL_RGB`, `GL_RGBA` and `GL_LUMINANCE`.
- The 4^{th} and 5^{th} parameters are the number of rows and columns respectively in the texture image. These **must** be a power of 2.
- The 6^{th} parameter is the border width. This is either 0 or 1. A border width of 1 can be used to create a smooth texture map when using filtering.

Texture Maps (2)

- The 3^{rd} parameter specifies the format of the texture image. There are several image formats, the most common being `GL_RGB`, `GL_RGBA` and `GL_LUMINANCE`.
- The 4^{th} and 5^{th} parameters are the number of rows and columns respectively in the texture image. These **must** be a power of 2.
- The 6^{th} parameter is the border width. This is either 0 or 1. A border width of 1 can be used to create a smooth texture map when using filtering.
- The 7^{th} and 8^{th} parameters specify the type and format of the texels.

Texture Maps (2)

- The 3^{rd} parameter specifies the format of the texture image. There are several image formats, the most common being `GL_RGB`, `GL_RGBA` and `GL_LUMINANCE`.
- The 4^{th} and 5^{th} parameters are the number of rows and columns respectively in the texture image. These **must** be a power of 2.
- The 6^{th} parameter is the border width. This is either 0 or 1. A border width of 1 can be used to create a smooth texture map when using filtering.
- The 7^{th} and 8^{th} parameters specify the type and format of the texels.
- The 9^{th} parameter is the texture image.

Texture Coordinates

Texture coordinates can be 1-D, 2-D, 3-D and 4-D (used internally). In 4-D the coordinates are specified by (s, t, r, q). They can be given as shorts, integers, floats or doubles. We will focus on 2-D floats (the other functions have a similar format).

```
void glTexCoord2f (float scoord, float tcoord)
void glTexCoord2v (float *coord)
```

Texture Coordinates

Texture coordinates can be 1-D, 2-D, 3-D and 4-D (used internally). In 4-D the coordinates are specified by (s, t, r, q). They can be given as shorts, integers, floats or doubles. We will focus on 2-D floats (the other functions have a similar format).

```
void glTexCoord2f (float scoord, float tcoord)
void glTexCoord2v (float *coord)
```

A typical usage could be:

```
glBegin (GL_QUADS);
  glTexCoord2f (0.0, 0.0);
  glVertex2fv (vertex [0]);
  glTexCoord2f (0.0, 1.0);
  glVertex2fv (vertex [1]);
  glTexCoord2f (1.0, 1.0);
  glVertex2fv (vertex [2]);
  glTexCoord2f (1.0, 0.0);
  glVertex2fv (vertex [3]);
glEnd ();
```

Texture Parameters

While `glTexImage*` and `glTextCoord*` are the main texture functions, there is a range of optional parameters that can be used to "fine control" the texturing process. These texture parameters are set using:

```
glTexParameteri (GLenum target, GLenum name, int
value)
glTexParameterf (GLenum target, GLenum name, float
value)
glTexParameteriv (GLenum target, GLenum name, int
*value)
glTexParameterf v(GLenum target, GLenum name, float
*value)
```

where **target** is either `GL_TEXTURE_1D`, `GL_TEXTURE_2D` or `GL_TEXTURE_3D`, `name` is the parameter being set to `value`.

Texture Parameters (2)

The following are some of the texture parameters.

Texture Parameters (2)

The following are some of the texture parameters.

`GL_TEXTURE_WRAP*`

Texture wrapping specifies how to handle situations where the texture coordinates are > 1.0. There are 2 options, `GL_REPEAT` that uses the fractional part for +ve numbers or adds the smallest integer to make it positive; or `GL_CLAMP` that uses the 1.0 value for all texture coordinates > 1.0 or the 0.0 values for all texture coordinates < 0.0.

```
glTexParamter (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParamter (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

Texture Parameters (3)

`GL_TEXTURE_MAG_FILTER` and `GL_TEXTURE_MAG_FILTER`

Depending on the texture coordinates, the size of the surface and the viewing angle, sometime a texel will map to several pixels (**magnification**) or each pixel will map to several pixels (**minification**). OpenGL provides 2 possible ways to handle both these cases. **Point sampling** maps the centre of the pixel to a point in texture space. This is fast and efficient and is called `GL_NEAREST`. **Linear filtering** performs an averaging over the group of pixels or texels. This avoids the artefacts that can occur with **point sampling** and is called `GL_LINEAR`.

```
glTexParamter (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParamter (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

Texture Example

```
#include <stdlib.h>
#include <GL/glut.h>


GLfloat vertices [][3] = {{-1.0, -1.0,1.0}, {-1.0, 1.0, 1.0},
                          {1.0, 1.0, 1.0}, {1.0, -1.0, 1.0},
                          {-1.0, -1.0, -1.0}, {-1.0, 1.0, -1.0},
                          {1.0, 1.0, -1.0}, {1.0, -1.0, -1.0}
};

GLfloat colours [][3] = {{1.0, 0.0, 0.0}, {0.0, 1.0, 1.0},
                         {1.0, 1.0, 0.0}, {0.0, 1.0, 0.0},
                         {0.0, 0.0, 1.0}, {1.0, 0.0, 1.0},
                         {0.0, 0.0, 0.0}, {1.0, 1.0, 1.0}
};
```

Texture Example (2)

```
void polygon (int a, int b, int c, int d)
{
  glBegin(GL_POLYGON);
    glColor3fv (colours [a]);
    glTexCoord2f (0.0, 0.0);
    glVertex3fv (vertices [a]);
    glColor3fv (colours [b]);
    glTexCoord2f (0.0, 1.0);
    glVertex3fv (vertices [b]);
    glColor3fv (colours [c]);
    glTexCoord2f (1.0, 1.0);
    glVertex3fv (vertices [c]);
    glColor3fv (colours [d]);
    glTexCoord2f (1.0, 0.0);
    glVertex3fv (vertices [d]);
  glEnd();
}
```

Texture Example (3)

```
void colour_cube ()
{
  // map vertices to faces
  polygon (0, 3, 2, 1);
  polygon (2, 3, 7, 6);
  polygon (3, 0, 4, 7);
  polygon (1, 2, 6, 5);
  polygon (4, 5, 6, 7);
  polygon (5, 4, 0, 1);
}

static GLfloat theta [] = {0.0, 0.0, 0.0};
static GLint axis = 2;

void display ()
{
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity ();
  glRotatef (theta[0], 1.0, 0.0, 0.0);
  glRotatef (theta[1], 0.0, 1.0, 0.0);
  glRotatef (theta[2], 0.0, 0.0, 1.0);
  colour_cube ();
  glutSwapBuffers ();
}
```

Texture Example (4)

```
void spin_cube ()
{
  theta[axis] += 2.0;
  if (theta[axis] > 360.0)
    theta[axis] -= 360.0;
  glutPostRedisplay ();
}

void mouse (int btn, int state, int x, int y)
{
  if (btn == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    axis = 0;
  if (btn == GLUT_MIDDLE_BUTTON && state == GLUT_DOWN)
    axis = 1;
  if (btn == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
    axis = 2;
}
```

Texture Example (5)

```
void mykey (int key)
{
  if (key == 'Q' || key == 'q')
    exit (0);
}

void reshape (int w, int h)
{
  glViewport (0, 0, w, h);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  if (w <= h)
    glOrtho (-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w, 2.0 * (G
  else
    glOrtho (-2.0 * (GLfloat) w / (GLfloat) h, 2.0 * (GLfloat) w /
  glMatrixMode (GL_MODELVIEW);
}
```

Texture Example (6)

```
int main (int argc, char **argv)
{
  GLubyte image [64][64][3];
  int i,j,r,c;

  // set up image

  for (i = 0; i < 64; i++)
  {
    for (j = 0; j < 64; j++)
    {
      c = ((((i & 0x08) == 0) ^ ((j & 0x08)) == 0)) * 255;
      image [i][j][0] = (GLubyte) c;
      image [i][j][1] = (GLubyte) c;
      image [i][j][2] = (GLubyte) c;
    }
  }

  glutInit (&argc, argv);
  glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize (500, 500);
  glutCreateWindow ("texture cube");
```

Texture Example (7)

```
    glutReshapeFunc (reshape);
    glutDisplayFunc (display);
    glutIdleFunc (spin_cube);
    glutMouseFunc (mouse);
    glutKeyboardFunc (mykey);

    glEnable (GL_DEPTH_TEST);
    glEnable (GL_TEXTURE_2D);

    glTexImage2D (GL_TEXTURE_2D, 0, GL_RGB, 64, 64, 0, GL_RGB,
                  GL_UNSIGNED_BYTE, image);
    glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                     GL_NEAREST);
    glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                     GL_NEAREST);

    glutMainLoop ();
}
```