CA417 - Computer Graphics

OpenGL - Part 2

David Sinclair

Callbacks

OpenGL is structured such that:

- the user sets up the graphics environment,

Callbacks

OpenGL is structured such that:

- the user sets up the graphics environment,
- enter the OpenGL main loop, `glutMainLoop`, and

Callbacks

OpenGL is structured such that:

- the user sets up the graphics environment,
- enter the OpenGL main loop, `glutMainLoop`, and
- processes any events that may occur by associating a function with each type of event.

Callbacks

OpenGL is structured such that:

- the user sets up the graphics environment,
- enter the OpenGL main loop, `glutMainLoop`, and
- processes any events that may occur by associating a function with each type of event.

The functions that are invoked to process each event that occurs in `glutMainLoop` are called **callback functions**.

Callbacks

OpenGL is structured such that:

- the user sets up the graphics environment,
- enter the OpenGL main loop, `glutMainLoop`, and
- processes any events that may occur by associating a function with each type of event.

The functions that are invoked to process each event that occurs in `glutMainLoop` are called **callback functions**.

In order for OpenGL to know which **callback function** is associated with each event type, a **callback function** needs to be **registered** with OpenGL by calling the appropriate OpenGL function.

Reshape Callback

The **reshape callback** is invoked whenever a window is resized. Each window can have its own **reshape callback** function.

Reshape Callback

The **reshape callback** is invoked whenever a window is resized. Each window can have its own **reshape callback** function.

The **reshape callback** for the current window is register by:

```
void glutReshapeFunc (void (*func) (int width, int
height))
```

The `width` and `height` parameters specify the wither and height of the new window in pixels.

Reshape Callback

The **reshape callback** is invoked whenever a window is resized. Each window can have its own **reshape callback** function.

The **reshape callback** for the current window is register by:

```
void glutReshapeFunc (void (*func) (int width, int height))
```

The `width` and `height` parameters specify the wither and height of the new window in pixels.

If the **reshape callback** is not registered or `NULL` is passed to `glutReshapeFunc` then the default **reshape callback** is used (which is `glViewport (0,0,width, height)`).

Reshape Callback (2)

The **reshape callback** is invoked when a window is created. The following **reshape callback** will ensure that in a resized window the shortest side will be at least 4 units in length and that aspect ratio will be maintained so that the relative shapes of the objects will not be distorted.

```
GLsizei ww, hh;        // globals to record the window's
                       // width and height

void myReshape (GLsizei w, GLsizei h)
{
  // adjust clipping window
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  if (w <= h)
    gluOrtho2D (-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w,
                            2.0 * (GLfloat) h / (GLfloat) w);
}
```

Reshape Callback (3)

```
  else
    gluOrtho2D (-2.0 * (GLfloat) w / (GLfloat) h,
                 2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0);

  glMatrixMode (GL_MODELVIEW);      // good manners!

  glViewport (0,0,w,h);             // adjust viewport

  ww = w;            // store new width and height
  hh = h;            // so other drawing functions can use them

  return;
}
```

Idle Callback

The **idle callback** function is called when ever the OpenGL event queue is empty. It is a global callback and is not associated with any particular window. It is registered with the OpenGL system by:

```
void glutIdleFunc (void (*f) (void))
```

Idle Callback

The **idle callback** function is called when ever the OpenGL event queue is empty. It is a global callback and is not associated with any particular window. It is registered with the OpenGL system by:

```
void glutIdleFunc (void (*f) (void))
```

```
int main ()
{
  glutIdelFunc (myIdle);
  ...
}
...
void myIdle ()
{
  glutPostRedisplay ();

  return;
}
```

`glutPostRedisplay` postpones the **display callback** until after the current callback is executed. The **display callback** then get execute once and not multiple times.

Keyboard Callback

The OpenGL function

```
void glutKeyboardFunc (void (*f) (unsigned char key,
int x, int y))
```

registers the function `f` as the **keyboard callback** for the current window. When the **keyboard callback** is invoked it is passed the `key` that was pressed and the `x` and `y` position (in pixels from the upper-left) of the mouse in the window when the `key` was pressed.

Keyboard Callback

The OpenGL function

```
void glutKeyboardFunc (void (*f) (unsigned char key,
int x, int y))
```

registers the function `f` as the **keyboard callback** for the current window. When the **keyboard callback** is invoked it is passed the `key` that was pressed and the `x` and `y` position (in pixels from the upper-left) of the mouse in the window when the `key` was pressed.

OpenGL has a separate callback for *special keys*.

```
void glutSpecialFunc (void (*f) (int key, int x,
                                 int y))
```

Special keys are defined in `glut.h`, e.g. `GLUT_KEY_F1`, `GLUT_KEY_UP` for the F1 key and Up Arrow key respectively.

Keyboard Callback (2)

The **keyboard callback**, the **special keys callback** and the **mouse callback** can access the state of the **modifier keys** using the `int glutGetModifiers ()` function. `glutGetModifiers` returns `GLUT_ACTIVE_SHIFT`, `GLUT_ACTIVE_CTRL` or `GLUT_ACTIVE_ALT` if the *shift*, *control* or *alt* key is pressed when the keyboard or mouse event was generated.

```
if (glutGetModifiers () == GLUT_ACTIVE_CTRL
    && (key == 'q') || (key == 'Q'))
  exit (0);
```

Mouse Callback

The **mouse callback** is registered with OpenGL by:

```
void glutMouseFunc (void (*f) (int button, int state,
                                  int x, int y))
```

where the **mouse callback** `f()` is passed the `button` that was pressed, the `state` of that button (`GLUT_UP` or `GLUT_DOWN`) and the `x` and `y` position in the window when the button was pressed.

Mouse Callback

The **mouse callback** is registered with OpenGL by:

```
void glutMouseFunc (void (*f) (int button, int state,
                                int x, int y))
```

where the **mouse callback** `f()` is passed the `button` that was pressed, the `state` of that button (`GLUT_UP` or `GLUT_DOWN`) and the `x` and `y` position in the window when the button was pressed.

The follow code uses the left button to exit and the right button to define a rectangle using two successive clicks. *Note* the inversion of the screen coordinates. The **mouse callback** uses screen coordinates (origin in upper-left) whereas the drawing functions use world coordinates (origin in lower-left).

Mouse Callback (2)

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

typedef int bool;
#define false 0
#define true !false

GLint x1, x2, y1, y2;
int hh, ww;  // global record of window height (Reshape callback)

void myMouse (int button, int state, int x, int y)
{
  static bool first = true;

  if (state == GLUT_DOWN && button == GLUT_LEFT_BUTTON)
    exit (0);

  if (state == GLUT_DOWN && button == GLUT_RIGHT_BUTTON)
  {
    if (first)
    {
      x1 = x; y1 = hh-y;
    }
```

Mouse Callback (3)

```
      else
      {
        x2 = x; y2 = hh - y;
        glutPostRedisplay ();
      }
      first = !first;
    }
}


void myDisplay ()
{
  glClear (GL_COLOR_BUFFER_BIT);

  glBegin (GL_POLYGON);
    glVertex2i (x1, y1);
    glVertex2i (x1, y2);
    glVertex2i (x2, y2);
    glVertex2i (x2, y1);
  glEnd ();

  glFlush ();
}
```

Mouse Callback (4)

```
void myReshape (int w, int h)
{
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  gluOrtho2D (0.0, (GLfloat) w, 0.0, (GLfloat) h);
  glMatrixMode (GL_MODELVIEW);
  glViewport (0, 0, w, h);
  hh = h;
  ww = w;
}


void init ()
{
  glClearColor (0.0,0.0,0.0,0.0);
  glColor3f (1.0,1.0,1.0);

  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  gluOrtho2D (-1.0, 1.0, -1.0, 1.0);
}
```

Mouse Callback (5)

```
int main (int argc, char **argv)
{
  glutInit (&argc, argv);
  glutCreateWindow ("mousesquare");
  glutReshapeFunc (myReshape);
  glutDisplayFunc (myDisplay);
  glutMouseFunc (myMouse);
  init ();

  glutMainLoop ();
}
```

Mouse Motion Callback

There are 3 mouse motion related callback functions.

Mouse Motion Callback

There are 3 mouse motion related callback functions.

Whenever the mouse enter or leaves a window there is an **entry event** that is handled by an **entry callback**.

```
void glutEntryFunc (void (*f) (int state))
```

where `state` is either `GLUT_ENTERED` or `GLUT_LEFT`.

Mouse Motion Callback

There are 3 mouse motion related callback functions.

Whenever the mouse enter or leaves a window there is an **entry event** that is handled by an **entry callback**.

```
void glutEntryFunc (void (*f) (int state))
```

where `state` is either `GLUT_ENTERED` or `GLUT_LEFT`.

If the mouse is moved while a mouse button is pressed there is a **move event** that is handled by a **motion callback**. If the mouse is moved while a mouse button is not pressed there is a **passive move event** that is handled by a **passive motion callback**. These callbacks are registered as follows.

```
void glutMotionFunc (void (*f) (int x, int y))
void glutPassiveMotionFunc (void (*f) (int x, int y))
```

where the `x` and `y` coordinates of the mouse are passed to `f()`.

NULL Callback

Callbacks can be redefined at any time during execution by registering a different callback function to an event.

NULL Callback

Callbacks can be redefined at any time during execution by registering a different callback function to an event.

A callback function can be removed by simply registering a `NULL` callback to the corresponding event.

```
glutIdleFunc (NULL);
```

Menus

GLUT provides **pop-up menus**. Menus are generally created from `main()` or some initialisation function called from `main()`.

Menus

GLUT provides **pop-up menus**. Menus are generally created from `main()` or some initialisation function called from `main()`.

Top level menus are created by:

```
int glutCreateMenu (void (*f) (int value))
```

where `value` is passed into the function `f()`. `glutCreateMenu` returns an unique identifier that is separate from the window identifier. The menu created becomes the current menu. `f()` is the **menu callback** function.

Menus

GLUT provides **pop-up menus**. Menus are generally created from `main()` or some initialisation function called from `main()`.

Top level menus are created by:

```
int glutCreateMenu (void (*f) (int value))
```

where `value` is passed into the function `f()`. `glutCreateMenu` returns an unique identifier that is separate from the window identifier. The menu created becomes the current menu. `f()` is the **menu callback** function.

The current menu can be changed to the menu with identifier `id` by:

```
int glutSetMenu (int id)
```

Menus (2)

Entries can be added to the current menu using:

```
void glutAddMenuEntry (char *name, int value)
```

where the entry displays `name` and passes `value` to the **menu callback** function.

Menus (2)

Entries can be added to the current menu using:

```
void glutAddMenuEntry (char *name, int value)
```

where the entry displays `name` and passes `value` to the **menu callback** function.

Menus can be attached/activated by a specific mouse button using:

```
void glutAttachMenu (int button)
```

where `button` is either `GLUT_RIGHT_BUTTON`, `GLUT_MIDDLE_BUTTON` or `GLUT_LEFT_BUTTON`.

Multiple Windows

`int glutCreateWindow (char *name)`

creates a top-level window with the title `name`. It return a unique identifier for the window. The created window becomes the current window.

Multiple Windows

`int glutCreateWindow (char *name)`

creates a top-level window with the title `name`. It return a unique identifier for the window. The created window becomes the current window.

`void glutDistryWindow (int id)`

destroys a top-level window with identifier `id`.

Multiple Windows

`int glutCreateWindow (char *name)`

creates a top-level window with the title `name`. It return a unique identifier for the window. The created window becomes the current window.

`void glutDistryWindow (int id)`

destroys a top-level window with identifier `id`.

`void glutSetWindow (int id)`

change the current window to the window with identifier `id`.

Multiple Windows

```
int glutCreateWindow (char *name)
```

creates a top-level window with the title `name`. It return a unique identifier for the window. The created window becomes the current window.

```
void glutDistryWindow (int id)
```

destroys a top-level window with identifier `id`.

```
void glutSetWindow (int id)
```

change the current window to the window with identifier `id`.

```
int glutCreateSubWindow (int id, int x, int y,
                              int width, int height)
```

creates a sub-window of `id` at position (x,y) of a specified `width` and `height`.

Multiple Windows

`int glutCreateWindow (char *name)`

creates a top-level window with the title `name`. It return a unique identifier for the window. The created window becomes the current window.

`void glutDistryWindow (int id)`

destroys a top-level window with identifier `id`.

`void glutSetWindow (int id)`

change the current window to the window with identifier `id`.

```
int glutCreateSubWindow (int id, int x, int y,
                         int width, int height)
```

creates a sub-window of `id` at position (x,y) of a specified `width` and `height`.

`void glutPostWindowRedisplay (int id)`

post a redisplay of window `id`.

Double Buffering

Generally graphics displays are **refreshed** at a fixed rate asynchronously to the program that is updating the image. This can result in a "flickering" if the image is refreshed before it is fully drawn/constructed.

Double Buffering

Generally graphics displays are **refreshed** at a fixed rate asynchronously to the program that is updating the image. This can result in a "flickering" if the image is refreshed before it is fully drawn/constructed.

In order to avoid this **double buffering** is used. One buffer, the **front buffer** is displayed (and constantly refreshed) while the program updates the image in another buffer, the **back buffer**. When the image is completely constructed, then the **front** and **back buffers** are swapped using:

```
void glutSwapBuffers ()
```

Callback and Windows Example

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

#define DEG_TO_RAD 0.017453

 int singleb, doubleb;      // window ids
 GLfloat theta = 0.0;


 int main (int argc, char **argv)
{
   glutInit (&argc, argv);

   // create single buffered window
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   singleb = glutCreateWindow ("single buffered");
   glutDisplayFunc (displays);
   glutReshapeFunc (myReshape);
   glutIdleFunc (spinDisplay);
   glutMouseFunc (mouse);
   glutKeyboardFunc (mykey);
```

Callback and Windows Example (2)

```
    // create double buffered window
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowPosition (400,0);  //create window to the right
    doubleb = glutCreateWindow ("double buffered");
    glutDisplayFunc (displayd);
    glutReshapeFunc (myReshape);
    glutIdleFunc (spinDisplay);
    glutMouseFunc (mouse);
    glutCreateMenu (quit_menu);
    glutAddMenuEntry ("quit", 1);
    glutAttachMenu (GLUT_RIGHT_BUTTON);

    glutMainLoop ();
}
```

Callback and Windows Example (3)

```
void displays ()
{
  glClear (GL_COLOR_BUFFER_BIT);
  glBegin(GL_POLYGON);
    glVertex2f (cos (DEG_TO_RAD * theta),
                sin (DEG_TO_RAD * theta));
    glVertex2f (-sin (DEG_TO_RAD * theta),
                cos (DEG_TO_RAD * theta));
    glVertex2f (-cos (DEG_TO_RAD * theta),
                -sin (DEG_TO_RAD * theta));
    glVertex2f (sin (DEG_TO_RAD * theta),
                -cos (DEG_TO_RAD * theta));
  glEnd ();
  glFlush ();
}
```

Callback and Windows Example (4)

```
void displayd ()
{
  glClear (GL_COLOR_BUFFER_BIT);
  glBegin(GL_POLYGON);
    glVertex2f (cos (DEG_TO_RAD * theta),
                sin (DEG_TO_RAD * theta));
    glVertex2f (-sin (DEG_TO_RAD * theta),
                cos (DEG_TO_RAD * theta));
    glVertex2f (-cos (DEG_TO_RAD * theta),
                -sin (DEG_TO_RAD * theta));
    glVertex2f (sin (DEG_TO_RAD * theta),
                -cos (DEG_TO_RAD * theta));
  glEnd ();
  glutSwapBuffers ();

  return;
}
```

Callback and Windows Example (5)

```
void display_stall ()
{
  glClear (GL_COLOR_BUFFER_BIT);
  glBegin(GL_POLYGON);
    glVertex2f (1.0, 0.0);
    glVertex2f (0.0, 1.0);
    glVertex2f (-1.0, 0.0);
    glVertex2f (0.0, -1.0);
  glEnd ();
  glFlush ();
}


void spinDisplay (void)
{
  theta += 2.0;
  if (theta > 360.0)
    theta -= 360.0;

  // draw single buffered window
  glutSetWindow (singleb);
  glutPostWindowRedisplay (singleb);
```

Callback and Windows Example (6)

```
  // draw double buffered window
  glutSetWindow (doubleb);
  glutPostWindowRedisplay (doubleb);

  return;
}


void mouse (int button, int state, int x, int y)
{
  if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    if (glutGetWindow () == 1)
      glutDisplayFunc (displays);
    else
      glutDisplayFunc (displayd);

  if (button == GLUT_MIDDLE_BUTTON && state == GLUT_DOWN)
    glutDisplayFunc (display_stall);

  return;
}
```

Callback and Windows Example (7)

```
void myReshape (int w, int h)
{
  glViewport (0, 0, w, h);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  gluOrtho2D (-2.0, 2.0, -2.0, 2.0);
  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity ();

  return;
}

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

void quit_menu (int id)
{
  if (id == 1) exit (0);
  return;
}
```

Cameras

The basic model used in OpenGL is the **synthetic camera model**. The image produced by a set of objects, a camera and a **projection plane** results from a set of **projection rays** that start at the camera (**centre of projection**) and end at the vertices that define the object. Where these **projection rays** intersect the view plane is where the corresponding vertices appear on the projection plane.

Cameras

The basic model used in OpenGL is the **synthetic camera model**. The image produced by a set of objects, a camera and a **projection plane** results from a set of **projection rays** that start at the camera (**centre of projection**) and end at the vertices that define the object. Where these **projection rays** intersect the view plane is where the corresponding vertices appear on the projection plane.

In **orthographic projection** the **projection rays** are parallel to the direction from the camera to the objects. The view volume is defined by a rectangular box.

Cameras

The basic model used in OpenGL is the **synthetic camera model**. The image produced by a set of objects, a camera and a **projection plane** results from a set of **projection rays** that start at the camera (**centre of projection**) and end at the vertices that define the object. Where these **projection rays** intersect the view plane is where the corresponding vertices appear on the projection plane.

In **orthographic projection** the **projection rays** are parallel to the direction from the camera to the objects. The view volume is defined by a rectangular box.

In **perspective projection** the **projection rays** emanate from the camera to the objects. The view volume is defined by a truncated pyramid called a **frustum**.

Orthographic Projections

The projection matrix that defines an orthographic projection is set up by the `glOrtho` function.

```
void glOrtho (GLdouble left, GLdouble right,
              GLdouble bottom, GLdouble top
              GLdoube near, GLdouble far)
```

where distances are measured from the camera and `right` > `left`, `top` > `bottom`, `far` > `near`.

Orthographic Projections

The projection matrix that defines an orthographic projection is set up by the `glOrtho` function.

```
void glOrtho (GLdouble left, GLdouble right,
              GLdouble bottom, GLdouble top
              GLdoube near, GLdouble far)
```

where distances are measured from the camera and `right` > `left`, `top` > `bottom`, `far` > `near`.

As `glOrtho` modifies the projection matrix, it usually modifies an identity matrix.

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluOrtho (-2.0, 2.0, -2.0, 2.0, -2.0, 2.0);
```

Locating the Camera

If we want to place the camera at any point other than the origin and orient it in a particular direction we can use the `gluLookAt` function to modify the `GL_MODELVIEW` matrix. To use this function we need to know where the camera (**eye point**) is located, what direction defines "up" in the image (**up vector**) and the point the camera is looking at (**at point**).

```
void gluLookAt (GLdouble eyex, GLdouble eyey,
                GLdouble eyez, GLdouble atx,
                GLdouble aty, GLdouble atz,
                GLdouble upx, GLdouble upy,
                GLdouble upz)
```

Simple Cube Program

```
#include <GLUT/glut.h>

void display (void)
{
  glClear (GL_COLOR_BUFFER_BIT);
  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity ();
  gluLookAt (1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
  glutWireCube (0.5);
  glutSwapBuffers ();

  return;
}


void reshape (int w, int h)
{
  glViewport (0, 0, w, h);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  glOrtho (-2.0, 2.0, -2.0, 2.0, -2.0, 2.0);
}
```

Simple Cube Program (2)

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

  return;
}


void init ()
{
  glClearColor (1.0, 1.0, 1.0, 1.0);
  glColor3f (0.0, 0.0, 0.0);
}
```

Simple Cube Program (3)

```
int main (int argc, char **argv)
{

  glutInit (&argc, argv);
  glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
  glutInitWindowSize (500, 500);
  glutInitWindowPosition (0, 0);
  glutCreateWindow ("cube");
  glutReshapeFunc (reshape);
  glutDisplayFunc (display);
  glutKeyboardFunc (mykey);
  init ();
  glutMainLoop ();
}
```

Perspective Projections

The projection matrix that defines a perspective projection is set up by the `glFrustum` function.

```
void glFrustum (GLdouble left, GLdouble right,
                GLdouble bottom, GLdouble top
                GLdoube near, GLdouble far)
```

where distances are measured from the camera at the origin pointing down the $-z$ axis and `right` $>$ `left`, `top` $>$ `bottom`, `far` $>$ `near`. The viewing frustum must be in front of the camera and the near plane is the projection plane.

Perspective Projections

The projection matrix that defines a perspective projection is set up by the `glFrustum` function.

```
void glFrustum (GLdouble left, GLdouble right,
               GLdouble bottom, GLdouble top
               GLdoube near, GLdouble far)
```

where distances are measured from the camera at the origin pointing down the $-z$ axis and `right` $>$ `left`, `top` $>$ `bottom`, `far` $>$ `near`. The viewing frustum must be in front of the camera and the near plane is the projection plane.

As `glFrustum` modifies the projection matrix, it usually modifies an identity matrix.

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
glFustrum (-2.0, 2.0, -2.0, 2.0, 1.0, 4.0);
```

Perspective Projections (2)

As with orthographic projection we can move the view point using `gluLookAt` function.

Perspective Projections (2)

As with orthographic projection we can move the view point using `gluLookAt` function.

The `glFrustum` function provides a general interface but can be difficult to use. As we change the `near` and `far` planes, the angles of the sides can change significantly and object that are near the sides can suddenly “fall outside” the frustum.

Perspective Projections (2)

As with orthographic projection we can move the view point using `gluLookAt` function.

The `glFrustum` function provides a general interface but can be difficult to use. As we change the `near` and `far` planes, the angles of the sides can change significantly and object that are near the sides can suddenly "fall outside" the frustum.

`gluPerspective` provides a more natural, but less general, way to specify the frustum.

```
void gluPerspective (GLdouble fov, GLdouble aspect,
                     GLdouble near, GLdouble far)
```

where `fov` is the angle of view in the y-direction and `aspect` is the width:height ratio of the `near` and `far` planes.

3D Objects

3-dimensional objects are specified by vertices in 3-dimensions. There are a few things we need to be careful about!

3D Objects

3-dimensional objects are specified by vertices in 3-dimensions. There are a few things we need to be careful about!

- 3 vertices, as long as they are not collinear, define a plane as well as the interior of a triangle.

3D Objects

3-dimensional objects are specified by vertices in 3-dimensions. There are a few things we need to be careful about!

- 3 vertices, as long as they are not collinear, define a plane as well as the interior of a triangle.
- When a polygon is defined by more then 3 vertices not all the vertices may lie in the same plane. This is not a mathematical correct polygon. OpenGL does not check if the vertices of a polygon lie in the same plane and will tessellate the vertices into polygon**s**.

3D Objects

3-dimensional objects are specified by vertices in 3-dimensions. There are a few things we need to be careful about!

- 3 vertices, as long as they are not collinear, define a plane as well as the interior of a triangle.
- When a polygon is defined by more then 3 vertices not all the vertices may lie in the same plane. This is not a mathematical correct polygon. OpenGL does not check if the vertices of a polygon lie in the same plane and will tessellate the vertices into polygon**s**.
- The order the vertices are specified in defines the front and back surfaces of a polygon. If the vertices are specified in counter-clockwise order, then we are look at the front surface (the surface normal is facing us).

Using Arrays to Build Objects

We could build a cube by specifying all 6 faces and the 4 vertices that specify each face.

```
void cube ()
{
  glColor (1.0, 0.0, 0.0);
  glBegin(GL_POLYGON);
    glVertex3f (-1.0, -1.0, -1.0);
    glVertex3f (-1.0, 1.0, -1.0);
    glVertex3f (-1.0, 1.0, 1.0);
    glVertex3f (-1.0, -1.0, 1.0);
  glEnd ();

  // and so on for the other 5 faces
}
```

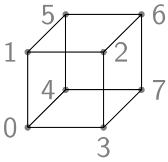
Not very neat! 42 function calls!

Using Arrays to Build Objects (2)

We can use arrays to “tidy up” the code. It doesn’t reduce the number of function calls but it does make it easier to read.

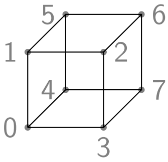
Using Arrays to Build Objects (2)

We can use arrays to "tidy up" the code. It doesn't reduce the number of function calls but it does make it easier to read. Let's number the vertices of a cube as follows.



Using Arrays to Build Objects (2)

We can use arrays to "tidy up" the code. It doesn't reduce the number of function calls but it does make it easier to read. Let's number the vertices of a cube as follows.




```
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},{01.0, 0.0, 1.0},
  };
```

Using Arrays to Build Objects (3)

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

void cube ()
{
  glColor3fv (colours [0]); polygon (0,3,2,1);
  glColor3fv (colours [2]); polygon (2,3,7,6);
  glColor3fv (colours [3]); polygon (3,0,4,7);
  glColor3fv (colours [0]); polygon (1,2,6,5);
  glColor3fv (colours [4]); polygon (4,5,6,7);
  glColor3fv (colours [5]); polygon (5,4,0,1);
}
```

Using Arrays to Build Objects (4)

This is such a useful idea that OpenGL has a built-in facility called **vertex arrays** that implements this idea and avoids most of the function calls.

Using Arrays to Build Objects (4)

This is such a useful idea that OpenGL has a built-in facility called **vertex arrays** that implements this idea and avoids most of the function calls.

OpenGL support 6 different types of **arrays**

- `GL_VERTEX_ARRAY`
- `GL_COLOR_ARRAY`
- `GL_INDEX_ARRAY`
- `GL_NORMAL_ARRAY`
- `GL_TEXTURE_COORD_ARRAY`
- `GL_EDGE_FLAG_ARRAY`

Using Arrays to Build Objects (4)

This is such a useful idea that OpenGL has a built-in facility called **vertex arrays** that implements this idea and avoids most of the function calls.

OpenGL support 6 different types of **arrays**

- `GL_VERTEX_ARRAY`
- `GL_COLOR_ARRAY`
- `GL_INDEX_ARRAY`
- `GL_NORMAL_ARRAY`
- `GL_TEXTURE_COORD_ARRAY`
- `GL_EDGE_FLAG_ARRAY`

These need to be enables or disabled respectively using:

```
void glEnableClientState (GLenum array)
void glDisableClientState (GLenum array)
```

Using Arrays to Build Objects (5)

We will focus on just vertices and colour. We then to describe the arrays that will hold the vertex and colour information.

```
void glVertexPointer (GLint dim, GLenum type,
                      GLsizei stride, GLvoid *array)
void glColorPointer (GLint dim, GLenum type,
                     GLsizei stride, GLvoid *array
```

where `dim` is the number of dimensions, `type` is `GL_SHORT`, `GL_INT`, `GL_FLOAT` or `GL_DOUBLE` and `stride` is how many bytes to skip between consecutive values.

Using Arrays to Build Objects (5)

We will focus on just vertices and colour. We then to describe the arrays that will hold the vertex and colour information.

```
void glVertexPointer (GLint dim, GLenum type,
                      GLsizei stride, GLvoid *array)
void glColorPointer (GLint dim, GLenum type,
                     GLsizei stride, GLvoid *array
```

where `dim` is the number of dimensions, `type` is `GL_SHORT`, `GL_INT`, `GL_FLOAT` or `GL_DOUBLE` and `stride` is how many bytes to skip between consecutive values.

We can then use the `glDrawElements` function.

```
void glDrawElements (GLenum mode, GLsizei n,
                     GLenum type, void *v_indices)
```

that draws elements of type `mode` using `n` indices from the array `v_indices` that are `type`.

Using Arrays to Build Objects (6)

```
void cube ()
{
  glEnableClientState (GL_VERTEX_ARRAY);
  glEnableClientState (GL_VERTEX_ARRAY);

  glVertexPointer (3,GL_FLOAT, 0, vertices);
  glColorPointer (3, GL_FLOAT, 0, colours);

  GLubyte cube_indices [] = {0,3,2,1,2,3,7,6,0,4,7,3,
                             1,2,6,5,4,5,6,7,0,1,5,4};

  glDrawElemets (GL_QUADS, 24, GL_UNSIGNED_BYTE, cube_indices);
}
```

Hidden Surface Removal

A **convex object** has the property that if you pick any 2 points on or inside the object, then all the points on a straight line between these 2 points are on or inside the object.

Hidden Surface Removal

A **convex object** has the property that if you pick any 2 points on or inside the object, then all the points on a straight line between these 2 points are on or inside the object.

For **convex objects** we can use **back face culling** for hidden surface removal.

```
glEnable (GL_CULL_FACE);
glCullFace (GL_BACK);
```

Hidden Surface Removal

A **convex object** has the property that if you pick any 2 points on or inside the object, then all the points on a straight line between these 2 points are on or inside the object.

For **convex objects** we can use **back face culling** for hidden surface removal.

```
glEnable (GL_CULL_FACE);
glCullFace (GL_BACK);
```

The more general algorithm that can be used in OpenGL is the **z-buffer algorithm**. This requires extra storage to record the "depth" of each pixel visible in the scene. It is initialised and enabled as follows.

```
glutInitDisplayMode (GLUT_RGB | GLUT_DEPTH);
glEnable (GL_DEPTH_TEST);
```

Hidden Surface Removal

A **convex object** has the property that if you pick any 2 points on or inside the object, then all the points on a straight line between these 2 points are on or inside the object.

For **convex objects** we can use **back face culling** for hidden surface removal.

```
glEnable (GL_CULL_FACE);
glCullFace (GL_BACK);
```

The more general algorithm that can be used in OpenGL is the **z-buffer algorithm**. This requires extra storage to record the "depth" of each pixel visible in the scene. It is initialised and enabled as follows.

```
glutInitDisplayMode (GLUT_RGB | GLUT_DEPTH);
glEnable (GL_DEPTH_TEST);
```

Wen we clear the colour buffer we also need to clear the depth buffer.

```
glClear (CL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

High Level GLU and GLUT Objects

The GLU and GLUT libraries provide high level objects that are polygonal approximations of **quadrics** (cones, cylinders, ellipsoids, spheres, etc.). Because these objects are more complex they require special handling.

High Level GLU and GLUT Objects

The GLU and GLUT libraries provide high level objects that are polygonal approximations of **quadrics** (cones, cylinders, ellipsoids, spheres, etc.). Because these objects are more complex they require special handling.

```
GLUquadricObj *gluNewQuadric ()
```

creates a new quadric object and returns a pointer to it.

High Level GLU and GLUT Objects

The GLU and GLUT libraries provide high level objects that are polygonal approximations of **quadrics** (cones, cylinders, ellipsoids, spheres, etc.). Because these objects are more complex they require special handling.

```
GLUquadricObj *gluNewQuadric ()
```

creates a new quadric object and returns a pointer to it.

```
void gluDeleteQuadric (GLUquadricObj *obj)
```

deletes the quadric pointed to by `obj`.

High Level GLU and GLUT Objects

The GLU and GLUT libraries provide high level objects that are polygonal approximations of **quadrics** (cones, cylinders, ellipsoids, spheres, etc.). Because these objects are more complex they require special handling.

```
GLUquadricObj *gluNewQuadric ()
```

creates a new quadric object and returns a pointer to it.

```
void gluDeleteQuadric (GLUquadricObj *obj)
```

deletes the quadric pointed to by `obj`.

```
void gluQuadricDrawStyle (GLUquadricObj *obj,
                          GLenum style)
```

set the drawing style for `obj` to either `GLU_POINT`, `GLU_LINE`, `GLU_FILL` or `GLU_SILHOUETTE`. **Silhouette mode** draws lines between vertices except when polygons are coplanar.

High Level GLU and GLUT Objects (2)

If we wish to generate normals and texture coordinates for a quadric `obj` we call:

```
void gluQaudricNormals (GLUquadricObj *obj, GLenum mode)
```

sets the normal mode for `obj` to either `GLU_NONE`, `GLU_FLAT` or `GLU_SMOOTH`.

High Level GLU and GLUT Objects (2)

If we wish to generate normals and texture coordinates for a quadric `obj` we call:

```
void gluQaudricNormals (GLUquadricObj *obj, GLenum
mode)
```

sets the normal mode for `obj` to either `GLU_NONE`, `GLU_FLAT` or `GLU_SMOOTH`.

```
void gluQaudricTexture (GLUquadricObj *obj, GLenum
mode)
```

generates texture coordinates for `obj` if `mode` is set to `GLU_TRUE`.

High Level GLU and GLUT Objects (3)

```
void gluSphere (GLUquadricObj *obj, GLdouble radius,
                GLint slices, GLint stacks)
```

where `obj` points to a sphere with a given `radius` and is approximated by `slices` lines of longitude and `stacks` line of latitude.

High Level GLU and GLUT Objects (3)

```
void gluSphere (GLUquadricObj *obj, GLdouble radius,
                GLint slices, GLint stacks)
```

where `obj` points to a sphere with a given `radius` and is approximated by `slices` lines of longitude and `stacks` line of latitude.

```
void gluCylinder (GLUquadricObj *obj, GLdouble base,
                  GLdouble top, GLdouble height,
                  GLint slices, GLint stacks)
```

where `obj` points to a cylinder with a given `top` and `bottom` radius and is approximated by `slices` lines of longitude and `stacks` line of latitude.

High Level GLU and GLUT Objects (4)

```
void gluDisk (GLUquadricObj *obj, GLdouble inner,
              GLdouble outer, GLint slices,
              GLint rings)
```

generates a disk in the $z = 0$ plane with `inner` and `outer` radii and is approximated by a given number of concentric `rings` and a given number of `slices` around the centre.

High Level GLU and GLUT Objects (4)

```
void gluDisk (GLUquadricObj *obj, GLdouble inner,
              GLdouble outer, GLint slices,
              GLint rings)
```

generates a disk in the $z = 0$ plane with `inner` and `outer` radii and is approximated by a given number of concentric `rings` and a given number of `slices` around the centre.

```
void gluPartialDisk (GLUquadricObj *obj,
                     GLdouble inner, GLdouble outer,
                     GLint slices, GLint rings,
                     GLdouble start, GLdouble angle)
```

generates a partial disk that begins at `start` degrees and is a wedge of `angle` degrees.

High Level GLU and GLUT Objects (5)

A typical sequence of GLU functions is:

```
GLUquadricObj *mySphere;
mySphere = gluNewQuadric ();
gluQuadricDrawStyle (mySphere, GLU_LINES);

gluSphere (mySphere, 1.0, 12, 12);
```

High Level GLU and GLUT Objects (5)

A typical sequence of GLU functions is:

```
GLUquadricObj *mySphere;
mySphere = gluNewQuadric ();
gluQuadricDrawStyle (mySphere, GLU_LINES);

gluSphere (mySphere, 1.0, 12, 12);
```

Some high level GLUT objects are:

```
void glutWireSphere (GLdouble radius, GLint slices,
                     GLint stacks)

void glutSolidSphere (GLdouble radius, GLint slices,
                      GLint stacks)
```

High Level GLU and GLUT Objects (6)

```
void glutWireCone (GLdouble base, GLdouble height,
                   GLint slices, GLint stacks)

void glutSolidCone (GLdouble base, GLdouble height,
                    GLint slices, GLint stacks)

void glutWireTorus (GLdouble inner, GLdouble outer,
                    GLint sides, GLint slices)

void glutSolidTorus (GLdouble inner, GLdouble outer,
                     GLint sides, GLint slices)
```