

## 1

## OpenGL Basics

Lights, Camera, Action!

**R. Mukundan** ([mukundan@canterbury.ac.nz](mailto:mukundan@canterbury.ac.nz))

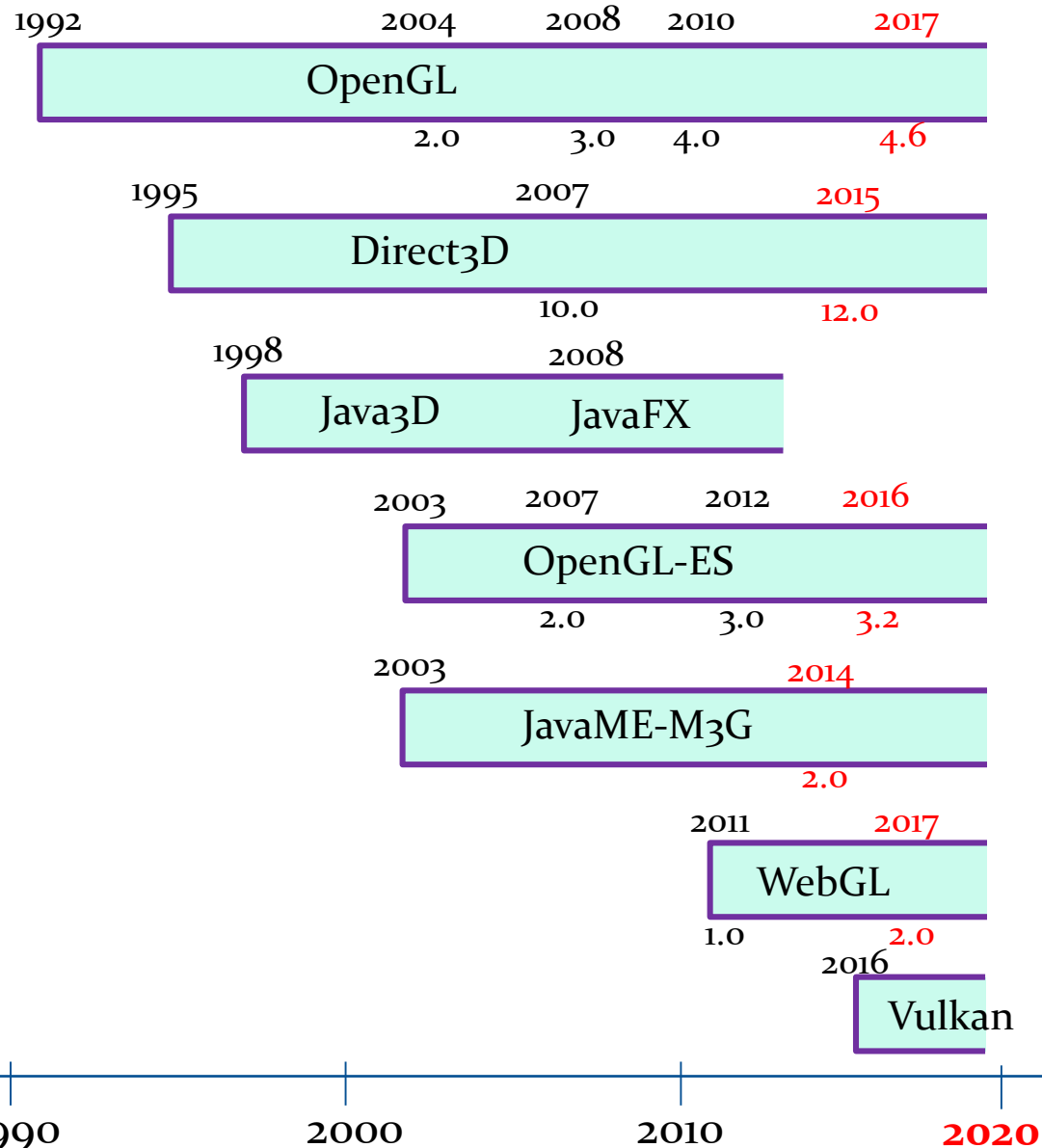
Department of Computer Science and Software Engineering  
University of Canterbury, New Zealand.



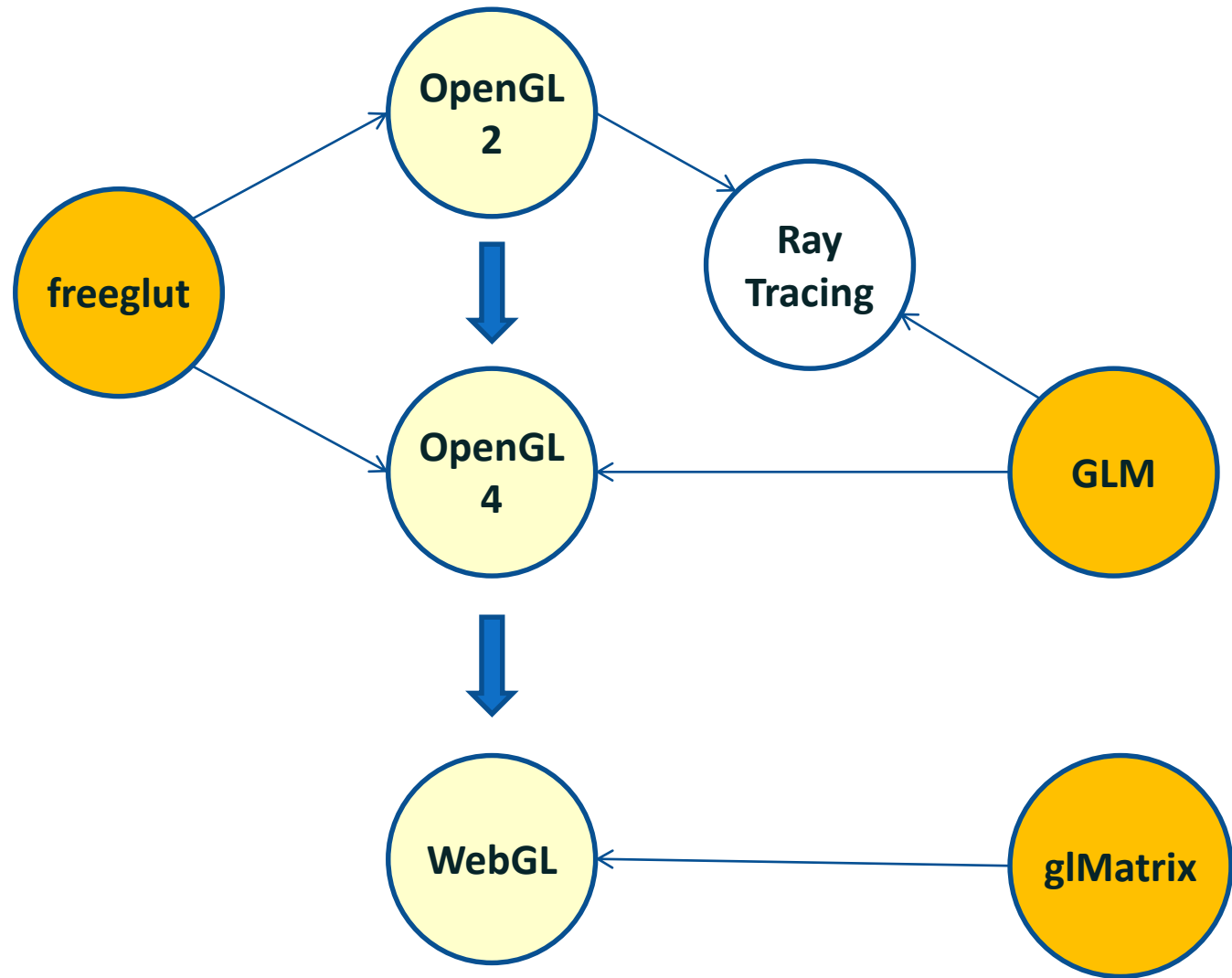
# OpenGL

- A widely popular, platform-independent library of functions for modelling and real-time rendering of three-dimensional scenes.
- A rendering library that provides a low-level programmer's interface with the graphics hardware.
- Does not support sound, video, and networking functionalities.
- API evolution:
  - OpenGL 1.0 released in 1992.
  - A thorough overhaul of the API began in 2007, with the design of OpenGL 3.0 (2008), and OpenGL 4.0 (2010). In version 4, GPU processing is given utmost importance.
  - Current version: 4.6

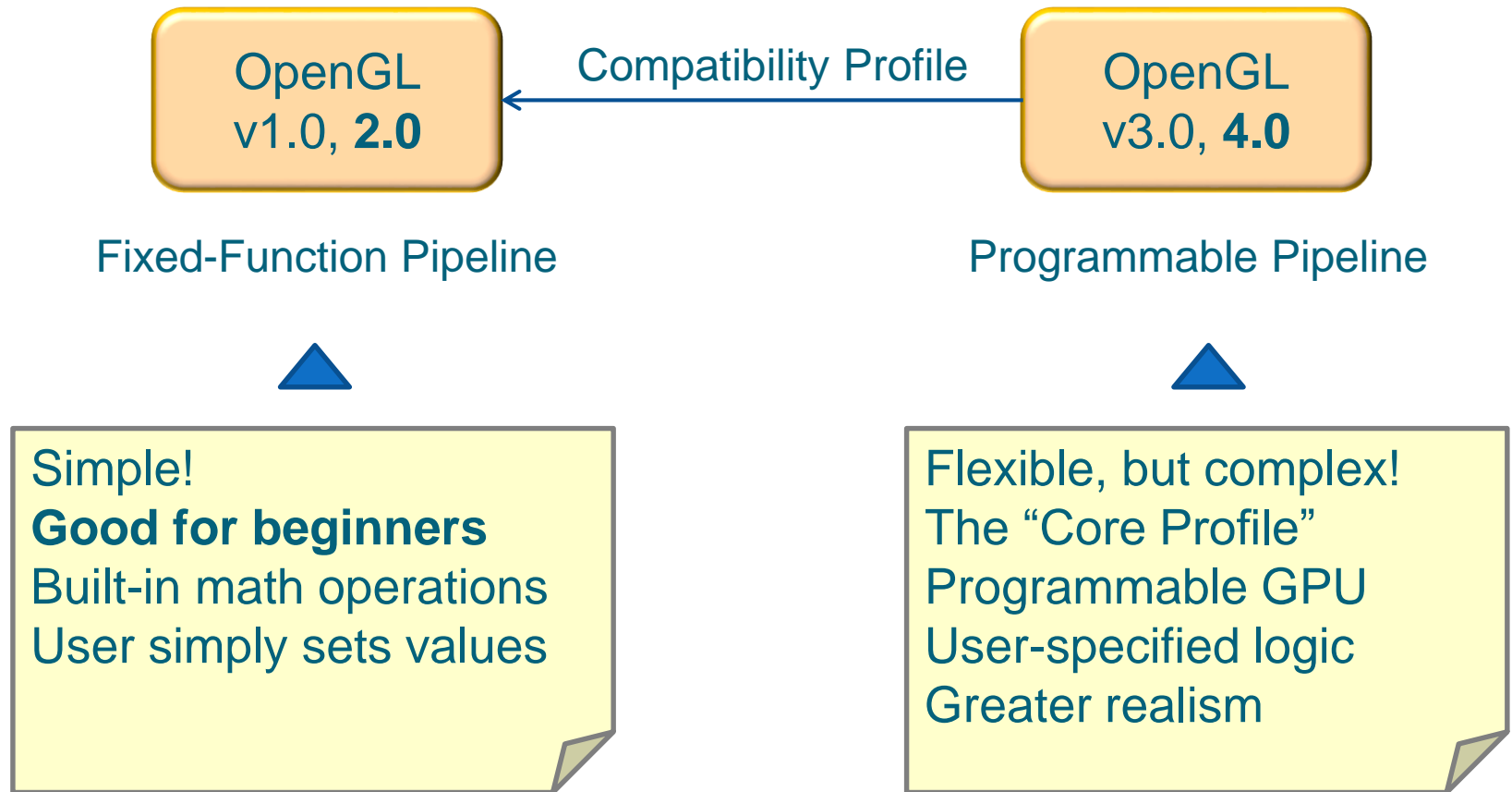
# Evolution of 3D Graphics APIs



# APIs and Libraries Used in the Course



# “Old” and “New” Programming Models



# OpenGL 2 Core Libraries

- OpenGL: Basic API, preloaded into the system.  
glVertex3f, glColor4f, glPushMatrix, glMatrixMode, glLightfv, glTranslatef, glRotatef, glFrustum, glOrtho, ...
- GLU (OpenGL Utility Library): Higher level functions for drawing and viewing.  
gluLookAt, gluPerspective, gluOrtho2D, gluCylinder, ...
- GLUT (OpenGL Utility Toolkit): Platform independent **interface for the native window system**, call-back based **event processing**, and some **built-in geometrical objects**.  
glutCreateWindow, glutMouseFunc, glutSolidTeapot  
<http://freeglut.sourceforge.net/>

# A Basic OpenGL Program

```
#include <GL/freeglut.h>
```

```
void display(void)  
{  
    ...  
}
```

All display  
commands

```
void initialize(void)  
{  
    ...  
}
```

Parameter/state  
initialization

```
int main(...)  
{  
    ...  
}
```

GLUT  
initialization

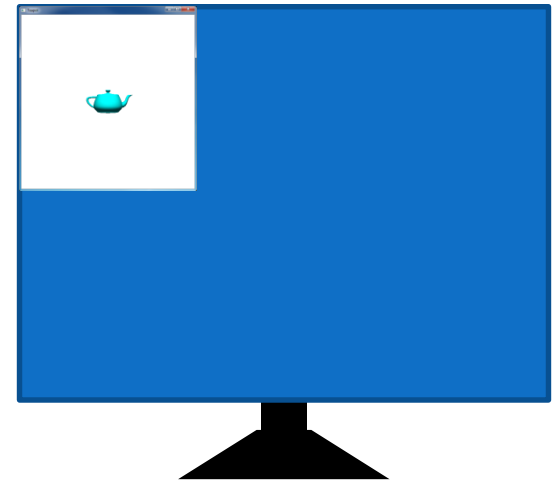
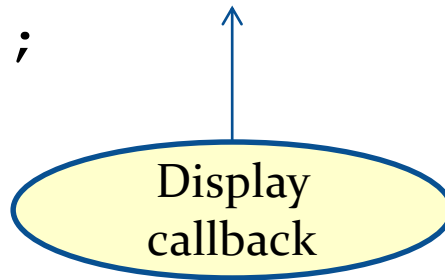
# Teapot.cpp [1]

```
int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_DEPTH);
    glutInitWindowSize(600, 600);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Teapot");
    initialize();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Display buffers



Display  
callback





# Teapot.cpp [2]

```
void initialize(void)
{
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_COLOR_MATERIAL);
    glEnable(GL_DEPTH_TEST);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-5.0, 5.0, -5.0, 5.0, 10.0, 1000.0);
}
```

Background  
colour

OpenGL  
state

Camera

# Teapot.cpp [3]

```
void display(void)
{
    float light_pos[4] = {0., 10., 10., 1.0};
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

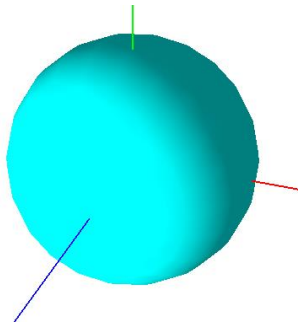
    gluLookAt(0, 0, 12, 0, 0, 0, 0, 1, 0);
    glLightfv(GL_LIGHT0, GL_POSITION, light_pos);

    glColor3f(0.0, 1.0, 1.0);
    glutSolidTeapot(1.0);
    glFlush();
}
```

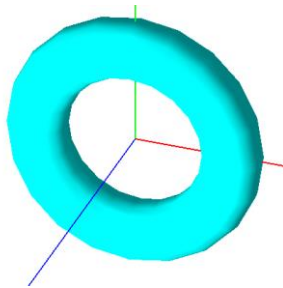
Camera  
Position

Light's  
Position

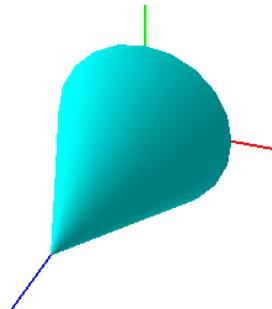
# GLUT Built-in Objects



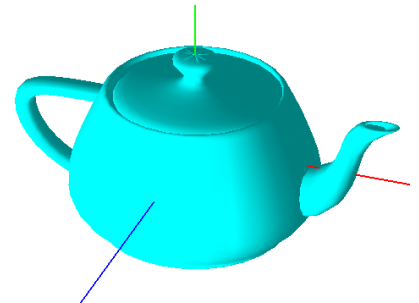
Sphere



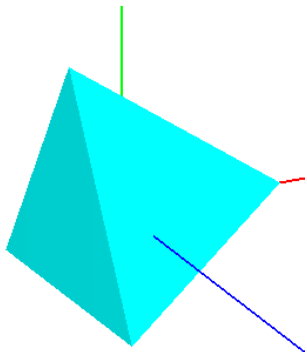
Torus



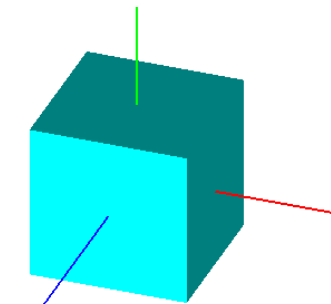
Cone



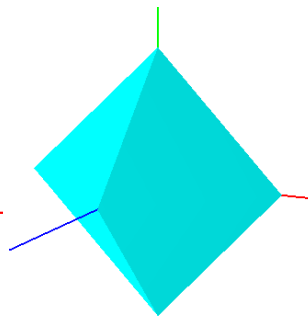
Teapot



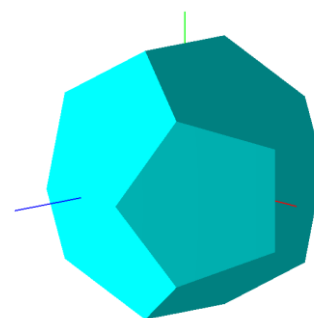
Tetrahedron



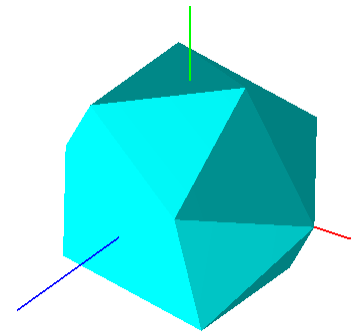
Cube



Octahedron



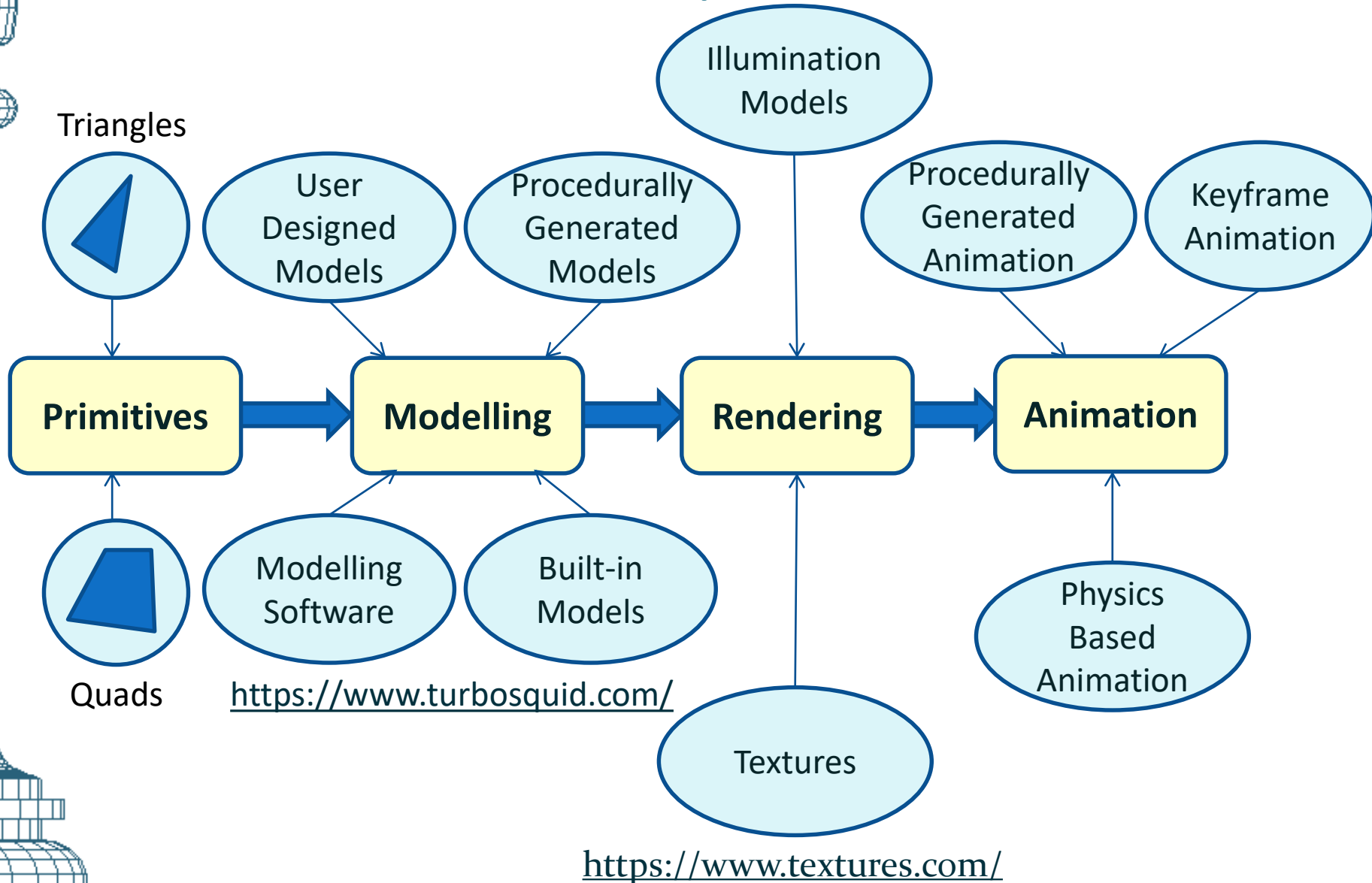
Dodecahedron



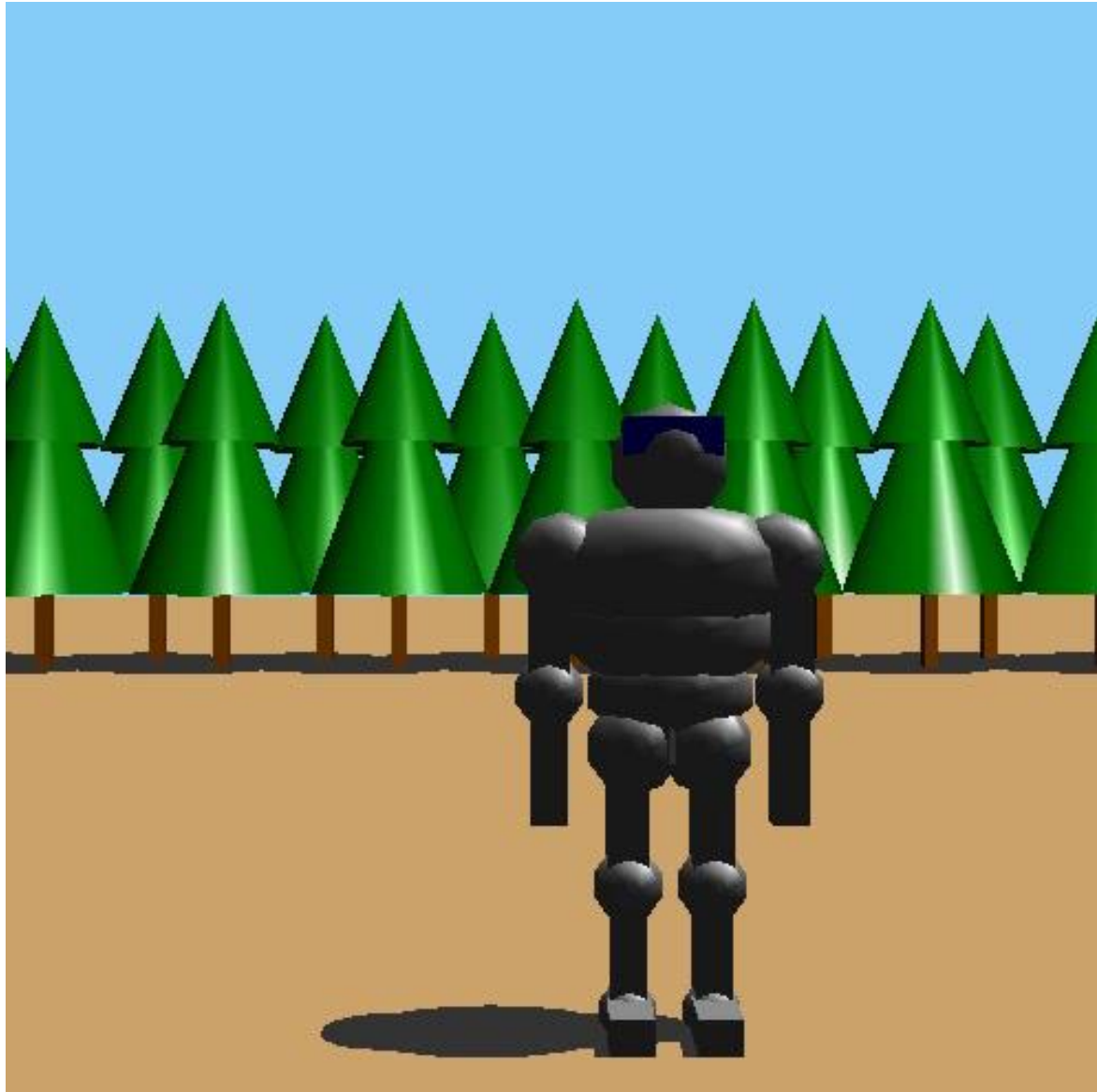
Icosahedron

For a description of the glut functions for rendering these objects, please refer to **GLUT-GLU-Objects.pdf** in section **Lab-01** on Learn

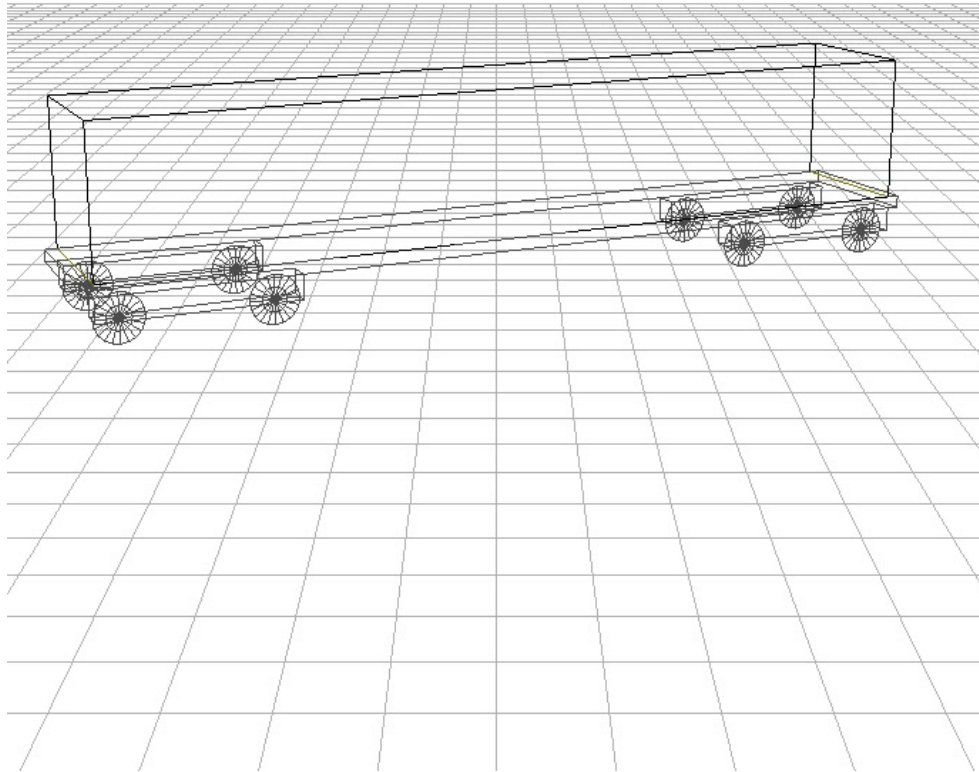
# Basic Pipeline



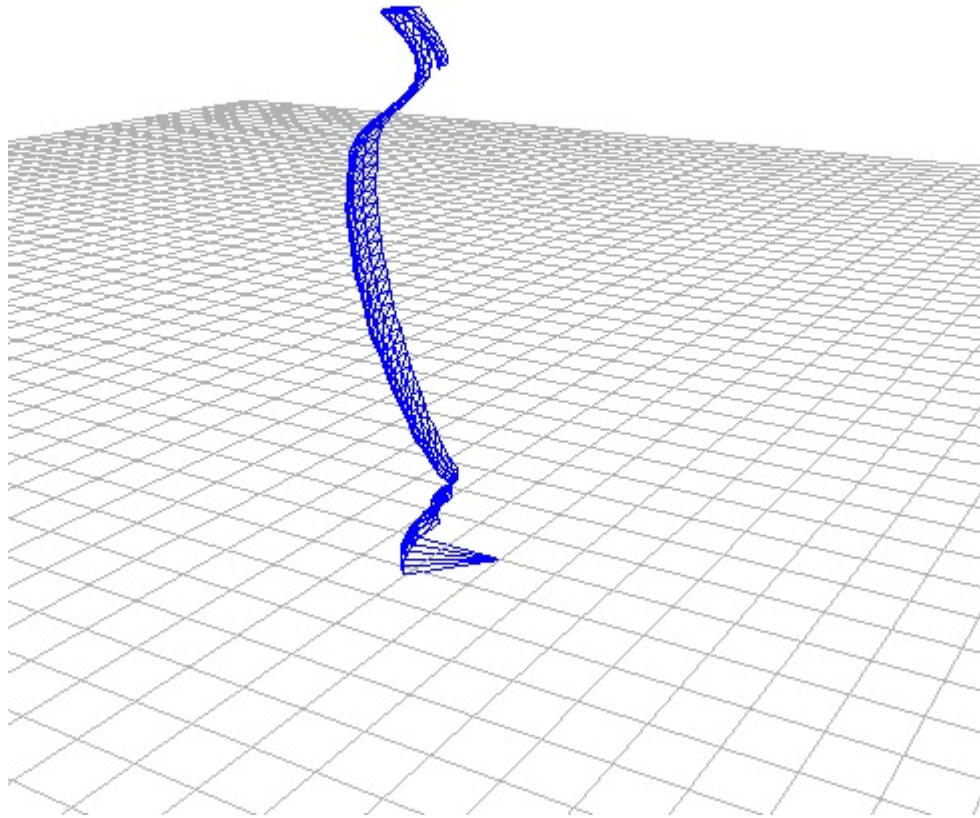
# A Simple Model Using Only GLUT Objects



# Modelling and Rendering: Example

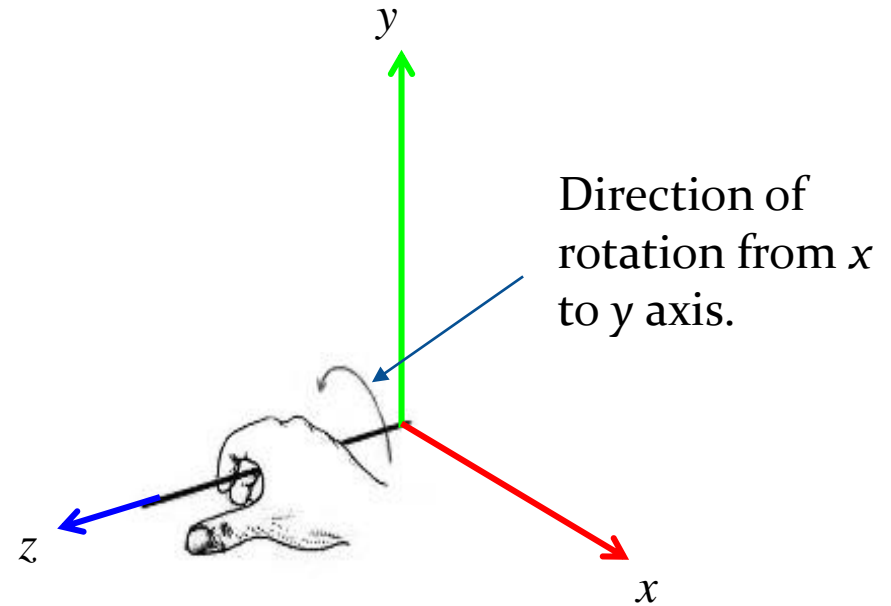
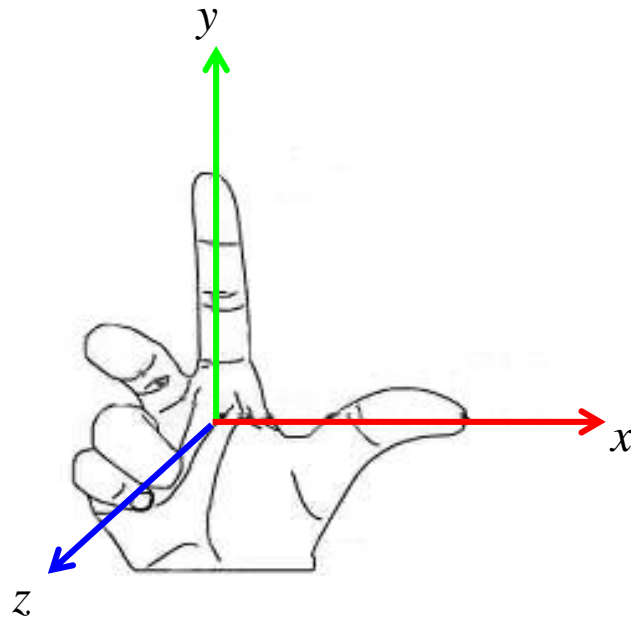


# Modelling and Rendering: Example



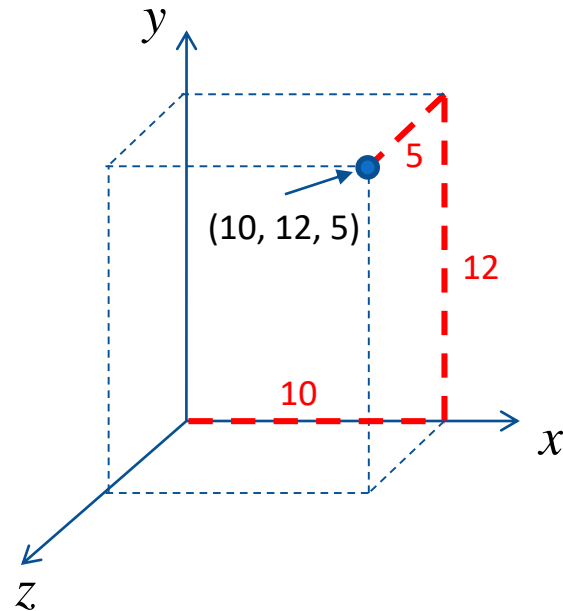
# Coordinate Frame

Three-dimensional **points**  $(x, y, z)$  and **vectors** are always defined in a **right-handed** coordinate reference frame.

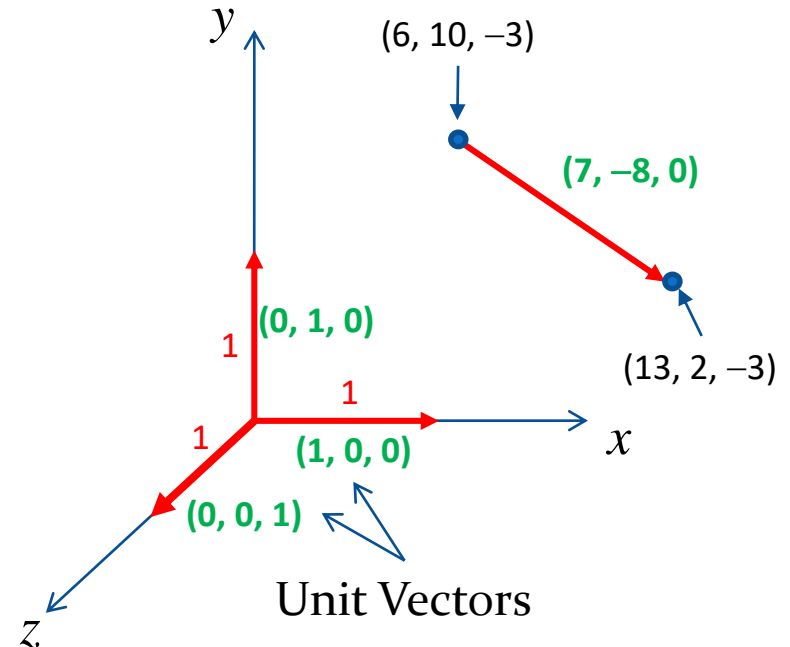




# Points and Vectors



3D coordinates of a point

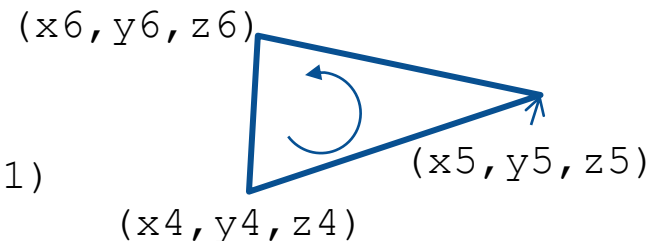
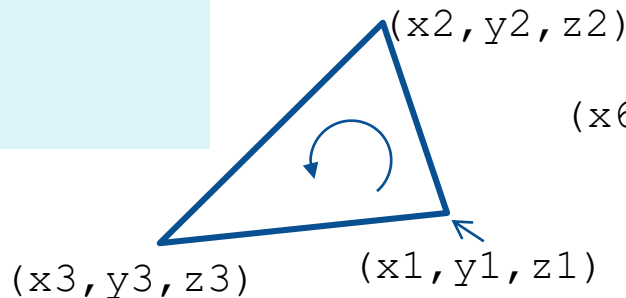
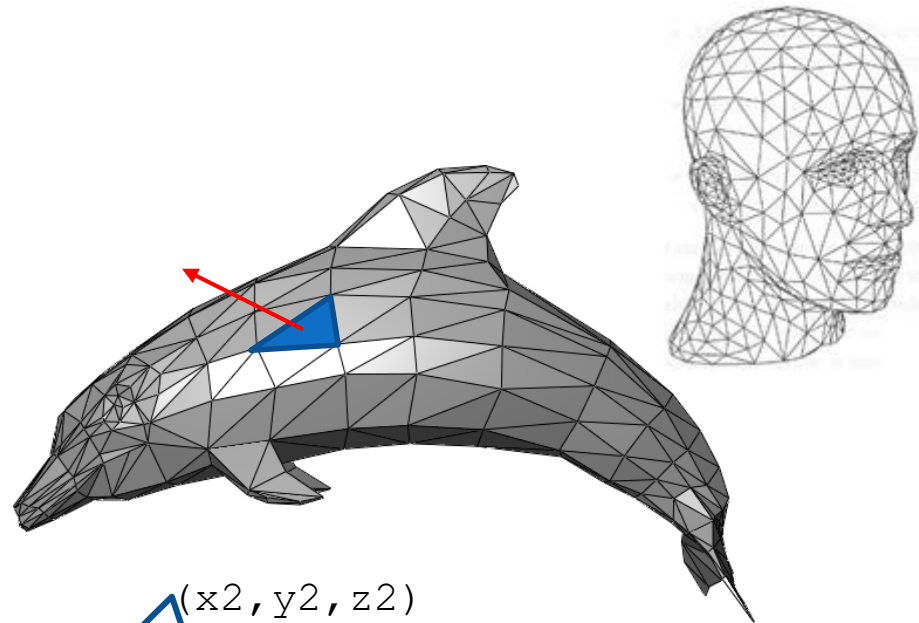


3D components of a vector

# Model Definition

3D models can be constructed using a collection of triangles.  
(Triangle meshes)

```
glBegin(GL_TRIANGLES);  
  glVertex3f(x1, y1, z1);  
  glVertex3f(x2, y2, z2);  
  glVertex3f(x3, y3, z3);  
  glVertex3f(x4, y4, z4);  
  glVertex3f(x5, y5, z5);  
  glVertex3f(x6, y6, z6);  
  ...  
  ...  
  ...  
glEnd();
```



**Important!**

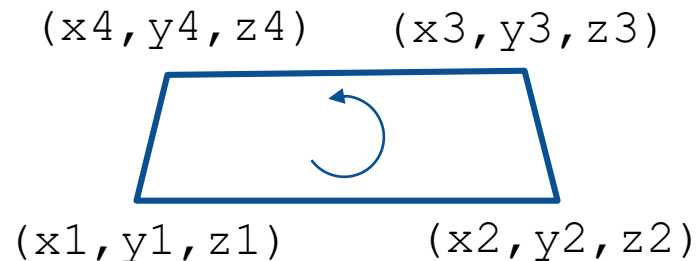
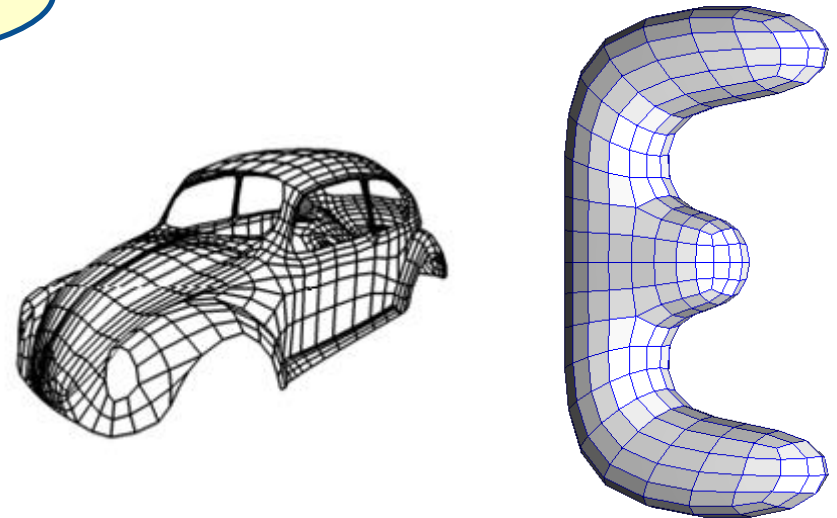
The vertices are always specified in **anti-clockwise** sense with respect to the outward normal.

# Model Definition

Some 3D models are constructed using only quadrilaterals.  
(Quad meshes)

Primitive  
Type

```
glBegin(GL_QUADS);  
  glVertex3f(x1, y1, z1);  
  glVertex3f(x2, y2, z2);  
  glVertex3f(x3, y3, z3);  
  glVertex3f(x4, y4, z4);  
  glVertex3f(x5, y5, z5);  
  glVertex3f(x6, y6, z6);  
  ...  
  ...  
  ...  
glEnd();
```



# Geometrical Objects

- Two other primitive types, **GL\_TRIANGLE\_STRIP** and **GL\_QUAD\_STRIP**, are also commonly used in object modelling. These will be introduced later..
- Several mesh file formats for storing object definitions exist: OBJ, PLY, **OFF**, DXF, 3DS, MAX, X3D ... etc. In addition to geometry data, a mesh file may also contain several other useful information such as normal components, texture coordinates, colour values, and material properties.
- Complex mesh geometries containing a large number of polygons are often stored in a compressed binary form.



19536 Triangles



69451 Triangles

# Object File Format (OFF)

The simplest ASCII mesh file format.

[Cube.off](#)

Always begins with keyword OFF

```
OFF
8 6 0
-0.5 -0.5 0.5
 0.5 -0.5 0.5
 0.5  0.5 0.5
-0.5  0.5 0.5
-0.5 -0.5 -0.5
 0.5 -0.5 -0.5
 0.5  0.5 -0.5
-0.5  0.5 -0.5
```

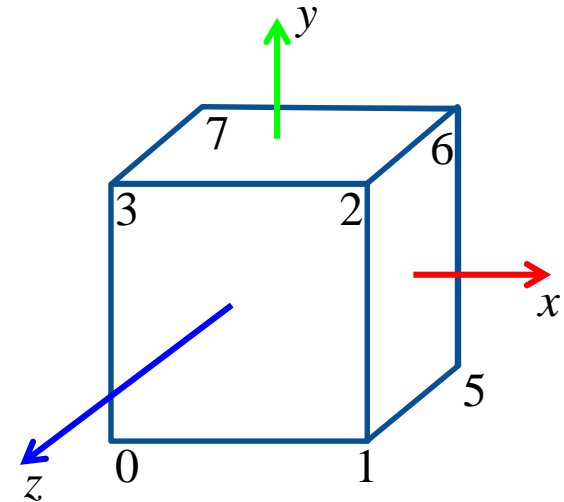
Number of vertices, polygons and edges

Vertex coords

```
4 0 1 2 3
4 1 5 6 2
4 3 2 6 7
4 3 7 4 0
4 0 4 5 1
4 7 6 5 4
```

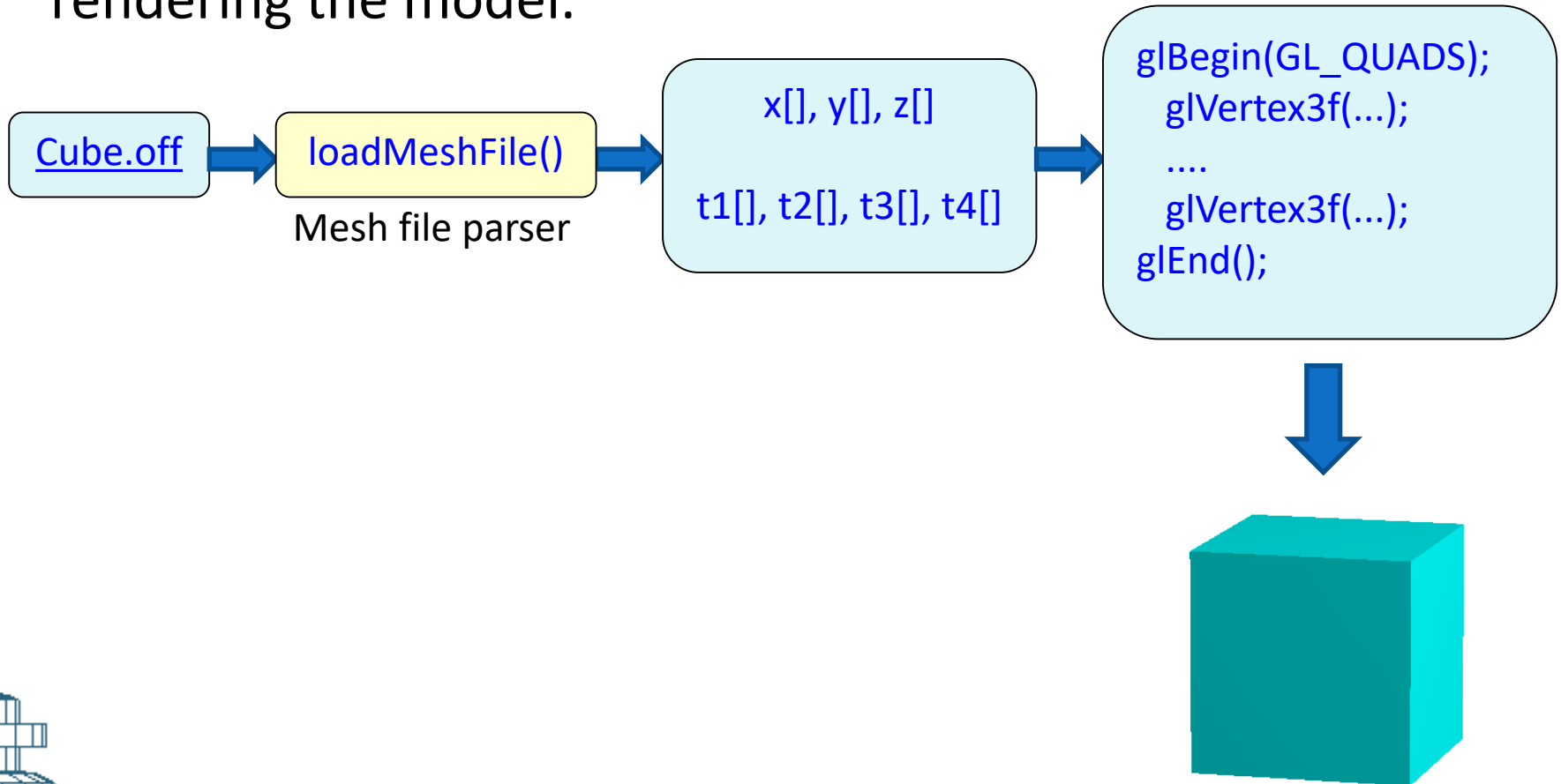
Face defn: No. of verts, face indices

Note anti-clockwise ordering of vertices



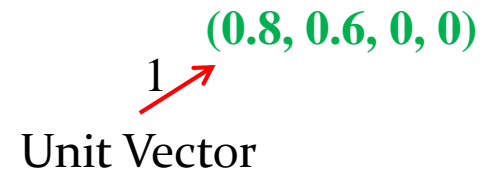
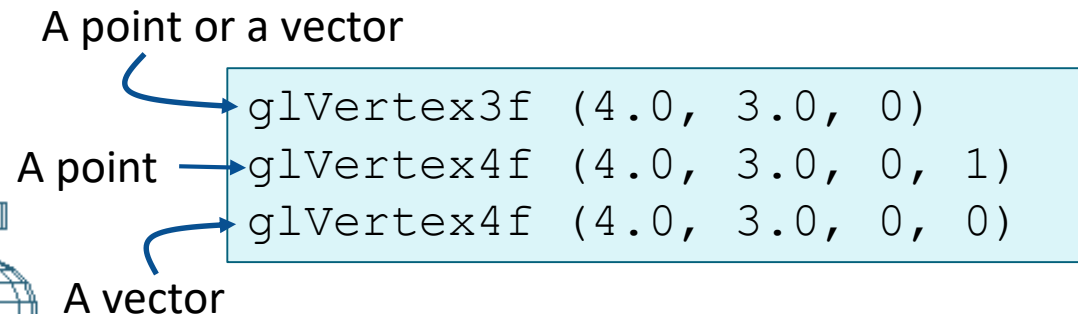
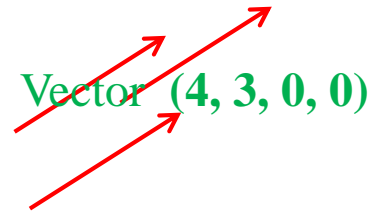
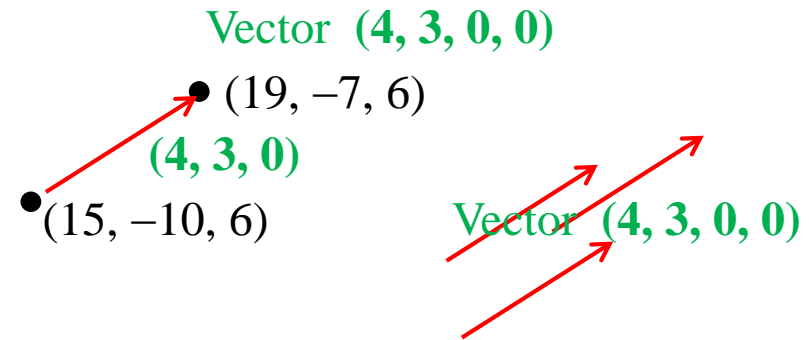
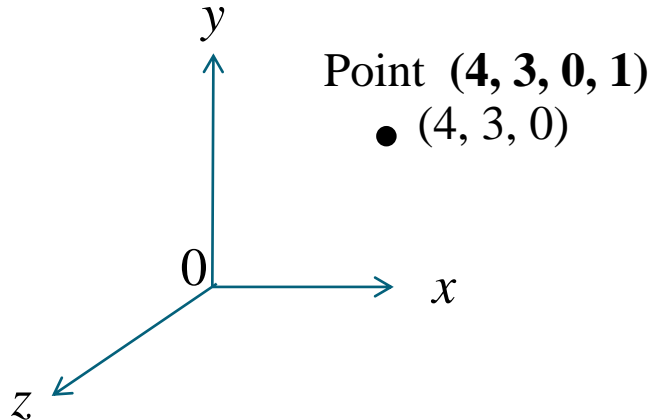
# Model Loaders

- A 3D model loader function parses a mesh file and extracts vertex, face and other related information required for rendering the model.



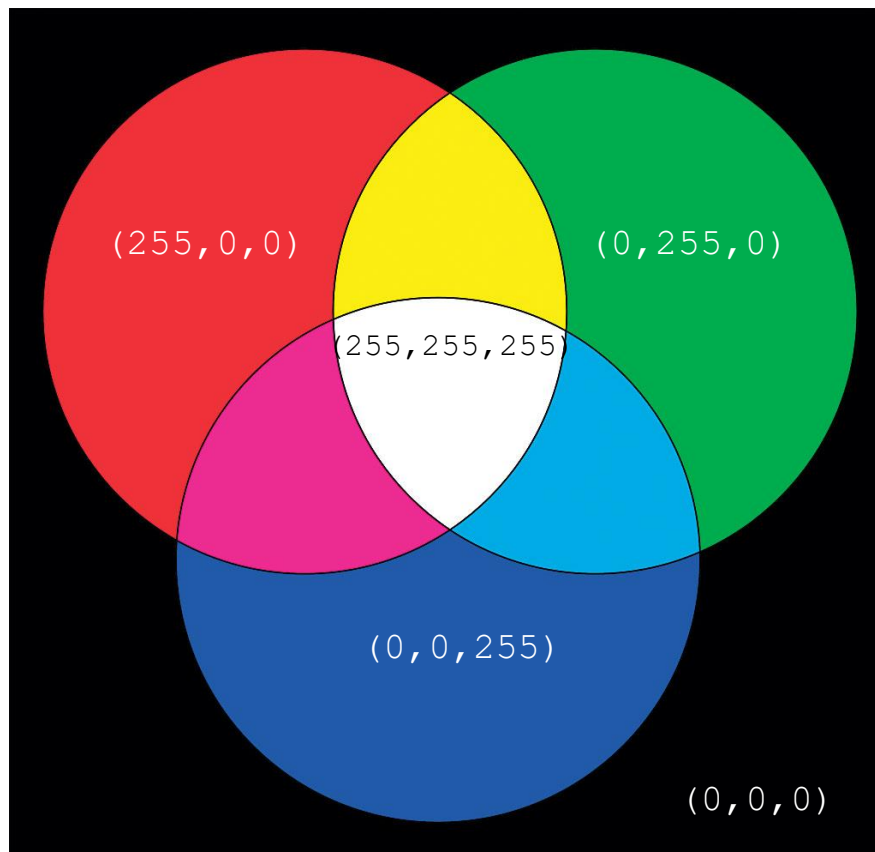
# Homogeneous Coordinates

We use **homogeneous coordinates** to distinguish between points and vectors.  $(x, y, z, 1)$  is a point while  $(x, y, z, 0)$  is a vector.



# Colour

Colors are defined using the additive RGB color space, with 3 components R, G, B, and an optional 4<sup>th</sup> alpha (transparency) component.



**Normalized Color Values**

Red	1	0	0
Green	0	1	0
Blue	0	0	1
Cyan	0	1	1
Magenta	1	0	1
Yellow	1	1	0
White	1	1	1
Black	0	0	0

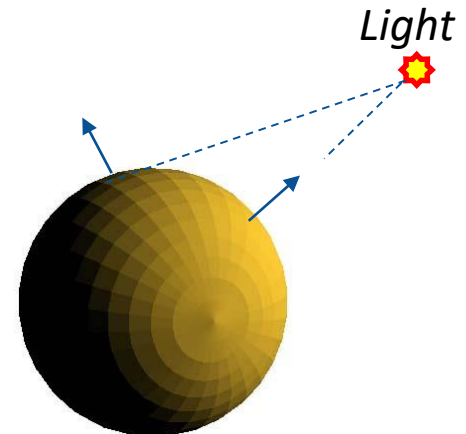
```
glColor3f(0, 1, 1);  
glColor4f(0, 1, 1, 1);  
glColor3ub(0, 255, 255);
```



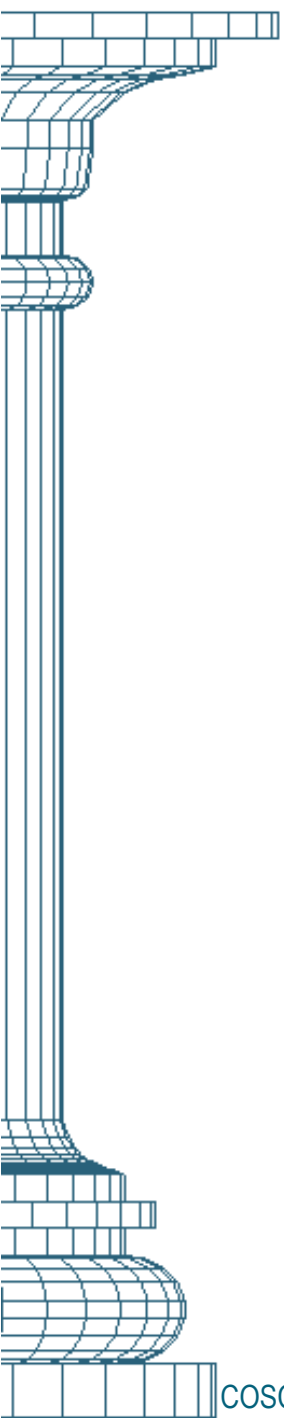
# Color

- Colors are used as **material properties** and **light properties**.
- A light's color is usually specified as white (1,1,1).
- The color on a diffusely reflecting surface varies from the specified material color to black (0,0,0), depending on the orientation of the surface with respect to the light source.

(1, 0, 0)                      (0.5, 0., 0.)      (0.2, 0., 0.)      (0., 0., 0.)

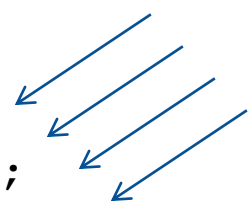


# Quiz!



# Lights

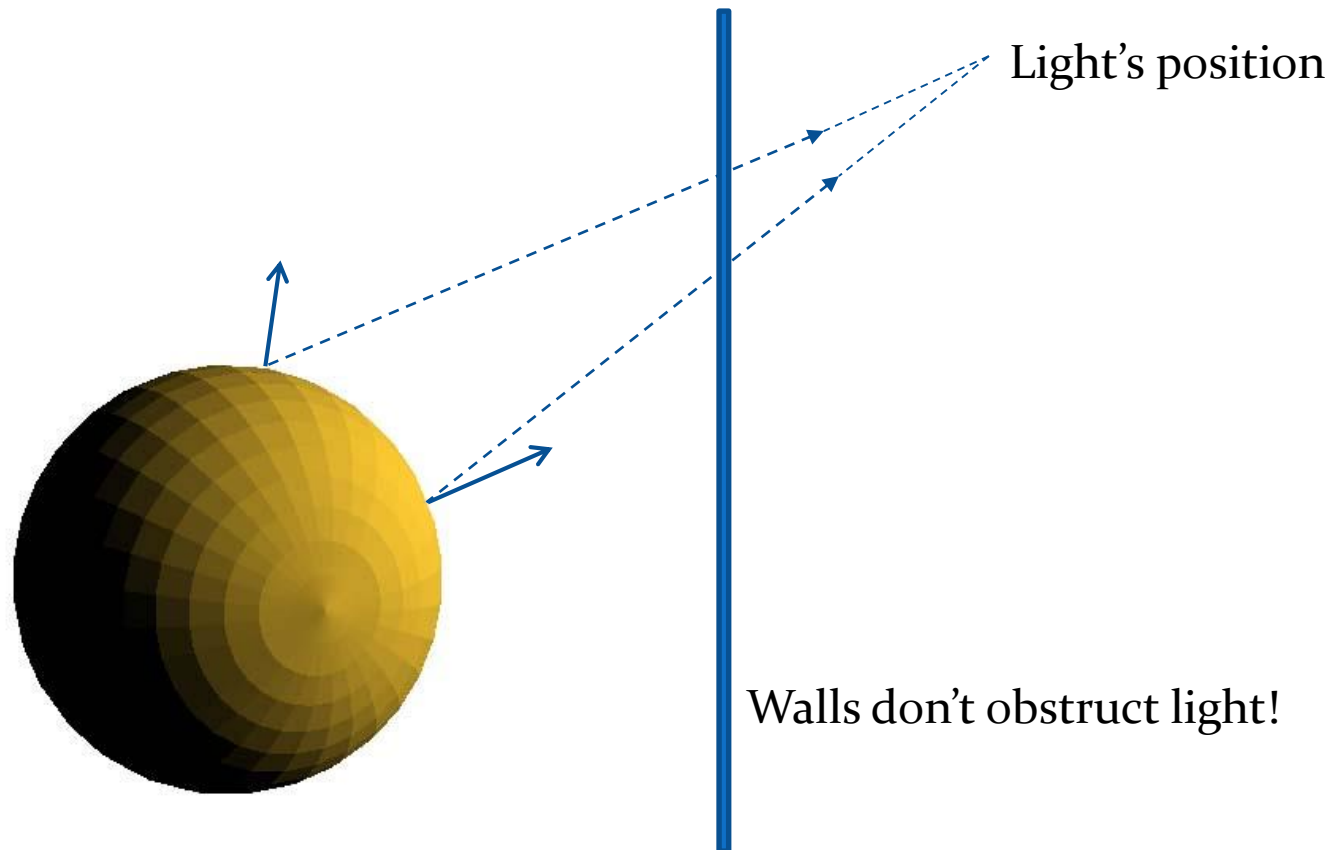
- In OpenGL, you can select up to 8 light sources:
  - `GL_LIGHT0, GL_LIGHT1, ..., GL_LIGHT7`
  - A light source is selected using `glEnable(GL_LIGHT#)` ; 0...7  
↓
- You should also enable lighting calculations using `glEnable(GL_LIGHTING)` ;
- Positioning light sources:
  - Omni-directional point sources: ☀ (x, y, z)  

```
float light_pos[] = {10, -3.5, 200, 1}  
glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
```
  - Directional light sources:   
`(x, y, z)`  

```
float light_pos[] = {10, -3.5, 200, 0}  
glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
```

# Lights

In OpenGL, a light source is just a virtual point. This position is used only to modulate the colour values at vertices based on the angle between the light vector and the surface normal.

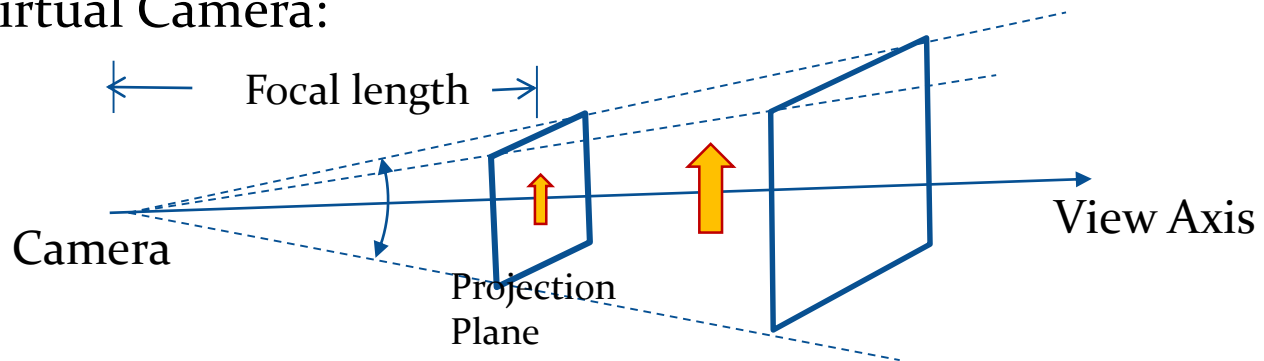


# Camera

Real Camera:

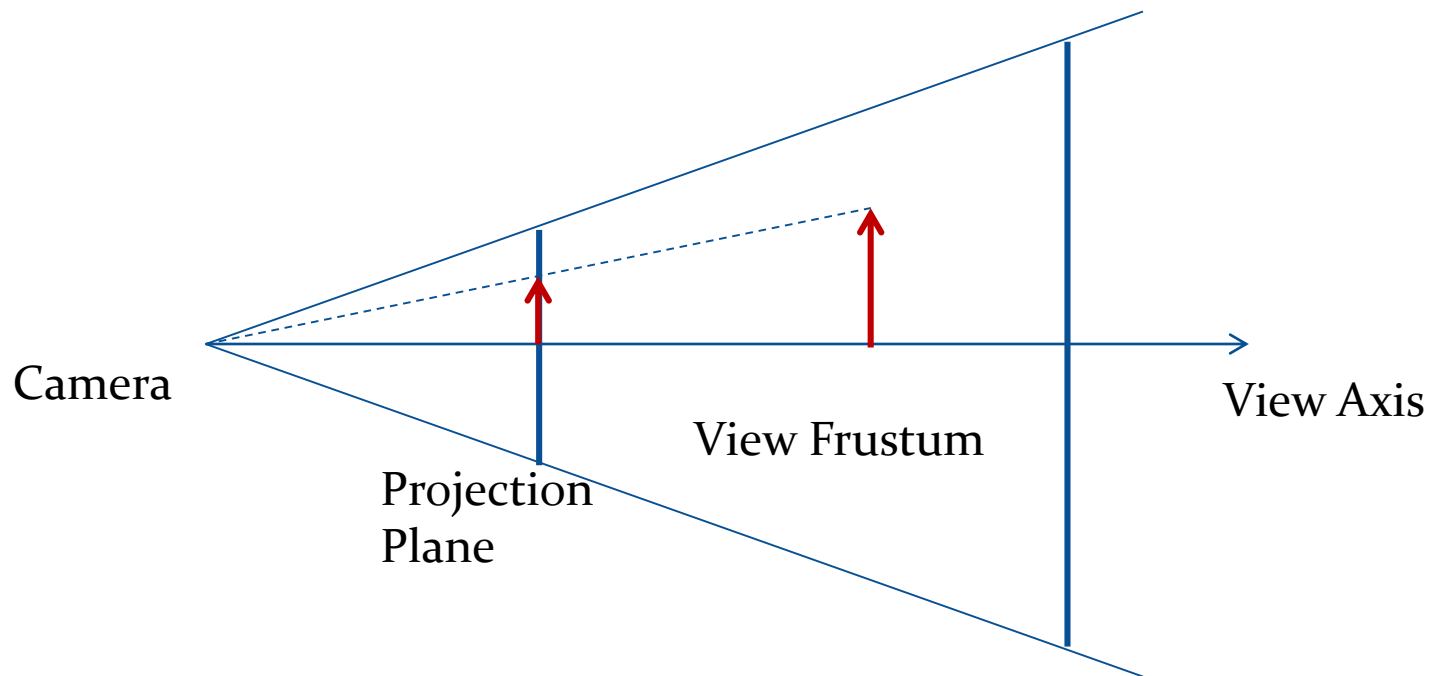


OpenGL Virtual Camera:

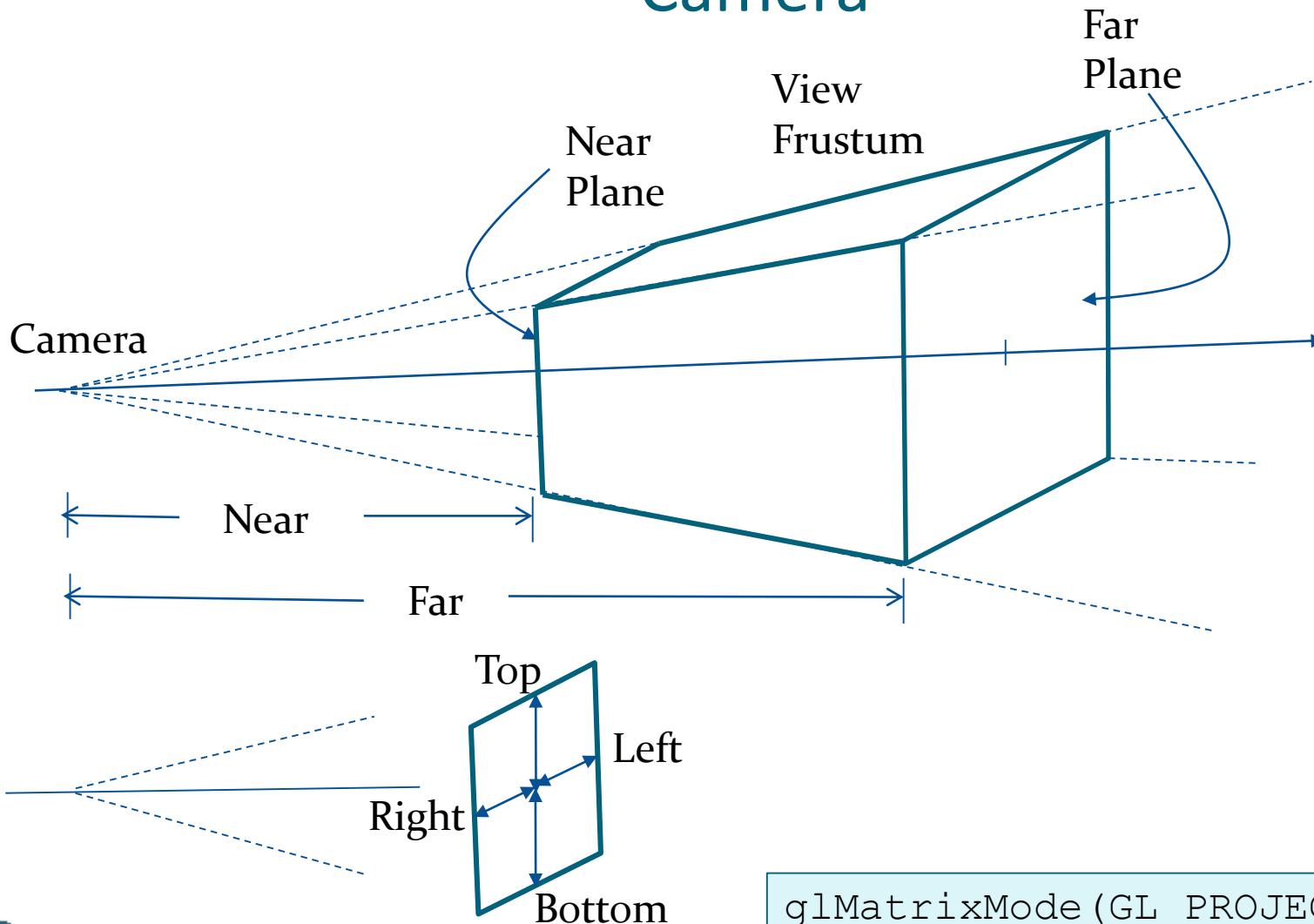


# Camera

OpenGL uses a simple perspective projection model based on a view frustum specified along the view axis of the camera. Only those vertices that fall within the view frustum are processed.



# Camera

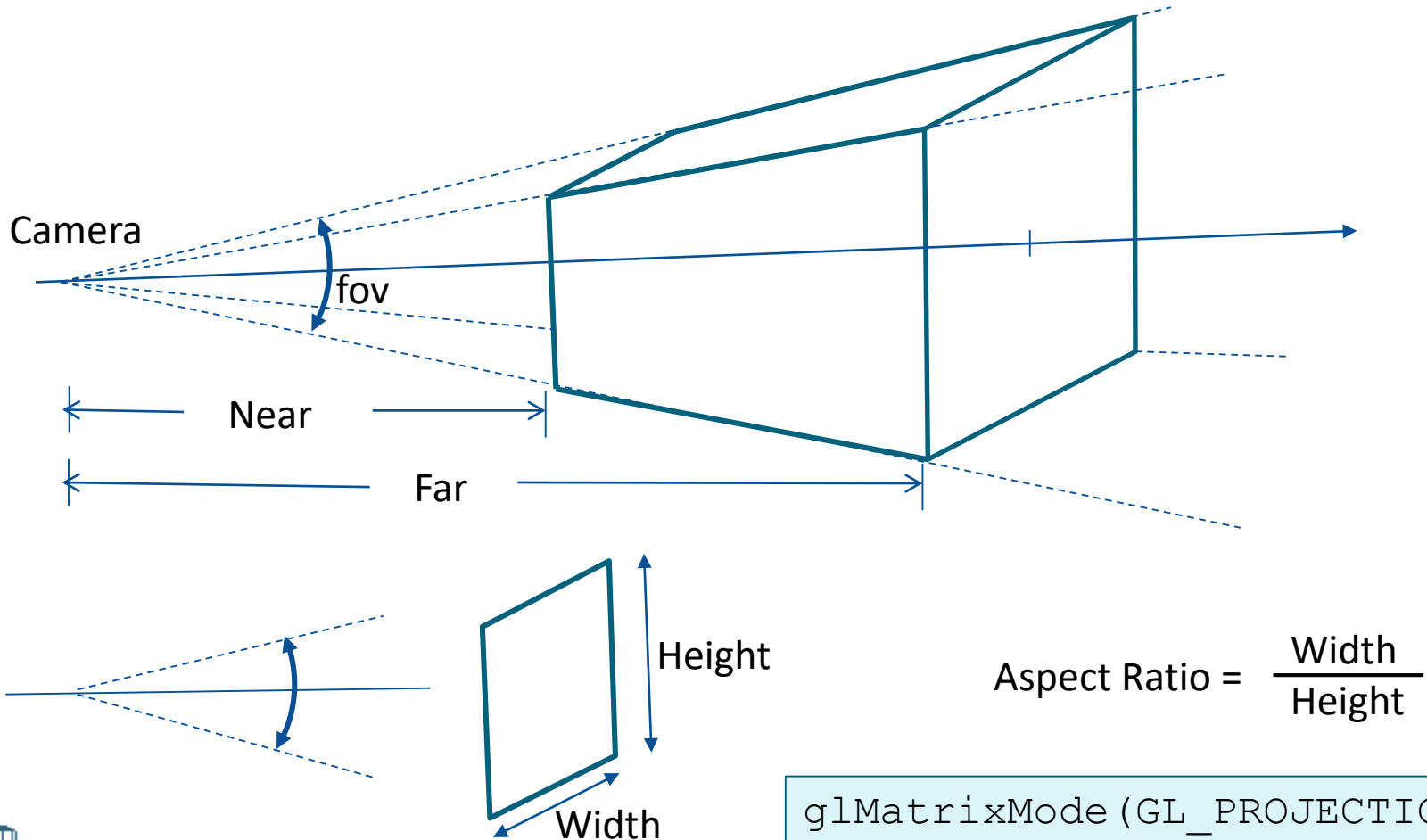


```
glMatrixMode(GL_PROJECTION);  
glLoadIdentity();
```

```
glFrustum(left, right, bottom, top, near, far);
```

# Camera

Alternate definition using field of view (fov) and aspect ratio:



```
glMatrixMode(GL_PROJECTION);  
glLoadIdentity();
```

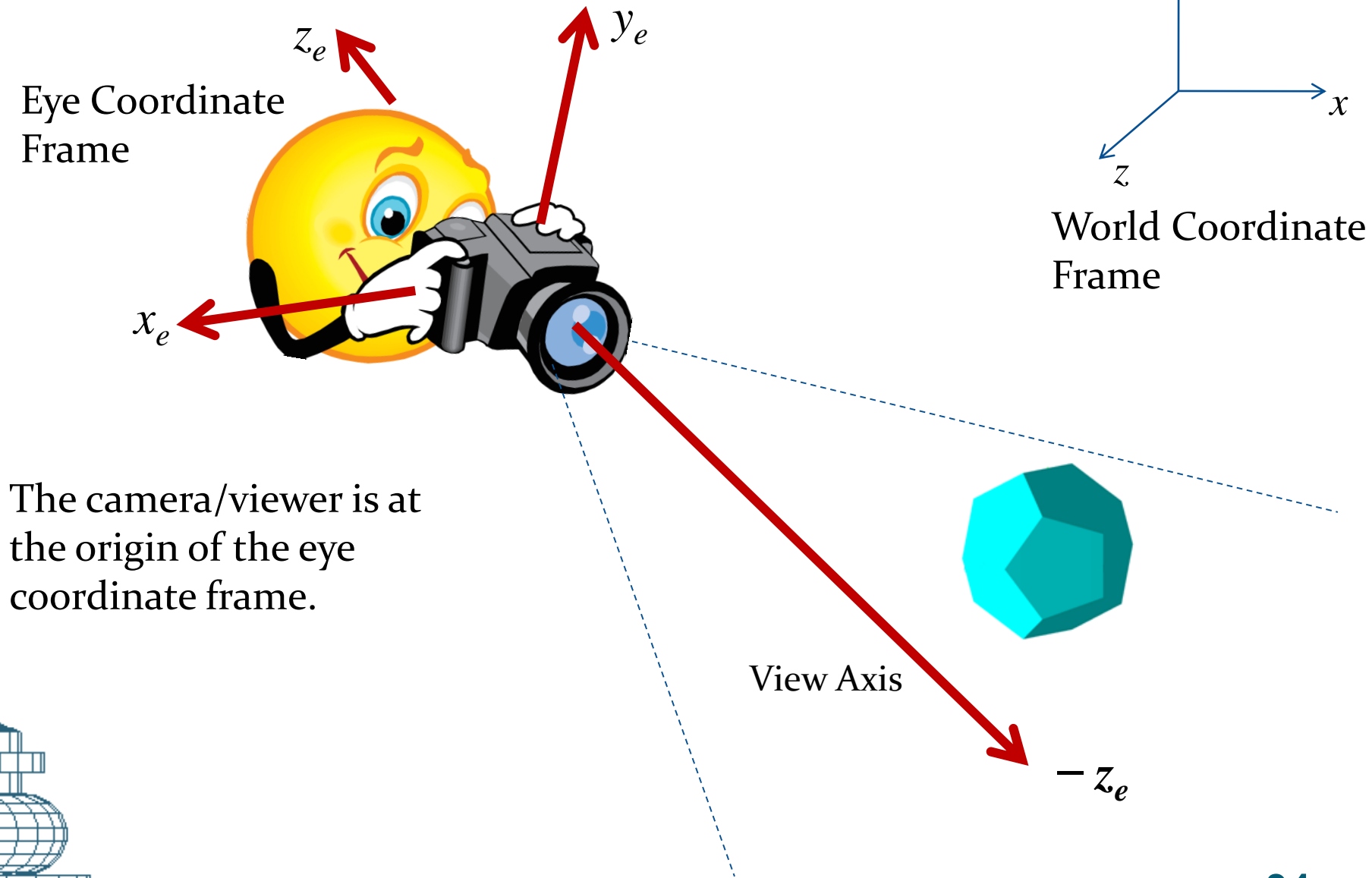
```
gluPerspective(fov, aspect, near, far);
```



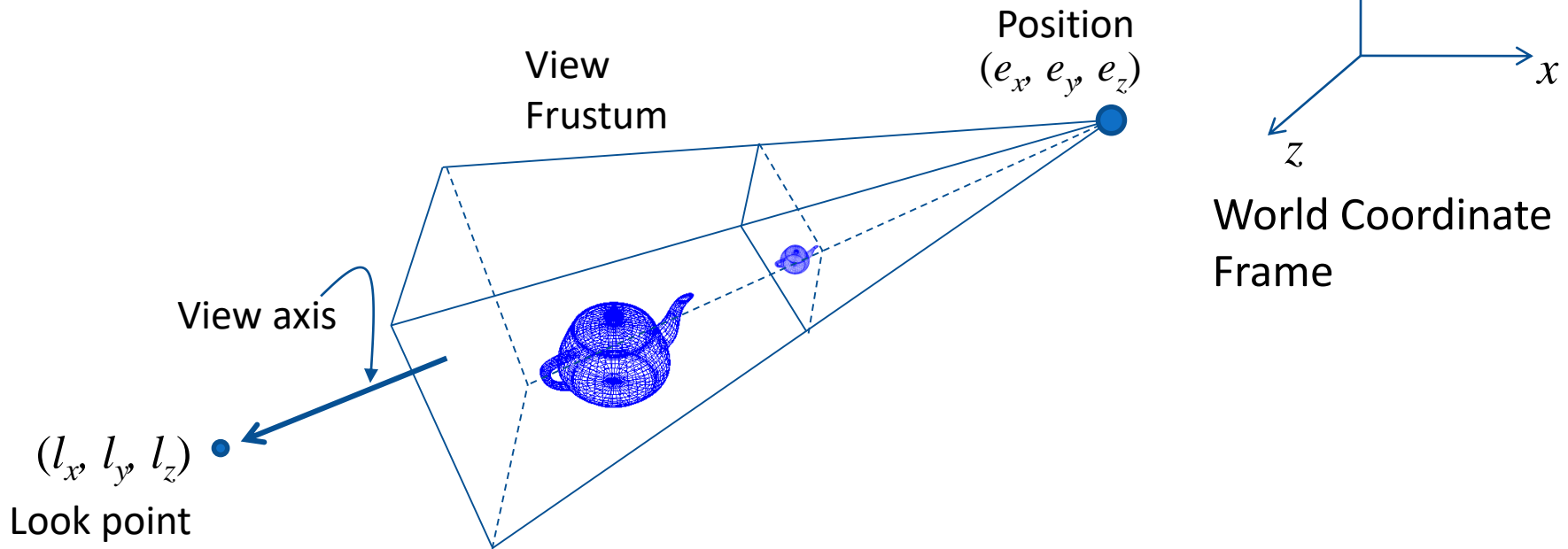
# Camera

- The OpenGL functions `glFrustum`, `gluPerspective` specify the projection mechanism. They only “select the lens of the camera”.
- We need to position the camera within the scene at the required location and orientation. This is done using the `gluLookAt ( . . )` function.
- By positioning and orienting the camera, you are actually defining a transformation from the world coordinate space (using which the scene is constructed) to the local coordinate space of the camera (eye coordinate frame).

# Camera View



# Camera Position

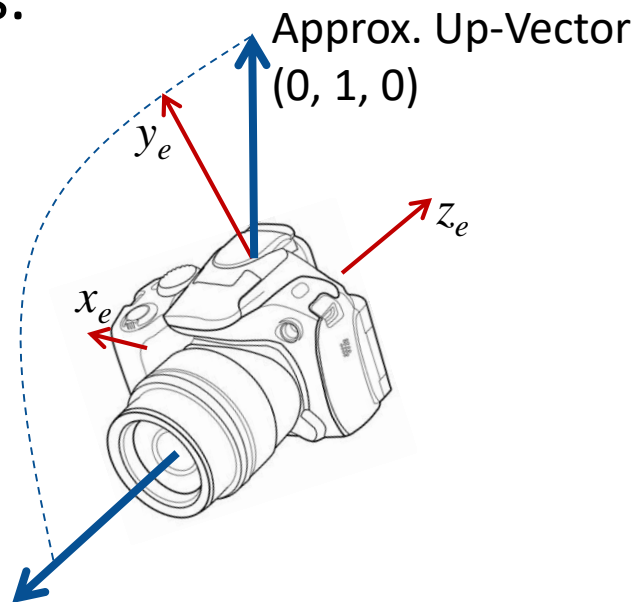


```
gluLookAt (ex, ey, ez, lx, ly, lz, 0, 1, 0) ;
```

Camera position      Look point      Approximate Up-vector

# Camera Orientation

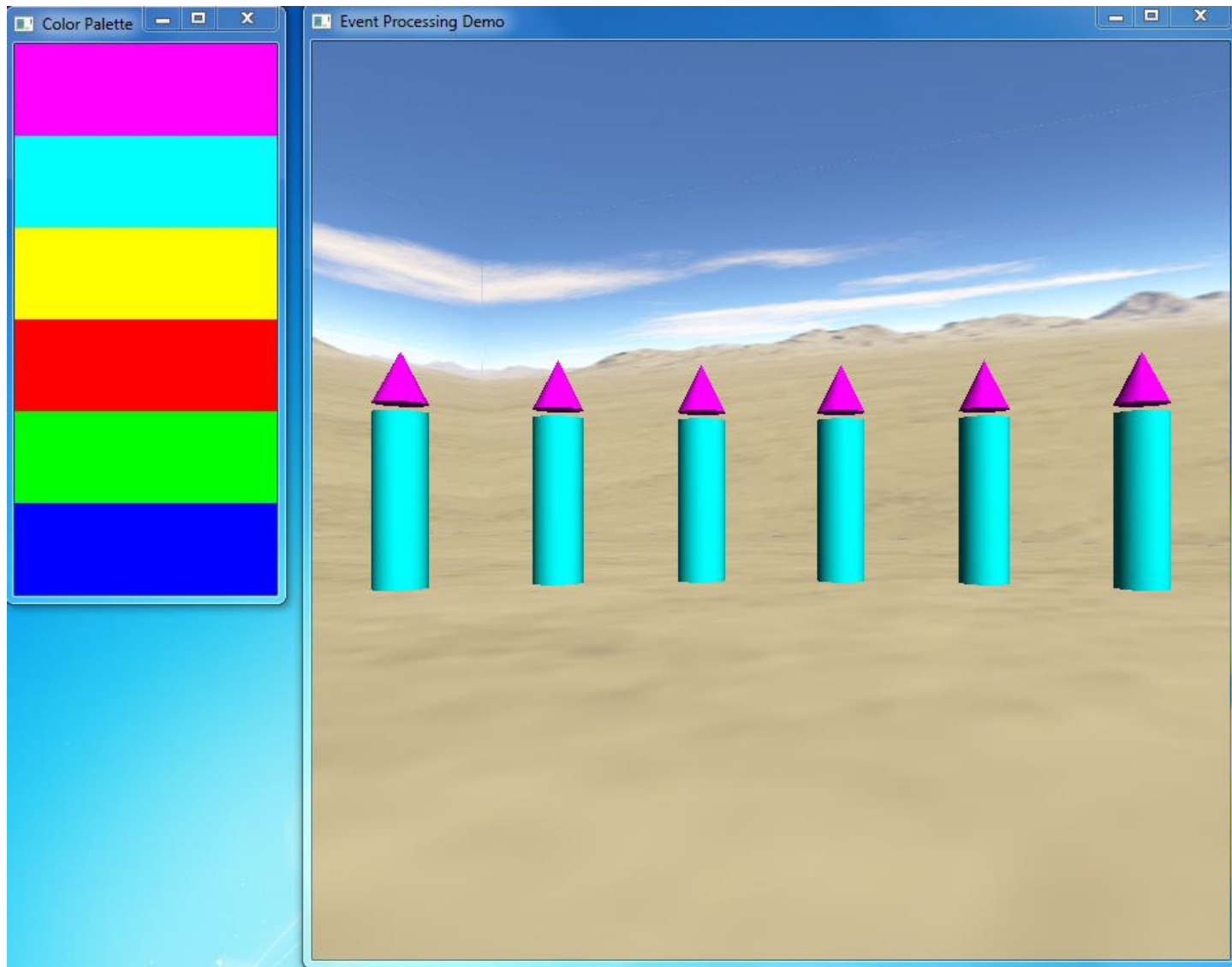
- The position and the look-point together define the camera's view axis (and therefore the  $z_e$  axis).
- We have to fix the rotation of the camera *about* the view axis. This is done by specifying the approximate up-vector – usually defined as  $(0, 1, 0)$ . The camera's true up-vector ( $y_e$ ) then gets fixed on the plane containing the approximate up-vector and the view axis.



# Event Processing (Interactive Graphics)

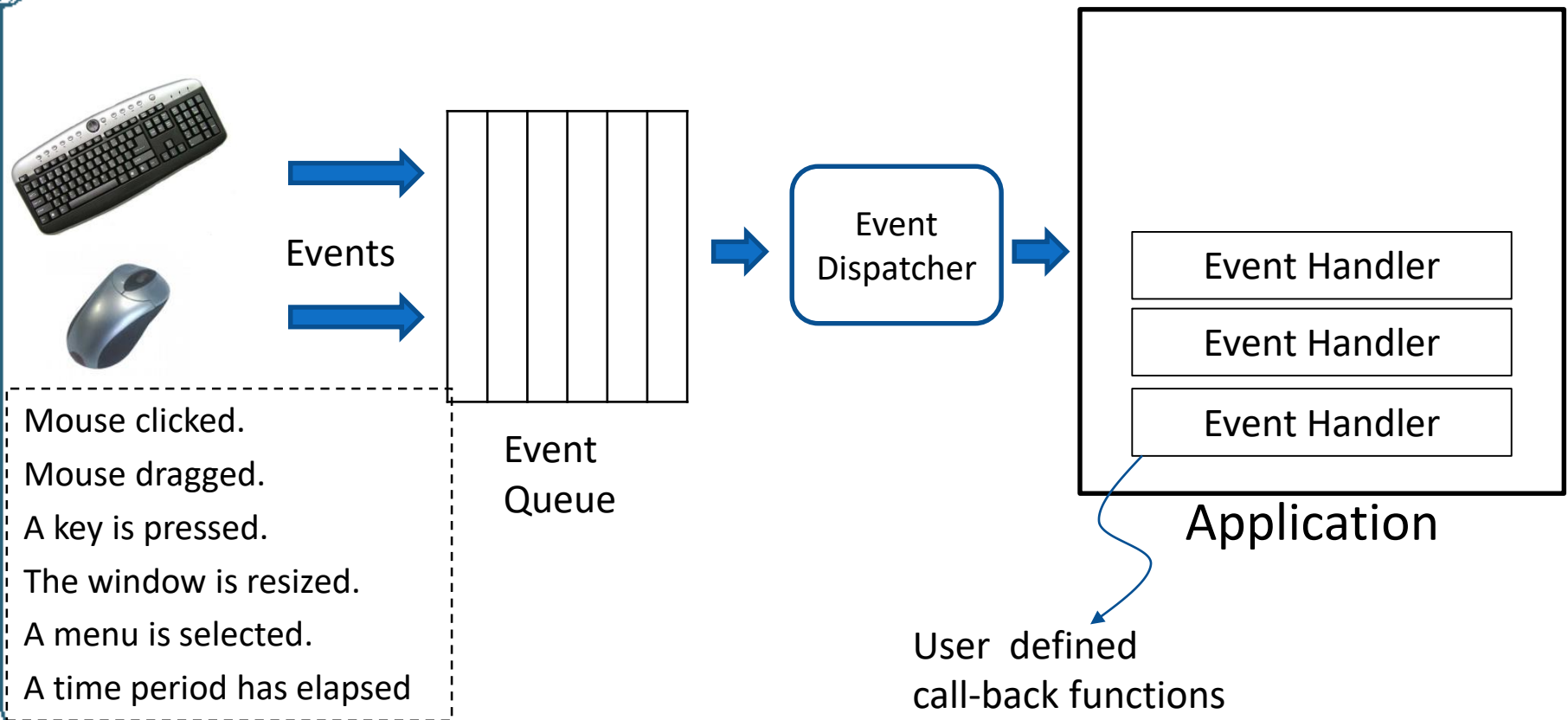
- A graphics application should allow the user to interact with it through keyboard and mouse inputs.
- The program should be able to receive **inputs** (egs. a key is pressed, mouse button clicked, mouse dragged) or **actions** (window resized, a time period has elapsed, a display refresh requested) while the application is running. These inputs and actions are called **events**.
- The program should be able to respond to events by executing certain functions. Such functions are called **event handlers** or **call-back functions**.
- GLUT provides an effective event-driven functionality common to all window systems.

# User Interaction (Demo)



# Event-Driven Programming

All call-back functions must be registered within the program.



# Keyboard Events

- A keyboard event is generated when the user presses a key that represents a printable (ASCII) character.

- The call-back function is registered using

```
glutKeyboardFunc (keyboard) ;
```

- The call-back function signature is

```
keyboard(unsigned char key, int x, int y)
```

`key` = key pressed

`x`, `y` = GLUT coordinates of the mouse position within the window, at time of key press (rarely used)



# Keyboard Event Example

```
void keyboard(unsigned char key, int x, int y){
    switch(key)
    {
        case 'a':    rotn += 5;  break;        //turn left
        case 'd':    rotn -= 5;  break;        //turn right
        case 'w':    step  = 2;  break;        //move forward
        case 'x':    step  = -2; break;        //move backward
    }
    glutPostRedisplay(); //update display
}
```

Generates  
display-refresh  
event

```
void main( int argc, char **argv)
{
    ...
    glutKeyboardFunc(keyboard) ;
    ...
}
```

# Special Keyboard Event

- Special keys represent non-printable characters. Egs: Arrow keys, Function keys F1..F12, PageUp, PageDown etc.
- Registered using **`glutSpecialFunc(special)`** ;
- Call-back signature:

```
special(int key, int x, int y);
```

where key is one of the following:

- **Function keys:** `GLUT_KEY_F1`, ... `GLUT_KEY_F12`
- **Arrow keys:** `GLUT_KEY_UP`, `GLUT_KEY_DOWN`,  
`GLUT_KEY_LEFT`, `GLUT_KEY_RIGHT`
- **Other keys:** `GLUT_KEY_HOME`, `GLUT_KEY_END`,  
`GLUT_KEY_PAGE_UP`, `GLUT_KEY_PAGE_DOWN`,  
`GLUT_KEY_INSERT`

**x, y :** GLUT coordinates of the current mouse position.

# Special Keyboard Callback Example

```
void special(int key, int x, int y)
{
    if(key == GLUT_KEY_F1)          glutFullScreen();
    else if(key == GLUT_KEY_F2)      animate();
    else if(key == GLUT_KEY_LEFT)    angle -= 5;
    else if(key == GLUT_KEY_RIGHT)   angle += 5;

    glutPostRedisplay();
}
```

```
void main( int argc, char **argv)
{
    ...
    glutSpecialFunc(special);
    ...
}
```

# Mouse Event

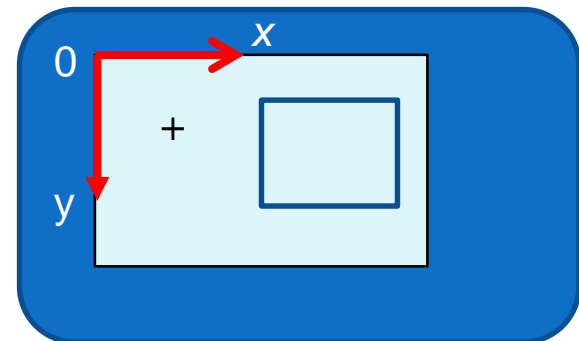
- A mouse event occurs when a user presses or releases a mouse button inside a window. The call-back function is registered using

**`glutMouseFunc (mouse) ;`**

- The call-back function signature is:

```
mouse(int button, int state, int x, int y);
```

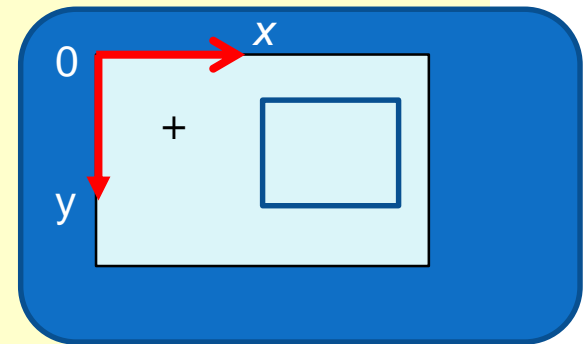
- *Button*: The mouse button that is pressed.  
GLUT\_LEFT\_BUTTON, GLUT\_RIGHT\_BUTTON, GLUT\_MIDDLE\_BUTTON
- *State*: The state of the button. GLUT\_UP, GLUT\_DOWN
- *x, y*: GLUT coordinates of the mouse position within the window.



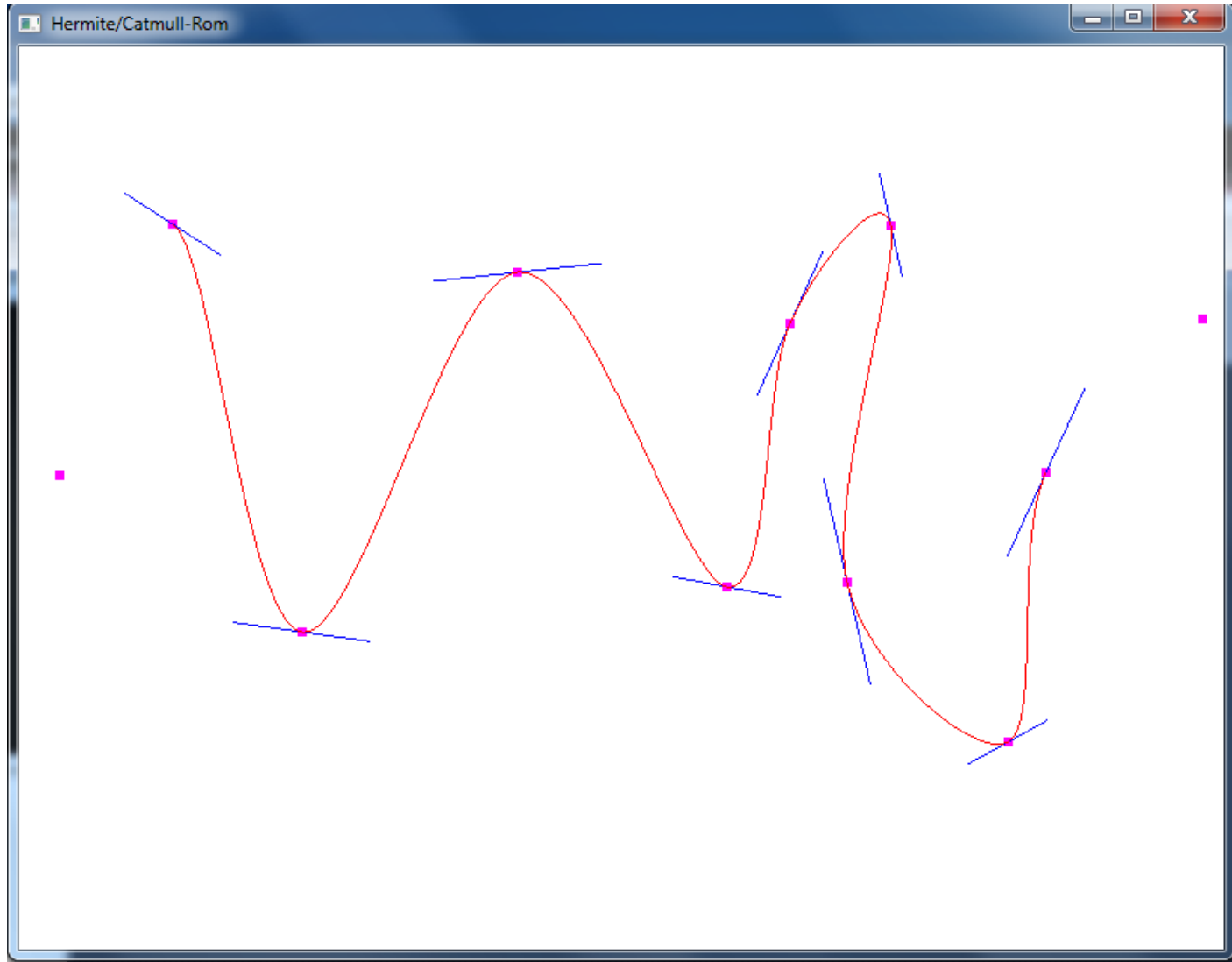
# Mouse Event Example

```
void mouse(int b, int s, int x, int y)
{
    int xpoint, ypoint;
    if(b == GLUT_LEFT_BUTTON && s == GLUT_DOWN)
    {
        xpoint = x;
        ypoint = winHeight - y;
        drawDot(xpoint, ypoint);
    }
    glutPostRedisplay();           //update display
}
```

```
void main( int argc, char **argv)
{
    ...
    glutDisplayFunc(display);
    glutMouseFunc(mouse);
    ...
    glutMainLoop();
}
```



# Input Using Mouse Events (Demo)



# Mouse Motion Events

- Mouse Motion Event

- Mouse is moved while one or more buttons are pressed. The call-back function is registered using

```
glutMotionFunc (motion) ;
```

- Mouse Passive Motion Event

- Mouse is moved while no buttons are pressed. The call-back function is registered using

```
glutPassiveMotionFunc (passiveMotion) ;
```

- Call-back signatures:

```
motion (int x, int y) ;
```

```
passiveMotion (int x, int y) ;
```

# Display Call-back

- The display call-back function is executed whenever the native windows system decides that the window must be refreshed.
  - The window is opened
  - The window is re-sized
  - An overlapping window is moved etc.
- The call-back function is registered using  
`glutDisplayFunc (display) ;`
- A window redraw event can also be generated by the user at any time by calling the function  
**`glutPostRedisplay ()`**



# Window Reshape Event

- The reshape event is triggered when the graphics window is resized. It is also triggered immediately before a window's first display call-back. The call-back is registered using

```
glutReshapeFunc(reshape);
```

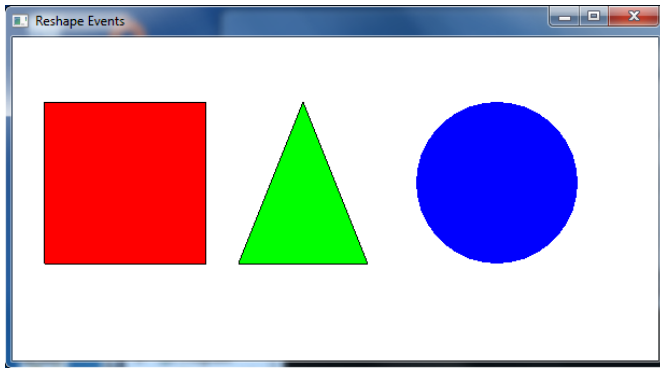
- The call-back signature is

```
reshape(int width, int height);
```

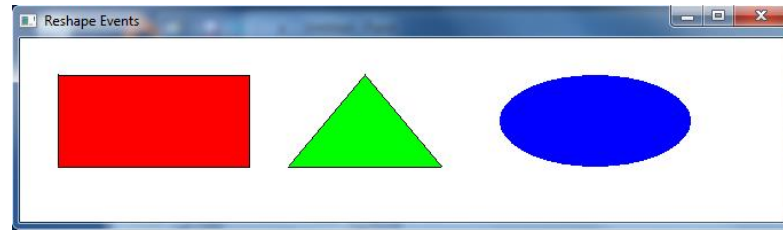
- Example:

```
void reshape(int width, int height)
{
    float aspectCurr = (float)width/(float)height;
    if(aspectCurr > aspectReqd)
        glViewport(0, 0, height*aspectReqd, height);
    else
        glViewport(0, 0, width, width/aspectReqd);
}
```

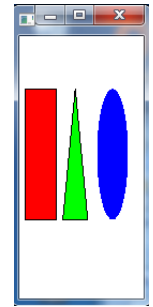
# Reshaping Windows (Demo)



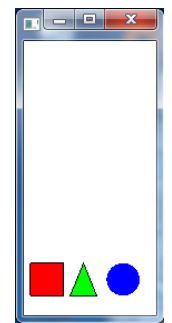
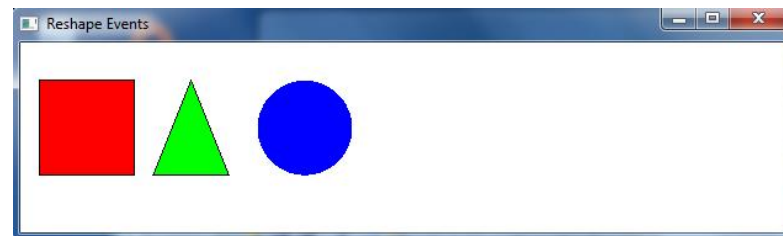
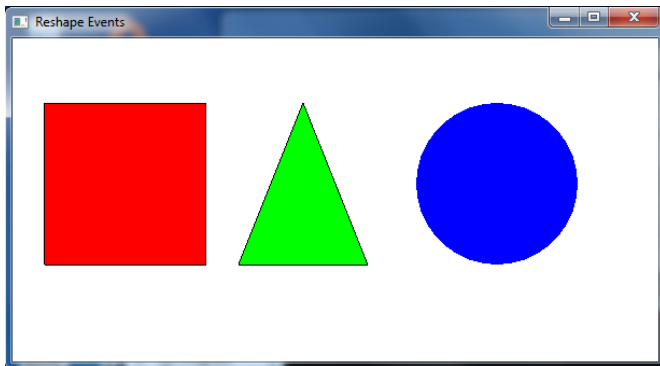
Aspect ratio = 2



Aspect ratio > 2



Aspect ratio < 2



# Timer Function

- The time function starts a timer in the event loop. It generates a timer event after a specified number of milliseconds. The call-back function is registered using `glutTimerFunc(delay, timer, value);`
  - The first parameter is the delay in milliseconds.
  - The second parameter is pointer to the callback function..
  - The third parameter is an integer value that is passed to the callback function.
- The signature of the call-back is `timer(value);`
- GLUT allows only a single timer. For a continuous animation sequence with the timer in the loop, it is necessary to start a new timer inside the call-back function.

# Timer Call-back Example

```
void myTimer (int value)
{
    if(value < 600)    //run animation for 1 minute.
    {
        ...           //update animation parameters
        glutPostRedisplay();
        value ++;
        glutTimerFunc(100, myTimer, value);
    }
}
```

```
//somewhere in the program..
```

```
glutTimerFunc(100, myTimer, 0);    //100m.Sec delay
```

# Multiple Windows

- GLUT can create and manage multiple windows, each with its own OpenGL context and call-back functions.
- The function `glutCreateWindow()` returns an integer that can be used as the index of the created window.
- The `glutSetWindow(index)` sets the window with the specified index as the current window.
- The events generated are associated with the current window.

# Multiple Windows

```
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize (700, 700);
    glutInitWindowPosition (300, 50);
    first = glutCreateWindow ("First Window");
    glutDisplayFunc (display1);
    glutMouseFunc (mouse1);
    glutKeyboardFunc (keyboard1);

    glutInitDisplayMode (GLUT_SINGLE);
    glutInitWindowSize(200, 420);
    glutInitWindowPosition(50, 50);
    second = glutCreateWindow ("Second Window");
    glutDisplayFunc (display2);
    glutMouseFunc (mouse2);

    glutSetWindow(first);
    glutMainLoop();
    return 0;
}
```