

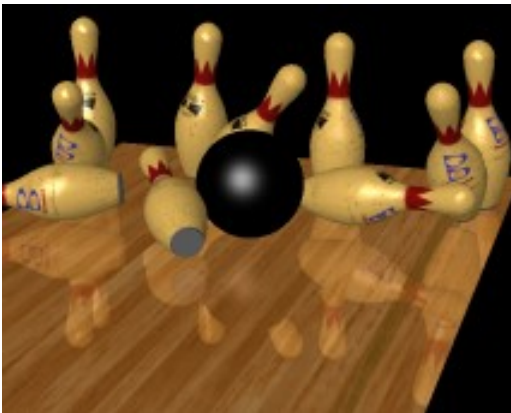
COMP 371 -- Winter 2012

Computer Graphics

- OpenGL Architecture
- Overview of OpenGL Programming
- First Program

What is Computer Graphics?

- Computer Graphics is concerned with producing images using a computer
 - **Modeling**: Create and represent the geometry of objects in the 3D world
 - **Rendering**: Generate 2D images of the objects
 - **Animation**: Describe how objects move
- Models come from a diverse and expanding sets of fields, and include physical, mathematical, engineering, architectural, and even conceptual structures, natural phenomena, and so on.

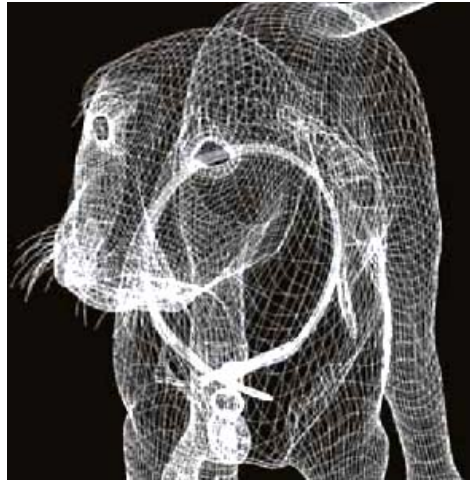


Modeling

- Take the real and turn it into a virtual
- Explain the real world or fantastic objects using mathematics
- If the image does not exist in real life, a blueprint is drawn by an artist

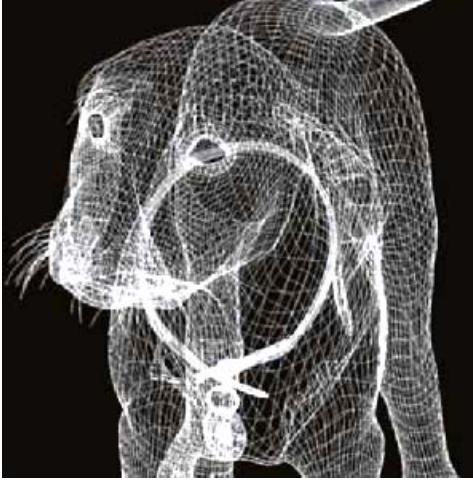


A wire frame is the simplest form of model



Wireframe Model

Rendering



Wireframe Model



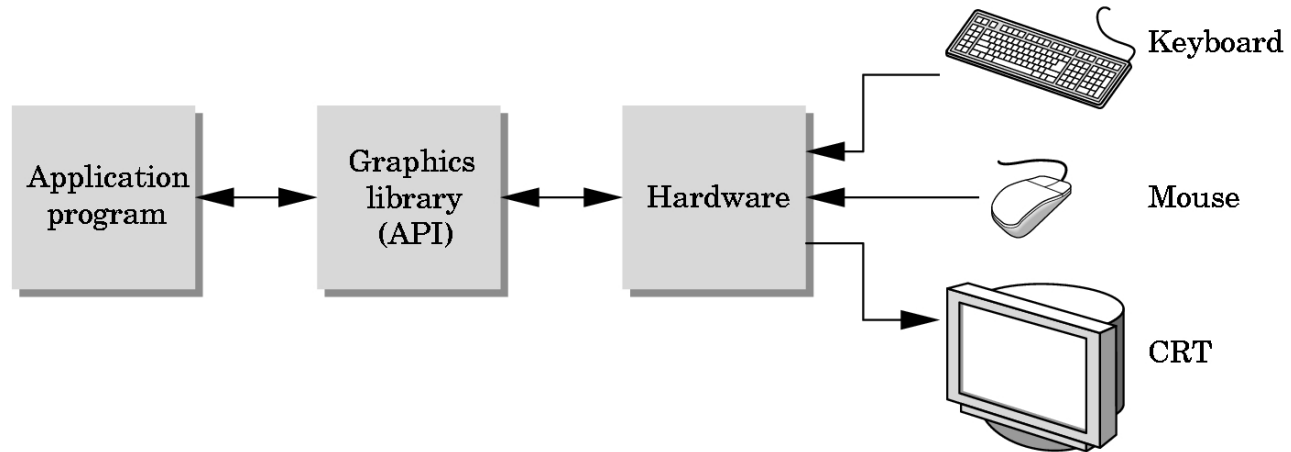
Final Render

- Draw the image on 2D screen
- Color
- Lighting
- Shading
- Surface texture
- Shadows
- Reflection and transparency



The Programmer's Interface

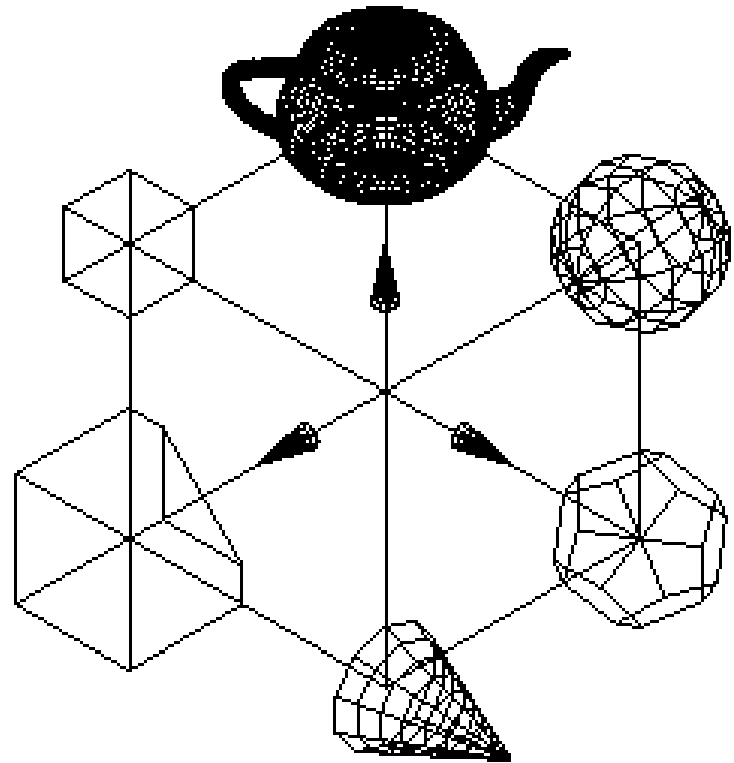
- Programmer sees the graphics system through the Application Programmer Interface (API)



- API contents: Functions that specify what we need to form an image
 - Objects
 - Viewer
 - Light Source(s)
 - Materials
- Other information
 - Input from devices such as mouse and keyboard

Computer Graphics: art of changing ideas into pixels

- Primitives Are Made of Pixels:
 1. Primitives are 2-D shapes (lines, triangles, circles, etc.)
 2. Even 3-D shapes are drawn with 2-D primitives
- Objects Are Made of Primitives:
 1. More primitives mean a more realistic object
 2. Thousands of primitives can make up each object:
- Scenes Are Made of Objects:
 1. Thousands of objects per scene
 2. Millions of primitives make up a scene
 3. Speed is key

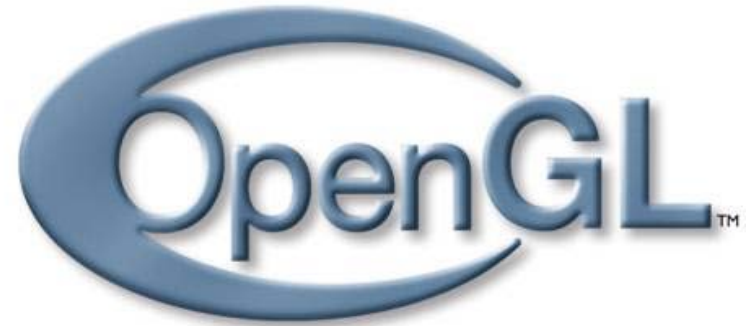


What is OpenGL ?

OpenGL is a programming interface mainly for 3D applications invented by *Silicon Graphics*. It renders 3D objects to the screen, providing the same set of instructions on different computers and graphics adapters. The OpenGL API was designed for use with the C and C++ programming languages, and allows developers in diverse markets such as broadcasting, CAD/CAM/CAE, entertainment, medical imaging, and virtual reality to produce and display incredibly compelling 2D and 3D graphics.

Why using OpenGL:

- Industry standard
- Stable
- Reliable and portable
- Easy to use
- Well-documented

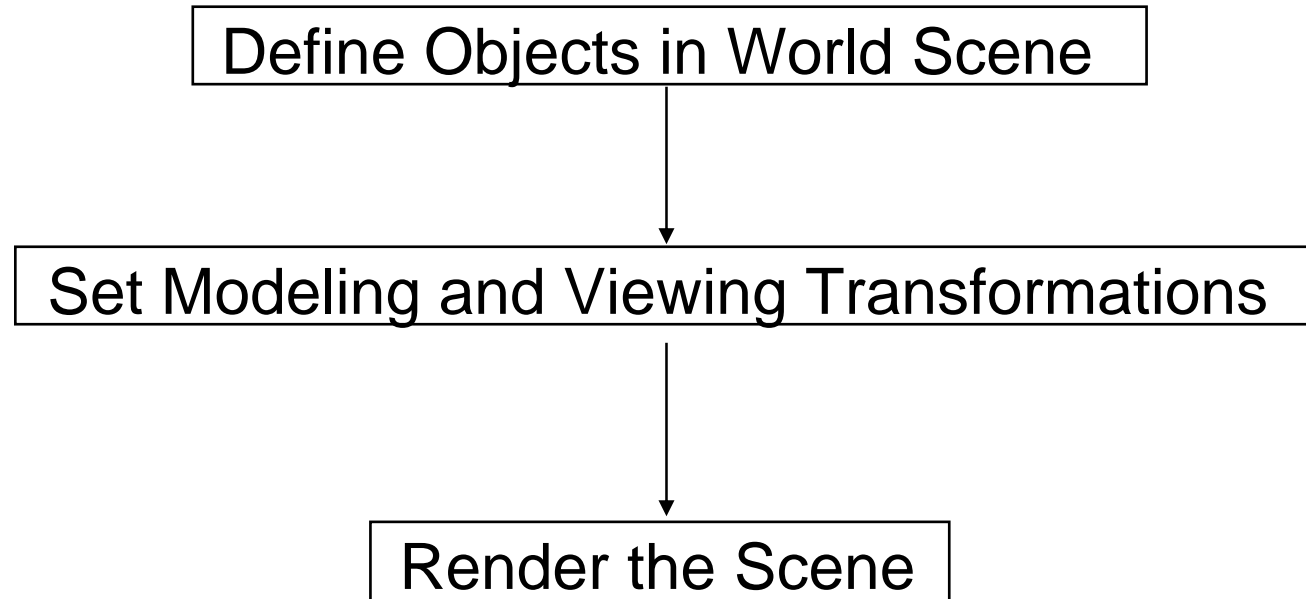


What OpenGL Does

- Allows definition of object shapes, material properties and lighting
- Arranges objects and interprets synthetic camera in 3D space
- Converts mathematical representations of objects into pixels (rasterisation)
- Calculates the colour of every object

- Provides no high-level rendering functions for complex objects; you must build your shapes from primitives (points, lines, polygons, etc.). The utility library GLU provides some support.

3 Stages in OpenGL



How OpenGL Works

- OpenGL is a state machine
 - You give it orders to set the current state of any one of its internal variables, or to query for its current status
 - The current state won't change until you specify otherwise
 - Ex.: if you set the current color to Red, everything you draw will be painted Red until you change the color explicitly
 - Each of the system's state variables has a default value

Structure of GLUT-Assisted Programs

- GLUT relies on user-defined callback functions, which it calls whenever some event occurs
 - Function to display the screen
 - Function to resize the viewport
 - Functions to handle keyboard and mouse events

Defining OpenGL Objects

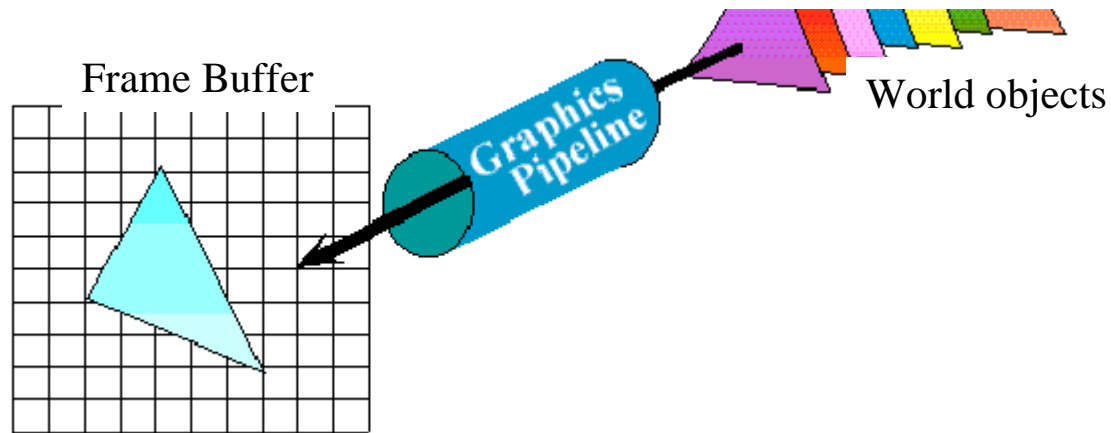
- OpenGL objects are represented as combinations of shapes, such as:
 - Points and lines
 - Triangles and Quadrilaterals
 - Polygons
 - Quadrics (spheres, cylinders, cones, etc.)
 - Curves and surfaces

Drawing Other Objects

- GLU contains calls to draw cylinders, cones and more complex surfaces called NURBS
- GLUT contains calls to draw spheres and cubes

Projective Rendering (OpenGL)

- Start with objects defined using polygons / primitives.



- In 2-D, map objects onto the screen
- In 3-D, project them onto the screen.
- This is called forward mapping.

The 3D Graphics Pipeline



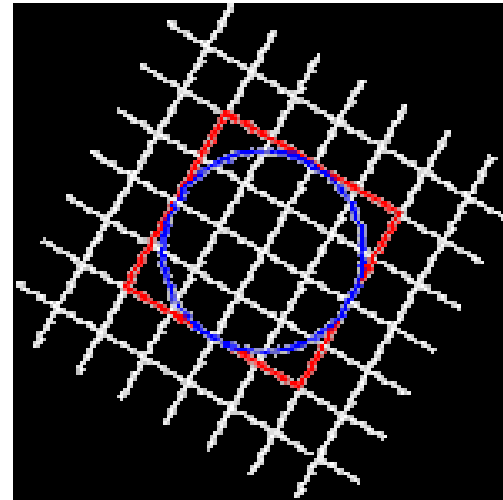
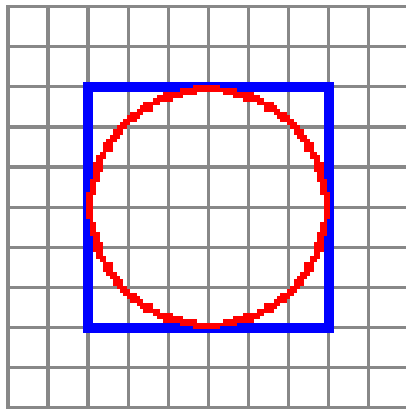
- Almost every course on 3D graphics begins like this one.
- Primitives are processed in a series of steps. Each step forwards its result onto the next step.
- It is a useful abstraction of projective rendering, as well as a block diagram for dedicated graphics hardware.

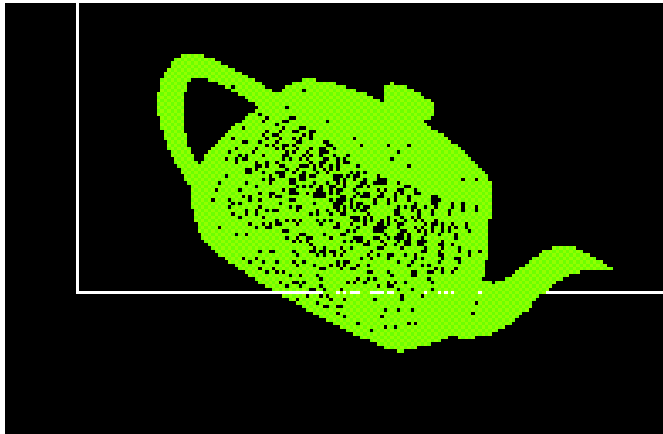
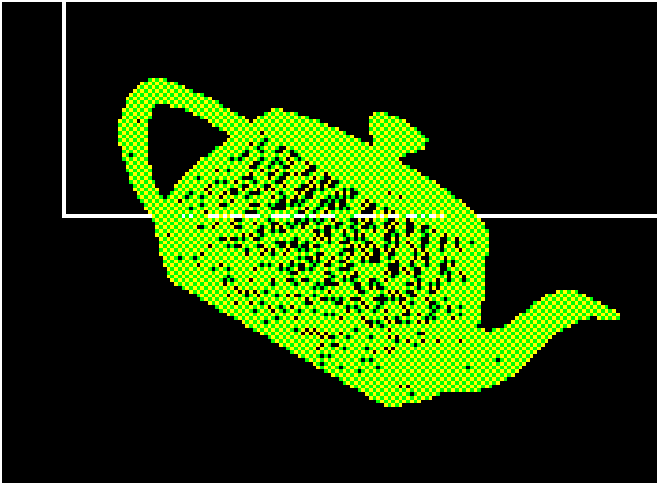
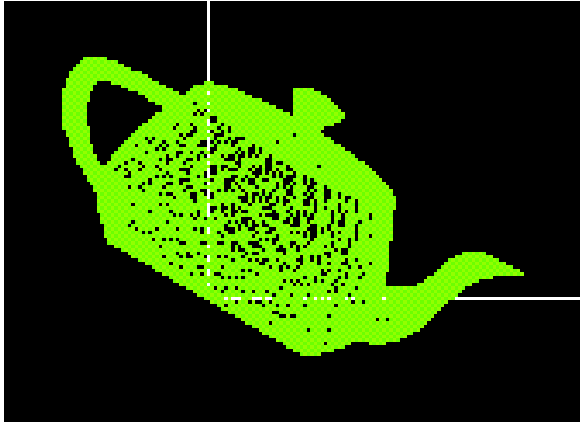
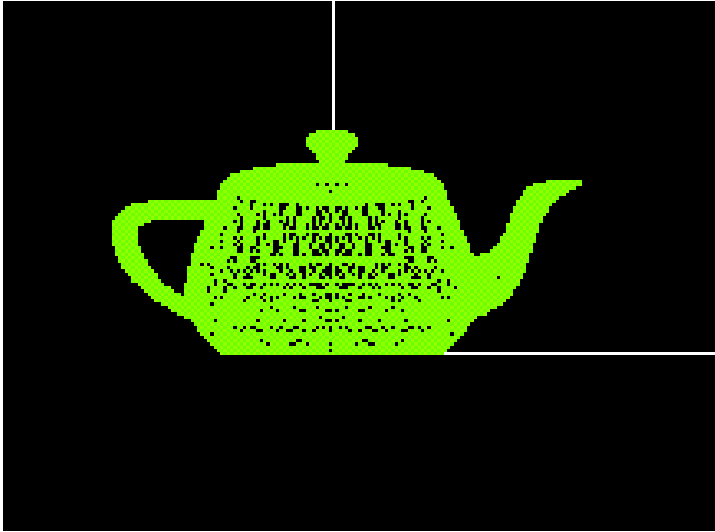
The OpenGL Rendering Pipeline

- First step: Modelling transformations
 - We start with 3D objects defined in their own coordinate system called “model space”
 - We apply transformations to move and orient the objects according to a common coordinate system called “world space”
 - All objects, light sources and cameras live in world space

Transformations

- Transformations are functions that map points from one place to another:





Transformations

- Almost every step in the rendering pipeline involves a change of coordinate systems.
- Transformations are central to understanding 3D computer graphics.
- The best way to implement transformations is with matrix operations.
- In the later classes we will discuss 2D matrices and transformations and then generalize to 3D matrices and transformations.
- Projections can also be expressed using matrices.

The OpenGL Rendering Pipeline

- Second Step: Trivial Rejections
 - OpenGL attempts to remove all objects that can't possibly be seen from the list of items to process.
 - This is an optimization, and not all implementations do it the same way

The OpenGL Rendering Pipeline

- Third Step: Illumination
 - Light sources influence the way objects are drawn
 - The lighting model (Phong illumination in OpenGL) influences the colors of all vertices
 - The shading model (Gouraud shading in OpenGL) influences how colors are interpolated across faces
 - Illumination also determines shadows

The OpenGL Rendering Pipeline

- Fourth Step: Viewing Transformation
 - We transform again, from world space to “eye space” coordinates
 - The camera is now located at the origin, and the viewing direction is aligned with some axis

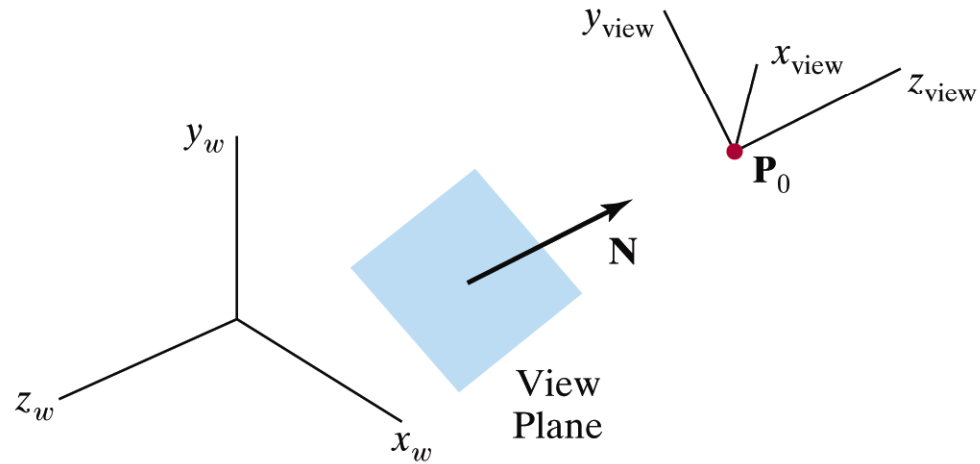


Figure 7-13

Orientation of the view plane and view-plane normal vector \mathbf{N} .

The OpenGL Rendering Pipeline

- Fifth Step: Clipping and Projection
 - A viewing volume is defined: objects located outside of this field of view are culled from the rendering.
 - Next, the remaining objects are projected into two dimensions, which transforms their coordinates into screen space

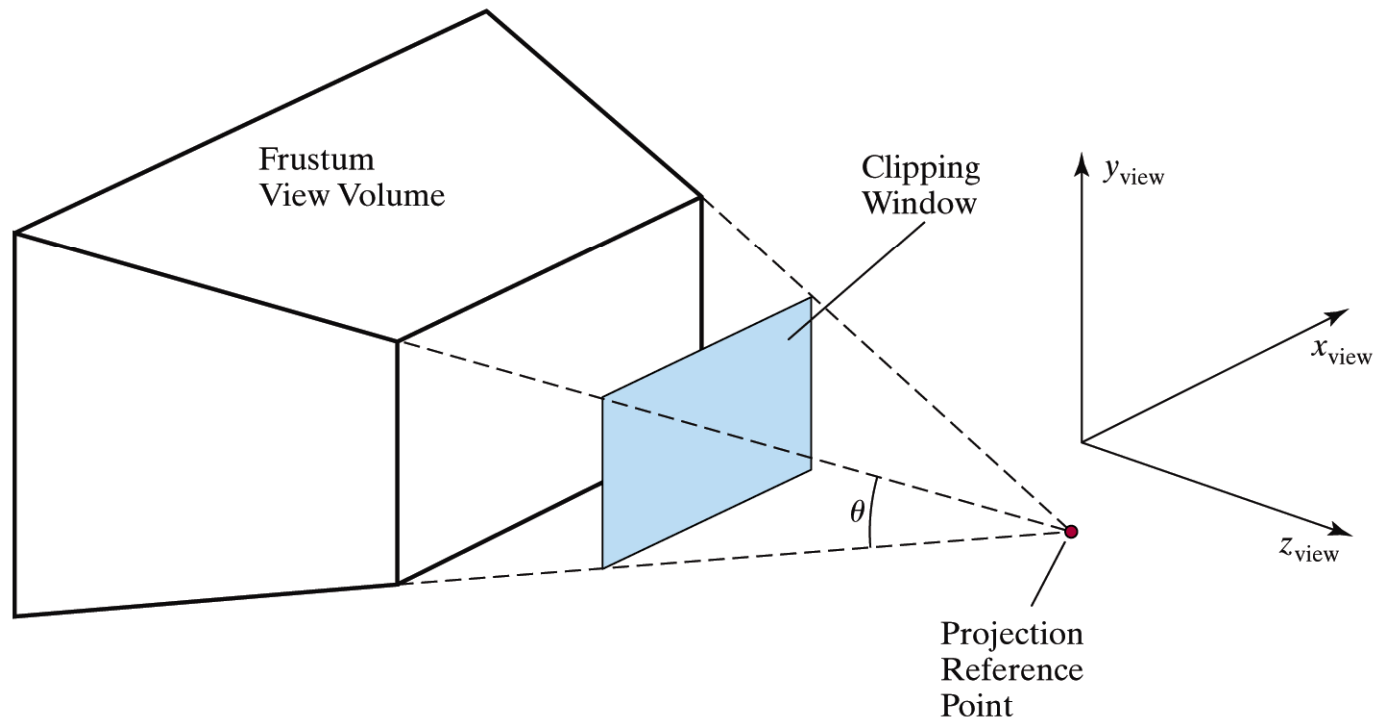


Figure 7-48

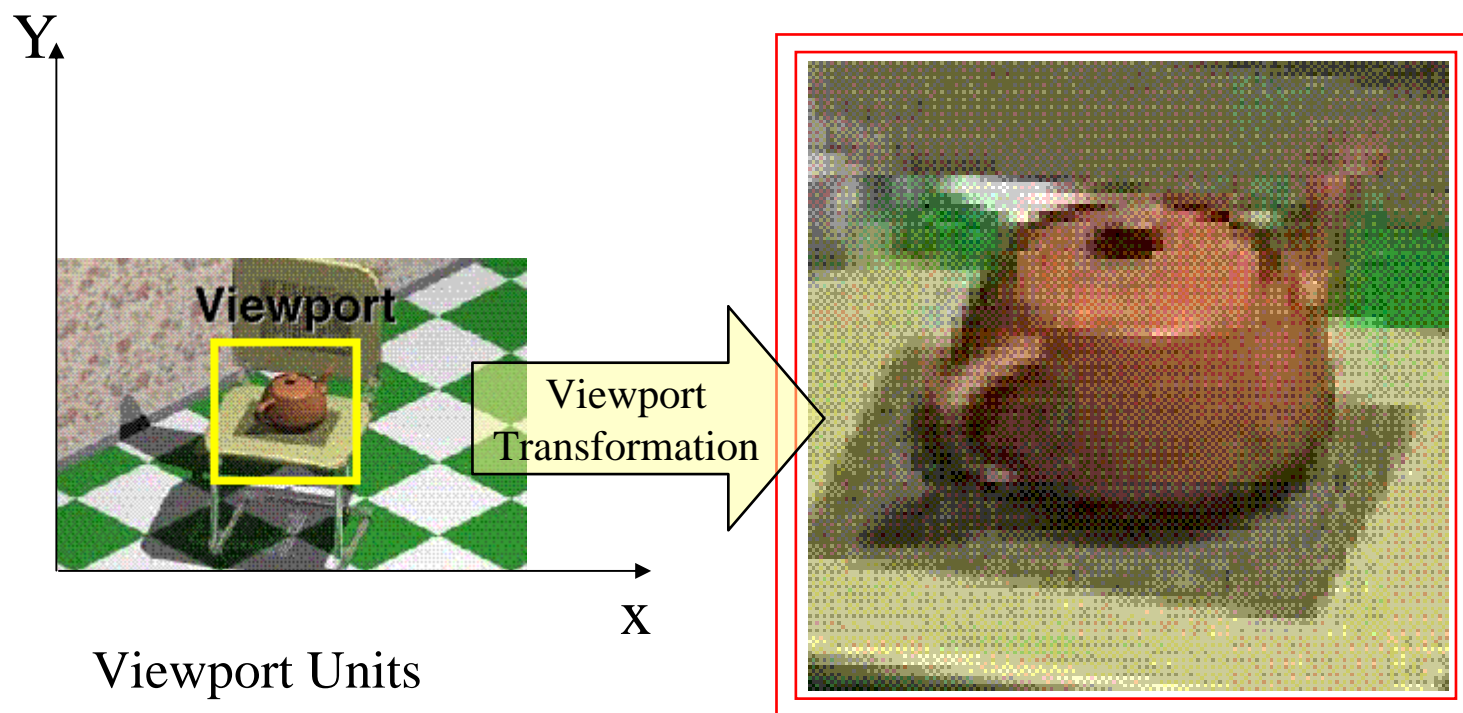
Field-of-view angle θ for a symmetric perspective-projection view volume, with the clipping window between the near clipping plane and the projection reference point.

The OpenGL Rendering Pipeline

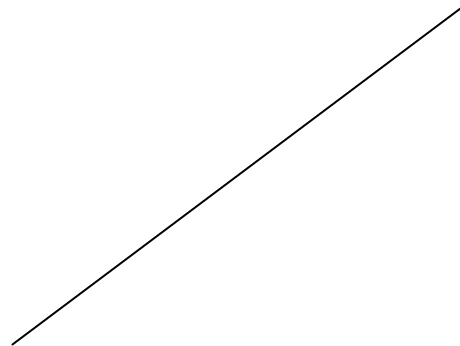
- Final Step: Rasterization and Display
 - Convert objects into pixels and display only those that are in front of any other (opaque) objects
 - Place pixels in the appropriate locations in the application window

Viewport Transformation

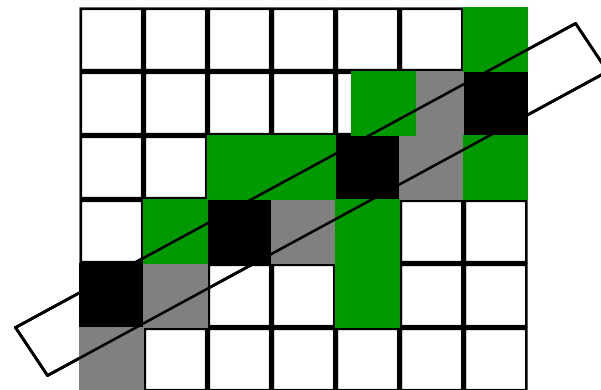
- Maps the 2D world in screen units into pixels in the display window.



Scan Conversion



From this $P = P_0 + D.t$
an ideal geometric primitive

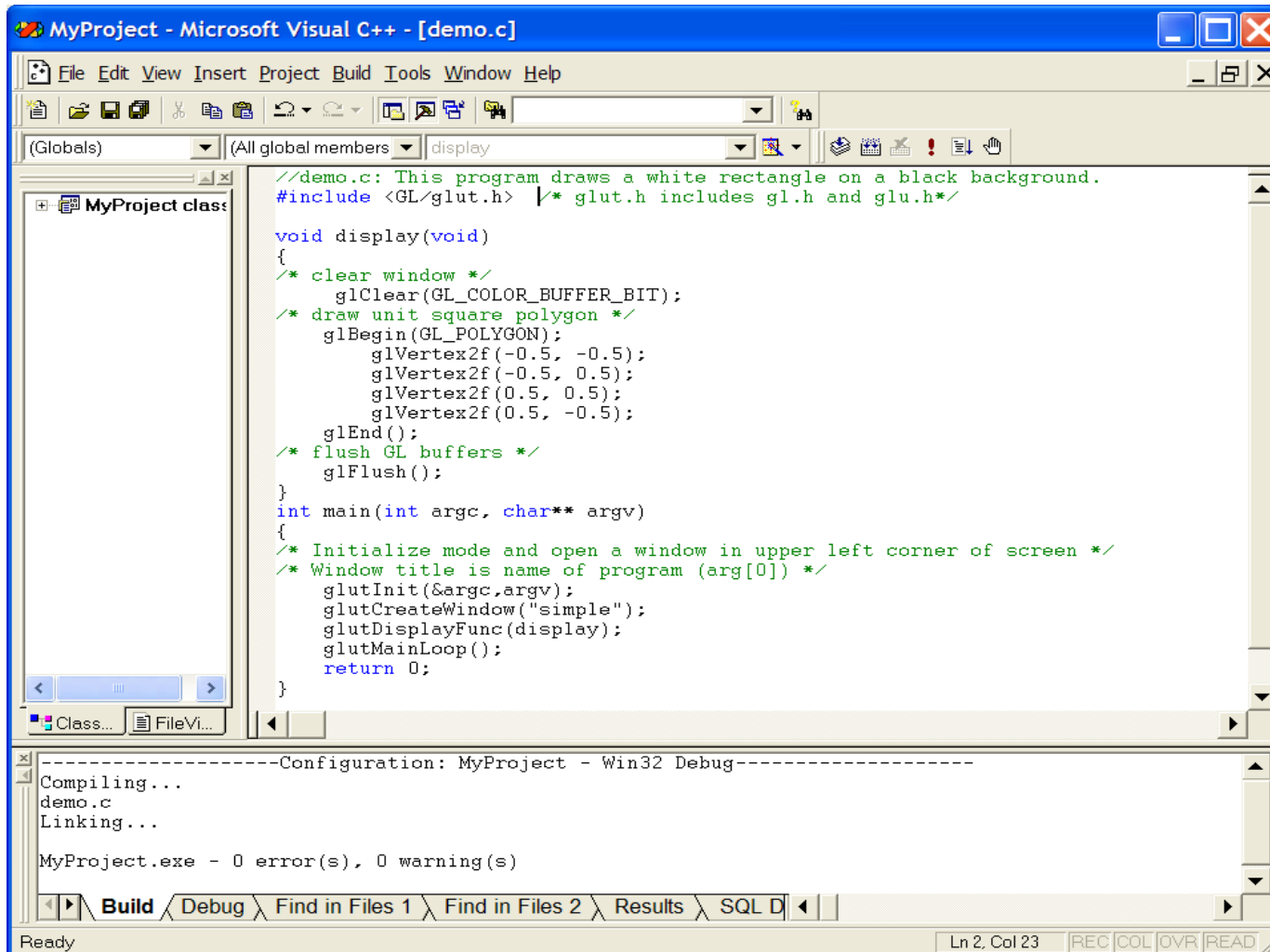


A bag of pixels of
different colours

Writing and Compiling your code

```
#include <GL/glut.h>
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_POLYGON);
            glVertex2f(-0.5, -0.5);
            glVertex2f(-0.5, 0.5);
            glVertex2f(0.5, 0.5);
            glVertex2f(0.5, -0.5);
        glEnd();
        glFlush();
}
int main(int argc, char** argv)
{
    glutInit(&argc,argv);
    glutCreateWindow("simple");
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Writing and Compiling your code



The screenshot shows the Microsoft Visual C++ IDE with a project named "MyProject". The main editor window displays the source code for a C program named "demo.c". The code uses GLUT to create a window titled "simple" and draw a white square on a black background. The code is as follows:

```
//demo.c: This program draws a white rectangle on a black background.
#include <GL/glut.h> /* glut.h includes gl.h and glu.h*/

void display(void)
{
    /* clear window */
    glClear(GL_COLOR_BUFFER_BIT);
    /* draw unit square polygon */
    glBegin(GL_POLYGON);
        glVertex2f(-0.5, -0.5);
        glVertex2f(-0.5, 0.5);
        glVertex2f(0.5, 0.5);
        glVertex2f(0.5, -0.5);
    glEnd();
    /* flush GL buffers */
    glFlush();
}

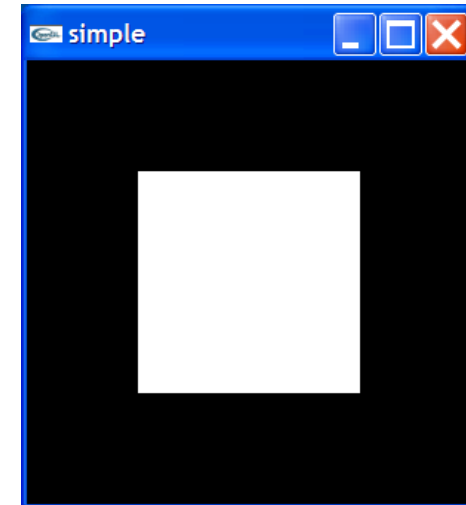
int main(int argc, char** argv)
{
    /* Initialize mode and open a window in upper left corner of screen */
    /* Window title is name of program (argv[0]) */
    glutInit(&argc, argv);
    glutCreateWindow("simple");
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

The IDE's output window at the bottom shows the compilation and linking process:

```
-----Configuration: MyProject - Win32 Debug-----
Compiling...
demo.c
Linking...

MyProject.exe - 0 error(s), 0 warning(s)
```

The status bar at the bottom indicates the current position is at line 2, column 23, and the keyboard shortcuts REC, COL, OVR, and READ are visible.



Graphics Definitions

- **Point**
 - a location in space, 2D or 3D
 - sometimes denotes one pixel
- **Line**
 - straight path connecting two points
 - infinitesimal width, consistent density
 - beginning and end on points
- **Vertex**
 - point in 2D or 3D
- **Edge**
 - line in 3D connecting two vertices
- **Polygon/Face/Facet**
 - arbitrary shape formed by connected vertices
 - fundamental unit of 3D computer graphics

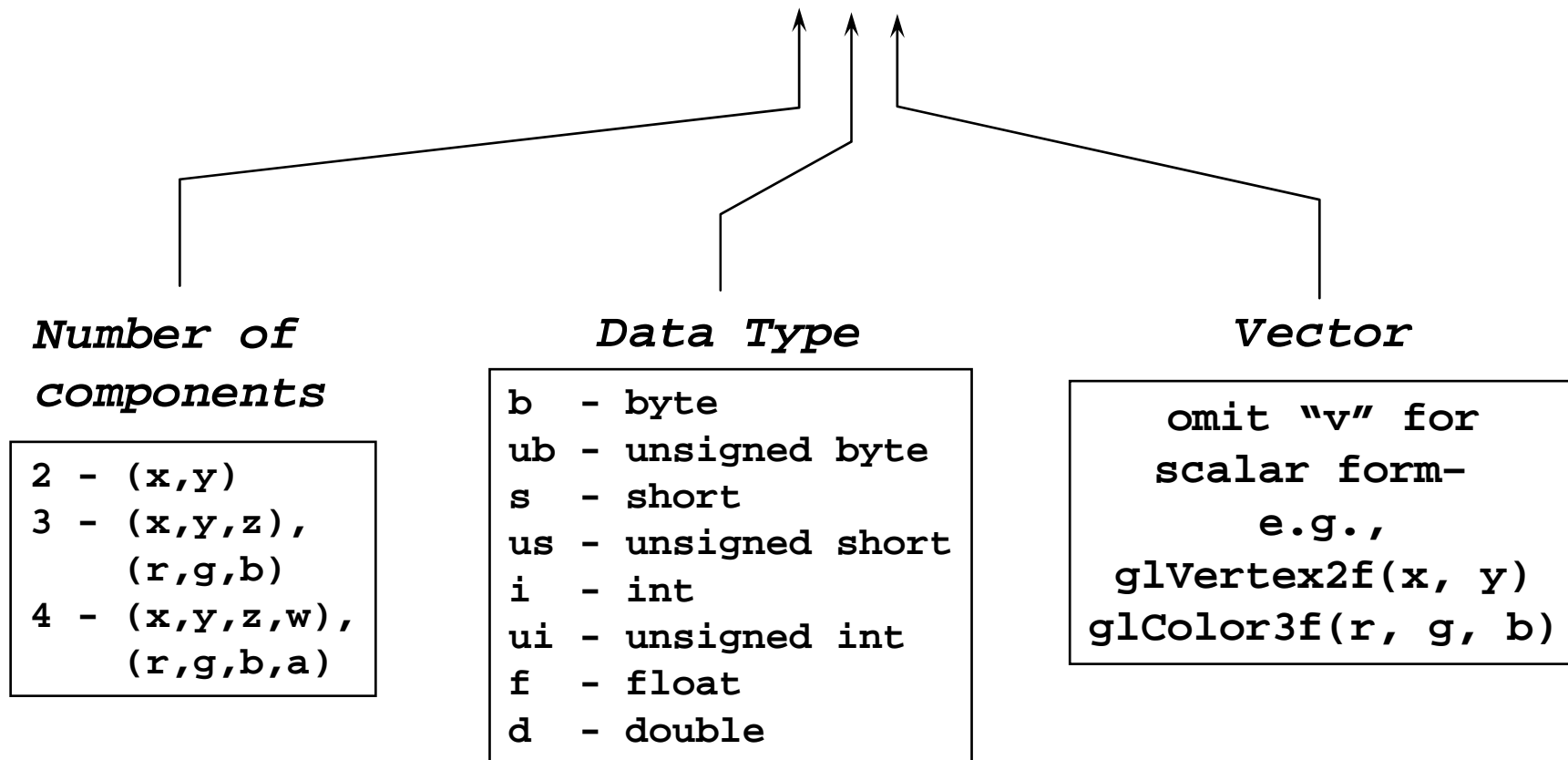
OpenGL Command Syntax

- All command names begin with gl
 - Ex.: `glVertex3f(0.0, 1.0, 1.0);`
- Constant names are in all uppercase
 - Ex.: `GL_COLOR_BUFFER_BIT`
- Data types begin with GL
 - Ex.: `GLfloat onevertex[3];`
- Most commands end in two characters that determine the data type of expected arguments
 - Ex.: `glVertex3f(...) => 3 GLfloat arguments`

OpenGL Command Formats

`glVertex3fv(v)`

`glColor3fv(v)`



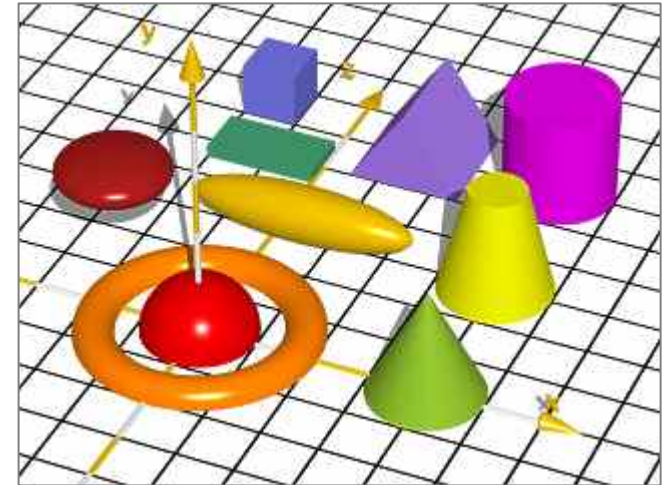
Specifying Object Vertices

- Every object is specified by vertices: `glVertex3f (2.0, 4.1, 6.0);`
 // specifies a vertex at the x, y, z coordinate (2.0, 4.1, 6.0).
 // The "3f" means 3 floating point coordinates.
 - Other examples:
`glVertex2i (4, 5);` // 2 integers for x and y. z = 0.
`glVertex3fv (vector);` // float vector[3] = {5.0, 3.2, 5.0};
- Current color affects any vertices
 - `glColor3f (0.0, 0.5, 1.0);`
 // no Red, half-intensity Green, full-intensity Blue
- Vertices are specified only between `glBegin(mode)` and `glEnd()`, usually in a counter-clockwise order for polygons.


```
glBegin (GL_TRIANGLES);
    glVertex2i (0, 0);
    glVertex2i (2, 0);
    glVertex2i (1, 1);
glEnd();
```

Object Specification

- Most APIs support a limited set of primitives including
 - Points (1D object)
 - Line segments (2D objects)
 - Polygons (3D objects)
 - Some curves and surfaces
 - Quadrics
 - Parametric polynomial
- All are defined through locations in space or *vertices*



```

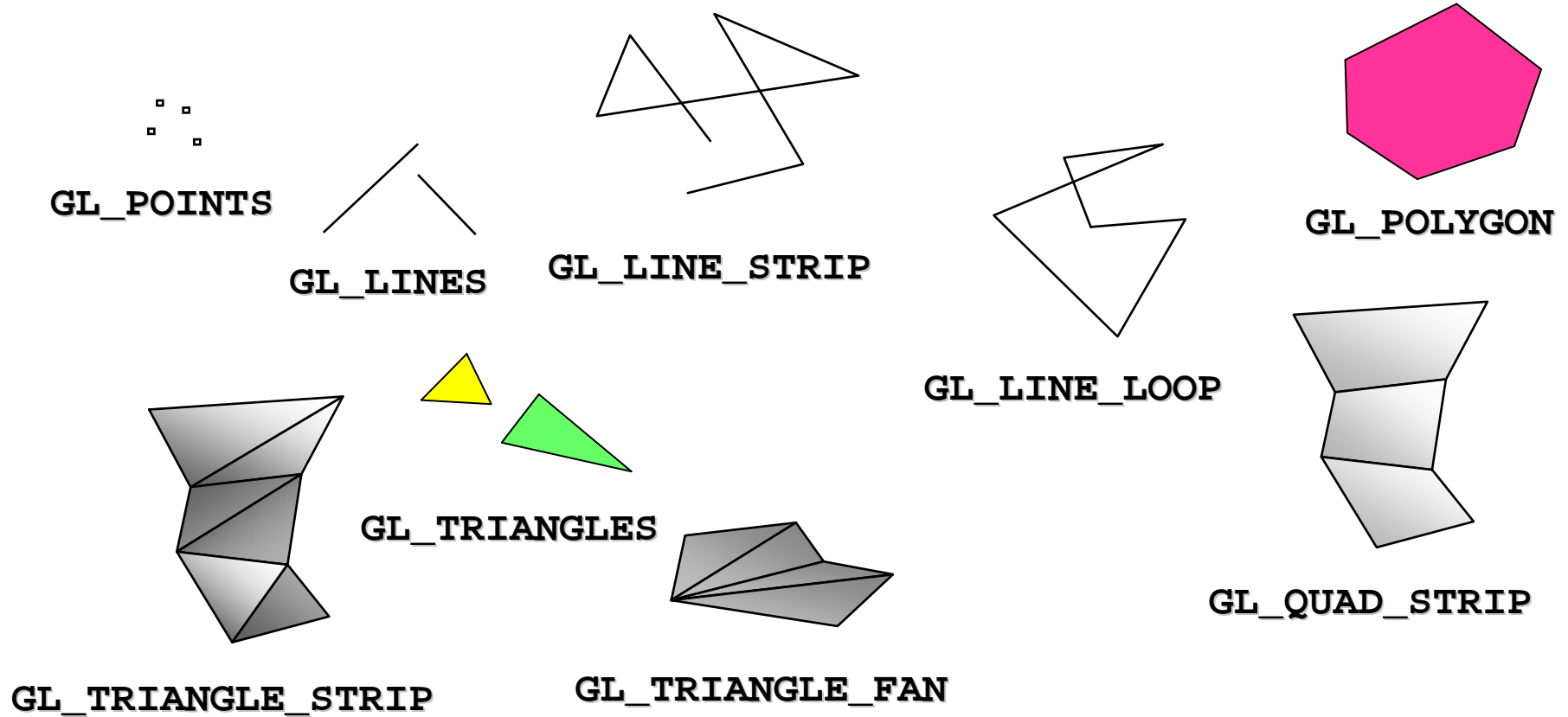
glBegin(GL_POLYGON)
  glVertex3f(0.0, 0.0, 0.0);
  glVertex3f(0.0, 1.0, 0.0);
  glVertex3f(0.0, 0.0, 1.0);
glEnd( );

```

type of object
 location of vertex
 end of object definition

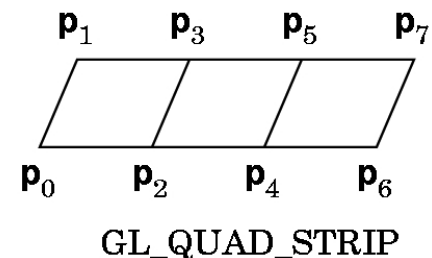
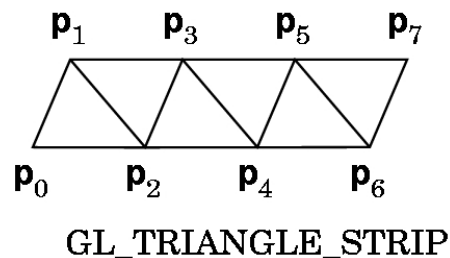
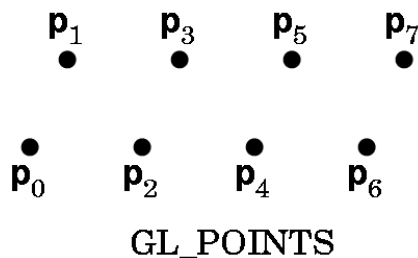
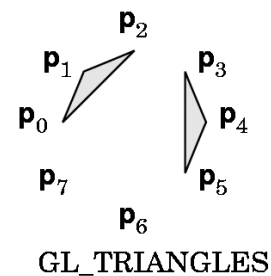
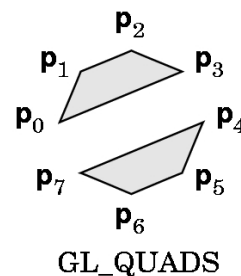
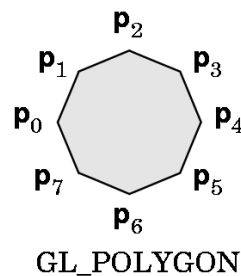
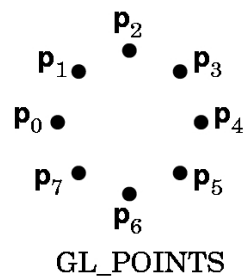
OpenGL Primitives

Primitives:



Polygon Types

- **Polygons** (`GL_POLYGON`) successive vertices define line segments, last vertex connects to first
- **Triangles** and **Quadrilaterals** (`GL_TRIANGLES`, `GL_QUADS`) successive groups of 3 or 4 interpreted as triangles or quads
- **Strips** and **Fans** (`GL_TRIANGLE_STRIP`, `GL_QUAD_STRIP`, `GL_TRIANGLE_FAN`) joined triangles or quads that share vertices



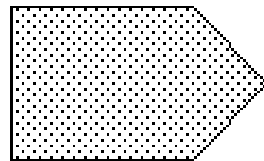
Drawing Vertices

- `void glVertex{234}{sifd}[v](TYPEcoords);`

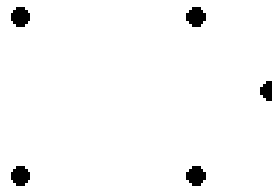
```
glBegin(GL_POINTS);  
    glVertex2s(2, 3); // point (2, 3, 0)  
    glVertex3d(0.0, 0.0, 3.1415926535898);  
    glVertex4f(2.3, 1.0, -2.2, 2.0); // point (1.15, 0.5, -1.1)  
    GLdouble dvect[3] = {5.0, 9.0, 1992.0};  
    glVertex3dv(dvect);  
glEnd();
```

Drawing Filled Polygons

```
glBegin( GL_POLYGON );  
    glVertex2f( 0.0, 0.0 );  
    glVertex2f( 0.0, 3.0 );  
    glVertex2f( 3.0, 3.0 );  
    glVertex2f( 4.0, 1.5 );  
    glVertex2f( 3.0, 0.0 );  
glEnd( );
```



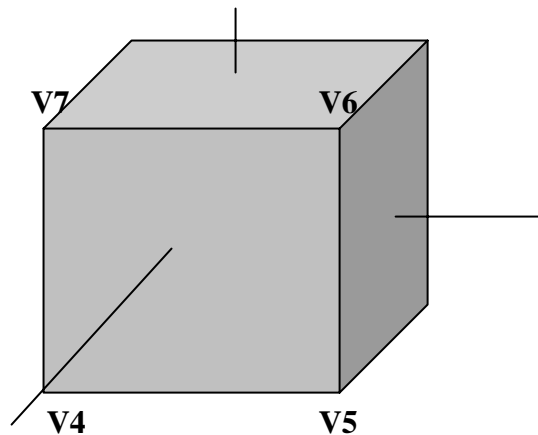
GL_POLYGON



GL_POINTS

OpenGL Example

```
void drawOneCubeface(size)
{
    static GLfloat v[8][3];
    v[0][0] = v[3][0] = v[4][0] = v[7][0] = -size/2.0;
    v[1][0] = v[2][0] = v[5][0] = v[6][0] = size/2.0;
    v[0][1] = v[1][1] = v[4][1] = v[5][1] = -size/2.0;
    v[2][1] = v[3][1] = v[6][1] = v[7][1] = size/2.0;
    v[0][2] = v[1][2] = v[2][2] = v[3][2] = -size/2.0;
    v[4][2] = v[5][2] = v[6][2] = v[7][2] = size/2.0;
    glBegin(GL_POLYGON);
        glVertex3fv(v[0]);
        glVertex3fv(v[1]);
        glVertex3fv(v[2]);
        glVertex3fv(v[3]);
    glEnd();
}
```



Other Commands in glBegin / glEnd blocks

- Not every OpenGL command can be located in such a block. Those that can be included, among others are:
 - glColor
 - glNormal (to define a normal vector)
 - glTexCoord (to define texture coordinates)
 - glMaterial (to set material properties)

Example

```
glBegin( GL_POLYGON );  
    glColor3f( 1.0, 1.0, 0.0 );  
    glVertex3f( 0.0, 0.0, 0.0 );  
    glColor3f( 0.0, 1.0, 1.0 );  
    glVertex3f( 5.0, 0.0, 0.0 );  
    glColor3f( 1.0, 0.0, 1.0 );  
    glVertex3f( 0.0, 5.0, 0.0 );  
glEnd();
```