Computer Graphics -Basics of OpenGL

Junjie Cao @ DLUT

Spring 2017

http://jjcao.github.io/ComputerGraphics/

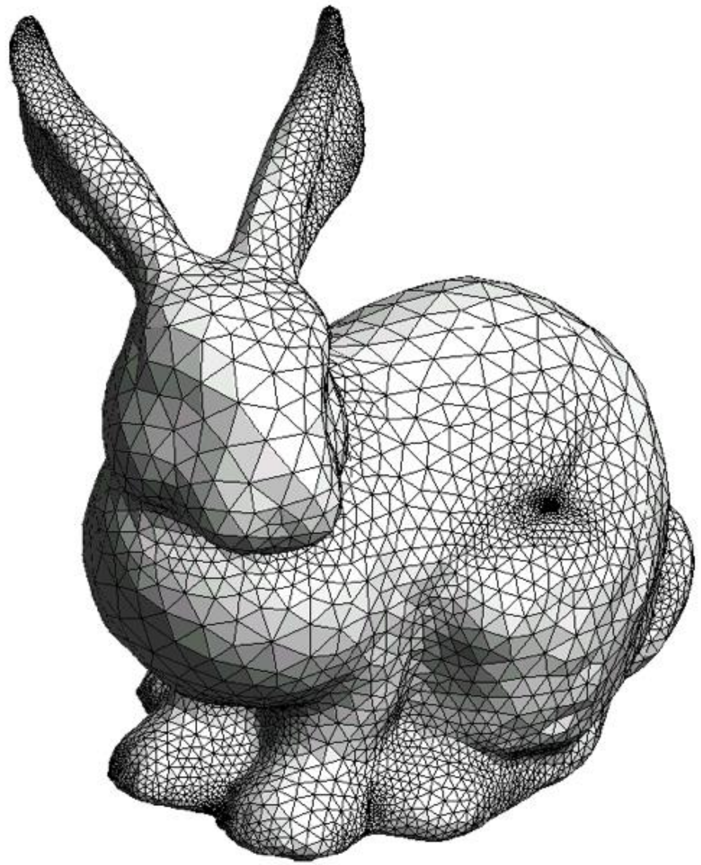
Primitives

- Specified via vertices

- General scheme

```
glBegin(type);
    glVertex3f(x1,y1,z1);
    …
    glVertex3f(xN,yN,zN);
glEnd();
```

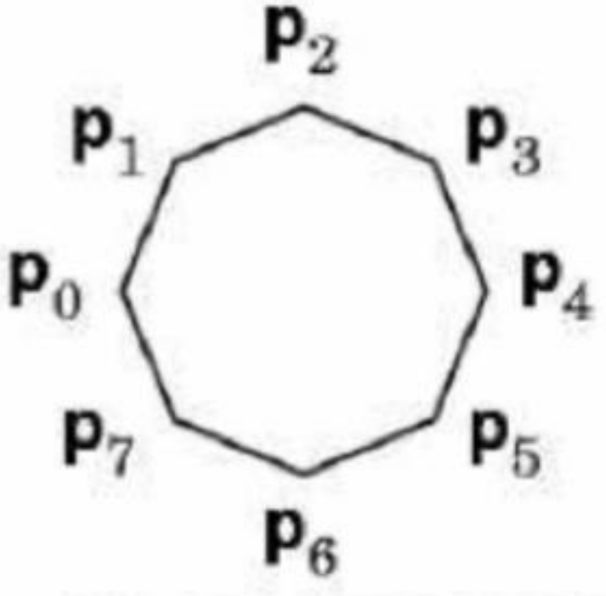
- **type** determines interpretation of vertices
- Can use glVertex2f(x,y) in 2D



Example: Draw Square Outline

- **Type = GL_LINE_LOOP**

```
glBegin(GL_LINE_LOOP);
    glVertex3f(0.0,0.0,0.0);
    glVertex3f(1.0,0.0,0.0);
    glVertex3f(1.0,1.0,0.0);
    glVertex3f(0.0,1.0,0.0);
glEnd()
```



GL_LINE_LOOP

- Calls to other functions are allowed betwen glBegin(Type) and glEnd()

OpenGL Geometric Drawing Primitives

- OpenGL geometric primitives can create a set of points, a line, or a polygon from vertices

- OpenGL support **ten** types of primitives

- A drawing primitive must start with

  glBegin(Type);
- And finish with

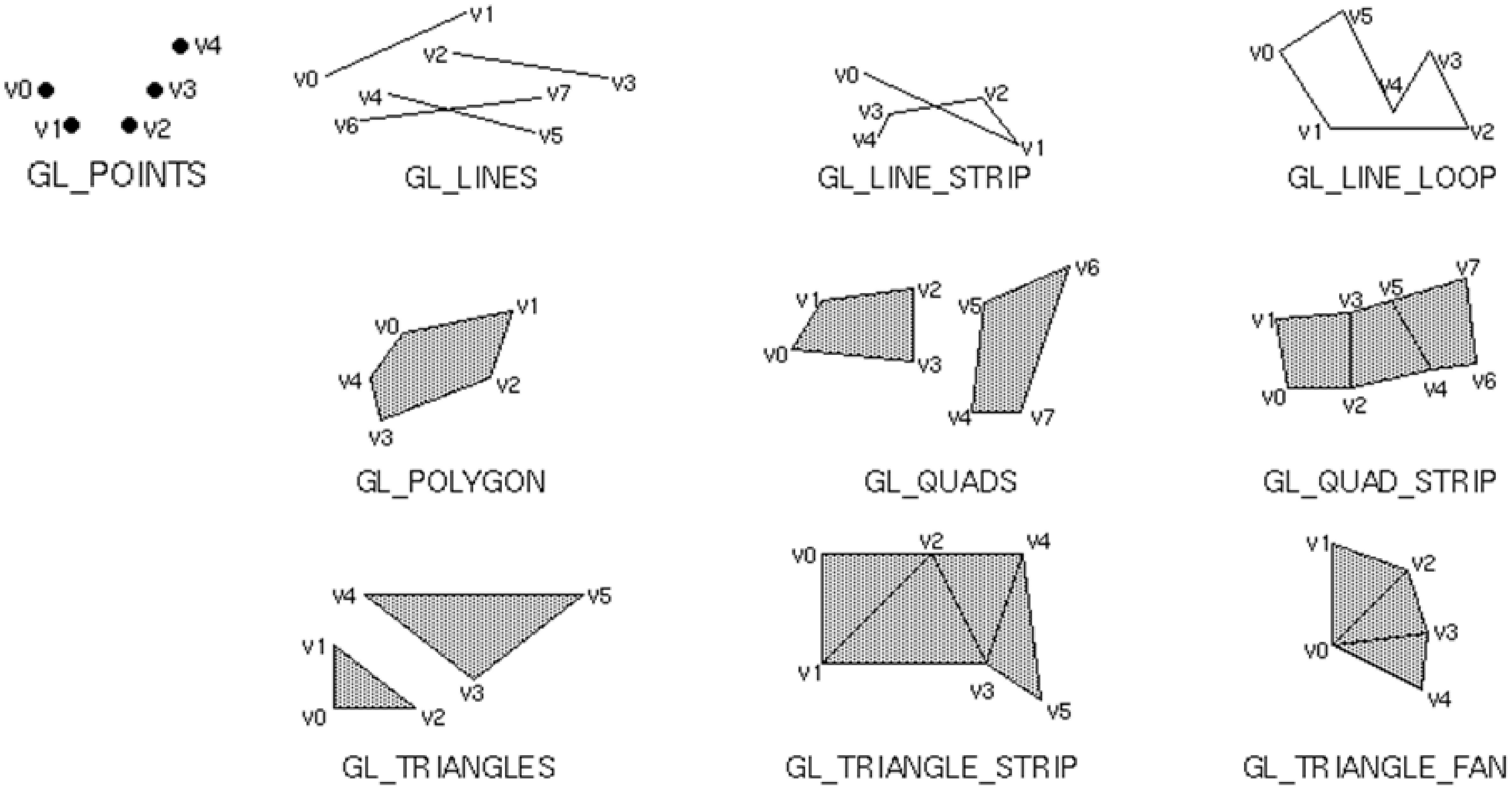
  glEnd();

- Calls to other functions are allowed betwen glBegin(Type) and glEnd()

- Between them the primitive
- Type=GL_POLYGON

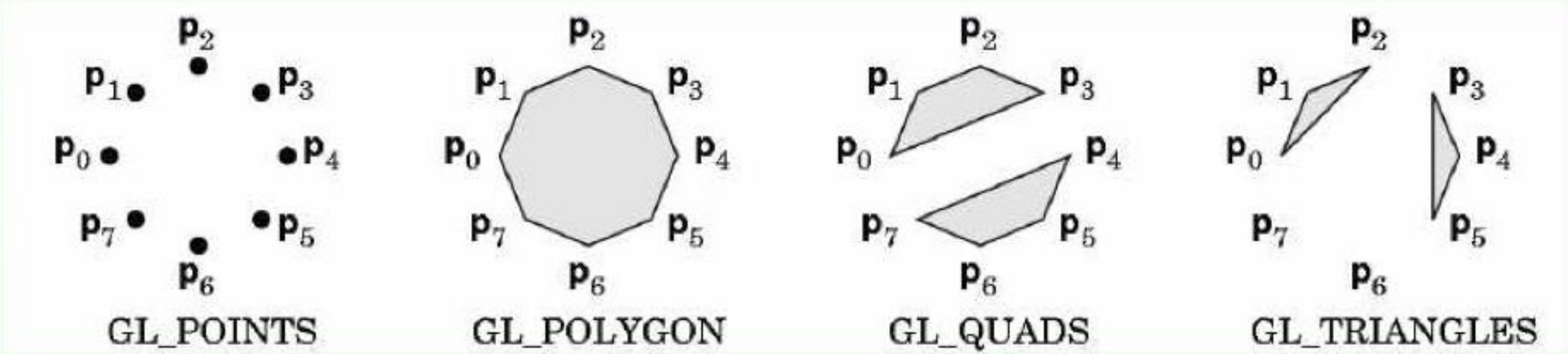
```
glBegin(GL_POLYGON);
    glVertex2f(-0.5, -0.5);
    glVertex2f(-0.5, 0.5);
    glVertex2f( 0.5, 0.5);
    glVertex2f( 0.5, -0.5);
glEnd();
```

OpenGL Geometric Drawing Primitives (cont)



Polygons

- **Polygons enclose an area**

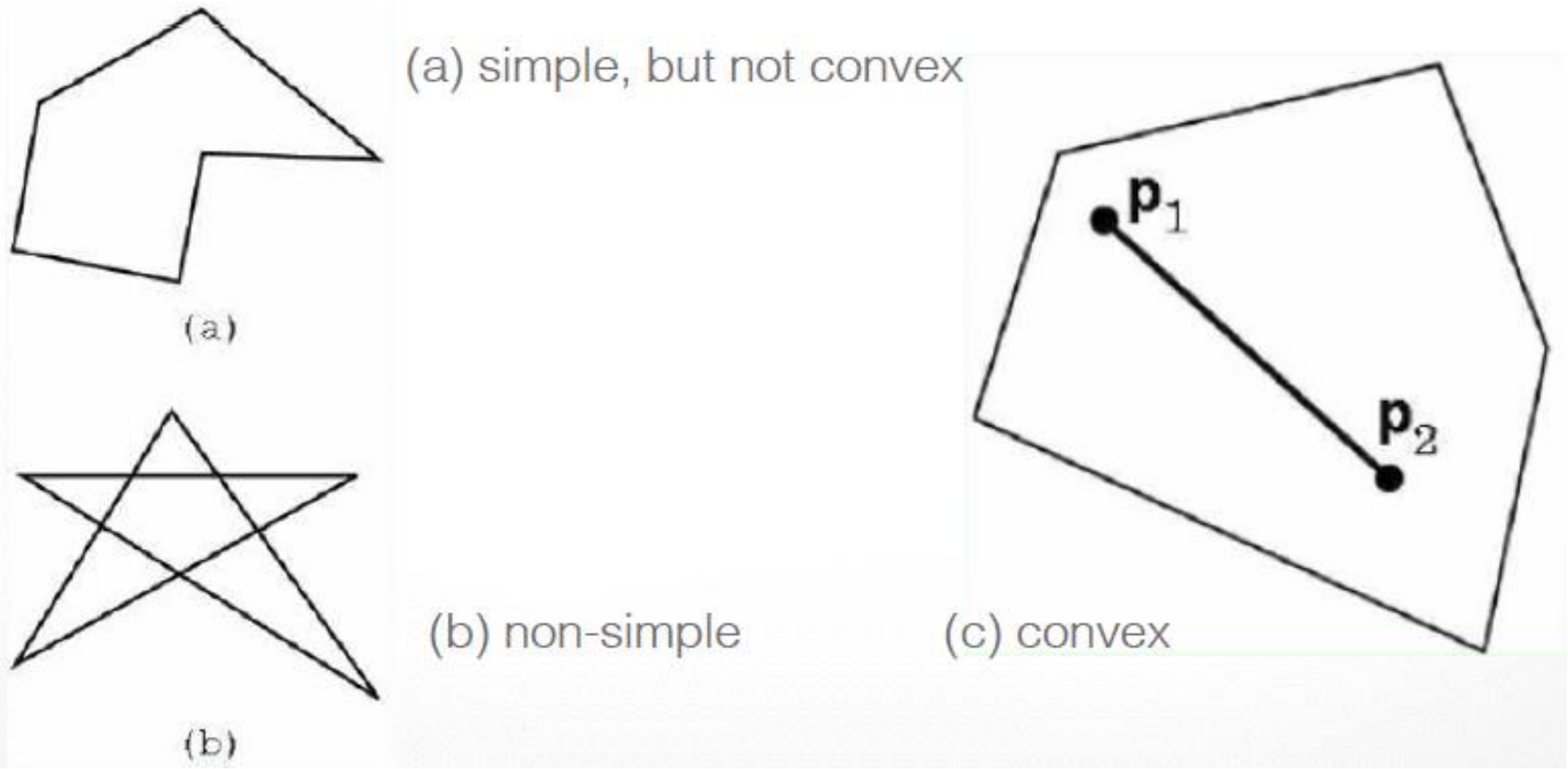


- Rendering of area (fill) depends on attributes

- **All vertices must be in one plane in 3D**

Polygons Restrictions

- OpenGL Polygons must be **simple**
- OpenGL Polygons must be **convex**



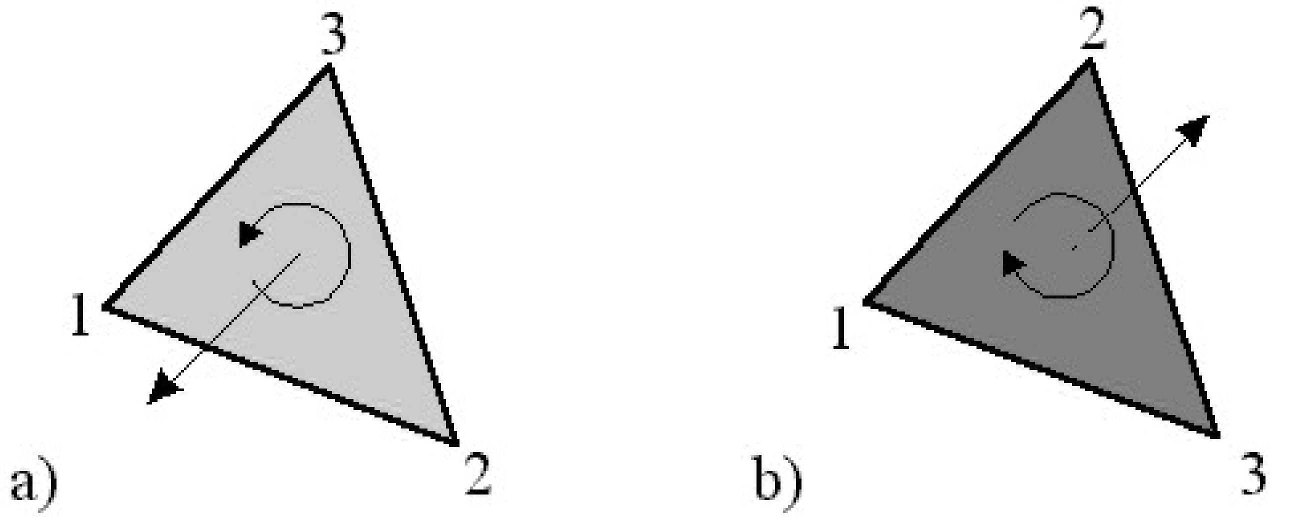
Why **Polygons Restrictions?**

- Non-convex and non-simple polygons are **expensive to process and render**
- Convexity and simplicity is **expensive to test**
- Behavior of **OpenGL** implementation on disallowed polygons is **“undefined”**
- Some tools in GLU for decomposing complex polygons (**tesselation**)
- **Triangles are most efficient**

Front/Back Rendering

- Polygons have a **front and a back**, possibly with **different attributes**!
- The **ordering of vertices** in the list determines which is the front side:

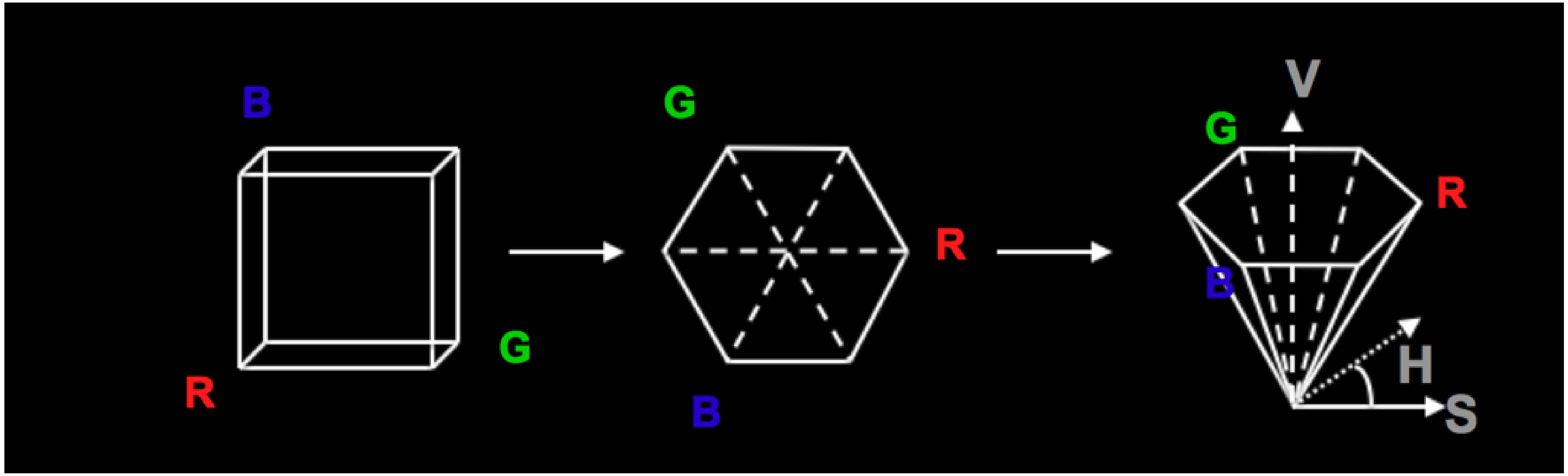
```
glBegin(GL_POLYGON);
    glVertex2f(-0.5, -0.5);
    glVertex2f(-0.5, 0.5);
    glVertex2f( 0.5, 0.5);
    glVertex2f( 0.5, -0.5);
glEnd();
```



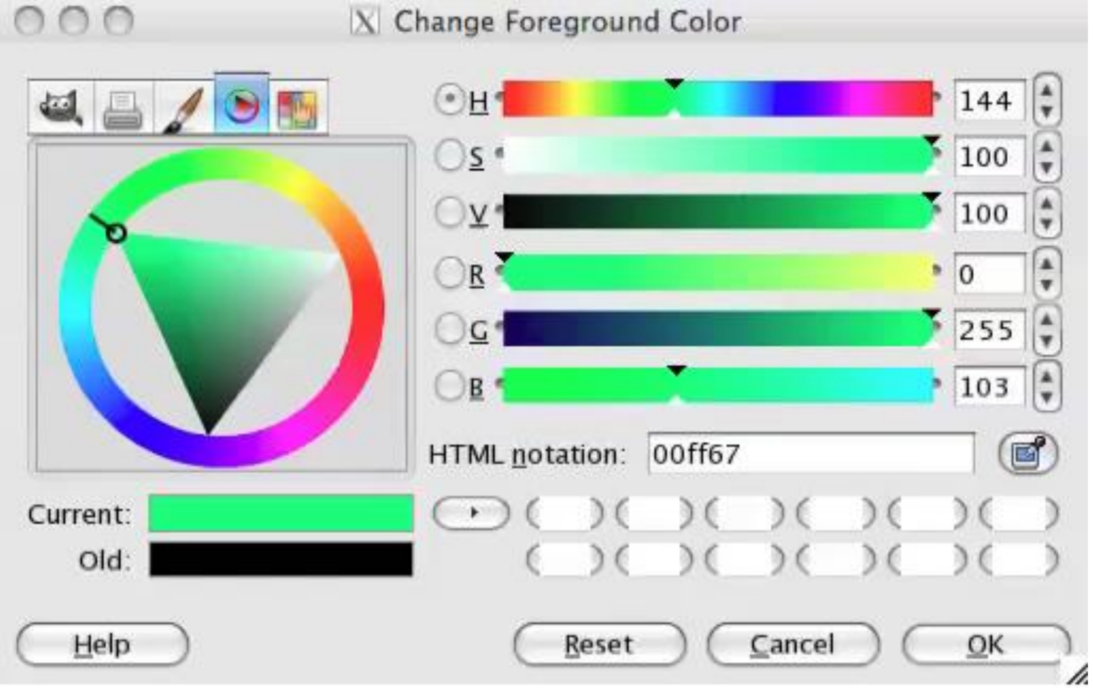
Attributes: Color, Shading, Reflections

- Part of the OpenGL **state**
- Set **before** primitives are drawn
- **Remain in effect until changed!**

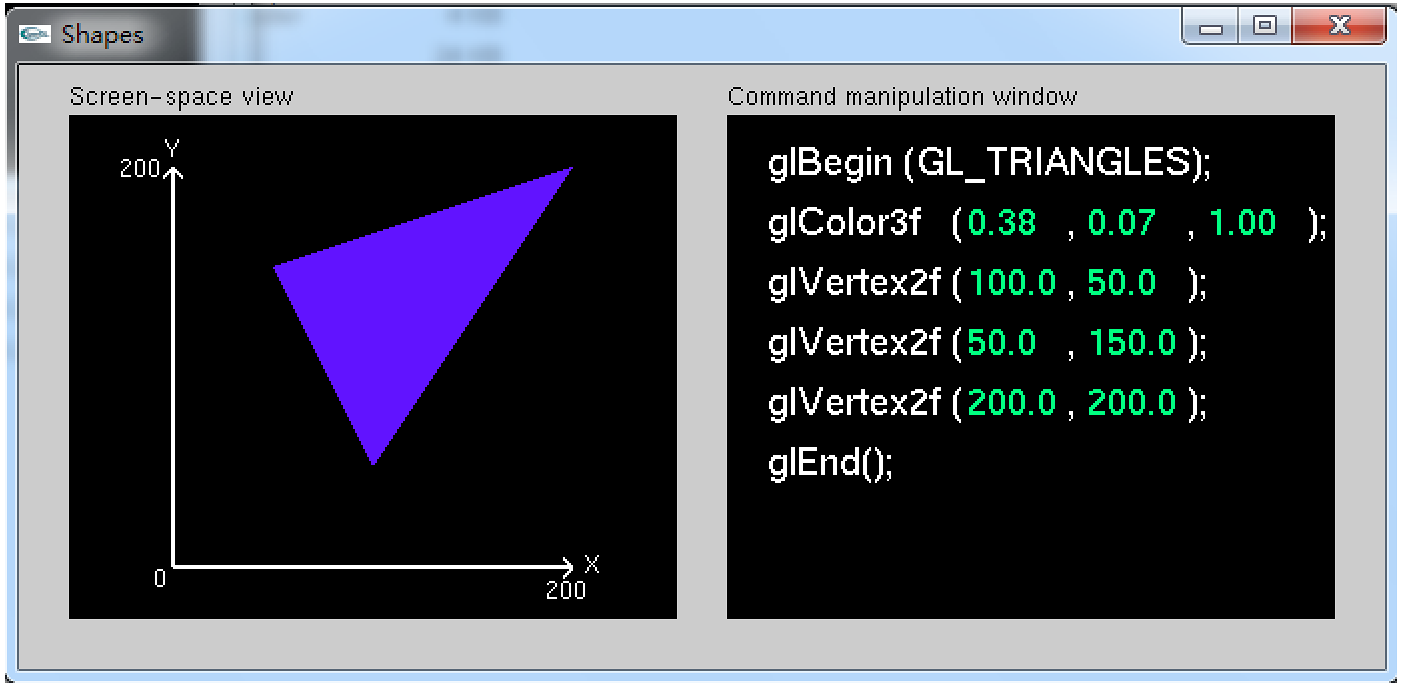
RGB vs HSV



- **RGB (Red, Green, Blue)**
  - Convenient for display
  - Can be unintuitive (3 floats in OpenGL)
- **HSV (Hue, Saturation, Value)**
  - Hue: what color?
  - Saturation: how far away from gray?
  - Value: how bright?
- Other formats for **movies and printing**



OpenGL 1.x: Geometric Drawing Primitives (cont)



Nate_Robins_tutorials: Shapes

Draw a complicated 3D Object in OpenGL 1.x

```
void DrawMeshWire( CMesh *m )
  glBegin( GL_TRIANGLES );
      for ( int i = 0; i < m->numFaces * 3; i+=3 )  {
          glVertex3f( m->vertex[m->faces[i]*3], m->vertex[m->faces[i]*3+1],
                            m->vertex[m->faces[i]*3+2] );
          glVertex3f( m->vertex[m->faces[i+1]*3], m->vertex[m->faces[i+1]*3+1],
                            m->vertex[m->faces[i+1]*3+2] );
          glVertex3f( m->vertex[m->faces[i+1]*3], m->vertex[m->faces[i+1]*3+1],
                            m->vertex[m->faces[i+1]*3+2] );
      }
  glEnd();
```

Use GLUT (OpenGL Utility Toolkit)

- For fast prototyping, you can use GLUT to interface with different window systems
- GLUT is a window independent API – programs written using OpenGL an d GLUT can be ported to X windows, MS windows, and Macintosh with no effort
- GLUT does not contain all the bells and whistles though (no sliders, no dialog boxes, no menu bar, etc)

Example: Drawing a shaded polygon

- Initialization: the “main” function

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

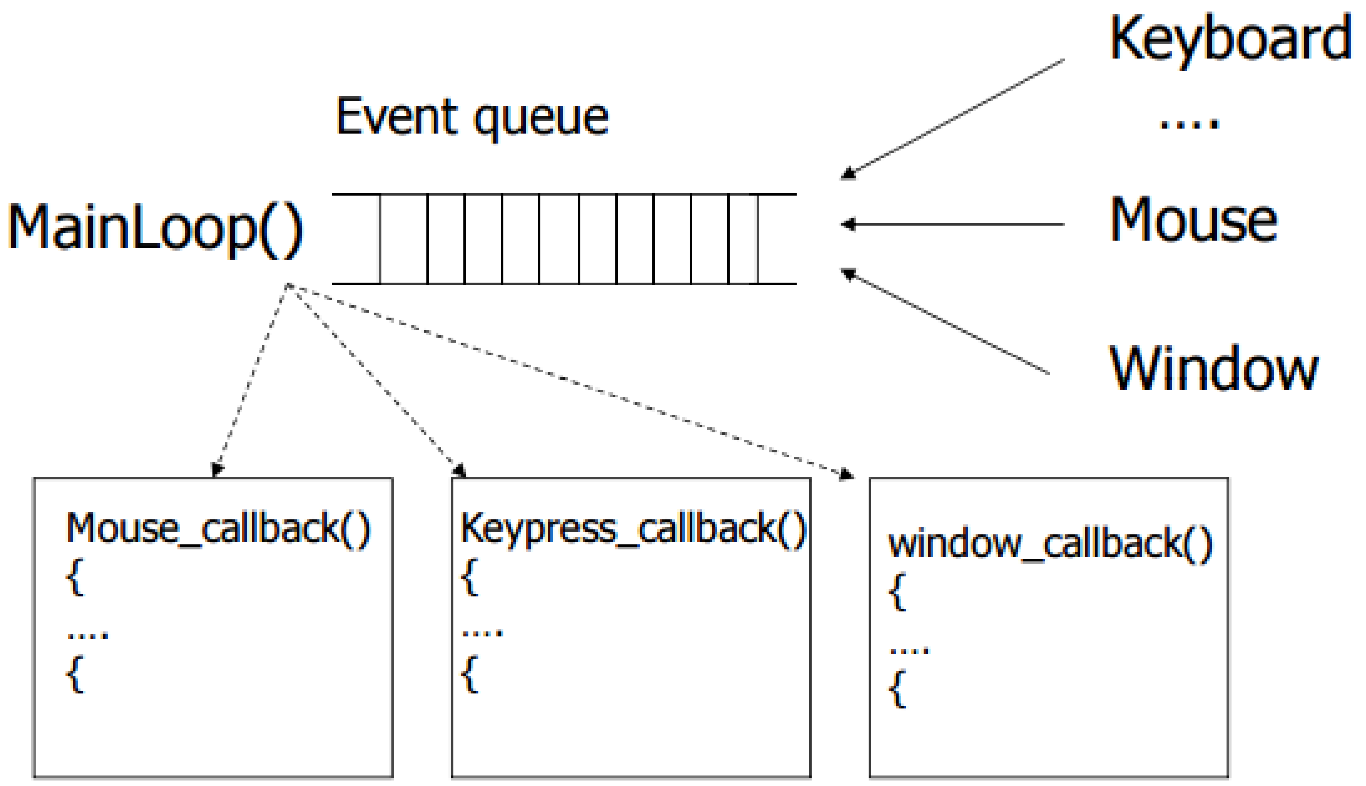
```
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
    glutInitWindowSize(500,500);
    glutInitWindowPosition(100,100);
    glutCreateWindow(argv[0]);
    init();
    …
```

GLUT Callbacks

- Window system **independent** interaction

- glutMainLoop processes events

```
    …
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```



Initializing Attributes

- Separate in “init” function

```
void init()
{
    glClearColor (0.0,0.0,0.0,0.0);
    // glShadeModel (GL_FLAT);
    glShadeModel (GL_SMOOTH);
}
```

The Display Callback

- The routine where you render the object

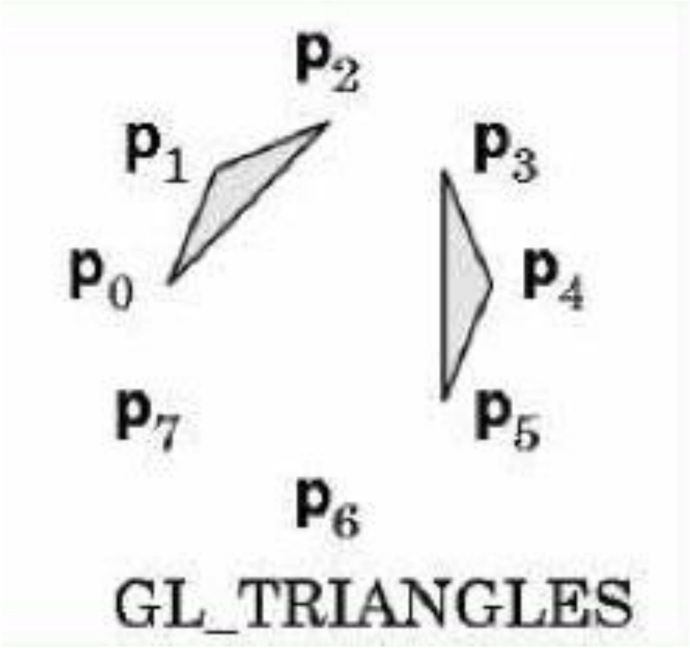
- Install with glutDisplayFunc(display)

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT); // clear buffer
    setupCamera(); // set up camera
    triangle(); // draw triangle
    glutSwapBuffers(); // force display
}
```

Drawing in OpenGL 1.x

- In world coordinates; remember state!

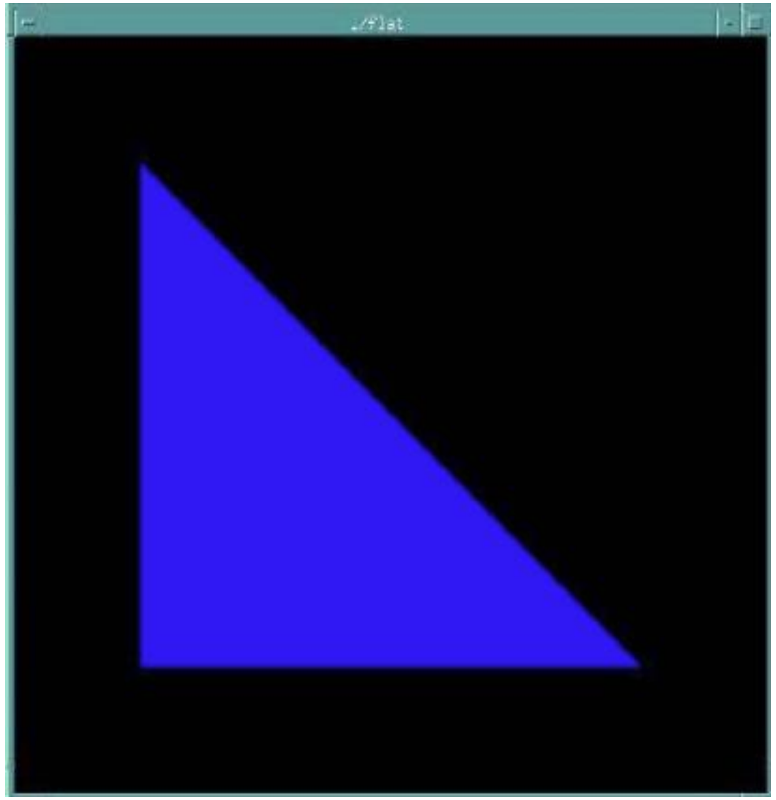
```
void triangle()
{
    glBegin(GL_TRIANGLES);
        glColor3f(1.0,0.0,0.0); // red
        glVertex2f(5.0,5.0);
        glColor3f(0.0,1.0,0.0); // green
        glVertex2f(25.0,5.0);
        glColor3f(0.0,0.0,1.0); // blue
        glVertex2f(5.0,25.0);
    glEnd();
}
```



The Image

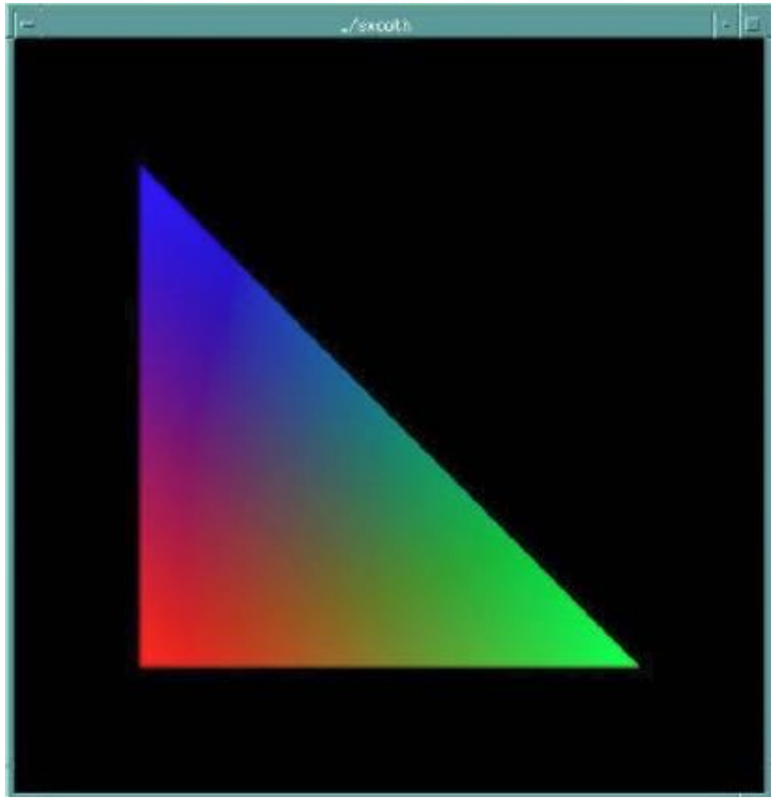
glShadeModel(GL_FLAT)

color of last vertex



glShadeModel(GL_SMOOTH)

each vertex separate color smoothly interpolated



Flat vs Smooth Shading

glShadeModel(GL_FLAT)

glShadeModel(GL_SMOOTH)

Projection

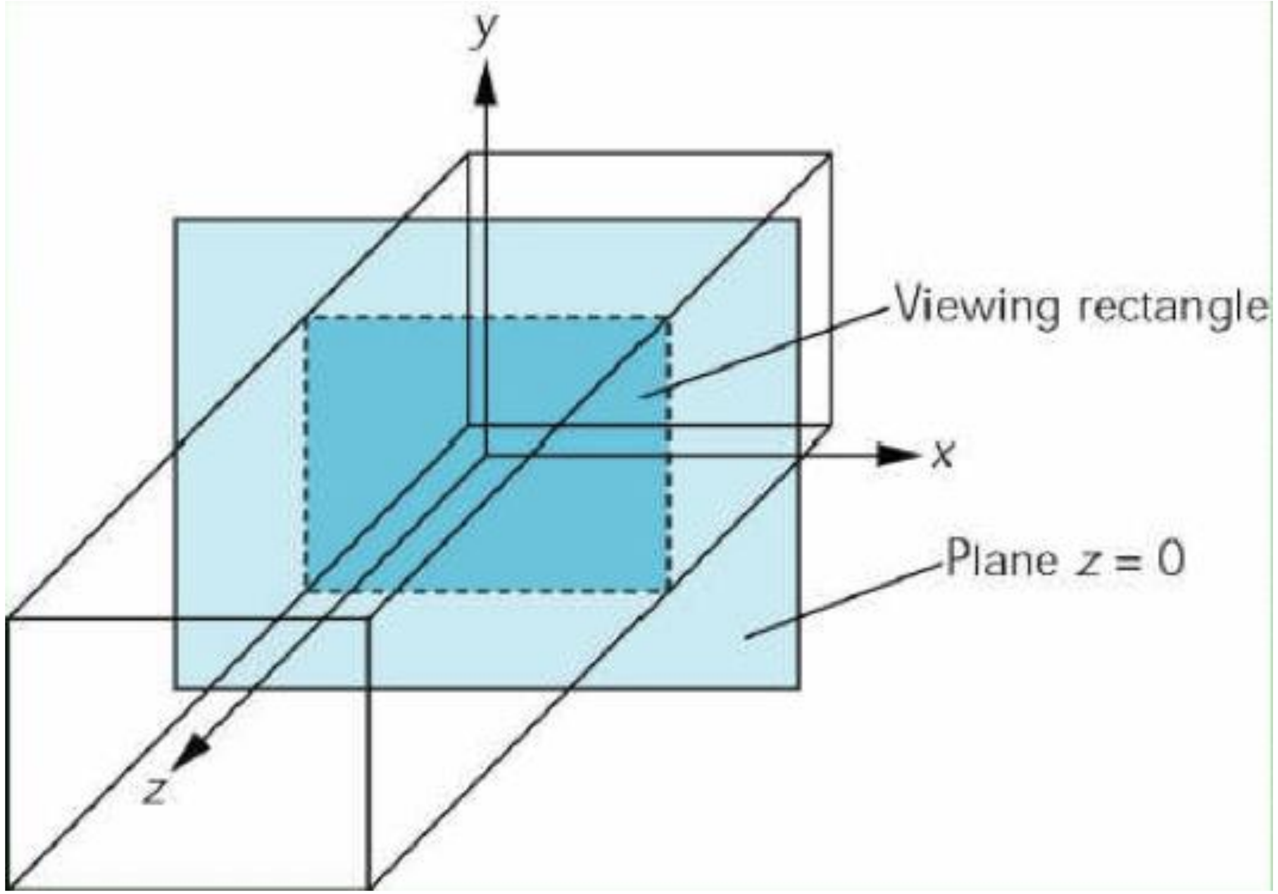
- Mapping world to screen coordinates

```
void reshape (int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if(w<=h)
        gluOrtho2D(0.0,30.0,0.0,30.0 * (GLfloat) h/(GLfloat) w);
    else
        gluOrtho2D(0.0,30.0 * (GLfloat) w/(GLfloat) h, 0.0,30.0);
    glMatrixMode(GL_MODELVIEW);
}
```

GL_PROJECTION and GL_MODELVIEW will be discussed later

Orthographic Projection

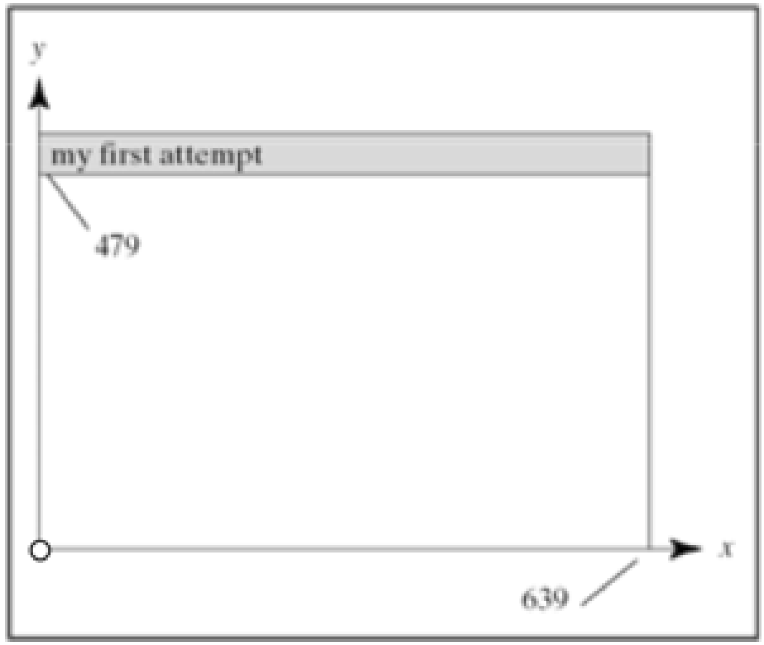
- gluOrtho2D(left, right, bottom, top)
- In world coordinates!



Screen coordinates

- Bottom left corner is origin

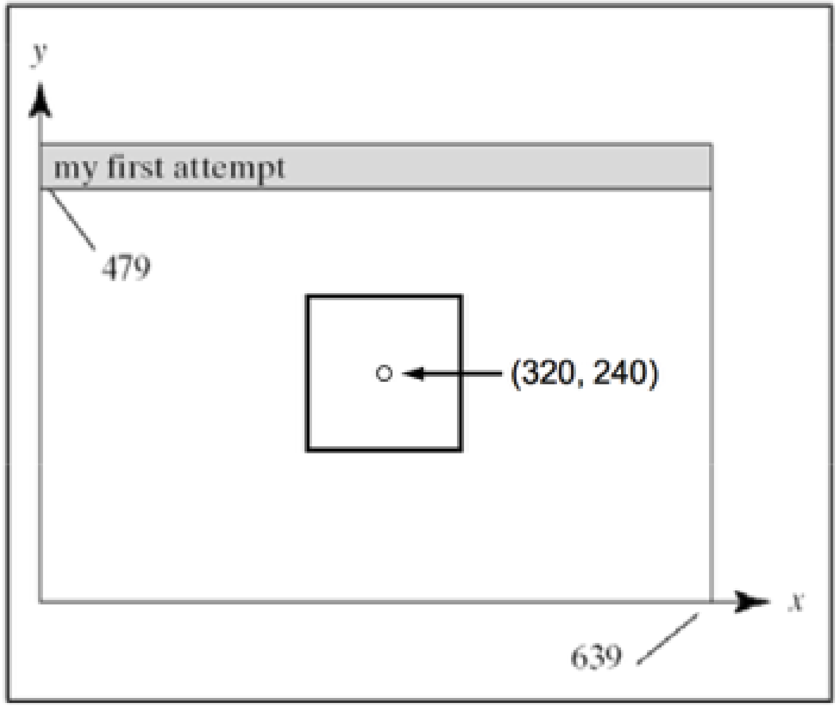
- `gluOrtho2D()` sets the units of the screen coordinate system
  - `gluOrtho2D(0, w, 0, h)` means the coordinates are in units of pixels
  - `gluOrtho2D(0, 1, 0, 1)` means the coordinates are in units of "fractions of window size" (regardless of actual window size)



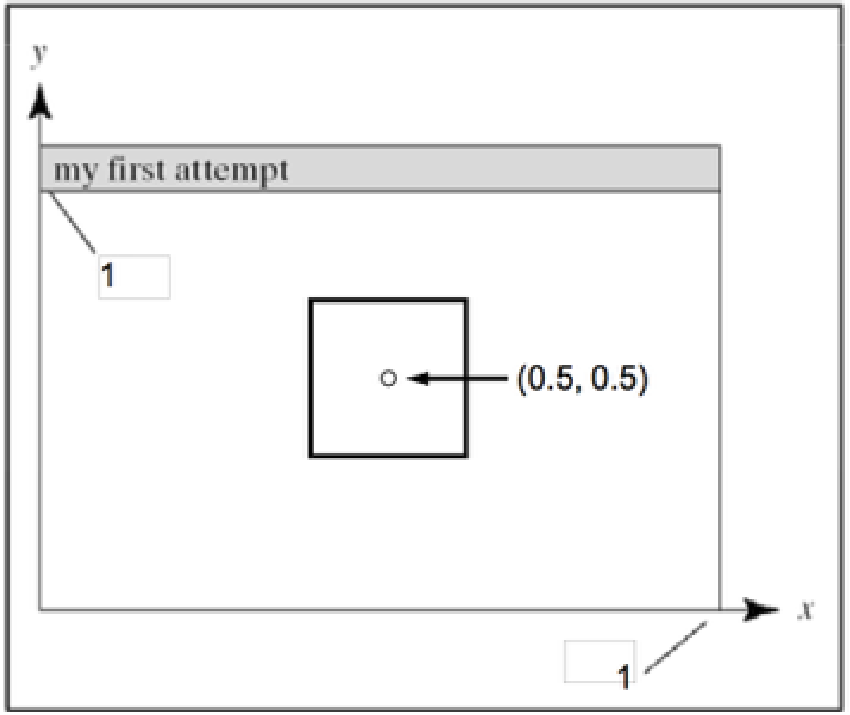
from Hill

Screen coordinates

`gluOrtho2D(0, 640, 0, 480)`

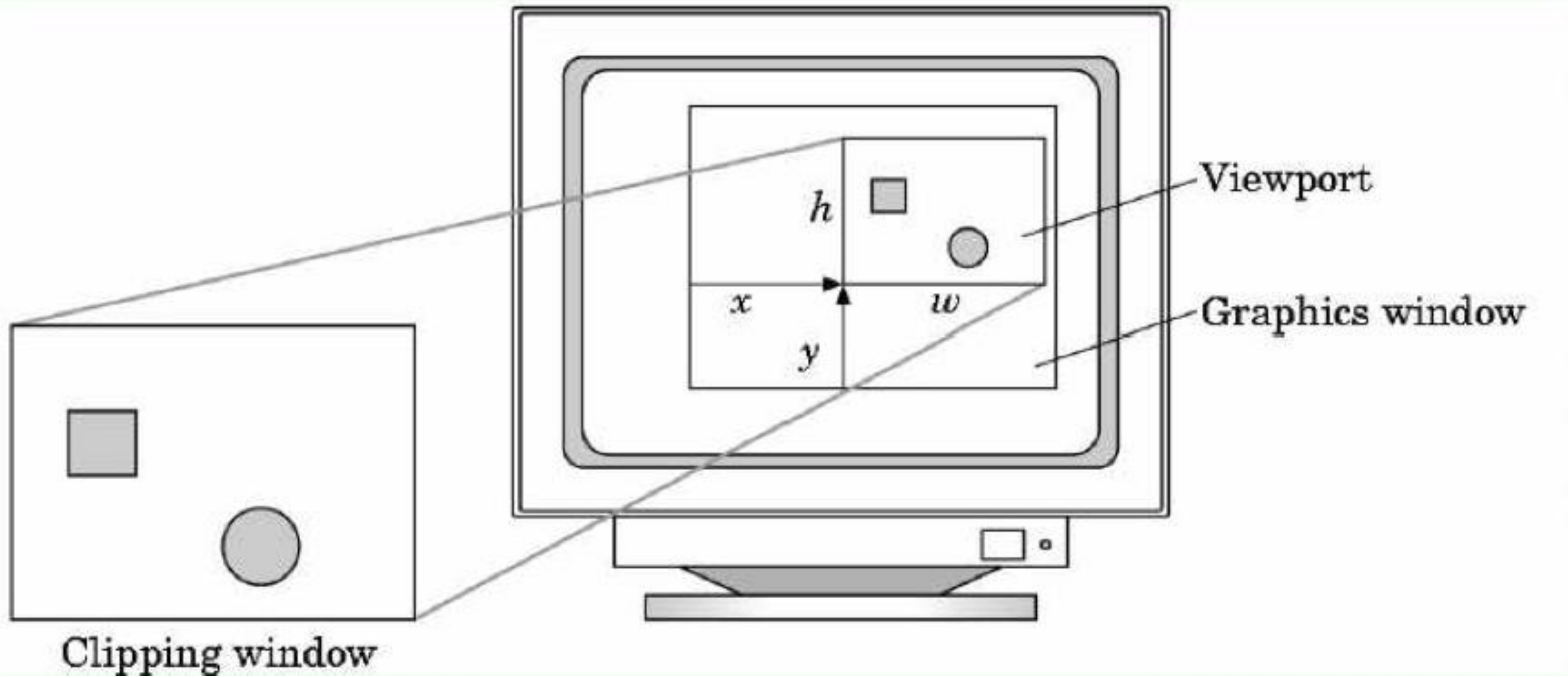


`gluOrtho2D(0, 1, 0, 1)`



Viewport

- Determines clipping in window coordinates
- glViewPort(x,y,w,h)



Screen Refresh & Double Buffering

- Screen Refresh: 60-100 Hz
- Flicker if drawing overlaps screen refresh
- Problem during animation
- Solution: use two separate frame buffers:
  - Draw into one buffer
  - Swap and display, while drawing into other buffer
- Desirable frame rate >= 30 fps (frames/sec)

Enabling Single/Double Buffering

- glutInitDisplayMode(GLUT_SINGLE);
- glutInitDisplayMode(GLUT_DOUBLE);

- Single buffering:
  - Must call glFlush() at the end of Display()

- Double buffering:
  - Must call glutSwapBuffers() at the end of Display()
  - Must call glutPostRedisplay() at the end of Idle()

- If something in OpenGL has no effect or does not work, check the modes in glutInitDisplayMode()

Let’s code a triangle!

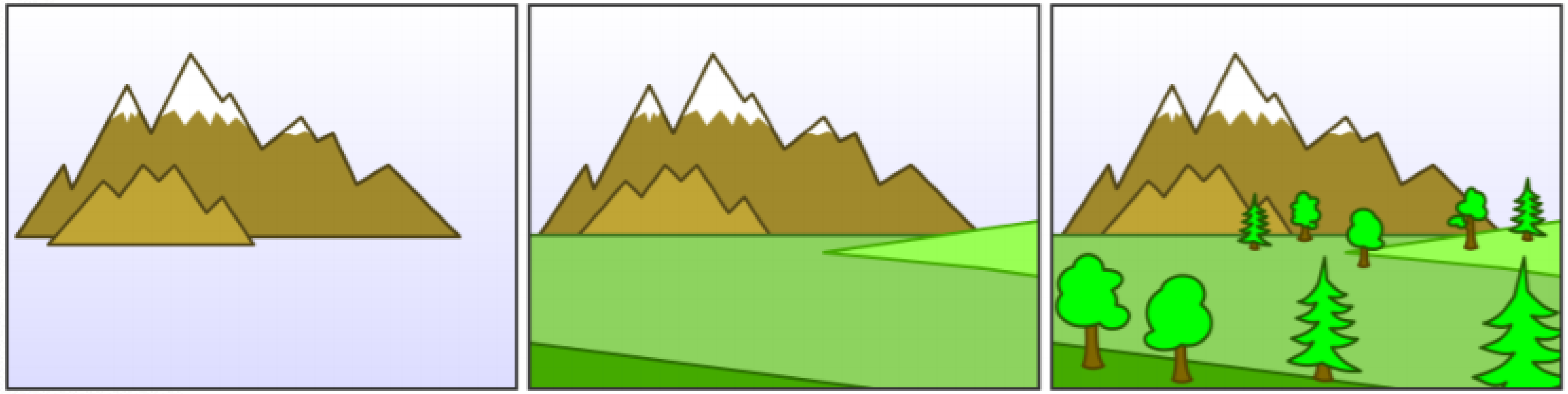
Hidden Surface Removal

- Classic problem of computer graphics

- what is visible after clipping and projection?

- Object-space vs image-space approaches
  - Object space: depth sort (Painter's algorithm)
  - Image space: z-buffer algorithm
- Related: back-face culling

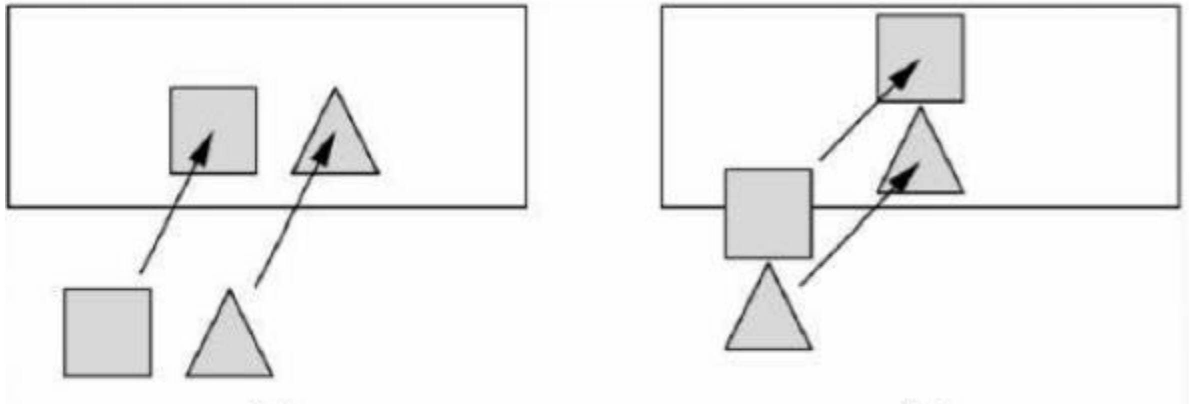
Object-Space Approach



- Painter’s algorithm: render back-to-front
- “Paint” over invisible polygons
- How to sort and how to test overlap?

Depth Sorting

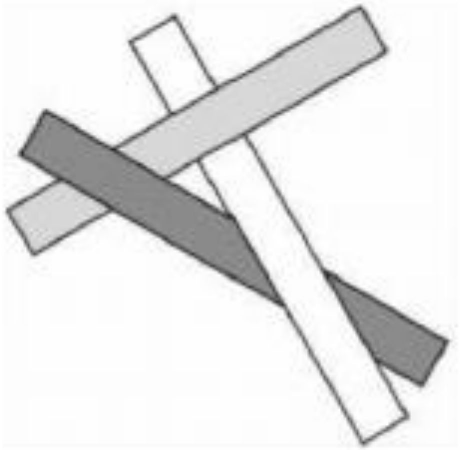
- First, sort by furthest distance z from viewer
- If minimum depth of A is greater than maximum depth of B, A can be drawn before B
- If either x or y extents do not overlap, A and B can be drawn independently



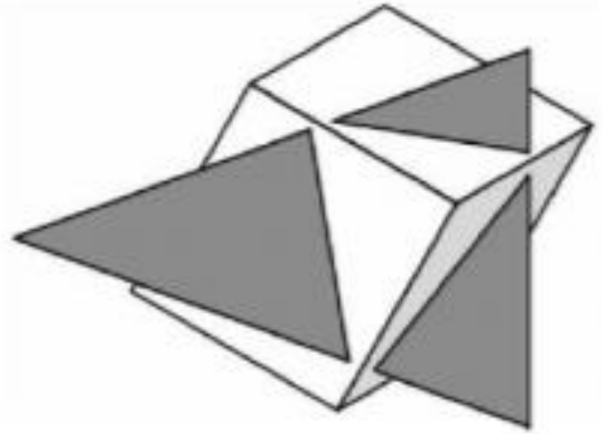
(a) (b)

Some Difficult Cases

- Sometimes cannot sort polygons



Cyclic overlap



Piercing Polygons

- One solution: compute intersections & subdivide
- Do while rasterizing (difficult in object space)

Painter’s Algorithm Assessment

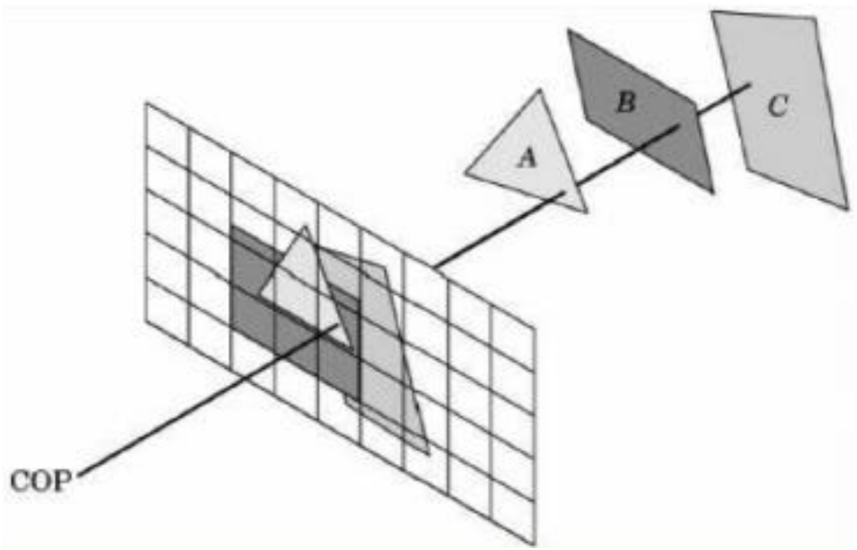
- Strengths
  - Simple (most of the time)
  - Handles transparency well
  - Sometimes, no need to sort (e.g., heightfield)

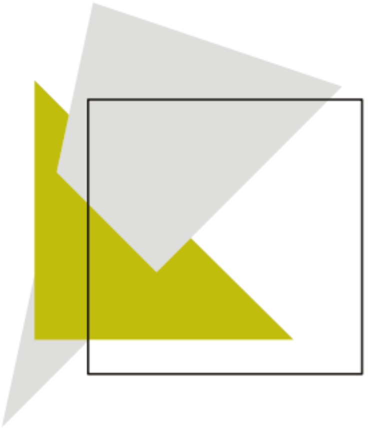
- Weaknesses
  - **Clumsy when geometry is complex**
  - **Sorting can be expensive**

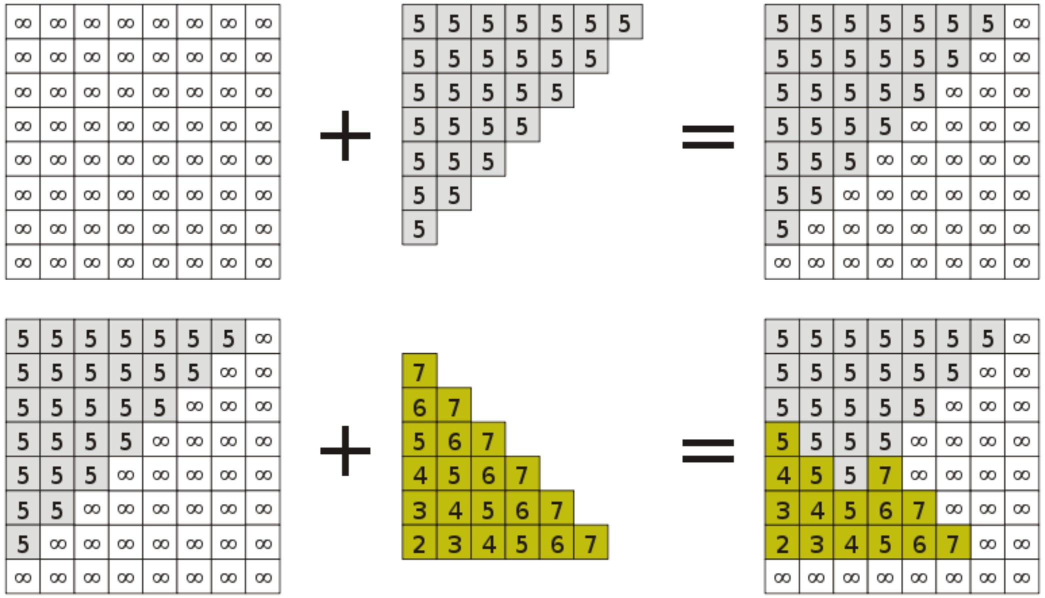
- Usage
  - PostScript interpreters
  - OpenGL: not supported (must implement Painter’s Algorithm manually)

Image-Space Approach

- Raycasting: intersect ray with polygons
- z-buffer stores depth values z for each pixel

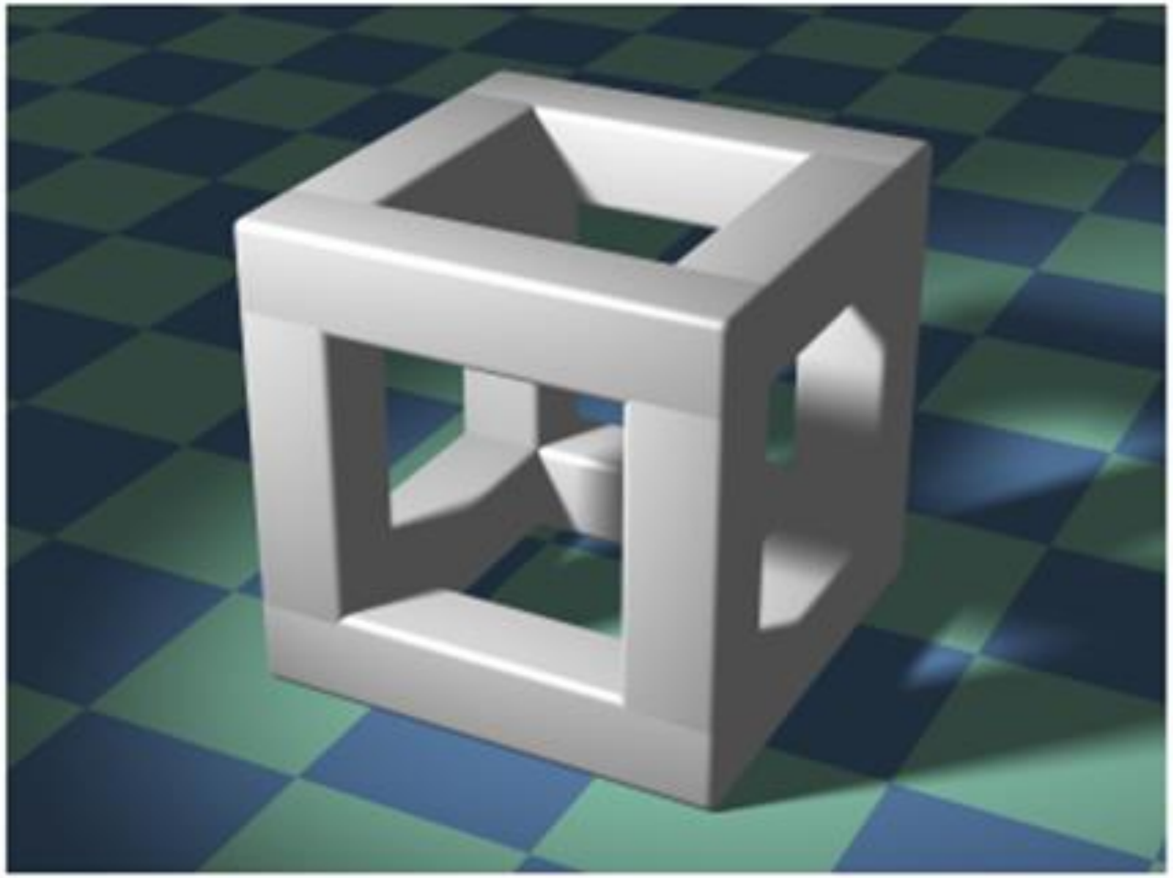




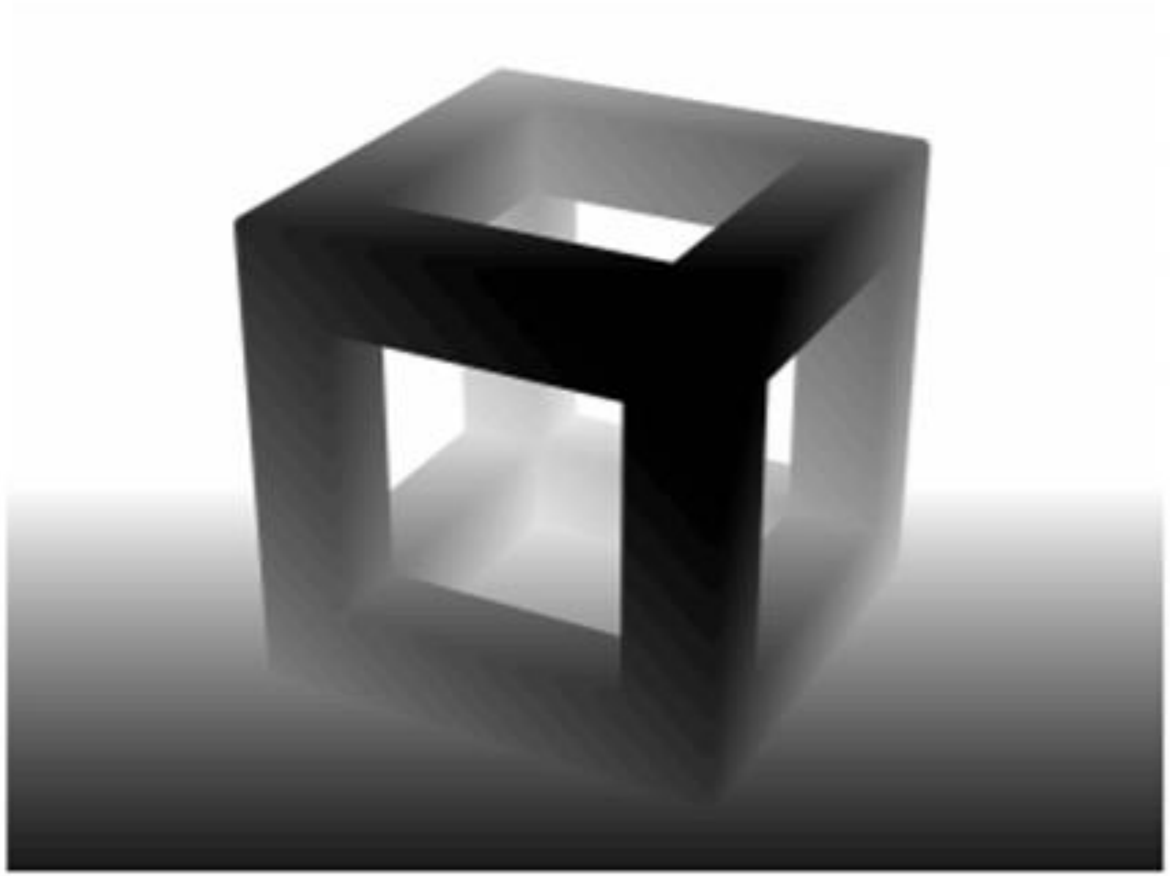


- O(k) worst case (often better)
- Images can be more jagged (need anti-aliasing)

Image-space approach



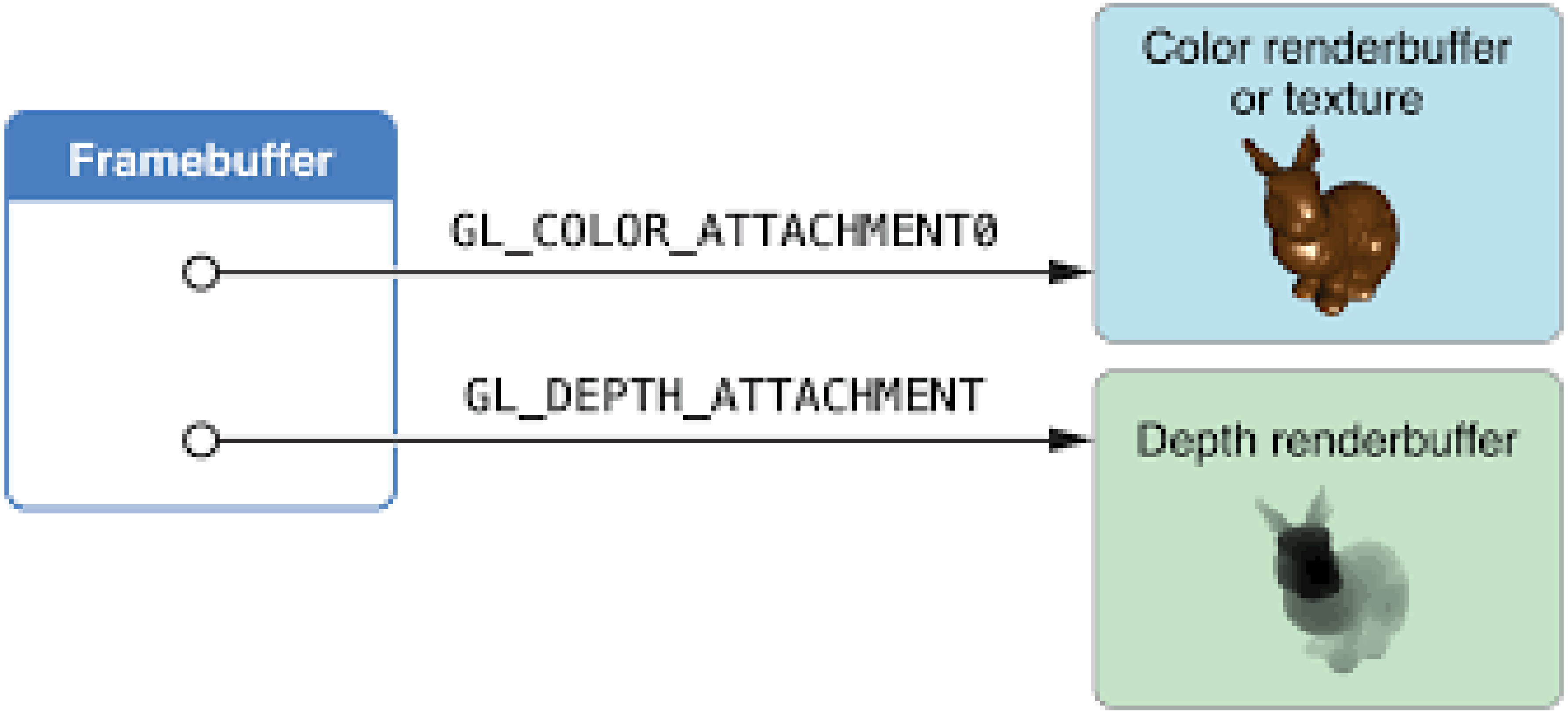
3d geometry



depth image/buffer (darker is closer)

How to render an image like the left figure?

glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH);



- glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
  - What does it mean?
  - Two color buffers + two depth buffers
  - Two color buffers + one depth buffer

glutInitDisplayMode

- glutInitDisplayMode(GLUT_SINGLE);
  - One color buffer with default type GLUT_RGBA

- glutInitDisplayMode(GLUT_DOUBLE);
  - Two color buffers, both with default type GLUT_RGBA

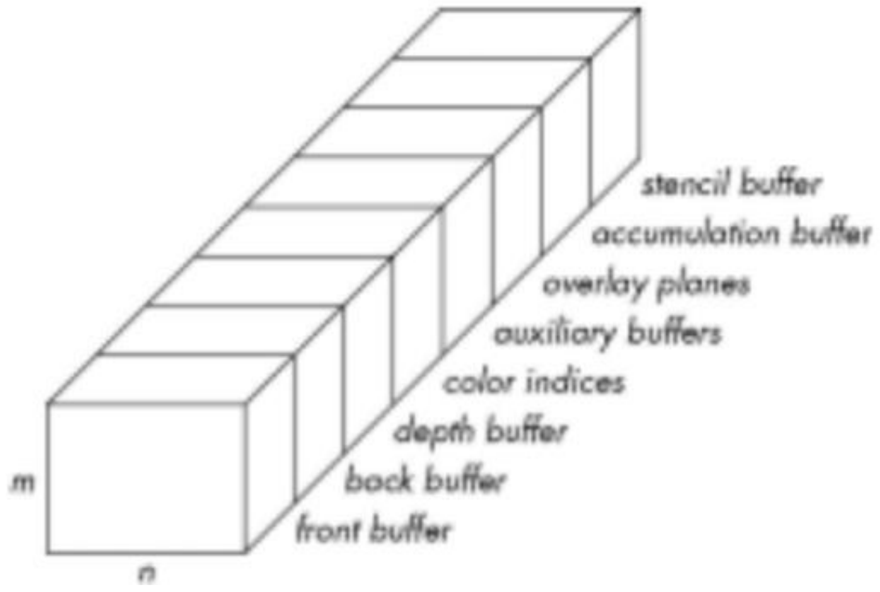
- glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
  - Two color buffers (both with type GLUT_RGBA) + one depth buffer(i.e. z-buffer)

- Does GL own any other buffers?

Framebuffer in OpenGL

- glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);

- GL has many buffers:
  - Color buffers
    - **front-left [default]**, front-right, back-left, back-right, and any number of auxiliary color buffers
    - left and right buffers are used for stereoscopic rendering
  - Depth buffer
  - Stencil buffer
  - Accumulation buffer



- Double buffer means?
  - two color buffers: front-left & back-left

The z-Buffer Algorithm Assessment

- Strengths
  - Simple (no sorting or splitting)
  - Independent of geometric primitives
- Weaknesses
  - Memory intensive 24 bit (but memory is cheap now)
  - Tricky to handle transparency and blending
  - Depth-ordering artifacts (numerical issues)
- Usage
  - z-Buffering comes standard with OpenGL;
    - + Object-Oriented Rendering strategy
  - **disabled by default; must be enabled**

Depth Buffer in OpenGL

- glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);

- glEnable (GL_DEPTH_TEST);

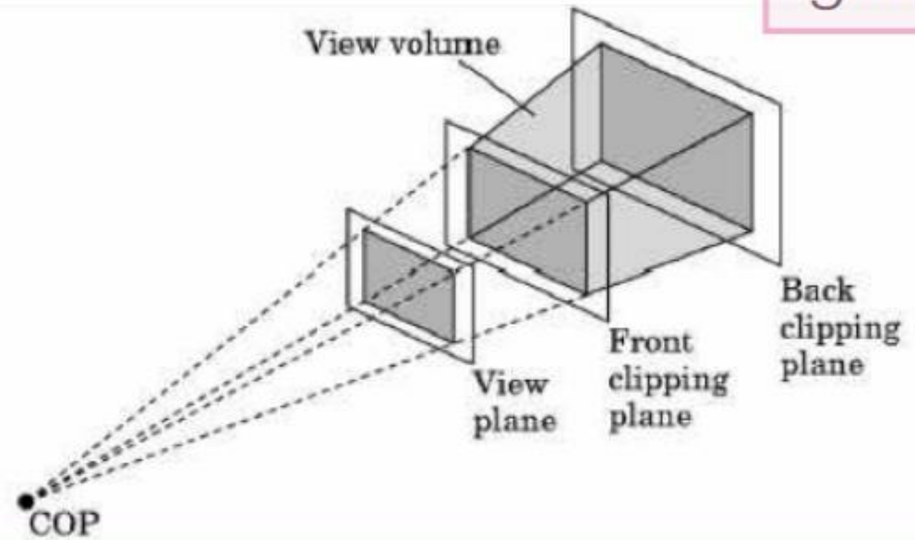
- Inside Display(): glClear (GL_DEPTH_BUFFER_BIT);

- Remember all of these!

Specifying the Viewing Volume

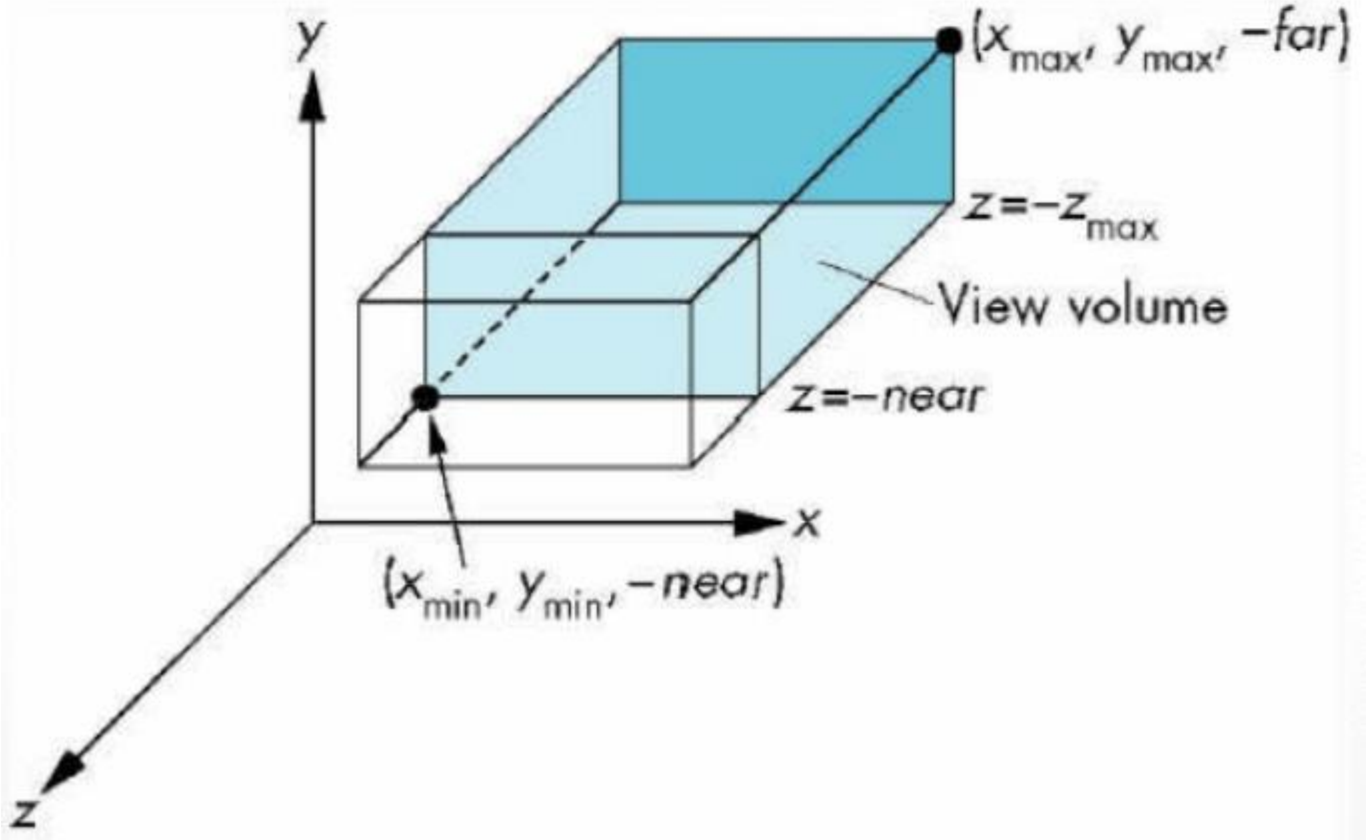
- Clip everything not in viewing volume
- Separate matrices for
  - GL_MODELVIEW: modeling (or viewing) transformation
  - GL_PROJECTION: projection transformation

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity();
... Set viewing volume ...
glMatrixMode(GL_MODELVIEW);
```



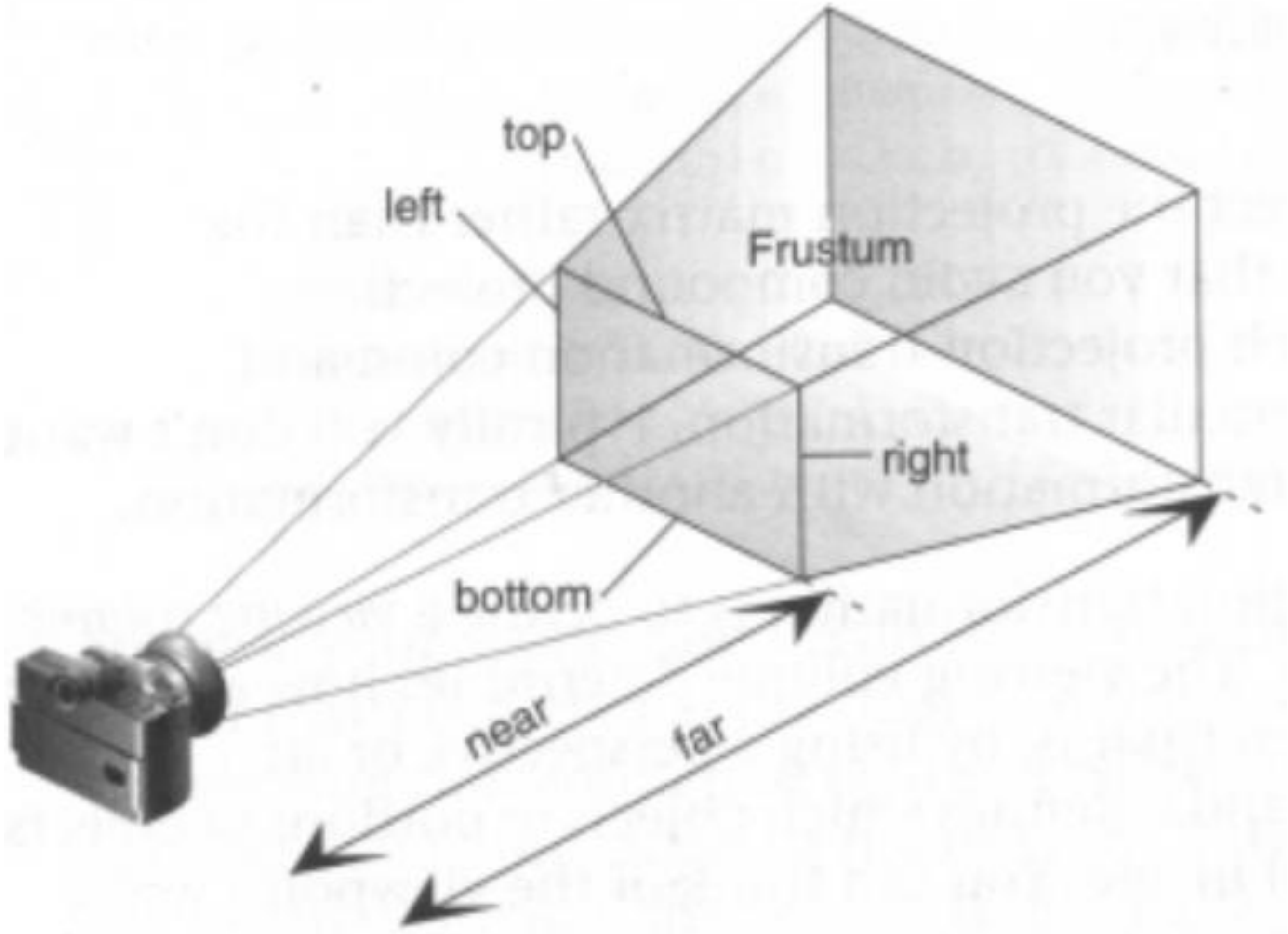
Parallel Viewing

- Orthographic projection
- Camera points in negative z direction
- glOrtho(xmin, xmax, ymin, ymax, near, far)



Perspective Viewing

- Slightly more complex
- glFrustum(left, right, bottom, top, near, far)



Simple Transformations

- Rotate by given angle (in degrees) about axis given by (x, y, z)
  - glRotate{fd}(angle, x, y, z);

- Translate by the given x, y, and z values
  - glTranslate{fd}(x, y, z);

- Scale with a factor in the x, y, and z direction
  - glScale{fd}(x, y, z);

Example: Rotating Color Cube

- Adapted from [Angel, Ch. 3]

- Problem
  - Draw a color cube
  - Rotate it about x, y, or z axis, depending on left, middle or right mouse click
  - Stop when space bar is pressed
  - Quit when q or Q is pressed

Step 1: Defining the Vertices

- Use parallel arrays for vertices and colors

```
/* vertices of cube about the origin */
GLfloat vertices[8][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}};

/* colors to be assigned to vertices */
GLfloat colors[8][3] =
    {{0.0, 0.0, 0.0}, {1.0, 0.0, 0.0},
    {1.0, 1.0, 0.0}, {0.0, 1.0, 0.0}, {0.0, 0.0, 1.0},
    {1.0, 0.0, 1.0}, {1.0, 1.0, 1.0}, {0.0, 1.0, 1.0}};
```

Step 2: Set Up z-buffer and Double Buffering

```
int main(int argc, char **argv)
  {
     glutInit(&argc, argv);
     /* double buffering for smooth animation */
     glutInitDisplayMode
        (GLUT_DOUBLE | GLUT_DEPTH | GLUT_RGB);
     ...    /* window creation and callbacks here */
     glEnable(GL_DEPTH_TEST);
     glutMainLoop();
     return(0);
  }
```

Step 3: Install Callbacks

- Create window and set callbacks

```
glutInitWindowSize(500, 500);
    glutCreateWindow("cube");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
    glutIdleFunc(spinCube);
    glutMouseFunc(mouse);
    glutKeyboardFunc(keyboard);
```

Step 4: Reshape Callback

- Set projection and viewport, preserve aspect ratio

```
void myReshape(int w, int h)
  {
    GLfloat aspect = (GLfloat) w / (GLfloat) h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h) /* aspect <= 1 */
        glOrtho(-2.0, 2.0, -2.0/aspect, 2.0/aspect, -10.0, 10.0);
    else /* aspect > 1 */
        glOrtho(-2.0*aspect, 2.0*aspect, -2.0, 2.0, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
  }
```

Step 5: Display Callback

- Clear, rotate, draw, flush, swap

```
GLfloat theta[3] = {0.0, 0.0, 0.0};

  void display(void)
  {
    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);
    colorcube();
    glutSwapBuffers();
  }
```

Step 6: Drawing Faces

- Call face(a, b, c, d) with vertex index
- Orient consistently

```
void colorcube(void)
  {
      face(0,3,2,1);
      face(2,3,7,6);
      face(0,4,7,3);
      face(1,2,6,5);
      face(4,5,6,7);
      face(0,1,5,4);
  }
```

Step 7: Drawing a Face

- Use vector form of primitives and attributes

```
void face(int a, int b, int c, int d)
  {
     glBegin(GL_POLYGON);
        glColor3fv(colors[a]);
        glVertex3fv(vertices[a]);
        glColor3fv(colors[b]);
        glVertex3fv(vertices[b]);
        glColor3fv(colors[c]);
        glVertex3fv(vertices[c]);
        glColor3fv(colors[d]);
        glVertex3fv(vertices[d]);
     glEnd();
  }
```

Step 8: Animation

- Set idle callback

```
GLfloat delta = 2.0;
   GLint axis = 2;
   void spinCube()
   {
      /* spin the cube delta degrees about selected axis */
      theta[axis] += delta;
      if (theta[axis] > 360.0) theta[axis] -= 360.0;




      /* display result (do not forget this!) */
      glutPostRedisplay();
   }
```

Step 9: Change Axis of Rotation

- Mouse callback

```
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;
}
```

Step 10: Toggle Rotation or Exit

- Keyboard callback

```
void keyboard(unsigned char key, int x, int y)
   {
      if (key=='q' || key == 'Q')
         exit(0);
      if (key==' ')
       stop = !stop;
      if (stop)
         glutIdleFunc(NULL);
      else
         glutIdleFunc(spinCube);
   }
```

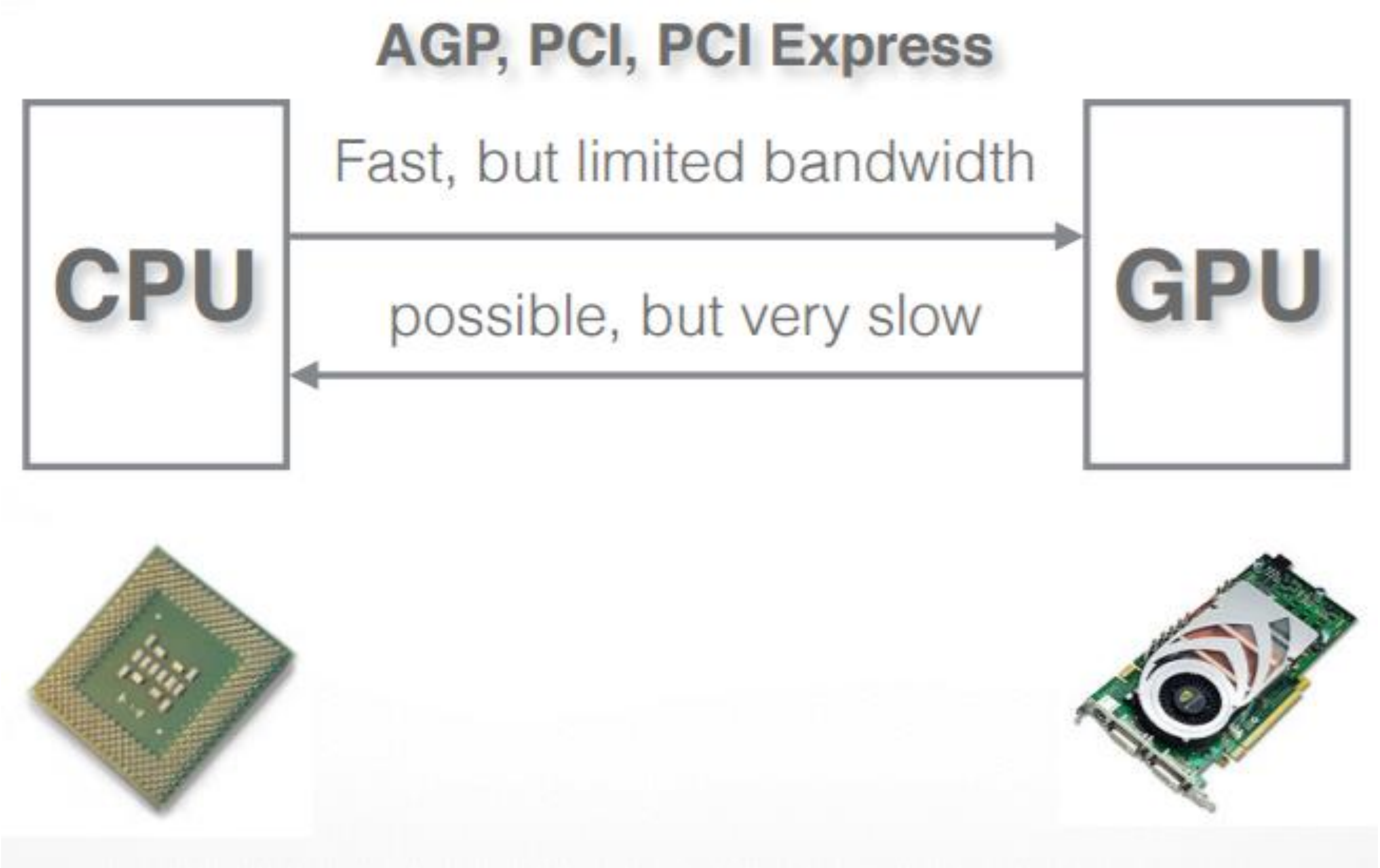
We need performance!

Client/Server Model

- Graphics hardware and caching

- Important for efficiency
- Need to be aware where data are stored
- Examples: vertex arrays, display lists

The CPU-GPU bus



How to send vertex data to graphics card

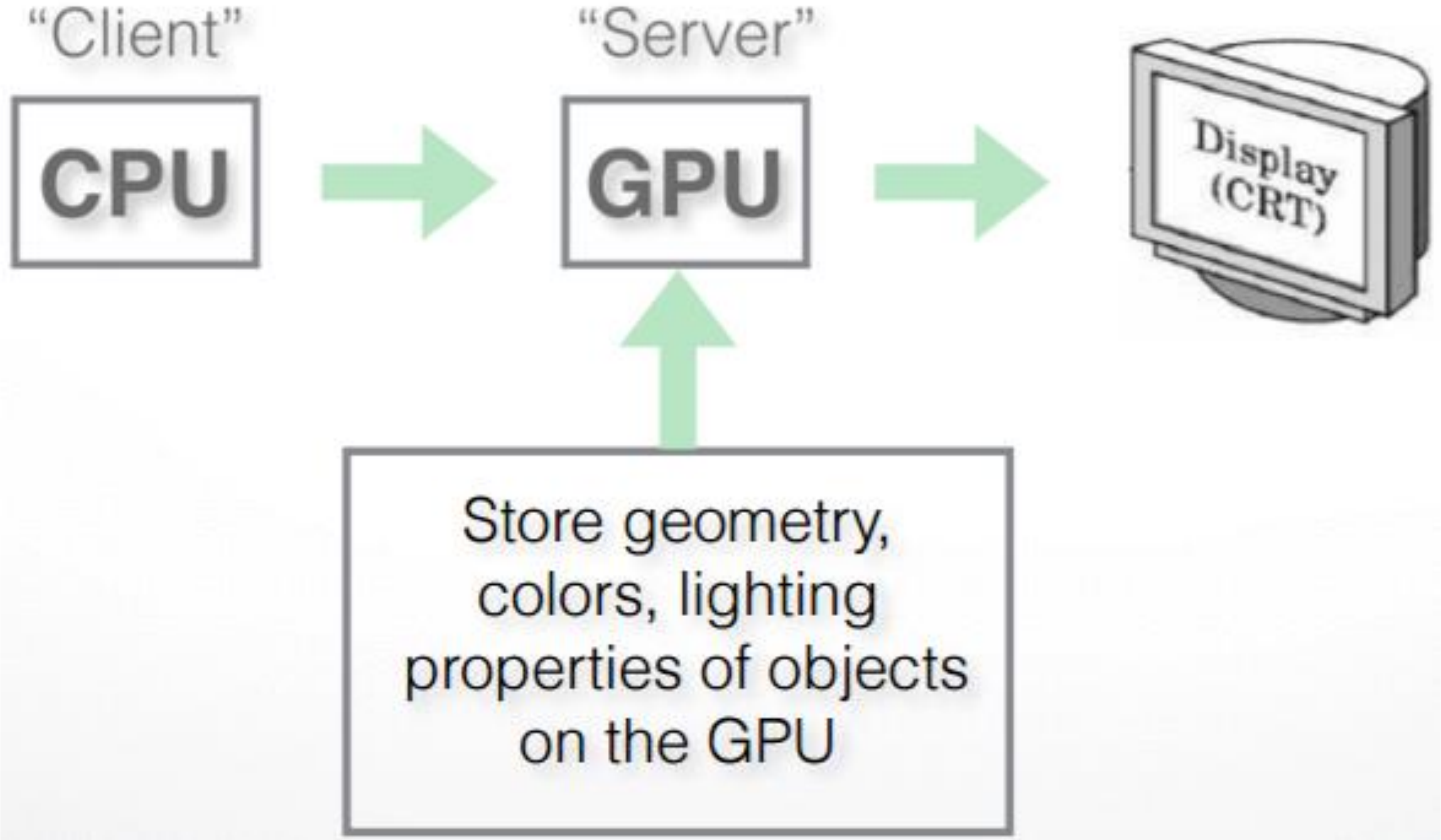
- Different ways:
  - Immediate mode (glBegin / glVertex / glEnd etc.)
  - Display Lists
  - Vertex Arrays
  - **Vertex Buffer Objects (VBO) etc.**
- Immediate Mode is quite inefficient, so it’s been dropped in OpenGL ES and depreciated in OpenGL 3.0.
- Vertex Arrays or VBO are way more efficient, but generally not as straightforward to setup and use.

Immediate mode (glBegin / glVertex / glEnd etc.)

Deprecated!

Display Lists

- Cache a sequence of drawing commands
- Optimize and store on server (GPU)



Display Lists

- Cache a sequence of drawing commands
- Optimize and store on server (GPU)

```
GLuint listName = glGenLists(1);    /* new list name */
    glNewList (listName, GL_COMPILE); /* new list */
        glColor3f(1.0, 0.0, 1.0);
        glBegin(GL_TRIANGLES);
            glVertex3f(0.0, 0.0, 0.0);
            ...
        glEnd();
    glEndList(); /* at this point, OpenGL compiles the list */
    glCallList(listName);     /* draw the object */
```

Display Lists Details

- Very useful with complex objects that are redrawn often (e.g., with transformations)

- Display lists can call other display lists
- **Display lists cannot be changed**
- Display lists can be erased / replaced

- Not necessary in first assignment
- Display lists are now **deprecated** in OpenGL
- For complex usage, use the VertexBufferObject(VBO) extension

Vertex Arrays

- Draw cube with 6*4=24 or with 8 vertices?
- Expense in drawing and transformation
- Strips help to some extent
- Vertex arrays provide general solution

- Advanced (since OpenGL 1.2)
  - Define (transmit) array of vertices, colors, normals
  - Draw using index into array(s)
  - Vertex sharing for efficient operations
- Not needed for first assignment

```
glBegin(GL_TRIANGLES);  // draw a cube with 12 triangles

    // front face =================
    glVertex3fv(v0);    // v0-v1-v2
    glVertex3fv(v1);
    glVertex3fv(v2);

    glVertex3fv(v2);    // v2-v3-v0
    glVertex3fv(v3);
    glVertex3fv(v0);

    // right face =================
    glVertex3fv(v0);    // v0-v3-v4
    glVertex3fv(v3);
    glVertex3fv(v4);

    glVertex3fv(v4);    // v4-v5-v0
    glVertex3fv(v5);
    glVertex3fv(v0);

    // top face ===================
    glVertex3fv(v0);    // v0-v5-v6
    glVertex3fv(v5);
    glVertex3fv(v6);

    glVertex3fv(v6);    // v6-v1-v0
    glVertex3fv(v1);
    glVertex3fv(v0);

    ...           // draw other 3 faces

glEnd();
```

```
GLfloat vertices[] = {...};          // 8 of vertex coords
GLubyte indices[] = {0,1,2, 2,3,0,   // 36 of indices
                     0,3,4, 4,5,0,
                     0,5,6, 6,1,0,
                     1,6,7, 7,2,1,
                     7,4,3, 3,2,7,
                     4,7,6, 6,5,4};
...
// activate and specify pointer to vertex array
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);

// draw a cube
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_BYTE, indices);

// deactivate vertex arrays after drawing
glDisableClientState(GL_VERTEX_ARRAY);
```

More on http://www.songho.ca/opengl/gl_vertexarray.html

Vertex Buffer Objects (VBOs)

- Display Lists: Fast / inflexible
- Immediate mode: Slowest / flexible
- Vertex Array: Slow with shared vertices / flexible

- OpenGL 3.x & up: use Vertex Buffer Objects (VBO)
  - Much more efficient since it allows the geometry to be stored in the graphics card and reduce the number of function calls
  - Best of between Display List and Vertex Array: Fast / flexible

```
glGenBuffers(1, &vboHandle); // create a VBO handle
glBindBuffer(GL_ARRAY_BUFFER, vboHandle); // bind the handle to the current VBO
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW); // allocate space and copy the data over
```

GLEW: The OpenGL Extension Wrangler Library

- A cross-platform open-source C/C++ extension loading library loading library (windows, OS X, Linux, FreeBSD)
- Why GLEW is needed?
- It is not possible to link directly to functions that are provided in newer version of OpenGL. In windows, this means OpenGL 1.2 and up
- GLEW does the tedious work to helps you find the function pointers (addresses of the functions) for OpenGL extensions
- GLEW can also help you to check if a particular OpenGL extension is available on your machine

GLEW usage

- Make sure you have GLEW on your machine
- Include the glew header file
  - #include <GL/glew.h>
- Initialize glew before calling any opengl functions
  - GLenum err = glewInit();
  - if (err != GLEW_OK) printf(" Error initializing GLEW! \n");
  - else printf(else printf("Initializing GLEW succeeded!\n);
- Check OpenGL features, for example, shaders
  - if (! GLEW_ARB_vertex_program)
    - printf(" ARB vertex program is not supported!!\n");
  - else printf(" ARB vertex program is supported!!\n");

Vertex Buffer Objects (VBO)
more efficient, but not straightforward

Vertex Buffer Object (VBO)

- Motivation
  - Replacing the out-dated functions such as glBegin(), glEnd(), glVertex*(), glNormal*(), glTexCoord*, glColor*, etc to define the geometry
  - Provide per-vertex input to the GPU
  - Allowing significant increases in vertex throughput between CPU and GPU

  - A mechanism to provide generic vertex attributes to the shader, and store vertex data in video RAM
  - The programmer is free to define an arbitrary set of pervertex attributes to the vertex shader

Creating a VBO

- Step 1: Generate a new buffer object with **glGenBuffers()**
  - Create buffer objects and returns the identifiers of the buffer objects
  - void glGenBuffers(Glsizei n, Gluint* ids);
- Bind the buffer object with **glBindBuffer()**
  - Specify the target (i.e., what kind of buffer) to which the buffer object is
  - bound
  - target: GL_ARRAY_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER, GL_PIXEL_UNPACK_BUFFER
  - GL_ARRAY_BUFFER is to provide the vertex attributes, and GL_ELEMENT_ARRAY_BUFFER is to provide the triangle indices
- Copy the vertex data to the buffer object
  - **glBufferData(**Glenum target, Glsizei size, const void* data, Glenum usage)
  - Usage is the access pattern: STATIC_, STREAM_, DYNAMIC_{DRAW, COPY, READ}

Example

```
typedef struct{
float location[4];
float color[4];
} Vertex;

Vertex verts[6]; // triangle vertices
GLubyte tindices[6]; // triangle vertex indices
GLuint vboHandle[1]; // a VBO that contains interleaved positions and colors
GLuint indexVBO;
```

Example (cont'd)

```
void InitGeometry()
{
verts[0].location[0] = -0.5; verts[0].location[1] = -0.5; verts[0].location[2] = 0; verts[0].location[3] = 1;
verts[1].location[0] = -0.5; verts[1].location[1] = 0.5; verts[1].location[2] = 0; verts[1].location[3] = 1;
verts[2].location[0] = 0.5; verts[2].location[1] = 0.5; verts[2].location[2] = 0; verts[2].location[3] = 1;
verts[3].location[0] = 0.5; verts[3].location[1] = 0.5; verts[3].location[2] = 0; verts[3].location[3] = 1;
verts[4].location[0] = 0.5; verts[4].location[1] = -0.5; verts[4].location[2] = 0; verts[4].location[3] = 1;
verts[5].location[0] = -0.5; verts[5].location[1] = -0.5; verts[5].location[2] = 0; verts[5].location[3] = 1;

verts[0].color[0] = 1; verts[0].color[1] = 1; verts[0].color[2] = 0; verts[0].color[3] = 1;
verts[1].color[0] = 1; verts[1].color[1] = 1; verts[1].color[2] = 0; verts[1].color[3] = 1;
verts[2].color[0] = 1; verts[2].color[1] = 1; verts[2].color[2] = 0; verts[2].color[3] = 1;
verts[3].color[0] = 1; verts[3].color[1] = 0; verts[3].color[2] = 0; verts[3].color[3] = 1;
verts[4].color[0] = 1; verts[4].color[1] = 0; verts[4].color[2] = 0; verts[4].color[3] = 1;
verts[5].color[0] = 1; verts[5].color[1] = 0; verts[5].color[2] = 0; verts[5].color[3] = 1;

// create triangle vertex indices.
tindices[0] = 0; tindices[1] = 1; tindices[2] = 2;
tindices[3] = 3; tindices[4] = 4; tindices[5] = 5;
}
```

Example (cont'd)

```
void InitVBO(){
glGenBuffers(1, vboHandle); // create VBO handle for position & color
glBindBuffer(GL_ARRAY_BUFFER, vboHandle[0]); // bind the handle
glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex)*6, verts, GL_STATIC_DRAW); // allocate space and copy the position data over
glBindBuffer(GL_ARRAY_BUFFER, 0); // clean up


glGenBuffers(1, &indexVBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexVBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLubyte)*6, tindices,
GL_STATIC_DRAW); // load the index data
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,0); // clean up
// by now, we moved the position and color data over to the graphics card. There will be no redundant datacopy at drawing time
}
```

Draw VBOs

- Bind (like activate) the VBOs
  - The **vertex (attributes)** and **element (indices)** arrays for example
- capabilities to handle/use vertex attribute arrays on the client (CPU) side
  - By default, all client-side capabilities are disabled.
  - http://www.opengl.org/sdk/docs/man/xhtml/glEnableClientState.xml
- Specify the starting positions and strides of the vertex attributes in the VBO
  - glColorPointer(4, GL_FLOAT, sizeof(Vertex), (char*) NULL+ 16);
  - glVertexPointer(4,GL_FLOAT, sizeof(Vertex), (char*) NULL+ 0);
- Draw the geometry
  - glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, (char*) NULL+0);
- Clean up
  - glDisableClientState(GL_VERTEX_ARRAY);
  - glDisableClientState(GL_COLOR_ARRAY);

Example (cont'd)

```
void display()
{
glClearColor(0,0,1,1); glClear(GL_COLOR_BUFFER_BIT);

glBindBuffer(GL_ARRAY_BUFFER, vboHandle[0]);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexVBO);

glEnableClientState(GL_VERTEX_ARRAY); // enable the vertex array on the client side
glEnableClientState(GL_COLOR_ARRAY); // enable the color array on the client side

glColorPointer(4, GL_FLOAT, sizeof(Vertex), (char*) NULL+ 16);
glVertexPointer(4,GL_FLOAT, sizeof(Vertex), (char*) NULL+ 0);

glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, (char*) NULL+0);

glDisableClientState(GL_VERTEX_ARRAY); glDisableClientState(GL_COLOR_ARRAY);
glutSwapBuffers();
}
```

From Now On

- From now on, let’s not use the old OpenGL methods to specify vertex
- attributes!!
- That is, no more glBegin()/glEnd() whenever possible please!!

glGenBuffers v.s. glGenBuffersARB

根据OpenGL所支持VBO的情况，有三种方式执行渲染

1. 支持OpenGL 1.5，使用标准的VBO函数
   - glGenBuffers
2. 不支持OpenGL 1.5，但以ARB扩展的形式支持VBO
   - glGenBuffersARB
3. 不支持VBO，使用Vertex Array代替

1. glGenBuffers() is a core OpenGL function in OpenGL 1.5 and later; glGenBuffersARB() was an extension implementing the same functionality in earlier versions.

2. Unless you're developing for an ancient system, there's no longer any reason to use the ARB extension.

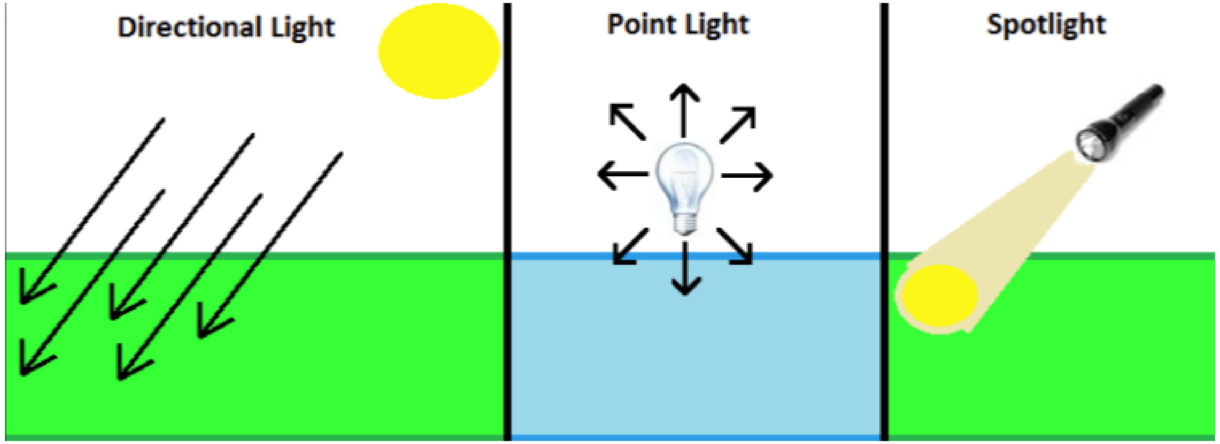
OpenGL Lighting

- Lighting calculations must be enabled and each light source must be enabled individually
  - glEnable(GL_LIGHTING);
  - glEnable(LIGHT1);

- Enabling lighting makes OpenGL to do the shading calculations

- Once **lighting is enabled**, colours assigned by **glColor() are no longer valid**

OpenGL Lighting: light source types



- GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
- glLightfv(GL_LIGHT0, GL_POSITION, light_position);
- w=0 => **directional light**, x,y,z describe its direction

- glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0); // **spotlight**
- GLfloat spot_direction[] = { -1.0, -1.0, 0.0 };
- glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);

OpenGL Lighting: light + material

- Each source has separate diffuse, specular and ambient RGB parameters
  - GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
  - …
  - glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
  - glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
  - glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
- Materials are modeled in a **complementary** manner
  - For each surface separate **ambient, diffuse and specular** components must be used
  - glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
  - glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
  - glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
- light components (LR, LG, LB)
- material components (MR, MG, MB),

OpenGL Lighting: light + material

- Each source has separate diffuse, specular and ambient RGB parameters
  - glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient); …
- Materials are modeled in a **complementary** manner
  - For each surface separate **ambient, diffuse and specular** components must be used
  - glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse); …

- light components (LR, LG, LB) + material components (MR, MG, MB) => (LR*MR, LG*MG, LB*MB).

Specifying Material Properties

- Material properties match the lighting properties
  - A material has reflectivity properties for each type of light

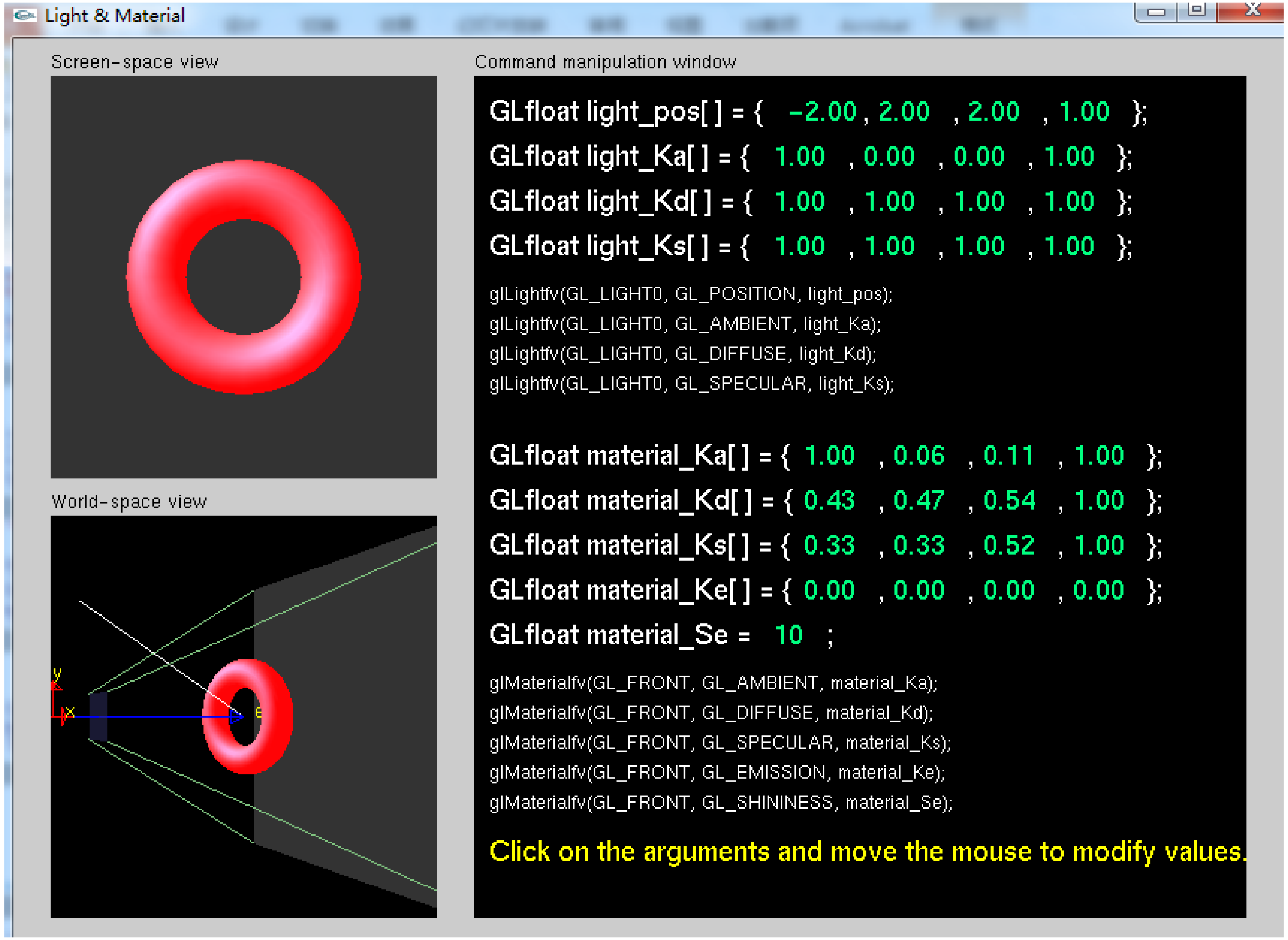
- The basic function for setting material properties is:
  - void **glMaterial**(GLenum face, GLenum name, TYPE value);

- **Diffuse** and **Ambient** Reflection
  - The GL_DIFFUSE and GL_AMBIENT parameters set with glMaterial*() affect the colour of the diffuse and ambient light reflected by an object

- **Specular** Reflection
  - Specular reflection from an object produces highlights.
  - OpenGL allows you to set the effect that the material has on reflected light (with GL_SPECULAR) and control the size and brightness of the highlight (with GL_SHININESS

- **Emission**
  - By specifying an RGBA color for GL_EMISSION, you can make an object appear to be giving off light of that color

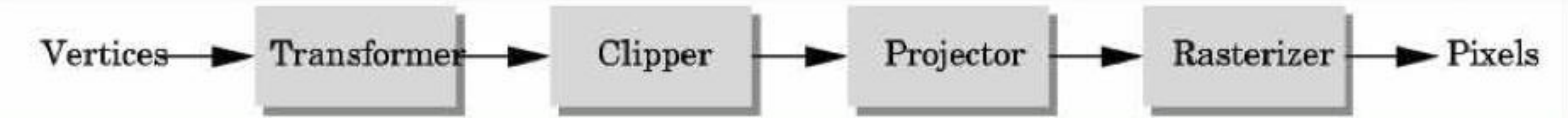
Light & Material



Nate_Robins_tutorials: lightmaterial

Summary

- **A Graphics Pipeline**
- The OpenGL **API**
- **Primitives**: vertices, lines, polygons
- **Attributes**: color
- Example: drawing a **shaded triangle**



Suggestions

- Most people do old OGL because they found an out of date tutorial online.
- Modern OpenGL (Shaders & VBOs [Vertex Buffer Objects])