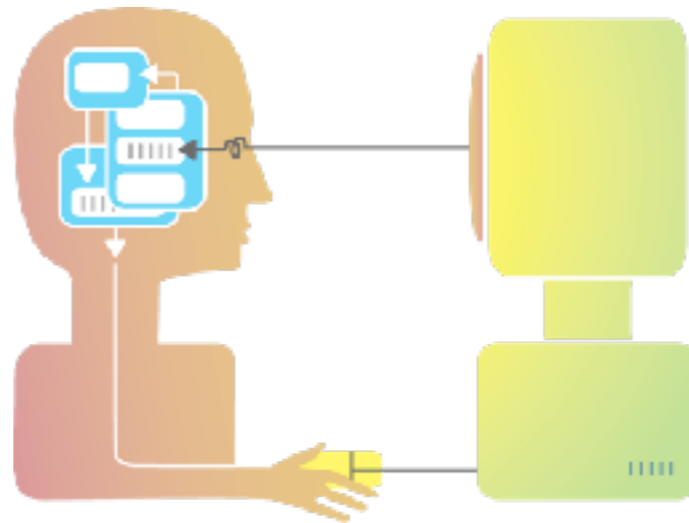


Fall 2018

CSCI 420: **Computer Graphics**



2.1 Input and Interaction

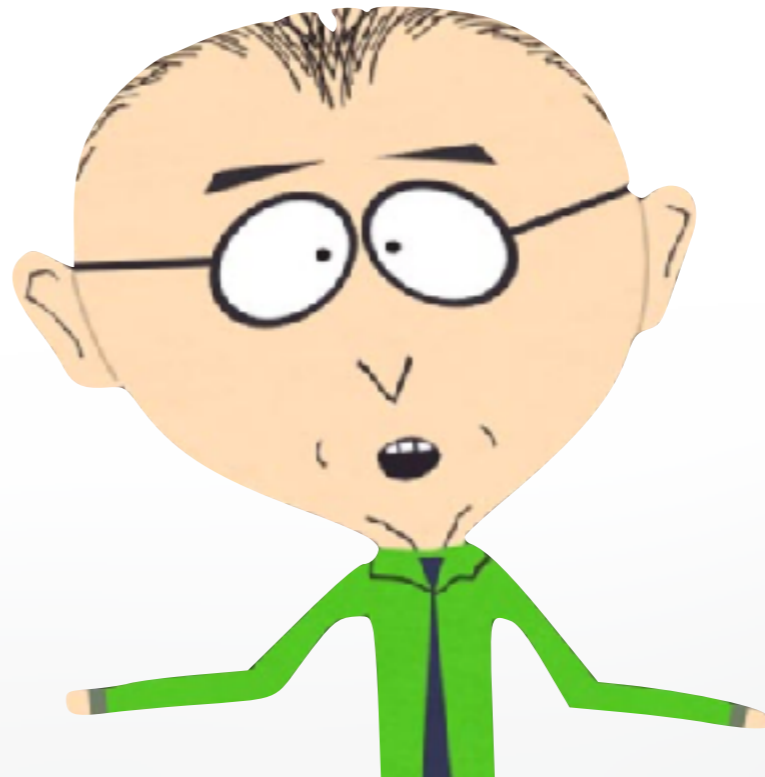


Hao Li

<http://cs420.hao-li.com>

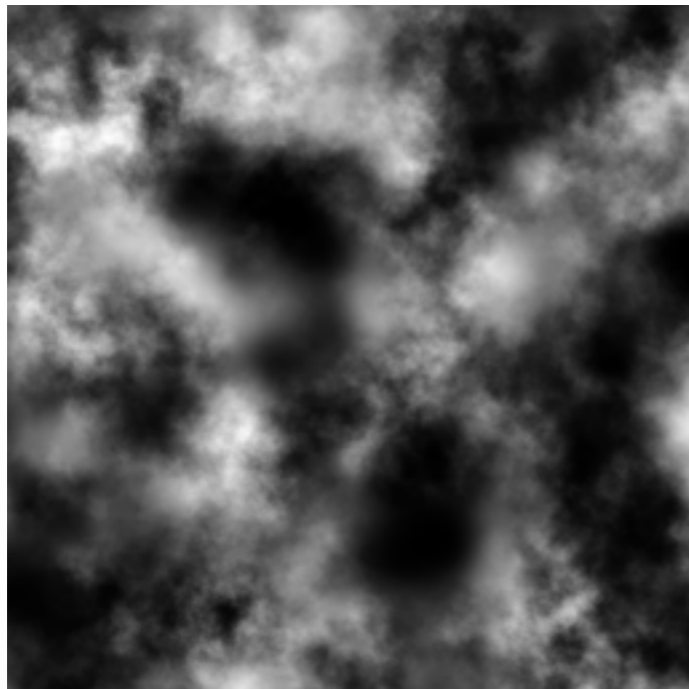
Administrative

- **Exercise 1:** It's out today!
- **Exercise 1 handout:** 11:59 PM, Monday, Sep 24
- **Hao Li (Me)**
 - Office Hour: Tue 2:00 PM - 3:00 PM, SAL 244
 - starting next week

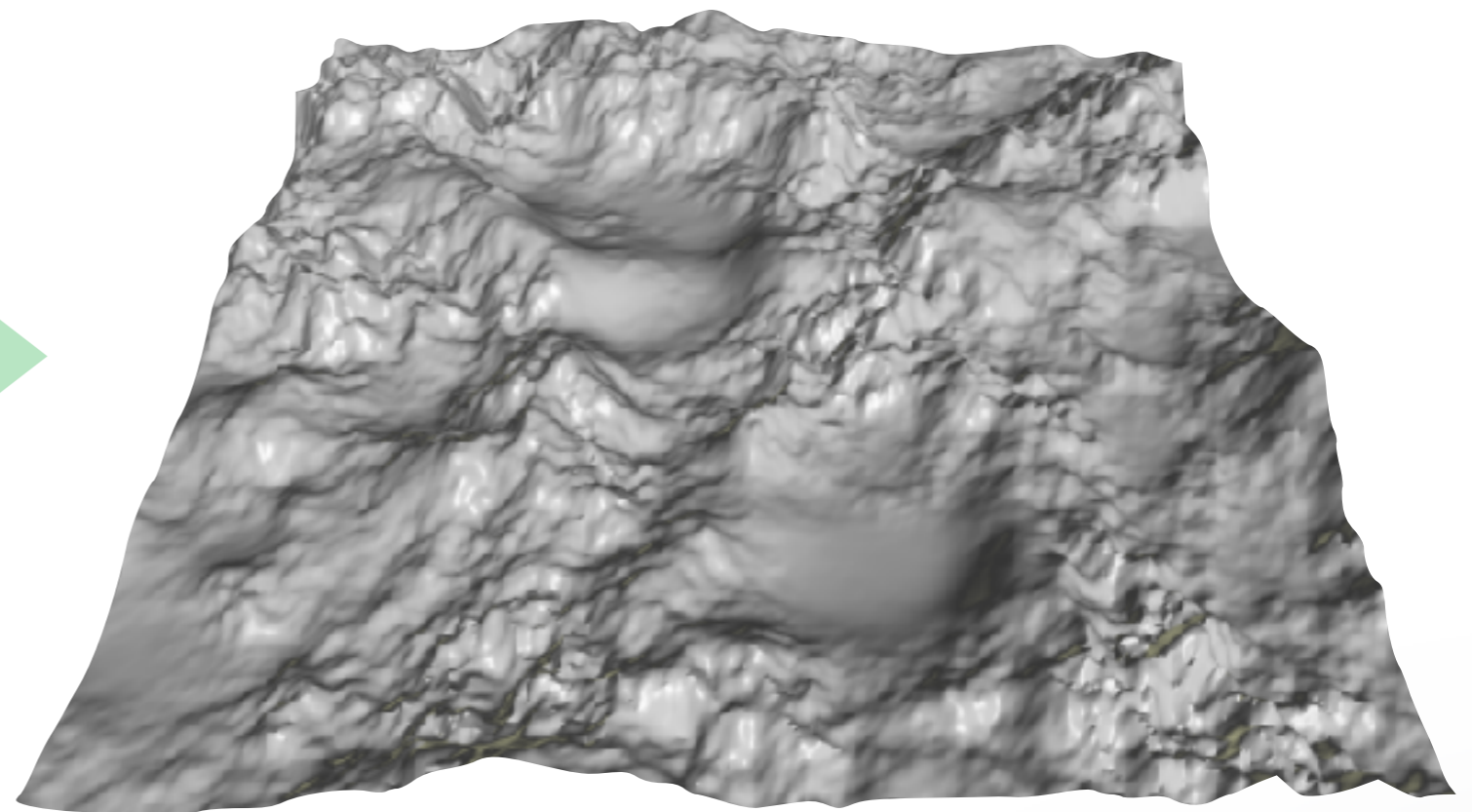
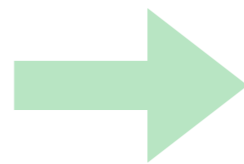


Exercise 1

Interactive 3D Heightfield Viewer and Fly-through!



input



output

Piazza



New Post

Search or add a post...

PINNED



Private Search for Teammates!

1



LAST WEEK

Private Introduce Piazza to your stu...

1



Private Get familiar with Piazza

1



Private Tips & Tricks for a successf...

1



Welcome to Piazza!

1



Read tips and tricks for a successful Piazza Class

Private Tips & Tricks for a successful class

read now

Enroll your students

Paste email addresses below in any format. Or visit [Manage Class](#) page to upload your student roster or share your Class Signup Link.

↳ Each will receive a welcome email.

jchr@email.com, smith@email.com

Enroll Students

Student Enrollment

..out of 50 (estimated) [Edit](#)

48 enrolled

Are there TAs/other instructors in your course?


yes

no

Exercise 1

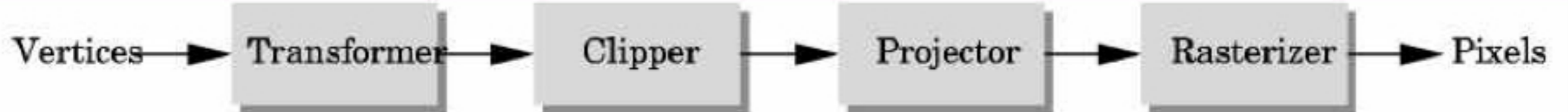
Where to find it?

Schedule

	Date	Topic		Material	Notes
0	<u>Tue</u> Aug 21, 2018	Introduction Basic Graphics Programming		[slides] [slides] [example]	
1	<u>Tue</u> Aug 28, 2018	Input and Interaction Transformations			[Ex. 1 Out]
2	<u>Tue</u> Sep 04, 2018	Viewing and Projection Hierarchical Modeling			

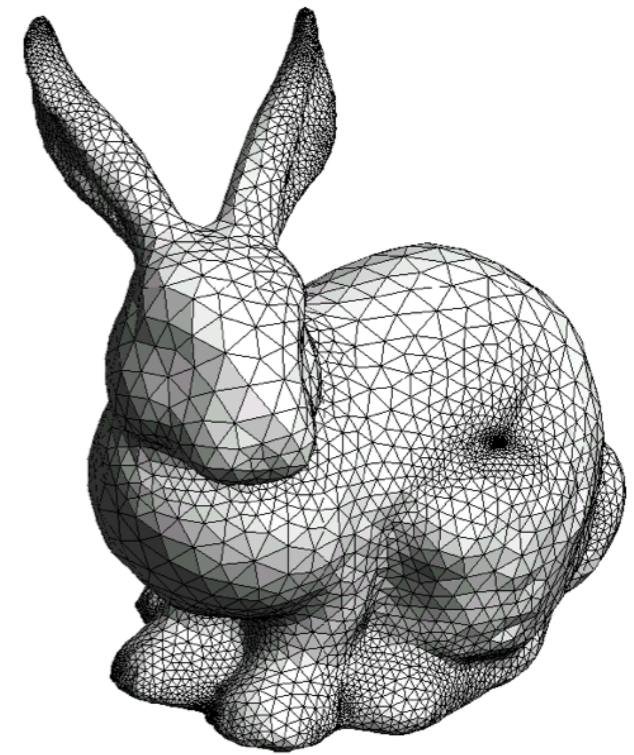
Last Time

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



Triangles (Clarification)

- Can be any shape or size
- Well-shaped triangles have advantages for numerical simulation
- Shape quality makes little difference for basic OpenGL rendering



Choice of Programming Language

- OpenGL lives close to the hardware
- OpenGL is not object-oriented
- OpenGL is not a functional language (as in, ML)
- Use C to expose and exploit low-level details
- Use C++, Java, ... for toolkits

Client/Server Model

- Graphics hardware and caching

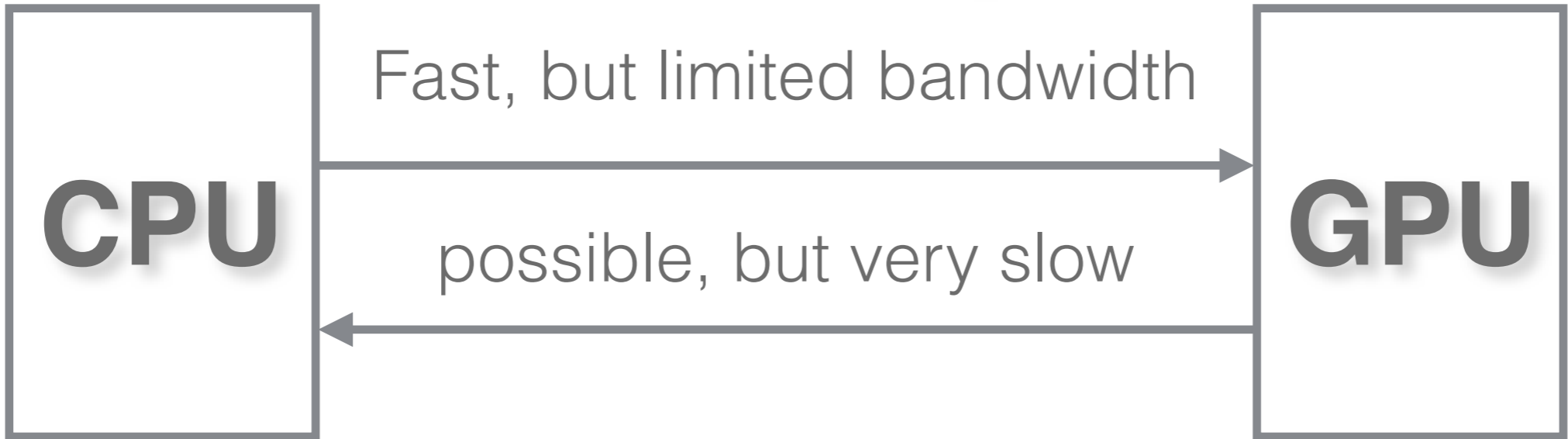


- Important for efficiency
- Need to be aware where data are stored
- Examples: vertex arrays, display lists

The CPU-GPU bus

AGP, PCI, PCI Express

Fast, but limited bandwidth



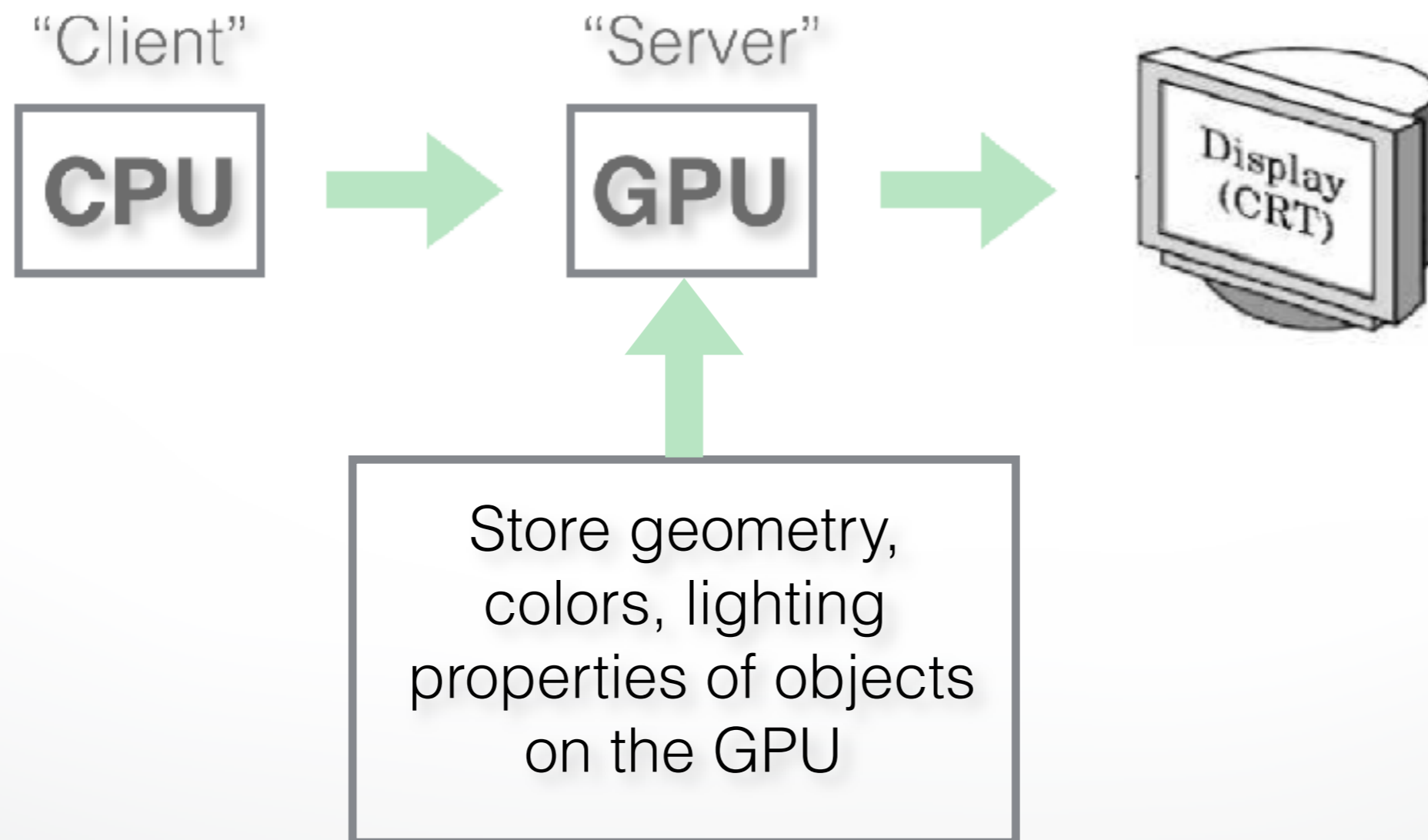
possible, but very slow



We need performance!

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 */
```

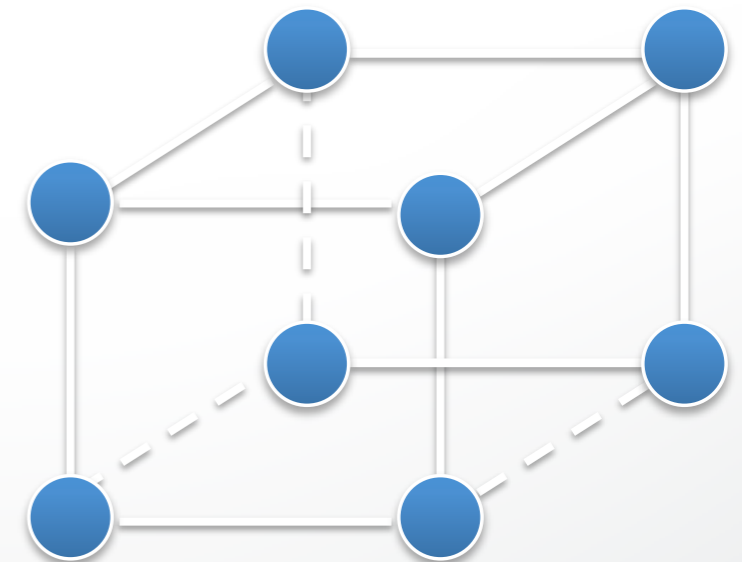
Display Lists Details

- Very useful with complex objects that are redrawn often (e.g., with transformations)
- Another example: fonts (2D or 3D)
- 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 $6 \times 4 = 24$ 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



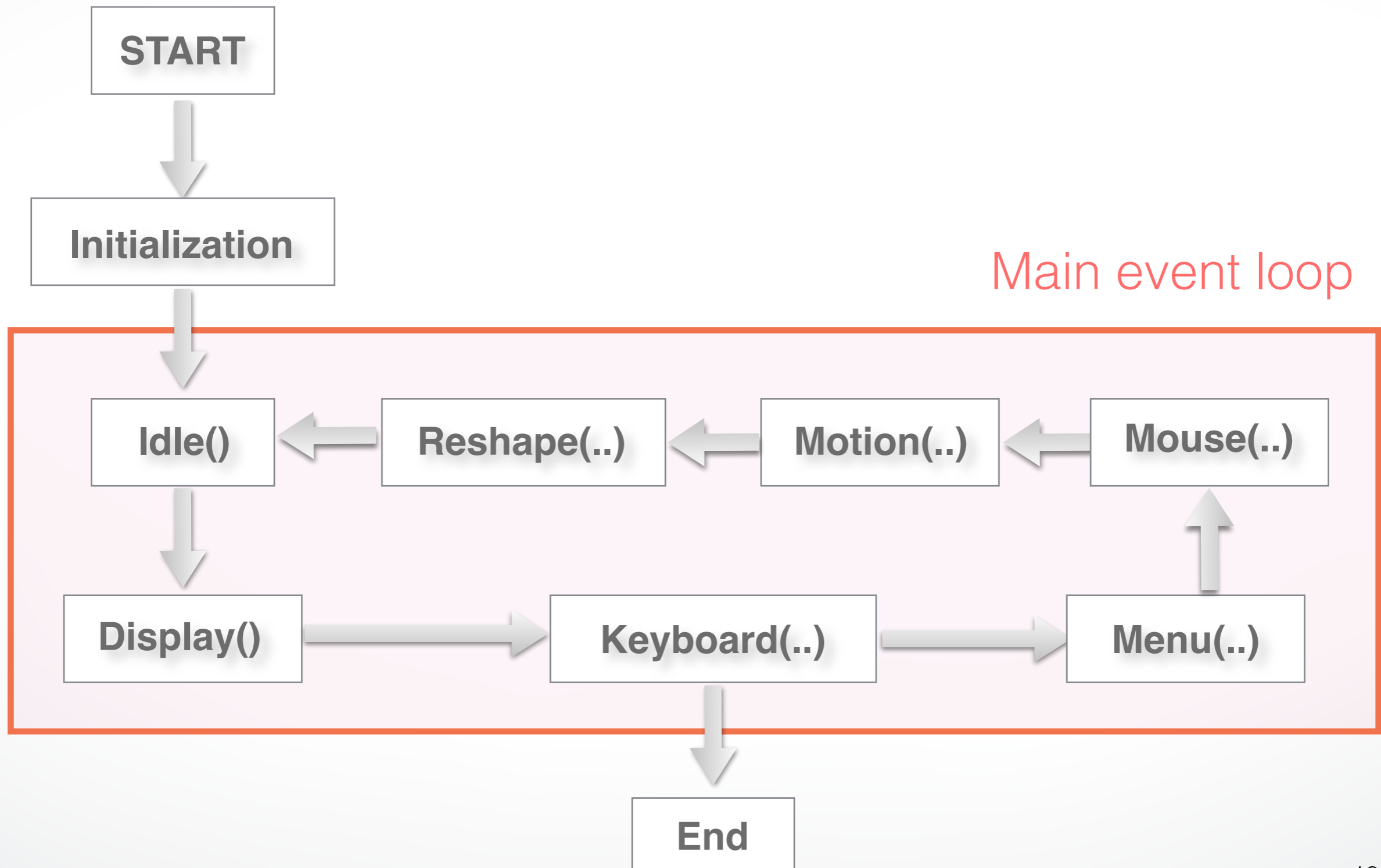
Vertex Buffer Objects (VBOs)

- Display Lists: Fast / inflexible
- Immediate mode: Slowest / flexible
- Vertex Array: Slow with shared vertices / flexible
- Vertex Buffer Objects (VBOs): Best of between Display List and Vertex Array: Fast / flexible
 - memory manager optimizes for buffer location in memory
 - mapping buffer into client memory space

Outline

- Client / Server Model
- **Callback**
- Double Buffering
- Hidden Surface Removal
- Simple Transformation
- Example

GLUT Program with Callbacks



Main Event Loop

- Standard technique for interaction (Glut, Qt, wxWidgets, ...)
- Main loop processes events
- Dispatch to functions specified by client
- Callbacks also common in operating systems
- “Poor man’s functional programming”

Types of Callbacks

- **Display()**: when window must be drawn
- **Idle()**: when no other events to be handled
- **Keyboard (unsigned char key, int x, int y)**: key pressed
- **Menu(...)**: mouse movement
- **Mouse(int button, int state, int x, int y)**: mouse button
- **Motion(...)**: mouse movement
- **Reshape(int w, int h)**: window resize
- Any callback can be NULL

Outline

- Client / Server Model
- Callback
- **Double Buffering**
- Hidden Surface Removal
- Simple Transformation
- Example

Screen Refresh

- Common: 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 `glFinish()` 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()`

Outline

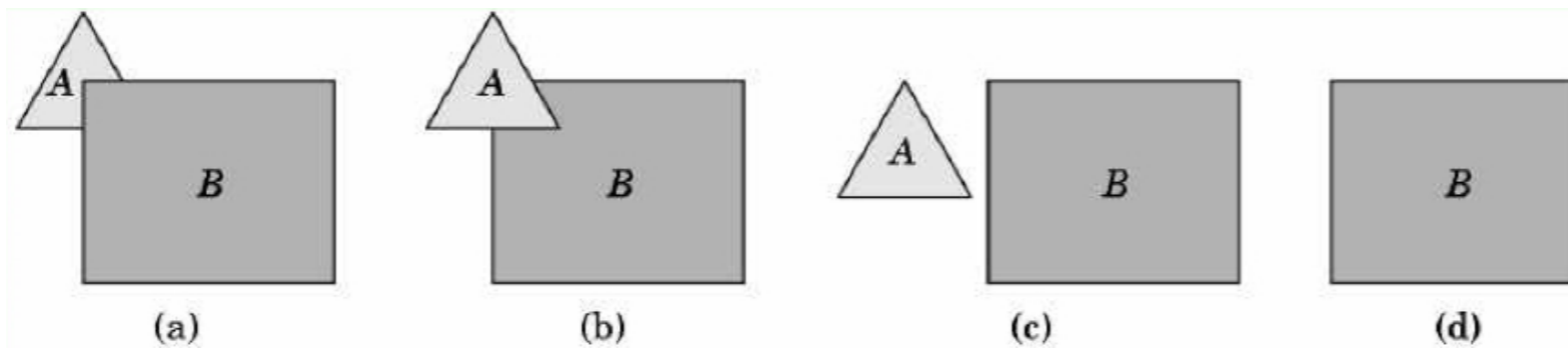
- Client / Server Model
- Callback
- Double Buffering
- **Hidden Surface Removal**
- Simple Transformation
- Example

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

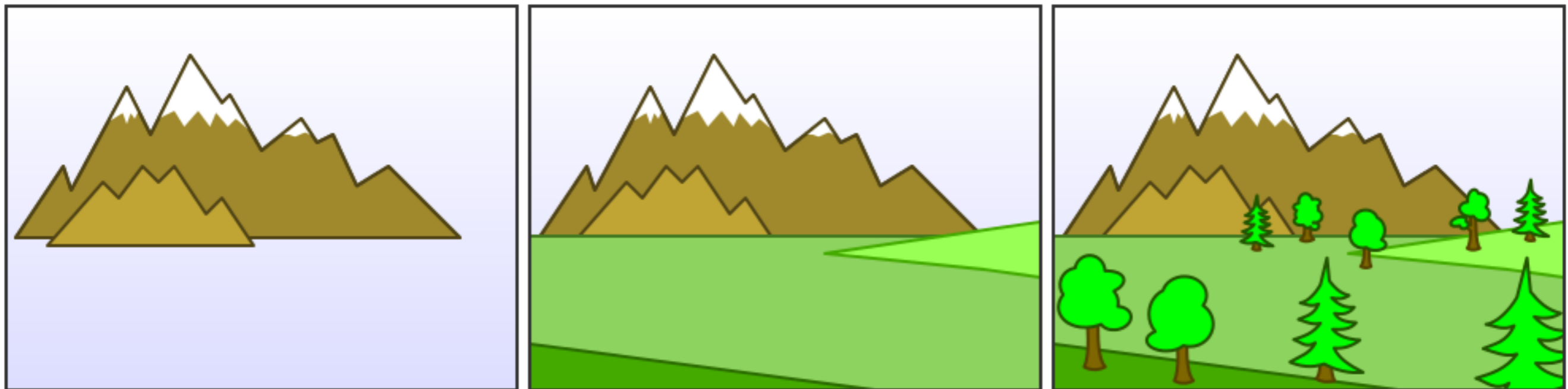
Object-Space Approach

- Consider objects pairwise



- Complexity $O(k^2)$ where $k = \#$ of objects
- Painter's algorithm: render back-to-front
- "Paint" over invisible polygons
- How to sort and how to test overlap?

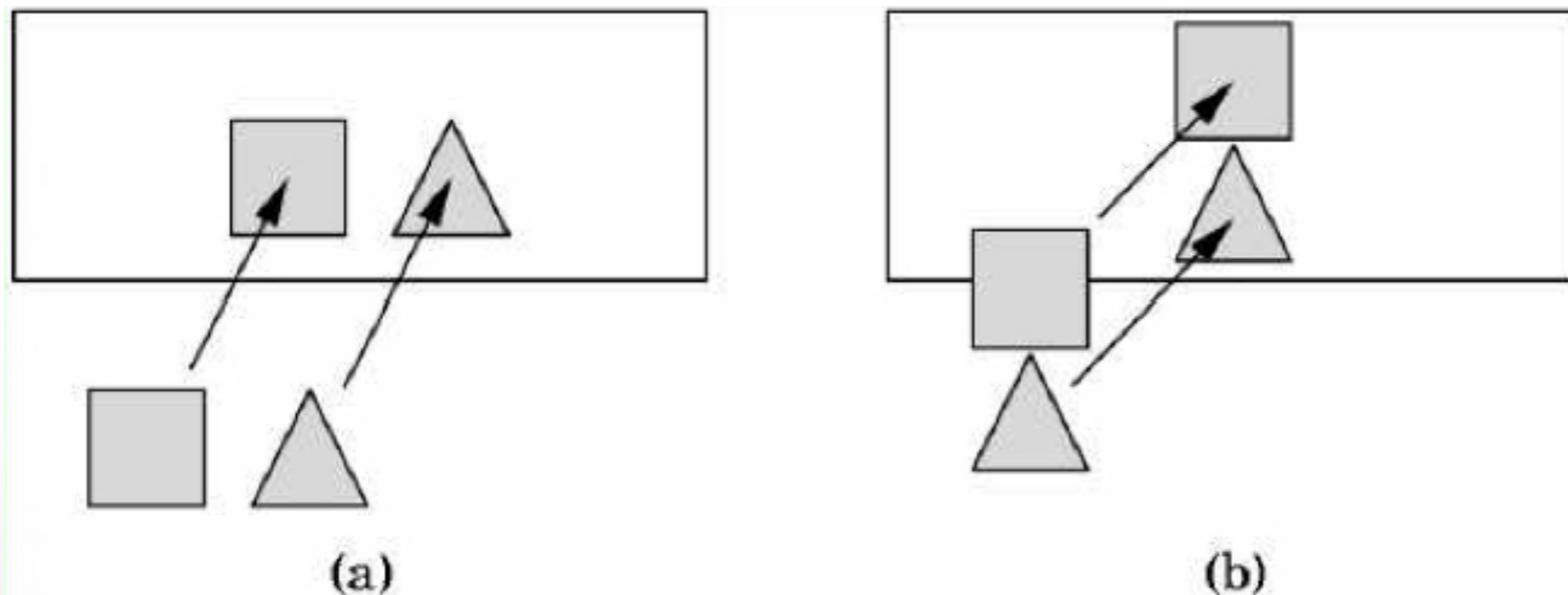
Painter's Algorithm



- Painter's VS reverse painter's algorithm

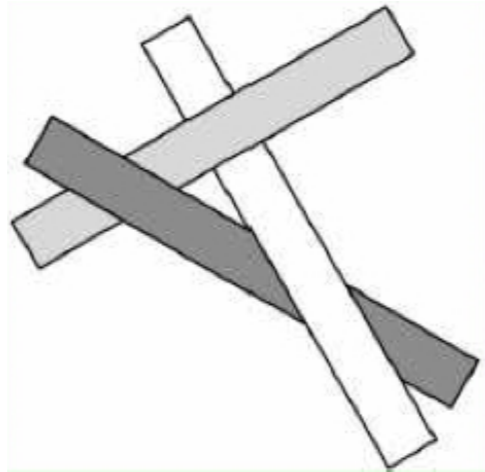
Depth Sorting

- 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

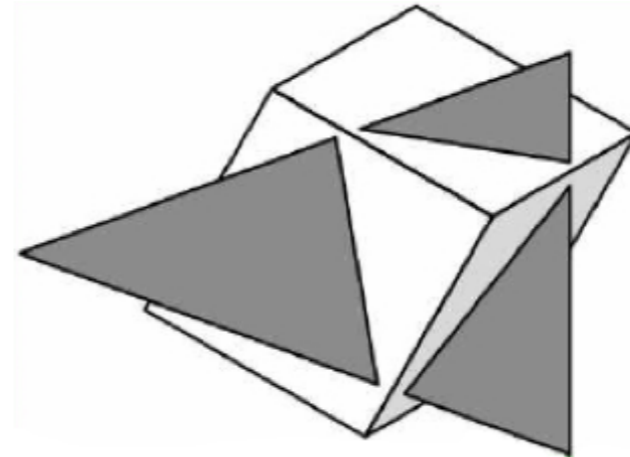


Some Difficult Cases

- Sometimes cannot sort polygons



Cyclic overlap



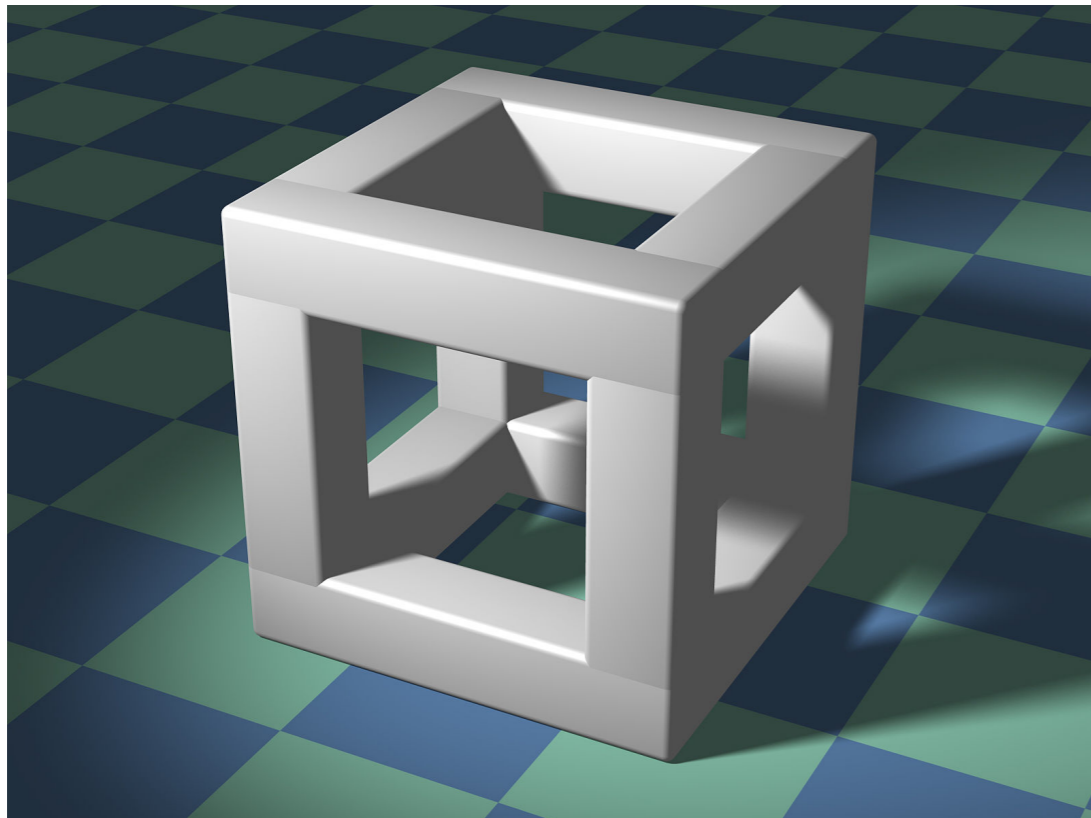
Piercing Polygons

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

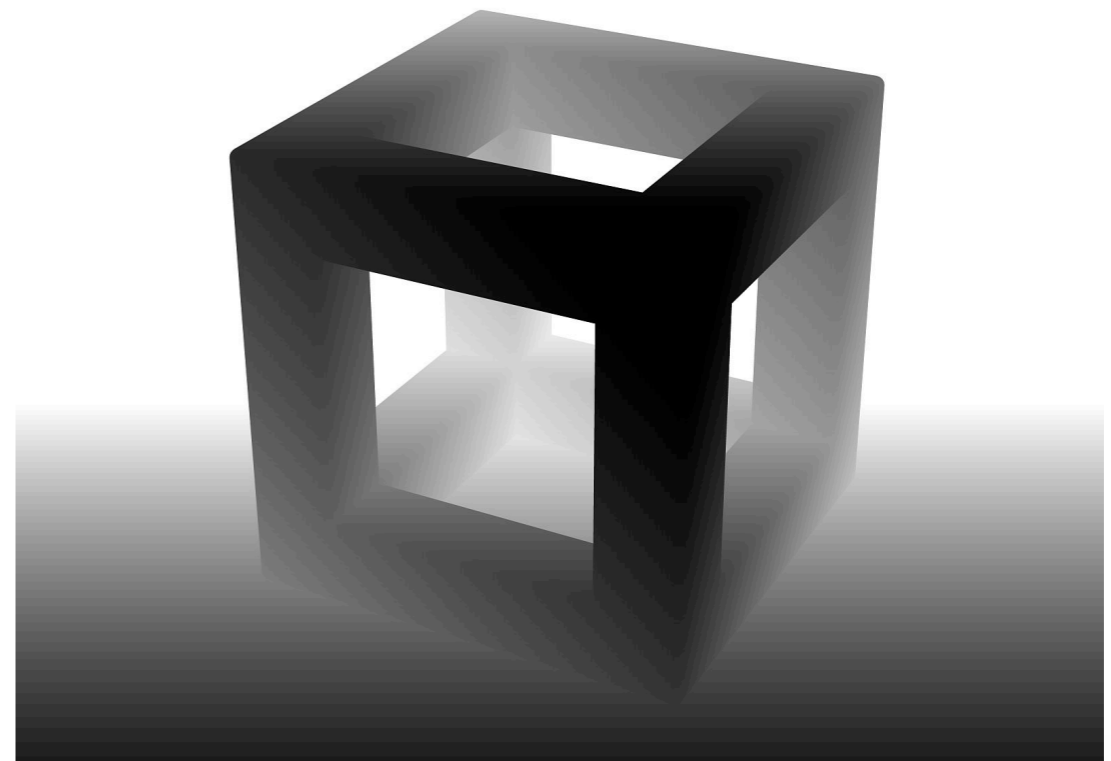
Painter's Algorithm Assessment

- 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



3D geometry



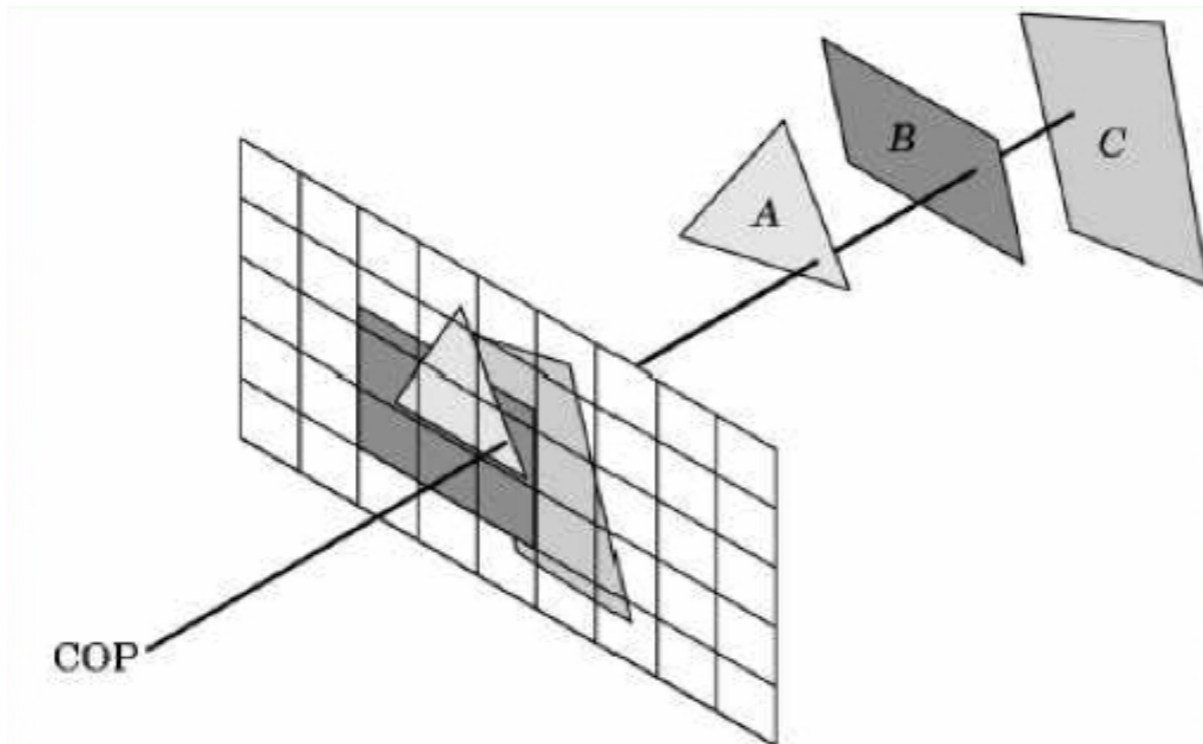
Depth Image
darker color is closer

Depth sensor camera



Image-Space Approach

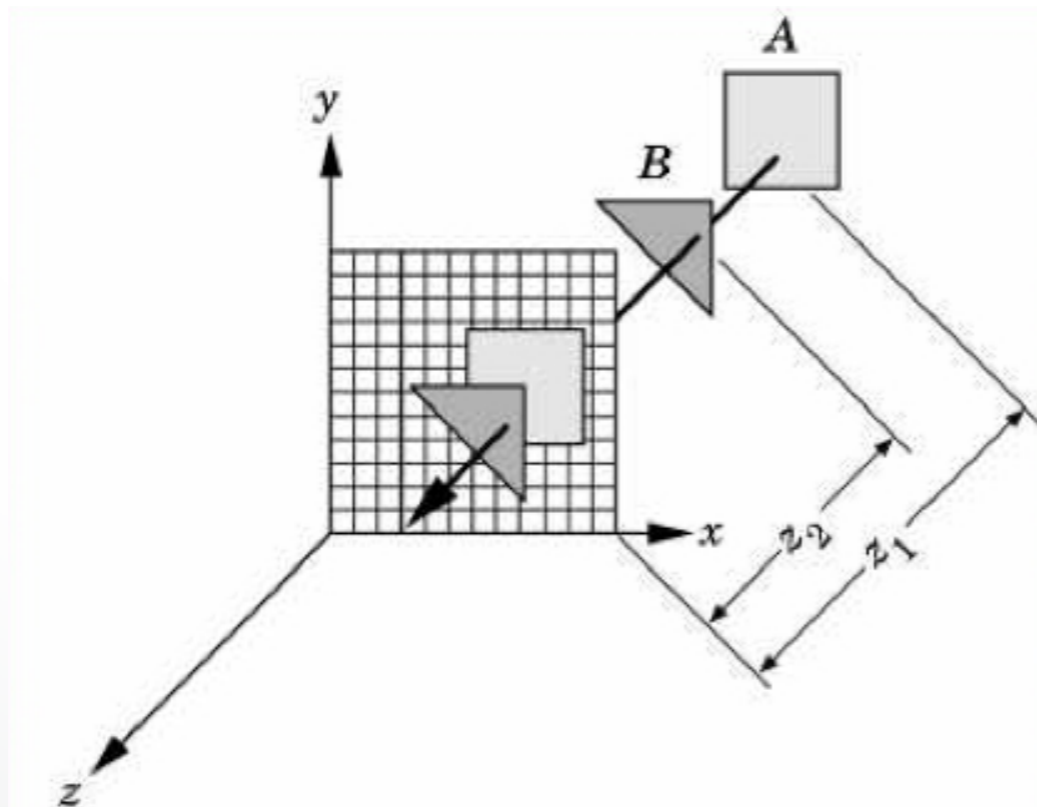
- Raycasting: intersect ray with polygons



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

The z-Buffer Algorithm

- z-buffer stores depth values z for each pixel
- Before writing a pixel into framebuffer:
 - Compute distance z of pixel from viewer
 - If closer, write and update z-buffer, otherwise discard



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; 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!

- Some “tricks” use z-buffer in read-only mode

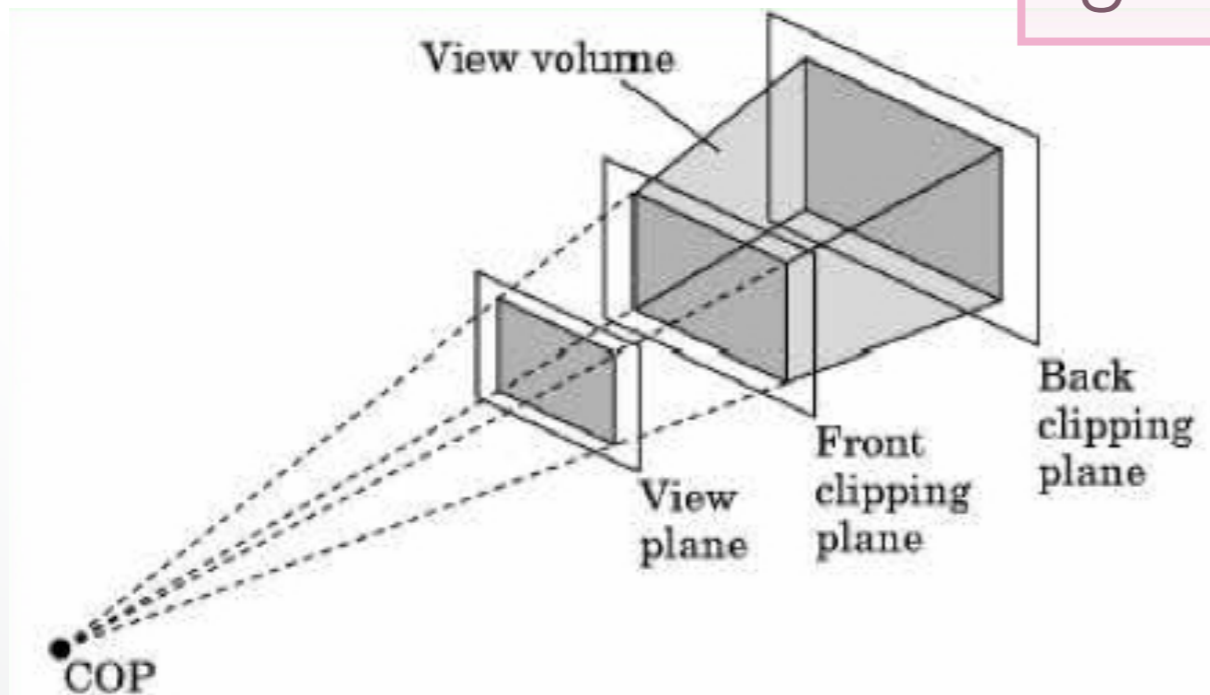
Outline

- Client / Server Model
- Callback
- Double Buffering
- Hidden Surface Removal
- **Simple Transformation**
- Example

Specifying the Viewing Volume

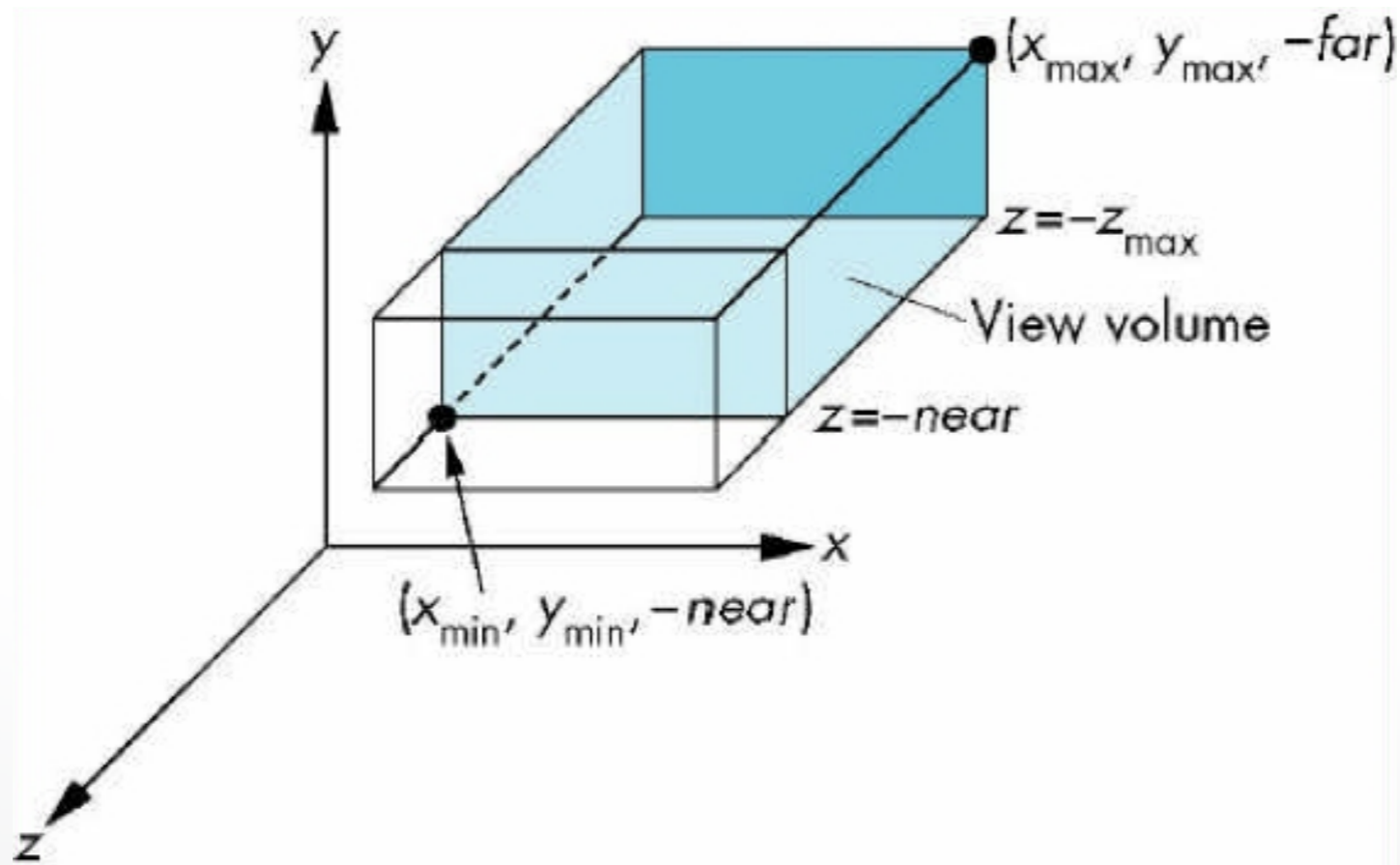
- Clip everything not in viewing volume
- Separate matrices for transformation and projection

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



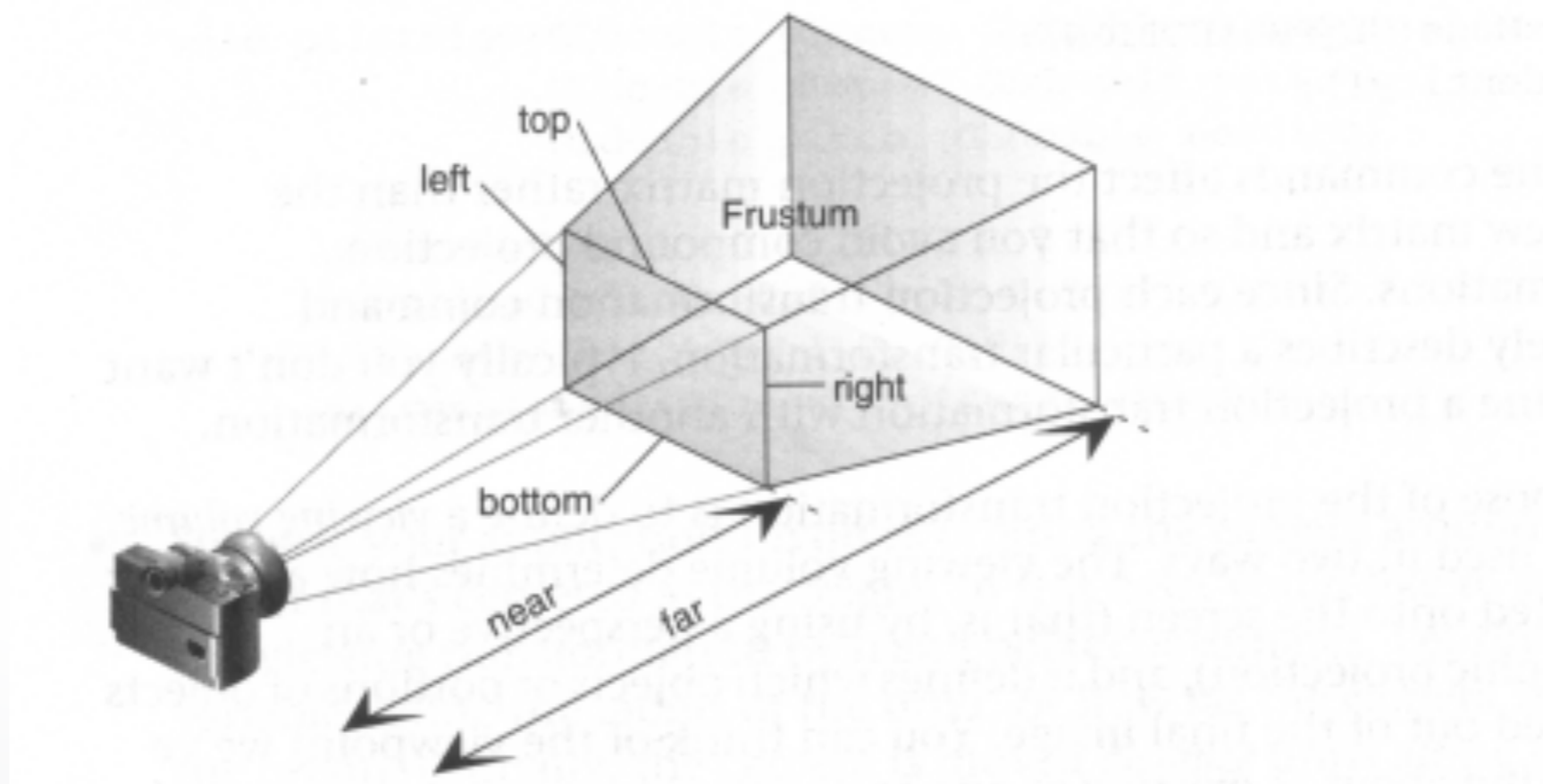
Parallel Viewing

- 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);
```

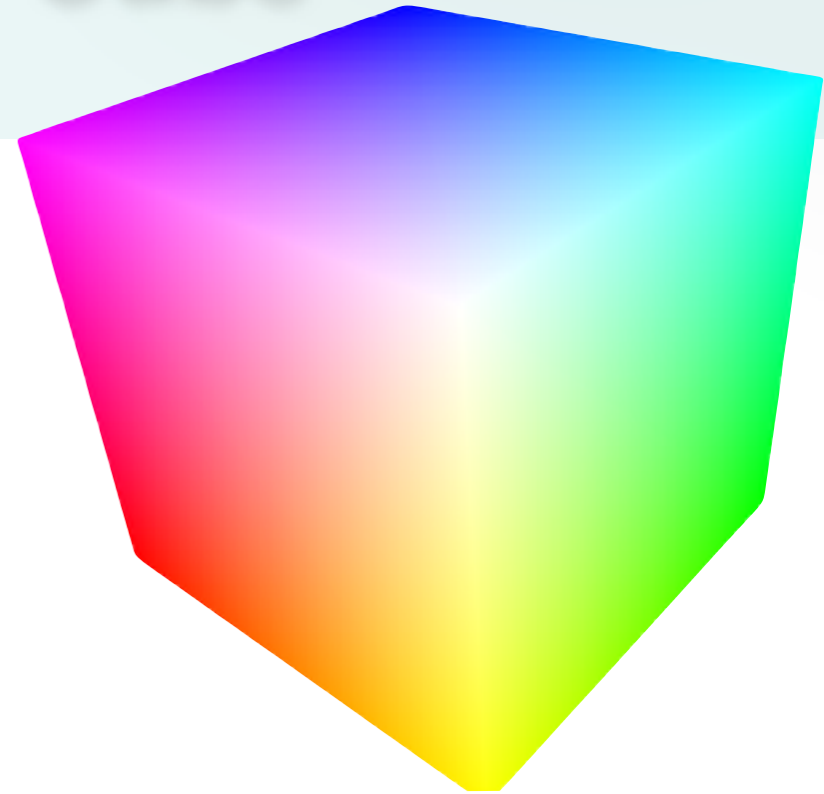
- Scale with a factor in the x, y, and z direction

```
glScale{fd}(x, y, z);
```

Outline

- Client / Server Model
- Callback
- Double Buffering
- Hidden Surface Removal
- Simple Transformation
- **Example**

Example: Rotating Color Cube



- Adapted from [Angel, Ch. 4]
- 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

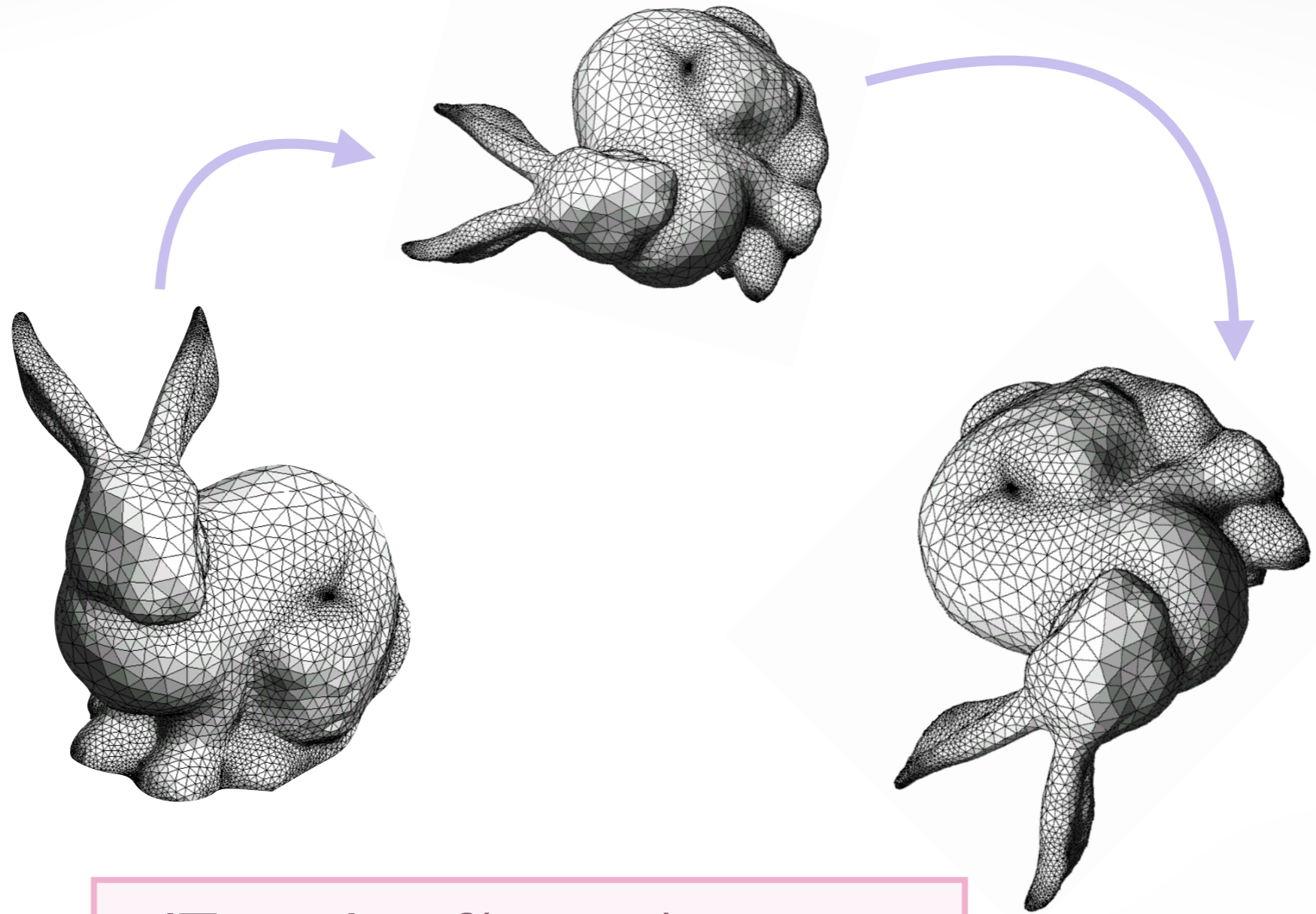
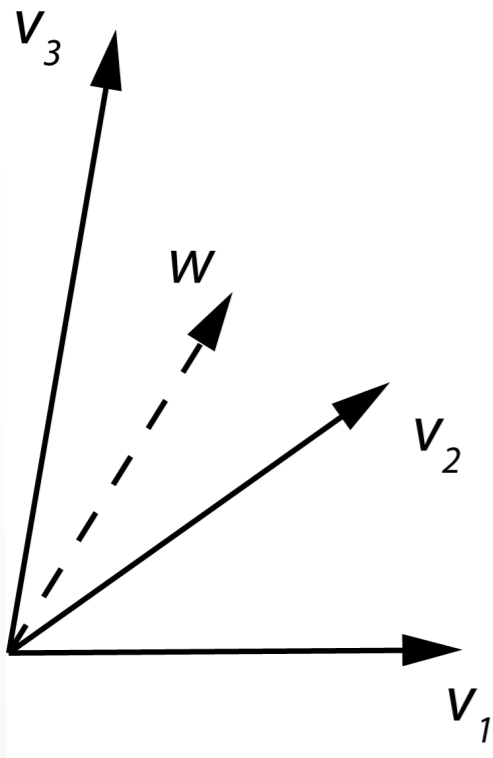
- Keyboard callback

```
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);
}
```

Summary

- Client/Server Model
- Callbacks
- Double Buffering
- Hidden Surface Removal
- Simple Transformations
- Example

Next Time: Transformations



```
glTranslatef(x, y, z);  
glRotatef(angle, ax, ay, az);  
glScalef(sx, sy, sz);
```

<http://cs420.hao-li.com>

Thanks!

