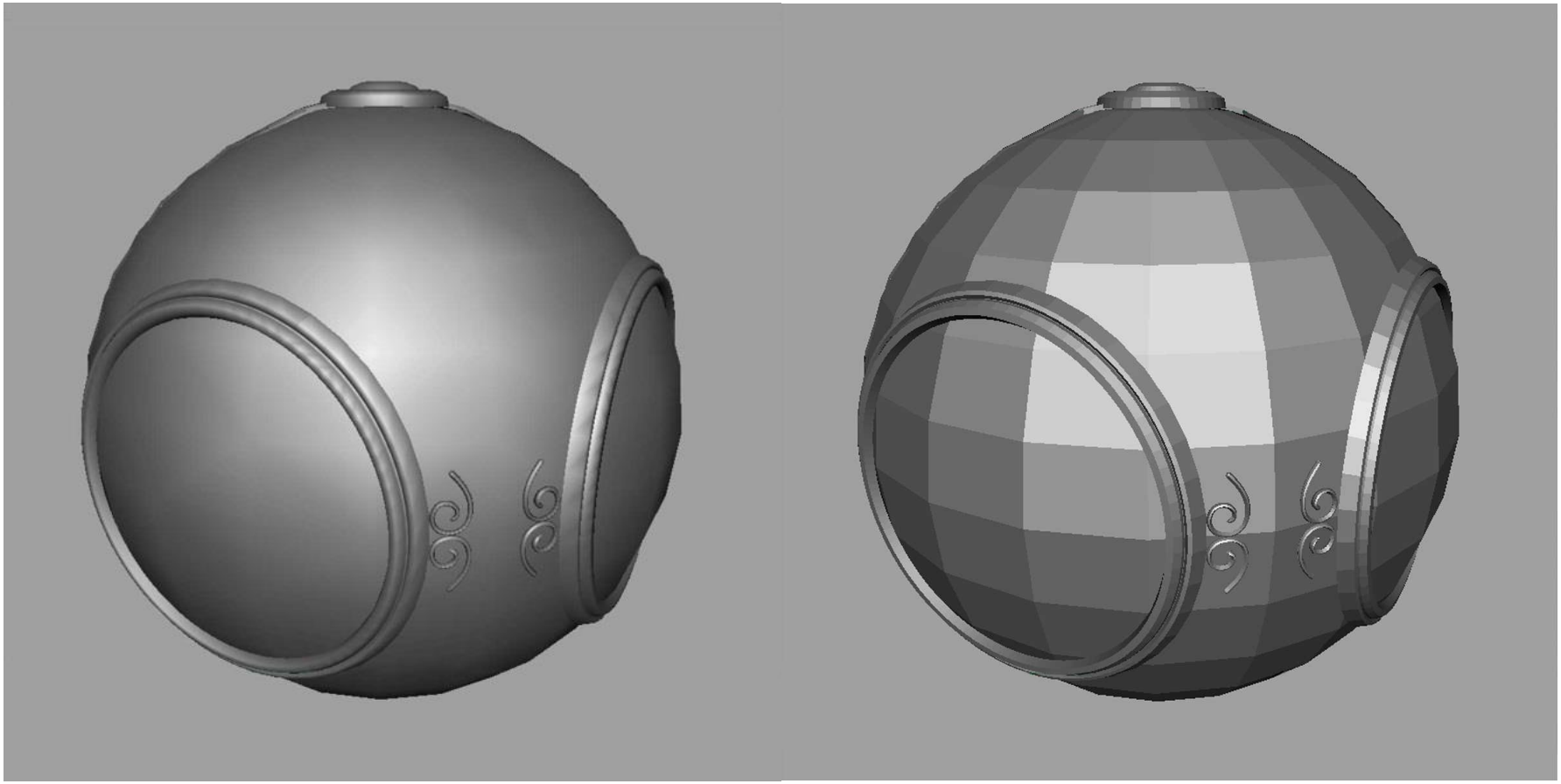
LEC16: Hidden Surface Removal

Ku-Jin Kim
School of Computer Science & Engineering
Kyungpook National University

Contents

- Flat shading
- Hidden surface removal
- Polygon issues

Smooth Shading vs. Flat Shading



Flat Shading in GLSL

- GLSL interpolates the vertex colors
  - Default : smooth shading
- For flat shading, required features are
  - ‘flat’ interpolation qualifier in GLSL
  - glProvokingVertex

glProvokingVertex

void **glProvokingVertex**(GLenum *provokeMode*);

`provokeMode`:
`GL_FIRST_VERTEX_CONVENTION` or
`GL_LAST_VERTEX_CONVENTION`

- specify the vertex to be used as the source of data for flat shaded in/out variables

Interpolation Qualifiers

- Outputs from shader (**out**) and inputs to a shader (**in**) can be further qualified with one of these
- interpolation qualifiers
  - **smooth:** perspective correct interpolation
  - **flat:** no interpolation
  - **noperspective:** linear interpolation

Flat Shading in GLSL (1)

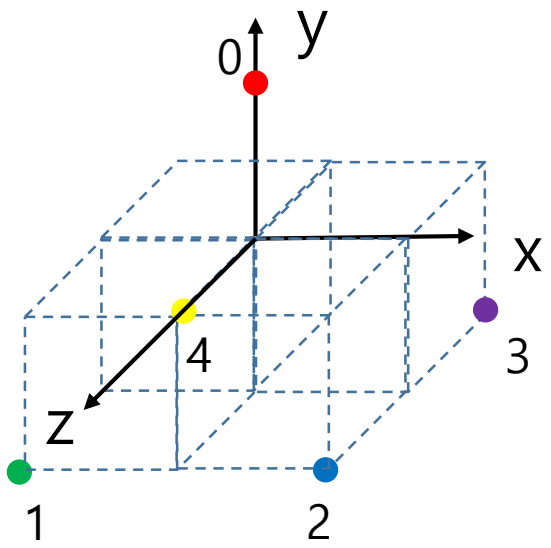
```
static char* vsSource = "#version 130 ₩n₩
in vec4 aPosition; ₩n₩
in vec4 aColor; ₩n₩
flat out vec4 vColor; ₩n₩
uniform mat4 urotate;  ₩n₩
void main(void) { ₩n₩
  mat4 scalemat = mat4(3.0); ₩n₩
  scalemat[3][3] = 1.0; ₩n₩
  gl_Position = urotate*scalemat*aPosition; ₩n₩
  vColor = aColor; ₩n₩
}";

static char* fsSource = "#version 130 ₩n₩
flat in vec4 vColor; ₩n₩
void main(void) { ₩n₩
  gl_FragColor = vColor; ₩n₩
}";

glProvokingVertex(GL_FIRST_VERTEX_CONVENTION);    // in myinit()
```

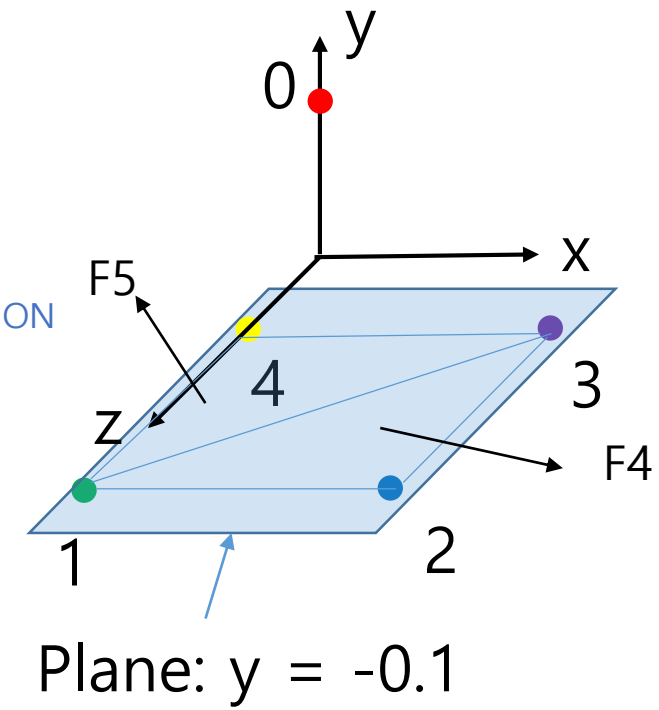
flat out, flat in :
GLSL 1.3 or upper versions

OpenGL 3.2 or upper versions

Flat Shading in GLSL (2)

```
GLfloat vertices[] = {
        0.0, 0.1, 0.0, 1.0,       // 0
        -0.1, -0.1, +0.1, 1.0,  // 1
        0.1, -0.1, +0.1, 1.0,  // 2
        0.1, -0.1, -0.1, 1.0,  // 3
        -0.1, -0.1, -0.1, 1.0, // 4
};
GLfloat colors[] = {
        1.0, 0.0, 0.0, 1.0,  //0 red
        0.0, 1.0, 0.0, 1.0,  //1 green
        0.0, 0.0, 1.0, 1.0,  //2 blue
        1.0, 0.0, 1.0, 1.0,  //3 purple
        1.0, 1.0, 0.0, 1.0    //4 yellow
};
GLushort indices[] = { //GL_FIRST_VERTEX_CONVENTION
        0, 1, 2,    // F0    red
        2, 3, 0,    // F1    blue
        4, 0, 3,    // F2    yellow
        1, 0, 4,    // F3    green
        2, 3, 1,    // F4    blue
        3, 4, 1     // F5    purple
};
```

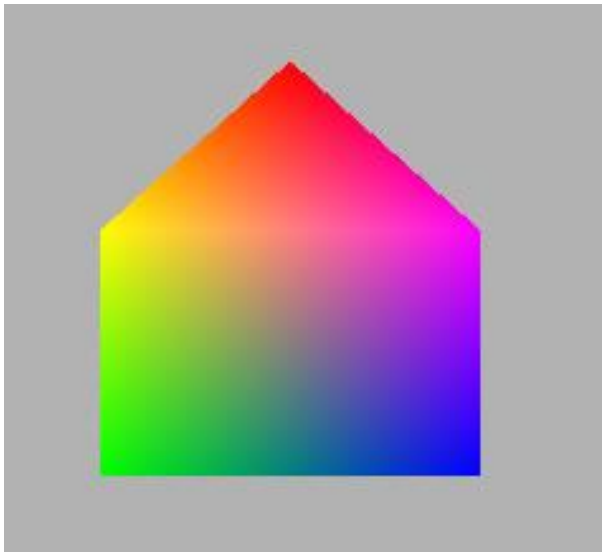


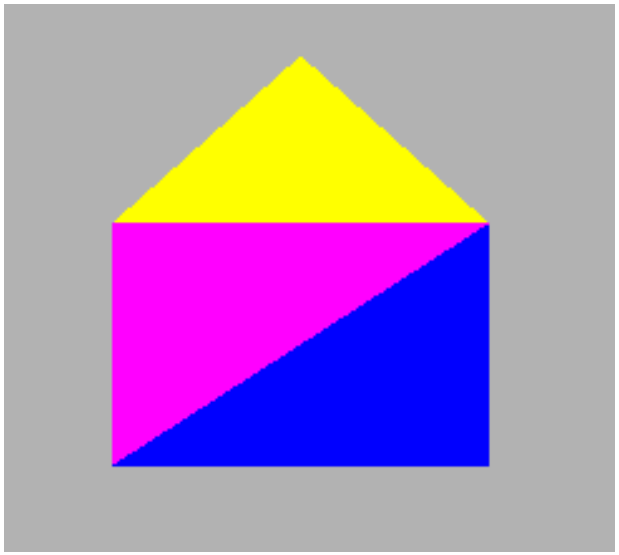


LEC16_flat_depth.c

Smooth Shading vs. Flat Shading

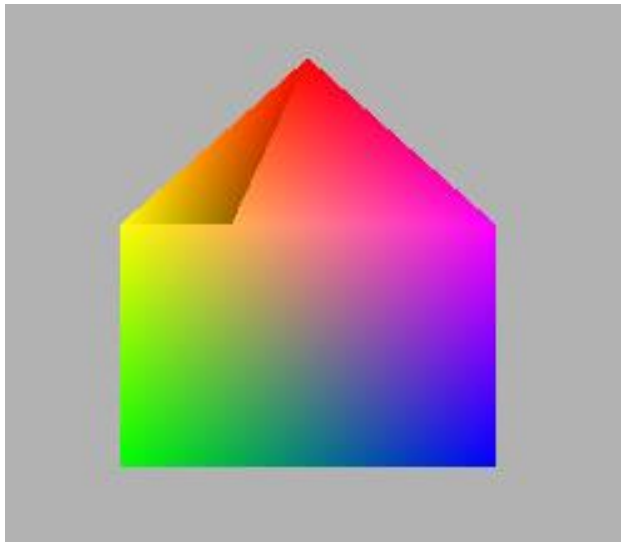
After rotating about x-axis

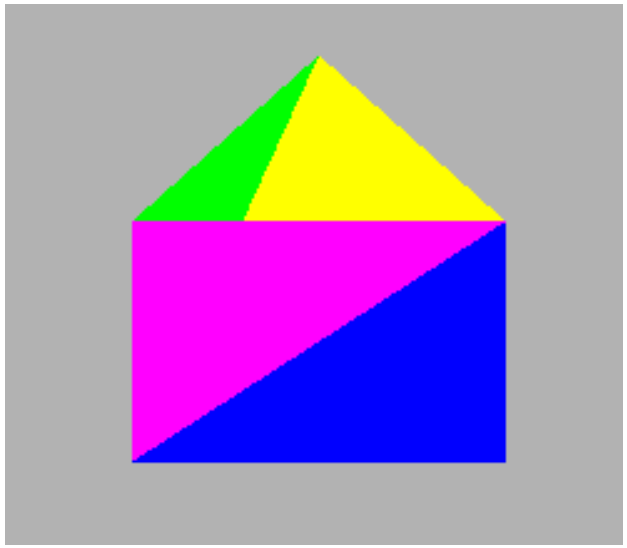




Possibly we can get …

After rotating about x-axis





Hidden surface removal was not applied !!!

Why Hidden Surface Removal?

- In OpenGL, faces (triangles) are drawn w.r.t. the order that they are defined
- No consideration of the hidden surfaces

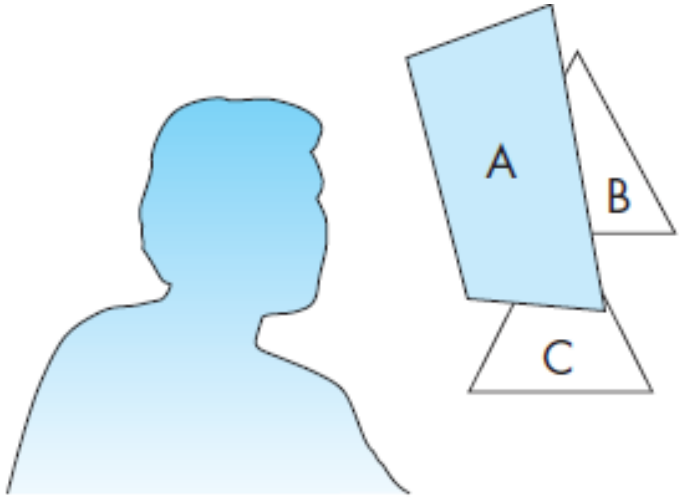
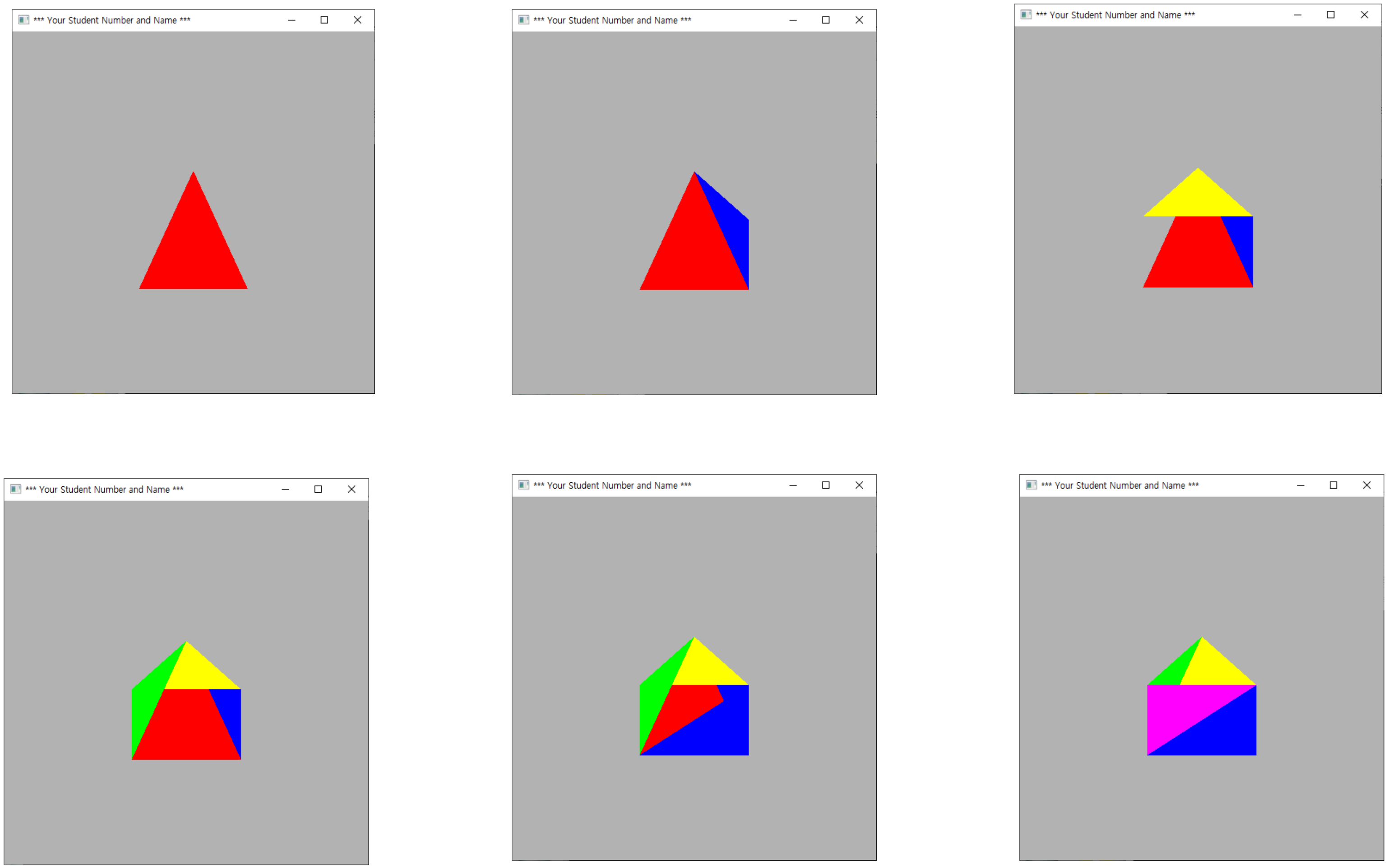


FIGURE 2.43 The hidden-surface problem.

Face Drawing One by One without Using Hidden Surface Removal



Hidden Surface Removal

- Painter's algorithm
  - Software solution
  - Sort the faces by the distance to the viewer
  - Draw the faces from the farthest to the closest
  - Some cases are not solvable

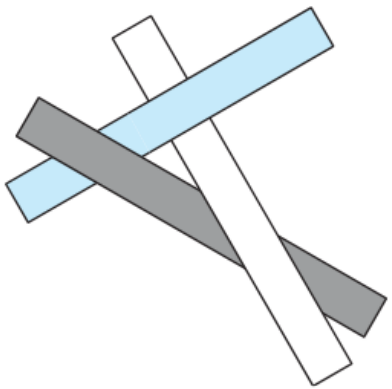


FIGURE 6.57 Cyclic overlap.

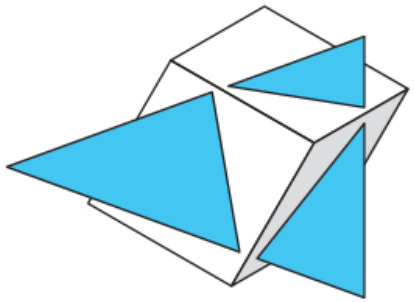


FIGURE 6.58 Piercing polygons.

- Z-buffer (depth buffer) algorithm
  - Hardware solution

Z-Buffer Algorithm (1)

- We need to determine which fragments correspond to objects that are visible, namely, those that are in the view volume and are not blocked from view by other objects closer to the camera.

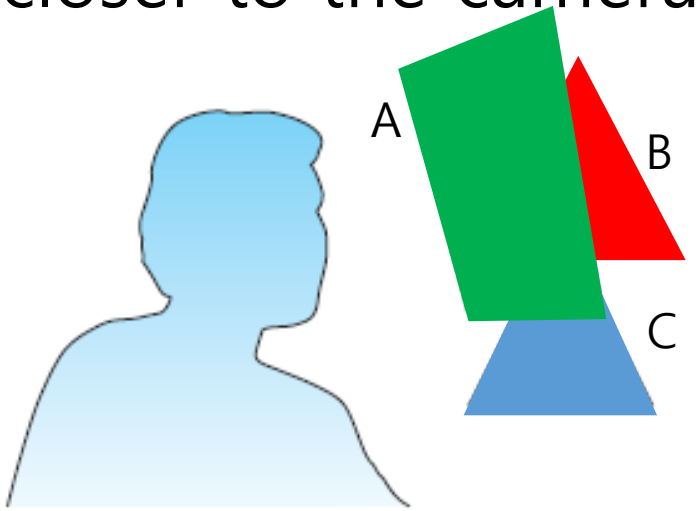
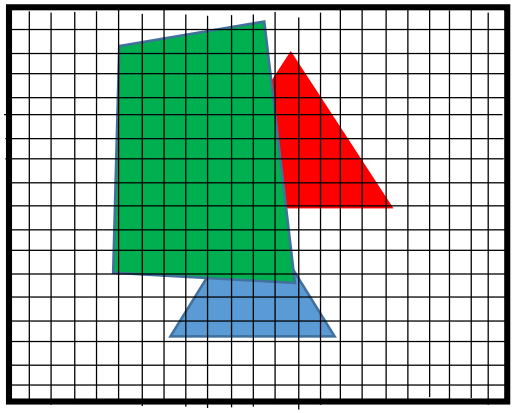


FIGURE 2.43 The hidden-surface problem.



Rendered image

Z-Buffer Algorithm (2)

1. 1-to-1 correspondence between the pixels in z-buffer and frame buffer
2. z-buffer pixel keeps the distance value of the closest object from the viewer
3. Corresponding frame buffer pixel is rendered with the closest object

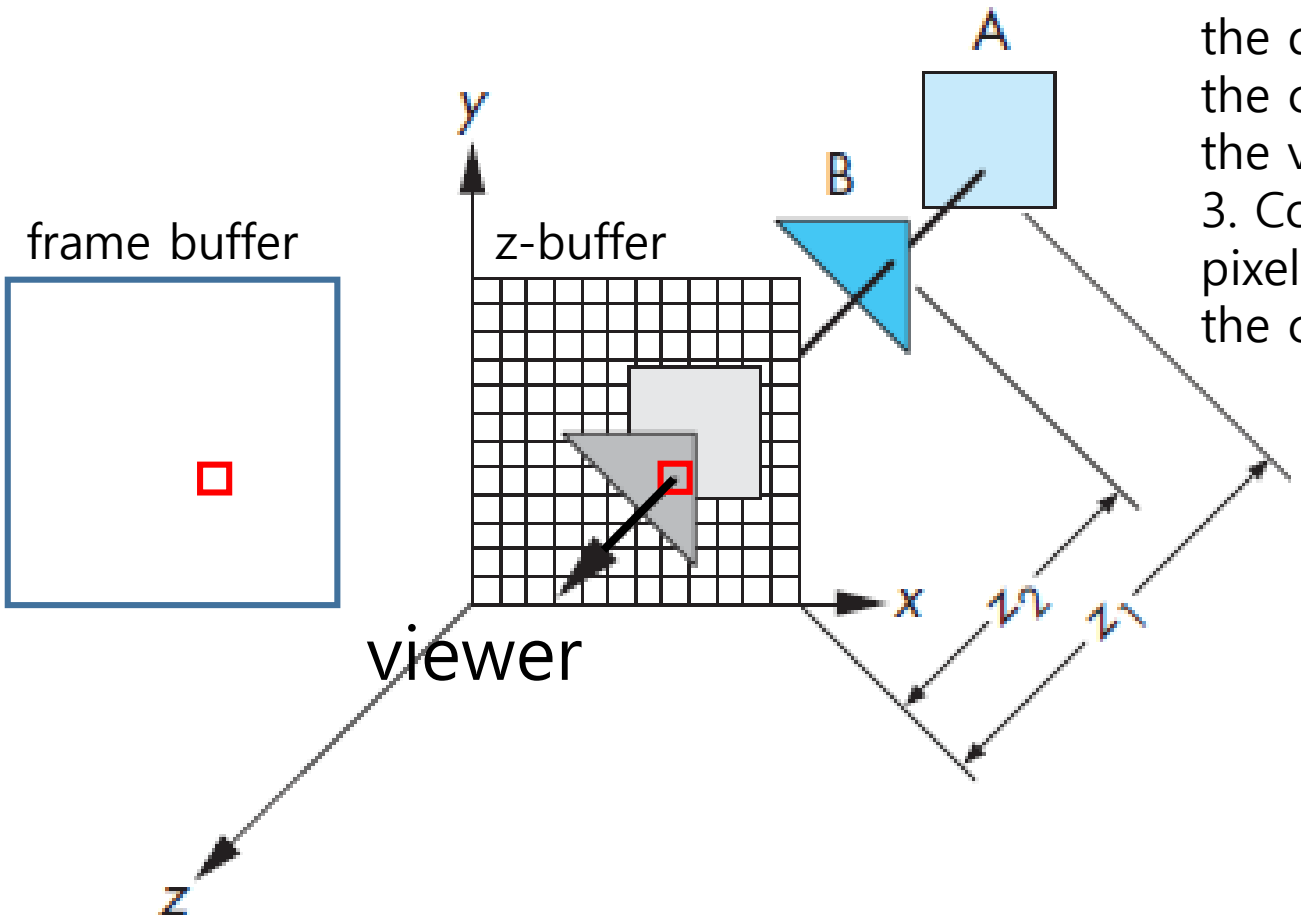


FIGURE 6.49 The z-buffer algorithm.

LEC16_flat_depth.c

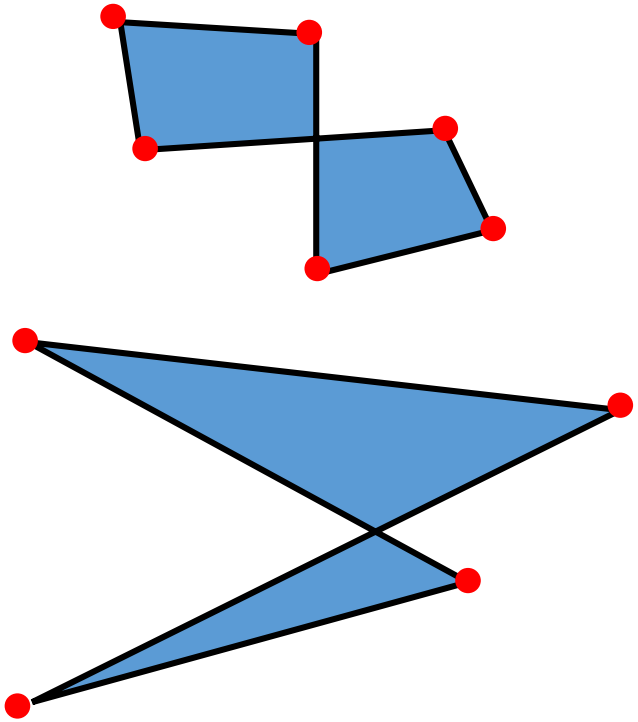
- To use depth buffer
  - glEnable(GL_DEPTH_TEST); // in myinit() function
  - glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  - glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);

Polygon Issues in OpenGL

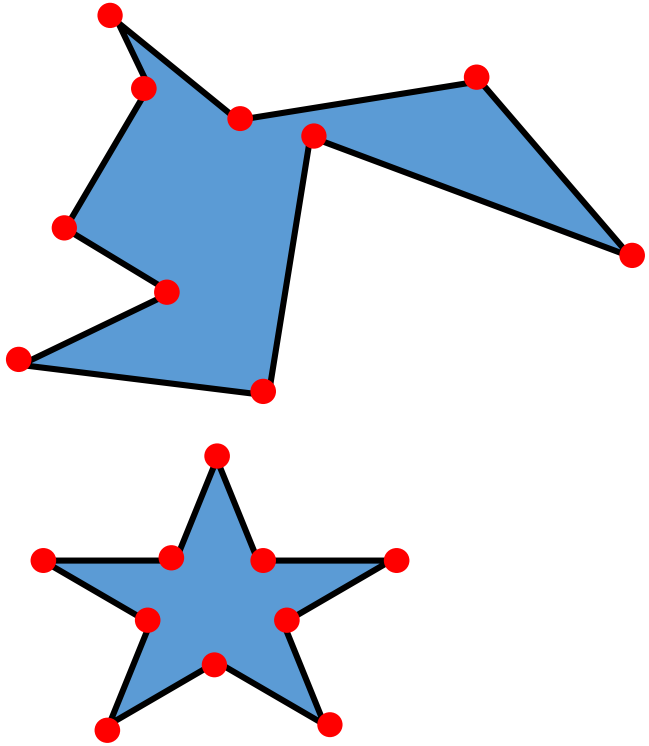
- Polygons must be
  - Simple
  - Convex
  - Flat

Simple Polygon

- If there is no intersection between two edges in the polygon, then the polygon is a simple polygon.



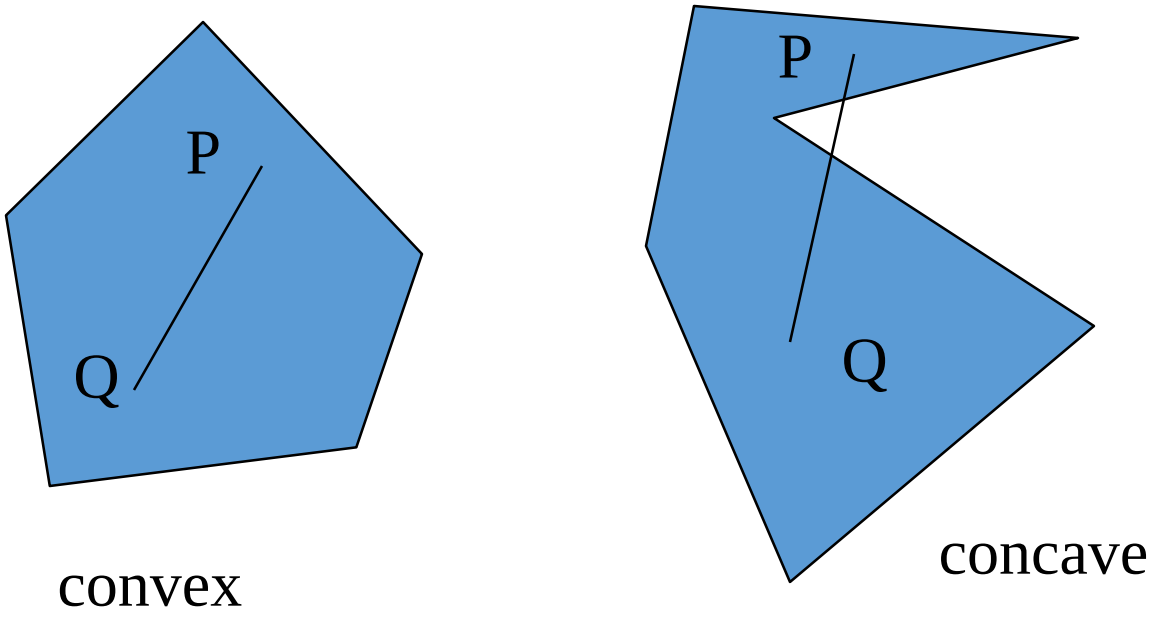
nonsimple polygon



simple polygon

Convex Polygon

- An object is convex iff for any two points in the object all points on the line segment between these points are also in the object



Flat Polygon

- The vertices of the polygon are embedded in a plane
- Can you make an object with 4 or more vertices as a flat polygon?

  Ex) GLfloat vertices[] = {

  0.5, 0.5, 0.5, 1.0,

  0.5, 0.3, 0.4, 1.0,

  -0.2, 0.2, 0.3, 1.0,

  -0.2, 0.4, 0.0, 1.0};

Polygon Testing

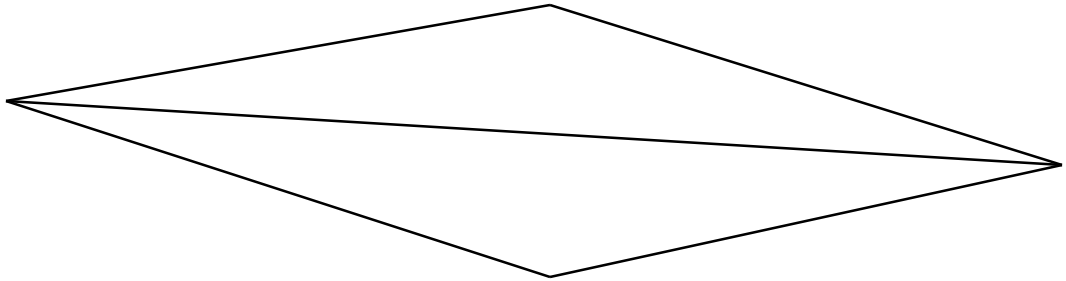
- Conceptually simple to test for simplicity and convexity
- Time consuming
- Earlier versions assumed both and left testing to the application
- Modern OpenGL only renders triangles
- Need algorithm to triangulate an arbitrary polygon

Why Triangles?

- OpenGL will display only triangles, since they are
  - Simple: edges cannot cross
  - Convex: all points on line segment between any two points in a polygon are also in the polygon
  - Flat: all vertices are in the same plane
- Application program must tessellate a polygon (or vertex set) into triangles (triangulation)
- OpenGL 4.1 contains a tessellator

Good and Bad Triangles

- Long thin triangles are rendered badly



- Equilateral triangles are rendered well
  - Maximize minimum angle
- Issues on triangulation for unstructured points
  - Delaunay triangulation

HW#16 Try the commands you learned today and file input

- NO SUBMISSION is required for this homework. This homework will not be evaluated.
- Try to run the program with depth buffer and flat shading.
- Write a program to input data from a file with the following commands:
  - fopen_s
  - fscanf_s
  - fclose