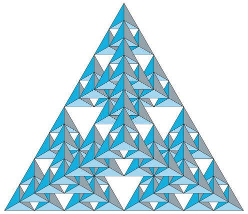
CITS3003 Graphics & Animation

Lecture 12:
3D Hidden Surface Removal

Content

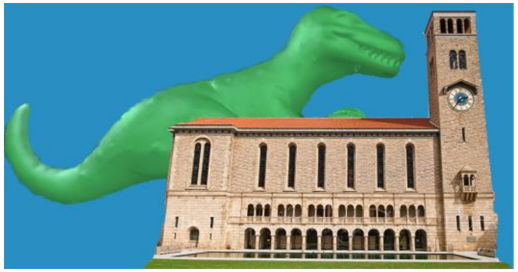
- Look into a more sophisticated three-dimensional example
  - Sierpinski gasket: a fractal



- Introduce hidden-surface removal
  - The *z-buffer algorithm*
  - The *Painter's algorithm*
- Animation and double buffering

Three-dimensional Applications

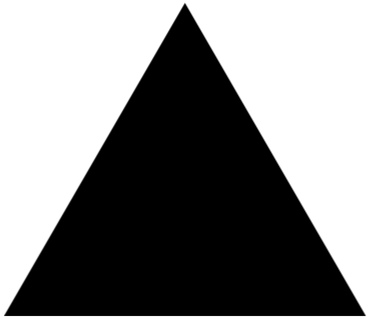
- In OpenGL, two-dimensional applications are a special case of three-dimensional graphics
- Going from 2D to 3D:
  - ❑ 2D points have (x,y) coordinates → 3D points have (x,y,z) coordinates
  - ❑ Use **vec3**, **glUniform3f**
  - ❑ Need to worry about the order in which primitives are rendered, or
  - ❑ Need to do hidden-surface removal



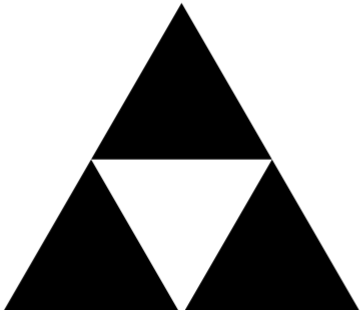
Example: Sierpinski Gasket (2D)

The recursive algorithm:

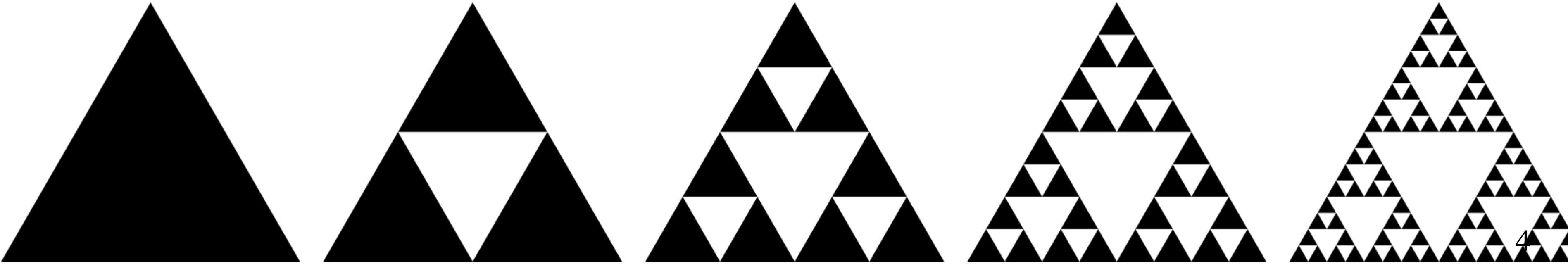
- Step 1
  - Start with one equilateral triangle



- Step 2
  - Cut smaller triangles out of its center



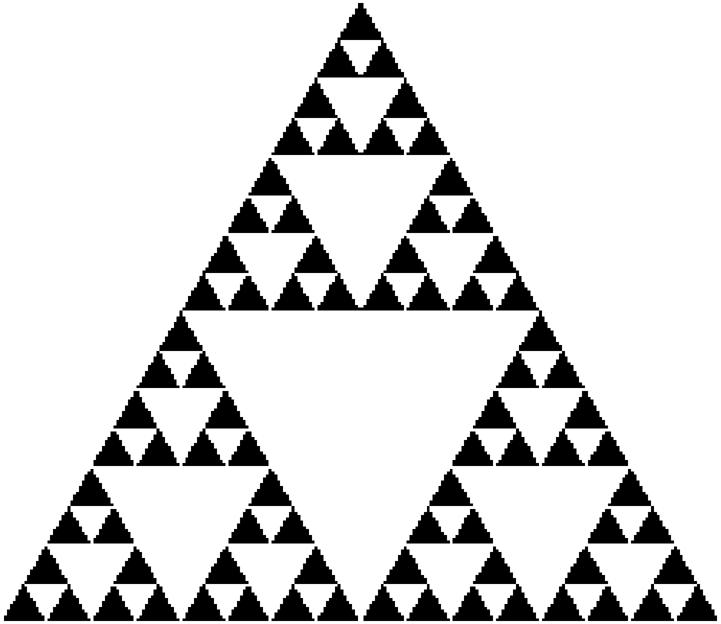
- Repeat the step#2 with each smaller Triangle



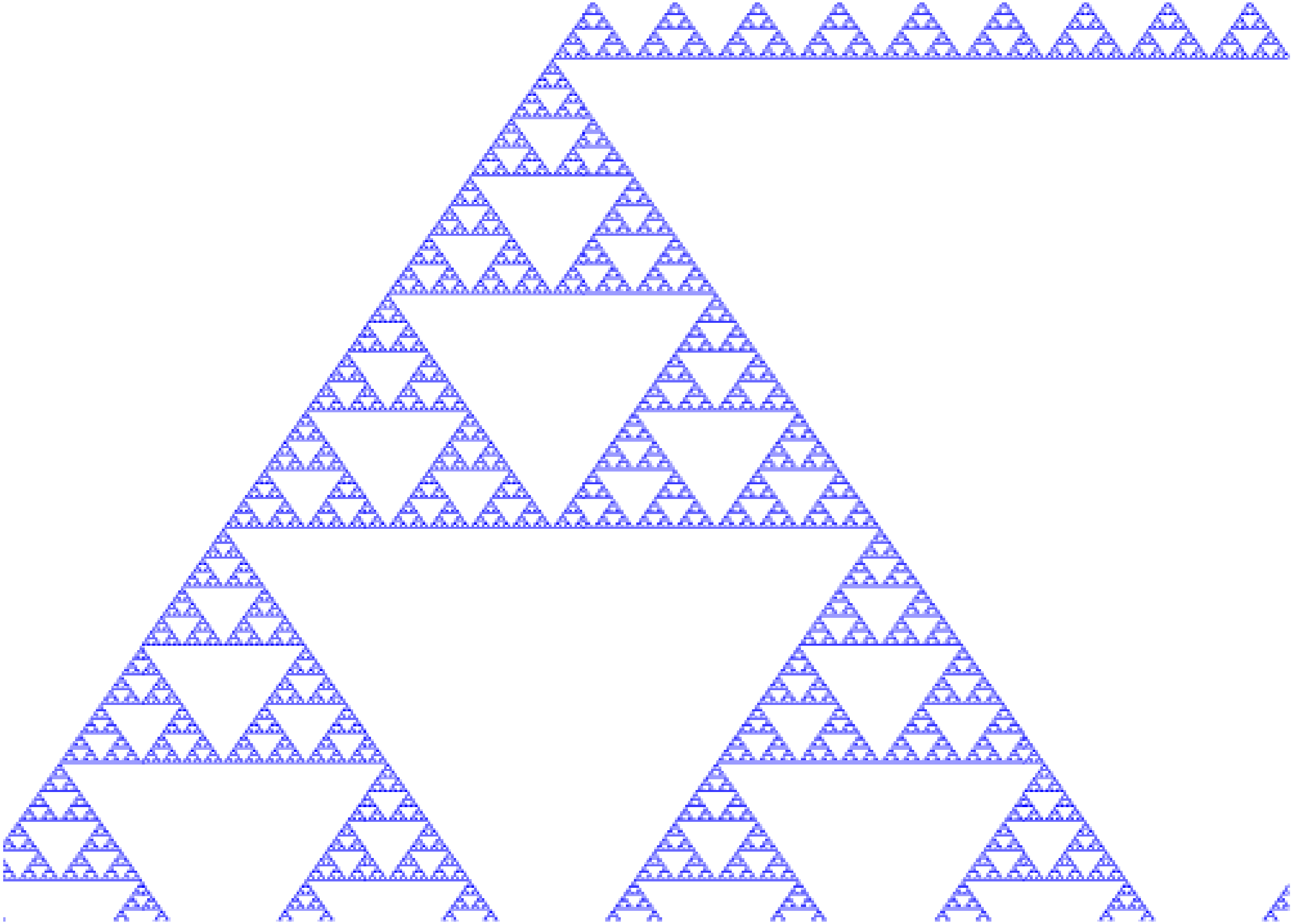
https://en.wikipedia.org/wiki/Sierpi%C5%84ski_triangle#/media/File:Sierpinski_triangle_evolution.svg

Example: Sierpinski Gasket (2D)

- Output from five subdivisions



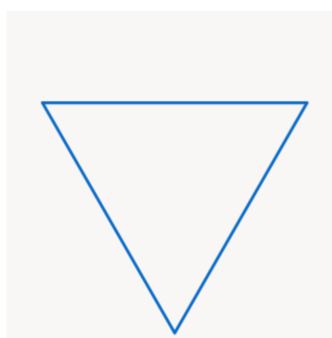
Self-similarity



https://upload.wikimedia.org/wikipedia/commons/6/6b/Sierpinski_zoom_2.gif

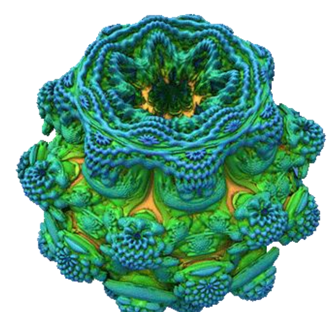
Fractals in Graphics

"My power flurries through the air into the ground.
My soul is spiraling in frozen fractals all around.
And one thought crystallizes like an icy blast-
I'm never going back; the past is in the past!"
Queen Elsa, Frozen



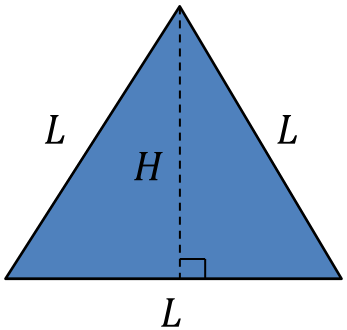
https://brilliant.org/daily-problems/koch-snowflake/

Mandlebulb



The Sierpinski Gasket is a Fractal

- Consider the filled area (blue) and the perimeter (the length of all the lines around the filled triangles)

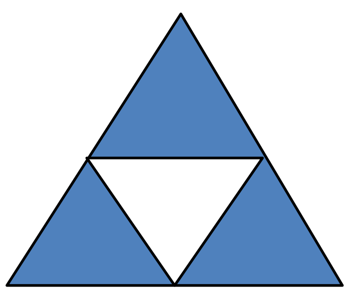


(equilaterial triangle)

  - **Level 0:** whole triangle is filled

    Area = $\frac{LH}{2}$

    Perimeter = $3L$



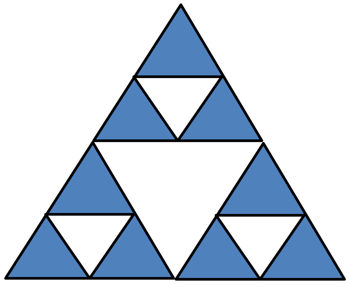
  - **Level 1:** 3 quarters of the triangle are filled

    Area = $\frac{3}{4} \times \frac{LH}{2}$

    Perimeter = $3 \times 3 \times \frac{L}{2} = \frac{3}{2} \times 3L$

The Sierpinski Gasket is a Fractal

- Consider the filled area (blue) and the perimeter (the length of all the lines around the filled triangles)



  - **Level 2:** 3 quarters of the filled triangles at level 1 are filled

    $$\text{Area} = \frac{3}{4} \times \frac{3}{4} \times \frac{LH}{2} = \left(\frac{3}{4}\right)^2 \frac{LH}{2}$$

    $$\text{Perimeter} = 9 \times 3 \times \frac{L}{4} = \left(\frac{3}{2}\right)^2 3L$$

  - **Level n:**

    $$\text{Area} = \left(\frac{3}{4}\right)^n \frac{LH}{2}$$

    $$\text{Perimeter} = \left(\frac{3}{2}\right)^n 3L$$

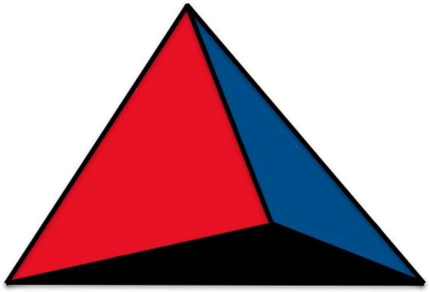
$\therefore$ As $n \to \infty$,
Area $\to 0$
Perimeter $\to \infty$

The Sierpinski Gasket is a Fractal

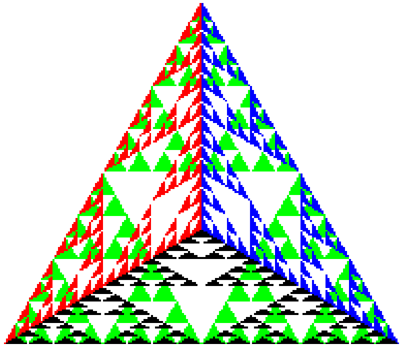
- As we continue subdividing
  - the area goes to zero
  - but the perimeter goes to infinity

A 3D Sierpinski Gasket

- We can easily extend the previous 2D Sierpinski triangle concept to 3D by defining a **tetrahedron** with four triangular faces.



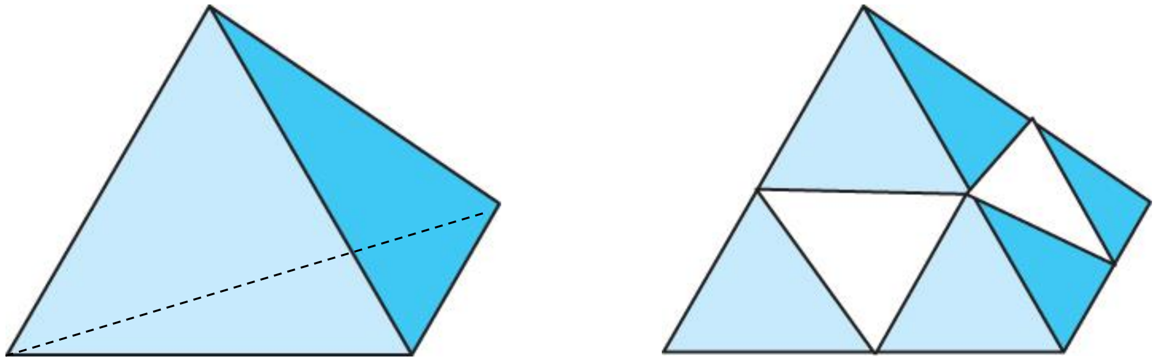
- We then divide up each face separately and draw each of the four faces using a different color.



"Interactive Computer Graphics – A Top-Down Approach with Shader-Based OpenGL" by Edward Angel and Dave Shreiner, 6th Ed, 2012, Section 2.10

A 3D Sierpinski Gasket (cont.)

- We can subdivide each of the four faces into triangles

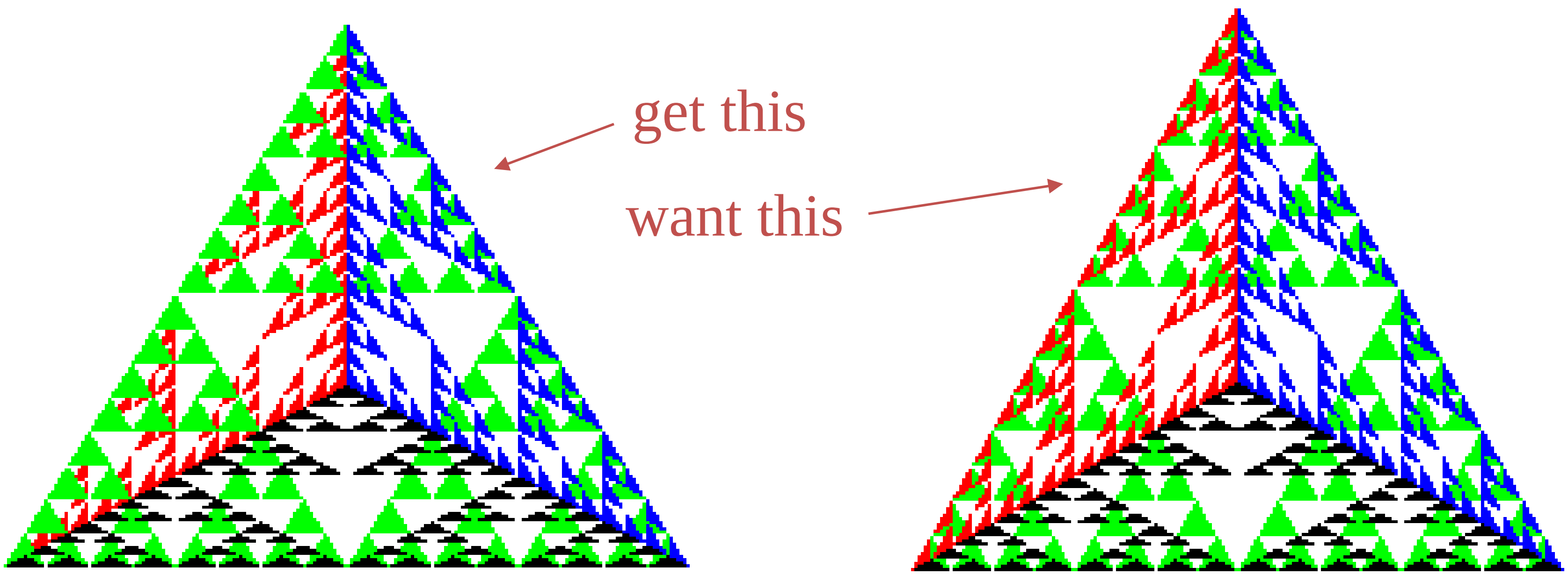


- It appears as if we remove a solid tetrahedron from the centre, leaving four smaller tetrahedra

A 3D Sierpinski Gasket (cont.)

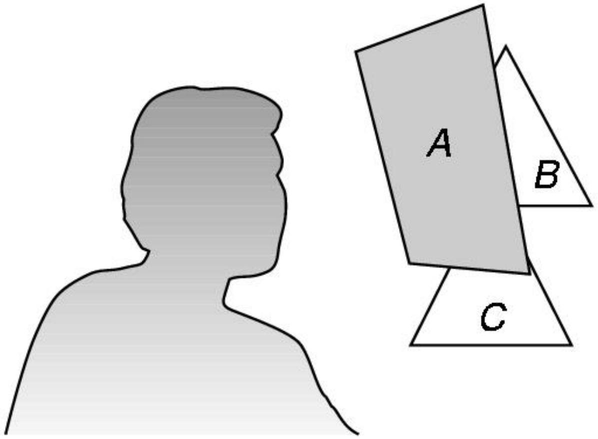
The result below is almost correct…

- Because the triangles are drawn in the order they are specified in the program, the front triangles are not always rendered in front of triangles behind them.



Hidden-Surface Removal

- We want to see only those surfaces that are in front of other surfaces
- OpenGL uses a *hidden-surface removal* method called the z-buffer algorithm, which saves the depth information of fragments as they are rendered so that only the front fragments appear in the image.



Hidden-Surface Removal

- Can hidden surface removal be done at vertex shader?

No

- It involves primitives, which are not formed at vertex processor.

The z-buffer Algorithm

The z-buffer algorithm

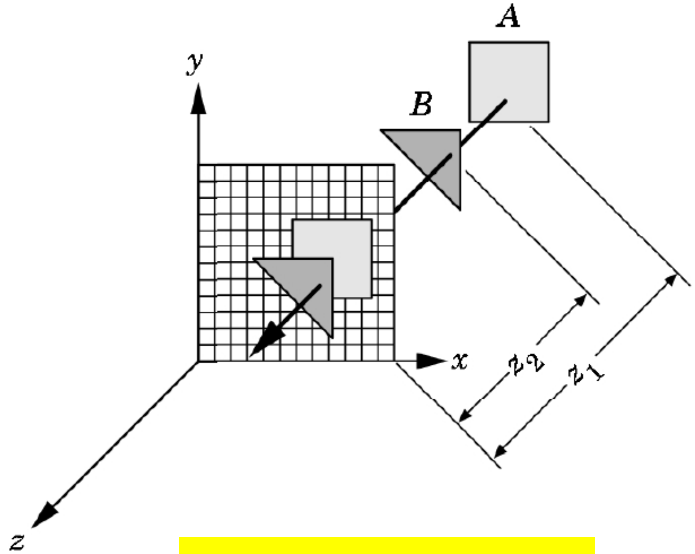
- is the most widely-used **hidden-surface-removal algorithm**
- has the advantages of being easy to implement, in either hardware or software
- is compatible with the pipeline architectures, where the algorithm can be executed at the speed at which fragments are passed through the pipeline

- The algorithm works in the **image space** and determines the visibility of each surface for each pixel position.
  - Paint pixel with color of **closest** object to the view plane

The z-buffer algorithm – How It works

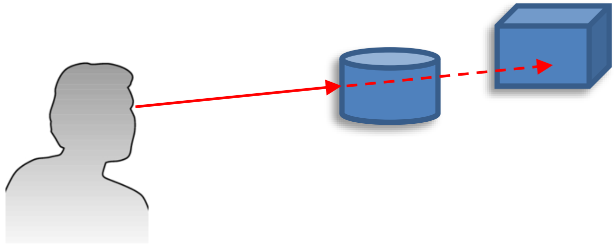
Suppose that we are in the process of rasterizing one of the two polygons shown on the right:

- We can compute a colour for each point on object (say point **p**)
- We must check whether **p** is visible. It is visible if it is the closest point to the camera
  - If we are rasterizing polygon B, then its shade will appear at that pixel on the screen, as $z_2 < z_1$
  - If we are rasterizing polygon A, then its shade won't appear at that pixel on the screen



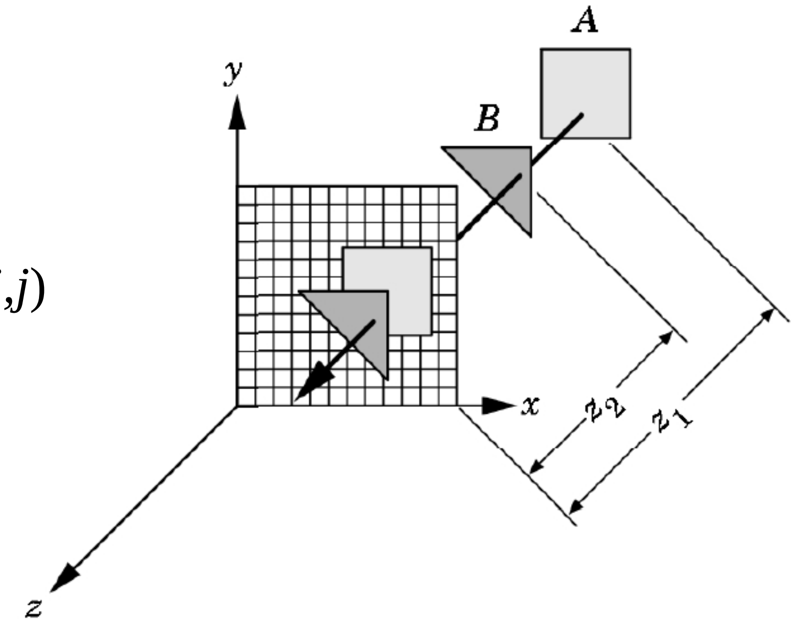
Note that **p** is a point on an object in the object space

Find depth (z) of every polygon at each pixel

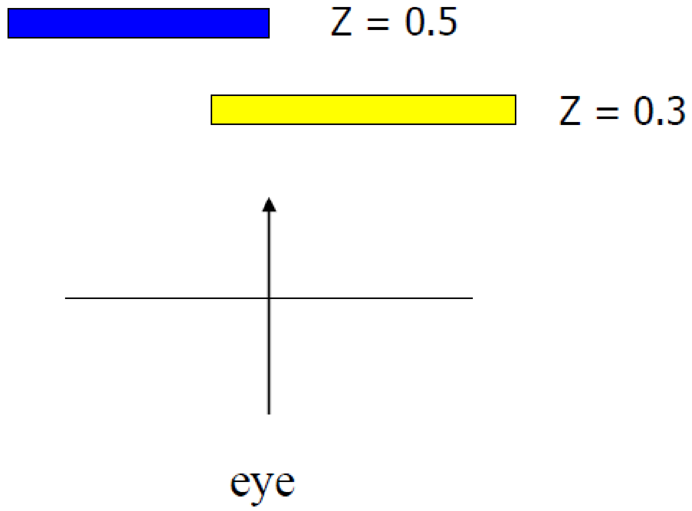


The z-buffer algorithm – How It works

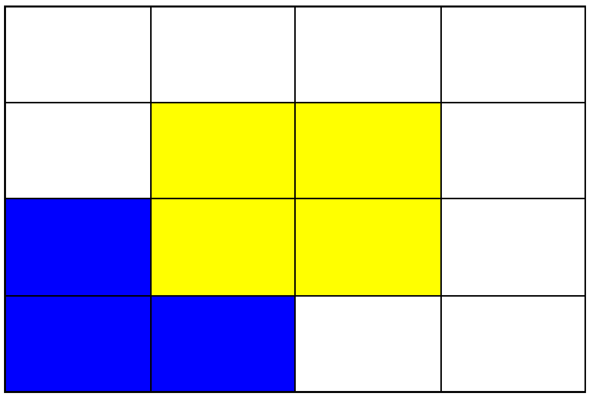
```
for each pixel (i,j) do
    Z-buffer[i,j] ← FAR
    Framebuffer[i,j] ← <background color>
end for
for each polygon do
    for each pixel (i,j) occupied by the polygon do
        Compute depth z and shade s of that polygon at (i,j)
        if z < Z-buffer[i,j] then
            Z-buffer[i,j] ← z
            Framebuffer[i,j] ← s
        end if
    end for
end for
```



The z-buffer algorithm – Illustration



Top View



Desired Final Image

Slide content credits:
Emmanuel Agu, WPI, CS 543

The z-buffer algorithm – Illustration

Step 1: Initialize the depth buffer, such that all values are set to maximum depth (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

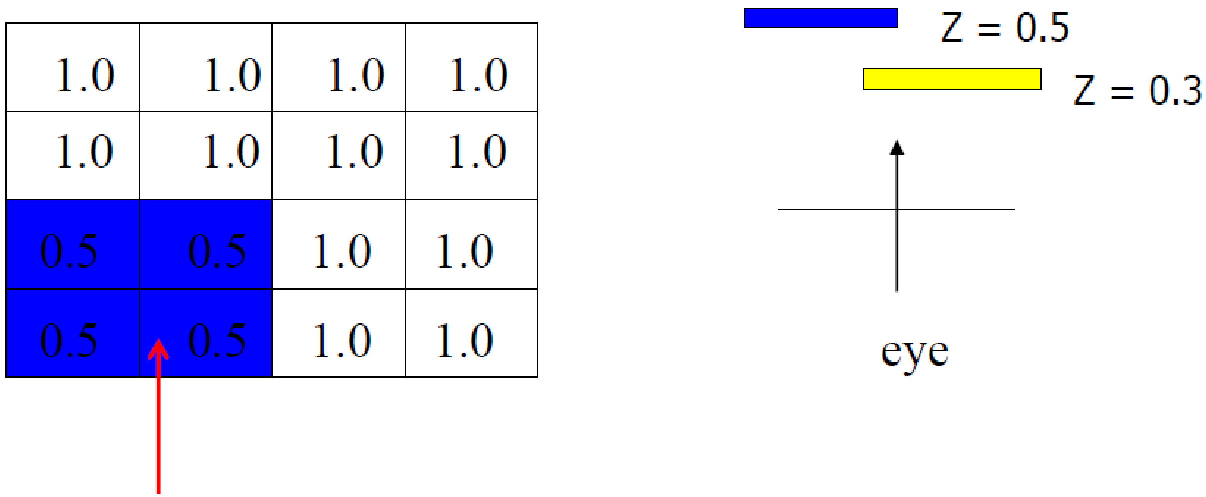
Largest possible z values is 1.0

Z-buffer

Slide content credits:
Emmanuel Agu, WPI, CS 543

The z-buffer algorithm – Illustration

Step 2: Process each polygon in a scene, one at a time. We start with the blue polygon (order does not matter)

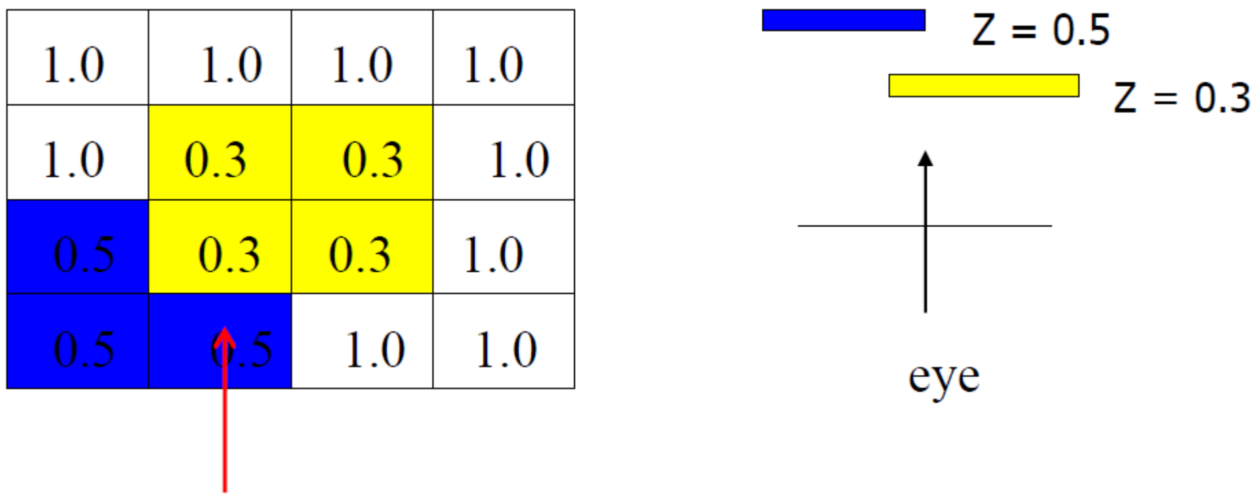


- For each projected (x,y) pixel position corresponding to the blue polygon, calculate the depth z
- If z< *Z-buffer*(x,y), set *Z-buffer*(x,y)=z and update the color for the pixels (corresponding to blue polygon) in frame buffer

Slide content credits:
Emmanuel Agu, WPI, CS 543

The z-buffer algorithm – Illustration

Step 3: We then draw the yellow polygon (order does not matter)



- For each projected (x,y) pixel position corresponding to the yellow polygon, calculate the depth z
- If z< *Z-buffer*(x,y), set *Z-buffer*(x,y)=z and update the color for the pixels (corresponding to yellow polygon) in frame buffer

Slide content credits:
Emmanuel Agu, WPI, CS 543

The z-buffer algorithm – Illustration

- The algorithm uses an extra buffer, the **z-buffer**, to store depth information as geometry travels down the pipeline
- In OpenGL, the z-buffer must be:

Select window with a depth buffer

  - **requested in *main*()**

    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)

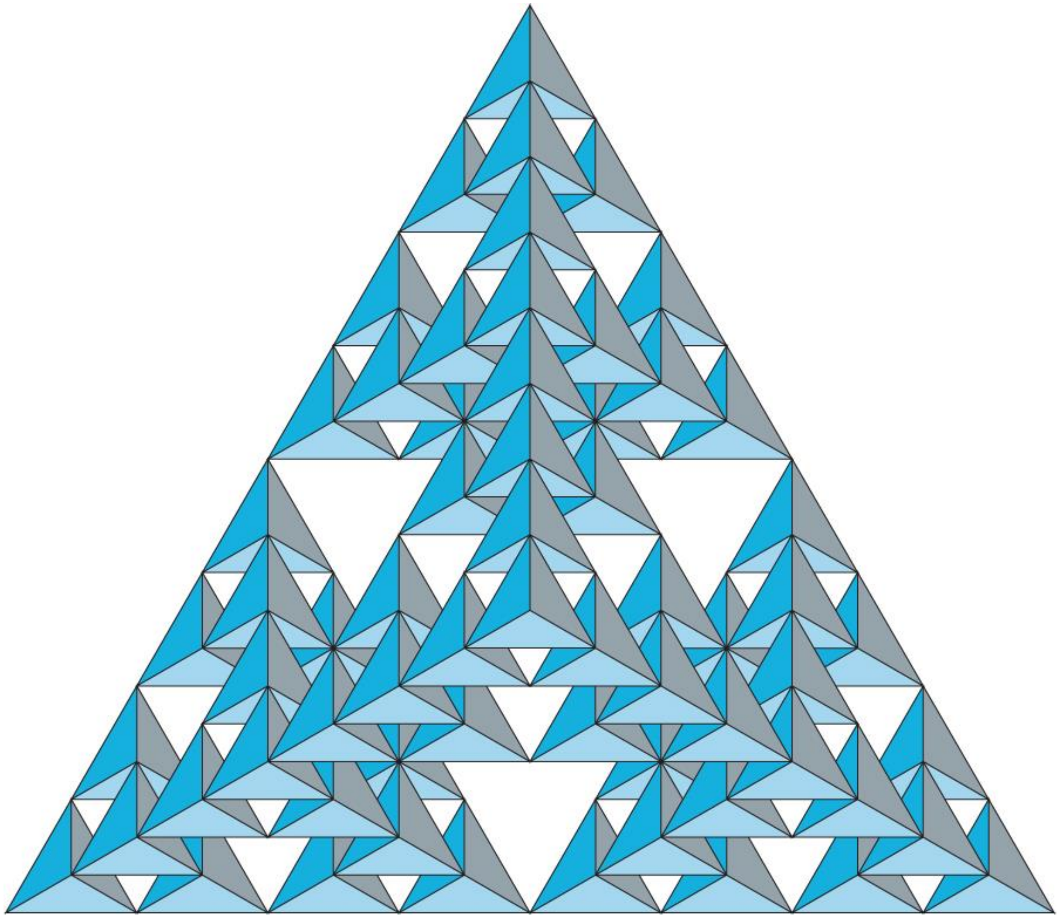
  - **enabled in *init*()**

    glEnable(GL_DEPTH_TEST)

  - **cleared in the *display* callback**

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

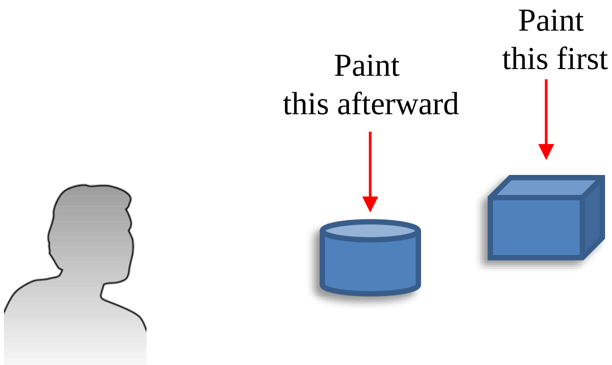
Sierpinski Gasket – After Hidden Surface Removal



The Painter's Algorithm

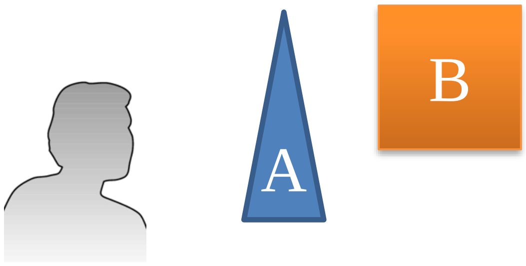
- Although **image-space methods** are dominant in hardware due to the efficiency and ease of implementation of the z-buffer algorithm, often **object-space methods** are used in combination.
- **Object-space algorithms** attempt to order the surfaces of the objects in the scene such that rendering surfaces in a particular order provides the correct image

- The painter's algorithm is an object-space approach to **hidden surface removal**. It is one of the simplest solutions to the visibility problem in 3D computer graphics

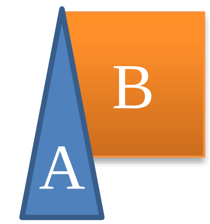


The Painter's Algorithm (cont.)

- Similar to painter layers oil paint
  - The name refers to the technique (employed by many painters) of painting distant parts of a scene before parts which are closer, thereby covering some areas of the distant parts.
- Render polygons farthest to nearest
  - The algorithm sorts all the polygons in a scene by their depths. Polygons are painted from the furthest to the closest depth.
  - Because of how the algorithm works, it is also known as a **depth-sort algorithm**.



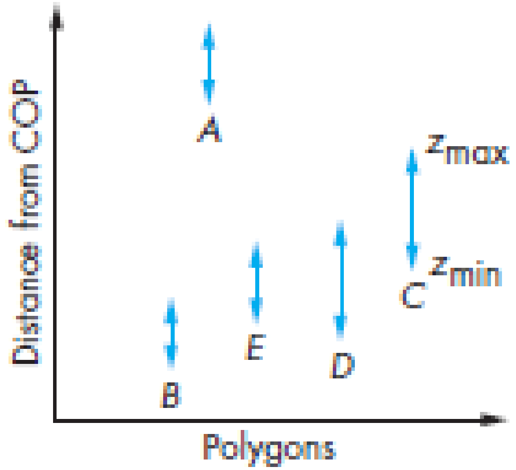
Viewer sees B behind A



Render B then A

The Painter's Algorithm (cont.)

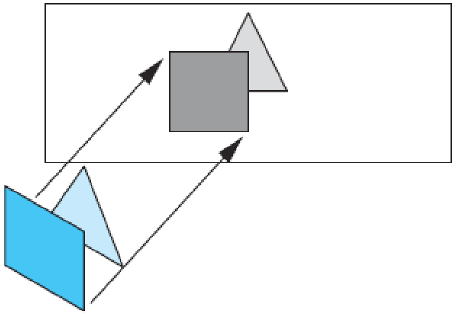
- Suppose that we have the *z*-extent of 5 polygons as shown on the right:
  - Polygon *A* can be painted first;
  - However, we can't determine the order for painting the other polygons
  - The algorithm needs to run a number of increasingly more difficult tests in order to find the painting ordering.

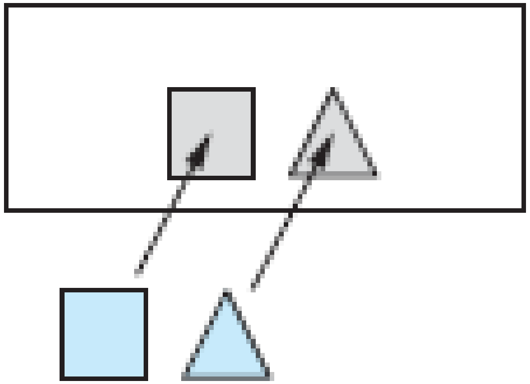


(COP = centre of projection)

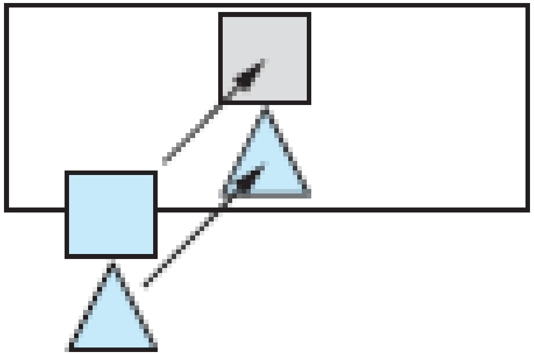
The Painter's Algorithm (cont.)

- The simplest test is to check the *x*- and *y*-extents of the polygons:
  - If either of the *x*- or the *y*-extents do not overlap, neither polygon can obscure the other. So they can be painted in any order.
- If the above test fails, can still determine the order of painting by testing if one polygon lies completely on one side of the other.





Test for overlap in the *x*-extents

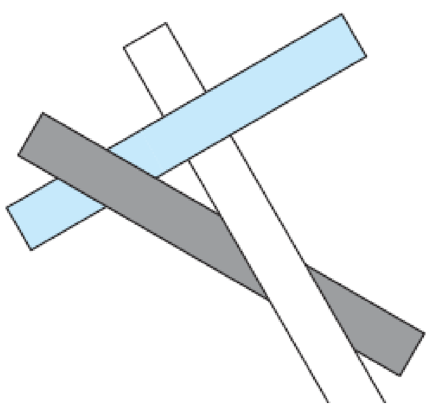


Test for overlap in the *y*-extents

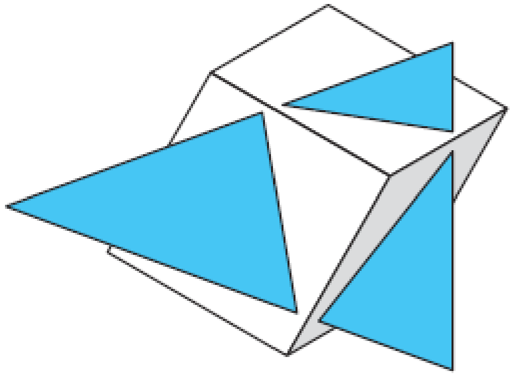
The Painter's Algorithm (cont.)

The algorithm fails in some cases, including

- Polygons that pierce (intersect) each other
- Polygons that form a cycle of depth overlap



Cyclic overlap



Piercing polygons

How to sort the polygons for rendering?

- Split the polygons to get the ordering→ complex process

Double Buffering

- In an earlier lecture, we saw that using a *uniform* variable opens the door to animation:
  - We can call glUniform in the *display* callback
  - We can then force a redraw through glutPostRedisplay()

Double buffering is particularly useful when the application deals with 3D objects

Double Buffering (cont.)

- To animate a scene smoothly, we need to prevent a partially redrawn frame buffer from being displayed.

- A way to prevent the above issue from happening is to use **double buffering** – i.e., we request two buffers:
  - While drawing is performed on the back buffer, the front buffer is being displayed
  - Swap buffers after the update on the back buffer is finished

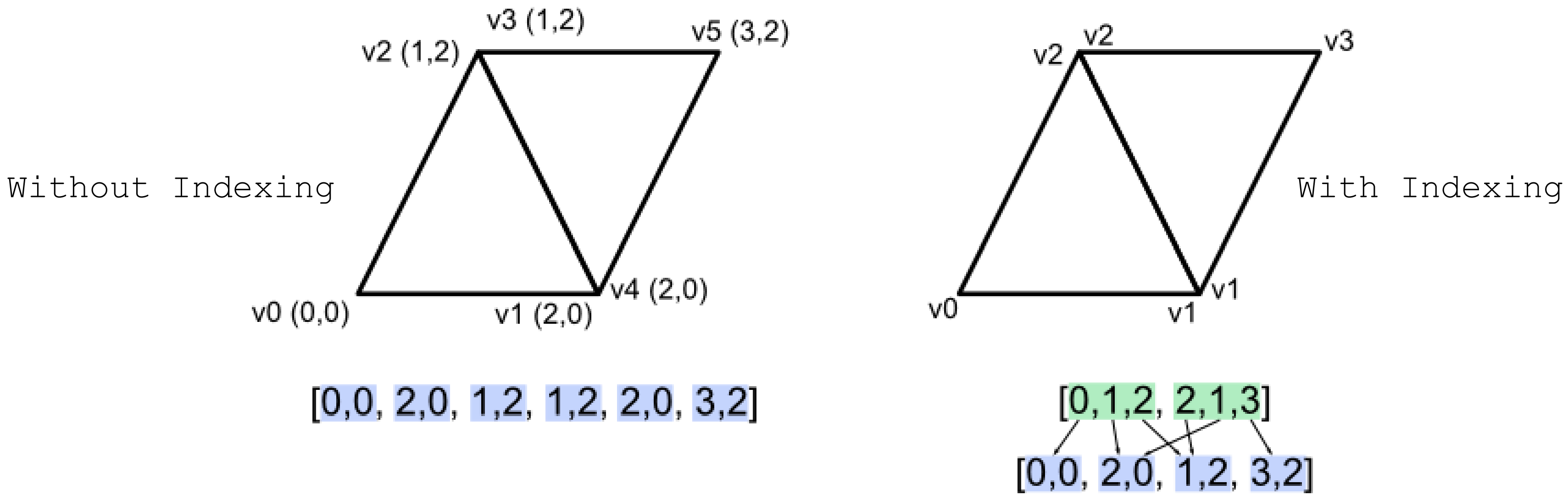
Adding Double Buffering

- Request a double buffer
  - glutInitDisplayMode(GLUT_DOUBLE)
- Swap buffers

```
void mydisplay()
{
        glClear(……);
        glDrawArrays(…);
        glutSwapBuffers();
}
```

Element Buffers

- Complex 3D models have thousands of triangles
- We can re-use the vertices while defining triangles (for efficiency)
- GL_ELEMENT_ARRAY_BUFFER is used for this



http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-9-vbo-indexing/

Element Buffers

- Lab 5, q4aIndex.cpp draws a cube by specifying only 8 vertices
- 8 vertices -> 6 squares -> 12 triangles

OpenGL Element Array Buffers https://www.youtube.com/watch?v=ZSsJbJ2hviI

3D Model File Format

- 3D model file formats follow a similar convention i.e.
  - A list of vertices as floats (x, y, z)
  - A list of elements as integers specifying which vertices connect to form a triangle

- Sometimes the vertex normals are also provided as floats

The PLY Format

```
ply
format ascii 1.0
comment zipper output
element vertex 28980
property float x
property float y
property float z
element face 56207
property list uchar int vertex_indices
end_header
44968.501119   -43787.362630   83846.209031
46090.448700   -44321.044193   81938.091386
39593.486637   -49592.734508   86290.454426
46243.264772   -42401.503638   83047.168327
45096.493171   -42006.610299   84810.068897
.
.
3 24028 24504 24620
3 20691 20688 20755
3 19350 19371 19384
3 942 377 1297
```

If you open a *.ply file in wordpad You see this

Some 3D File Extensions

- PLY files have *.ply extension
- VRML files have a *.wrl extension
- 3D Studio files have a *.3ds extension
- Blender files have a *.blend extension
- Object files have a *.obj extension
- DirectX files have a *.x extension

DirectX format

```
xof 0303txt 0032
Frame Root {
 FrameTransformMatrix {
   1.000000, 0.000000, 0.000000, 0.000000,
   0.000000,-0.000000, 1.000000, 0.000000,
   0.000000, 1.000000, 0.000000, 0.000000,
   0.000000, 0.000000, 0.000000, 1.000000;;
 }
 Frame Grid {
  FrameTransformMatrix {
    1.000000, 0.000000, 0.000000, 0.000000,
    0.000000, 1.000000, 0.000000, 0.000000,
    0.000000, 0.000000, 1.000000, 0.000000,
    0.000000, 0.000000, 0.000000, 1.000000;;
  }
  Mesh { // Grid mesh
   324;
   -0.111111;-1.000000; 0.000000;,
    0.111111;-1.000000; 0.000000;,
    0.111111;-0.777778; 0.000000;,
   -0.111111;-0.777778; 0.000000;,
.
.
81;
   4;3,2,1,0;,
   4;7,6,5,4;,
   4;11,10,9,8;,
   4;15,14,13,12;,
```

DirectX format is more complicated.

It has point, polygons, as well as textures and animations

Further Reading

"Interactive Computer Graphics – A Top-Down Approach with Shader-Based OpenGL" by Edward Angel and Dave Shreiner, 6th Ed, 2012

- Sec. 2.10.3 *Hidden-Surface Removal* (pages 96-98)
- Sec. 4.8 *Hidden-Surface Removal* (pages 239-241)
- Sec. 6.11.5 *The Z-Buffer Algorithm* (pages 335-338)
- Sec. 6.11.7 *Depth Sort and the Painter's Algorithm* (pages 340-342)

Computer Graphics using OpenGL, 3rd edition, Hearn and Kelly, Chapter 9