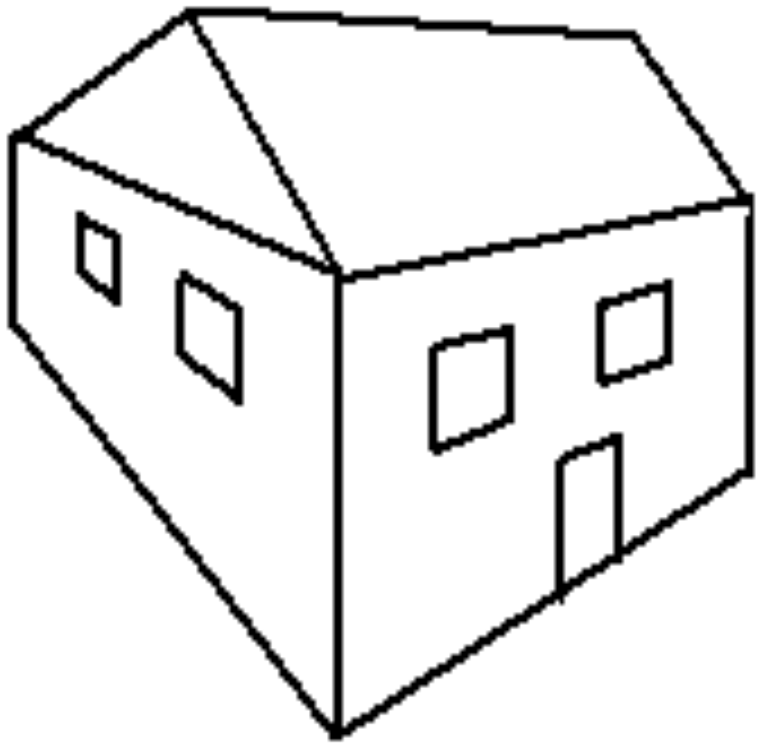
Computer Graphics

Lecturer: Simon Mcloughlin

Lecture 7

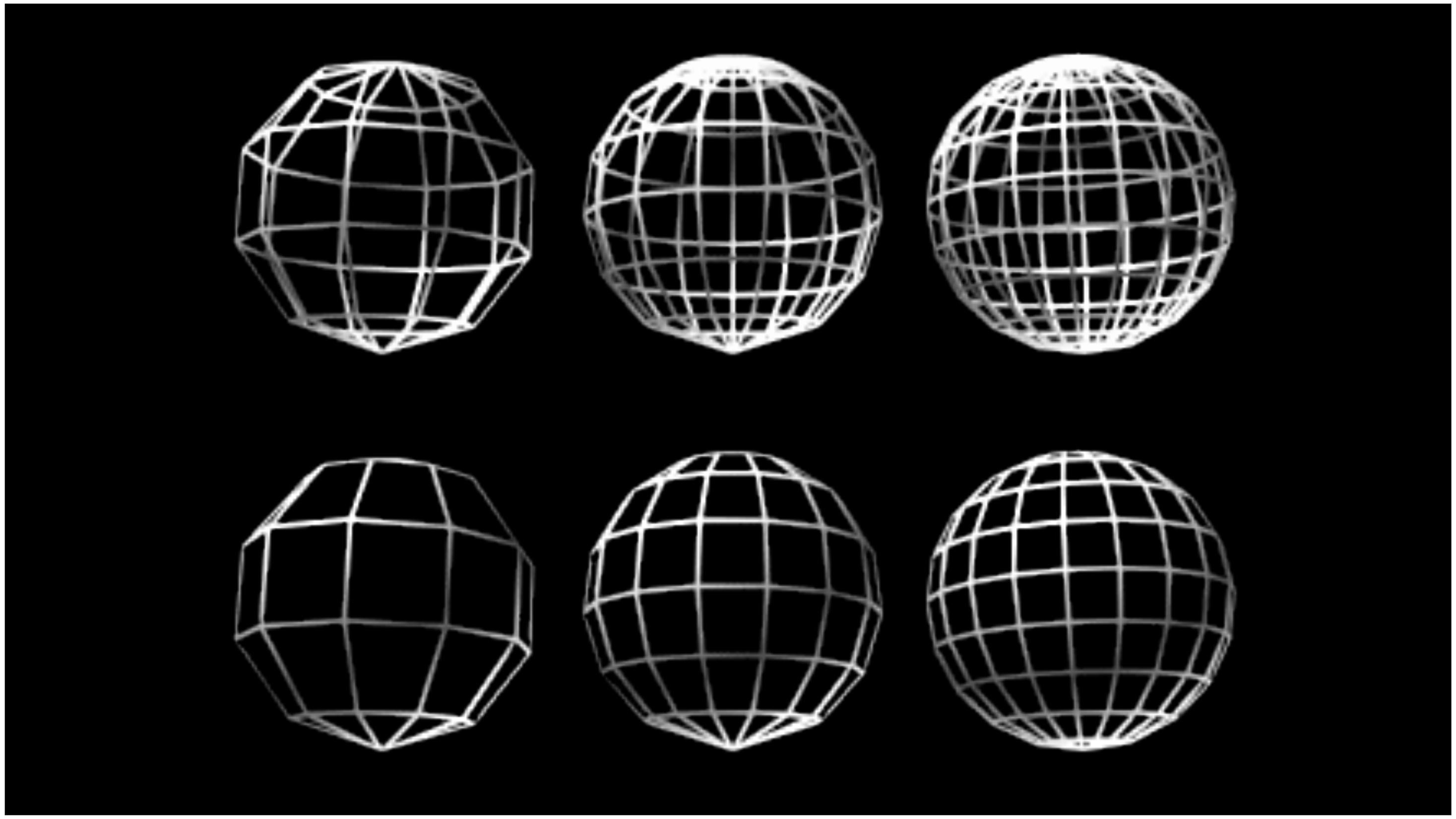


Review and Overview

- A major consideration in the generation of realistic graphics displays of 3-d objects is **determining which parts/components of these objects are visible** for a particular viewing direction
- Many approaches and methods exist to go about solving this problem
- Collectively they are known as **“visible surface detection”** or “hidden surface removal” algorithms
- VSD algorithms are broadly classified into one of two classes, **object space methods or image space methods**
- Object space methods compare objects and object components (polygons etc.) to each other **in the scene** to determine which surfaces are visible
- Image space methods determine what is visible **for each pixel on the display or projection plane**
- There are four reasons why object faces should not be rendered, can you think of them?

Visible Surface Detection

- Visible Surface Detection (VSD) is the process by which the rendering system figures out which polygons are visible from the current viewpoint and which ones are not.

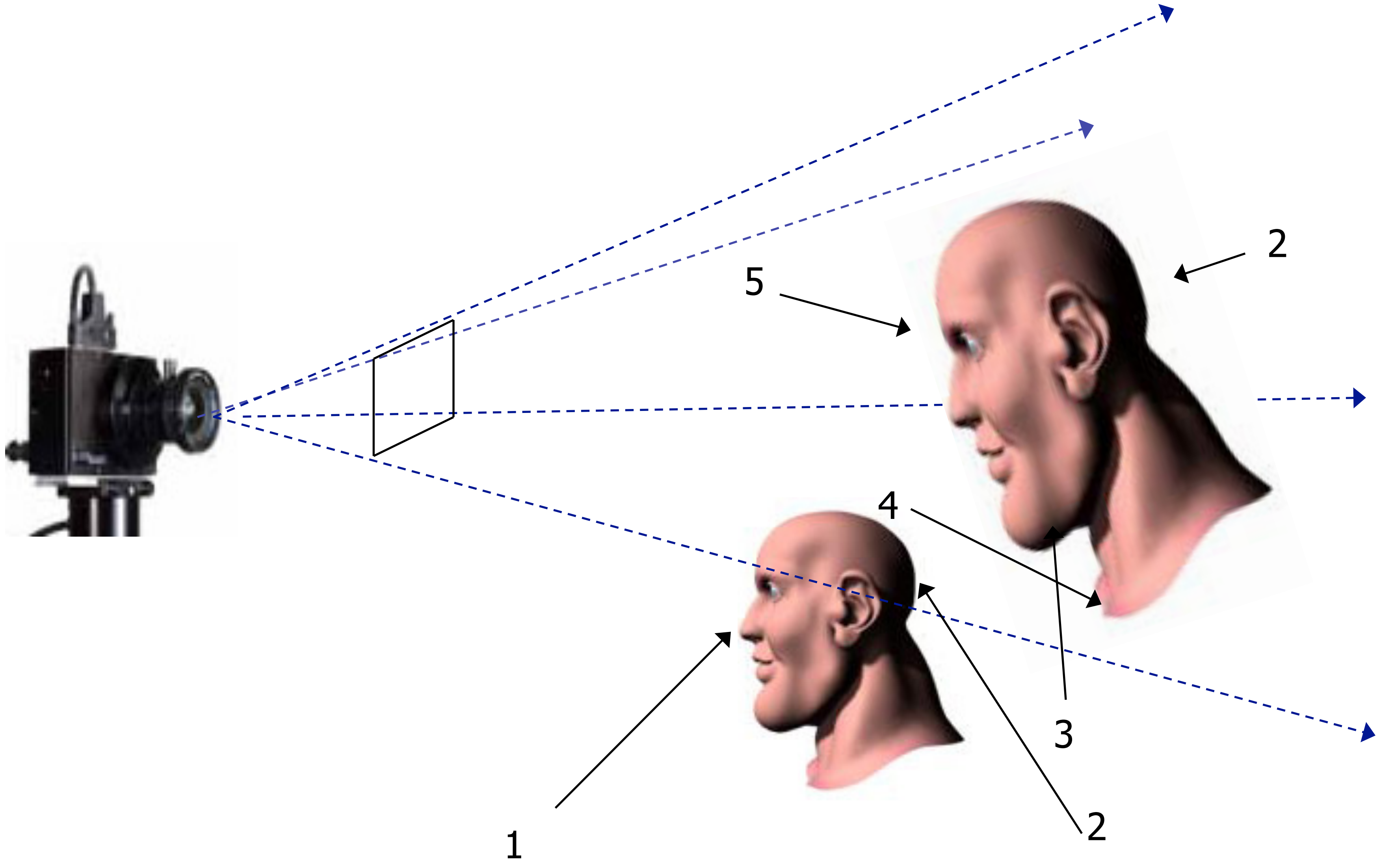


Hidden Surface Removal

- Polygons may not be visible for four reasons:
  1. They are at the ‘back’ of the object
  2. Not in the viewing volume
  3. They are obscured by another object
  4. They are obscured by another part of the same object
- So if we consider a scene as comprising of a set of polygons, P, then, with respect to a particular viewing setup, the system must examine each polygon in turn.

- For every polygon it is either
  1. Not in the viewing volume=> Not visible
  2. At the back of the object => Not visible
  3. Obscured by another part of the same object => Not Visible
  4. Obscured by another object => Not Visible
  5. None of the above => Visible

Visible vs. Not Visible Polygons



Hidden Surface Removal

- We should note that we can often be dealing with a very large set of polygons. For example the face model previously has several thousand polygons.
- This means a lot of computation.
- Also note that every time the viewpoint or the scene geometry changes we have to recompute visibility for every polygon. In an interactive dynamic system (e.g. a game) this might have to be done 40 times per second.
- HSR has always been the bottleneck of real-time 3D graphics.
- Only in the last few years, with the widespread availability of specialized 3D hardware graphics cards, has this bottleneck been overcome.

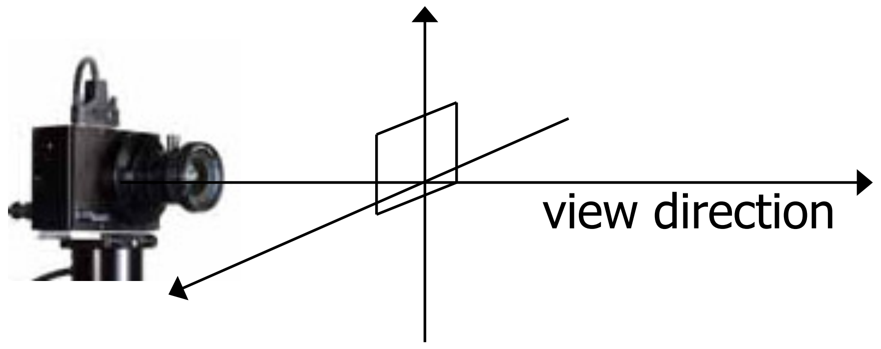
Visible vs Not Visible Polygons

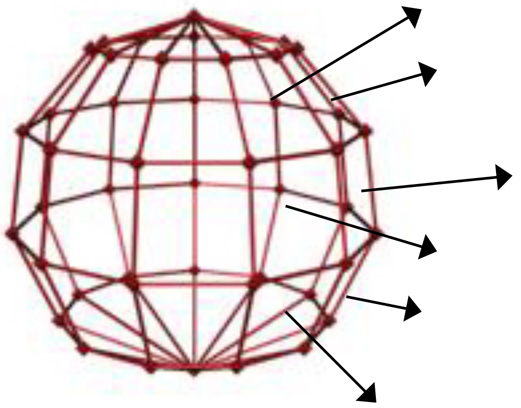
- If we call the entire set of polygons P then Hidden Surface Removal (HSR) is the process of determining a subset of P called V whereby the elements of V are the subset of visible polygons.
- Let us also denote the following:
  - F is the set of polygons that fall into category 2 (i.e. outside) $F \subset P$
  - B is the set of polygons that fall into category 1 (i.e. back faces). $B \subset P$
  - O is the set of polygons that fall into category 3 and 4 (i.e. obscured) $O \subset P$
  - Then we have that

$$V = ((P \setminus F) \setminus B) \setminus O$$

Hidden Surface Removal with Backfaces

- Suppose we want to determine backfaces? These are polygons that are at the back of an object with respect to the current viewing direction.
- This turns out to be pretty easy to do.
- Polygons at the back will be facing in the same general direction as the view direction.

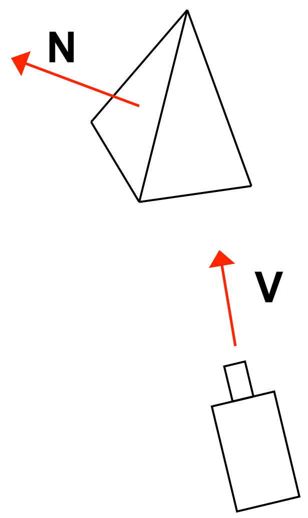




- In other words their surface normals will point the same way.
- Polygons at the front will have surface normals pointing in the opposite direction to the view direction.

VSD algorithms - Back Face Detection

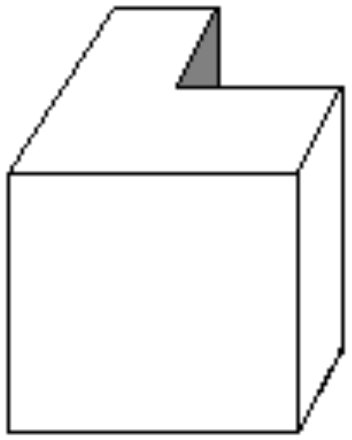
- A fast and simple object-space method for identifying the **back faces of objects** is based on the **surface normal**
- Consider the following diagram



- Here V is the viewing direction and N is the surface normal for a polygon
- If $\mathbf{V}.\mathbf{N} > 0$ then the polygon with surface normal N is a back face and cannot be seen
- Remember V.N gives the cosine of the angle between the two vectors

VSD algorithms - Back Face Detection

- If our scene consists of **non-overlapping convex polyhedron** then the backface detection algorithm can be used to **detect all hidden surfaces**
- For other objects such as **concave polyhedrons** more tests need to be carried out since some **front surfaces may be obscured by others**



Backface Removal

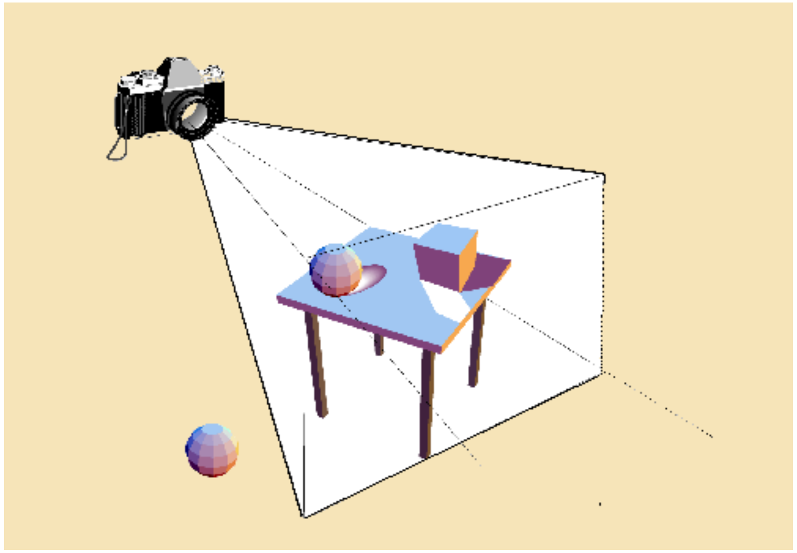
- Carry out this test on every polygon, every one that gives a positive value, is an element of B.
- We can therefore eliminate those polygons i.e. calculate a new set P' where P' is P difference (minus) B.
- So where we used to have 20,000 potentially visible polygons we now have maybe 10,000 potentially visible polygons.
- Remember the dot product just involves 3 multiplications and 2 additions so its very fast to carry out (as long as all the polygon normals have been pre-computed).
- In openGL face culling can be enabled by

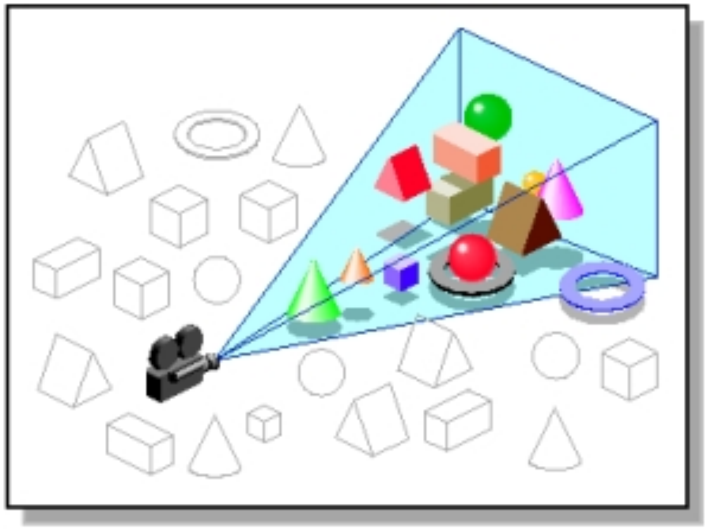
  glEnable(GL.GL_CULL_FACE);
- To cull back faces then you can call glCullFace(…)

  glCullFace(GL.GL_BACK);

Clipping

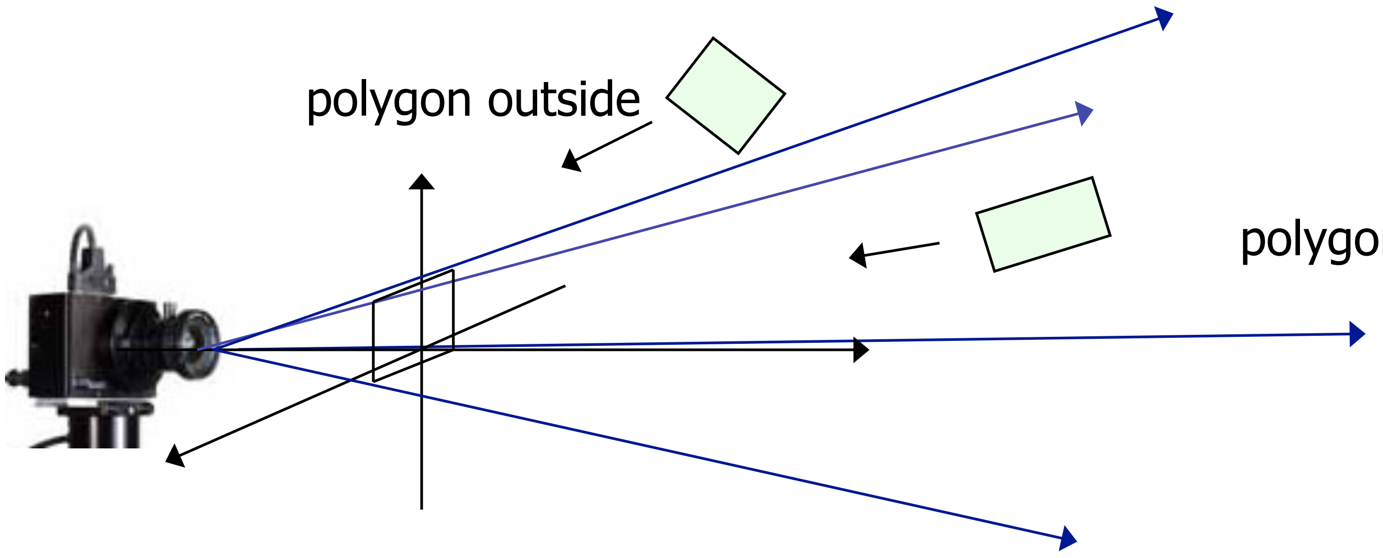
- Clipping is the process by which those polygons outside the viewing frustum are removed from consideration i.e. clipping is used to determine the set F.





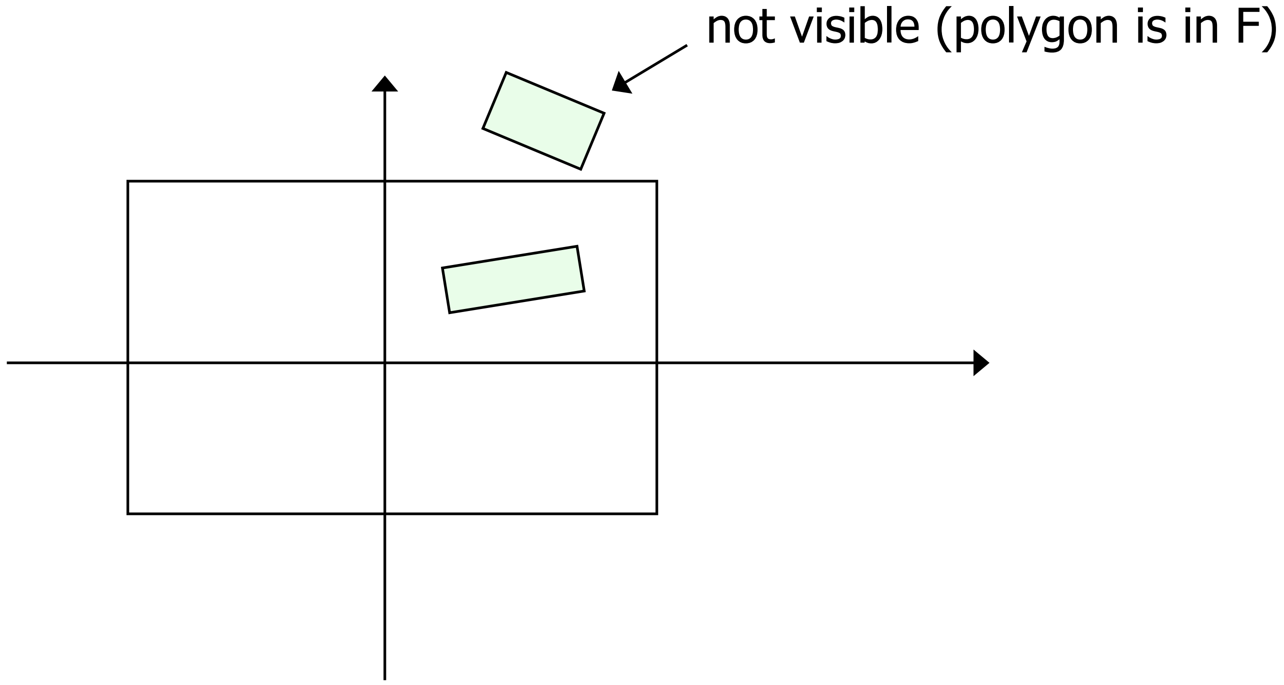
Clipping

- There are a number of methods for going about dealing with this.
- The most straightforward one is to carry out the clipping in 2D space after projection has been done.
- A more efficient approach might do some tests in world space to determine which objects are inside the view volume



Clipping

- When these polygons are projected onto the projection plane one of them will land inside the viewing window and one of them will land outside. As shown below.

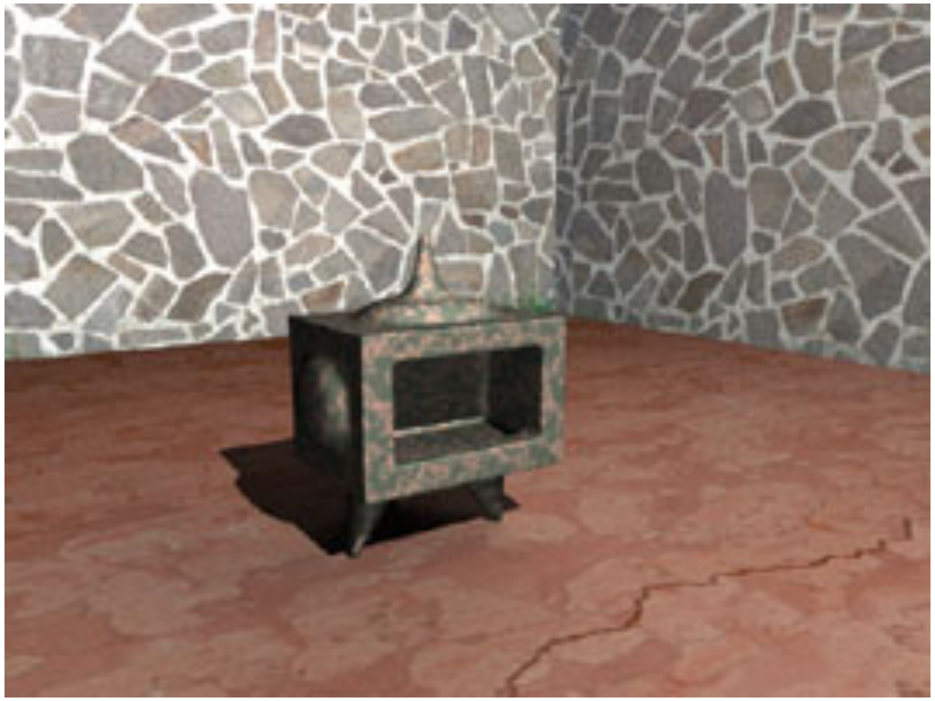


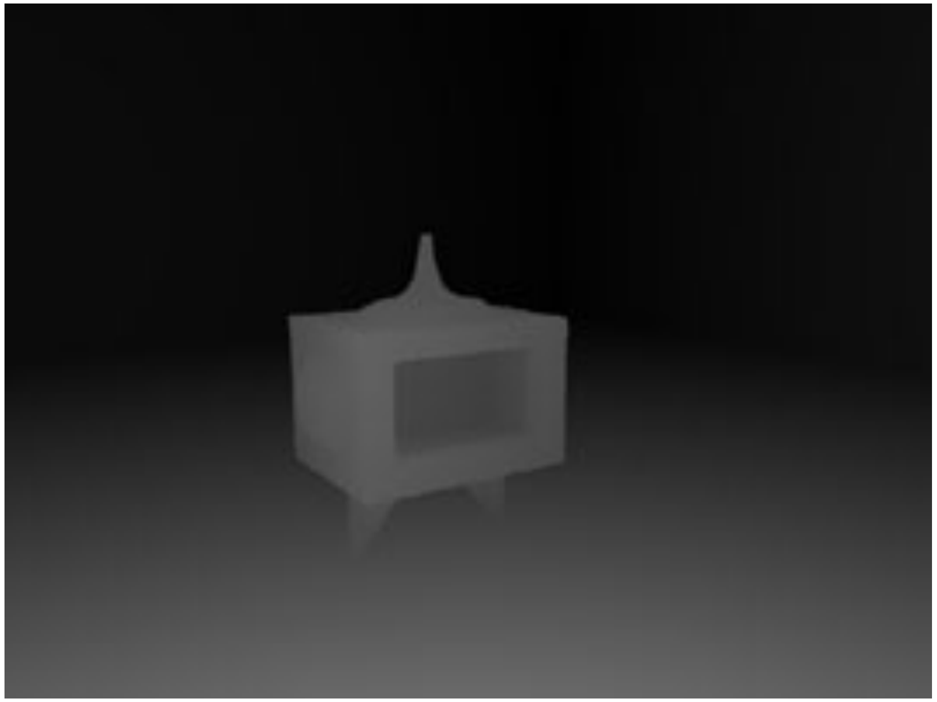
- There are various 2D clipping algorithms such as Cohen-Sutherland that can be used to quickly determine what's outside the rectangle and what's inside.
- This is the easy bit so it would be done last …….

VSD algorithms – Depth-Buffer Method

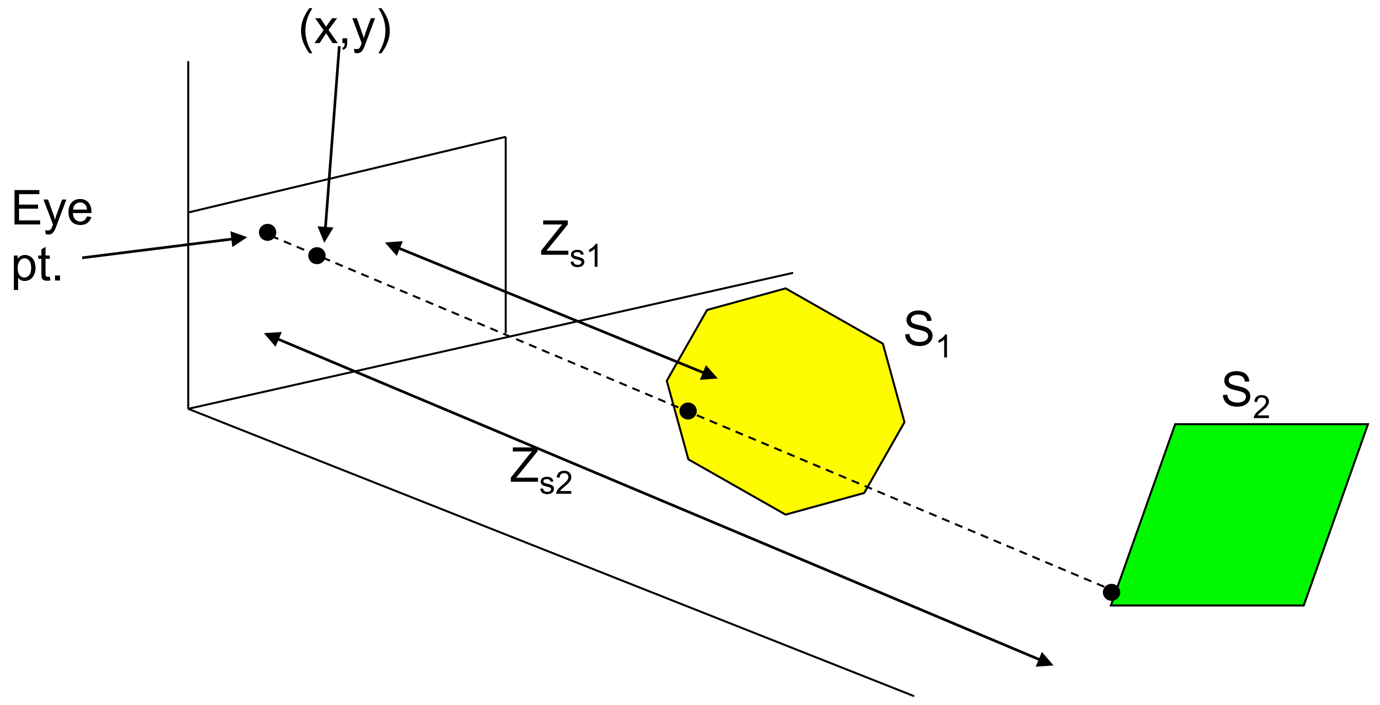
- A commonly used approach to determining visible surfaces in a scene with multiple concave/convex polygons is the **depth buffer** or **z-buffer** method (used in openGL)
- This technique is very easy to implement for scenes containing only polygon surfaces because depth information can be computed easily for polygons
- It operates by using a new area of memory called the **depth buffer** which contains the depth value of the **CLOSEST** scene point to the view plane
- The depth of **points on all surfaces** inside the viewing volume are **compared** to what is already in the depth buffer. If that scene point has the same view-plane (u,v) location but a **smaller depth** value then it's depth is recorded in the depth buffer for that location and it's **intensity in the frame buffer.**
- **Two buffers** required (big memory overhead). May paint the **same pixel several times** which may be costly
- Easy to understand and implement

VSD algorithms – Depth-Buffer Method


Colour buffer

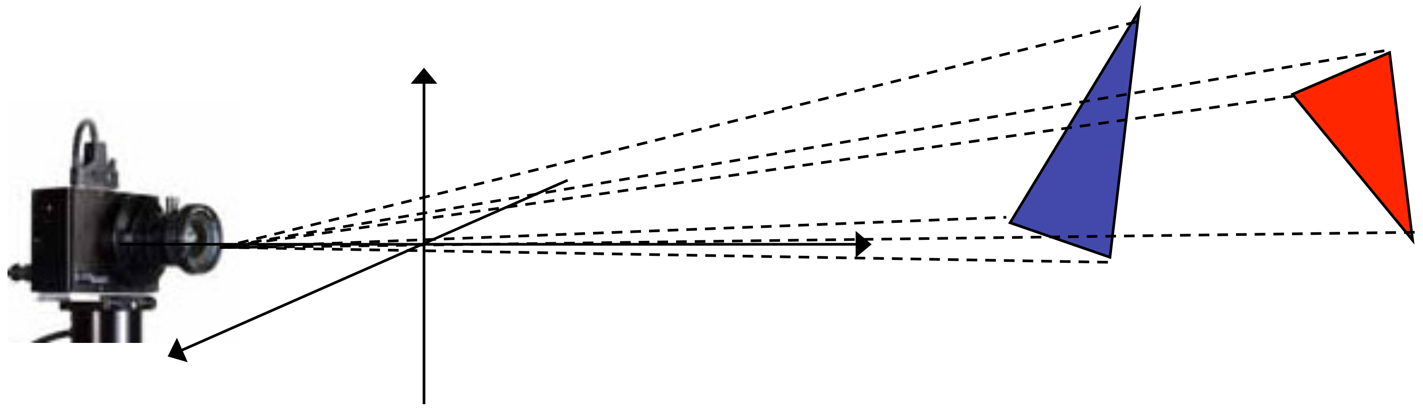

Depth buffer

VSD algorithms – Depth-Buffer Method

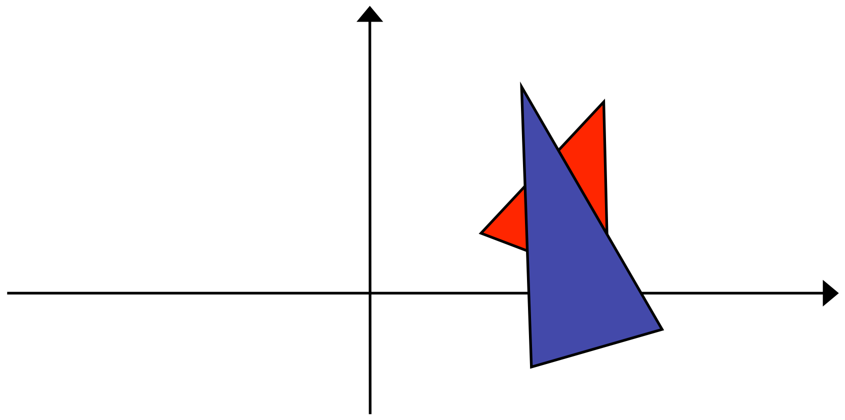


- For each surface in the scene it's **projection point** on the view plane is calculated and if the **z value for that point is smaller** than what is already in the z-buffer for particular **x,y location** then **replace** it with the **smaller z value**
- Z_{s1} will be stored in the depth buffer for the location x,y in example above

The Depth-Buffer Algorithm



- The diagram below might represent the projection of the two triangles above



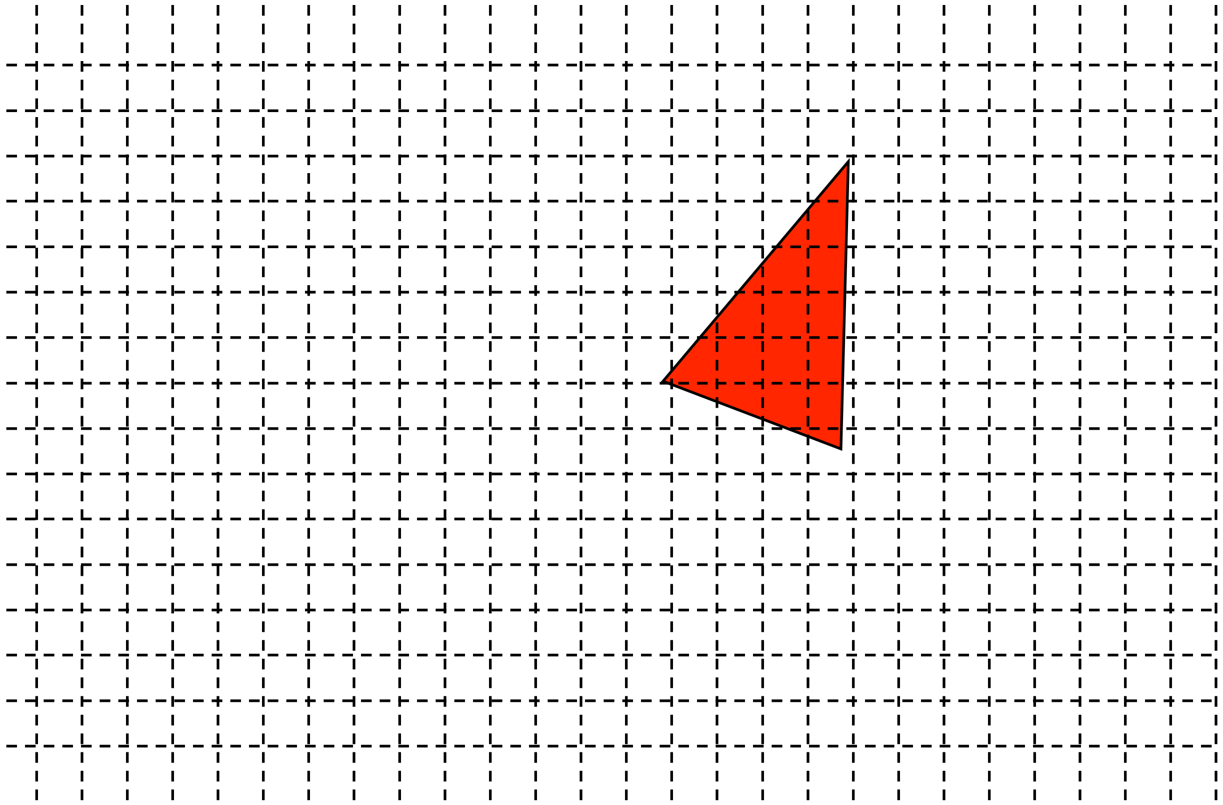
The Z-Buffer Algorithm

- I know the blue one is in front so I drew it correctly but how does the computer work this out?
- What is has to do is **compare the depths** of the triangles in order to determine which one is closest to the viewpoint.
- One way to try and solve this is to **depth-sort** the polygons before they are projected. **Project the ones that are furthest away first** and then if subsequent polygons land in the same place on the screen they will simply overwrite them. But because they are closer that’s fine.

- This approach is okay if we have relatively small numbers of polygons.
- You should know from doing sorting algorithms that sorting lists of things is a computationally expensive process. As the length of the polygon list increases so does the length of time needed to sort them - exponentially for some algorithms …
- Now remember that we will be dealing with an entire set P’ comprising of thousands of polygons. Not really feasible to carry out a full sort over 40 times per second.

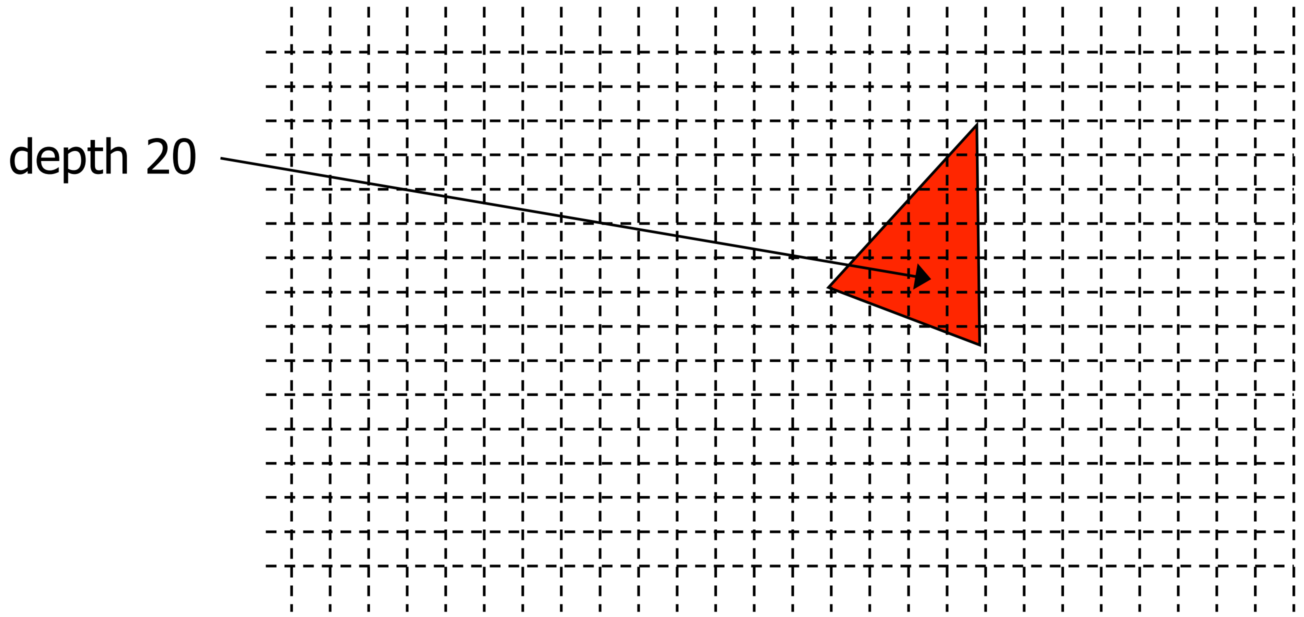
The Z-Buffer Algorithm

- We can improve the situation though by remembering that the polygons are being projected onto the screen which is a grid of pixels.



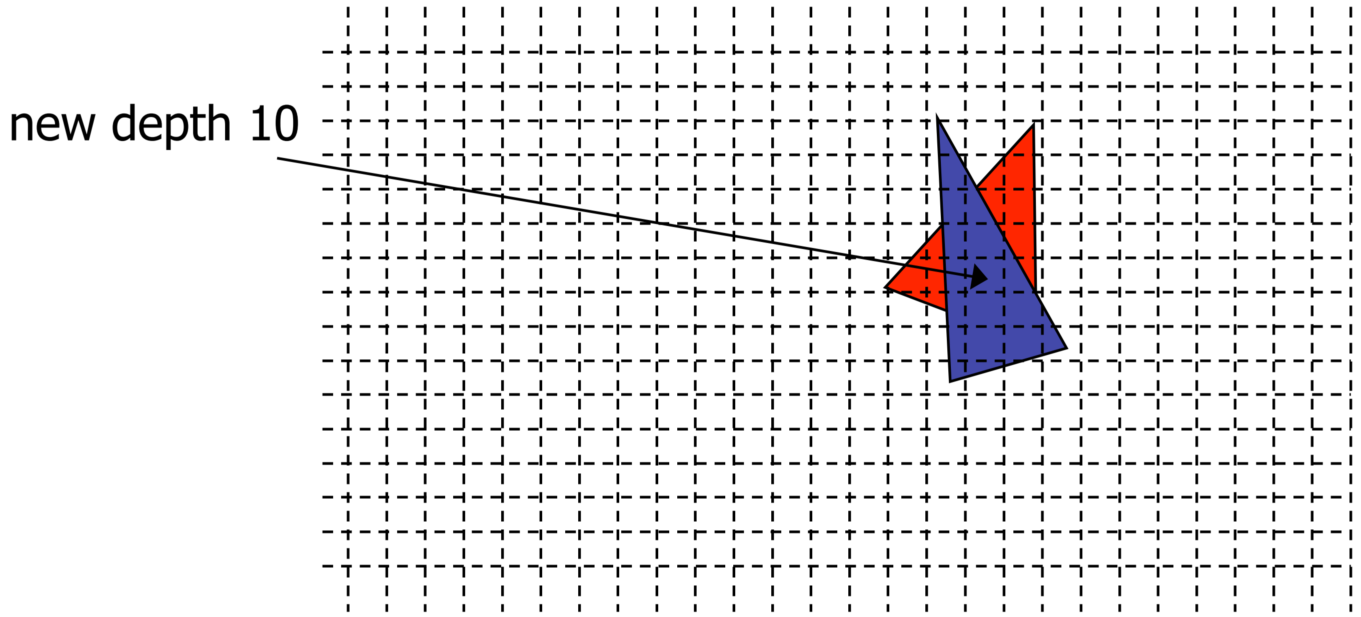
The Z-Buffer Algorithm

- After the first polygon has been projected every pixel is either empty or contains the red polygon.
- For each pixel that contains the red polygon we can associate a depth value that records the distance from the viewpoint to the polygon at that pixel.



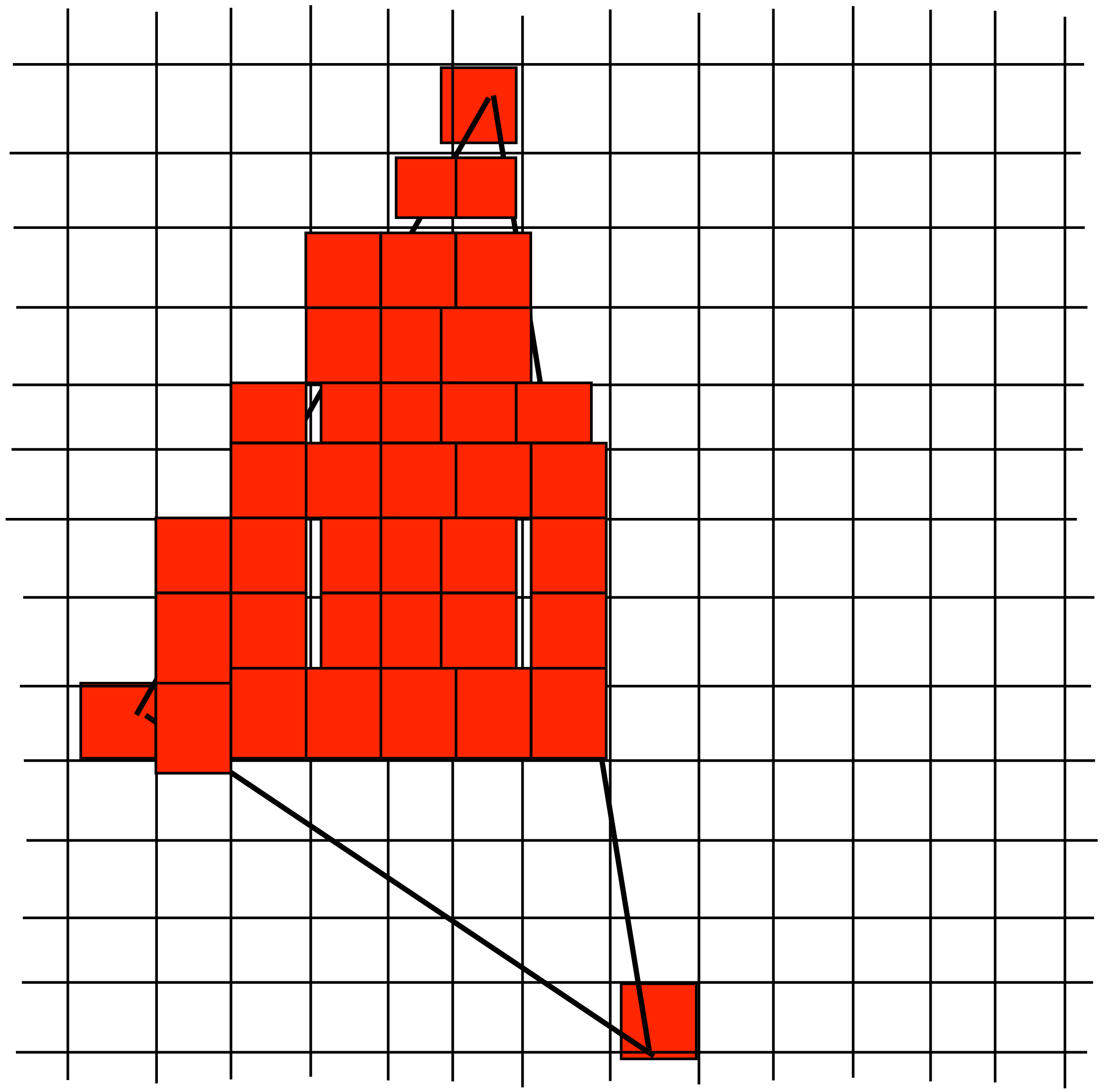
The Z-Buffer Algorithm

- As the second polygon is being projected check to see if any of its pixels overlap with something already there (i.e. pixel not empty)
- If there is an overlap check the depth value. If new one is behind ignore, otherwise overwrite and store new depth value for that pixel.

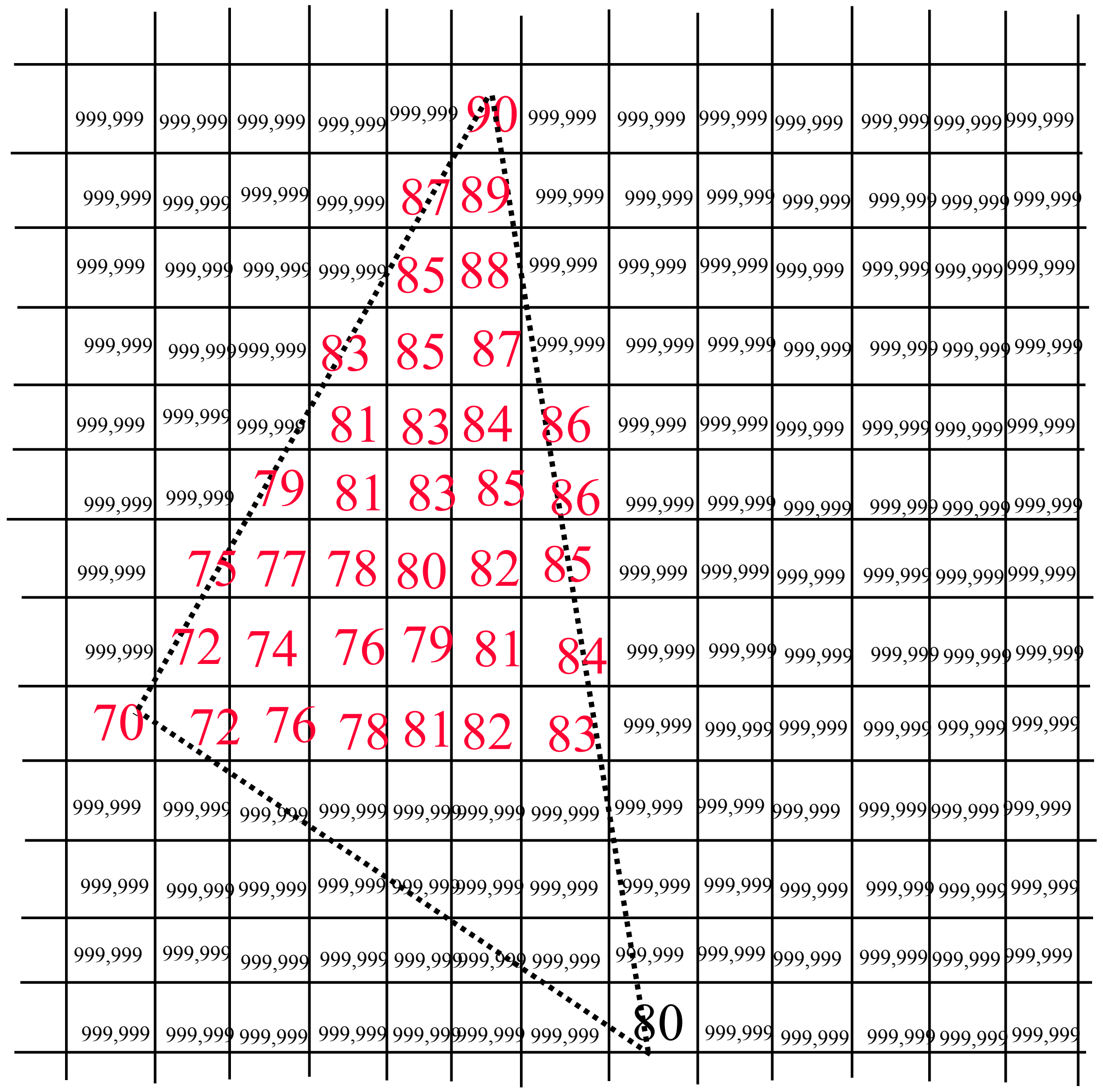


The Z-Buffer Algorithm

- These depth values are stored in an area of memory called the z-buffer. The z-buffer contains an entry for every pixel on the viewport.
- This is in addition to the refresh buffer that stores a colour value for every pixel.
- The only tricky bit is calculating the depth values. We know the depth values for the vertices of a polygon (because we have their original (x,y,z) coordinates) but the algorithm will have to estimate the depth values for points inside the polygon.
- The following slides show the z-buffer process in action as it projects two polygons.

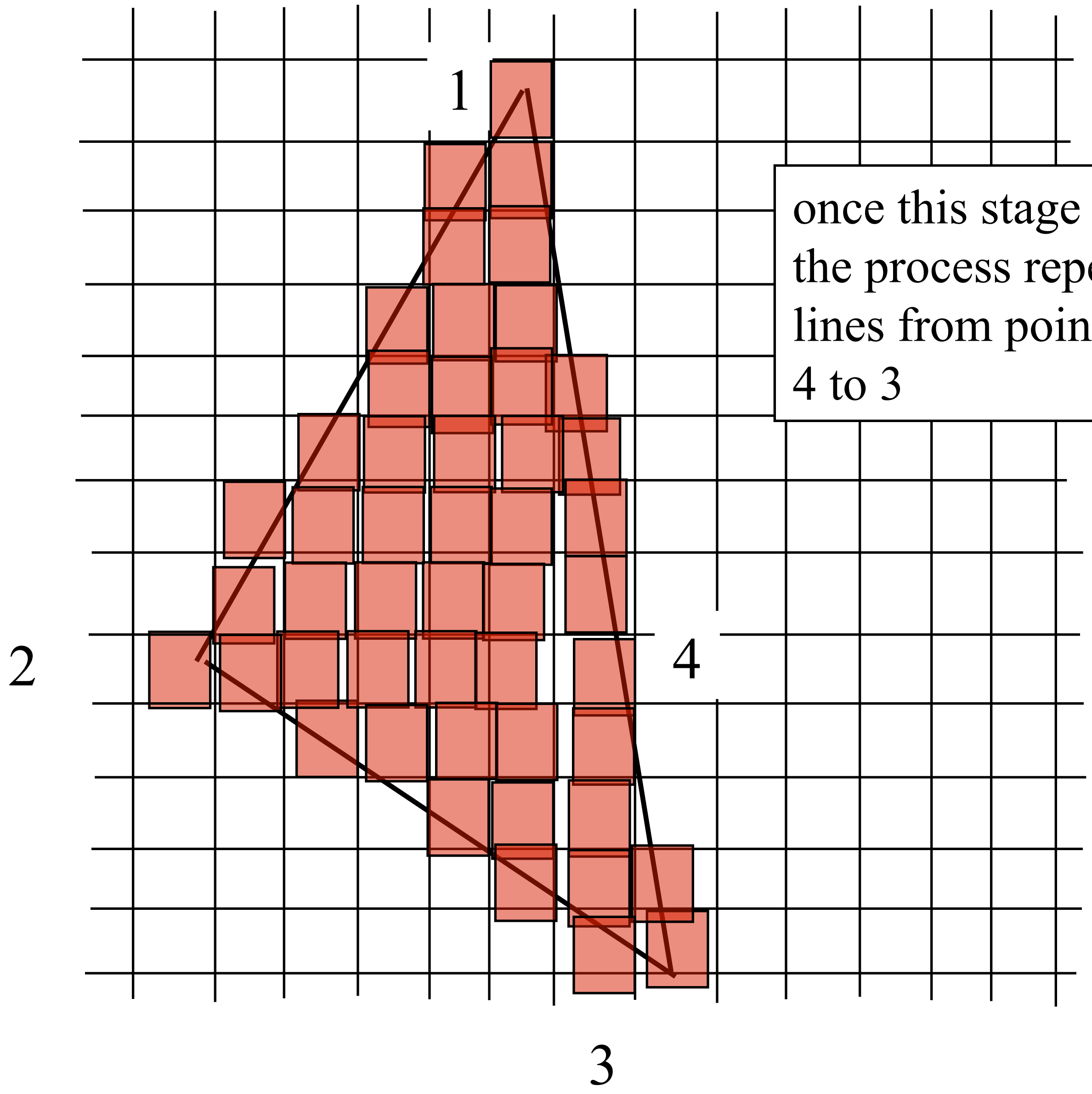


- Start off with the first polygon.
- Its vertices get projected and map onto three pixels on the screen.
- The rest of the pixels are then filled in by moving down the edges and filling in across the rows.

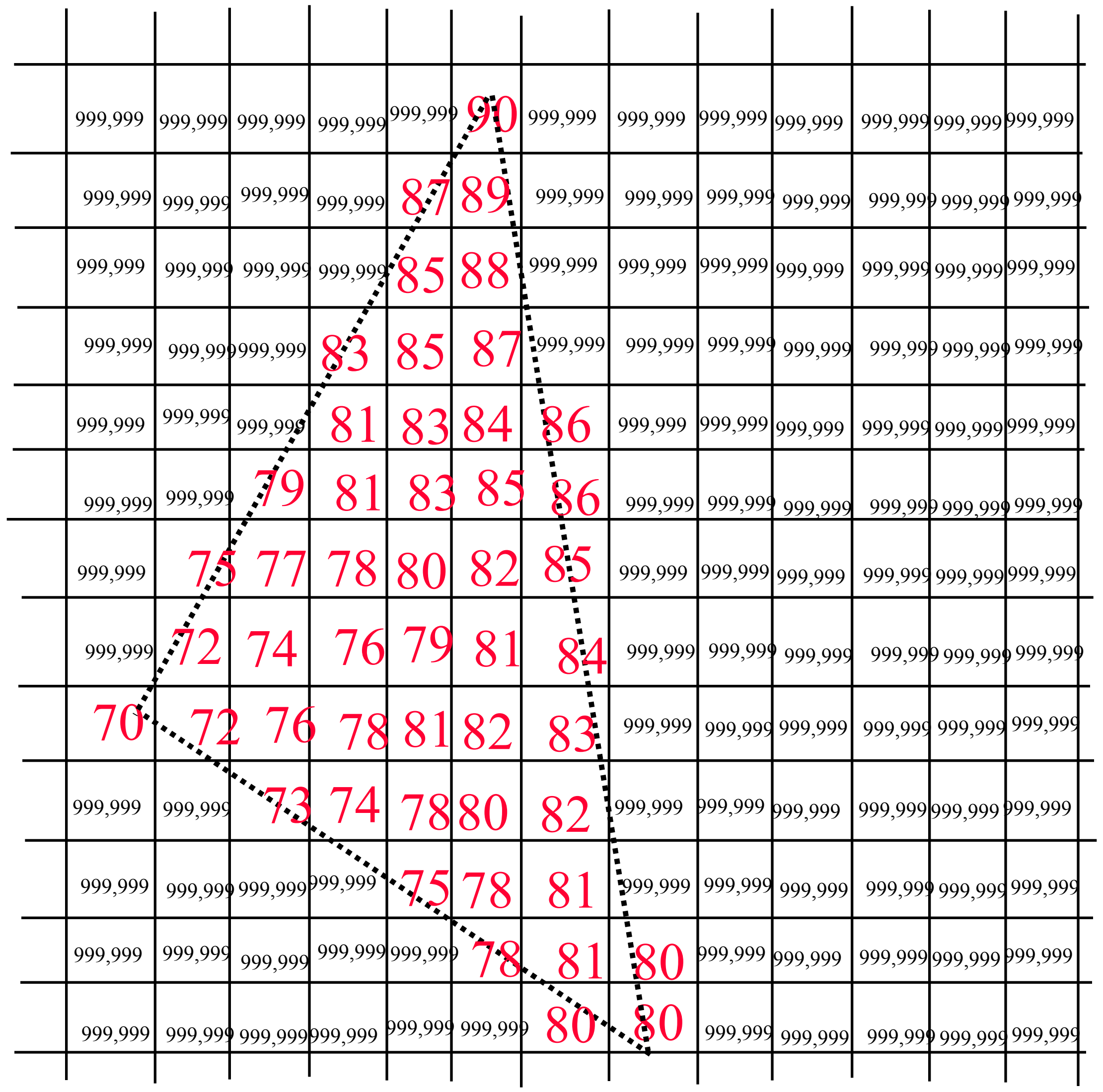


999,999	999,999	999,999	999,999	999,999	90	999,999	999,999	999,999	999,999	999,999	999,999	999,999
999,999	999,999	999,999	999,999	87	89	999,999	999,999	999,999	999,999	999,999	999,999	999,999
999,999	999,999	999,999	999,999	85	88	999,999	999,999	999,999	999,999	999,999	999,999	999,999
999,999	999,999	999,999	83	85	87	999,999	999,999	999,999	999,999	999,999	999,999	999,999
999,999	999,999	999,999	81	83	84	86	999,999	999,999	999,999	999,999	999,999	999,999
999,999	999,999	79	81	83	85	86	999,999	999,999	999,999	999,999	999,999	999,999
999,999	75	77	78	80	82	85	999,999	999,999	999,999	999,999	999,999	999,999
999,999	72	74	76	79	81	84	999,999	999,999	999,999	999,999	999,999	999,999
70	72	76	78	81	82	83	999,999	999,999	999,999	999,999	999,999	999,999
999,999	999,999	999,999	999,999	999,999	999,999	999,999	999,999	999,999	999,999	999,999	999,999	999,999
999,999	999,999	999,999	999,999	999,999	999,999	999,999	999,999	999,999	999,999	999,999	999,999	999,999
999,999	999,999	999,999	999,999	999,999	999,999	999,999	999,999	999,999	999,999	999,999	999,999	999,999
999,999	999,999	999,999	999,999	999,999	999,999	999,999	80	999,999	999,999	999,999	999,999	999,999

the z buffer now looks a bit like this

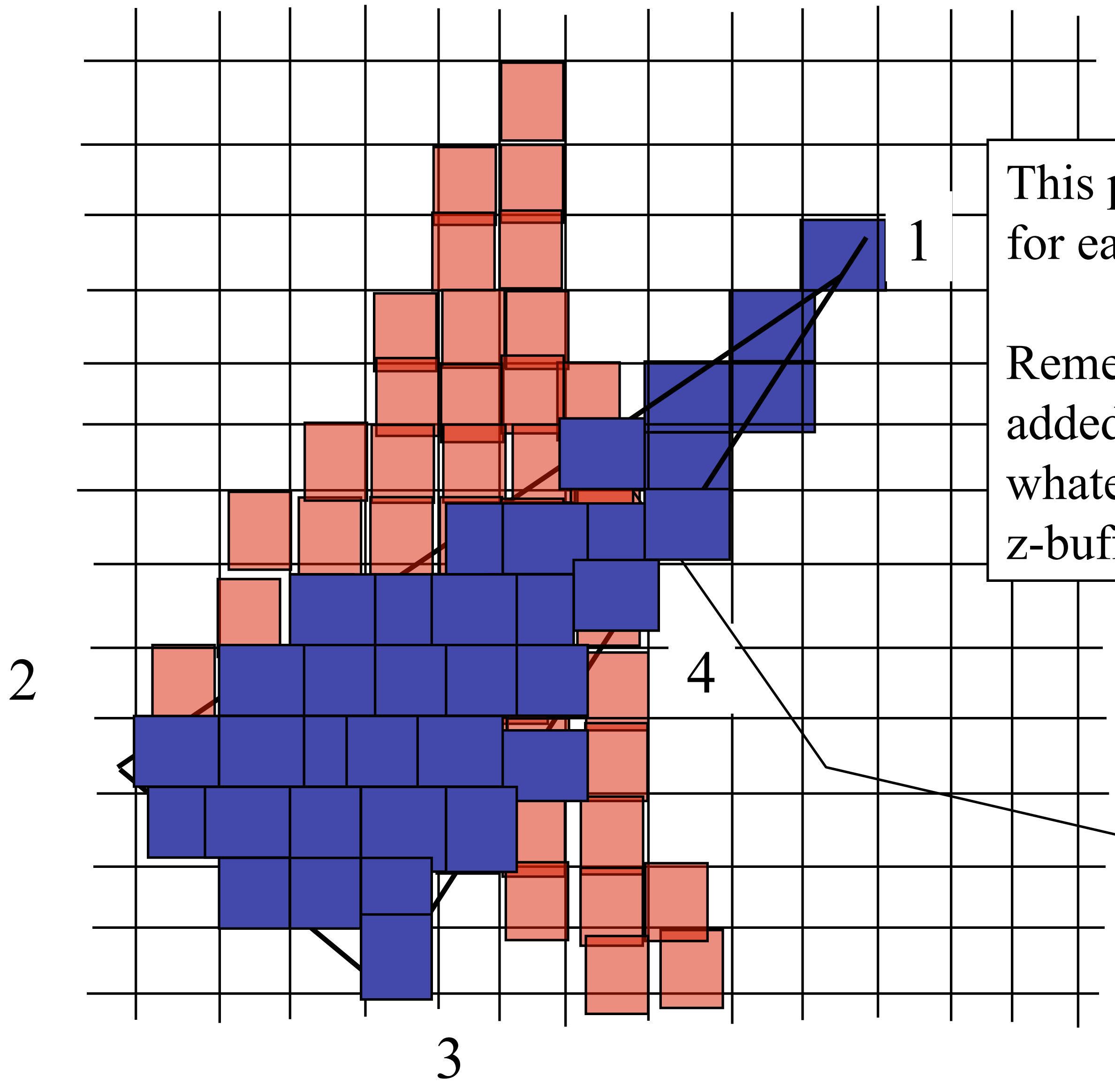


once this stage is reached the process repeats, drawing lines from point 2 to 3 and 4 to 3



999,999	999,999	999,999	999,999	999,999	90	999,999	999,999	999,999	999,999	999,999	999,999	999,999
999,999	999,999	999,999	999,999	87	89	999,999	999,999	999,999	999,999	999,999	999,999	999,999
999,999	999,999	999,999	999,999	85	88	999,999	999,999	999,999	999,999	999,999	999,999	999,999
999,999	999,999	999,999	83	85	87	999,999	999,999	999,999	999,999	999,999	999,999	999,999
999,999	999,999	999,999	81	83	84	86	999,999	999,999	999,999	999,999	999,999	999,999
999,999	999,999	79	81	83	85	86	999,999	999,999	999,999	999,999	999,999	999,999
999,999	75	77	78	80	82	85	999,999	999,999	999,999	999,999	999,999	999,999
999,999	72	74	76	79	81	84	999,999	999,999	999,999	999,999	999,999	999,999
70	72	76	78	81	82	83	999,999	999,999	999,999	999,999	999,999	999,999
999,999	999,999	73	74	78	80	82	999,999	999,999	999,999	999,999	999,999	999,999
999,999	999,999	999,999	999,999	75	78	81	999,999	999,999	999,999	999,999	999,999	999,999
999,999	999,999	999,999	999,999	999,999	78	81	80	999,999	999,999	999,999	999,999	999,999
999,999	999,999	999,999	999,999	999,999	999,999	80	80	999,999	999,999	999,999	999,999	999,999

the z buffer now looks a bit like this

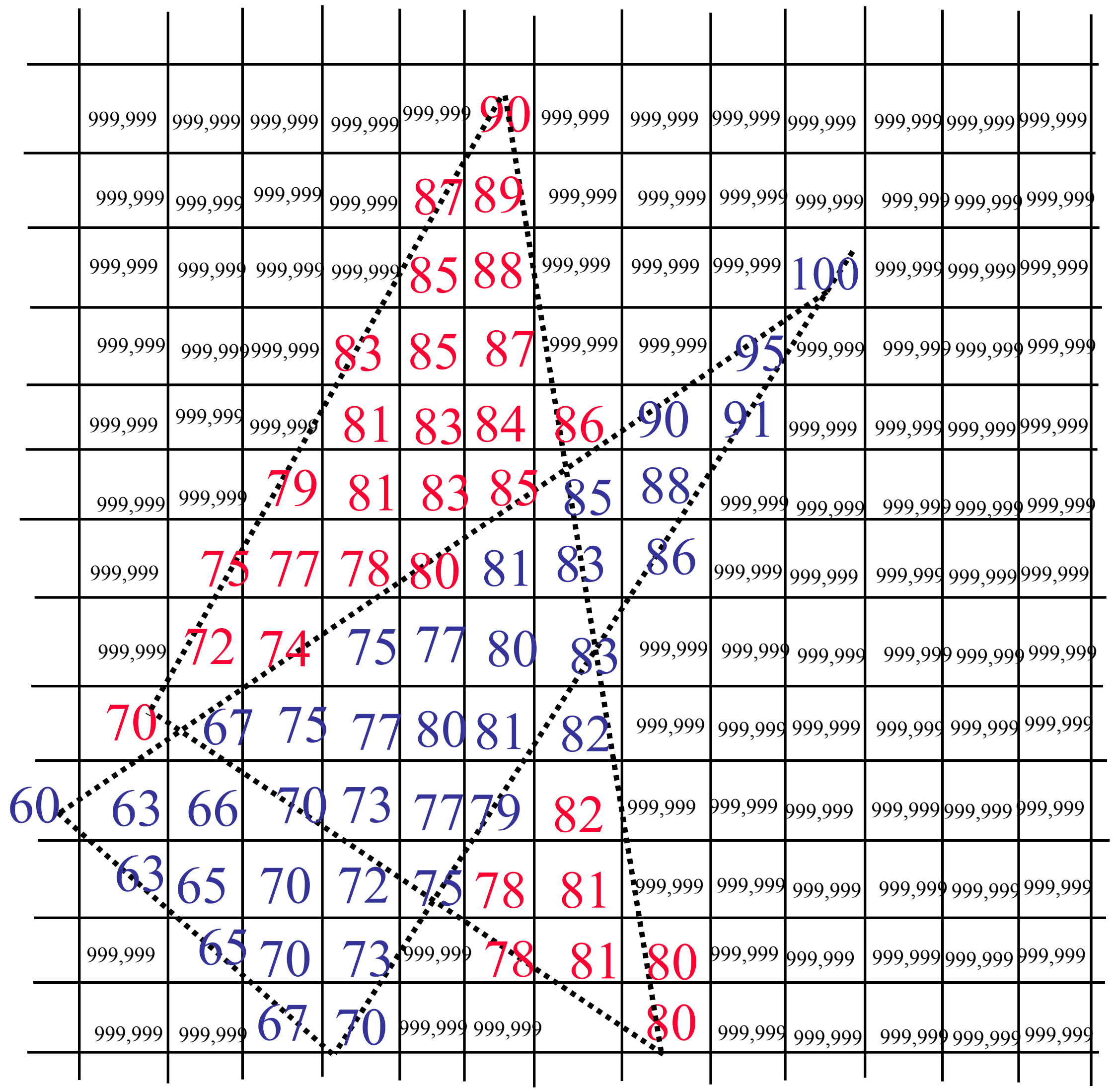


This process is repeated for each triangle.

Remember a pixel is only added if the z value is < whatever's already in the z-buffer.

look at this for crack...

it will figure out that the blue ones are closer and overwrite



	999,999	999,999	999,999	999,999	999,999	90	999,999	999,999	999,999	999,999	999,999	999,999	999,999
	999,999	999,999	999,999	999,999	87	89	999,999	999,999	999,999	999,999	999,999	999,999	999,999
	999,999	999,999	999,999	999,999	85	88	999,999	999,999	999,999	100	999,999	999,999	999,999
	999,999	999,999	999,999	83	85	87	999,999	999,999	95	999,999	999,999	999,999	999,999
	999,999	999,999	999,999	81	83	84	86	90	91	999,999	999,999	999,999	999,999
	999,999	999,999	79	81	83	85	85	88	999,999	999,999	999,999	999,999	999,999
	999,999	75	77	78	80	81	83	86	999,999	999,999	999,999	999,999	999,999
	999,999	72	74	75	77	80	83	999,999	999,999	999,999	999,999	999,999	999,999
	70	67	75	77	80	81	82	999,999	999,999	999,999	999,999	999,999	999,999
60	63	66	70	73	77	79	82	999,999	999,999	999,999	999,999	999,999	999,999
	63	65	70	72	75	78	81	999,999	999,999	999,999	999,999	999,999	999,999
	999,999	65	70	73	999,999	78	81	80	999,999	999,999	999,999	999,999	999,999
	999,999	999,999	67	70	999,999	999,999		80	999,999	999,999	999,999	999,999	999,999

The Z-Buffer Algorithm

- So the algorithm can be written down as follows:

```
For each pixel (x,y) on the screen
    Set refresh (x,y) to black
    Set depth (x,y) to very large number

For each polygon in P'
    For each vertex (xv,yv,zv) in the polygon
        Project the vertex
    For each pixel (xp,yp)
        Calculate d, the depth of (xp,yp) on the polygon
        If d < depth (xp,yp)
                depth (xp,yp)=d
                refresh (xp,yp)=colour of polygon
```

The Z-Buffer Algorithm

- The z-buffer itself is just an array with an entry for every pixel.
- The same trade-offs regarding space and accuracy are present with z-buffers as are with refresh buffers.
- How many bytes would be need for each entry in the z-buffer?
- Well suppose we use 1 byte (8 bits). That gives us 256 possible values. That means that objects in our 3D world can have 256 possible depth values (or 256 possible z values effectively).
- This is hardly enough. 2 bytes would give us 65,536 which seems more reasonable. Multiply this 2 bytes out by a 1280x768 display and you begin to see the amount of memory required to store the z-buffer.
- In the 70s/80s z-buffers would be programmed in software (i.e. it’s a big

  array being tossed around in memory).

The Z-Buffer Algorithm

- In the 90s we began to see dedicated hardware z-buffers (sometimes called depth buffers) as part of specialised 3D graphic cards for desktop PCs.
- This speeds things up dramatically.
- For example the nVidia geForce2 Ultra card has a 32-bit hardware z-buffer and can process 31 million triangles per second.
- Z-buffer is the standard form of rendering carried out on nearly all 3D graphics systems.

The Z-Buffer in OpenGL

- Primarily three commands to do HSR
  - Instruct OpenGL to create a depth buffer

    glutInitDisplayMode(GLUT_DEPTH | GLUT_RGB)

- Enable depth testing

    glEnable(GL_DEPTH_TEST)

- Initialize the depth buffer every time we draw a new picture

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)