CSE251
Basics of Computer Graphics
Module: Visibility and Culling Module

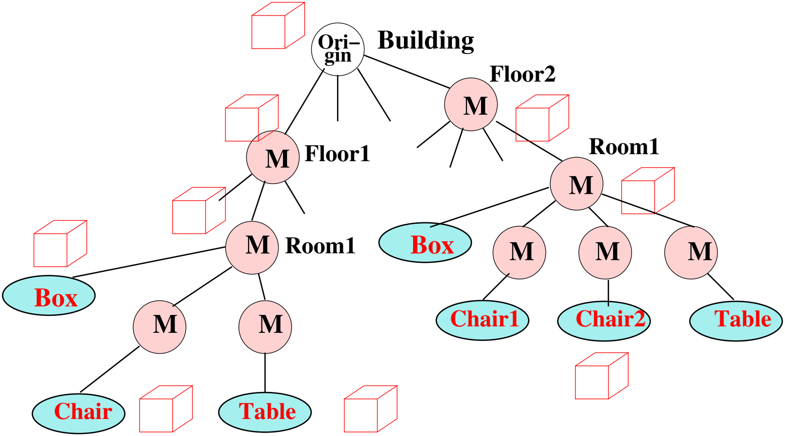
Avinash Sharma

Spring 2017

Visible Surface Determination (VSD)

- Multiple object points can project to the same image pixel in 3D. Which one gives the colour to the pixel?
- **Visible line/surface determination** or
  Hidden line/surface elimination.
- Consider objects to be drawn or pixels to be painted.
- Need to analyze the scene from the camera.
- We will look at 3 types: **Broad or Object** level, **Medium or Primitive** level, and **Fine or Pixel** level

Good Hierarchical Model



Hierarchical Model: Properties

- Leaf level: objects with geometry/shape, like table/chair
  - **Bounding Box** encloses the object completely
- Intermediate level: Create a bounding box that encloses the child nodes completely
- Bounding box of a node covers geometry below it totally!
  - Root bounding box covers the whole scene!
- Question: If cameras **View Frustum does not intersect a bounding box: ......**

Hierarchical Model: Properties

- Leaf level: objects with geometry/shape, like table/chair
  - **Bounding Box** encloses the object completely
- Intermediate level: Create a bounding box that encloses the child nodes completely
- Bounding box of a node covers geometry below it totally!
  - Root bounding box covers the whole scene!
- Observation: If View Frustum doesnt intersect the BBox, **no object in it can be visible to the camera!**

View Frustum Culling

- Eliminate objects that are outside the view volume.
- Large parts of the scene will be eliminated this way.

  Compare bounding volume of `node` with view volume

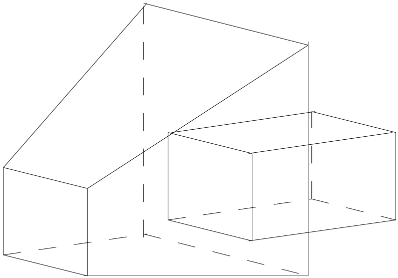
      If intersection is null, eliminate tree from root

      If volume contained in frustum, render without clipping

      Else, recursively check for each child of `node` till leaf

- Object hierarchy **really** helps. Whole campus, each building, each floor, each room, etc., in the hierarchy.

Box-Frustum Intersection



- Orthographic: Intersection of two boxes. Simultaneous X-Y-Z overlaps.
- Perspective: Clip against 6 planes of the frustum.
- **AABB**: Axis-Aligned Bounding Box
  OBB: Oriented Bounding Box
- AABB: Easy to find the box, but not efficient
  OBB: More accurate, but more computation

AABB vs OBB

- Calculating the AABB of a model: How?

AABB vs OBB

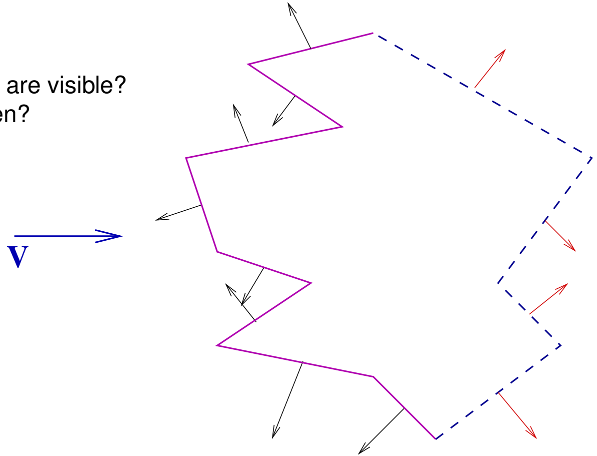
- Calculating the AABB of a model: How?
- In ORC, find separately the minimum and maximum values of X, Y, and Z
- What are the plusses and minuses of AABB?
- How can we calculate the OBB of a given model?
  - Find the smallest bounding volume: somewhat involved
- Find the major/minor axes for the shape and fit a box

View Frustum Culling: VFC

- Efficient frustum-box intersection methods exist
- Large portions of the scene eliminated from consideration
- Note: **Drawing the discarded objects will still yield the correct picture!**
- VFC is primarily for speed, reducing the number of objects to be drawn
- This is done before drawing, at the start of the pipeline

Viewing a Solid Object

- Which polygons are visible?
  Which are hidden?



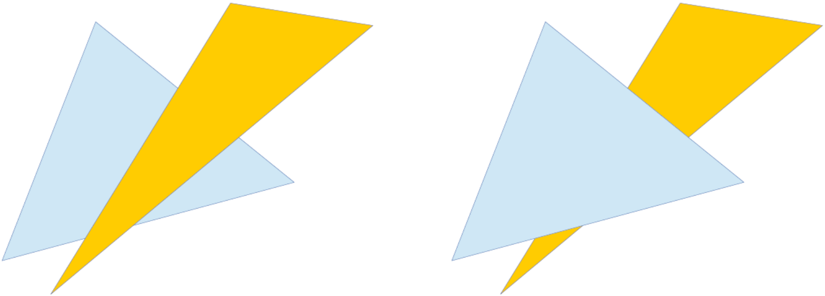
Back-Face Culling

- Only the front side can be seen of solid, opaque objects.
- If surface normals point out of the object, polygons with normals pointing away from the viewer cannot be seen.
- If $\mathbf{v} \cdot \mathbf{n} < \mathbf{0}$, draw. Else discard. $\mathbf{v}$ is a vector from CoP to any point on the polygon and $\mathbf{n}$ the surface normal.
- After normalizing, DoP is $[0\ 0\ \text{-}1]^{\mathsf{T}}$. Eliminate polygons with negative z component in the normal vector.
- Half the polygons eliminated on the average.
- If there is only one object, no other VSD necessary.

Culling vs Visibility

- View Frustum Culling (VFC): Identify and ignore entire objects outside the current view volume.
- Back Face Culling (BFC): Identify and ignore triangles facing away.
- Eliminating portions that are guaranteed to be not visible. Improves efficiency or speed.
- Visibility between triangles inside the view frustum involve their relative arrangement.
  - **Pixel-level visibility needs to be determined.**

Which of the two cases?



- Parts of one primitive is in front. Need correct picture
- Back-to-front drawing gives correct picture, when possible

Object-Precision Algorithm

for each object in the world {
    Determine unobstructed parts of the object
    Draw those parts with appropriate colours
}

- Complexity depends on the number of objects.
- If the image size changes, only the drawing step needs to be redone.
- Works in the original continuous object space.

Image-Precision Algorithm

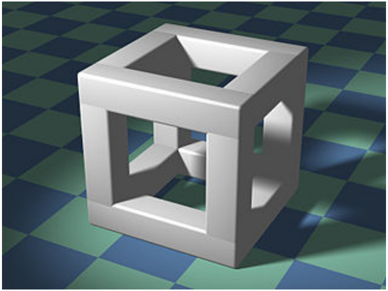
for each pixel in the image {
    Determine closest object in the direction of projector
    Draw the pixel with appropriate colours
}

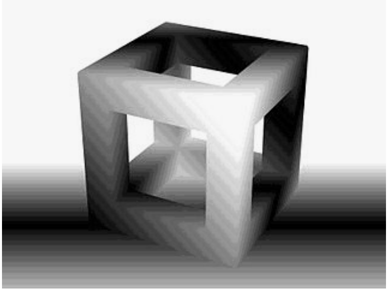
- Works in the discrete image space.
- Complexity depends on the number of pixels.
- A natural choice for raster graphics.
- Image resizing involves repeating the entire work.

Z-buffer or Depth-buffer Algorithm

- An image-precision algorithm that needs a z-buffer parallel to the frame buffer.
- z values after the normalizing transformation is stored into the z-buffer along with colour info to the frame buffer. We have $0 \geq z \geq -1$.
- Larger z implies a closer 3D point in any direction
- Write to frame buffer and z-buffer only when the z value is larger than previously stored value.
- **FrameBuffer + DepthBuffer**

Colour and Depth





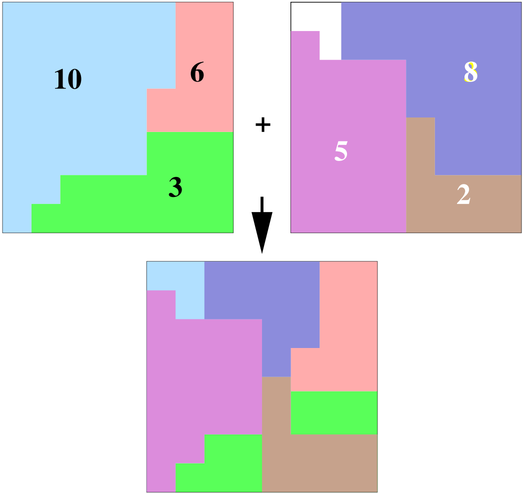
- `Color[i, j]` holds the colour and `Depth[i,j]` the distance (or z-value) of the **nearest** object encountered in the direction of pixel `(i, j)` at *any point of time,* during the rendering

Pseudocode

```
for 0 ≤ y ≤ Ymax
        for 0 ≤ x ≤ Xmax
                WritePixel (x, y, bgnd_colour)
                WriteZ (x, y, -1)      // Farthest value

for each polygon
        for each pixel in polygon's projection
                pz = polygon's z-value at pixel (x, y)
                if (pz > ReadZ(x, y))
                        WriteZ()
                        WritePixel()
```

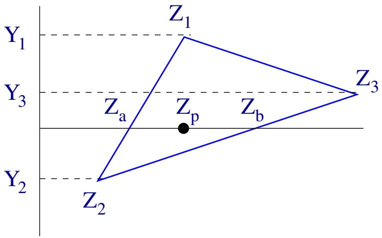
Depth Buffer



Computing z for Interior Points

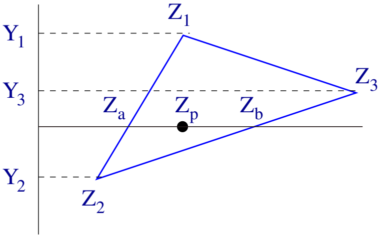
- Exploit coherence in depth values over the plane to compute interior z values given the vertex values.
- $z_a = z_1 + ...$
  $z_b =??$ $\quad z_p =???$
- When x or y increments by 1 along a side or a scan line, z changes by a constant Δz.

  $\Delta z =??$ along edge 12



Computing z for Interior Points

- $z_a = z_1 + (z_2 - z_1)\frac{(y_a - y_1)}{(y_2 - y_1)}$
  $z_b = z_2 + (z_3 - z_2)\frac{(y_b - y_2)}{(y_3 - y_2)}$
  $z_p = z_a + (z_b - z_a)\frac{(x_p - x_a)}{(x_b - x_a)}$
- When x or y increments along a side or a scan line, z changes by a constant Δz

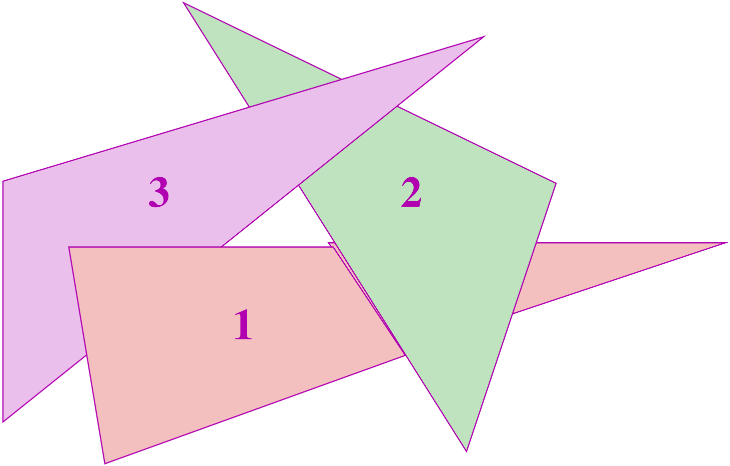


$\Delta z = (z_2 - z_1)/(y_2 - y_1)$ along edge 12
$\Delta z = (z_b - z_a)/(x_b - x_a)$ along scan line

List Priority Algorithms

- Reorder objects such that the correct picture results if you draw them in that order. If objects do not overlap in z, draw them from back to front.
- Objects may need splitting if no unique ordering exists.
- **Needs expensive sorting/reordering every frame!!**
- Ordering and splitting polygons: Object-precision operation.
- Overwriting farther points while scan conversion: Image-precision operation.

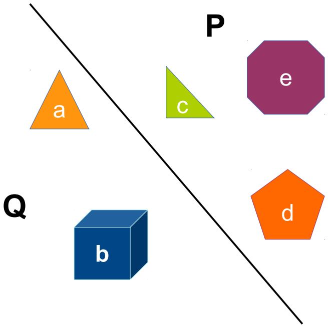
A Difficult Case for Ordering



Binary Space Partitioning Trees

- Can we have simple linear display algorithm, perhaps using expensive preprocessing?
- Consider a plane in space that divides scene into two halfs.
- Objects on the same side of the plane as the eye cannot be blocked by objects on the other side.

Binary Space Partitioning Trees

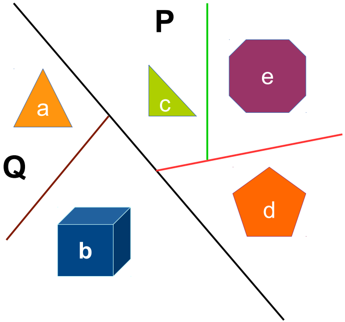


- Ordering of **b** and **c** from viewpoint **P**? From **Q**?

Binary Space Partitioning Trees

- Each side of the plane can further be divided using other planes till we reach a single object.
- If enviroments consist of clusters of objects, separate them using an appropriate plane.
- We end up with the **BSP Tree** representation of the scene.
- Internal nodes contain partitioning planes; leaf nodes are polygons.
- Some preprocessing to construct the tree, but simple algorithm to render using it from any viewpoint.

Binary Space Partitioning Trees

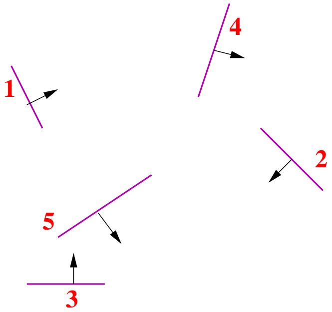


- Total ordering from viewpoint **P**? From **Q**?

BSP Trees

- Use planes of polygons in the scene as partitioning planes.
- Normal direction indicates the “front” side of the plane.
- Each plane divides space into two sides.
- If a polygon lies on both sides of the plane, divide it into two parts.
- Continue this recursively till each side contains exactly one polygon.

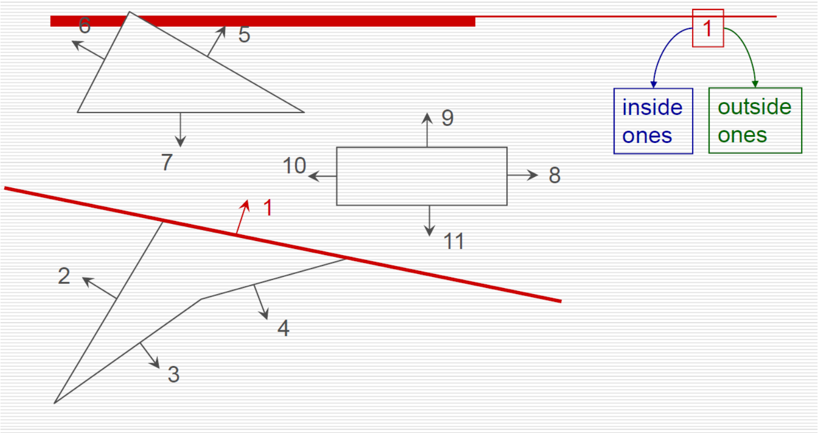
Example: BSP Tree



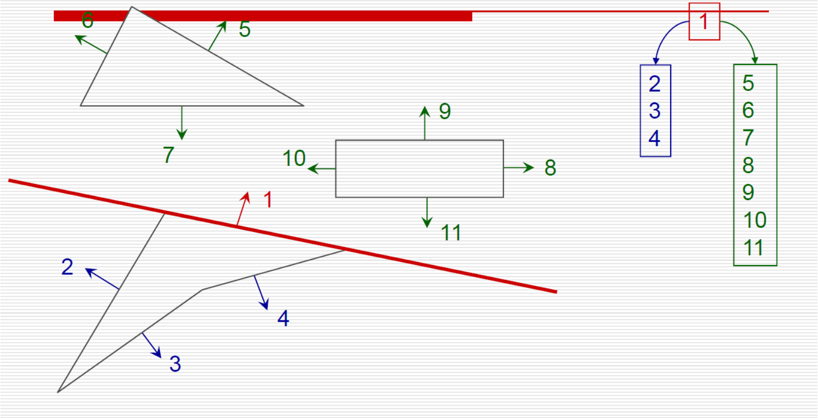
Psuedocode: BSP Tree Construction

```
makeBSPTree(pList){
    if (pList is empty)
        return NULL
    end
    root ← selAndRemove(pList);
    bList, fList ← NULL
    for each polygon p in pList
        if (p is in front of root)
            addToList(p, fList)
        elsif (p is in back of root)
            addToList(p, bList)
        else
            splitPoly(p, fp, bp)
            addToList(fp, fList);
            addToList(bp, bList)
            return combineTree(makeBSPTree(fList), root, makeBSPTree(bList))
        end
    end
}
```

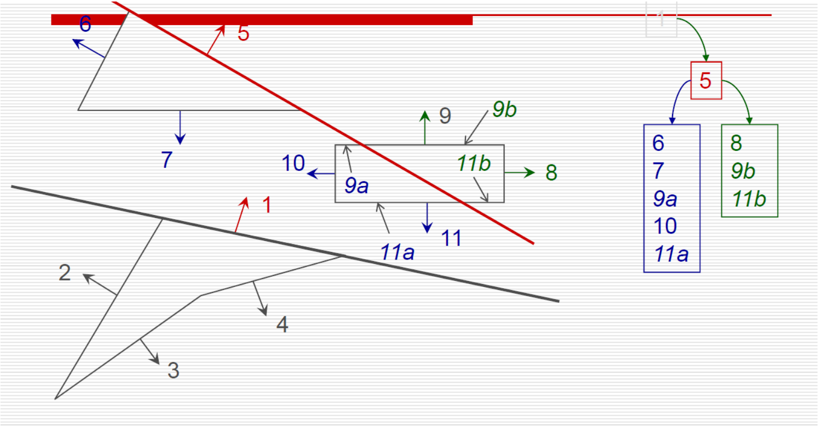
Example:BSP Tree Construction



Example:BSP Tree Construction (cont.)

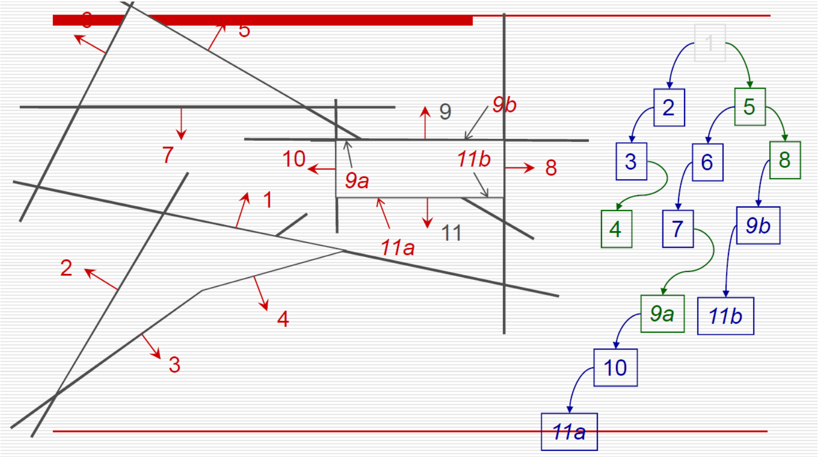


Example:BSP Tree Construction (cont.)

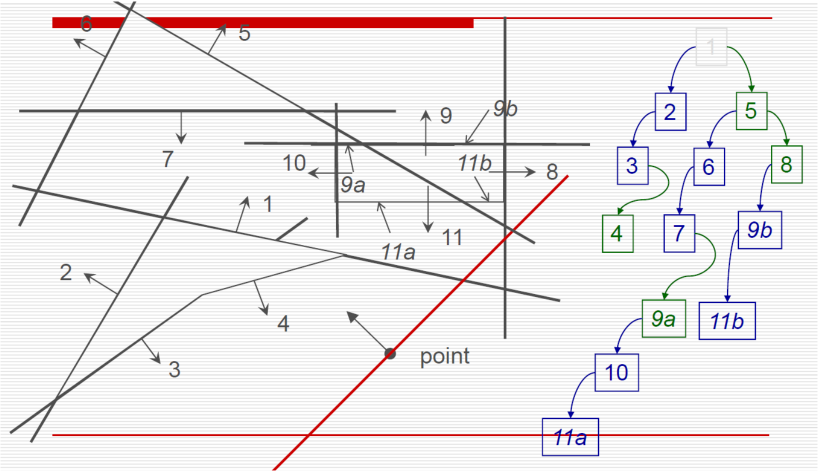


Bing-Yu Chen, National Taiwan University

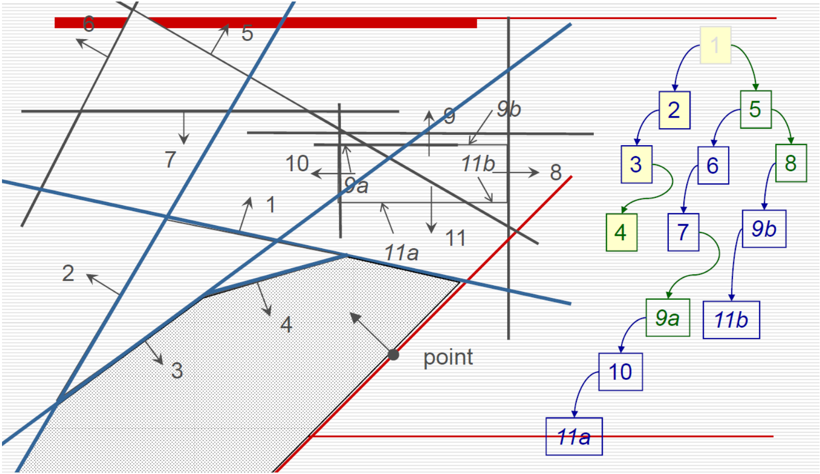
Example:BSP Tree Construction (cont.)



Example:BSP Tree Construction (cont.)



Example: BSP Tree Rendering



Psudeocode: Displaying a BSP Tree

```
displayBSPTree(bTree)
    if (empty(bTree)) return
        if (eye in front of root)
            displayBSPTree(bTree→bChild)
            displayPoly(root)
            displayBSPTree(bTree→fChild)
        else
            display(BSPTree(bTree→fChild)
             displayPoly(root)
             display(BSPTree(bTree→bChild)
```

Psudeocode: Displaying a BSP Tree

```
displayBSPTree(bTree)
    if (empty(bTree)) return
        if (eye in front of root)
            displayBSPTree(bTree→bChild)
            displayPoly(root)
            displayBSPTree(bTree→fChild)
        else
            display(BSPTree(bTree→fChild)
            if (no back-face culling)
                displayPoly(root)
                display(BSPTree(bTree→bChild)
```

Performance Considerations

- Construction of the BSP tree is expensive.
- No splitting while rendering; everything is done during preprocessing.
- Straightforward display algorithm. Back-face culling woven into it.
- Strategy of selecting the root has a great impact.
- Select the polygon that splits least number of polygons.

Other Methods

- **Painter's Algorithm**: Reorder polygons back-to-front from the camera
  - Involves sorting the polygons for each view point
  - Sometimes, polygons need to be split as no unique ordering
- **Ray-Casting**: Examine each ray from the camera center
  - Expensive operation to *trace each ray from camera*
  - Can provide very high visual realism in addition to visibility

Depth-Sort Algorithms: Discussion

- Ensures back-to-front ordering for proper rendering.
- No aliasing effects introduced as objects are reordered/split.
- Reordering and splitting of polygons have to be done at run time.
- Redo the whole calculations if the view-point changes.
- Computationally expensive.

Z-buffering: Discussion

- Any shape with per pixel z can be handled correctly.
- Time is independent of number of primitives.
- Easy to implement; can do with a single scan-line Z-buffer.
- Z-buffer can be read back and saved.
- Needs extra memory, but memory is cheap.
- Can cause aliasing or **z-fighting** (*shimmering*).

VSD: Summary

- Sorting in the right order is key to all of them.
- If expensive lighting/shading is used, do not shade an image pixel more than once.
- For quick rendering, z-buffer algorithms are better.
- BSP Trees can be fast if environment is static.
- Ease of implementation and scope of hardware acceleration are also important.
- Z-buffering is popular due to memory being cheap.