Xi'an Jiaotong-Liverpool University
西交利物浦大学

CPT205 Computer Graphics

Hidden-Surface Removal

Lecture 12
2025-26

Yong Yue and Nan Xiang

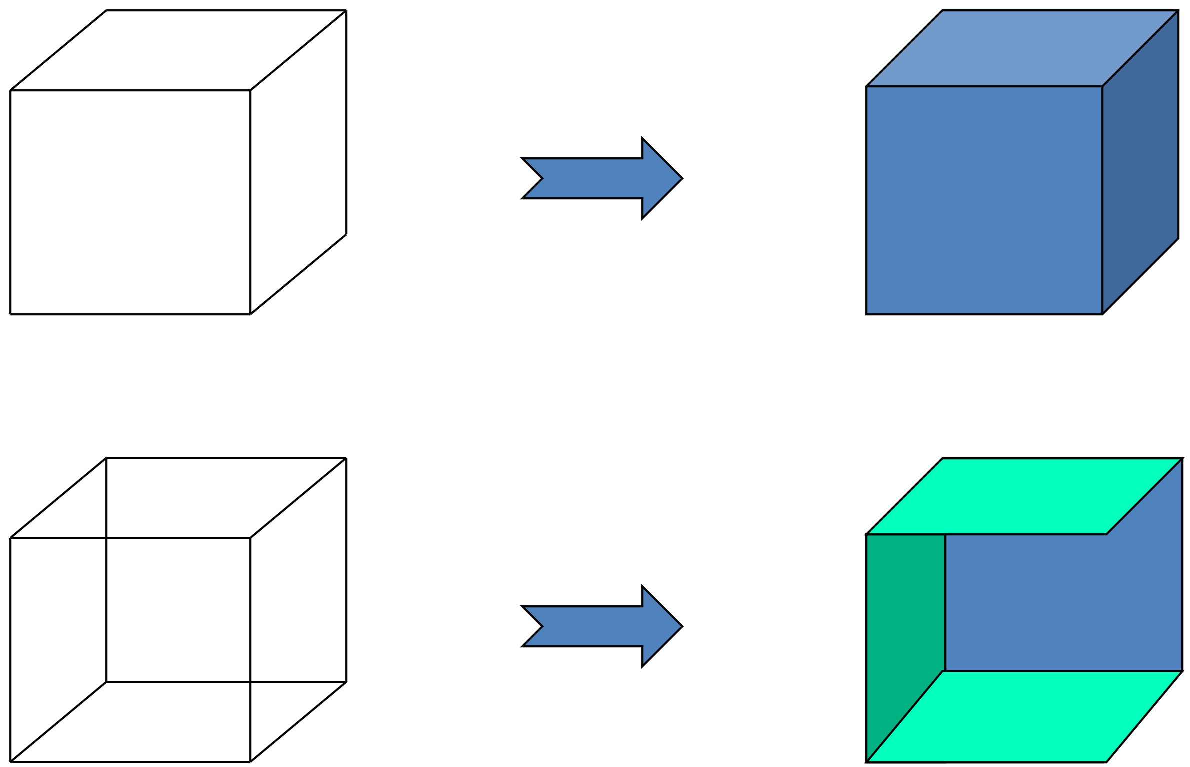
Topics for today

- **Concepts**
  - Clipping vs Hidden Surface Removal
  - What is hidden-surface removal?
  - Why is hidden-surface removal important?
- **Hidden-surface removal algorithms**
  - Object space vs image space
  - Painter’s algorithm
  - Back-face culling
  - Z-buffer
  - BSP tree
- **Hidden-surface removal in OpenGL**

Clipping and hidden-surface removal

- Clipping has much in common with hidden-surface removal.
- In both cases, we try to remove objects (or parts of objects) that are not visible to the camera.
- Often we can use visibility or occlusion testing early in the process to eliminate as many polygons as possible before going through the entire pipeline (rasterization).

Concepts (1)



Concepts (2)

- Display all visible surfaces, and do not display any occluded surfaces.

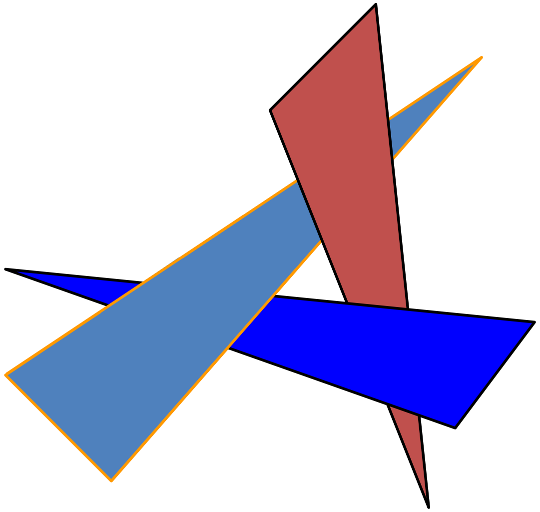
- Other names
  - Visible-surface detection
  - Hidden-surface elimination

- Determine which surfaces are visible and which are not
  - Z-buffer is just one of hidden-surface removal algorithms

- We can categorise into
  - Object-space methods
  - Image-space methods

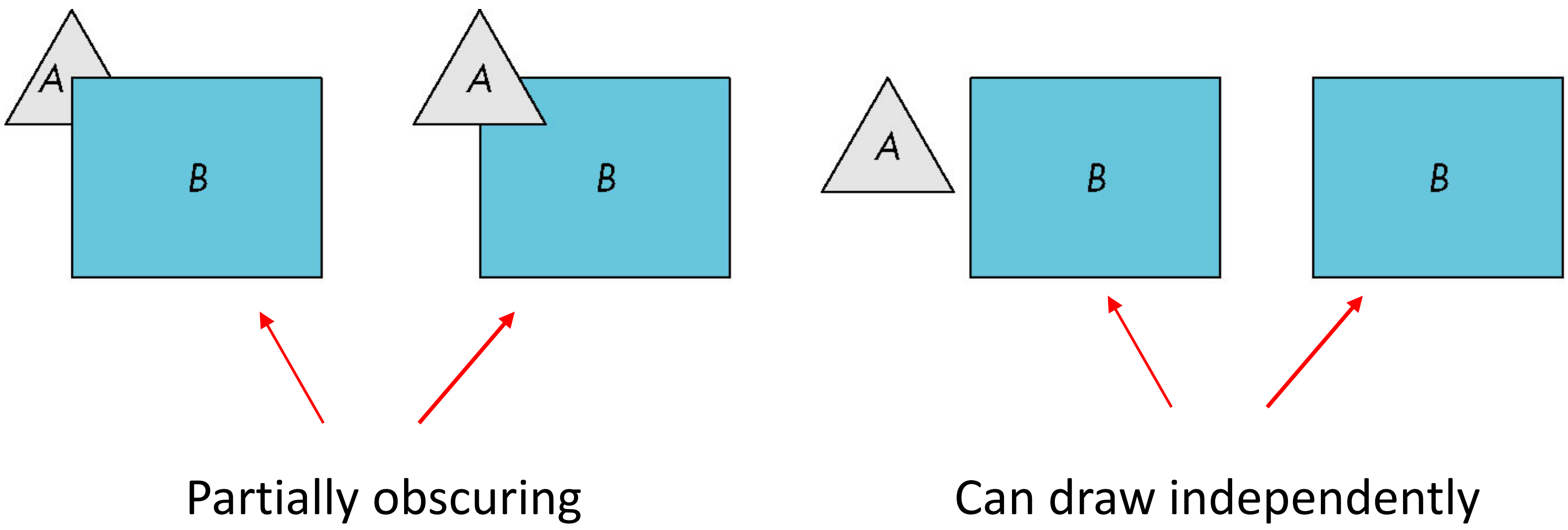
Concepts (3)

- Object space algorithms: determine which objects are in front of others
  - Resize does not require recalculation;
  - Works for static scenes;
  - May be difficult / impossible to determine.
- Image space algorithms: determine which object is visible at each pixel
  - Resize requires recalculation;
  - Works for dynamic scenes.



Object-space approach

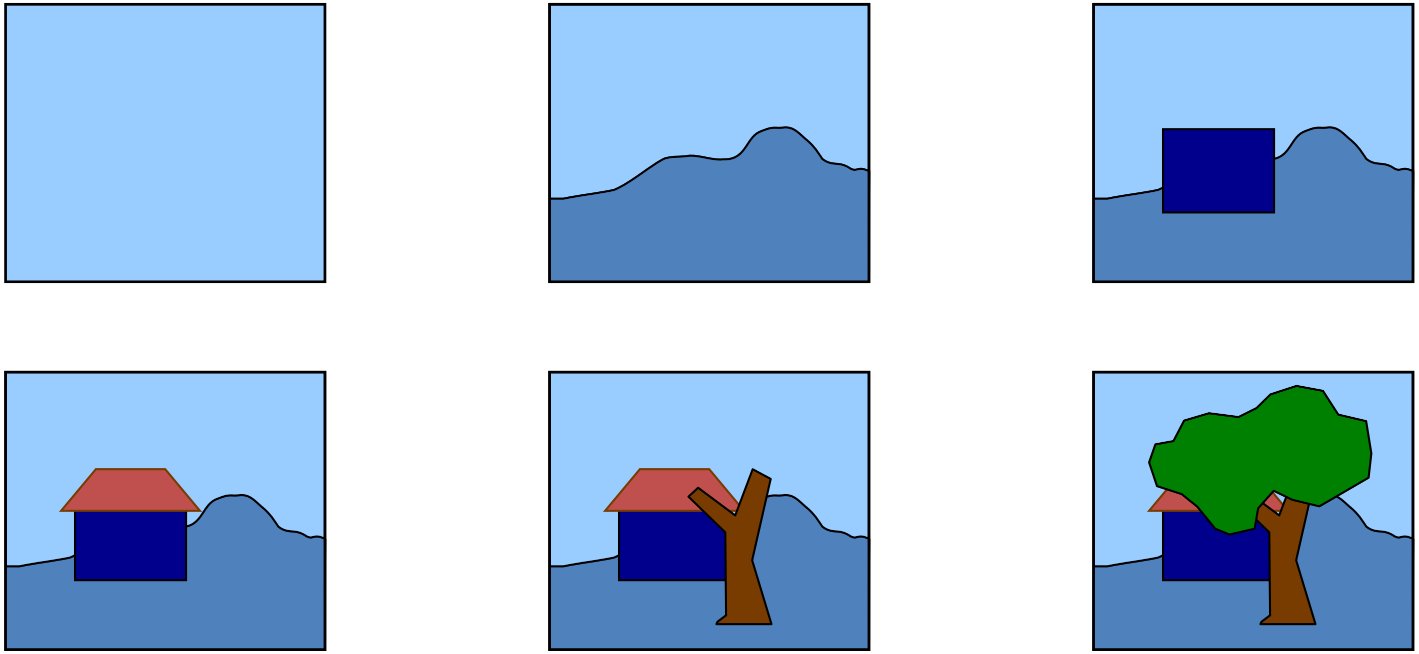
- It uses pairwise testing between polygons (objects).



- Worst case complexity $O(n^2)$ for n polygons.

Painter's algorithm

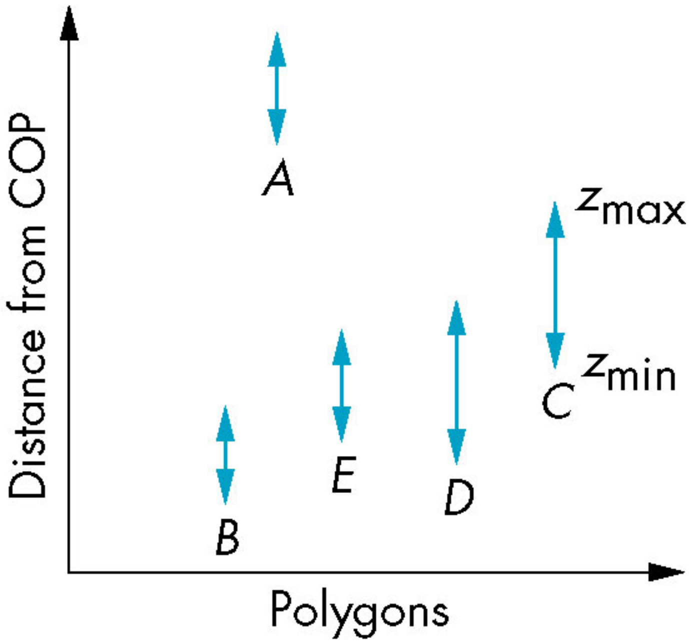
- It is an object-space algorithm.
- Render polygons in the back to front order so that polygons behind others are simply painted over.
- Sort surfaces/polygons by their depth (z value).



Painter's algorithm: Depth sort

- Requires ordering of polygons first
  - O(n log n) calculations for ordering
  - Not every polygon is either in front or behind all other polygons

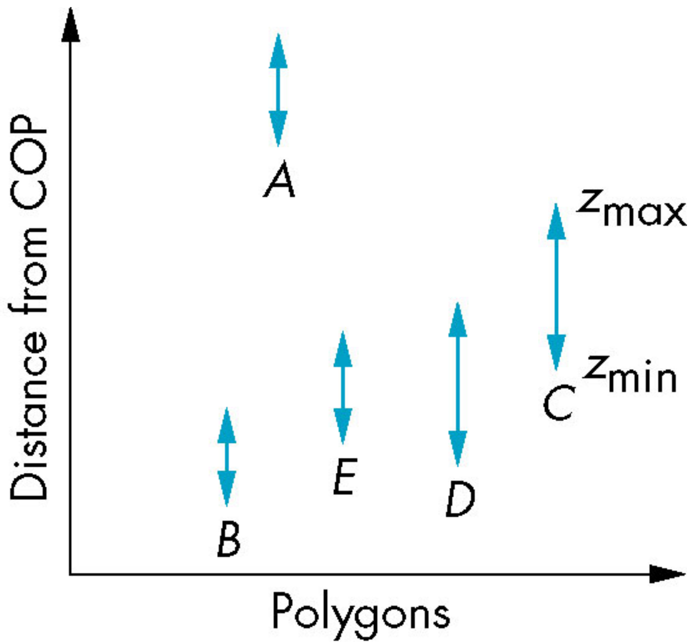
- Orders polygons and deals with easy cases first and hard later.

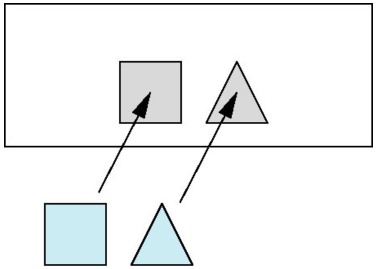


Polygons sorted by distance from Centre Of Projection (COP)

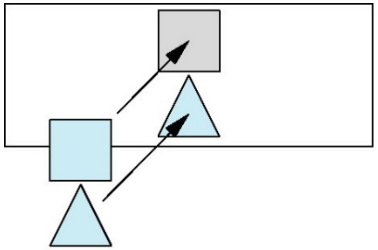
Painter's algorithm: Easy cases

- Polygon A lies entirely behind all other polygons, and can be painted first.
- Polygons which overlap in *z* (depth) but not in either *x* or *y*, can be painted independently.



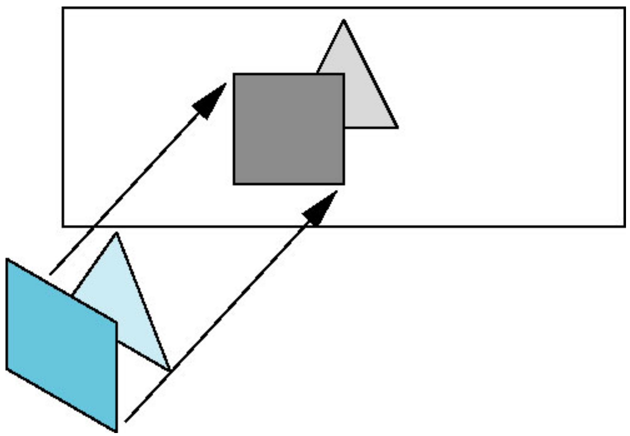


Test of overlap in *x*

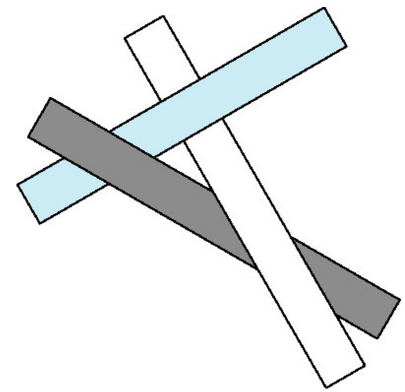


Test of overlap in *y*

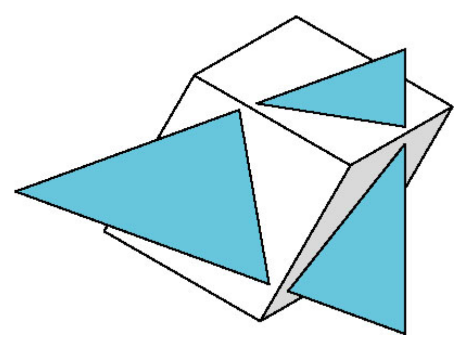
Painter's algorithm: Hard cases



Overlap in all directions
(polygons with
overlapping projections)



Cyclic overlap



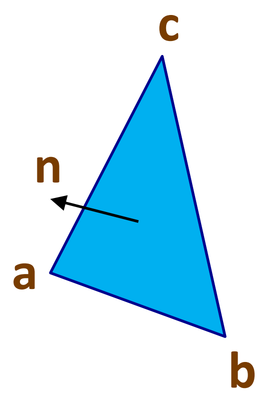
Penetration

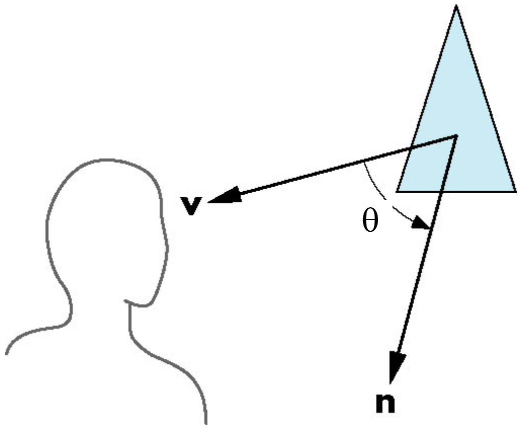
Back-face culling (1)

- Back-face culling is the process of comparing the position and orientation of polygons against the viewing direction **v**, with polygons facing away from the camera eliminated.
- This elimination minimises the amount of computational overhead involved in hidden-surface removal.
- Back-face culling is basically a test for determining the visibility of a polygon, and based on this test the polygon can be removed if not visible – also called back-surface removal.

Back-face culling (2)

- Assume the object is a solid polyhedron.
- Compute polygon normal **n**:
  - Assume a counter-clockwise vertex order
  - For a triangle (**abc**): **n** = (**ab**) × (**bc**)
- If 90°≥ θ ≥ -90°, i.e. **v•n** ≥ 0,
  the polygon is a visible face.
- Furthermore, if the object descriptions are converted to projection co-ordinates and the viewing direction is parallel to the viewing z_v axis, i.e. **n** = $(0,0,1,1)^T$, only the sign of the *z* component of **n** is tested.

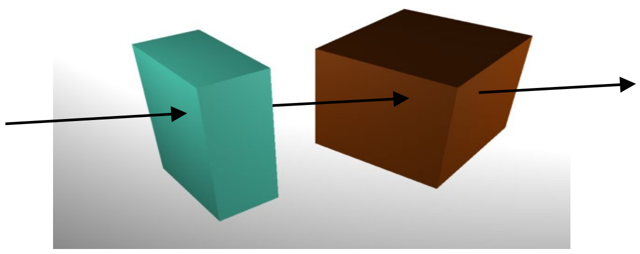




Back-face culling (3)

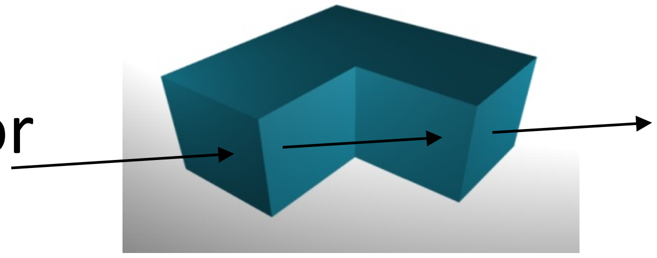
For each polygon P_i

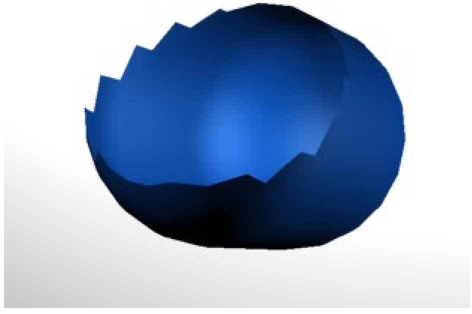
- ➢ Find polygon normal **n**
- ➢ Find viewer direction **v**
- ➢ If $\mathbf{v} \bullet \mathbf{n} < 0$, then cull P_i



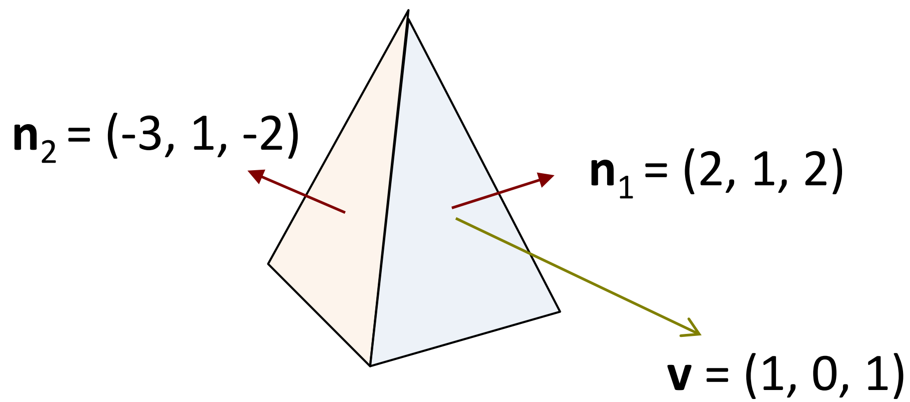
Back-face culling does not work for

- ➢ Overlapping front faces due to
  - Multiple objects
  - Concave objects
- ➢ Non-polygonal models
- ➢ Non-closed objects





Back-face culling: Example

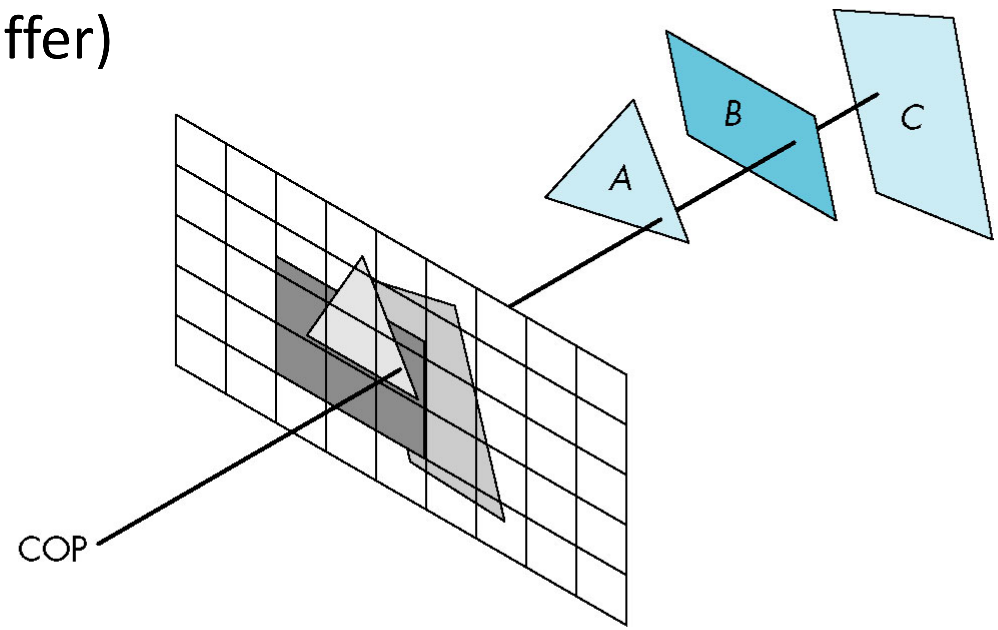


$\mathbf{n}_1 \cdot \mathbf{v} = (2, 1, 2) \cdot (1, 0, 1)$
$= 2 + 0 + 2 = 4,$
i.e. $\mathbf{n}_1 \cdot \mathbf{v} > 0$,
so face_1 is a front facing polygon.

$\mathbf{n}_2 \cdot \mathbf{v} = (-3, 1, -2) \cdot (1, 0, 1)$
$= -3 + 0 - 2 = -5$
i.e. $\mathbf{n}_2 \cdot \mathbf{v} < 0$,
so face_2 is a back facing polygon.

Image space approach

- Look at each projector (n*m projectors for an n*m framebuffer) and find the closest among k polygons
- Complexity O(nmk)
- Ray tracing
- Z-buffer (depth buffer)



Z-buffer (1)

- We now know which pixels contain which object(s).
- However since some pixels may contain two or more objects, we must calculate which of these objects are visible and which are hidden.
- In graphics hardware, hidden-surface removal is generally accomplished using the Z-buffer algorithm.

Z-buffer (2)

- In this algorithm, we set aside a two-dimensional array of memory (the Z-buffer) of the same size as the screen (number of rows * number of columns).
- This is in addition to the colour buffer (frame buffer) which we use to store the colour values of pixels to be displayed.
- The Z-buffer will hold values which are depths (quite often z-values).
- The Z-buffer is initialised so that each element has the value of the far clipping plane (the largest possible z-value after clipping is performed).
- The colour buffer is initialised so that each element contains a value which is the background colour.

Z-buffer (3)

- Now for each polygon, we have a set of pixel values which the polygon covers.
- For each of the pixels, we compare its depth (z-value) with the value of the corresponding element already stored in the Z-buffer:
  - If this value is less than the previously stored value, the pixel is nearer the viewer than the previously encountered pixel;
  - Replace the value of the Z- buffer with the z value of the current pixel, and replace the value of the colour buffer with the value of the current pixel.
- Repeat for all polygons in the image.
- The implementation is typically done in normalised co-ordinates so that depth values range from 0 at the near clipping plane to 1.0 at the far clipping plane.

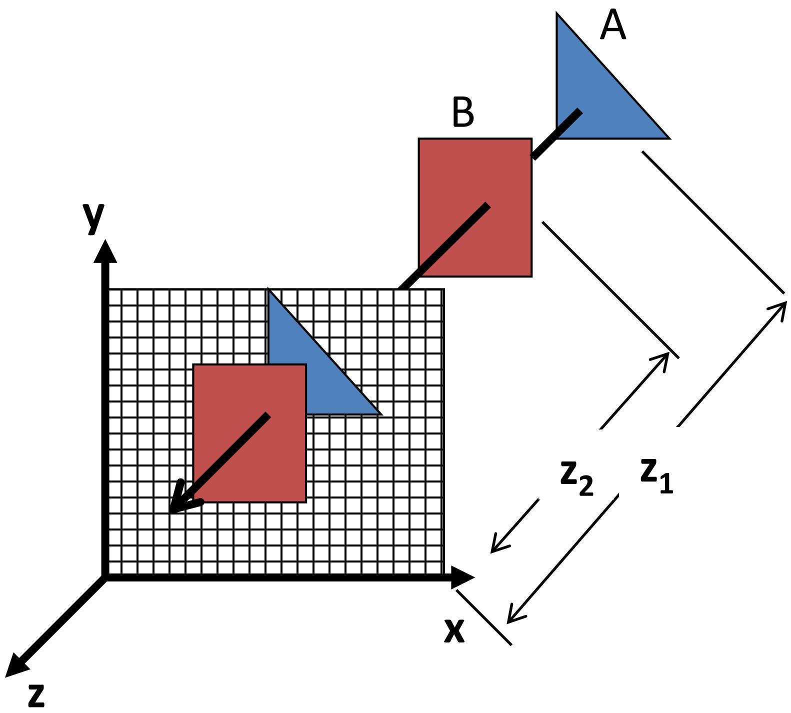
Z-buffer (4)

1. Initialise all *depth(x,y)* to 1.0 and *refresh(x,y)* to background colour

2. For each pixel
   Get current value for depth(x,y)
   Evaluate depth value z

   If(z < depth(x,y))
   {
     depth(x,y) = z
     refresh(x,y) = $I_s(x,y)$
   }

Calculate this using relevant algorithms /illumination/fill colour/texture



Z-buffer (5)

Advantages

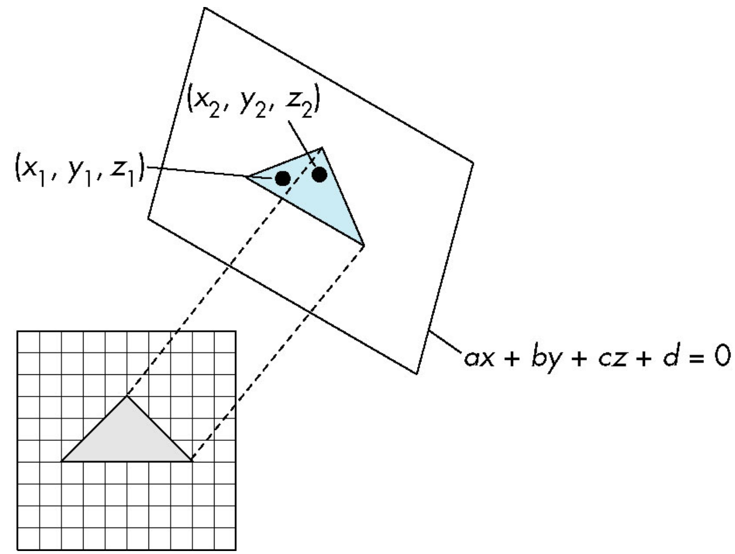
- The most widely used hidden-surface removal algorithm;
- An image-space algorithm which traverses scene and operates per polygon rather than per pixel;
- Rasterize polygon by polygon and determine which (part of) polygons get drawn on the screen;
- Relatively easy to implement in hardware or software.

Memory requirements

- Relies on a secondary buffer called the Z-buffer or depth buffer;
- Z-buffer has the same width and height as the framebuffer;
- Each cell contains the z-value (distance from viewer) of the object at that pixel position.

Scan-line

If we work scan line by scan line, as we move across a scan line, the depth changes satisfy $a\Delta x + b\Delta y + c\Delta z = 0$, noticing that the plane which contains the polygon is represented by
$ax + by + cz + d = 0$



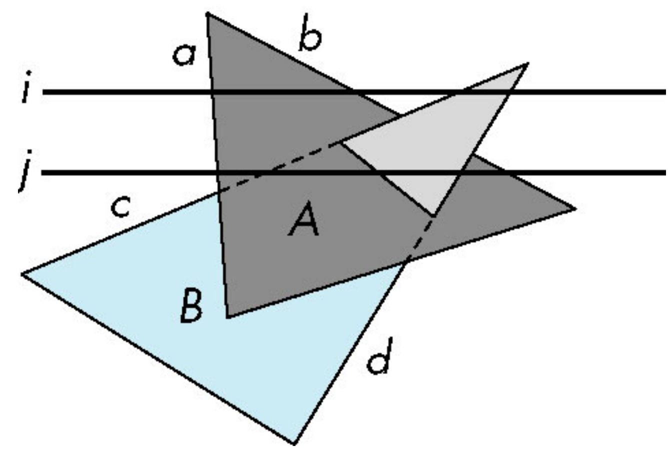
Along scan line
$\Delta y = 0$
$\Delta z = -(a/c)*\Delta x$
In screen space $\Delta x = 1$
Only computed once per polygon.

- Worst-case performance of an image-space algorithm is proportional to the number of primitives;
- Performance of z-buffer is proportional to the number of fragments generated by rasterization, which depends on the area of the rasterized polygons.

Scan-line algorithm

We can combine shading and hidden-surface removal through scan line algorithm.

- Scan line i: no need for depth information, can only be in no or one polygon.
- Scan line j: need depth information only when in more than one polygon.



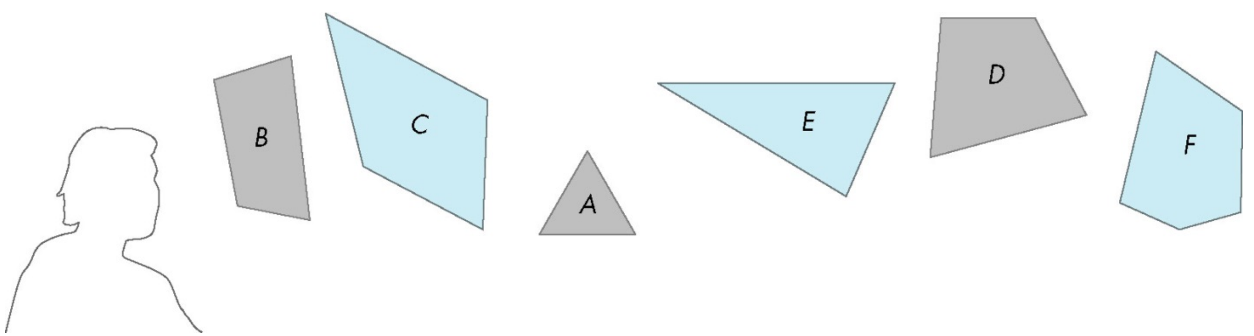
Scan-line algorithm: implementation

Need a data structure to store

- Flag for each polygon (inside/outside)
- Incremental structure for scan lines that stores which edges are encountered
- Parameters for planes

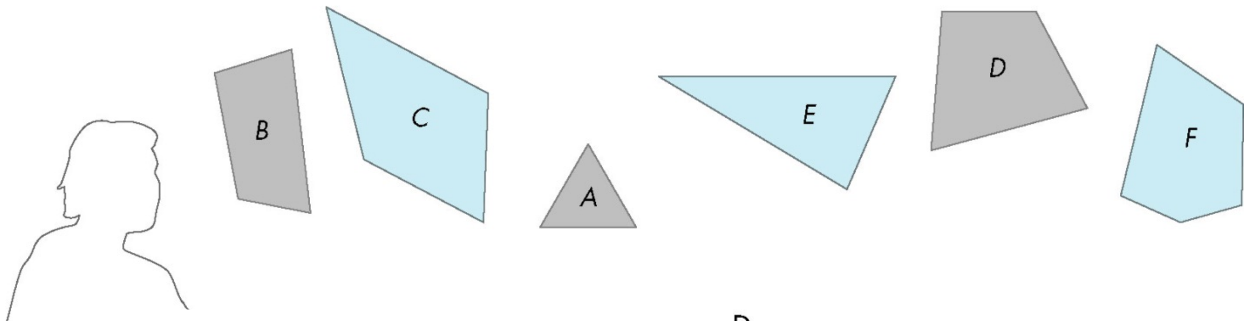
BSP tree (1)

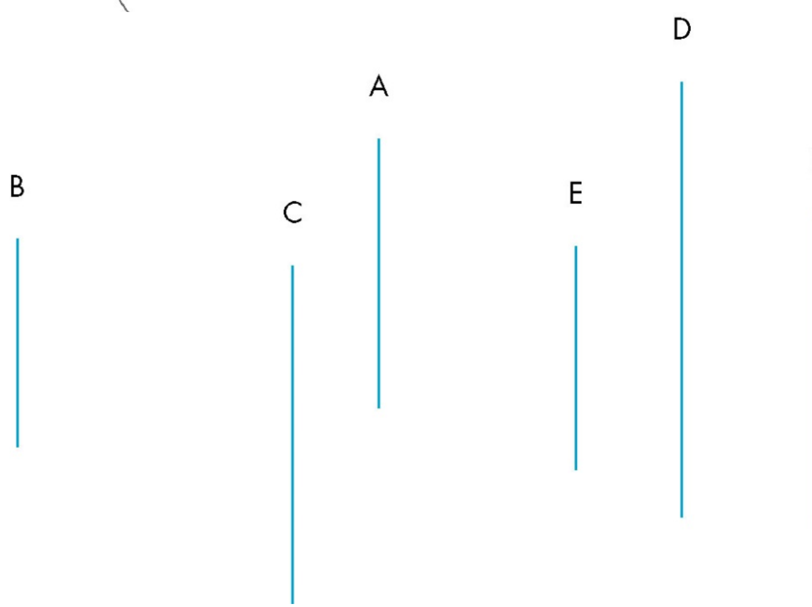
- In many real-time applications, such as games, we want to eliminate as many objects as possible within the application so that we can
  - reduce burden on pipeline
  - reduce traffic on bus
- Partition space with Binary Spatial Partition (BSP) Tree



BSP tree (2)

Consider 6 parallel planes. Plane A separates planes B and C from planes D, E and F.



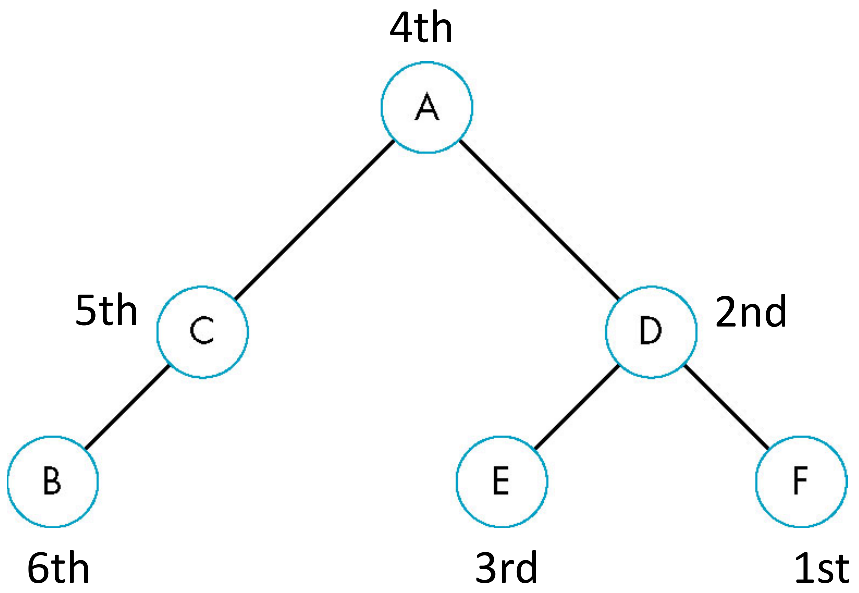


Top view

BSP tree (3)

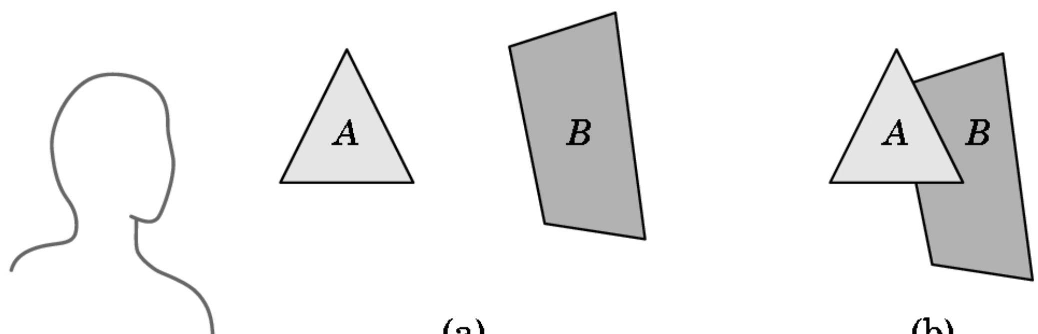
- We can continue recursively
  - plane C separates B from A
  - plane D separates E and F

- We can put this information in a BSP tree
  - use the BSP tree for visibility (inside) and occlusion (outside) testing



BSP tree

- The *painter's algorithm* for hidden-surface removal works by drawing all faces, from back to front.
- How can we get a listing of the faces in the back-to-front order?
- Put them into a binary tree and traverse the tree, but in what order?



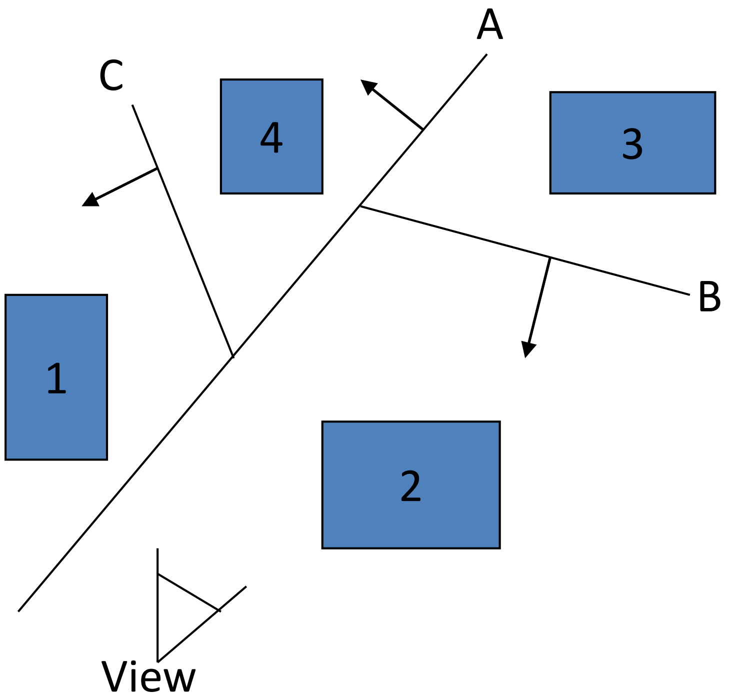
BSP tree

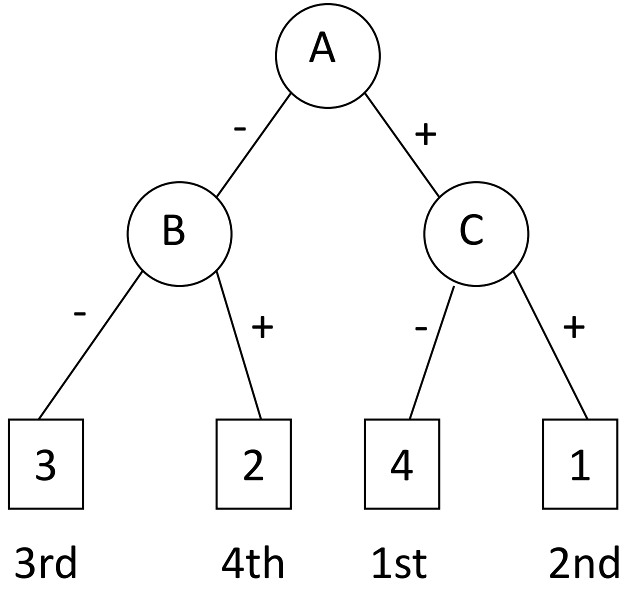
- Choose a polygon (arbitrary)
- Split the cell using the plane on which the polygon lies
  - May have to chop polygons into two if they intersect with the plane (clipping!)
- Continue until each cell contains only one polygon fragment
- Splitting planes could be chosen in other ways, but there is no efficient optimal algorithm for building BSP trees
  - BSP trees are not unique – there can be a number of alternatives for the same set of polygons
  - Optimal means minimum number of polygon fragments in a balanced tree

BSP tree: Rendering

- Observation: things on the opposite side of a splitting plane from the viewpoint cannot obscure things on the same side as the viewpoint.
- The rendering is recursive of the BSP tree.
- At each node (for back to front rendering):
  - Recurse down the side of the sub-tree that does not contain the viewpoint – Test viewpoint against the split plane to decide the polygon
  - Draw the polygon in the splitting plane – Paint over whatever has already been drawn
  - Recurse down the side of the tree containing the viewpoint

BSP tree: Rendering example

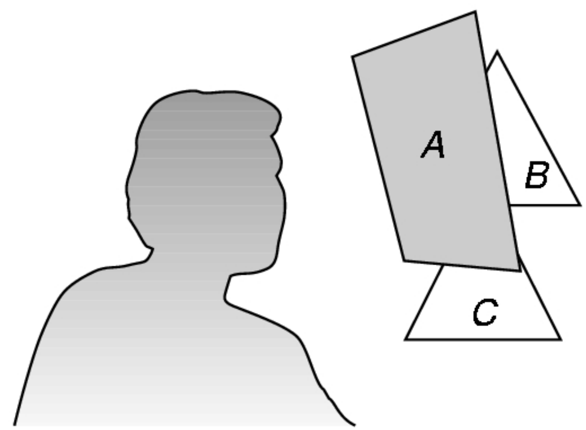




For further explanation, see
https://www.youtube.com/watch?v=YT8ya4H-Z8Q

OpenGL functions for hidden-surface removal

- How is hidden-surface removal done in OpenGL?
- OpenGL uses the *z*-buffer algorithm that saves depth information as objects/triangles are rendered so that only the front objects appear in the image.



OpenGL Z-buffer functions

- The algorithm uses an extra buffer, the z-buffer, to store depth information as geometry travels down the pipeline.

- It must be
  - 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)
    ```

OpenGL functions (1)

- OpenGL basic library provides functions for back-face removal and depth-buffer visibility testing.
- In addition, there are hidden-line removal functions for wireframe display, and scenes can be displayed with depth cueing.

OpenGL functions (2)

Face-culling functions - Back-face removal (and back-face culling) is accomplished with the functions

```
glEnable(GL_CULL_FACE);
glCullFace(GLenum);
```

- The parameter mode includes GL_BACK, GL_FRONT and GL_FRONT_AND_BACK. So front faces can be removed instead of back faces, or both front and back faces can be removed.

- The default value for `glCullFace()` is GL_BACK. Therefore, if `glEnable(GL_CULL_FACE)` is called to enable face culling without explicitly calling `glCullFace()`, back faces in the scene will be removed.

OpenGL functions (3)

Depth-buffer functions

- The first function is about the GLUT initialisation

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB |
                    GLUT_DEPTH);
```

- Depth buffer can then be initialised

```
glClear(GL_DEPTH_BUFFER_BIT);
```

- The depth-buffer visibility detection routines are activated / deactivated with

```
glEnable(GL_DEPTH_TEST);
glDisable(GL_DEPTH_TEST);
```

OpenGL functions (4)

- The depth values are normalised in the range [0.0, 1.0]. But to speed up the surface processing, a maximum depth value, maxDepth, can be specified between 0.0 and 1.0

```
glClearDepth(maxDepth);
```

- Projection co-ordinates are normalised in the range [-1.0, 1.0], and the depth values between the near and far clipping planes are further normalised in the range [0.0, 1.0] where 0.0 and 1.0 correspond to the near and far clipping planes respectively.

OpenGL functions (5)

- The function below allows adjustment of the clipping planes

  `glDepthRange(nearNormDepth, farNormDepth);`

  where the default values for the two parameters are 0.0 and 1.0 respectively, and they can have a value in the range of [0.0, 1.0].

- Another function for extra options is

  `glDepthFunc(testCondition);`

  `testCondistion` can have the following values: GL_LESS (default), GL_GREATER, GL_EQUAL, GL_NOTEQUAL, GL_LEQUAL, GL_GEQUAL, GL_NEVER (no points are processed) and GL_ALWAYS (all points are processed) for the application.

OpenGL functions (6)

- The status of the depth buffer can be set to read-only or read-write

```
glDepthMask(writeStatus);
```

- When **`writeStatus`** = GL_FALSE, the write mode in the depth buffer is disabled and only retrieval of values for comparison is possible.

- It is useful when a complex background is used with display of different foreground objects.
  - After storing the background in the depth buffer, the write mode is disabled for processing the foreground, allowing the generation of a series of frames with different foreground objects or with one object in different positions for an animated sequence.
  - It is also useful in displaying transparent objects.

OpenGL functions (7)

- **Wireframe surface-visibility functions -** A wireframe display of an object can be generated with

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

  // In this case, both visible and hidden edges are displayed.

- **Depth-cueing functions -** The brightness of an object is varied as a function of its distance to the viewing position

```
glEnable(GL_FOG);
glFog{if}[v](GL_FOG_MODE, GL_LINEAR);
```

  // linear depth function for colour in [0.0, 1.0]

```
glFog{if}[v](GL_FOG_START, minDepth);
```

  // specifies a different value for d_{min}

```
glFog{if}[v](GL_FOG_END, maxDepth);
```

  // specifies a different value for d_{max}

Summary

- Concepts of hidden-surface removal
- Clipping vs hidden-surface removal
- Important techniques for hidden-surface removal
  - Object-space and image-space approaches
  - Painter's algorithms
  - Back-face culling
  - Z-buffering
  - BSP tree
- OpenGL functions

  <u>`glutInitDisplayMode()`</u>, <u>`glClear()`</u>, `glClearDepth(maxDepth)`, `glDepthRange(nearNormDepth,farNormDepth);` `glDepthFunc(testCondition)`, `glDepthMask(writeStatus)`, `glPolygonMode()`, <u>`glEnable(GL_DEPTH_TEST)`</u>, `glEnable(GL_FOG)`, `glCullFace()`