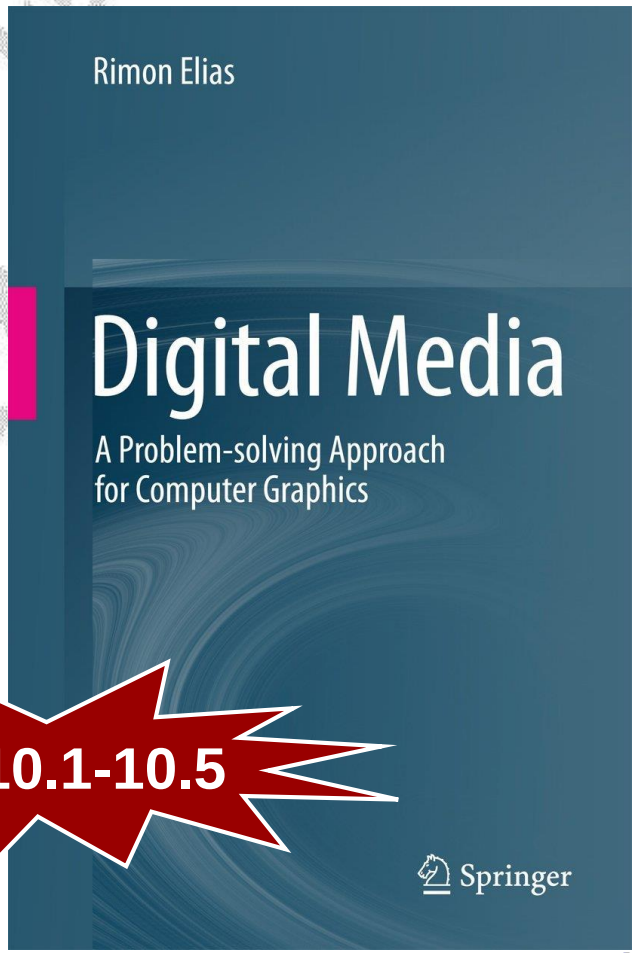
DMET 502/701
Computer Graphics

Visibility

Assoc. Prof. Dr. Rimon Elias

Contents

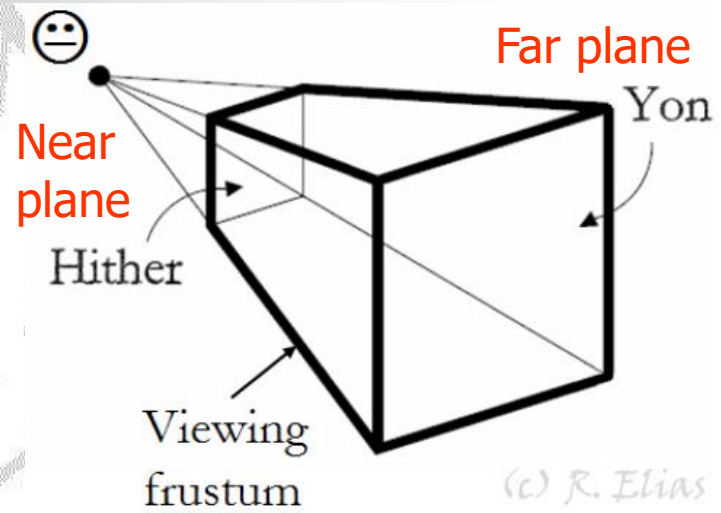
- Viewing Frustum
- Algorithms:
  - Painter's Algorithm
  - Back-face Culling
  - Z-Buffer
  - Space Partitioning
    - Binary space partitioning trees



Sec 10.1-10.5

Viewing Frustum

- **Viewing frustum** is the camera field of view.
- Any objects outside this frustum will not appear on the generated image unless they are reflected off or refracted through other objects in the frustum.



- The depth of this frustum is determined by two planes; the *hither* and the *yon* planes (also known as the *front* and *back* planes or the *near* and *far* planes with respect to the camera or the eye).

- Note that the viewing direction is perpendicular to both planes.

Will Everything in the Viewing Frustum be Visible?

- Some objects in the viewing frustum will hide other objects behind.
- Also, surfaces of a single object facing away from the camera will not appear in the generated image.

- Determining surfaces that are hidden is important
  - For getting the correct rendering results
  - For speeding up the rendering process by removing hidden surfaces from the calculations.

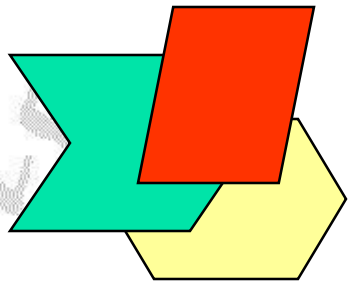
- Determining which surface is visible and which is invisible is referred to as the **visibility problem**.

Will Everything in the Viewing Frustum be Visible?

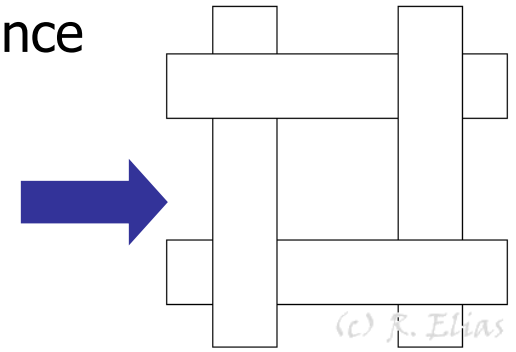
- There are many algorithms that are devised to deal with the visibility problem. For example:
  - Painter's Algorithm
  - Back-face Culling
  - Z-Buffer
  - Space Partitioning
  - Ray casting
  - Ray tracing

Painter's Algorithm

- This name refers to what a simple-minded painter would do:
  - Sort the polygons by their depth.
  - Using the sorting order, start by painting farthest polygons followed by closer polygons that will hide all or part of what has been painted first.



- Note that:
  - Since the closest polygon is painted last, it will be on the top.
  - This method requires all polygons at once in order to sort.
  - Overlapping polygons can cause the algorithm to fail.



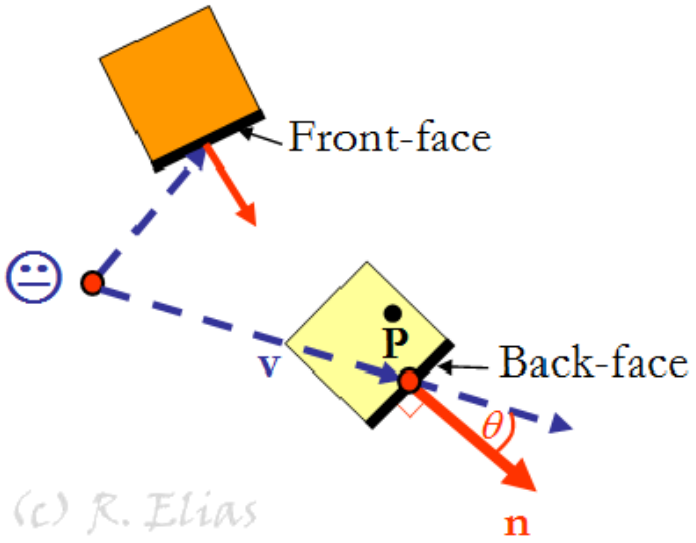
Back-face Culling

- The backside of some faces or polygons will never face the camera (or the viewer).
- Back-faces (i.e., facing away from the viewer) should be removed from the calculations.
- **Back-face culling** or removing is a method used to determine what surfaces are visible and what surfaces are hidden in order to remove the hidden surfaces.

Back-face Culling

- **How can we specify that a polygon is back-facing?**
  - Calculate the normal to its plane **n.**
  - Get the direction vector **v** from the eye to point on polygon.
  - Calculate the dot product of the two vectors:

$$\mathbf{n} \bullet \mathbf{v} = \|\mathbf{n}\| * \|\mathbf{v}\| * \cos\theta$$

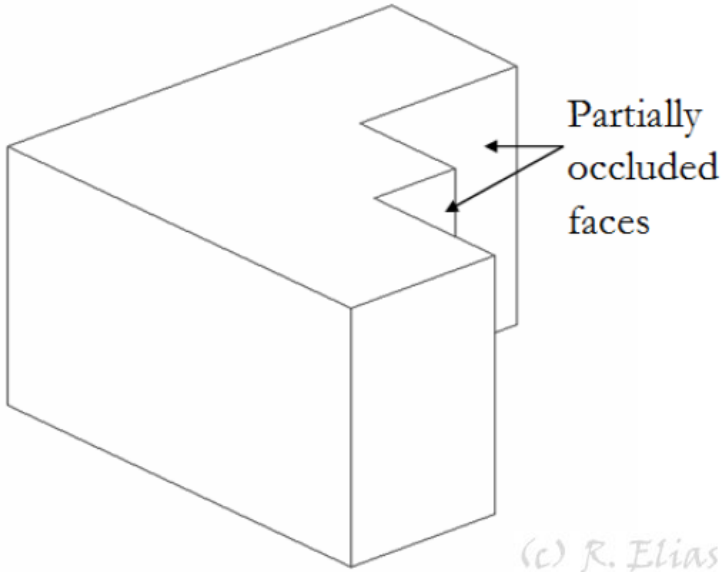


  - If the result is nonnegative ($-90^\circ \le \theta \le +90^\circ$), then the polygon is back-facing and should be culled.
  - Almost half the faces can be removed this easy way!

Back-face Culling

- Warning: Not all hidden surfaces can be removed this way.

- This example shows a situation when parts of front-faces may not be visible.
  - Back-face culling test is not enough in such a situation.



Back-face Culling: An Example

- **Example:** Assume that a virtual camera is located at $[2, 4, 6]^T$ and pointed towards the point $[7, 8, 9]^T$.
- Determine whether or not the planar surface with a surface normal $\mathbf{n} = [-2, 5, -5]^T$ is back-facing with respect to this virtual camera.
- **Answer:**

point camera

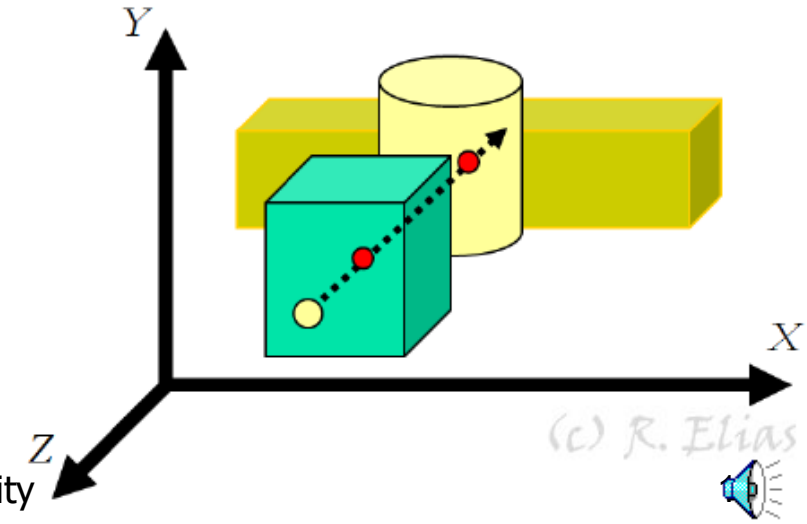
$$\mathbf{v} = \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} - \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \\ 3 \end{bmatrix}$$

$$\mathbf{n} \bullet \mathbf{v} = \begin{bmatrix} -2 \\ 5 \\ -5 \end{bmatrix} \bullet \begin{bmatrix} 5 \\ 4 \\ 3 \end{bmatrix} = (-2) \times 5 + 5 \times 4 + (-5) \times 3 = -5$$

- The surface is front-facing as the dot product results in a -ve result.

Z-buffer

- **Z-buffer** (or **depth buffer**) is used to solve the visibility problem by maintaining the nearest depth coordinates.
- Assume that the 3D points are projected onto the *xy*-plane that corresponds to the image plane.
  - The *z*-buffer is a 2D array where each element of the array corresponds to one image pixel.
  - The values stored are the *z*-coordinates of generated points.
- In case a new object projects onto a visited pixel, the new and stored *z*-coordinates are compared and the closer is kept.
- Consequently, farther objects are hidden behind closer ones.



Z-buffer

Algorithm 10.1 *Z-buffer*

Input: x_{min}, x_{max}, y_{min}, y_{max}, $polygon\_list$
Output: $zBuffer$

```
1: for (x = x_min to x_max) do
2:     for (y = y_min to y_max) do
3:         zBuffer(x, y) = 0
4:     end for
5: end for
6: for (each polygon in polygon_list) do
7:     for (each pixel [x, y]^T in the polygon) do
8:         z = z value at [x, y]^T
9:         if (z ≥ zBuffer(x, y)) then
10:            zBuffer(x, y) = z
11:        end if
12:    end for
13: end for

end
```


Z-buffer: An Example

- **Example:** Consider the *z*-buffer shown (left) where the numbers indicate the *z*-values.

- If a square (right) is to be added to the *z*-buffer, determine the final state of the *z*-buffer.

Algorithm 10.1 *Z-buffer*

Input: $x_{min}, x_{max}, y_{min}, y_{max}, polygon\_list$
Output: $zBuffer$

```
1:  for (x = x_min to x_max) do
2:      for (y = y_min to y_max) do
3:          zBuffer(x, y) = 0
4:      end for
5:  end for
6:  for (each polygon in polygon_list) do
7:      for (each pixel [x, y]^T in the polygon) do
8:          z = z value at [x, y]^T
9:          if (z ≥ zBuffer(x, y)) then
10:             zBuffer(x, y) = z
11:         end if
12:     end for
13: end for
```

end

0	0	0	0	0	0	0	0
0	0	0	0	0	2	0	0
0	0	0	0	2	2	2	0
0	0	0	2	2	2	2	2
0	0	0	0	2	2	2	0
0	0	0	0	0	2	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

			1	2	3	4	
			1	2	3	4	
			1	2	3	4	
			1	2	3	4	

Z-buffer: An Example

- **Answer:**

0	0	0	0	0	0	0	0
0	0	0	0	0	2	0	0
0	0	0	0	2	2	2	0
0	0	0	2	2	2	2	2
0	0	0	0	2	2	2	0
0	0	0	0	0	2	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

			1	2	3	4	
			1	2	3	4	
			1	2	3	4	
			1	2	3	4	

0	0	0	0	0	0	0	0
0	0	0	0	0	2	0	0
0	0	0	0	2	2	2	0
0	0	0	2	2	3	4	2
0	0	0	1	2	3	4	0
0	0	0	1	2	3	4	0
0	0	0	1	2	3	4	0
0	0	0	0	0	0	0	0

Space Partitioning

- **Space partitioning** is a process that partitions the space into two or more disjoint non-overlapping adjacent parts or sub-spaces.
- Each part or sub-space can be split recursively again into other parts.
- These parts can be organized into a tree that is referred to as a *space partitioning tree*.
- Such a tree can be traversed easily to obtain the correct order of the parts in space.

Space Partitioning

- **How to divide a space?**
  - A *hyperplane* is used to partition the space.
  - Points on one side of the hyperplane belong to one sub-space while points on the other side belong to the other sub-space.
  - Points on the hyperplane are assigned arbitrarily to any of the sides.
- **Q: What is a hyperplane?**
  - **A:** The answer depends on the dimension of space.

Space dimension	Space	Hyperplane	Hyperplane divides the space into
1D	Line	Point	2 rays
2D	Plane	Line	2 half-planes
3D	3D space	Plane	2 half-spaces

Space Partitioning

- Tree examples:
  - Binary space partitioning trees
  - Quadtrees
  - Octrees

Name	# dividing planes	# sub-spaces formed	Space dimension
BSP	1	2	nD
Quadtree	2	4	Usually 2D
Octree	3	8	Usually 3D

Binary Space Partitioning Tree

- Any space can be split or partitioned using a single plane into **two** sub-spaces.
- This splitting process can be expressed as a binary tree where a root node representing the whole space has two children representing the two sub-spaces after splitting.
- The splitting continues recursively to form a *binary space partitioning tree* or a *BSP tree*.

- The BSP tree algorithm is an efficient approach to solve the visibility problem for static scenes (while viewpoint moves).
- We will look at how to construct the BSP tree for a given scene and how to use this tree to render the scene.

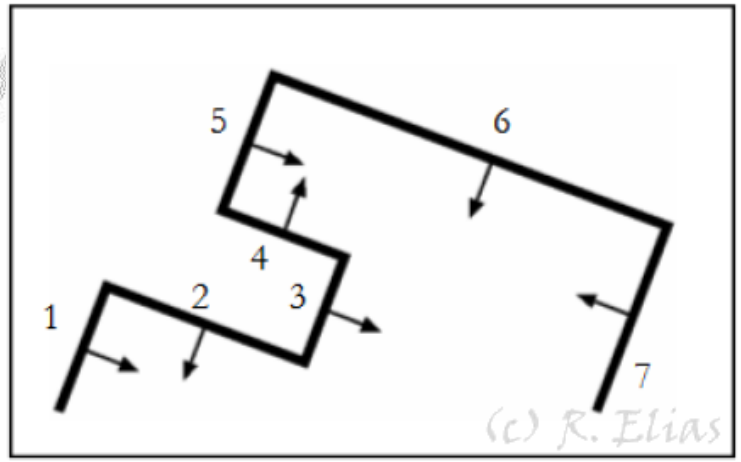
Binary Space Partitioning Tree

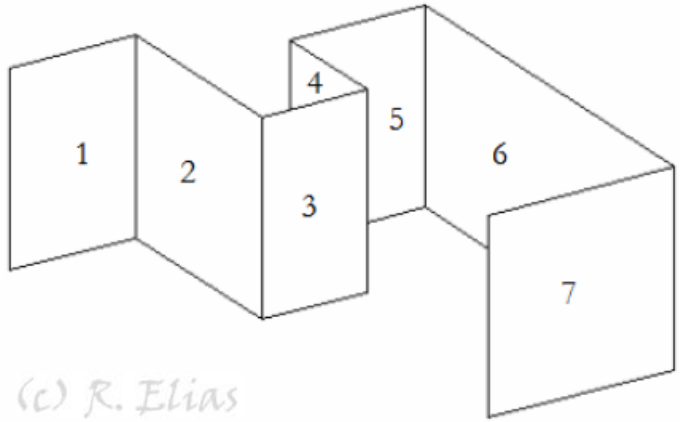
- Any space can be split or partitioned using a single plane into **two** sub-spaces.
- This splitting process can be expressed as a binary tree where a root node representing the whole space has two children representing the two sub-spaces after splitting.
- The splitting continues recursively to form a *binary space partitioning tree* or a *BSP tree*.

- The BSP tree algorithm is an efficient approach to solve the visibility problem for static scenes (while viewpoint moves).
- We will look at how to construct the BSP tree for a given scene and how to use this tree to render the scene.

Constructing BSP Trees: An Example

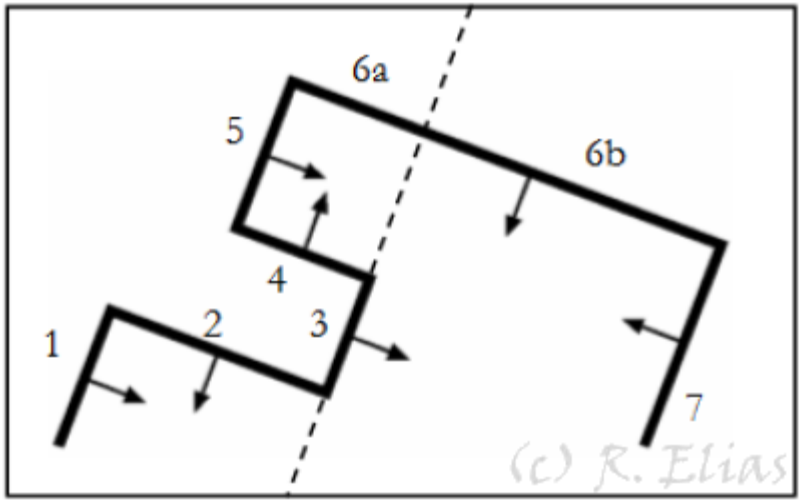
- **Example:** A 3D game is designed as consisting of a number of rooms specified by their walls. One of those rooms is shown as a top view (left) and as a 3D model (right). The normal vectors to the planes of these walls are pointed to the inside of the room as shown.
- It is required to partition this space using a BSP tree. You may use wall "3" as a root for the tree.

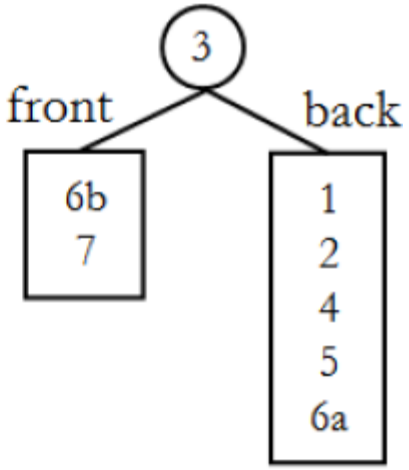




Constructing BSP Trees: An Example

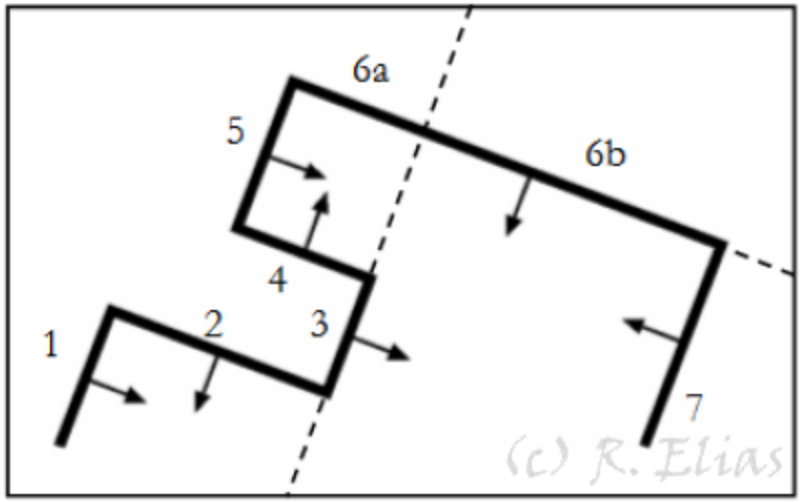
- **Answer:** Wall 3.

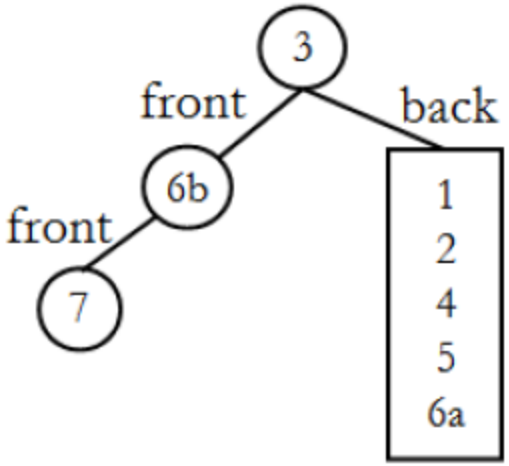




Constructing BSP Trees: An Example

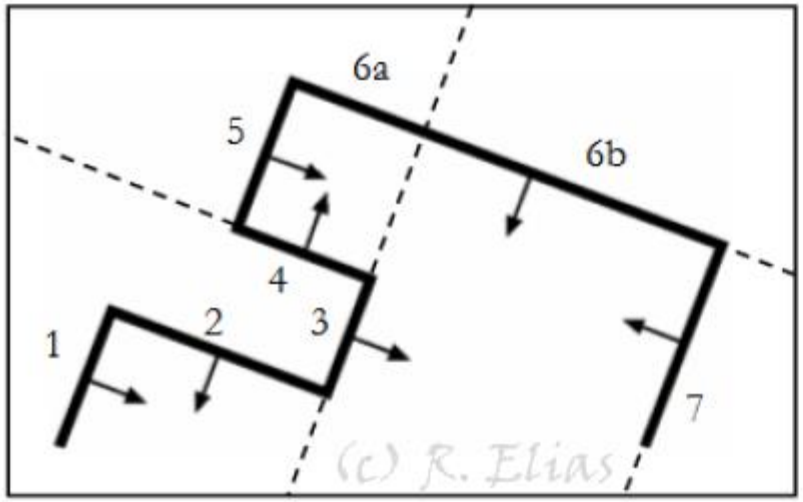
- Wall 6b.

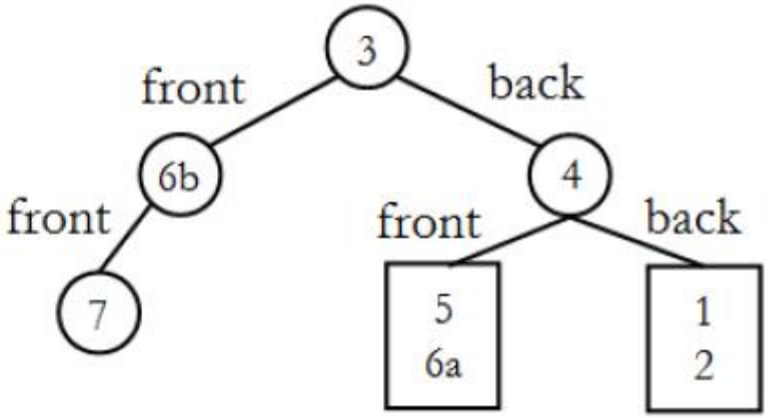




Constructing BSP Trees: An Example

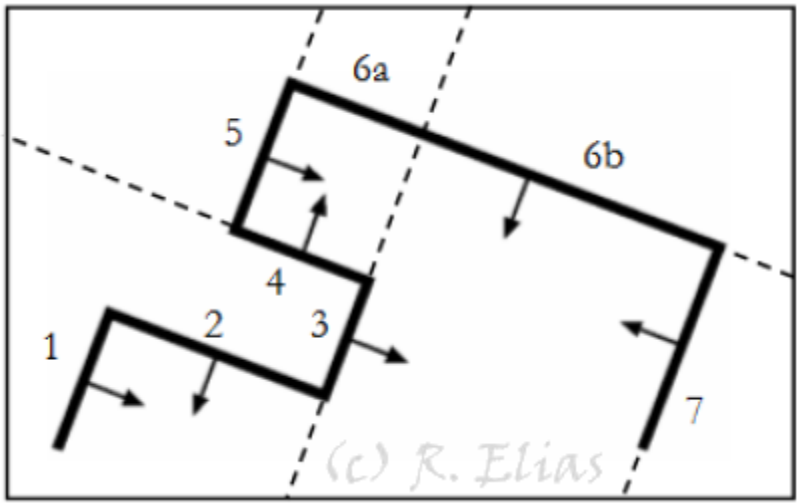
- Wall 4.

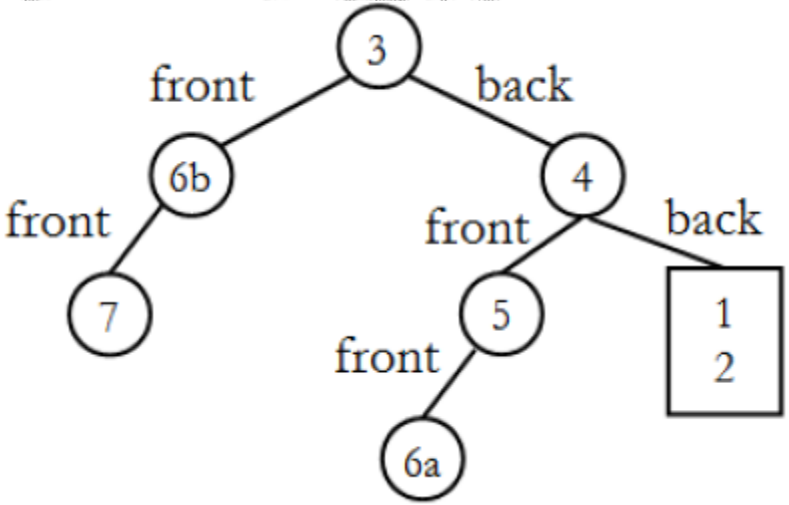




Constructing BSP Trees: An Example

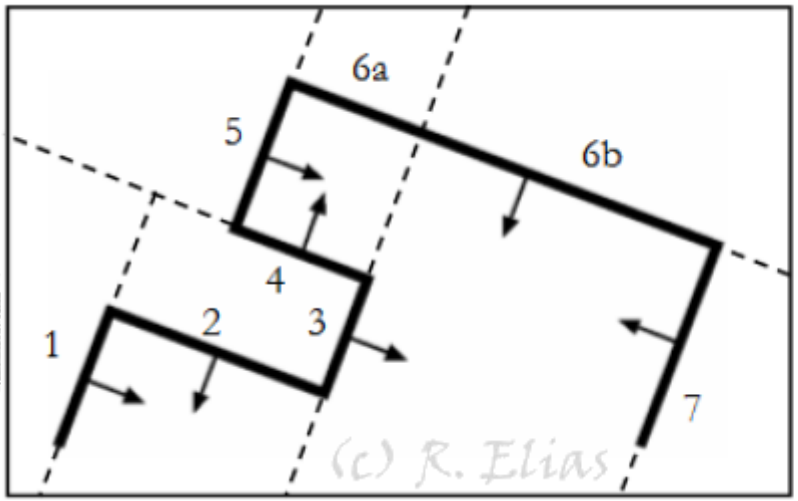
- Wall 5.

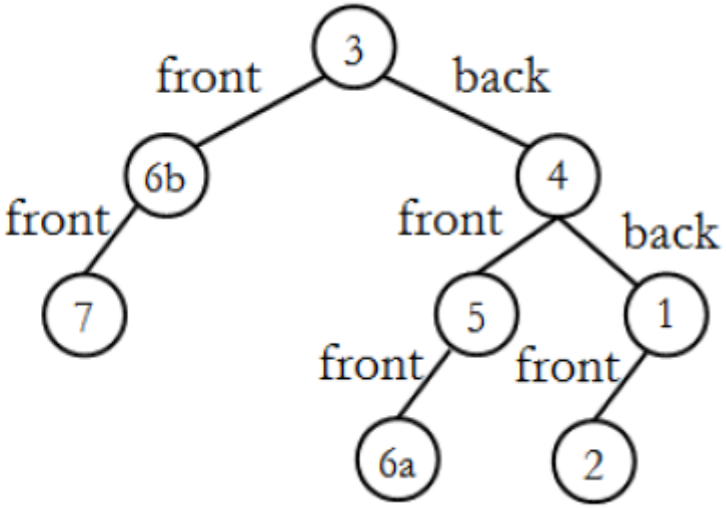




Constructing BSP Trees: An Example

- Wall 1.



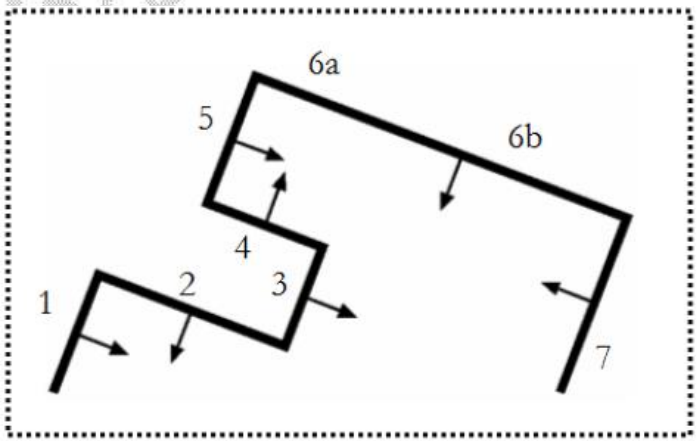


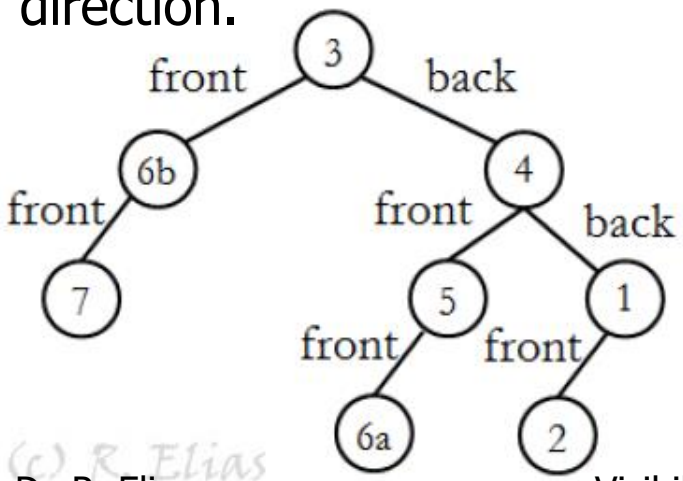
Rendering using BSP Trees

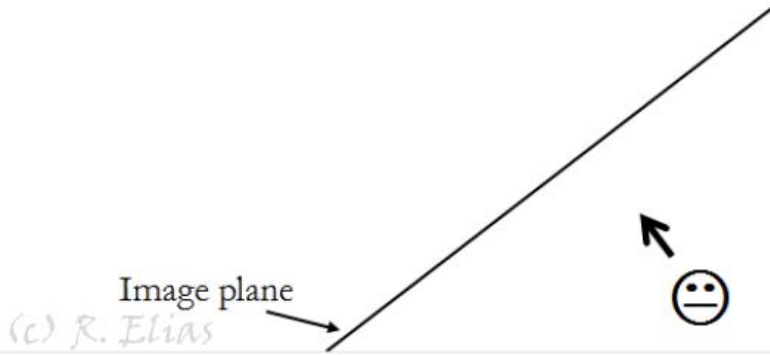
- Once a BSP tree has been built for a given static scene, it can be used to render the scene for an arbitrary viewpoint.
- The BSP tree can be traversed to lead to a priority-ordered polygon list according to the viewpoint.
- Considering the root of the tree, if the viewpoint is in the front side of the root's plane, then
  1. Display polygons in the back side.
  2. Display the root that may hide polygons in the back side.
  3. Display polygons in the front side, which may hide the root.
- The order of displaying polygons is reversed if the viewpoint is in the back side of the root.
- The algorithm is applied recursively.

Rendering using BSP Trees: An Example

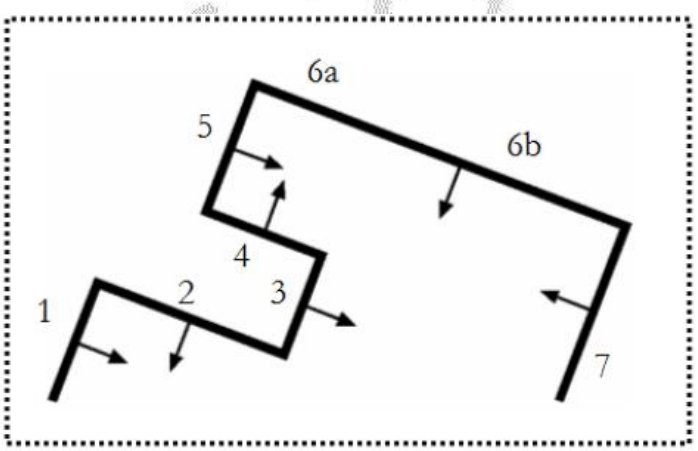
- **Example:** Shown are the top view for a scene (right) and the BSP tree (left) used to partition this space. Using the BSP tree, determine the order of displaying the polygons of the scene according to the location of the viewpoint as well as the viewing direction.



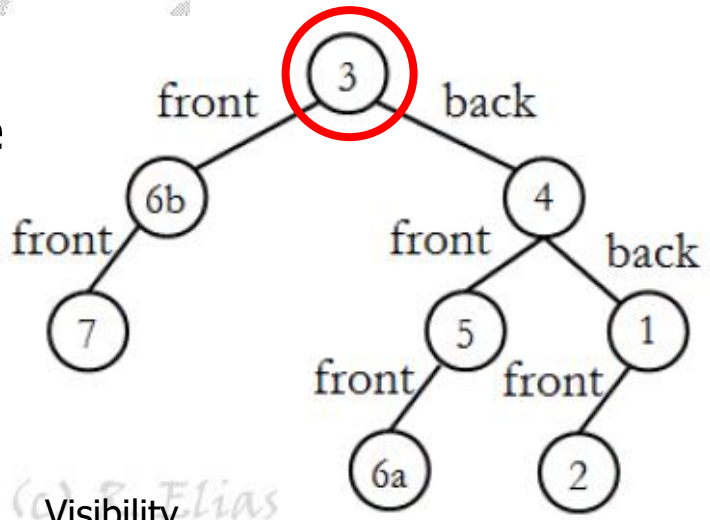




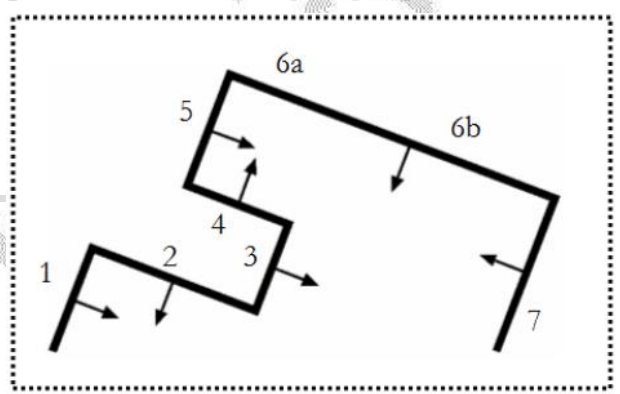
Rendering using BSP Trees: An Example



- **Answer:** The first step in the solution is to determine the relative position of the viewpoint with respect to the root node. In our case, the viewpoint is in the front sub-space of the root node. Hence,
  - Display the back sub-space
  - Display the root "3"
  - Display the front sub-space

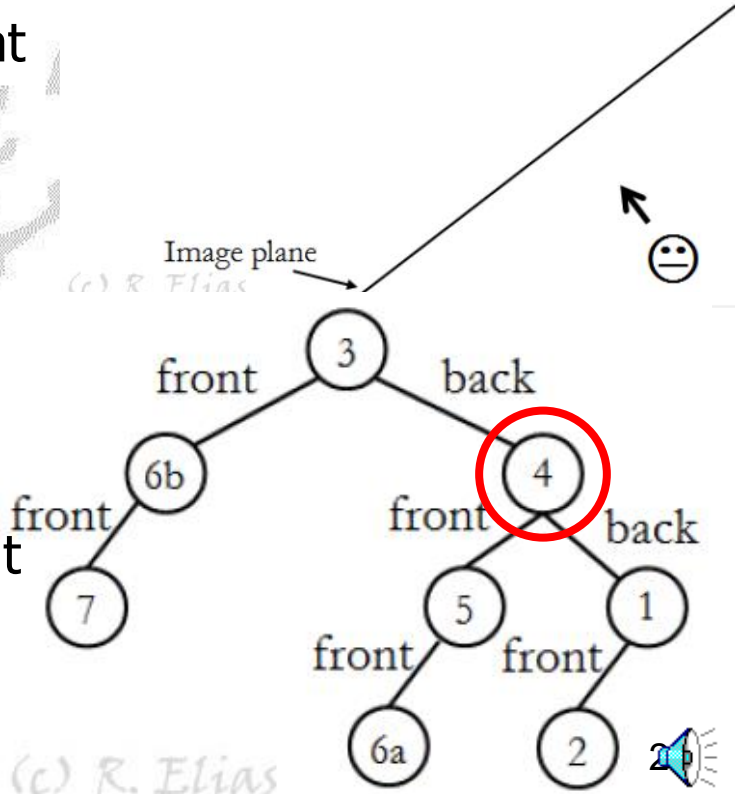


Rendering using BSP Trees: An Example

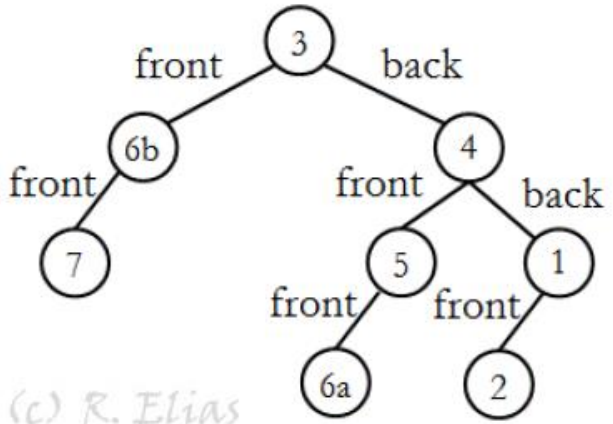


Back subtree rooted at node "4":

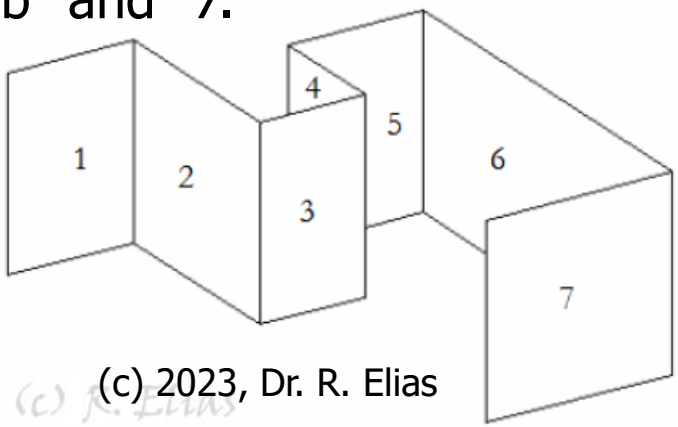
- The viewpoint is in back of node "4." Hence,
  - Display the front sub-space rooted at node "5" . Viewpoint is in front of node "5"
    - Display back sub-space: nothing
    - Display node "5"
    - Display front sub-space: "6a"
  - Display node "4"
  - Display the back sub-space rooted at node "1" …..

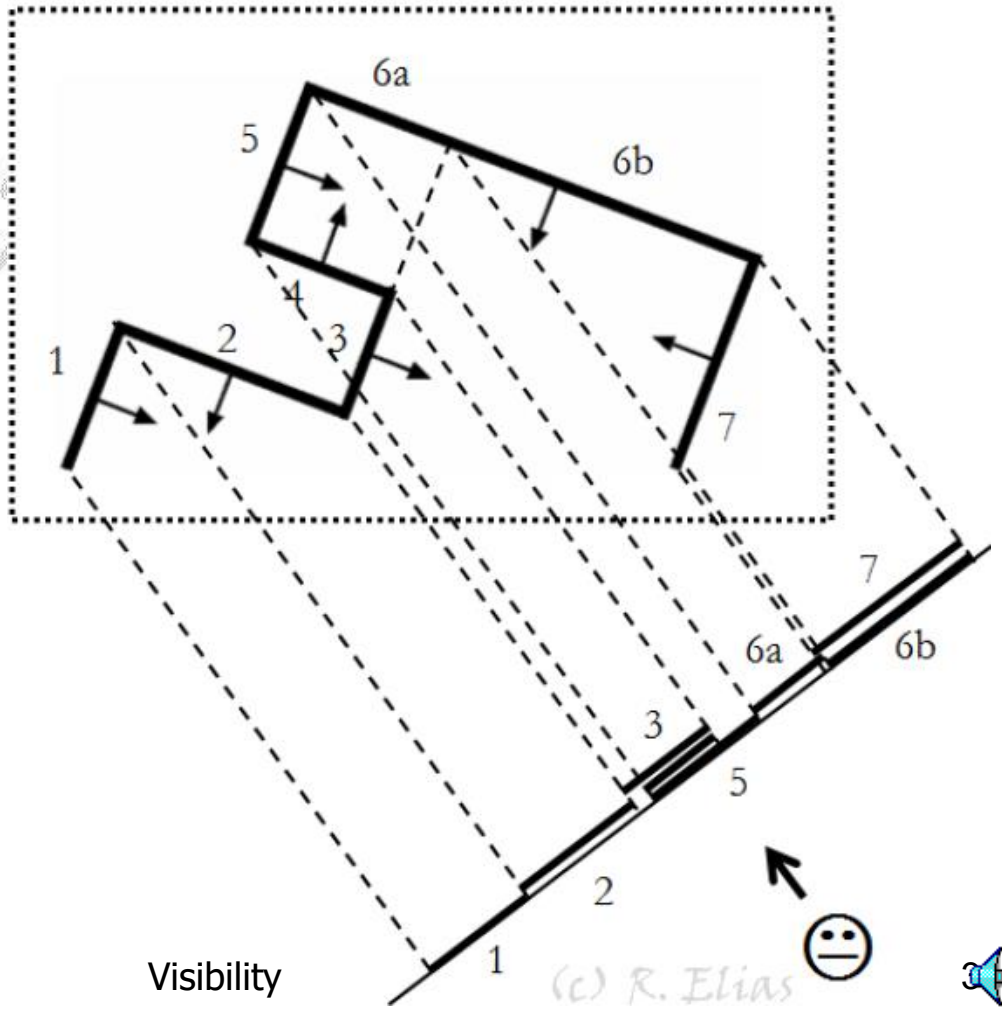


Rendering using BSP Trees: An Example



The sequence of displaying: "5," "6a," "4," "1," "2," "3," "6b" and "7."





Binary Space Partitioning Tree

- The process is time consuming and obviously it is inefficient to implement moving objects directly into the tree.
- One way to overcome this problem is by using a *z*-buffer together with the BSP tree.
- The *z*-buffer would be used in this case to merge the movable object with the background.

Summary

- Viewing Frustum
- Algorithms:
  - Painter's Algorithm
  - Back-face Culling
  - Z-Buffer
  - Space Partitioning
    - Binary space partitioning trees