Hidden Surface Algorithms

CSE 457, Autumn 2003

Graphics

http://www.cs.washington.edu/education/courses/457/03au/

Readings and References

Readings

- Sections 6.6 (esp. intro and subsections 1, 4, and 8–10), 12.1.4, *3D Computer Graphics*, Watt

Other References

- Foley, van Dam, Feiner, Hughes, Chapter 15
- I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, A characterization of ten hidden surface algorithms, *ACM Computing Surveys* 6(1): 1-55, March 1974.
  - » http://www.acm.org/pubs/citations/journals/surveys/1974-6-1/p1-sutherland/

Introduction

- In the previous lecture, we figured out how to transform the geometry so that the relative sizes will be correct if we drop the *z* component.
- But, how do we decide which geometry actually gets drawn to a pixel?
- Known as the **hidden surface elimination problem** or the **visible surface determination problem**.
- There are dozens of hidden surface algorithms.
- They can be characterized in at least three ways:
  - » Object-precision vs. image-precision (a.k.a., object-space vs. image-space)
  - » Object order vs. image order
  - » Sort first vs. sort last

Object-precision algorithms

- Basic idea:
  - » Operate on the geometric primitives themselves. (We'll use "object" and "primitive" interchangeably.)
  - » Objects typically intersected against each other
  - » Tests performed to high precision
  - » Finished list of visible objects can be drawn at any resolution
- Complexity:
  - » For n objects, can take $O(n^2)$ time to compute visibility.
  - » For an *mxm* display, have to fill in colors for m^2 pixels.
  - » Overall complexity can be $O(k_{obj}\, n^2 + k_{disp}\, m^2)$.
- Implementation:
  - » Difficult to implement
  - » Can get numerical problems

Image-precision algorithm

- Basic idea:
  - » Find the closest point as seen through each pixel
  - » Calculations performed at display resolution
  - » Does not require high precision
- Complexity:
  - » Naïve approach checks all n objects at every pixel. Then, $O(n\ m^2)$.
  - » Better approaches check only the objects that *could* be visible at each pixel. Let's say, on average, *d* objects are visible at each pixel (a.k.a., depth complexity). Then, $O(d\ m^2)$.
- Implementation:
  - » Very simple to implement.
    - Used a lot in practice.

Object order vs. image order

- Object order:
  » Consider each object only once, draw its pixels, and move on to the next object.
  » Might draw the same pixel multiple times.

- Image order:
  » Consider each pixel only once, find nearest object, and move on to the next pixel.
  » Might compute relationships between objects multiple times.

Sort first vs. sort last

- <u>Sort first:</u>
  - » Find some depth-based ordering of the objects relative to the camera, then draw back to front.
  - » Build an ordered data structure to avoid duplicating work.

- <u>Sort last:</u>
  - » Sort implicitly as more information becomes available.

Outline of Lecture

- Z-buffer
- Ray casting
- Binary space partitioning (BSP) trees

Z-buffer

- The **Z-buffer** or **depth buffer** algorithm [Catmull, 1974] is probably the simplest and most widely used.
- Here is pseudocode for the Z-buffer hidden surface algorithm:

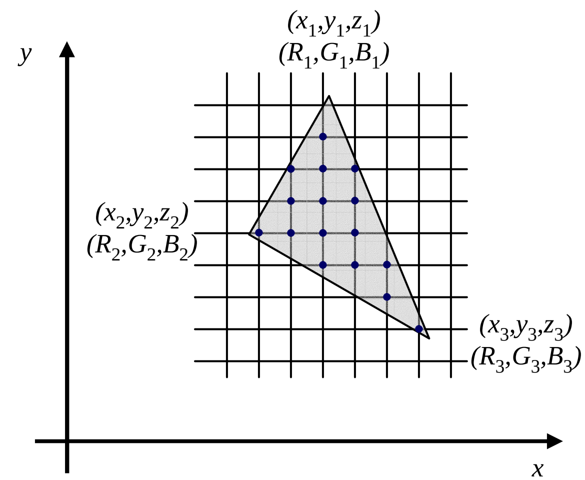
for each pixel *(i,j)* **do**
    Z-buffer *[i,j]* $\leftarrow$ *FAR*
    Framebuffer[i,j] $\leftarrow$ <background color>
end for
for each polygon *A* **do**
    for each pixel in *A* **do**
        Compute depth *z* and shade *s* of *A* at *(i,j)*
        if z > *Z-buffer [i,j]* **then**
            Z-buffer [i,j] $\leftarrow$ *z*
            Framebuffer[i,j] $\leftarrow$ *s*
        end if
    end for
end for

Q: What should FAR be set to?

Rasterization

- The process of filling in the pixels inside of a polygon is called **rasterization**.

During rasterization, the *z* value and shade *s* can be computed incrementally (ie, quickly!).



<u>Interesting fact:</u>

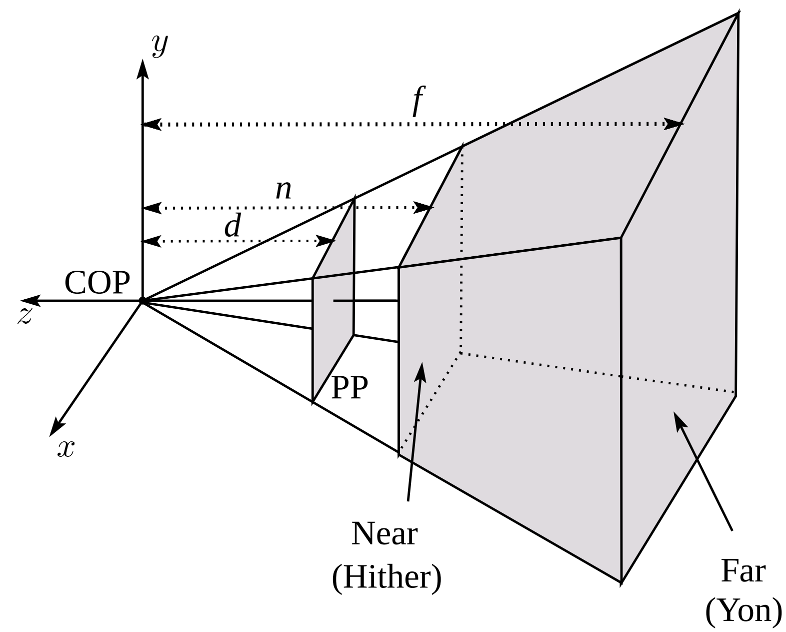
- Described as the "brute-force image space algorithm" by [SSS]
- Mentioned only in Appendix B of [SSS] as a point of comparison for <u>huge</u> memories, but written off as totally impractical.

Today, Z-buffers are commonly implemented in hardware. Tomorrow ...

http://www.cs.washington.edu/education/courses/457/03au/misc/power-trends.png

Clipping and the viewing frustum

- The center of projection and the portion of the projection plane that map to the final image form an infinite pyramid. The sides of the pyramid are **clipping planes**.
- Frequently, additional clipping planes are inserted to restrict the range of depths. These clipping planes are called the **near** and **far** or the **hither** and **yon** clipping planes.
- All of the clipping planes bound the the **viewing frustum**.



Computing *z*

- In the lecture on projections, we said that we would apply the following 3x4 projective transformation:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

- and keep the *z*-component to do Z-buffering (ie, $z' = z$)
- Strictly speaking, in order for interpolated *z* to work correctly, we actually need to map it according to:

$$z' = A + B/z$$

- For $B < 0$, is depth ordering preserved?
- In addition, we have finite precision and would like all of our *z* bits to be uniformly distributed between the clipping planes.

Computing z, cont'd

- These requirements lead to the following 4x4 projective transformation:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \dfrac{f+n}{d(f-n)} & \dfrac{2fn}{d(f-n)} \\ 0 & 0 & -1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} =$$

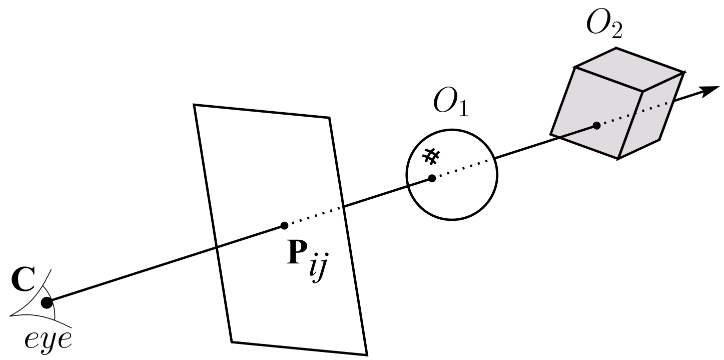
- What is z' after the perspective divide?

- What do $z=-n$ and $z=-f$ get mapped to?

Z-buffer: Analysis

- Classification?
- Easy to implement?
- Easy to implement in hardware?
- Incremental drawing calculations (uses coherence)?
- Pre-processing required?
- On-line (doesn’t need all objects before drawing begins)?
- If objects move, does it take more work than normal to draw the frame?
- If the viewer moves, does it take more work than normal to draw the frame?
- Typically polygon-based?
- Efficient shading (doesn’t compute colors of hidden surfaces)?
- Handles transparency?
- Handles refraction?

Ray casting

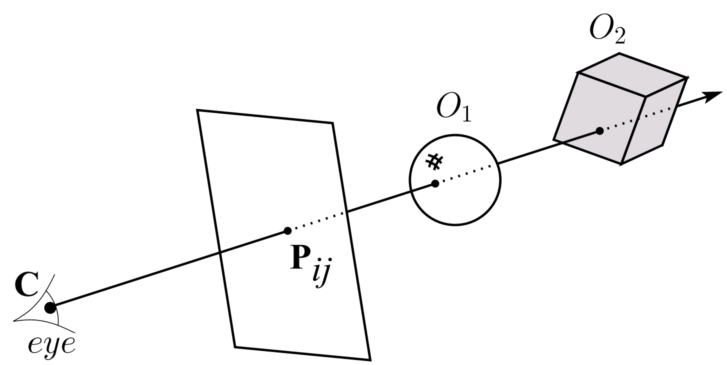


- Idea: For each pixel center $\boldsymbol{P}_{ij}$
  - » Send ray from eye point (COP), **C**, through $\boldsymbol{P}_{ij}$ into scene.
  - » Intersect ray with each object.
  - » Select nearest intersection.

Ray casting, cont.

Implementation:

- » Might parameterize each ray: $\mathbf{r}(t) = \mathbf{C} + t\,(\boldsymbol{P}_{ij} - \mathbf{C})$
- » Each object O_k returns $t_k > 0$ such that first intersection with O_k occurs at $\mathbf{r}(t_k)$.



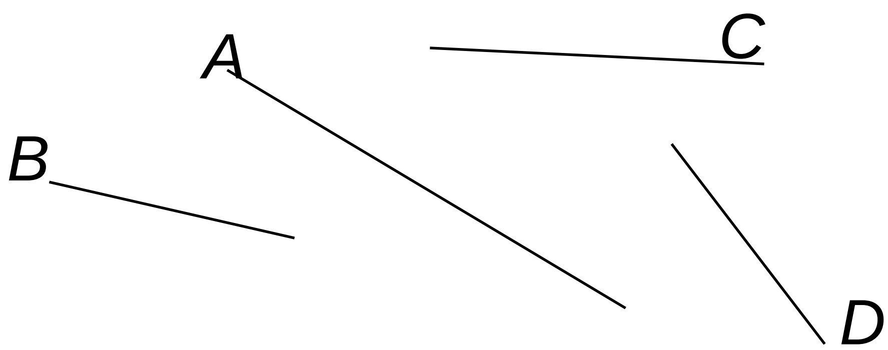
Q: Given the set $\{t_k\}$ what is the first intersection point?

Note: these calculations generally happen in <u>world</u> coordinates. No projective matrices are applied.

Ray casting: Analysis

- Classification?
- Easy to implement?
- Easy to implement in hardware?
- Incremental drawing calculations (uses coherence)?
- Pre-processing required?
- On-line (doesn’t need all objects before drawing begins)?
- If objects move, does it take more work than normal to draw the frame?
- If the viewer moves, does it take more work than normal to draw the frame?
- Typically polygon-based?
- Efficient shading (doesn’t compute colors of hidden surfaces)?
- Handles transparency?
- Handles refraction?

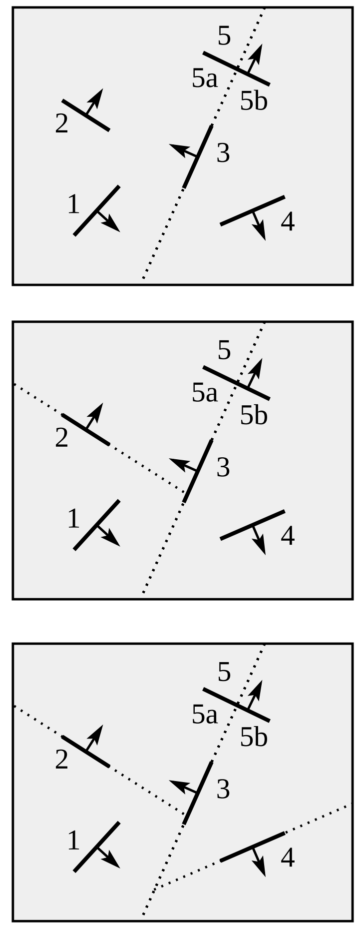
Binary-space partitioning (BSP) trees



- <u>Idea:</u>
  - » Do extra preprocessing to allow quick display from <u>any</u> viewpoint.

- <u>Key observation:</u> A polygon *A* is painted in correct order if
  - » Polygons on far side of *A* are painted first
  - » *A* is painted next
  - » Polygons in front of *A* are painted last.

BSP tree creation



BSP tree creation (cont'd)

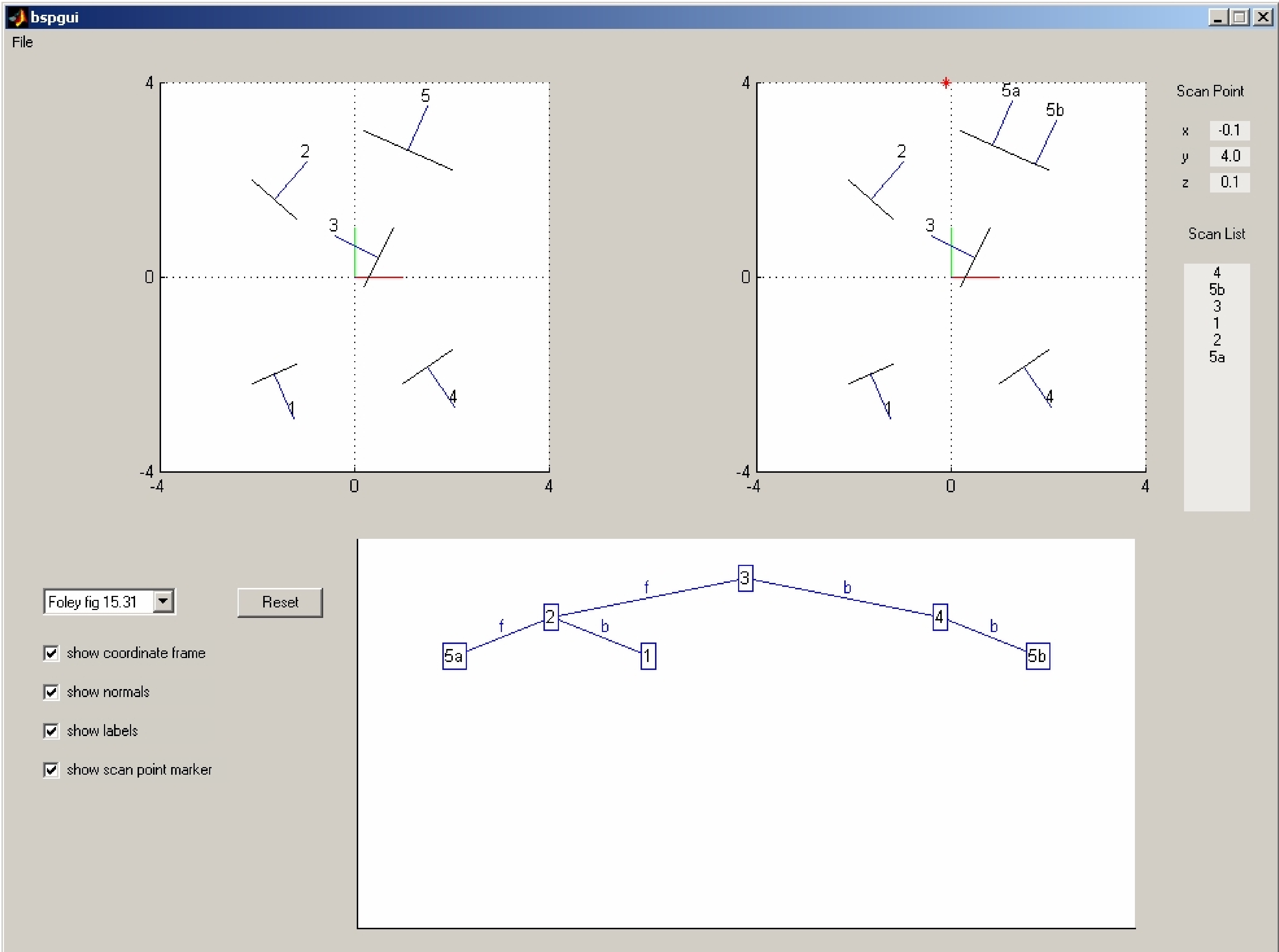
```
procedure MakeBSPTree:
takes PolygonList L
returns BSPTree
   Choose polygon A from L to serve as root
   Split all polygons in L according to A
   node ← A
   node.neg ← MakeBSPTree(Polygons on neg. side of A)
   node.pos ← MakeBSPTree(Polygons on pos. side of A)
   return node
end procedure
```

Note: Performance is improved when fewer polygons are split --- in practice, best of ~ 5 random splitting polygons are chosen.

Note: BSP is created in *world* coordinates. No projective matrices are applied.

BSP tree display

```
procedure DisplayBSPTree:
Takes BSPTree T
   if T is empty then return
   if viewer is in front half-space of T.node
       DisplayBSPTree(T. _____ )
       Draw T.node
       DisplayBSPTree(T._____)
   else
       DisplayBSPTree(T. _____)
       Draw T.node
       DisplayBSPTree(T. _____)
    end if
end procedure
```


bspgui
File
Scan Point
x -0.1
y 4.0
z 0.1
Scan List
4
5b
3
1
2
5a
Foley fig 15.31
Reset
show coordinate frame
show normals
show labels
show scan point marker

BSP trees: Analysis

- Classification?
- Easy to implement?
- Easy to implement in hardware?
- Incremental drawing calculations (uses coherence)?
- Pre-processing required?
- On-line (doesn't need all objects before drawing begins)?
- If objects move, does it take more work than normal to draw the frame?
- If the viewer moves, does it take more work than normal to draw the frame?
- Typically polygon-based?
- Efficient shading (doesn't compute colors of hidden surfaces)?
- Handles transparency?
- Handles refraction?

Cost of Z-buffering

- Z-buffering is ***the*** algorithm of choice for hardware rendering, so let's think about how to make it run as fast as possible…
- The steps involved in the Z-buffer algorithm are:
  - Send a triangle to the graphics hardware.
  - Transform the vertices of the triangle using the modeling matrix.
  - Shade the vertices.
  - Transform the vertices using the projection matrix.
  - Set up for incremental rasterization calculations
  - Rasterize and update the framebuffer according to *z*.
- What is the overall cost of Z-buffering?

Cost of Z-buffering, cont'd

We can approximate the cost of this method as:

$$k_{bus}\, v_{bus} + k_{xform}\, v_{xform} + k_{shade}\, v_{shade} + k_{setup}\, \Delta_{rast} + d\, m^2$$

Where:

k_{bus} = bus cost to send a vertex

v_{bus} = number of vertices sent over the bus

$k_{shade,xform}$ = cost of transforming and shading a vertex

$v_{shade,xform}$ = number of vertices transformed and shaded

k_{setup} = cost of setting up for rasterization

Δ_{rast} = number of triangles being rasterized

d = depth complexity (average times a pixel is covered)

m^2 = number of pixels in frame buffer

Visibility tricks for Z-buffers

Given this cost function:

$$k_{bus}\, v_{bus} + k_{xform}\, v_{xform} + k_{shade}\, v_{shade} + k_{setup}\, \Delta_{rast} + d\, m^2$$

what can we do to accelerate Z-buffering?

Summary

- What to take home from this lecture:
  - » Classification of hidden surface algorithms
  - » Understanding of Z-buffer, ray casting, and BSP tree hidden surface algorithms
  - » Familiarity with some Z-buffer acceleration strategies