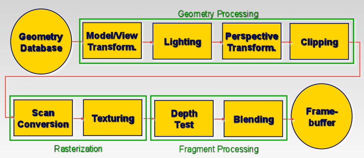
Hidden Surface Removal/ Visibility

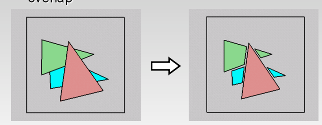
CPSC 314

The Rendering Pipeline



Occlusion

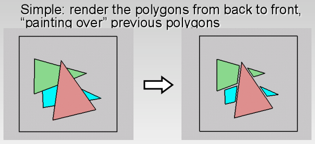
- For most interesting scenes, some polygons overlap



- To render the correct image, we need to determine which polygons occlude which

Painter's Algorithm

- Simple: render the polygons from back to front, "painting over" previous polygons



- Draw cyan, then green, then red

will this work in the general case?

Painter's Algorithm: Problems

- ***Intersecting polygons*** present a problem
- Even non-intersecting polygons can form a cycle with no valid visibility order:

Hidden Surface Removal

Object Space Methods:

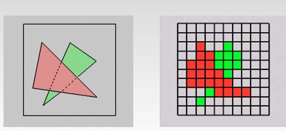
- Work in 3D before scan conversion
  - *E.g. Painter's algorithm*
- Usually independent of resolution
  - *Important to maintain independence of output device (screen/printer etc.)*

Image Space Methods:

- Work on per-pixel/per fragment basis after scan conversion
- Z-Buffer/Depth Buffer
- Much faster, but resolution dependent

The Z-Buffer Algorithm

- What happens if multiple primitives occupy the same pixel on the screen?
- Which is allowed to paint the pixel?



The Z-Buffer Algorithm

Idea: retain depth after projection transform

- Each vertex maintains z coordinate
  - *Relative to eye point*
- Can do this with canonical viewing volumes

The Z-Buffer Algorithm

Augment color framebuffer with Z-buffer

- Also called depth buffer
- Stores z value at each pixel
- At frame beginning, initialize all pixel depths to ∞
- When scan converting: interpolate depth (z) across polygon
- Check z-buffer before storing pixel color in framebuffer and storing depth in z-buffer
- don't write pixel if its z value is more distant than the z value already stored there

Z-Buffer

Store (r,g,b,z) for each pixel

- typically 8+8+8+24 bits, can be more

```
for all i,j {
 Depth[i,j] = MAX_DEPTH
 Image[i,j] = BACKGROUND_COLOUR
}
for all polygons P {
  for all pixels in P {
    if (Z_pixel < Depth[i,j]) {
      Image[i,j] = C_pixel
      Depth[i,j] = Z_pixel
    }
  }
}
```

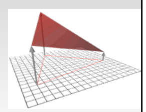

Interpolating Z

Edge walking

- Just interpolate Z along edges and across spans

Barycentric coordinates

- Interpolate z like other parameters
- E.g. color



The Z-Buffer Algorithm (mid-70's)

History:

- Object space algorithms were proposed when memory was expensive
- First 512x512 framebuffer was >$50,000!

Radical new approach at the time

- The big idea:
  - *Resolve visibility independently at each pixel*

Depth Test Precision

- Reminder: projective transformation maps eye-space *z* to generic *z*-range (NDC)
- Simple example:

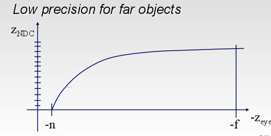
$$T\left(\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}\right) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Thus:

$$z_{NDC} = \frac{a \cdot z_{eye} + b}{z_{eye}} = a + \frac{b}{z_{eye}}$$

Depth Test Precision

- Therefore, depth-buffer essentially stores 1/z, rather than z!
- Issue with integer depth buffers
  - *High precision for near objects*
  - *Low precision for far objects*



Depth Test Precision

- Low precision can lead to depth fighting for far objects
  - *Two different depths in eye space get mapped to same depth in framebuffer*
  - *Which object "wins" depends on drawing order and scan-conversion*
- Gets worse for larger ratios $f:n$
  - Rule of thumb: $f:n < 1000$ for 24 bit depth buffer
- With 16 bits cannot discern millimeter differences in objects at 1 km distance

Z-Buffer Algorithm Questions

- How much memory does the Z-buffer use?
- Does the image rendered depend on the drawing order?
- Does the time to render the image depend on the drawing order?
- How does Z-buffer load scale with visible polygons? with framebuffer resolution?

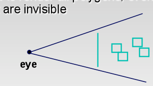
Z-Buffer Pros

- Simple!!!
- Easy to implement in hardware
  - *Hardware support in all graphics cards today*
- Polygons can be processed in arbitrary order
- Easily handles polygon interpenetration

Z-Buffer Cons

Poor for scenes with high depth complexity

- Need to render all polygons, even if most are invisible



Shared edges are handled inconsistently

- *Ordering dependent*

Z-Buffer Cons

Requires lots of memory
- (e.g. 1280x1024x32 bits)

Requires fast memory
- Read-Modify-Write in inner loop

Hard to simulate transparent polygons
- We throw away color of polygons behind closest one
- Works if polygons ordered back-to-front
  - *Extra work throws away much of the speed advantage*

Object Space Algorithms

Determine visibility on object or polygon level
- Using camera coordinates

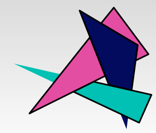
Resolution independent
- Explicitly compute visible portions of polygons

Early in pipeline
- After clipping

Requires depth-sorting
- Painter's algorithm
- BSP trees

Object Space Visibility Algorithms

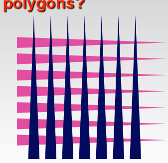
- Early visibility algorithms computed the set of visible ***polygon fragments*** directly, then rendered the fragments to a display:



Object Space Visibility Algorithms

What is the minimum worst-case cost of computing the fragments for a scene composed of *n* polygons?

Answer:
$O(n^2)$



Object Space Visibility Algorithms

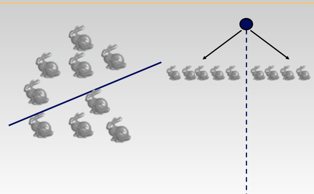
- So, for about a decade (late 60s to late 70s) there was intense interest in finding efficient algorithms for hidden surface removal
- We'll talk about one:
  - Binary Space Partition (BSP) Trees
  - *Still in use today for ray-tracing, and in combination with z-buffer*

Binary Space Partition Trees (1979)

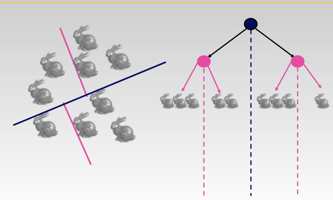
BSP Tree: partition space with binary tree of planes
- Idea: divide space recursively into half-spaces by choosing splitting planes that separate objects in scene
- Preprocessing: create binary tree of planes
- Runtime: correctly traversing this tree enumerates objects from back to front

Creating BSP Trees: Objects

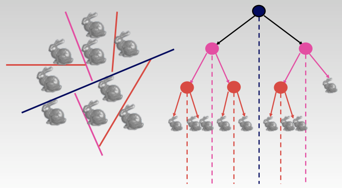
Creating BSP Trees: Objects



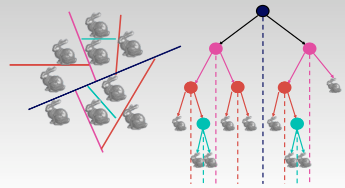
Creating BSP Trees: Objects



Creating BSP Trees: Objects



Creating BSP Trees: Objects

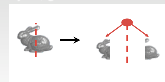


Splitting Objects

No bunnies were harmed in previous example

But what if a splitting plane passes through an object?

- Split the object; give half to each node



Traversing BSP Trees

Tree creation independent of viewpoint

- Preprocessing step

Tree traversal uses viewpoint

- Runtime, happens for many different viewpoints

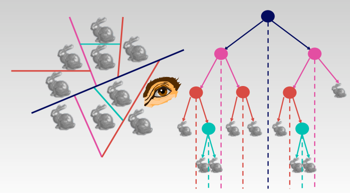
Each plane divides world into near and far

- For given viewpoint, decide which side is near and which is far
  - *Check which side of plane viewpoint is on independently for each tree vertex*
  - *Tree traversal differs depending on viewpoint!*
- Recursive algorithm
  - *Recurse on far side*
  - *Draw object*
  - *Recurse on near side*

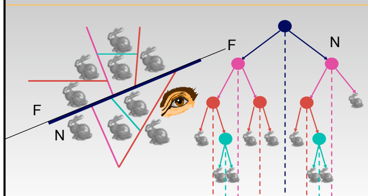
Traversing BSP Trees

```
renderBSP(BSPtree *T)
    BSPtree *near, *far;
    if (eye on left side of T->plane)
     near = T->left; far = T->right;
    else
     near = T->right; far = T->left;
    renderBSP(far);
    if (T is a leaf node)
     renderObject(T)
  renderBSP(near);
```

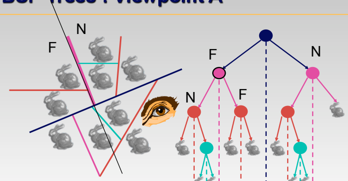

BSP Trees : Viewpoint A



BSP Trees : Viewpoint A

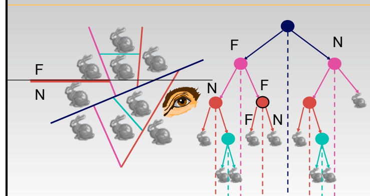


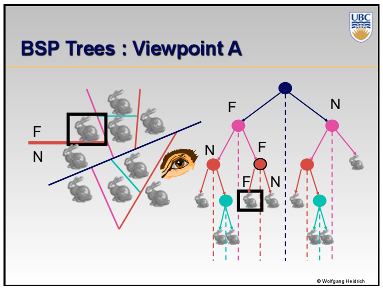
BSP Trees : Viewpoint A

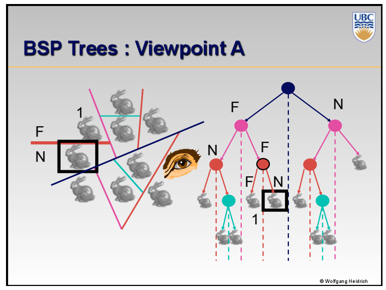


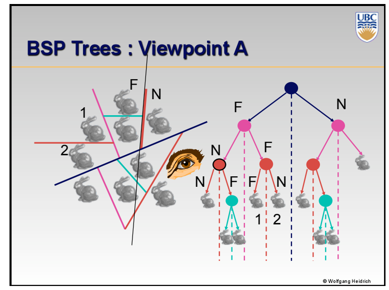
- decide independently at each tree vertex
- not just left or right child!

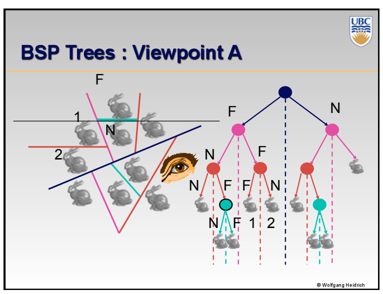
BSP Trees : Viewpoint A

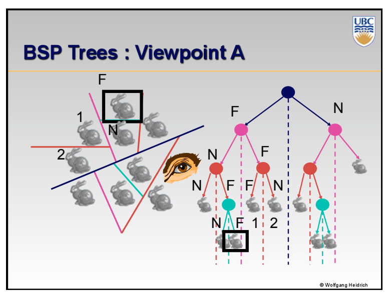


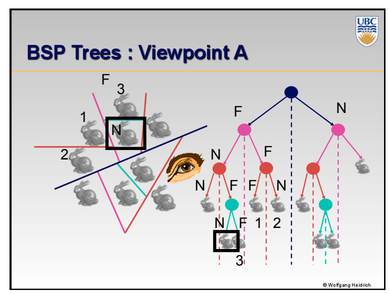

BSP Trees : Viewpoint A

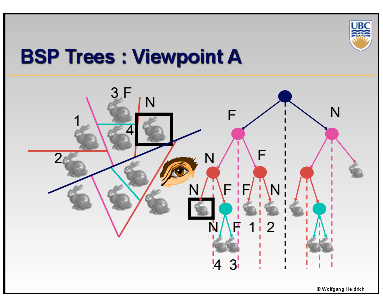

BSP Trees : Viewpoint A

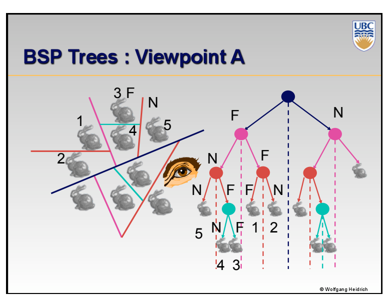

BSP Trees : Viewpoint A

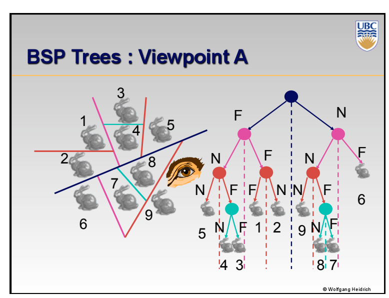

BSP Trees : Viewpoint A


BSP Trees : Viewpoint A

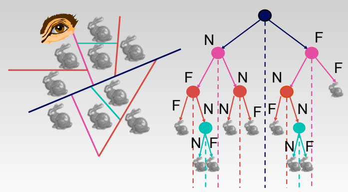

BSP Trees : Viewpoint A


BSP Trees : Viewpoint A

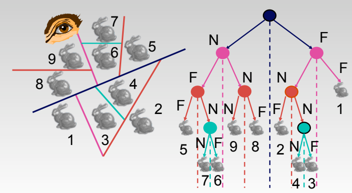

BSP Trees : Viewpoint A


BSP Trees : Viewpoint A

BSP Trees : Viewpoint B



BSP Trees : Viewpoint B



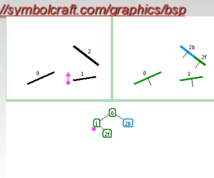
BSP Tree Traversal: Polygons

- Split along the plane defined by any polygon from scene
- Classify all polygons into positive or negative half-space of the plane
  - *If a polygon intersects plane, split polygon into two and classify them both*
- Recurse down the negative half-space
- Recurse down the positive half-space

BSP Demo

Useful demo:

http://symbolcraft.com/graphics/bsp



Summary: BSP Trees

Pros:

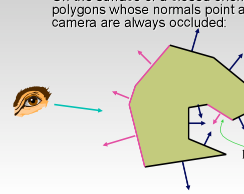
- Simple, elegant scheme
- Correct version of painter's algorithm back-to-front rendering approach
- Still very popular for video games (but getting less so)

Cons:

- Slow(ish) to construct tree: O(n log n) to split, sort
- Splitting increases polygon count: O(n^2) worst-case
- Computationally intense preprocessing stage restricts algorithm to static scenes

Optimization using Visibility: Back-Face Culling

- On the surface of a closed orientable manifold, polygons whose normals point away from the camera are always occluded:



note: backface culling alone doesn't solve the hidden-surface problem!

Back-Face Culling

Not rendering backfacing polygons improves performance

- Reduces by about half the number of polygons to be considered for each pixel
- Optimization when appropriate

Back-Face Culling

Most objects in scene are typically "solid"

rigorously: orientable closed manifolds

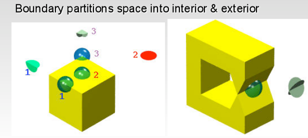
- **Orientable**: must have two distinct sides
  - *Cannot self-intersect*
  - *A sphere is orientable since has two sides, 'inside' and 'outside'.*
  - *A Mobius strip or a Klein bottle is not orientable*
- **Closed**: surface encloses a volume
  - *Sphere is closed manifold*
  - *Plane is not*

Back-Face Culling

Most objects in scene are typically "solid"

Rigorously: orientable closed manifolds

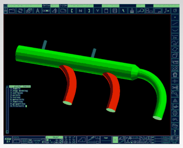
- **Manifold**: local neighborhood of all points isomorphic to disc
- Boundary partitions space into interior & exterior



Manifold

Examples of* manifold *objects:

- Sphere
- Torus
- Well-formed CAD part

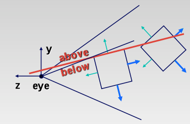


Back-Face Culling

Examples of non-manifold objects:

- A single polygon
- A terrain or height field
- Polyhedron w/ missing face
- Anything with cracks or holes in boundary
- One-polygon thick lampshade

Back-face Culling: VCS



first idea: cull if $N_Z < 0$

sometimes misses polygons that should be culled

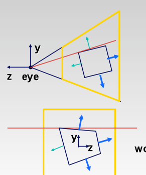
better idea: cull if eye is below polygon plane

Back-face Culling: NDCS

VCS

NDCS

eye



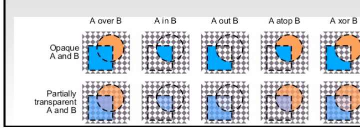
works to cull if $N_Z > 0$

Blending

Blending

How might you combine multiple elements?

- New color **A**, old color **B**



Premultiplying Colors

Specify opacity with alpha channel: (r,g,b,α)

- α=1: opaque, α=.5: translucent, α=0: transparent

A over B

- **C** = α**A** + (1-α)**B**

But what if B is also partially transparent?

- **C** = α**A** + (1-α) β**B** = α**A** + β**B** - α β**B**
- γ = β + (1-β)α = β + α – αβ
  - *3 multiplies, different equations for alpha vs. RGB*

Premultiplying by alpha

- **C' =** γ **C, B' =** β**B, A' =** α**A**
- **C' = B' + A' -** α**B'**
- γ = β + α – αβ
  - *1 multiply to find C, same equations for alpha and RGB*

OpenGL Blending

In OpenGL:

- Enable blending
  - *glEnable(GL_BLEND)*
- Specify alpha channel for colors
  - *glColor4f(r, g, b, alpha)*
- Specify blending function
  - *E.g: glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPH)*
    - C= alpha_new*Cnew + (1-alpha_new)*Cold

OpenGL Blending

Caveats:

- Note: alpha blending is an order-dependent operation!
  - *It matters which object is drawn first AND*
  - *Which surface is in front*
- For 3D scenes, this makes it necessary to keep track of rendering order explicitly
  - *Possibly also viewpoint-dependent!*
    - E.g. always draw "back" surface first
- Also note: interaction with z-buffer

Double Buffer

Double Buffering

Framebuffer:

- Piece of memory where the final image is written
- Problem:
  - *The display needs to read the contents, cyclically, while the GPU is already working on the next frame*
  - *Could result in display of partially rendered images on screen*
- Solution:
  - *Have TWO buffers*
    - One is currently displayed (front buffer)
    - One is rendered into for the next frame (back

Double Buffering

Front/back buffer:

- Each buffer has both color channels and a depth channel
  - *Important for advanced rendering algorithms*
  - *Doubles memory requirements!*

Switching buffers:

- At end of rendering one frame, simply exchange the pointers to the front and back buffer
- GLUT toolkit: glutSwapBuffers() function
  - *Different functions under windows/X11 if not using GLUT*

Picking/Object Selection

Interactive Object Selection

Move cursor over object, click

- How to decide what is below?

Ambiguity

- Many 3D world objects map to same 2D point

Common approaches

- Manual ray intersection
- Bounding extents
- Selection region with hit list (OpenGL support)

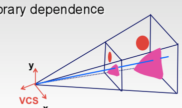
Manual Ray Intersection

Do all computation at application level

- Map selection point to a ray
- Intersect ray with all objects in scene.

Advantages

- No library dependence



Manual Ray Intersection

Do all computation at application level

- Map selection point to a ray
- Intersect ray with all objects in scene.

Advantages

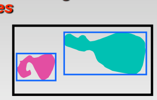
- No library dependence

Disadvantages

- Difficult to program
- Slow: work to do depends on total number and complexity of objects in scene

Bounding Extents

Keep track of axis-aligned bounding rectangles



Advantages

- Conceptually simple
- Easy to keep track of boxes in world space

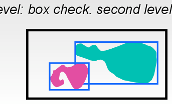
Bounding Extents

Disadvantages

- Low precision
- Must keep track of object-rectangle relationship

Extensions

- Do more sophisticated bound bookkeeping
  - *First level: box check. second level: object check*



OpenGL Picking

"Render" image in picking mode

- Pixels are never written to framebuffer
- Only store IDs of objects that would have been drawn

Procedure

- Set unique ID for each pickable object
- Call the regular sequence of glBegin/glVertex/glEnd commands
  - *If possible, skip glColor, glNormal, glTexCoord etc. for performance*

Select/Hit

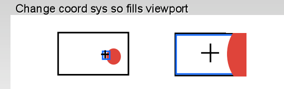
OpenGL support

- Use small region around cursor for viewport
- Assign per-object integer keys (names)
- Redraw in special mode
- Store hit list of objects in region
- Examine hit list

Viewport

Small rectangle around cursor

- Change coord sys so fills viewport



Why rectangle instead of point?

- People aren't great at positioning mouse
  - *Fitts's Law: time to acquire a target is function of the distance to and size of the target*
- Allow several pixels of slop

Viewport

Tricky to compute

- Invert viewport matrix, set up new orthogonal projection

Simple utility command

- gluPickMatrix(x,y,w,h,viewport)
  - *x,y: cursor point*
  - *w,h: sensitivity/slop (in pixels)*
- Push old setup first, so can pop it later

Render Modes

glRenderMode(mode)

- GL_RENDER: normal color buffer
  - *default*
- GL_SELECT: selection mode for picking
- (GL_FEEDBACK: report objects drawn)

Name Stack

- "names" are just integers

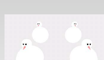
  glInitNames()
- flat list

  glLoadName(name)
- or hierarchy supported by stack

  glPushName(name), glPopName
  - *Can have multiple names per object*
  - *Helpful for identifying objects in a hierarchy*

Hierarchical Names Example

```
for(int i = 0; i < 2; i++) {
  glPushName(i);
  for(int j = 0; j < 2; j++) {
    glPushMatrix();
    glPushName(j);
    glTranslatef(i*10.0,0,j * 10.0);
      glPushName(HEAD);
      glCallList(snowManHeadDL);
      glLoadName(BODY);
      glCallList(snowManBodyDL);
      glPopName();
    glPopName();
    glPopMatrix();
  }
  glPopName();
}
```



http://www.lighthouse3d.com/opengl/picking/

Hit List

- glSelectBuffer(int buffersize, GLuint *buffer)
  - *Where to store hit list data*
- If object overlaps with pick region, create **hit record**
- Hit record
  - *Number of names on stack*
  - *Minimum and minimum depth of object vertices*
    - Depth lies in the z-buffer range [0,1]
    - Multiplied by 2^32 -1 then rounded to nearest int
  - *Contents of name stack (bottom entry first)*

Using OpenGL Picking

Example code:

```
int numHitEntries;
GLuint buffer[1000];
glSelectBuffer( 1000, buffer );
glRenderMode( GL_SELECT );
drawStuff(); // includes name stack calls
numHitEntries= glRenderMode( GL_RENDER );
// now analyze numHitEntries different hit records
// in the selection buffer
...
```

Integrated vs. Separate Pick Function

Integrate: use same function to draw and pick

- Simpler to code
- Name stack commands ignored in render mode

Separate: customize functions for each

- Potentially more efficient
- Can avoid drawing unpickable objects

Select/Hit

Advantages

- Faster
  - *OpenGL support means hardware accel*
  - *Only do clipping work, no shading or rasterization*
- Flexible precision
  - *Size of region controllable*
- Flexible architecture
  - *Custom code possible, e.g. guaranteed frame rate*

Disadvantages

- More complex