Clarification: Blinn-Phong Model
• only change vs Phong model is to have the specular calculation to use \((n \cdot n)\) instead of \((n \cdot r)\)
• full Blinn-Phong lighting model equation has ambient, diffuse, specular terms
  \[ I_{\text{total}} = k_a I_{\text{ambient}} + \sum_{i=1}^{n} I_i (k_d (n \cdot l_i) + k_s (n \cdot h_i) R_{\text{shiny}}) \]
• just like full Phong model equation
  \[ I_{\text{total}} = k_a I_{\text{ambient}} + \sum_{i=1}^{n} I_i (k_d (n \cdot l_i) + k_s (v \cdot r_i) R_{\text{shiny}}) \]

Reading for Hidden Surfaces
• FCG Sect 8.2.3 Z-Buffer
• FCG Sect 12.4 BSP Trees
  • (8.1, 8.2 2nd ed)
• FCG Sect 3.4 Alpha Compositing
  • (N/A 2nd ed)

Analytic Visibility Algorithms
• early visibility algorithms computed the set of visible polygon fragments directly, then rendered the fragments to a display:
  
  Painter’s Algorithm
• simple: render the polygons from back to front, “painting over” previous polygons
  • draw blue, then green, then orange
  • 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:

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
### 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

```c
renderBSP(BSPtree *T)

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);
```

query: given a viewpoint, produce an ordered list of (possibly split) objects from back to front:
BSP Trees : Viewpoint B

The Z-Buffer Algorithm (mid-70's)
• BSP trees proposed when memory was expensive
  • first 512x512 framebuffer was >$50,000!
  • Ed Catmull proposed a radical new approach called z-buffering
• the big idea:
  • resolve visibility independently at each pixel

Z-Buffer
• store (r,g,b,z) for each pixel
  • typically 8+8+8+24 bits, can be more

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

The Z-Buffer Algorithm
• what happens if multiple primitives occupy the same pixel on the screen?
  • which is allowed to paint the pixel?
  - order of insertion can affect half-plane extent

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
  • with 16 bits cannot discern millimeter differences in objects at 1 km distance
  - demo: sjbaker.org/steve/omniv/love_your_z_buffer.html

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

Summary: BSP Trees
• pros:
  • simple, elegant scheme
  • correct version of painter's algorithm back-to-front rendering approach
  • was very popular for video games (but getting less so)
• cons:
  • slow to construct tree: O(n log n) to split, sort
  • splitting increases polygon count: O(n^2) worst-case
  • computationally intense pre-processing stage restricts algorithm to static scenes

BSP Demo
• useful demo:
  http://symbolcraft.com/graphics/bsp

Depth Test Precision
• reminder: perspective transformation maps
  eye-space (view) z to NDC z
  - thus: \( z_{NDC} = \frac{C + D z_{eye}}{1} \)

Interpolating Z
• barycentric coordinates
  • interpolate Z like other planar parameters

BSP Demo
• order of insertion can affect half-plane extent

BSP Demo
• reminder: perspective transformation maps
  eye-space (view) z to NDC z
  - thus: \( z_{NDC} = \frac{C + D z_{eye}}{1} \)

The Z-Buffer Algorithm
• augment color framebuffer with Z-buffer or depth buffer which stores Z value at each pixel
  • at frame beginning, initialize all pixel depths to \( \infty \)
  • when rasterizing, 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

BSP Demo
• order of insertion can affect half-plane extent

Summary: BSP Trees
• pros:
  • simple, elegant scheme
  • correct version of painter's algorithm back-to-front rendering approach
  • was very popular for video games (but getting less so)
• cons:
  • slow to construct tree: O(n log n) to split, sort
  • splitting increases polygon count: O(n^2) worst-case
  • computationally intense pre-processing stage restricts algorithm to static scenes

The Z-Buffer Algorithm
• we know how to rasterize polygons into an image discretized into pixels:

The Z-Buffer Algorithm
• augment color framebuffer with Z-buffer or depth buffer which stores Z value at each pixel
  • at frame beginning, initialize all pixel depths to \( \infty \)
  • when rasterizing, 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

BSP Demo
• order of insertion can affect half-plane extent

Summary: BSP Trees
• pros:
  • simple, elegant scheme
  • correct version of painter's algorithm back-to-front rendering approach
  • was very popular for video games (but getting less so)
• cons:
  • slow to construct tree: O(n log n) to split, sort
  • splitting increases polygon count: O(n^2) worst-case
  • computationally intense pre-processing stage restricts algorithm to static scenes

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 or depth buffer which stores Z value at each pixel
  • at frame beginning, initialize all pixel depths to \( \infty \)
  • when rasterizing, 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

Depth Test Precision
• reminder: perspective transformation maps
  eye-space (view) z to NDC z
  - thus: \( z_{NDC} = \frac{C + D z_{eye}}{1} \)

Interpolating Z
• barycentric coordinates
  • interpolate Z like other planar parameters

The Z-Buffer Algorithm
• we know how to rasterize polygons into an image discretized into pixels:

The Z-Buffer Algorithm
• augment color framebuffer with Z-buffer or depth buffer which stores Z value at each pixel
  • at frame beginning, initialize all pixel depths to \( \infty \)
  • when rasterizing, 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

Depth Test Precision
• reminder: perspective transformation maps
  eye-space (view) z to NDC z
  - thus: \( z_{NDC} = \frac{C + D z_{eye}}{1} \)

Interpolating Z
• barycentric coordinates
  • interpolate Z like other planar parameters

The Z-Buffer Algorithm
• we know how to rasterize polygons into an image discretized into pixels:

The Z-Buffer Algorithm
• augment color framebuffer with Z-buffer or depth buffer which stores Z value at each pixel
  • at frame beginning, initialize all pixel depths to \( \infty \)
  • when rasterizing, 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

Depth Test Precision
• reminder: perspective transformation maps
  eye-space (view) z to NDC z
  - thus: \( z_{NDC} = \frac{C + D z_{eye}}{1} \)

Interpolating Z
• barycentric coordinates
  • interpolate Z like other planar parameters

The Z-Buffer Algorithm
• we know how to rasterize polygons into an image discretized into pixels:
More: Integer Depth Buffer

- reminder from picking discussion
- depth lies in the NDC z range [0,1]
- format: multiply by $2^n - 1$ then round to nearest int
  - where $n =$ number of bits in depth buffer
- 24 bit depth buffer = $2^{24} = 16,777,216$ possible values
- small numbers near, large numbers far
- consider depth from VCS: $(1<<N) \times (a + b / z) $
  - $N =$ number of bits of Z precision
  - $a = 2 f a r / (2 f a r - 2 n e a r )$
  - $b = 2 f a r \times 2 n e a r / (2 n e a r - 2 f a r )$
  - $z =$ distance from the eye to the object

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 interpretation
- enables deferred shading
  - rasterize shading parameters (e.g., surface normal) and only shade final visible fragments

Z-Buffer Cons

- poor for scenes with high depth complexity
  - need to render all polygons, even if most are invisible

Z-Buffer Cons

- requires lots of memory
  - (e.g. 1280x1024x32 bits)
- requires fast memory
- Read-Modify-Write in inner loop
- hard to simulate translucent 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

Hidden Surface Removal

- two kinds of visibility algorithms
  - object space methods
  - image space methods

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

Image Space Algorithms

- perform visibility test for in screen coordinates
  - limited to resolution of display
- Z-buffer: check every pixel independently performed late in rendering pipeline

Projective Rendering Pipeline

Rendering Pipeline

- object world viewing
  - OCS
  - WCS
  - viewing VCS
  - after $w$ (Frustum(...)
    - modeling transformation
    - viewing VCS
    - projection transformation
    - clipping
  - CCS

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 back-facing polygons improves performance
  - by how much?
  - 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 it has two sides, “inside” and “outside”
    - a Mobius strip or a Klein bottle is not orientable
  - closed: cannot “walk” from one side to the other
    - sphere is closed manifold
    - plane is not

Back-Face Culling

- examples of non-manifold objects:
  - a single polygon
    - terrain or height field
    - polyhedron with missing face
    - anything with cracks or holes in boundary
    - one-polygon thick lampshade

Back-face Culling: VCS

- first idea: cut if $N_y < 0$
- sometimes misses polygons that should be culled

Examples of non-manifold objects:

- a single polygon
  - terrain or height field
  - polyhedron with missing face
  - anything with cracks or holes in boundary
  - one-polygon thick lampshade

Examples of non-manifold objects:

- a single polygon
  - terrain or height field
  - polyhedron with missing face
  - anything with cracks or holes in boundary
  - one-polygon thick lampshade
Back-face Culling: NDCS

- why might a polygon be invisible?
  - polygon outside the field of view / frustum
    - solved by clipping
  - polygon is backfacing
    - solved by backface culling
  - polygon is occluded by object(s) nearer the viewpoint
    - solved by hidden surface removal

VCS

eye

NDCS

works to cull if $N_z > 0$

Invisible Primitives

- why might a polygon be invisible?
  - polygon outside the field of view / frustum
    - solved by clipping
  - polygon is backfacing
    - solved by backface culling
  - polygon is occluded by object(s) nearer the viewpoint
    - solved by hidden surface removal

Visibility is determined by the z buffer.

Blending

- blend white and clear equally (50% each)
  - white is (1,1,1,1), clear is (0,0,0,0), black is (0,0,0,1)
  - premultiplied: multiply componentwise by 50% and just add together
  - (1, 1, 1, 1) is indeed half-transparent white in non-premultiplied format
  - 4-tuple would mean half-transparent grey in non-premultiplied format

Alpha Examples

- blend white and clear equally (50% each)
  - white is (1,1,1,1), clear is (0,0,0,0), black is (0,0,0,1)
  - premultiplied: multiply componentwise by 50% and just add together
  - (1, 1, 1, 1) is indeed half-transparent white in non-premultiplied format
  - 4-tuple would mean half-transparent grey in non-premultiplied format

- premultiply allows both conventional blend and additive blend
  - alpha 0 and RGB nonzero: glowing/transparent
  - price for particle systems, stay tuned!

for more: see nice writeup from Alvy Ray Smith
  - technical academy award for Smith, Catmull, Porter, Duff
  - http://www.alvyray.com/Awards/AwardsAcademy96.htm