Visibility

H&B chapter 9 (similar to notes)

- Of these polygons, which are visible? (in front, etc.)
- Very large number of different algorithms known. Two main (very rough) classes:
  - Object precision: computations that decompose polygons in world coordinates
  - Image precision: computations at the pixel level
- Depth order in standard view box is same as depth order in 3D, so can work with the box.

Visibility

H&B chapter 9 (similar to notes)

- Essential issues:
  - must be capable of handling complex rendering databases.
  - in many complex worlds, few things are visible
  - efficiency - don't render pixels many times.
  - accuracy - answer should be right, and behave well when the viewpoint moves
  - aliasing

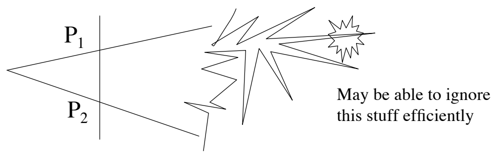
Image Precision

- Typically simpler algorithms (e.g., Z-buffer, ray cast)
- Pseudocode (conceptual!)

```
– For each pixel
    • Determine the closest surface which intersects the projector
    • Draw the pixel the appropriate color
```

Image Precision

- "Image precision" means that we can save time not computing precise intersections of complicated objects



- But the algorithms are subject to aliasing problems, and the sampling needs to be redone when the view changes, even if only a simple window resize

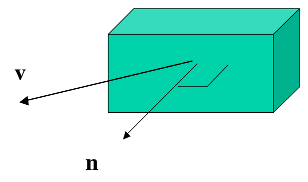
Object Precision

- The algorithms are typically more complex
- Pseudocode (conceptual)
  - For each object
    - Determine which parts are viewed without obstruction by other parts of itself of other objects
    - Draw those parts the appropriate color

Visibility - Back Face Culling

- Simple, preliminary step, to reduce the amount of work.
- Polygons from solid objects have a front face and back face
- If the viewer sees the back face, then the plane can be culled.
- Why would the viewer see the back face?
  - Because they are on the back side of the plane of the polygon.

Visibility - Back Face Culling



v is direction from any point on the plane to the center of projection (the eye).

If **n**.**v** > 0, then display the plane

Note that we are calculating which side of the plane the eye is on.

Question: How do we get **n**? (e.g., for the assignment)

Visibility - Back Face Culling

Question: How do we get **n**? (e.g., for the assignment)

Answer

When you render the parallelepiped, you have to create the faces which are sequences of vertices.

To compute **n** from vertices, use cross product.

You need to store vertices consistently so that you can get the sign of n. Consider storing them so that you can get the sign of **n** by RHR.

Depending on the situation you may find it easier to
1) compute **n** early on, and transform it using the correct formula
2) recompute it from transformed vertices.

Visibility - Back Face Culling

Question: In which coordinate frames can/should we do this?

Visibility - Back Face Culling

Question: In which coordinate frames can/should we do this?

Answer

All of them. It is perhaps more natural to attempt do this in the standardized view box where perspective projection has become parallel projection.

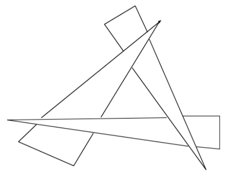
Here, **e**=(0,01) (why?), so the test **n.e**>0 is especially easy (n_z>0).

But be careful with the transformation which lead to **n**!

Also, an efficiency argument can be made for culling before division.

Visibility - painters algorithm

- Algorithm
  - Choose an order for the polygons based on some choice (e.g. depth to a point on the polygon)
  - Render the polygons in that order, deepest one first
- This renders nearer polygons over further.
- Works for some important geometries (2.5D - e.g. VLSI, mazes--but more efficient algorithms exist)
- Doesn’t work in this form for most geometries (see figure)



The Z - buffer

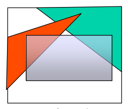
- For each pixel on screen, have a second memory location - called the z-buffer
- Initialize this buffer to a value corresponding to the furthest point possible.
- As a polygon is filled in, compute the depth value of each pixel
  - if depth < z buffer depth, fill in pixel and new depth
  - else disregard
- Typical implementation: Compute Z while scan-converting. A ∂Z for every ∂X can be easy to work out.

The Z - buffer

- Advantages:
  - simple; hardware implementation common
  - efficient z computations are easy.
  - ok with lots of surfaces (if there are lots, they tend to be small, and not much difference to this algorithm)
- Disadvantages:
  - over renders - can be slow for very large collections of polygons - may end up scan converting many hidden objects
  - quantization errors can be annoying (not enough bits in the buffer)
  - doesn’t help with transparency, or filtering for anti-aliasing.

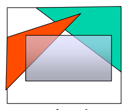
The A - buffer

- For transparent surfaces and filter based anti-aliasing:
- Algorithm (1): filling buffer
  - at each pixel, maintain a pointer to a list of polygons sorted by depth.
  - when filling a pixel:
    - if polygon is opaque and covers pixel, insert into list, removing all polygons farther away
    - if polygon is opaque and only partially covers pixel, insert into list, but don’t remove farther polygons


pixel

The A - buffer

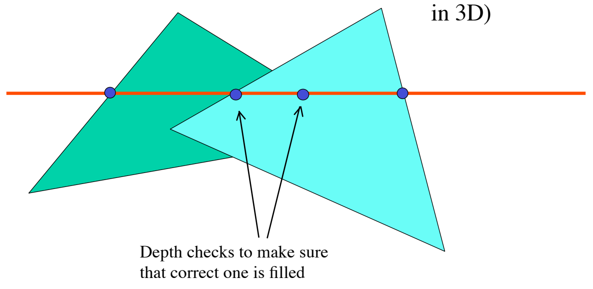
- Algorithm (2): rendering pixels
  - at each pixel, traverse buffer using brightness values in polygons to fill.
  - values are used for either for calculations involving transparency or for filtering for aliasing


pixel

Scan line algorithm

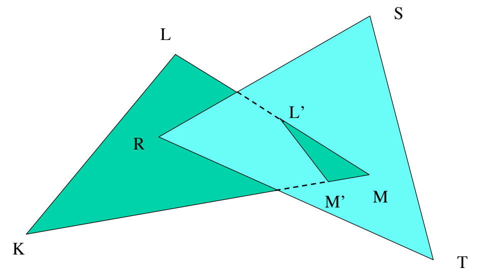
- Assume (for a moment) that polygons do not intersect one another (in 3D).
- Observation: on any given scan line, the visible polygon can change only at an edge.
- Algorithm:
  - fill all polygons simultaneously at each scan line, have all edges that cross scan line in AEL
  - keep record of current depth at current pixel
  - use current depth to decide which polygon to use for a span when a new edge is encountered

Scan line algorithm



Scan line algorithm

- To deal with penetrating polygons, split them up



Scan line algorithm

- Advantages:
  - potentially fewer quantization errors (typically more bits available for depth because a separate frame size buffer is not needed)
  - filter anti-aliasing can be made to work.
- Disadvantages:
  - invisible polygons clog AEL and ET. (Can get expensive for complex scenes).

Depth sorting

- Logically like painter's algorithm, except that problem cases are handled properly (painters++).
- First, sort polygons in order of decreasing depth
  - (based on closest point)
- Render in sorted order as described next slide.
- Each polygon encountered is checked using several tests to see if it is safe to paint (otherwise fix).
- Structure cases so that cheap tests are done first, and hope that expensive tests do not occur too often.

Depth sorting

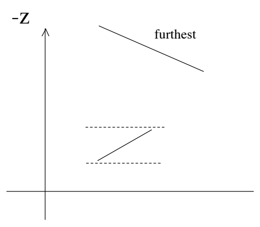
- For surface S with greatest depth
  - if no depth overlaps (z extent do not intersect) with other surfaces, then render (like painter's algorithm), and remove surface from list
    - test for depth overlaps by considering the z bounds (extents).
  - if a depth overlap is found, test for problem overlap in image plane
    - see next slides
  - if S, S' overlap in depth and in image plane, swap and try again
  - if S, S' have been swapped already, split one across plane of other (like clipping) and reinsert

Depth sorting

- Testing image plane problem overlaps (test get increasingly expensive):
  - xy bounding boxes do not intersect
  - *or* S is behind the plane of S'
  - *or* S' is in front of the plane of S
  - *or* S and S' do not intersect

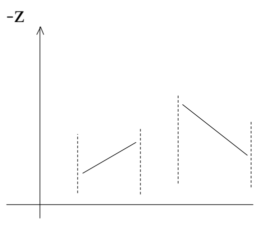
See figures

Depth sorting (2D illustrations)



No depth overlap between furthest object and rest of list--paint it.

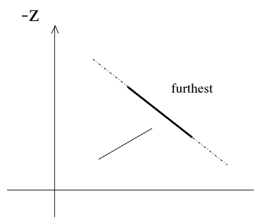
Depth sorting (2D illustrations)



Overlap in depth, but no overlap in image plane.

This can determined by studying bounding boxes.

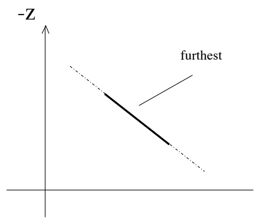
Depth sorting (2D illustrations)



It is safe to paint the furthest, but figuring this out requires observing that the near one is all on the same side of the plane (dotted line) of the furthest.

In other words: the near polygon is in front of the plane of the far polygon.

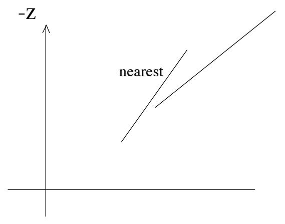
Depth sorting (2D illustrations)



It is safe to paint the furthest, but figuring this out requires observing that it is all on the other side of the plane (dotted line) of the nearest plane.

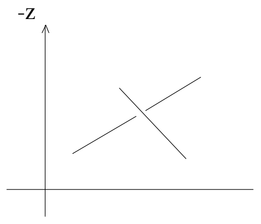
In other words: the far polygon is behind the plane of the near polygon.

Depth sorting (2D illustrations)



The furthest (as defined by the closest point) obscures the “nearest”. It is safe to paint the “nearest”, but figuring this out requires reversing the nearest and furthest, and then reapplying one of the previous tests.

Depth sorting (2D illustrations)



If the preceding tests fail, we have to split the far polygon (line in the drawing) with the plane of the near polygon (**basically a clip operation**), and put the pieces into the list, and carry on.

Depth sorting

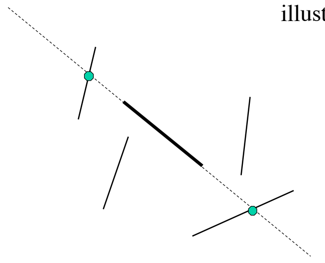
- Advantages:
  - filter anti-aliasing works fine (with good over-painting rules)
  - no depth quantization error
  - works well if not too much depth overlap (rarely get to expensive cases)
- Disadvantages:
  - gets expensive with lots of depth overlap (over-renders)

BSP - trees

- Construct a tree that gives a rendering order
- Tree recursively splits 3D world into cells, each of which contain at most one piece of polygon.
- Constructing tree:
  - choose polygon (arbitrary)
  - split its cell using plane on which polygon lies
  - continue until each cell contains only one polygon

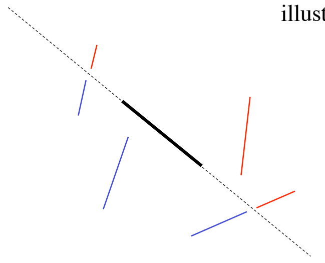
BSP - trees

2D version for illustration



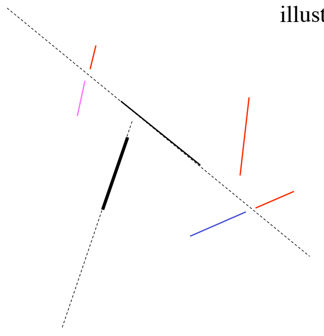
BSP - trees

2D version for illustration



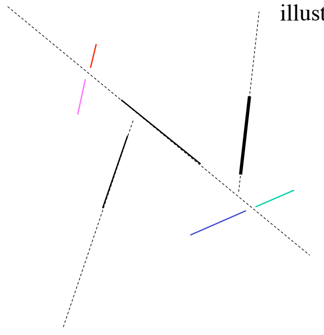
BSP - trees

2D version for illustration



BSP - trees

2D version for illustration

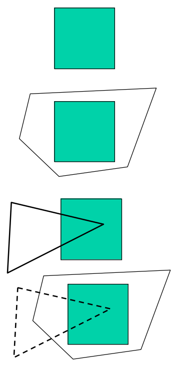


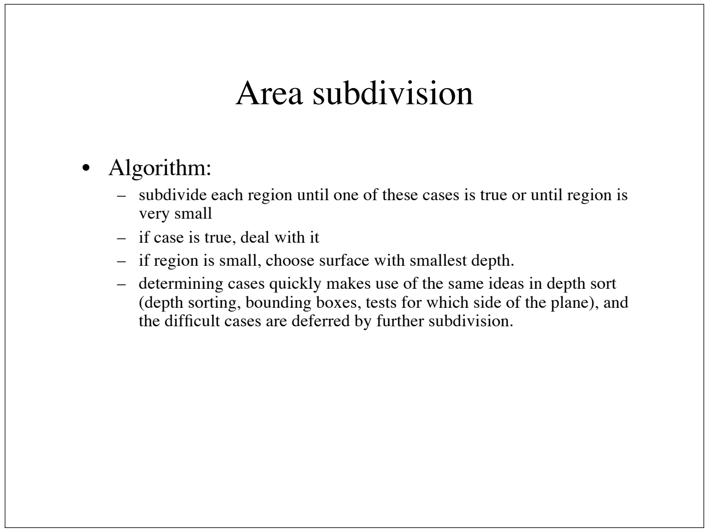
BSP - trees

- Rendering tree:
  - recursive descent
  - render back, node polygon, front
- Disadvantages:
  - many small pieces of polygon (more splits than depth sort!)
  - over rendering (does not work well for complex scenes with lots of depth overlap)
- Advantages:
  - one tree works for all focal points (good for cases when scene is static)
  - filter anti-aliasing works fine, as does transparency
  - data structure is worth knowing about
- Comment
  - expensive to get approximately optimal tree, but for many applications this can be “off-line” in a pre-processing step.

Area subdivision

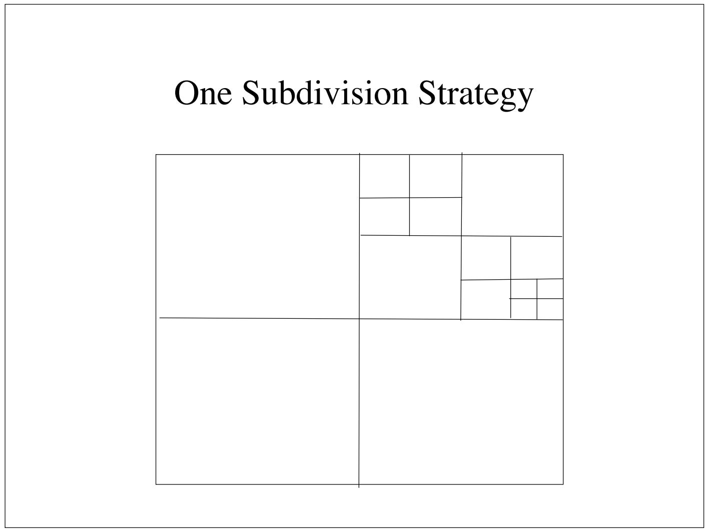
- Four tractable cases for a given region in image plane:
  - no surfaces project to the region
  - only one surface completely surrounds the region
  - only one surface is completely inside the region or overlaps the region
  - a polygon is completely in front of everything else in that region, where this can be determined by considering depths of the polygons at the corners of the region



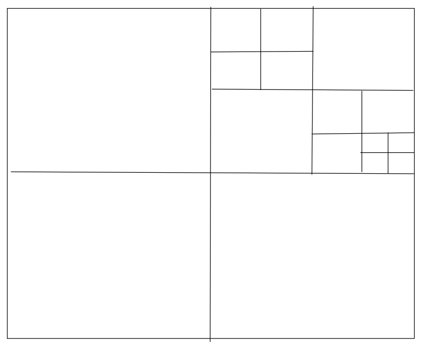


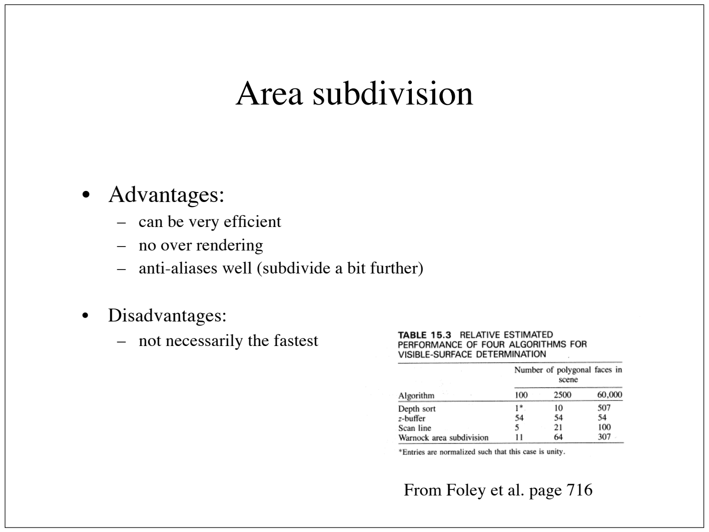
Area subdivision

- Algorithm:
  - subdivide each region until one of these cases is true or until region is very small
  - if case is true, deal with it
  - if region is small, choose surface with smallest depth.
  - determining cases quickly makes use of the same ideas in depth sort (depth sorting, bounding boxes, tests for which side of the plane), and the difficult cases are deferred by further subdivision.



One Subdivision Strategy





Area subdivision

- Advantages:
  - can be very efficient
  - no over rendering
  - anti-aliases well (subdivide a bit further)
- Disadvantages:
  - not necessarily the fastest

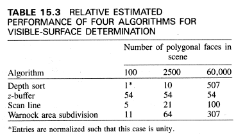
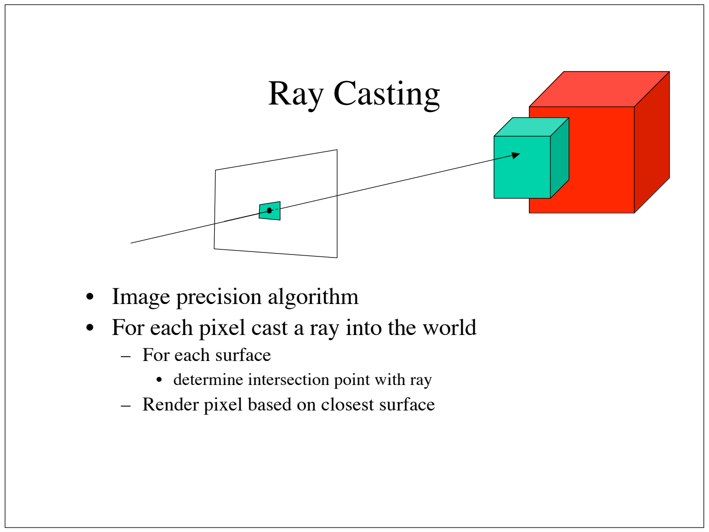


TABLE 15.3 RELATIVE ESTIMATED PERFORMANCE OF FOUR ALGORITHMS FOR VISIBLE-SURFACE DETERMINATION

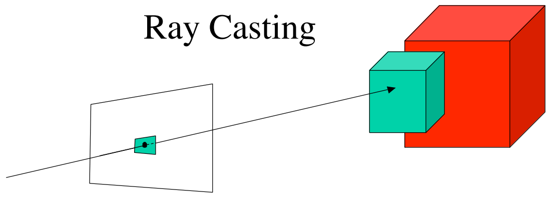
	Number of polygonal faces in scene		
Algorithm	100	2500	60,000
Depth sort	1*	10	507
z-buffer	54	54	54
Scan line	5	21	100
Warnock area subdivision	11	64	307

*Entries are normalized such that this case is unity.

From Foley et al. page 716



Ray Casting



- Image precision algorithm
- For each pixel cast a ray into the world
  - For each surface
    - determine intersection point with ray
  - Render pixel based on closest surface

Ray Casting

- First step in ray tracing algorithm
- Expensive
  - Good performance usually requires clever data structures such as bounding volumes for object groups or storing world occupancy information in octrees.
- Very useful for "picking"--not expensive here (why?)
- Other than performance, main problem is computing intersection.
- Our cleverly transformed spaces are less relevant
  - Canonical box and frustum were developed for other purposes.
  - Ray casting usually proceeds in original (of similar) coords.
  - For polygons, we could use the standardized orthographic space.
  - Spheres are normally easy (but not after any transformations that have made them non-spherical–careful!).

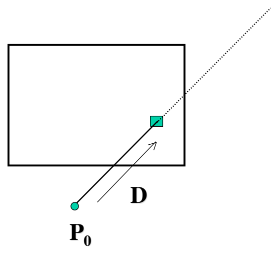
May be helpful for assignments.

Picking (details)

- Pick a polygon by shooting a ray from the focal point through a pixel
- (Many ways). Perhaps easiest to determine the ray in the space that you have after the first three transformations towards the canonical frustum.
  - Worried about performance due to transforming the world? Transform the pixel *back* to world coordinates.

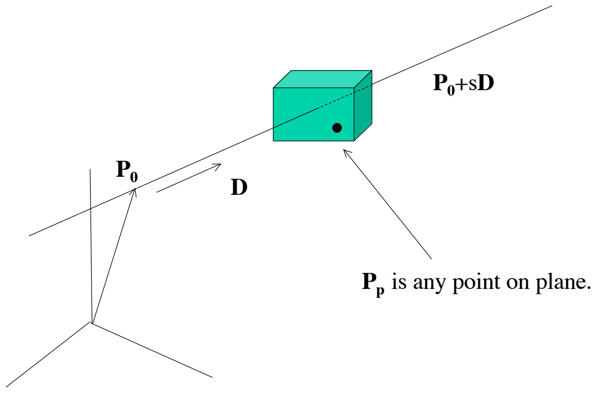
May be helpful for assignments.

Picking (details)



Poly details

May be helpful for assignments.



Poly details

May be helpful for assignments.

To find the intersection of the ray and the plane, solve:

$$\left(\mathbf{P}_0 + s\mathbf{D} - \mathbf{P}_p\right) \bullet \mathbf{n} = 0$$

Once you have the point of intersection, $\mathbf{P}_i$, test that it is inside by testing against all other faces indexed by j.

$$\left(\mathbf{P}_i - \mathbf{Q}_j\right) \bullet \mathbf{n}_j < 0$$

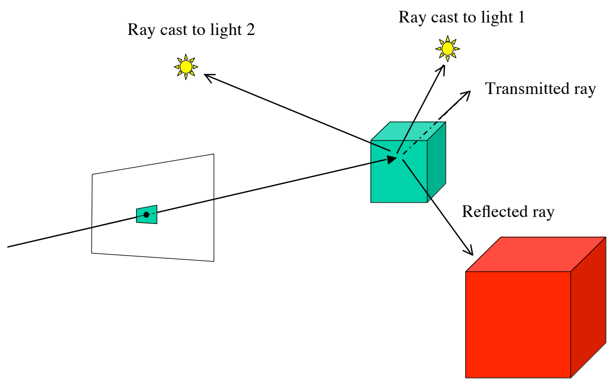
Note that $\mathbf{Q}_j$ are on those *other* faces.

Ray Tracing--teaser

- Idea is very simple--follow light around
- Following **all** the light around is intractable, so we follow the light that makes **the most** difference
- Work backwards from what is seen
- Simple ray tracer
  - Cast a ray through each pixel (as in ray casting for visibility)
  - From intersection point cast additional rays to determine the color of the pixel.
    - For diffuse component, must cast rays to the lights
    - We may also add in some "ambient" light
    - For mirrors, must cast ray in mirror direction (recursion--what is the stopping condition)

Recursive ray tracing

H&B, page 597



Current state of intro students graphic's ability

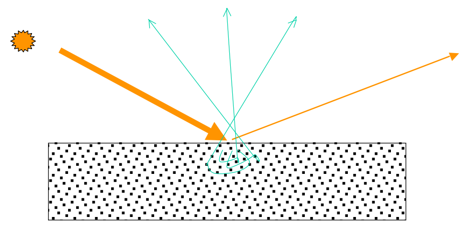
Know how to draw polygons
Know about cameras
Know how to map 3D polygons onto the screen
Know how to draw the bits closest to the cameras

Issues
- Should we live in a polygonal world?
- How do you get polygons for complex objects?
- What color should each pixel be?

Light interacting with the world

- The signal reaching your eye from a surface is the result of the surface interacting with the light following on it.
- Many effects when light strikes a surface. It could be:
  - absorbed
  - transmitted
  - reflected
  - scattered (in a variety of directions!)

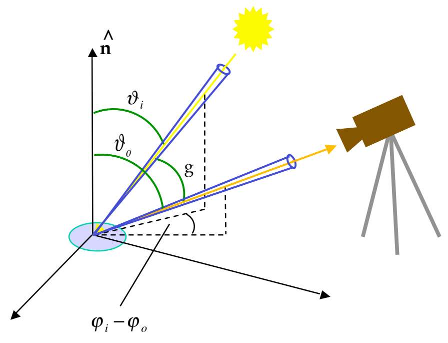
This shows some possibilities that can happen to the line from **one** direction.



Bidirectional Reflectance Distribution Function (BRDF)

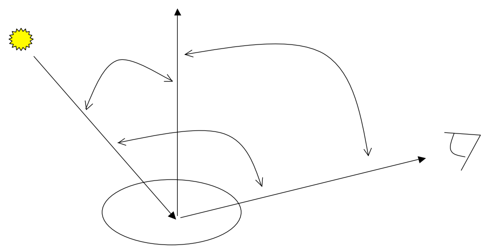
- The BRDF is a technical way of specifying how light from sources interacts with the matter in the world
- Understanding images requires understanding that this varies as a function of materials. The following "look" different
  - mirrors
  - white styrofoam
  - colored construction paper
  - colored plastic
  - gold
- The BRDF is the **ratio** of what comes out to what came in
- What comes out <--> "radiance"
- What goes in <--> "irradiance"
- Details on the BRDF available as supplementary material

This shows angular effects. There are also spectral (color effects).



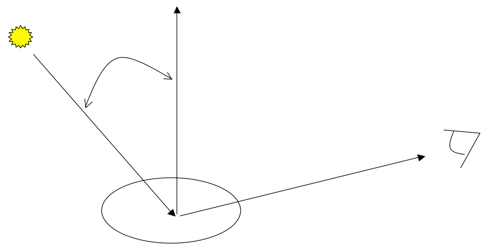
Isotropic surfaces

The BRDF for many surfaces can be well approximated as a function of 3 variables (angles), not 4. In this case, turning the surface around the normal has no effect. The surface is said to be *isotropic*.



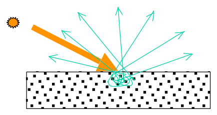
Lambertian surfaces

- Even simpler case--the BRDF does not depend on the viewing (output) direction (e.g., Lambertian).



Lambertian surfaces

- Simple special case of reflectance: ideal diffuse or matte surface--e.g. cotton cloth, matte paper.
- Surface appearance is independent of viewing angle.
- Typically such a surface is the result of lots of scattering---the light “forgets” where it came from, and it could end up going in any random direction.



- What counts is how much light power reaches the surface

Lambertian surfaces and albedo

- We will refer later to “radiosity” as a unit to describe light leaving the surface taken as whole
  - Technically, it is the total power leaving a point on the surface, per unit area on the surface (Wm^{-2})
- Recall that for a Lambertian surface, the direction that light leaves is not an issue.
- Percentage of light leaving the surface compared with that falling onto it, is often called diffuse reflectance, or *albedo* for a Lambertian surface.

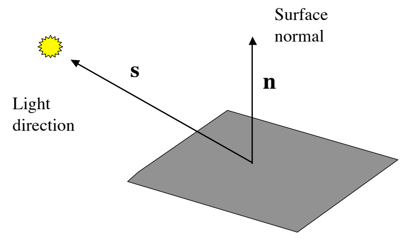
Lambertian surfaces

The Lambertian assumption leads to very simple rule to shade an object. Specifically, we attenuate brightness by

$$\underbrace{\mathbf{n}}_{\text{Surface normal}} \cdot \underbrace{\mathbf{s}}_{\text{Light source direction}}$$

Must know this

Lambertian Reflection



Brightness is proportional to **n•s**

Comments on light source direction

The direction to a nearby light changes as you move around in the scene.

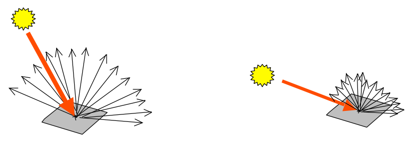
If we say a light source is “at infinity”, we mean that it is so far away that only the direction is important.

Example: On the scale of a city, the sun is at infinity.

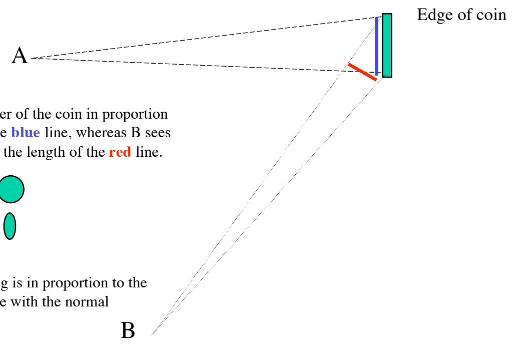
Lambertian Reflection

Why is brightness proportional to **n•s** ?

Intuitive argument: The surface scatters light in all directions equally, but as the angle of the light becomes oblique, the amount of light per unit area is reduced (foreshortening) by a factor of the cosine of the angle.



Foreshortening illustrated



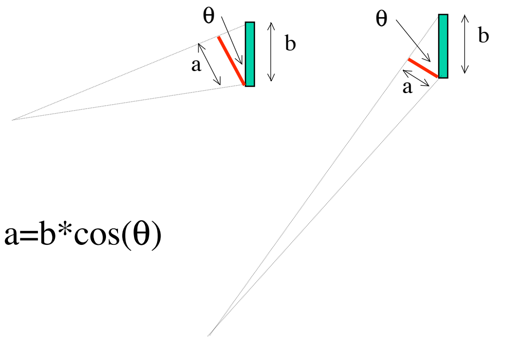
A sees the diameter of the coin in proportion to the length of the **blue** line, whereas B sees it in proportion to the length of the **red** line.

Further, A sees ●

But B sees ⬮

The foreshortening is in proportion to the cosine of the angle with the normal (next slide)

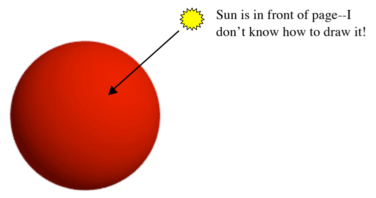
Foreshortening illustrated



$a=b*\cos(\theta)$

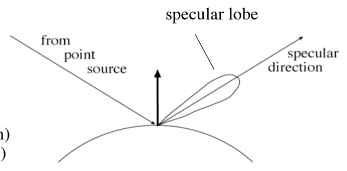
Lambertian surfaces

- Surface brightness is only a function of the foreshortening of the incident light (the more oblique it is, the less bright the surface).

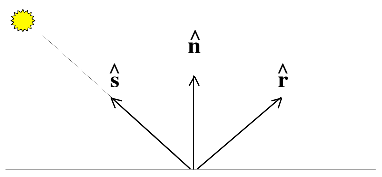


Specular surfaces

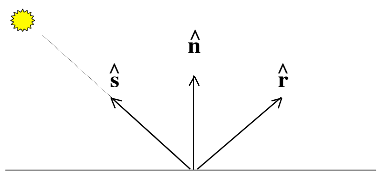
- Another important class of surfaces is specular (mirror-like).
  - specular surfaces reflect a significant amount of energy in the specular (mirror) direction
  - produces "highlights"
- Two related cases
  - a perfect mirror
  - a fuzzy mirror
- Typically there is a diffuse (Lambertian) component as well (effects are additive)



Computing reflection (specular) direction



Computing reflection (specular) direction



$\hat{\mathbf{s}} + \hat{\mathbf{r}} = k\hat{\mathbf{n}}$ and $\hat{\mathbf{n}} \bullet \hat{\mathbf{s}} = \hat{\mathbf{n}} \bullet \hat{\mathbf{r}}$

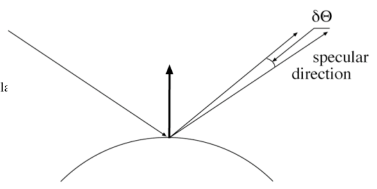
$\hat{\mathbf{n}} \bullet \hat{\mathbf{s}} + \hat{\mathbf{n}} \bullet \hat{\mathbf{r}} = k \Rightarrow k = \mathbf{2}\hat{\mathbf{n}} \bullet \hat{\mathbf{s}}$

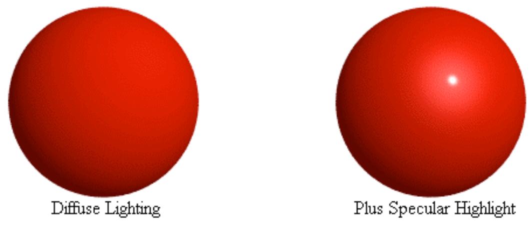
So $\hat{\mathbf{r}} = \mathbf{2}(\hat{\mathbf{n}} \bullet \hat{\mathbf{s}})\hat{\mathbf{n}} - \hat{\mathbf{s}}$

Phong's model of specularities

- There are very few cases where the exact shape of the specular lobe matters.
- Typically:
  - very, very small --- mirror
  - small -- blurry mirror
  - bigger -- see only light sources as "specula
  - very big -- faint specularities
- Phong's model
  - reflected energy falls off with

$$\cos^n(\delta\vartheta)$$





from
http://www.geocities.com/SiliconValley/Horizon/6933/shading.html