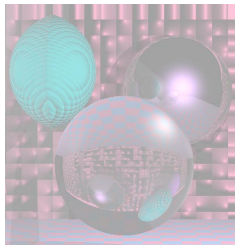
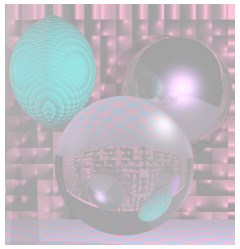


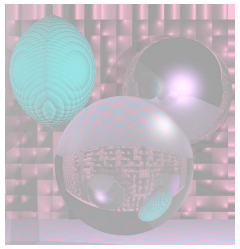
CS4300
Computer Graphics

Prof. Harriet Fell
Fall 2011
Lecture 22 – October 27 ,2011



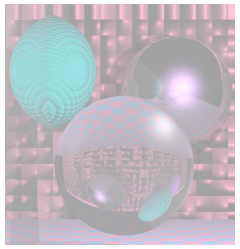
Today's Topics

- Poly Mesh
  - Hidden Surface Removal
  - Visible Surface Determination
  - More about the First 3D Project
  - First Lighting model



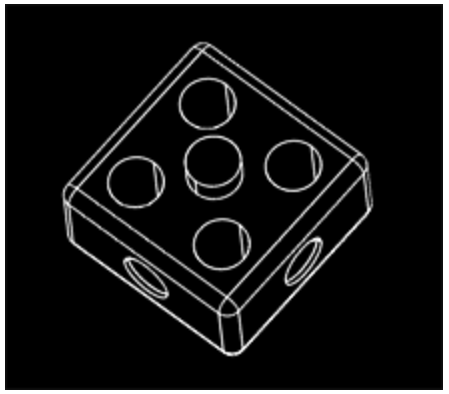
Rendering a Polymesh

- Scene is composed of triangles or other polygons.
- We want to view the scene from different view-points.
  - Hidden Surface Removal
    - Cull out surfaces or parts of surfaces that are not visible.
  - Visible Surface Determination
    - Head right for the surfaces that are visible.
    - Ray-Tracing is one way to do this.

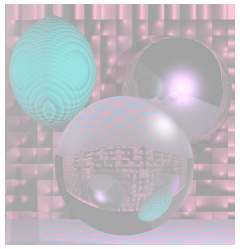


Wireframe Rendering

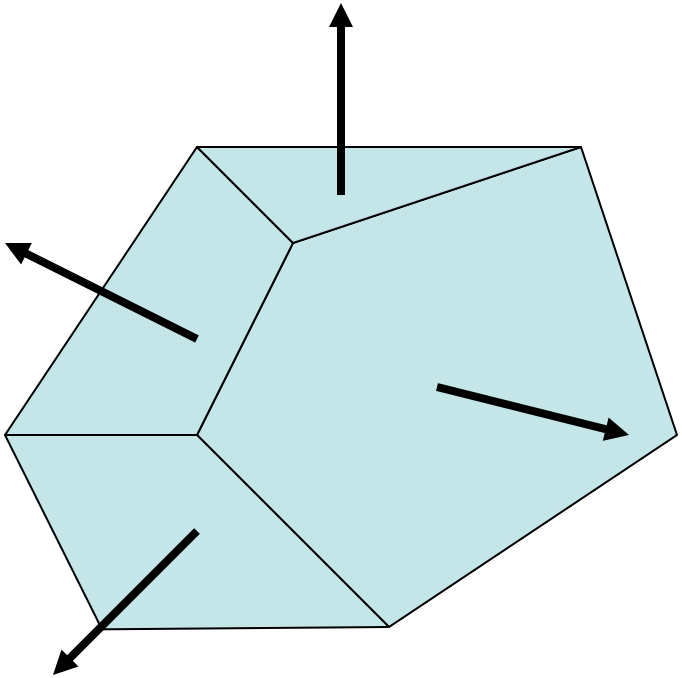
Hidden-Line Removal



Hidden-Face Removal



Convex Polyhedra

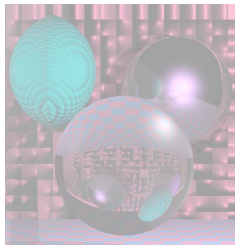


We can see a face if and only if its normal has a component toward us.

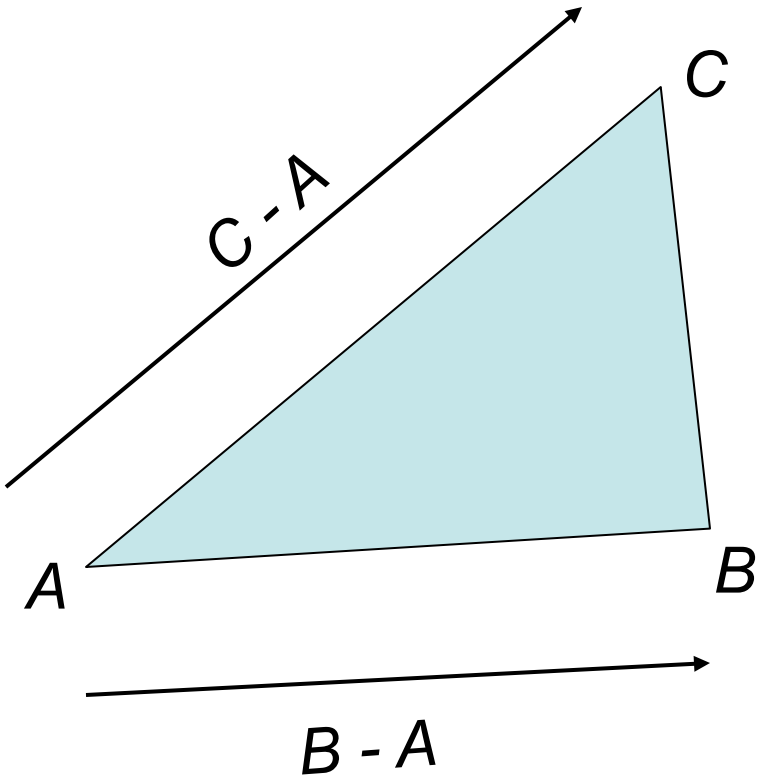
$$\boldsymbol{N}\cdot\boldsymbol{V} > 0$$

$\boldsymbol{V}$ points from the face toward the viewer.

$\boldsymbol{N}$ point toward the outside of the polyhedra.



Finding N

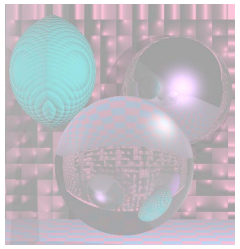


$N = (B - A) \times (C - A)$

is a normal to the triangle that points toward you.

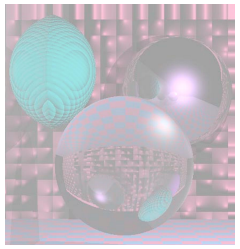
$$\frac{N}{\|N\|}$$

is a unit normal that points toward you.



Code for *N*

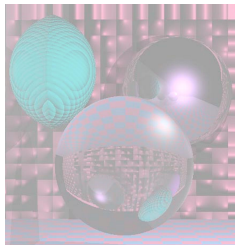
```
private Vector3d findNormal(){
        Vector3d u = new Vector3d();
        u.scaleAdd(-1, verts[0], verts[1]);
        Vector3d v = new Vector3d();
        v.scaleAdd(-1, verts[0], verts[2]);
        Vector3d uxv = new Vector3d();
        uxv.cross(u, v);
        return uxv;
}
```



Finding V

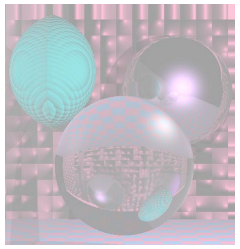
- Since we are just doing a simple orthographic projection, we can use
  $V = k = (0, 0, 1)$.
- Then
  $N \bullet V$ = the z component of N

```
public boolean faceForward() {
  return (normal.z > 0);
}
```



Find *L*

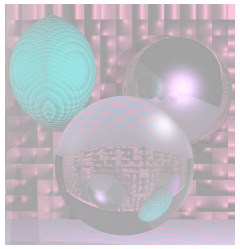
- *L* is a unit vector from the point you are about to render toward the light.
- For the faceted icosahedron use the center point of each face.
  - *cpt* = (*A* + *B* + *C*)/3



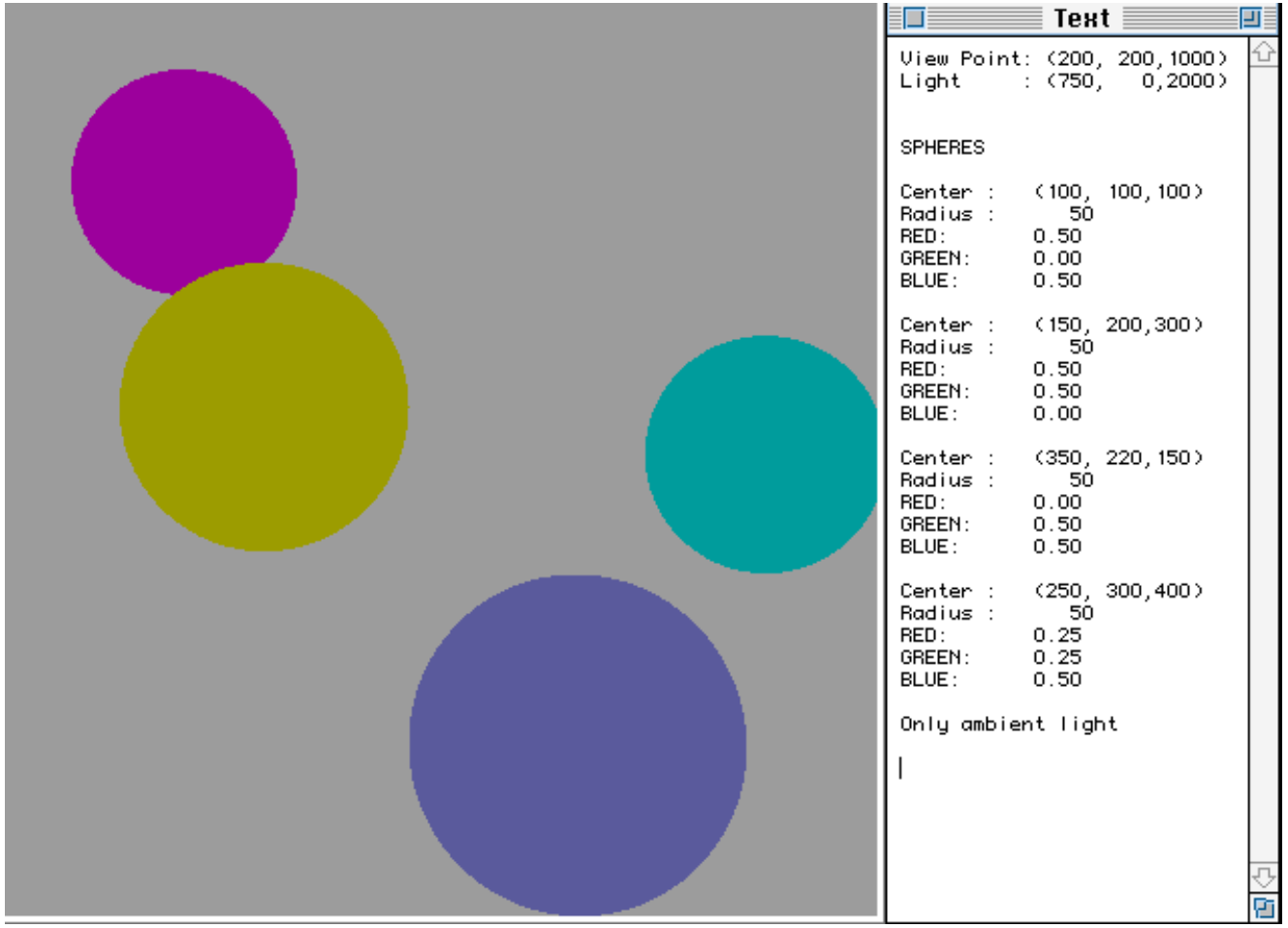
First Lighting Model

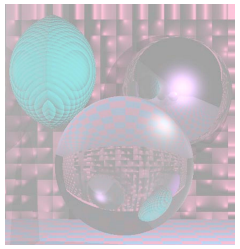
- Ambient light is a global constant *ka*.
  - Try *ka* = .2.
  - If a visible object S has color (S_R, S_G, S_B) then the ambient light contributes

    $(.2* S_R, .2* S_G, .2* S_B)$.
- Diffuse light depends of the angle at which the light hits the surface. We add this to the ambient light.
- We will also add a spectral highlight.

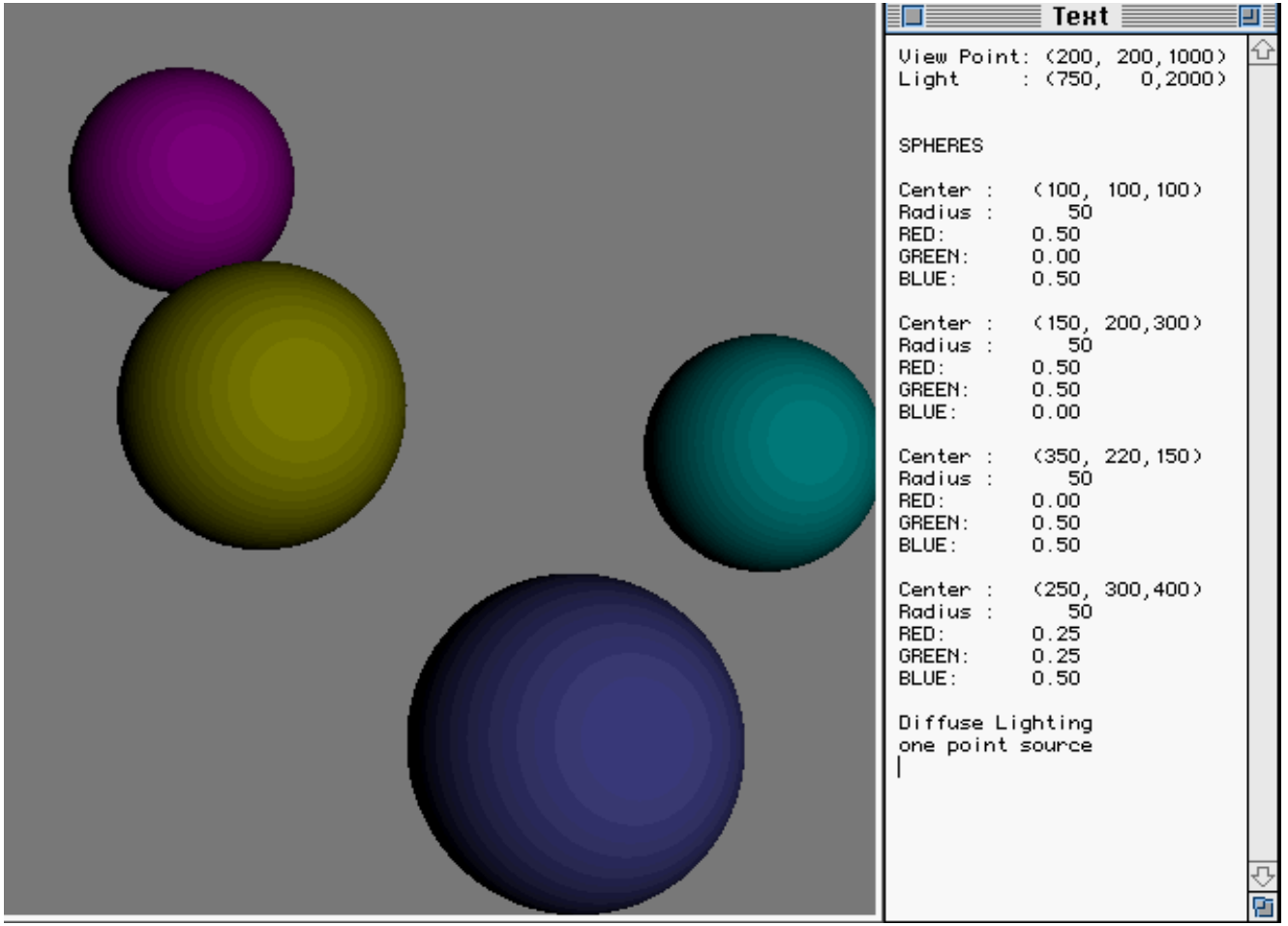


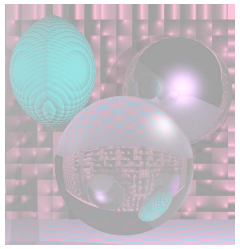
Visible Surfaces
Ambient Light





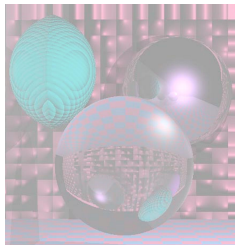
Diffuse Light





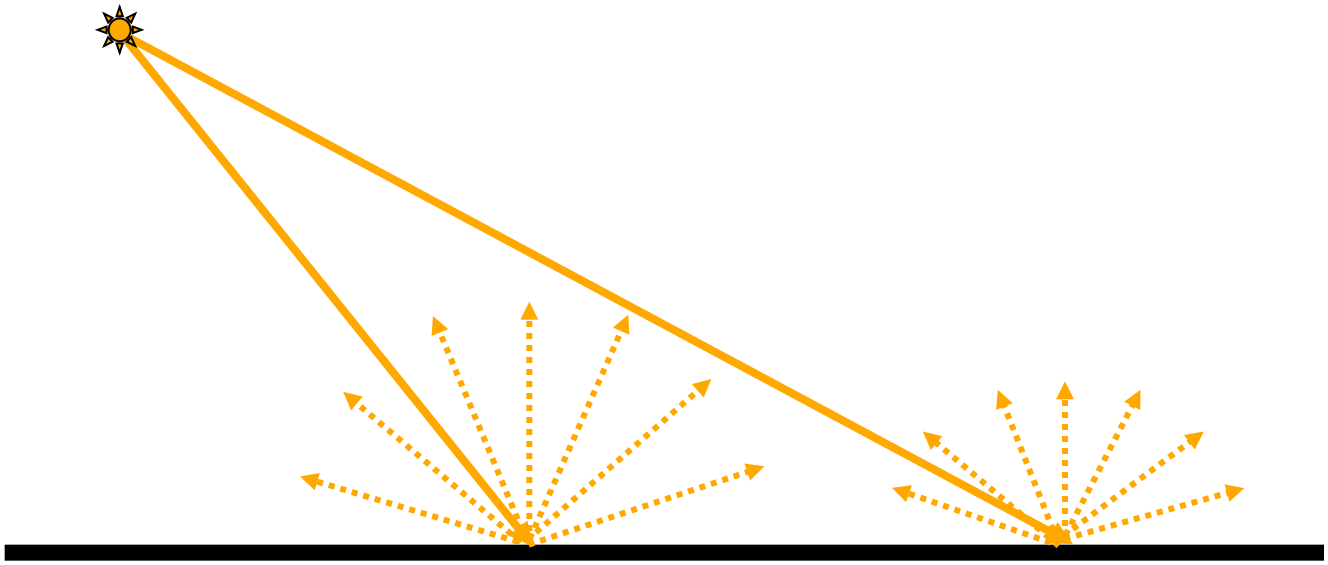
Lambertian Reflection Model Diffuse Shading

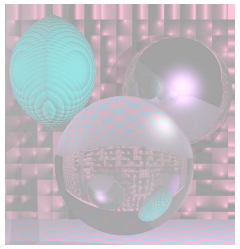
- For matte (non-shiny) objects
- Examples
  - Matte paper, newsprint
  - Unpolished wood
  - Unpolished stones
- Color at a point on a matte object does not change with viewpoint.



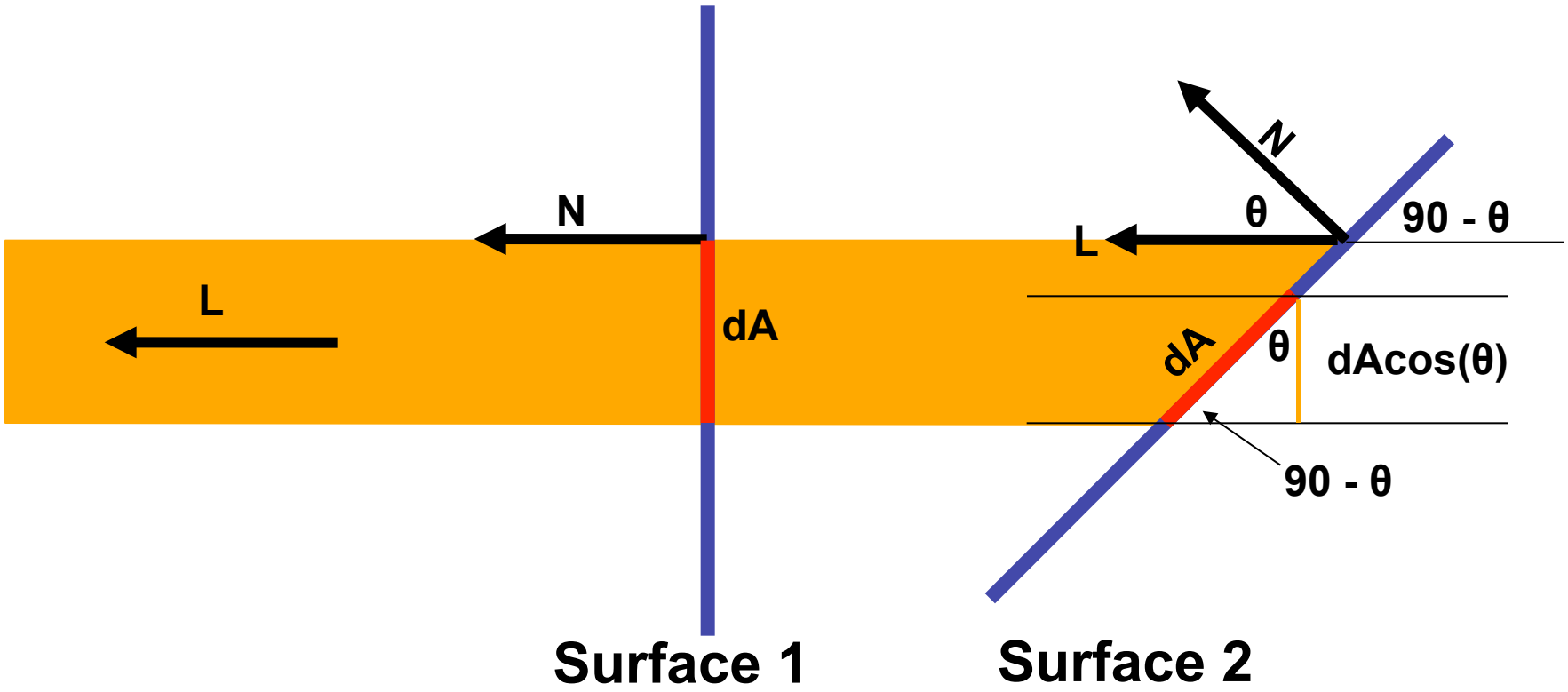
Physics of Lambertian Reflection

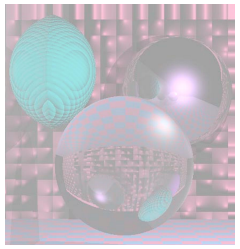
- Incoming light is partially absorbed and partially transmitted equally in all directions



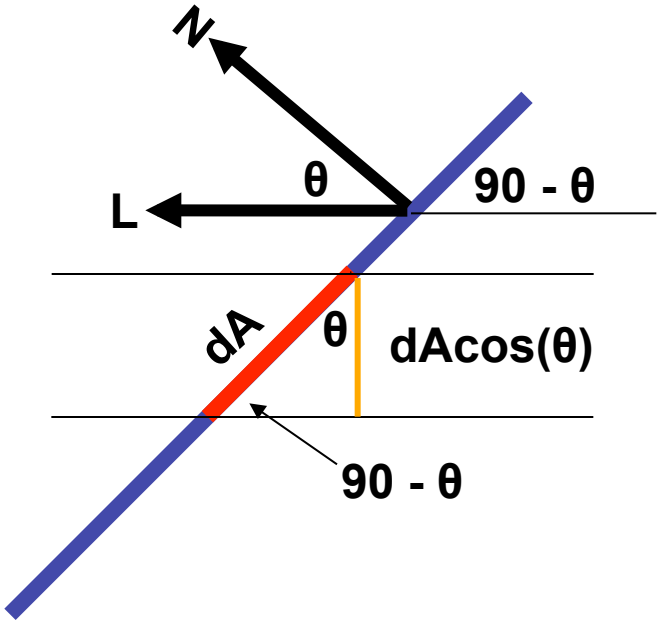


Geometry of Lambert's Law



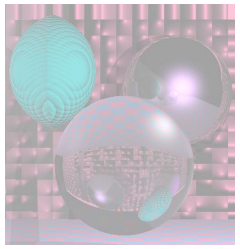


cos(θ)=N•L



Surface 2

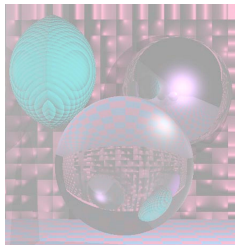
Cp= ka (SR, SG, SB) + kd **N•L** (SR, SG, SB)



Hidden Surface Removal

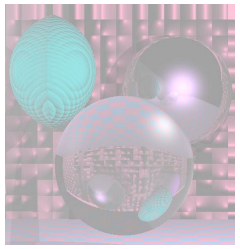
- Backface culling
  - Never show the back of a polygon.
- Viewing frustum culling
  - Discard objects outside the camera's view.
- Occlusion culling
  - Determining when portions of objects are hidden.
    - Painter's Algorithm
    - Z-Buffer
- Contribution culling
  - Discard objects that are too far away to be seen.

http://en.wikipedia.org/wiki/Hidden_face_removal



Visible Surface Determination

- If most surfaces are invisible, don't render them.
  - Ray Tracing
    - We only render the nearest object.
  - Binary Space Partitioning (BSP)
    - Recursively cut up space into convex sets with hyperplanes.
    - The scene is represented by a BSP-tree.



Sorting the Polygons

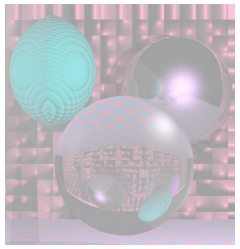
The first step of the Painter's algorithm is:

Sort objects back to front relative to the viewpoint.

The relative order may not be well defined.

We have to reorder the objects when we change the viewpoint.

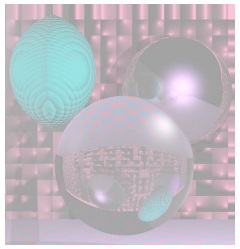
The BSP algorithm and BSP trees solve these problems.



Binary Space Partition

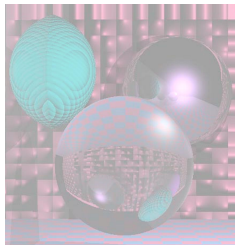
- Our scene is made of triangles.
  - Other polygons can work too.
- Assume no triangle crosses the plane of any other triangle.
  - We relax this condition later.

following Shirley *et al.*



BSP – Basics

- Let a plane in 3-space (or line in 2-space) be defined implicitly, i.e.
  - $f(\boldsymbol{P}) = f(x, y, z) = 0$ in 3-space
  - $f(\boldsymbol{P}) = f(x, y) = 0$ in 2-space
- All the points $\boldsymbol{P}$ such that $f(\boldsymbol{P}) > 0$ lie on one side of the plane (line).
- All the points $\boldsymbol{P}$ such that $f(\boldsymbol{P}) < 0$ lie on the other side of the plane (line).
- Since we have assumed that all vertices of a triangle lie on the same side of the plane (line), we can tell which side of a plane a triangle lies on.



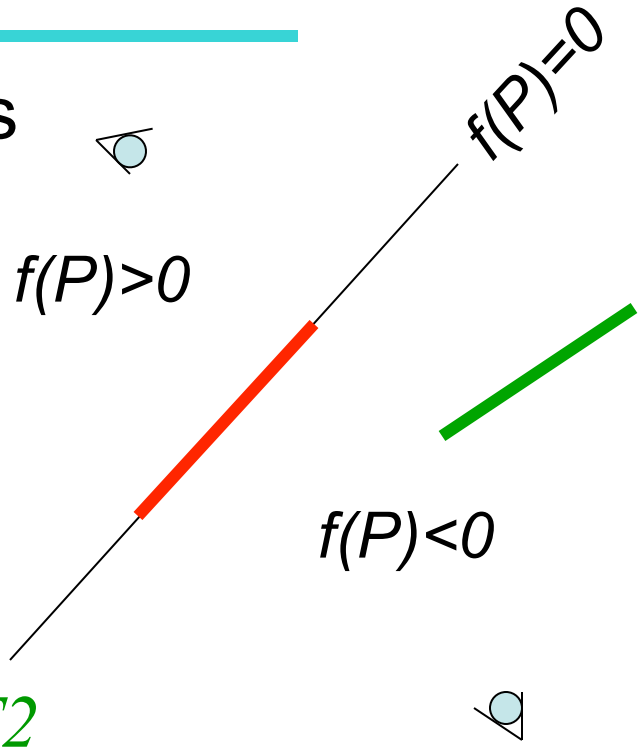
BSP on a Simple Scene

Suppose scene has 2 triangles

T1 on the plane $f(\boldsymbol{P}) = 0$

T2 on the $f(\boldsymbol{P}) < 0$ side

e is the eye.

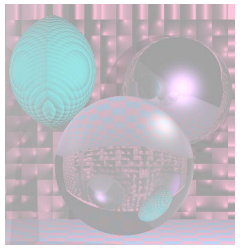


if $f(e) < 0$ **then**

draw *T1*; draw *T2*

else

draw *T2*; draw *T1*



The BSP Tree

Suppose scene has many triangles, $T1, T2, \ldots$.

We still assume no triangle crosses the plane of any other triangle.

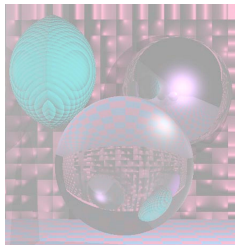
Let $f_i(\boldsymbol{P}) = 0$ be the equation of the plane containing Ti.

The *BSPTREE* has a node for each triangle with $T1$ at the root.

At the node for Ti,

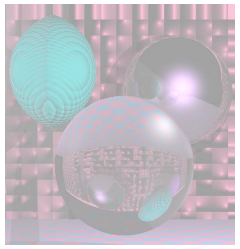
the minus subtree contains all the triangles whose vertices have $f_i(\boldsymbol{P}) < 0$

the plus subtree contains all the triangles whose vertices have $f_i(\boldsymbol{P}) > 0$.



BSP on a non-Simple Scene

```
function draw(bsptree tree, point e)
if (tree.empty) then
        return
if (f_tree.root(e) < 0) then
        draw(tree.plus, e)
        render tree.triangle
        draw(tree.minus, e)
else
        draw(tree.minus, e)
        render tree.triangle
        draw(tree.plus, e)
```



2D BSP Trees Demo

http://www.symbolcraft.com/graphics/bsp/index.php

This is a demo in 2 dimensions.

The objects are line segments.

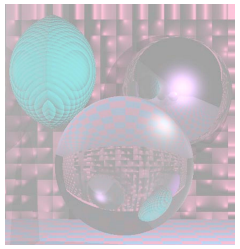
The dividing hyperplanes are lines.

Building the BSP Tree

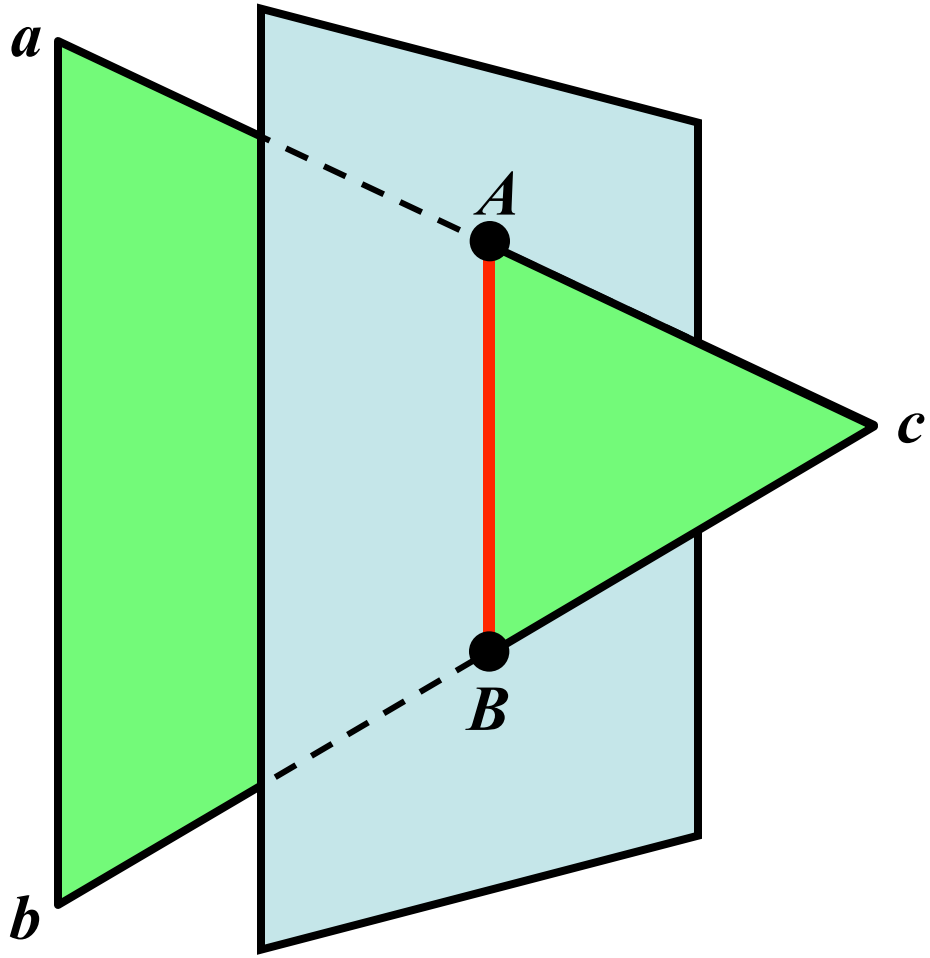
We still assume no triangle crosses the plane of another triangle.

```
tree = node(T1)
for  i in {2, …, N} do tree.add(Ti)

function add (triangle T)
if (f(a) < 0 and f(b) < 0 and f(c) < 0) then
   if (tree.minus.empty) then
      tree.minus = node(T)
   else
       tree.minus.add(T)
else if (f(a) > 0 and f(b) > 0 and f(c) > 0) then
    if (tree.plus.empty) then
       tree.plus = node(T)
   else
     tree.plus.add(T)
```

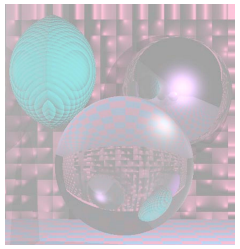


Triangle Crossing a Plane

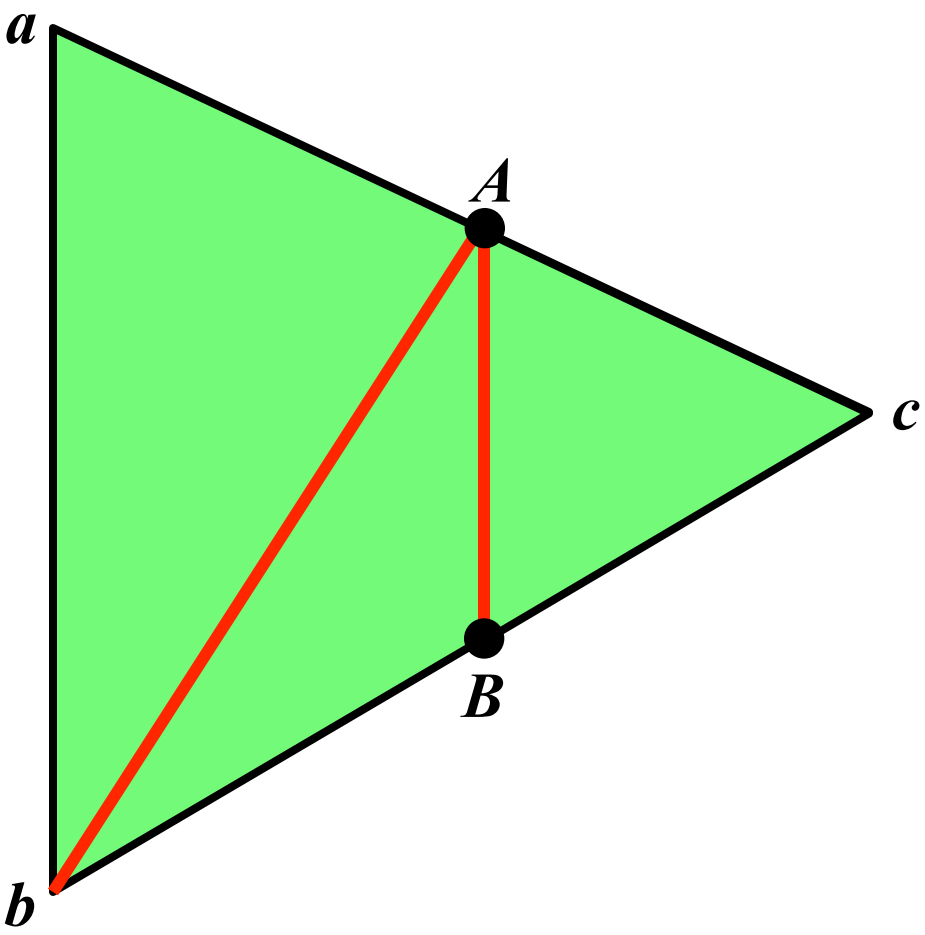


Two vertices, a and b, will be on one side and one, c, on the other side.

Find intercepts , A and B, of the plane with the 2 edges that cross it.

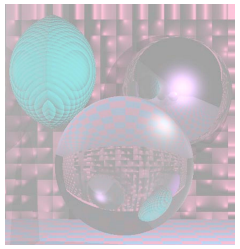


Cutting the Triangle



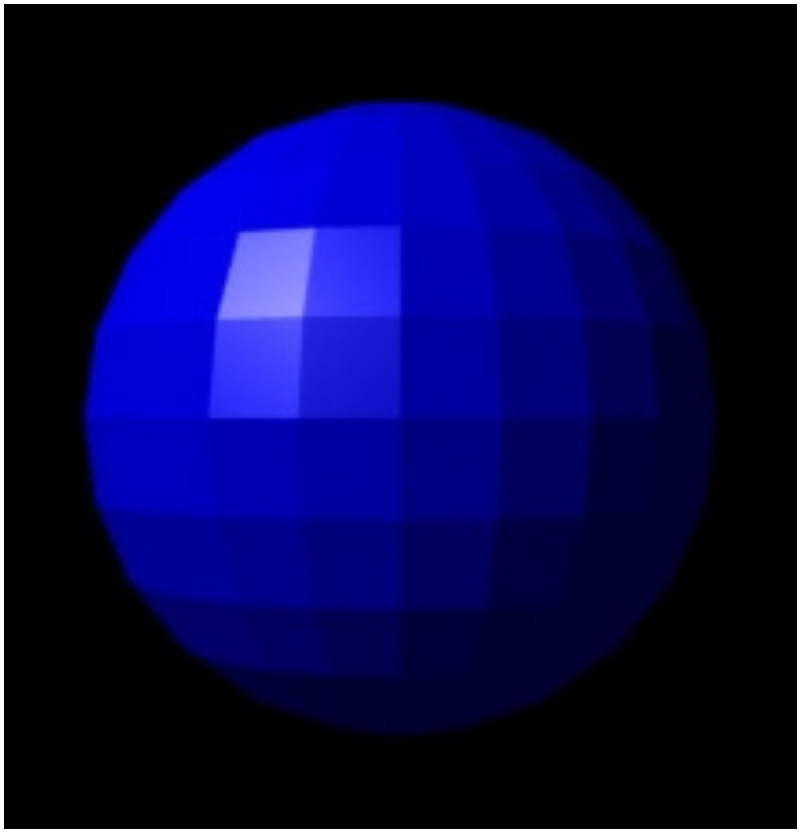
Cut the triangle into three triangles, none of which cross the cutting plane.

Be careful when one or more of ***a***, ***b***, and ***c*** is close to or on the cutting plane.

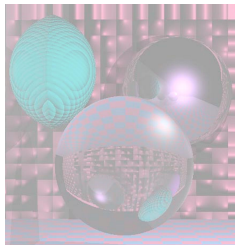


Flat Shading

- A single normal vector is used for each polygon.
- The object appears to have facets.

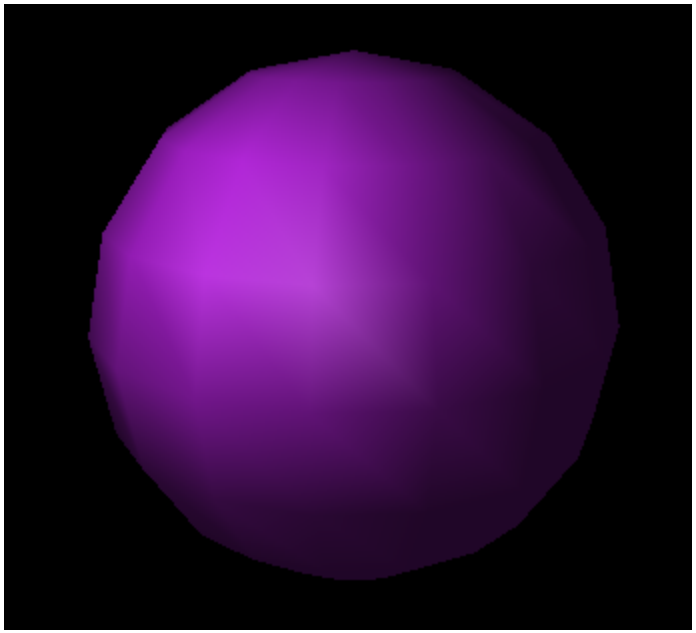


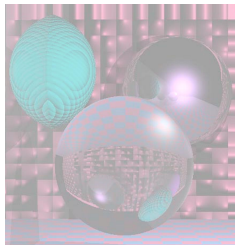
http://en.wikipedia.org/wiki/Phong_shading



Gouraud Shading

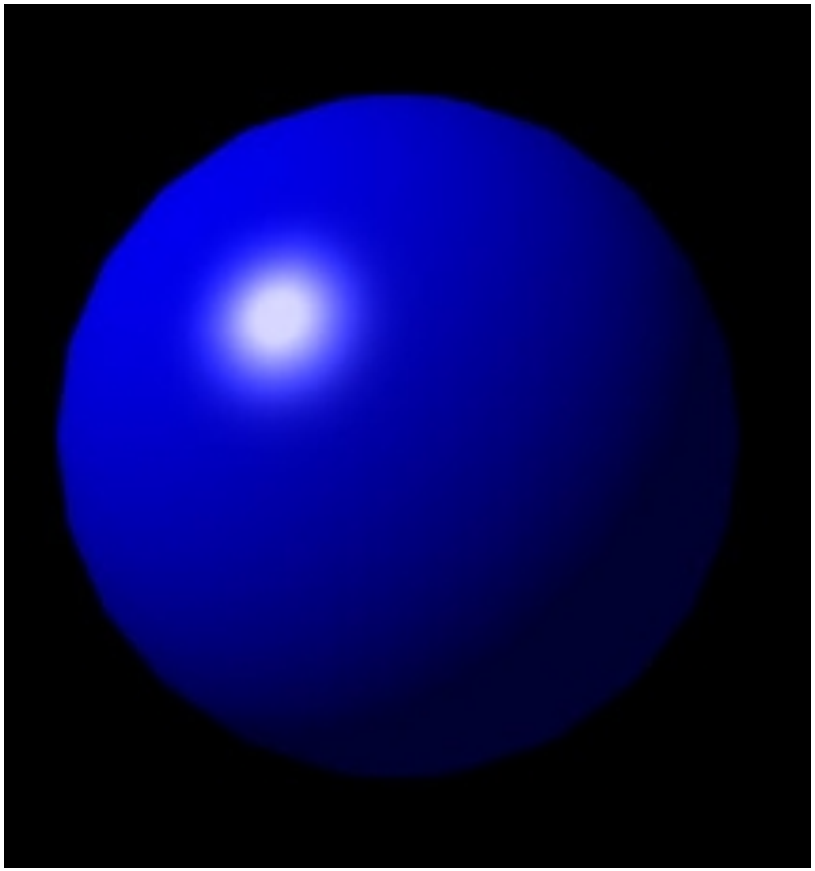
- Average the normals for all the polygons that meet a vertex to calculate its surface normal.
- Compute the color intensities at vertices base on the Lambertian diffuse lighting model.
- Average the color intensities across the faces.





Phong Shading

- Gouraud shading lacks specular highlights except near the vertices.
- Phong shading eliminates these problems.
- Compute vertex normals as in Gouraud shading.
- Interpolate vertex normals to compute normals at each point to be rendered.
- Use these normals to compute the Lambertian diffuse lighting.



http://en.wikipedia.org/wiki/Phong_shading