# CSC418
# Lecture Notes

Yuchen Wang

June 14, 2020

## Contents

# 1 Ray Tracing

## 1.1 Shading

**Notation 1.1.** The important variables in light reflection are unit vectors
Light direction **l**: a unit vector pointing toward the light source;
View direction **v**: a unit vector pointing toward the eye or camera;
Surface normal **n**: a unit vector perpendicular to the surface at the point where reflection is taking place.

### 1.1.1 Lambertian Shading

An observation by Lambert in the 18th century: the amount of energy from a light source that falls on an area of surface depends on the angle of the surface to the light.

**Definition 1.1** (Lambertian shading model)**.** The vector **l** is computed by subtracting the intersection point of the ray and the surface from the light source position.
The pixel color

$$L = k_d I \max(0, \mathbf{n} \cdot \mathbf{l})$$

where $k_d$ is the *diffuse coefficient*, or the surface color; and $I$ is the intensity of the light source.

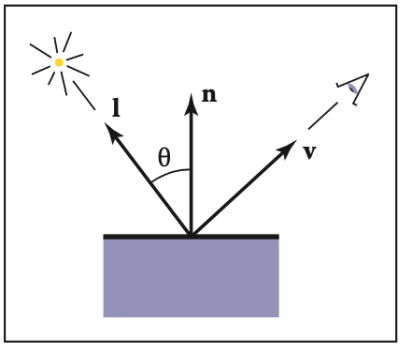



Figure 1: Geometry for Lambertian shading

**Remark 1.1.** Because **n** and **l** are unit vectors, we can use $\mathbf{n} \cdot \mathbf{l}$ as a convenient shorthand for $\cos\theta$. This equation applies separately to the three color channels.

**Remark 1.2.** Lambertian shading is *view independent:* the color of a surface does not depend on the direction from which you look. Therefore it does not produce any highlights and leads to a very matte, chalky appearance.

### 1.1.2 Blinn-Phong Shading

A very simple and widely used model for specular highlights by Phong (1975) and J.F.Blinn (1976).

**Idea** Produce reflection that is at its brightest when $\mathbf{v}$ and $\mathbf{l}$ are symmetrically positioned across the surface normal, which is when mirror reflection would occur; reflection then decreases smoothly as the vectors move away from a mirror configuration.
Compare the half vector $\mathbf{h}$ with $\mathbf{n}$: if $\mathbf{h}$ is near the surface normal, the specular component should be bright and vice versa.

**Definition 1.2** (Blinn-Phong shading model)**.**

$$\mathbf{h} = \frac{\mathbf{v}+\mathbf{l}}{||\mathbf{v}+\mathbf{l}||}$$
$$L = k_d I \max(0, \mathbf{n}\cdot\mathbf{l}) + k_s I \max(0, \mathbf{n}\cdot\mathbf{h})^p$$

where $k_s$ is the *specular coefficient*, or the specular color of the surface, and $p > 1$.

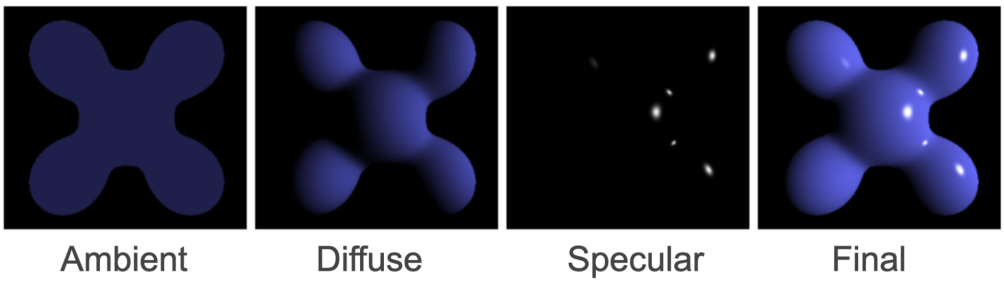
### 1.1.3 Ambient Shading

A heuristic to avoid black shadows is to add a constant component to the shading model, one whose contribution to the pixel color depends only on the object hit, with no dependence on the surface geometry at all, as if surfaces were illuminated by ambient light that comes equally from everywhere.

**Definition 1.3** (simple shading model / Blinn-Phong model with ambient shading)**.**

$$L = k_a I_a + k_d I \max(0, \mathbf{n}\cdot\mathbf{l}) + k_s I \max(0, \mathbf{n}\cdot\mathbf{h})^p$$

where $k_a$ is the surface's ambient coefficient or "ambient color", and $I_a$ is the ambient light intensity.




### 1.1.4 Multiple Point Lights

**Property 1.1** (superposition)**.** The effect by more than one light source is simply the sum of the effects of the light sources individually.

**Definition 1.4** (extended simple shading model)**.**

$$L = k_a I_a + \sum_{i=1}^{N} [k_d I_i \max(0, \mathbf{n}\cdot\mathbf{l}_i) + k_s I_i \max(0, \mathbf{n}\cdot\mathbf{h}_i)^p]$$

where $I_i, \mathbf{l}_i$ and $\mathbf{h}_i$ are the intensity, direction, and half vector of the $i$-th light source.

### 1.2 A Ray-Tracing Program

```
for each pixel do
    compute viewing ray
    if (ray hits an object with t ∈ [0, ∞)) then
        Compute n
        Evaluate shading model and set pixel to that color
    else
        set pixel color to background color
```

### 1.3 Shadows

Recall from 1.1 that light comes from direction $\mathbf{l}$. If we imagine ourselves at a point $\mathbf{p}$ on a surface being shaded, the point is in shadow if we "look" in direction $\mathbf{l}$ and see an object. If there are no objects, then the light is not blocked.

```
function raycolor( ray e + t d, real t0, real t1 )
hit-record rec, srec
if (scene→hit(e + t d, t0, t1, rec)) then
    p = e + (rec.t) d
    color c = rec.k_a  I_a
    if (not scene→hit(p + s l, ε, ∞, srec)) then
        vector3 h = normalized(normalized(l) + normalized(−d))
        c = c + rec.k_d I max (0, rec.n · l) + (rec.k_s) I (rec.n · h)^rec.p
    return c
else
    return background-color
```

**Remark 1.3.** The usual adjustment to avoid the problem of intersecting $\mathbf{p}$ with the surface that generates it is to make the shadow ray check for $t \in [\epsilon, \infty)$ where $\epsilon$ is some small positive constant.

**Remark 1.4.** The code above assumes that $\mathbf{d}$ and $\mathbf{l}$ are not necessarily unit vectors.

### 1.4 Ideal Specular Reflection

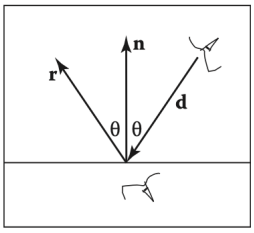



Figure 2: When looking into a perfect mirror, the viewer looking in direction **d** will see whatever the viewer "below" the surface would see in direction **r**

$$\mathbf{r} = \mathbf{d} - 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}$$

In the real world, a surface may reflect some colors more efficiently than others, so it shifts the colors of the objects it reflects (i.e. some energy is lost when the light reflects from the surface).
We implement the reflection by a recursive call of *raycolor*:

```
color c = c + km * raycolor(p + sr, epsilon, max_t);
```

where $k_m$ is the specular RGB color, $p$ is the intersection of the viewing ray and the surface, $s \in [\epsilon, max\_t)$ for the same reason as we did with shadow rays: we don't want the reflection ray to hit the object that generates it.
To make sure that the recursive call will terminate, we need to add a maximum recursion depth.

## 2 Data Structures for Graphics

### 2.1 Spatial Data Structures

#### 2.1.1 Bounding Boxes

**Definition 2.1.** We first consider a 2D ray whose direction vector has positive $x$ and $y$ components. The 2D bounding box is defined by two horizontal and two vertical lines:

$$x = x_{\min}$$

$$x = x_{\max}$$

$$y = y_{\min}$$

$$y = y_{\max}$$

The points bounded by these lines can be described in interval notation:

$$(x, y) \in [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$$

The intersection test can be phrased in terms of these intervals. First, we compute the ray parameter where the ray hits the line $x = x_{\min}$:

$$t_{xmin} = \frac{x_{\min} - x_e}{x_d}$$

We then make similar computations for $t_{xmax}, t_{ymin}$, and $t_{ymax}$. The ray hits the box if and only if the intervals $[t_{xmin}, t_{xmax}]$ and $[t_{ymin}, t_{ymax}]$ overlap.

$$t_{\text{xmin}} = (x_{\min} - x_e)/x_d$$
$$t_{\text{xmax}} = (x_{\max} - x_e)/x_d$$
$$t_{\text{ymin}} = (y_{\min} - y_e)/y_d$$
$$t_{\text{ymax}} = (y_{\max} - y_e)/y_d$$
**if** $(t_{\text{xmin}} > t_{\text{ymax}})$ or $(t_{\text{ymin}} > t_{\text{xmax}})$ **then**
  **return** false
**else**
  **return** true

Figure 3: The algorithm pseudocode.

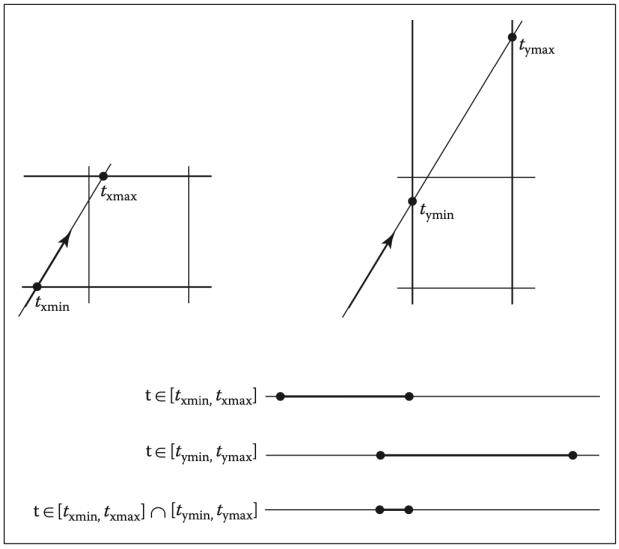



**Figure 12.24.** The ray will be inside the interval $x \in [x_{\min}, x_{\max}]$ for some interval in its parameter space $t \in [t_{\text{xmin}}, t_{\text{xmax}}]$. A similar interval exists for the $y$ interval. The ray intersects the box if it is in both the $x$ interval and $y$ interval at the same time, i.e., the intersection of the two one-dimensional intervals is not empty.

**Issue 1** The first thing we must address is the case when $x_d$ or $y_d$ is negative. If $x_d$ is negative, then the ray will hit $x_{\max}$ before it hits $x_{\min}$. Thus the code for computing $t_{xmin}$ and $t_{xmax}$ expands to:

**if** $(x_d \geq 0)$ **then**
  $t_{\text{xmin}} = (x_{\min} - x_e)/x_d$
  $t_{\text{xmax}} = (x_{\max} - x_e)/x_d$
**else**
  $t_{\text{xmin}} = (x_{\max} - x_e)/x_d$
  $t_{\text{xmax}} = (x_{\min} - x_e)/x_d$

A similar code expansion must be made for the $y$ cases.

**Issue 2** A major concern is that horizontal and vertical rays have a zero value for $y_d$ and $x_d$, respectively. This will cause divide by zero which may be a problem. We define the **rules for divide by zero**:
For any positive real number $a$,

$$+a/0 = +\infty$$
$$-a/0 = -\infty$$

$a = 1/x_d$
**if** $(a \geq 0)$ **then**
  $t_{\min} = a(x_{\min} - x_e)$
  $t_{\max} = a(x_{\max} - x_e)$
**else**
  $t_{\min} = a(x_{\max} - x_e)$
  $t_{\max} = a(x_{\min} - x_e)$

Figure 4: The new code segment.

### 2.1.2 Hierarchical Bounding Boxes

Basic idea: Place an axis-aligned 3D bounding box around all the objects. Since testing for intersection with the box is not free, rays that hit the bounding box will be more expensive to compute than in a brute force search, but rays that miss the box are cheaper. Such bounding boxes can be made hierarchical by partitioning the set of objects in a box and placing a box around each partition.

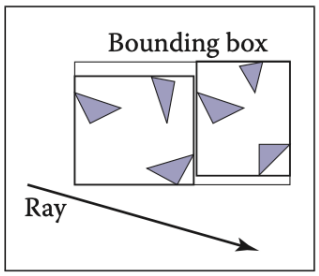



**Figure 12.26.** The bounding boxes can be nested by creating boxes around subsets of the model.

The data structure for the hierarchy might be a tree with the large bounding box at the root and the two smaller bounding boxes as left and right subtrees.

```
if (ray hits root box) then
   if (ray hits left subtree box) then
      check three triangles for intersection
   if (ray intersects right subtree box) then
      check other three triangles for intersection
   if (an intersections returned from each subtree) then
      return the closest of the two hits
   else if (a intersection is returned from exactly one subtree) then
      return that intersection
   else
      return false
else
   return false
```

**Remark 2.1.** Each object belongs to one of two sibling nodes, whereas a point a space may be inside both sibling nodes.

### 2.1.3 Uniform Spatial Subdivision

In spatial division, each point in space belongs to exactly one node, whereas objects may belong to many nodes.
The scene is partitioned into axis-aligned boxes. These boxes are all the same size, although they are not necessarily cubes. The ray traverses these boxes. When an object is hit, the traversal ends.

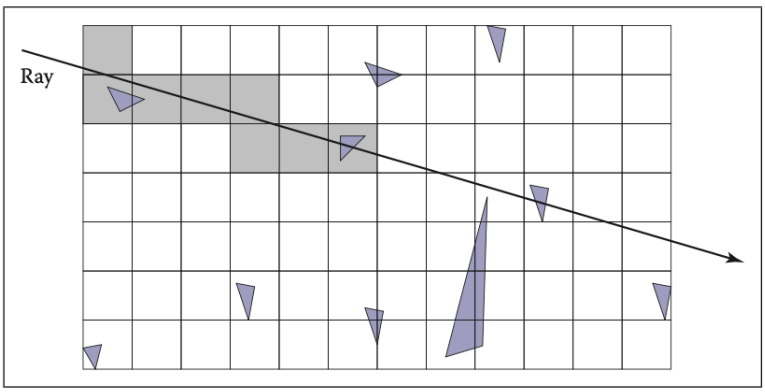



**Figure 12.28.** In uniform spatial subdivision, the ray is tracked forward through cells until an object in one of those cells is hit. In this example, only objects in the shaded cells are checked.

This traversal is done in an increment fashion. We need to find the index $(i, j)$ of the first cell hit by the ray $\mathbf{e} + t\mathbf{d}$. Then we need to traverse the cells in an appropriate order.

## 2.2 Axis-Aligned Binary Space Partitioning

We can also partition space in a hierarchical data structure such as a *binary space partitioning tree* (BSP tree). It's common to use axis-aligned cutting planes for ray intersection.
A node in this structure contains a single cutting plane and a left and right subtree. Each subtree contains all the objects on one side of the cutting plane. Objects that pass through the plane are stored in both subtrees. If we assume the cutting plane is parallel to the $yz$ plane at $x = D$, then the node class is:

```
class bsp-node subclass of surface
    virtual bool hit(ray e + td, real t0, real t1, hit-record rec)
    virtual box bounding-box()
    surface-pointer left
    surface-pointer right
    real D
```

The intersection code can then be called recursively in an object-oriented style. The code considers the four cases:

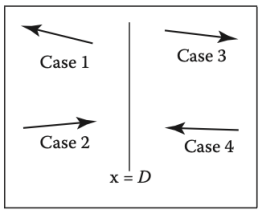



**Figure 12.32.** The four cases of how a ray relates to the BSP cutting plane $x = D$.

The origin of these rays is a point at parameter $t_0$:

$$\mathbf{p} = \mathbf{a} + t_0\mathbf{b}$$

The four cases are:

1. The ray only interacts with the left subtree, and we need not test it for intersection with the cutting plane. It occurs for $x_p < D$ and $x_b < 0$.

2. The ray is tested against the left subtree, and if there are no hits, it is then tested against the right subtree. We need to find the ray parameter at $x = D$, so we can make sure we only test for intersections within the subtree. This case occurs for $x_p < D$ and $x_b > 0$.

3. This case is analogous to case 1 and occurs for $x_p > D$ and $x_b > 0$.

4. This case is analogous to case 2 and occurs for $x_p > D$ and $x_b < 0$.

The resulting traversal code handling these cases in order is:

```
function bool bsp-node::hit(ray a + tb, real t0, real t1,
                            hit-record rec)
  xp = xa + t0xb
  if (xp < D) then
    if (xb < 0) then
      return (left ≠ NULL) and (left→hit(a + tb, t0, t1, rec))
    t = (D − xa)/xb
    if (t > t1) then
      return (left ≠ NULL) and (left→hit(a + tb, t0, t1, rec))
    if (left ≠ NULL) and (left→hit(a + tb, t0, t, rec)) then
      return true
    return (right ≠ NULL) and (right→hit(a + tb, t, t1, rec))
  else
    analogous code for cases 3 and 4
```