CS 543 - Computer Graphics: Ray Tracing Detail, Part 3

by
Robert W. Lindeman
gogo@wpi.edu
(with help from Emmanuel Agu ;-)

`hit( )` Function for Sphere

- Recall that for generic sphere, there are two hit times, t_1 and t_2 corresponding to the solutions

  $$t_h = -\frac{B}{A} \pm \frac{\sqrt{B^2 - AC}}{A}$$

  which are the solutions to the quadratic equation $At^2 + 2Bt + C = 0$, where $A = |\mathbf{c}|^2$, $B = S.\mathbf{c}$ and $C = |S|^2 - 1$
- Thus the **`hit( )`** function for a sphere is as follows:

```
Bool Sphere::hit( Ray &r, Intersection inter ) {
  Ray genRay;   // need to make the generic ray
  xfrmRay( genRay, invTransf, r );
  double A, B, C
  ...
```

`hit( )` Function for Sphere

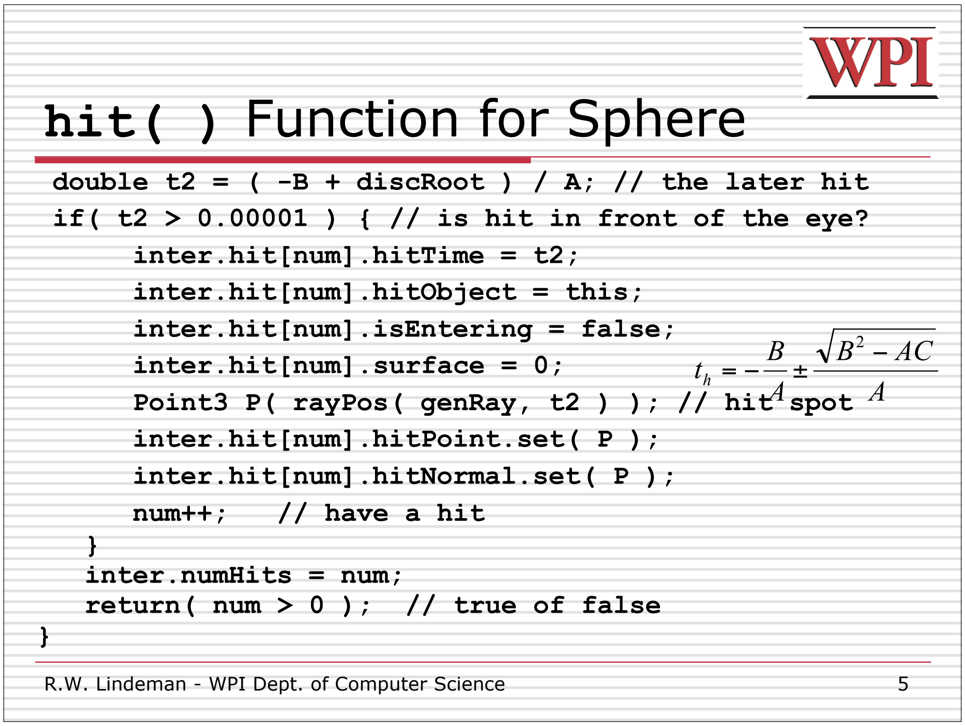
```
A = dot3D( genRay.dir, genRay.dir );
B = dot3D( genRay.start, genRay.dir );
C = dot3D( genRay.start, genRay.start ) - 1.0;

double discrim = B * B - A * C;
if( discrim < 0.0 )  { // ray misses
  return( false );
}
int num = 0;   // the # of hits so far
double discRoot = sqrt( discrim );
double t1 = ( -B - discRoot ) / A; // the earlier hit
...
```

$$t_h = -\frac{B}{A} \pm \frac{\sqrt{B^2 - AC}}{A}$$

`hit( )` Function for Sphere

```
if( t1 > 0.00001 )  { // is hit in front of the eye?
    inter.hit[0].hitTime = t1;
    inter.hit[0].hitObject = this;
    inter.hit[0].isEntering = true;
    inter.hit[0].surface = 0;
    Point3 P(rayPos(genRay, t1)); // hit spot
    inter.hit[0].hitPoint.set(P);
    inter.hit[0].hitNormal.set(P);
    num = 1;   // have a hit
}
```

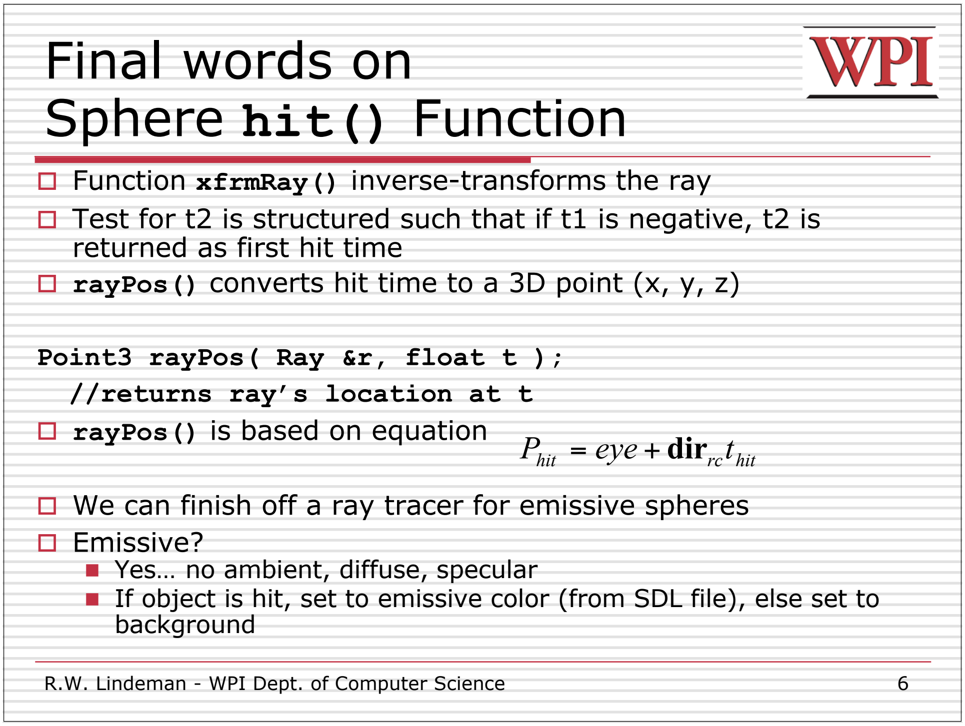
$$t_h = -\frac{B}{A} \pm \frac{\sqrt{B^2 - AC}}{A}$$



`hit( )` Function for Sphere

```
double t2 = ( -B + discRoot ) / A; // the later hit
if( t2 > 0.00001 ) { // is hit in front of the eye?
     inter.hit[num].hitTime = t2;
     inter.hit[num].hitObject = this;
     inter.hit[num].isEntering = false;
     inter.hit[num].surface = 0;
     Point3 P( rayPos( genRay, t2 ) ); // hit spot
     inter.hit[num].hitPoint.set( P );
     inter.hit[num].hitNormal.set( P );
     num++;   // have a hit
  }
  inter.numHits = num;
  return( num > 0 );  // true of false
}
```

$$t_h = -\frac{B}{A} \pm \frac{\sqrt{B^2 - AC}}{A}$$



Final words on Sphere `hit()` Function

- Function **`xfrmRay()`** inverse-transforms the ray
- Test for t2 is structured such that if t1 is negative, t2 is returned as first hit time
- **`rayPos()`** converts hit time to a 3D point (x, y, z)

```
Point3 rayPos( Ray &r, float t );
  //returns ray's location at t
```

- **`rayPos()`** is based on equation $P_{hit} = eye + \mathbf{dir}_{rc} t_{hit}$
- We can finish off a ray tracer for emissive spheres
- Emissive?
  - Yes… no ambient, diffuse, specular
  - If object is hit, set to emissive color (from SDL file), else set to background

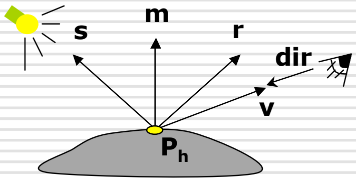
Emissive `shade( )` Function

```
Color3 Scene::shade( Ray& ray )  { // is hit in front of the eye?
  Color3 color;
  Intersection best;
  getFirstHit(ray, best);
  if( best.numHits == 0 ) return( background );
  Shape* myObj = ( Shape* )best.hit[0].hitObject; // hit object
  color.set( myObj->mtrl.emissive );
  return( color );
}
```

- Need hit functions for more shapes (cube, square, cylinder, etc.)

Adding Ambient, Diffuse, Specular to `shade( )` Function

- Recall Phong's illumination model



$$I = I_a k_a + I_d k_d \times \mathbf{lambert} + I_{sp} k_s \times \mathbf{phong}^f$$

- Where light vector **s** = Light positon – hit Point
- View vector **v = -dir**

$$\mathbf{lambert} = \max\left(0, \frac{\mathbf{s} \bullet \mathbf{m}}{|\mathbf{s}||\mathbf{m}|}\right) \qquad \mathbf{phong} = \max\left(0, \frac{\mathbf{h} \bullet \mathbf{m}}{|\mathbf{h}||\mathbf{m}|}\right)$$

Adding Ambient, Diffuse, Specular to `shade( )` Function

- **h** is Blinn's halfway vector given by **h = s + v**
- To handle colored lights and object surfaces, we separate the equation

$$I = I_a k_a + I_d k_d \times \mathbf{lambert} + I_{sp} k_s \times \mathbf{phong}^f$$

- into separate R G and B parts so that

$$I_r = I_{ar} k_{ar} + I_{dr} k_{dr} \times \mathbf{lambert} + I_{spr} k_{sr} \times \mathbf{phong}^f$$
$$I_g = I_{ag} k_{ag} + I_{dg} k_{dg} \times \mathbf{lambert} + I_{spg} k_{sg} \times \mathbf{phong}^f$$
$$I_b = I_{ab} k_{ab} + I_{db} k_{db} \times \mathbf{lambert} + I_{spb} k_{sb} \times \mathbf{phong}^f$$

- Lambert and phong terms use transformed object normal **m** at hit point
- *How do we get transformed normal?*

Finding Normal at Hit Spot

- *How do we get transformed normal?*
- We set generic object normal at hit point
- E.g., in sphere hit function, set hit point normal = hit point for generic sphere
  - We did

```
inter.hit[0].hitNormal.set(P);
```

So, we have normal for generic object **m'**

- To get transformed object normal **m**, simply (see section 6.5.3)

$$\mathbf{m} = M^{-T}\mathbf{m}'$$

Adding Ambient, Diffuse, Specular to **shade( )** Function

- You specify ambient, diffuse and specular values of materials in your SDL file
- Each cylinder, cube, etc. is a GeomObj class
- GeomObj class has material **mtrl** as member
- Your ray tracer can then access ka, kd … as **mtrl.ambient**, **mtrl.diffuse** and **mtrl.specular**
- For more realistic look, can use carefully measure values from McReynolds and Blythe.

```
! first define values
def Copper{ ambient 0.19125 0.0735 0.0225
            diffuse 0.7038 0.27048 0.0828
            specular 0.256777 0.137622 0.086014 exponent 12.8}

!then use defined values
Use Copper sphere
```

Adding Ambient, Diffuse, Specular to **shade( )** Function

- Can now define full shade function with ambient, diffuse and specular contributions

```
Color3 Scene::shade( Ray& ray ) { // is hit in front of the eye?
  Get the first hit using getFirstHit( r, best );
  Make handy copy h = best.hit[0];// data about first hit
  Form hitPoint based on h.hitTime
  Form v = -ray.dir; // direction to viewer
  v.normalize( );

  Shape* myObj = ( Shape * )h.hitObject; // point to hit object
  Color3 color( myObj->mtrl.emissive ); // start with emissive
  color.add( ambient contribution ); // compute ambient color
```

Adding Ambient, Diffuse, Specular to **shade( )** Function

$$\textbf{lambert} = \max\left(0, \frac{\mathbf{s} \bullet \mathbf{m}}{|\mathbf{s}||\mathbf{m}|}\right)$$

```
Vector3 normal;

// transform the generic normal to the world normal
xfrmNormal( normal, myObj->invTransf, h.hitNormal );
normal.normalize( );  // normalize it
for( each light source, L )  { // sum over all sources
  if(isInShadow(..)) continue; // skip L if it’s in shadow
  Form s = L.pos – hitPoint; // vector from hit pt to src
  s.normalize( );
  float mDotS = s.dot( normal );   // Lambert term
  if(mDotS > 0.0 ) {  // hit point is turned toward the light
    Form diffuseColor = mDotS * myObj->mtrl.diffuse * L.color
    color.add( diffuseColor );  // add the diffuse part
  }
```

Adding Ambient, Diffuse, Specular to **shade( )** Function

$$\textbf{phong} = \max\left(0, \frac{\mathbf{h} \bullet \mathbf{m}}{|\mathbf{h}||\mathbf{m}|}\right)$$

```
    Form h = v + s; // the halfway vector
    h.normalize( );
    float mDotH = h.dot( normal ); // part of phong term
    if( mDotH <= 0 ) continue; // no specular contribution

    float phong = pow( mDotH, myObj->mtrl.specularExponent );
    specColor = phong * myObj->mtrl.specular * L.color;
    color.add( specColor );
  }
  return( color );
}
```

- ☐ isInShadow( ) is function to tests if point is in shadow. Implement next!

Adding Shadows to Raytracing

- Shadows are important visual cues for humans
- Previously discussed limited shadow algorithms
- Limited due to OpenGL
- Raytracing adds shadows with little programming effort
- So far, all hit points rendered with all shading components (ambient, diffuse, specular, emissive)
- If hit point is in shadow, render using only ambient (and emissive). Leave out specular and diffuse
- 3 possible cases
  - A: no other object between hit point and light source
  - B: another object between hit point and light source (occlusion)
  - C: object blocks itself from light source (back face)

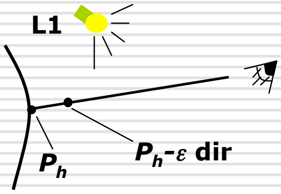
Adding Shadows

- Need routine `isInShadow()` which tests to see if hit point is in shadow
- `isInShadow()` returns
  - `true` if hit point is in shadow
  - `false` otherwise
- `isInShadow()` spawns new ray called **shadow feeler** emanating from hit point at time t=0 and reaching light source at t=1
- So, parametric equation of shadow feeler is

  $P_h + (L - P_h)t$
- So, shadow feeler is built, and each object in object list is scanned for intersection (just like eye ray)
- If any valid intersection in time range t=[0,1] `isInShadow()` returns true, otherwise returns false

Adding Shadows

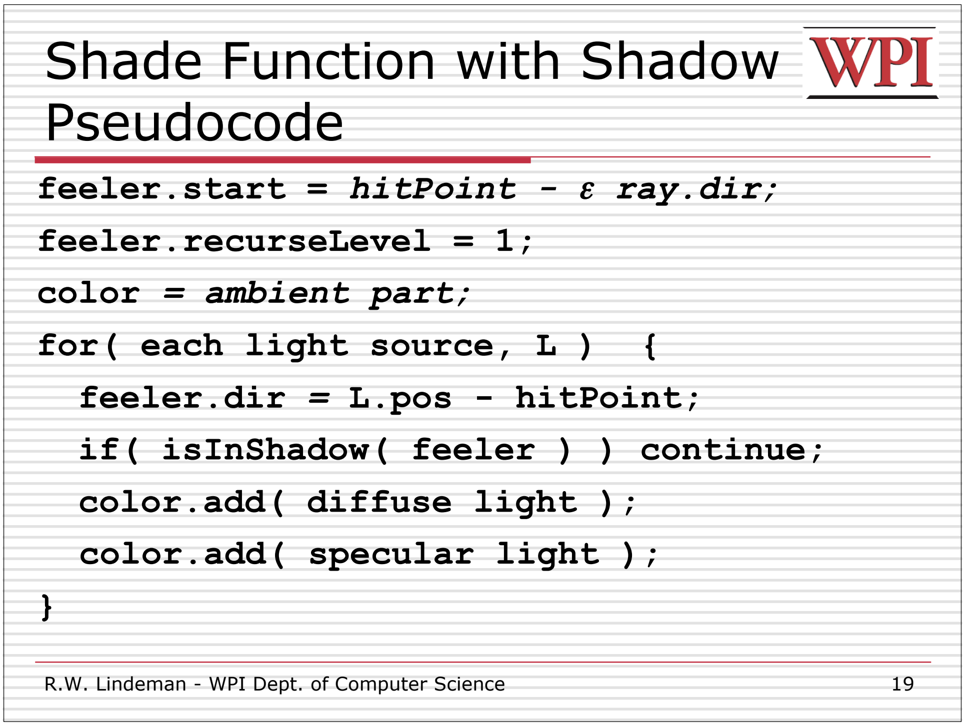
- **Note**: since we made hit function general, takes ray as argument, once we build shadow feeler, reuse hit( ) functions
- One more sticky point: self-shadowing!!
- How? Since shadow feeler starts at hit point at t=0, `isInShadow()` always intersects with object itself (returns true)
- Can fix this by starting shadow ray slightly away from hit point. E.g., in figure, start shadow feeler starts at **P_h-ε dir**



Note: feeler is ε toward eye NOT light

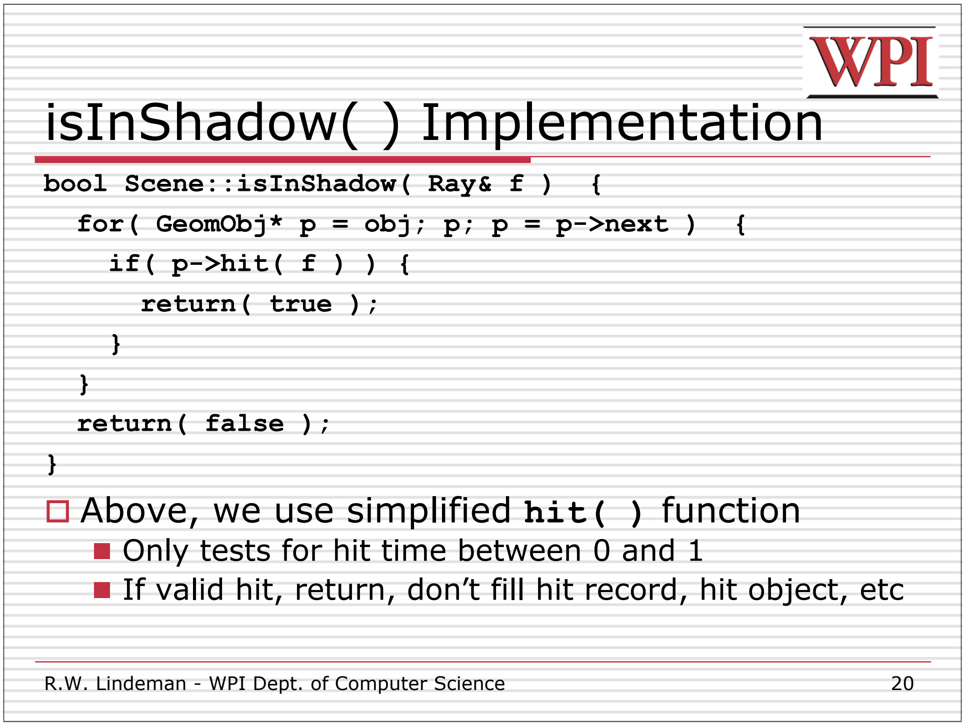
Adding Shadows

- How to put this back into `shade( )` function?
- After `getFirstHit( )` returns closest hit point, add ambient component
- Next, build shadow feeler (per light source) with start point of **P_h-ε dir**
- Feeler direction is set to `(Light position – feeler start)`
- Call `isInShadow( feeler )` to determine object intersections (and hit times)
- If any valid intersections with object (t between 0 and 1), diffuse and specular components are skipped else add them
- Variable `recurseLevel` is used to control how many times `hit( )` function can call itself. Set it to 1 for shadow ray
- More on `recurseLevel` when we discuss reflection



Shade Function with Shadow Pseudocode

```
feeler.start = hitPoint - $\varepsilon$ ray.dir;
feeler.recurseLevel = 1;
color = ambient part;
for( each light source, L )  {
  feeler.dir = L.pos - hitPoint;
  if( isInShadow( feeler ) ) continue;
  color.add( diffuse light );
  color.add( specular light );
}
```



isInShadow( ) Implementation

```
bool Scene::isInShadow( Ray& f )  {
  for( GeomObj* p = obj; p; p = p->next )  {
    if( p->hit( f ) ) {
      return( true );
    }
  }
  return( false );
}
```

- Above, we use simplified `hit( )` function
  - Only tests for hit time between 0 and 1
  - If valid hit, return, don't fill hit record, hit object, etc

References

- Hill, chapter 12