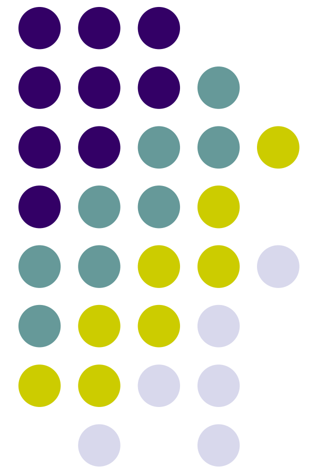
Computer Graphics (CS 543)
Lecture 11 (Part 1): Ray Tracing (Part 3)

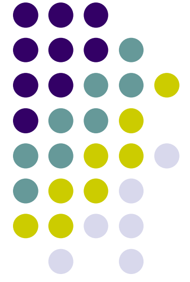
Prof Emmanuel Agu

Computer Science Dept.

Worcester Polytechnic Institute (WPI)

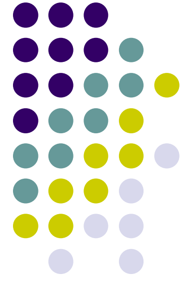


Recall: Where are we?



```
Define the objects and light sources in the scene
Set up the camera
for(int r = 0; r < nRows; r+= blockSize){
   for(int c = 0; c < nCols; c+= blockSize){
         1. Build the rc-th ray
         2. Find all object intersections with rc-th ray
         3. Identify closest object intersection
         4. Compute the "hit point" where the ray hits the
             object, and normal vector at that point
         5. Find  color (clr) of light to eye along ray
         color_rect(r, g, b), r, c, blockSize);
   }
}
```

Recall: Find Object Intersections with rc-th ray



- Much of work in ray tracing lies in finding intersections with generic objects
- Break into two parts
  - Deal with untransformed, generic (dimension 1) shape
  - Then embellish to deal with transformed shape
- Ray generic object intersection best found by using implicit form of each shape. E.g. generic sphere is

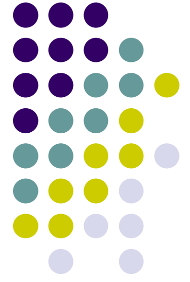
$$F(x, y, z) = x^2 + y^2 + z^2 - 1$$

- Approach: ray r(t) hits a surface when its implicit eqn = 0
- So for ray with starting point S and direction **c**

$$r(t) = S + \mathbf{c}t$$

$$F(S + \mathbf{c}t_{hit}) = 0$$

Recall: Ray Intersection with Generic Sphere



- Generic sphere has form

$$x^2 + y^2 + z^2 = 1$$

$$x^2 + y^2 + z^2 - 1 = 0$$

$$F(x, y, z) = x^2 + y^2 + z^2 - 1$$

$$F(P) = |P|^2 - 1$$

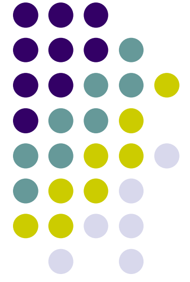
- Substituting S + **c**t in *F(P) = 0,* we get

$$|S + \mathbf{c}t|^2 - 1 = 0$$

$$|\mathbf{c}|^2 t^2 + 2(S \cdot \mathbf{c})t + (|S|^2 - 1) = 0$$

- This is a quadratic equation of the form $At^2 + 2Bt + C = 0$ where $A = |\mathbf{c}|^2$, $B = S.\mathbf{c}$ and $C = |S|^2 - 1$

Recall: Ray Intersection with Generic Sphere

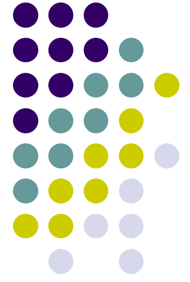


- Solving

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

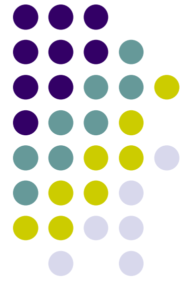
- If discrimant $B^2 - AC$ is negative, no solutions, ray misses sphere
- If discriminant is zero, ray grazes sphere at one point and hit time is –B/A
- If discriminant is +ve, two hit times t1 and t2 (+ve and –ve) discriminant

What about transformed Objects



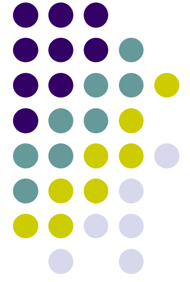
- Generic objects are untransformed:
  - No translation, scaling, rotation
- Real scene: generic objects instantiated, then transformed by a composite matrix *T,*
- We can easily find the inverse transform *T'*
- **Problem definition:** We want to find ray intersection with transformed object
- Easy by just simply finding the implicit form of the transformed object
- May be tough to find implicit form of transformed object
- Hmmm… is there an easier way?

What about transformed Objects



- Yes! Basic idea: if object is transformed by *T*, then ray–object intersection is the same as inverse transformed ray with generic object
- Algorithm
  - Find *T'* from initial *T* transform matrix of object
  - Inverse transform the ray to get (S' + $\mathbf{c}'t$)
  - Find intersection time, t_{hit} of the ray with the generic object
  - Use the *same* t_{hit} in S + $\mathbf{c}t$ to identify the actual hit point
- This beautiful trick greatly simplifies ray tracing
- We only need to come up with code that intersects ray with *generic* object
- Remember that programmer does transforms anyway, so we can easily track and get *T*

Dealing with Transformed Objects



- Thus we want to solve the equation

$$F(T^{-1}(S+\mathbf{c}t))=0$$

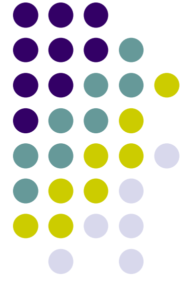
- Since transform *T* is linear

$$T^{-1}(S+\mathbf{c}t)=(T^{-1}S)+(T^{-1}\mathbf{c})t$$

- Thus inverse transformed ray is

$$\tilde{r}(t)=M^{-1}\begin{pmatrix}S_x\\S_y\\S_z\\1\end{pmatrix}+M^{-1}\begin{pmatrix}c_x\\c_y\\c_z\\0\end{pmatrix}t=\tilde{S}'+\mathbf{c}'t$$

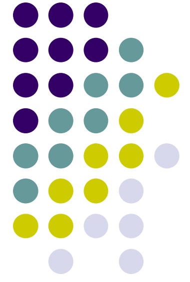
Dealing with transformed Objects



- So, for each final CTM M transform matrix, we need to calculate its inverse transform M^{-1}
- Example transform matrices and its inverse are

$$M = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 4 & 0 & 4 \\ 0 & 0 & 4 & 9 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad M^{-1} = \begin{pmatrix} 1 & 0 & 0 & -2 \\ 0 & \frac{1}{4} & 0 & -4 \\ 0 & 0 & \frac{1}{4} & -\frac{9}{4} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

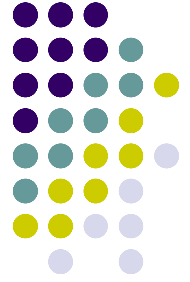
Organizing a Ray Tracer



- Need data structures to store ray, scene, camera, etc
- There are many ways to organize ray tracer
- Previously in C, declare **struct**
- These days, object-oriented religion?
- Friend once wrote ray tracer as java applet in Prof. Hill's class
- We've developed camera class (HW3: slide, roll, etc)
- Now just add a **raytrace** method to camera class

```
void Camera::raytrace(int blockSize);
```

Organizing a Ray Tracer

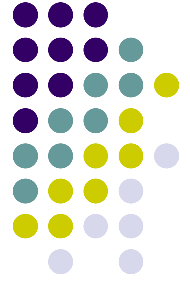


- Call camera raytrace method from display (redisplay) function

```
void display(void){
   glClear(GL_COLOR_BUFFER_BIT); // clear the screen
   cam.raytrace(blockSize); // generates NxN image
   glDrawArrays(GL_TRIANGLES, 0, 6); // draws rectangle
}
```

- Thus ray tracer fires up and starts scanning pixel by pixel (or block by block) till entire screen is ray traced
- Subtlety: we raytrace to generate texture, map texture onto rectangle, then draw!!

Organizing a Ray Tracer

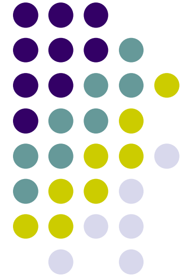


- Need Ray class with start, dir variables and methods to set them

```
Class Ray{
Public:
   points3 start;
   vec3 dir;
   void setStart(point3& p){start.x = p.x; etc…}
   void setDir(Vector3& v){dir.x = v.x; etc…}
   // other fields and methods
};
```

- We can now develop a basic raytrace( ) skeleton function

Camera raytrace( ) skeleton



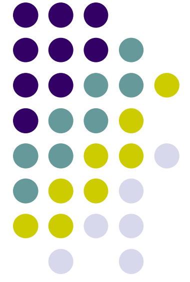
```
 Glfloat image[N][M][3];

 // insert other VBO, VAO, texture and rectangle setup


void Camera::raytrace(Scene& scn, int blockSize)
{
   Ray theRay;
   Color3 clr;
   theRay.setStart(eye);


   //begin ray tracing
```

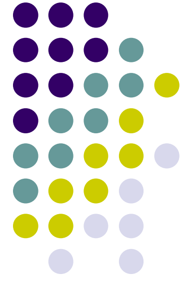
Camera raytrace( ) skeleton



```
for(int row = 0; row < nRows; rows += blockSize)
for(int col = 0; col < nCols; cols += blockSize)
{
    compute ray direction
    theRay.setDir(<direction>); // set the ray's direction
    clr.set(shade(theRay));  // find the color
    color_rect(clr.red, clr.green, clr.blue), row, col,
                                           blockSize);
}
}
```

- shade( ) function does most of ray tracing work

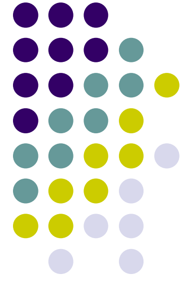
shade( ) skeleton



```
Color3 shade(Ray& ray)
{   // return color of this ray
    Color3 color; // total color to be returned
    Intersection best; // data for best hit so far
    Hit(ray, best); // Only sphere,. fill "best" record
    if(best.numHits == 0) // did ray miss all objects?
        return background;
    color.set(the emissive color of object);
    color.add(ambient, diffuse and specular); // add contrib.
    color.add(reflected and refracted components);
    return color;
}
```

- Intersection class used to store each object's hit information

shade( ) skeleton

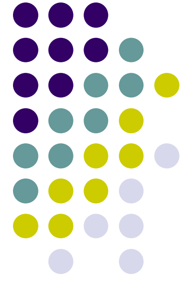


- Intersection class used to store each object's hit information

```
Class Intersection{
Public:
    int numHits;     // # of hits at positive hit times
    HitInfo hit[8]; //list of hits – may need more than 8 later
    …. various hit methods
}
```

- hitInfo stores actual hit information for each hit
- For simple convex objects (e.g. sphere) at most 2 hits
- For torus up to 4 hits
- For boolean objects, all shapes possible so no limit to number of hits

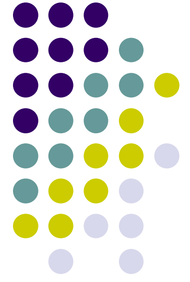
HitInfo( ) class



```
class HitInfo{
Public:
   double hitTime;        // the hit time
   bool isEntering;       // is the ray entering or exiting
   int surface;           // which surface is hit?
   points3 hitPoint;       // hit point
   vec3 hitNormal;    // normal at hit point
   …. various hit methods
}
```

- Surface applies if it is convenient to think of object as multiple surfaces. E.g. cylinder cap, base and side are 3 different surfaces

hit( ) Function for Sphere



- Recall that for generic sphere, there are two hit times, t1 and t2 corresponding to the solutions

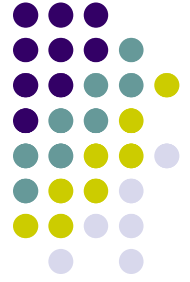
$$t_h = -\frac{B}{A} \pm \frac{\sqrt{B^2 - AC}}{A}$$

- 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
```

$A = |\mathbf{c}|^2$, $B = S.\mathbf{c}$ and $C = |S|^2 - 1$

hit( ) Function for Sphere



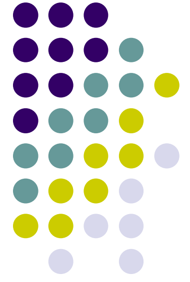
```
A = dot3D(genRay.dir, genRay.dir);
B = dot3D(genRay.start, genRay.dir);
C = dot3D(genRay.start, genRay.start) - 1.0;

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

…………………………..
```

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

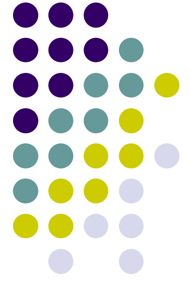
Hit( ) Function for Sphere



```
If(t1 > 0.00001) // is hit in front of the eye?
{
    inter.hit[0].hitTime = t1;
    inter.hit[0].isEntering = true;
    inter.hit[0].surface = 0;
    points3 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

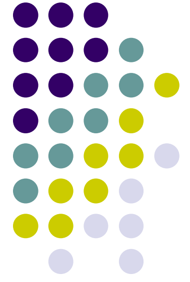


```
double t2 = (-B + discRoot)/A; // the later hit

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

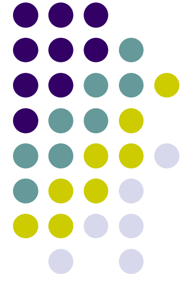


- Function **xfrmRay( )** inverse transforms the ray
- Test for t2 is structured such that if t1 is negative, t2 is returned as first hit time
- **rayPos** converts hit time to a 3D point (x, y, z)

```
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 sphere
- Emissive?
  - Yes… no ambient, diffuse, specular
  - If object is hit, set to emissive color of sphere else set to background

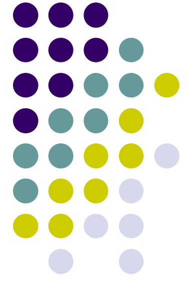
Emissive shade( ) Function



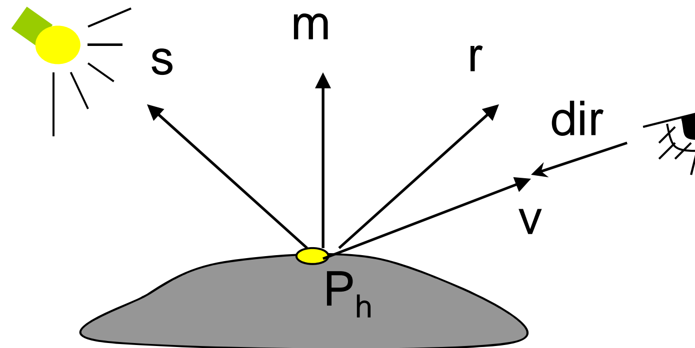
```
Color3 shade(Ray& ray) // is hit in front of the eye?
{
    Color3 color;
    Intersection best;
    Hit(ray, best);
    if(best.numHits == 0) return background;
    color = emissive color of sphere;
    return color;
}
```

- Need hit functions for more shapes (cube, square, cylinder, etc)
- At this point, will take things out of order..
- Next, add ambient diffuse, specular
- Return later to do more intersections

Adding Ambient, Diffuse, Specular to shade( ) Function



- Recall Phong's illumination model

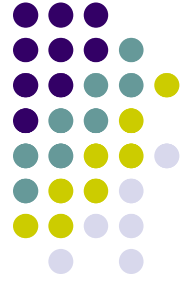


$$I = I_a k_a + I_d k_d \times \mathbf{lambert} + I_{sp} k_s \times \mathbf{phong}^f$$

- 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

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

into separate R G and B parts so that

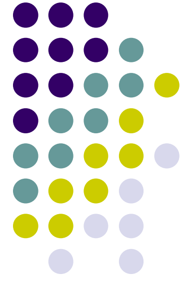
$$I_r = I_{ar} k_{ar} + I_{dr} k_{dr} \times \textbf{lambert} + I_{spr} k_{sr} \times \textbf{phong}^f$$

$$I_g = I_{ag} k_{ag} + I_{dg} k_{dg} \times \textbf{lambert} + I_{spg} k_{sg} \times \textbf{phong}^f$$

$$I_b = I_{ab} k_{ab} + I_{db} k_{db} \times \textbf{lambert} + I_{spb} k_{sb} \times \textbf{phong}^f$$

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

Finding Normal at Hit Spot



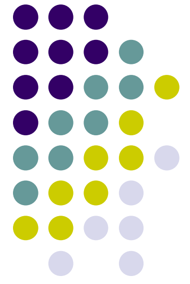
- *How do we get transformed normal?*
- We set generic object normal at hit point
- E.g. in sphere hit function, set hit point normal = hit point for generic sphere, we did

```
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)

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

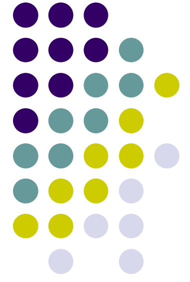
Adding Ambient, Diffuse, Specular to shade( ) Function



- You specify ambient, diffuse and specular values for realistic materials
- For more realistic look, can use carefully measure values from McReynolds and Blythe.
- E.g. Copper parameters

```
ambient 0.19125 0.0735 0.0225
diffuse 0.7038 0.27048 0.0828
specular 0.256777 0.137622 0.086014 exponent 12.8
```

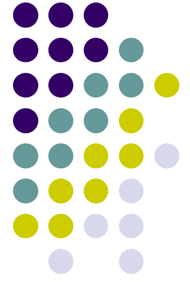
Recall: Coefficients for Real Materials



Material	Ambient Kar, Kag,kab	Diffuse Kdr, Kdg,kdb	Specular Ksr, Ksg,ksb	Exponent, α
Black plastic	0.0 0.0 0.0	0.01 0.01 0.01	0.5 0.5 0.5	32
Brass	0.329412 0.223529 0.027451	0.780392 0.568627 0.113725	0.992157 0.941176 0.807843	27.8974
Polished Silver	0.23125 0.23125 0.23125	0.2775 0.2775 0.2775	0.773911 0.773911 0.773911	89.6

Figure 8.17, Hill, courtesy of McReynolds and Blythe

Adding Ambient, Diffuse, Specular to shade( ) Function

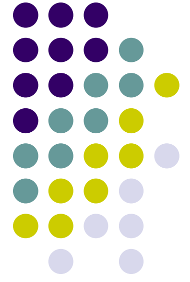


- Can now define full shade function with ambient, diffuse and specular contributions

```
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( );

  Color3 color(emissive color of sphere); // start with emissive
   color.add(ambient contribution); // compute ambient color
```

Adding Ambient, Diffuse, Specular to shade( ) Function

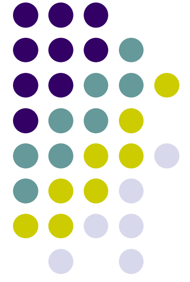


$$\textbf{lambert} = \max\left(0, \frac{\mathbf{s} \bullet \mathbf{m}}{|\mathbf{s}||\mathbf{m}|}\right)$$

```
Vector3 normal;

// transform the generic normal to the world normal
xfrmNormal(normal, 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 * diffuse * L.color
   color.add(diffuseColor);  // add the diffuse part
 }
```

Adding Ambient, Diffuse, Specular to shade( ) Function



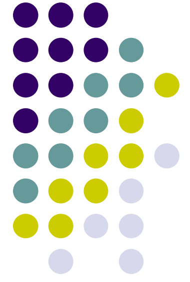
$$\mathbf{phong} = \max\left(0, \frac{\mathbf{h} \bullet \mathbf{m}}{|\mathbf{h}||\mathbf{m}|}\right)$$

```
    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, specularExponent);
    specColor = phong * specular * L.color;
    color.add(specColor);
  }
  return color;
}
```

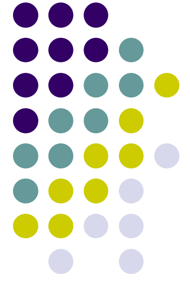
- isInShadow( ) is function to tests if point is in shadow. Implement next!

Adding Shadows to Raytracing



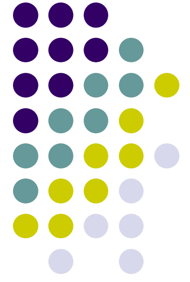
- Shadows are important visual cues for humans
- Previously discussed limited shadow algorithms
- Limited due to OpenGL
- Raytracing adds shadows with little programming effort
- So far, all hit points rendered with all shading components (ambient, diffuse, specular, emissive)
- Now add shadows

Adding Shadows to Raytracing



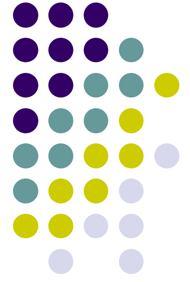
- If hit point is in shadow, render using only ambient (and emissive). Leave out specular and diffuse
- 3 possible cases
  - A: no other object between hit point and light source
  - B:another object between hit point and light source (occlusion)
  - C: object blocks itself from light source (back face)

Adding Shadows



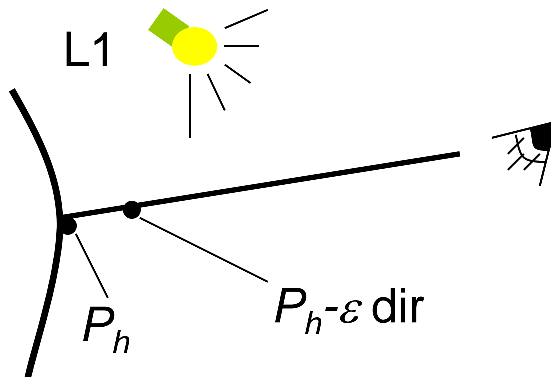
- Need routine **`isInShadow( )`** which tests to see if hit point is in shadow
- **`isInShadow( )`** returns
  - **`true`** if hit point is in shadow
  - **`false`** otherwise
- **`isInShadow( )`** spawns new ray called **shadow feeler** emanating from hit point at time t=0 and reaching light source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



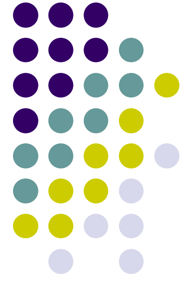
- **Note**: since we made hit function general, takes ray as argument, once we build shadow feeler, reuse hit( ) functions
- One more sticky point: self-shadowing!!
- How? Since shadow feeler starts at hit point at t=0, **isInShadow** always intersects with object itself (returns true)
- Can fix this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

  E.g. in figure, start shadow feeler starts at $\boldsymbol{P_h - \varepsilon}$ **dir**



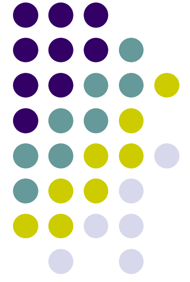
Note: feeler is ε toward eye NOT light

Adding Shadows



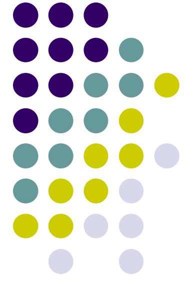
- How to put this back into shade( ) function?
- After **getFirstHit( )** returns closest hit point, add ambient component
- Next, build shadow feeler (per light source) with start point of $P_h - \varepsilon$ **dir**
- Feeler direction is set to **(Light position – feeler start)**
- Call **isInShadow(feeler)** 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 - ε 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);
}
```

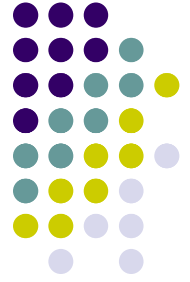
isInShadow( ) Implementation



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

- For real scene objects stored in linked list of **GeomObj**
- For simple scene, just check hardcoded objects
- 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 and Kelley, Computer Graphics using OpenGL, 3rd edition, Chapter 12