EDAN30 Photorealistic Computer Graphics

Seminar 1

Whitted Ray Tracing

Magnus Andersson, PhD student (magnusa@cs.lth.se)

Structure of Assignments

- Starting point: Ray tracer *skeleton*
- Evolved throughout the course
- Each assignment will build upon the last
  - *No (or little) ”fresh” code releases in-between assignments!*
  - *I.e. You will continue to use your own code in the following assignments*

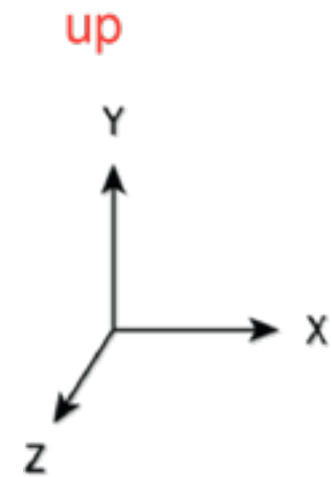
Structure of Assignments

- Starting point: Ray tracer *skeleton*
- Evolved throughout the course
- Each assignment will build upon the last
  - *No (or little) ”fresh” code releases in-between assignments!*
  - *I.e. You will continue to use your own code in the following assignments*

- ***Write good code ;)***

prTracer Overview

- Written in C++
- Mac / Windows
- Right handed coordinate system



- Convenient utility classes
  - *Vector, Matrix, Point, Color, Image, ...*
- Output format
  - *OpenEXR (.exr)*

Color class

RGB Color

```
Color a = Color(1.0f, 0.0f, 0.0f);
Color b(0.3f, 0.9f, 0.4f);
```

Arithmetic operations

```
Color c, d;
d = a * b;                // Component-wise multiplication
c = (a + b) / 2.0f;       // Average of two colors
```

Vector class

Direction in 3D space

```
Vector3D w = (0.5f, 1.0f, 0.3f);
Vector3D v(1.0f, 0.7f, 1.5f);
```

Intuitive math operations (+, -, *, /, +=, etc)

```
Vector3D q = 2.0f * u - v + w;
q = v * w;                    // Dot product (.dot())
q = v % w;                    // Cross product (.cross())
q.normalize();                // Normalize length to 1
```

Point class

Position in 3D space

```
Point3D p = (0.5f, 1.0f, 0.3f);
Point3D q(1.0f, 0.7f, 1.5f);
```

Similar math operations (+, -, *, /, +=, etc)

```
Vector3D d = Vector(0.0f, 10.0f, -5.0f);
Point3D r = p + d;                // Point = Point + Vector
Vector3D w = r - p;               // Vector from p to r
```

Common operations

Debug printout

```
std::cout << p << std::endl;// p can be Point, Vector or Color
```

Element access

```
float x = p.x;                    // Vector or Point
float r = c.r;                    // Color
```

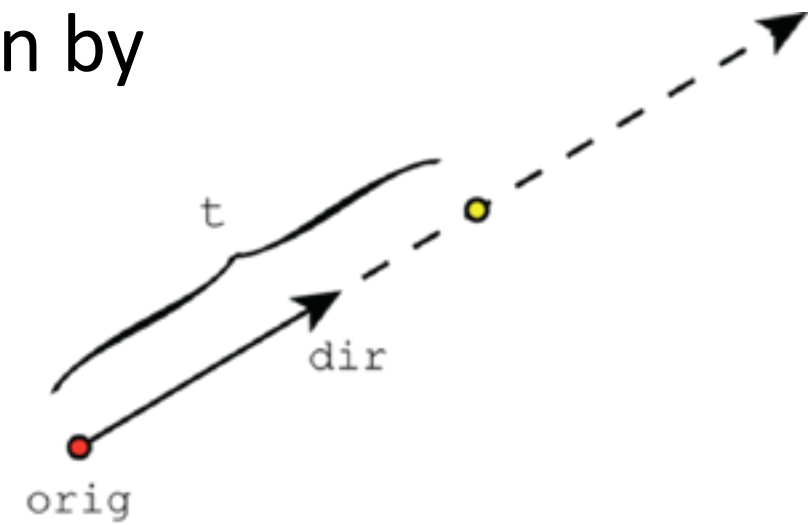
Ray class

Class describing a **ray** in 3D

- Origin (*Point3D*)
- Direction (*Vector3D*)

A point along a ray is given by

```
Ray r = ...
Point3D p = r.orig + t * r.dir;
```



Scene class

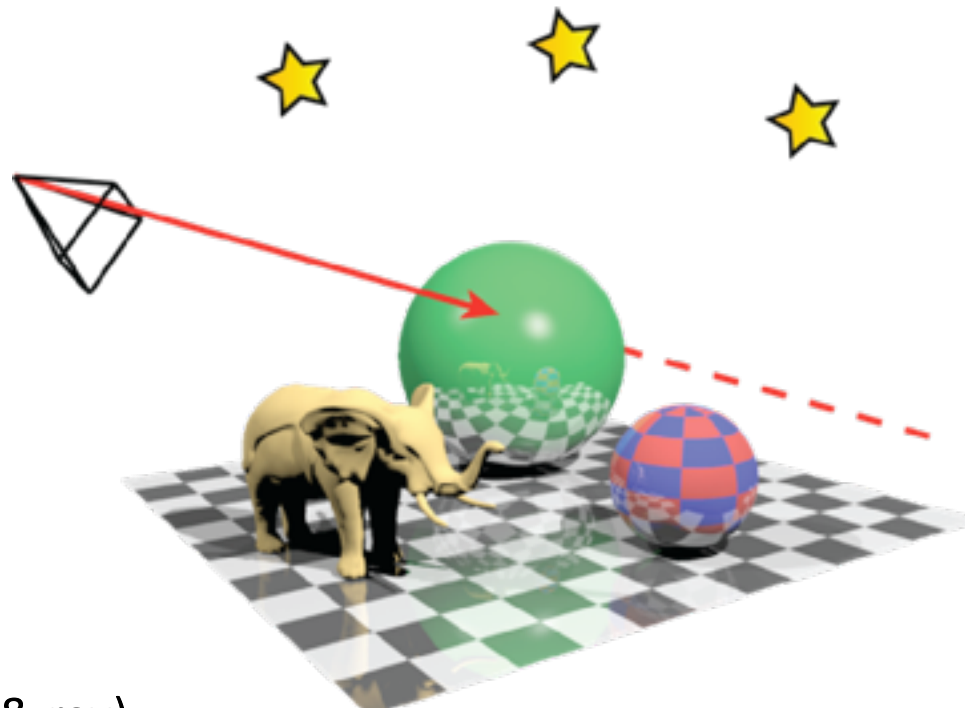
Stores all objects in the 3D world

- Primitives
- Light sources
- Cameras

Scene class

Stores all objects in the 3D world

- Primitives
- Light sources
- Cameras



```
bool Scene::intersect(const Ray& ray);
bool Scene::intersect(const Ray& ray, Intersection& is);
```

Ray-Scene Intersection

Boolean test

```
bool Scene::intersect(const Ray& ray);
```

- Is *anything* hit? (Y/N)
- No intersection information
- Used for shadow rays
  - *(we'll get back to this later)*
- Faster

Ray-Scene Intersection

Closest hit test

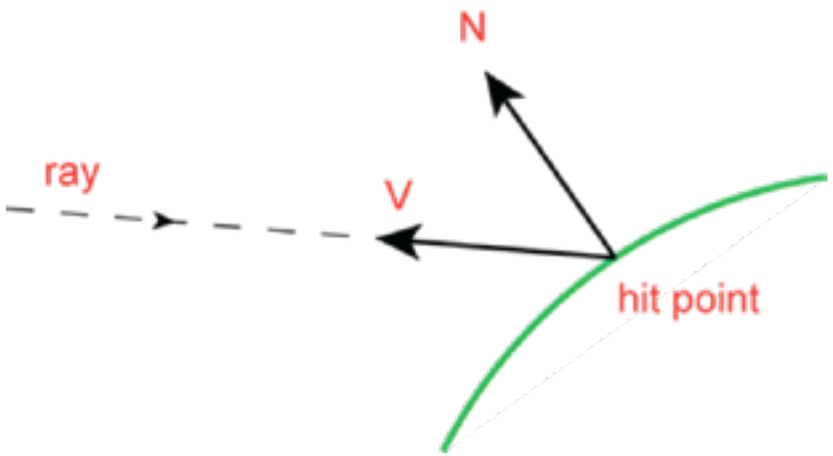
```
bool Scene::intersect(const Ray& ray, Intersection& is);
```

– Returns intersection information
– Used for shading, reflection, refraction, ...
– Slower

Intersection class

Useful information about the intersection

```
class Intersection {
    Point3D      mPosition;      // Position of hit point
    Vector3D     mNormal;        // Surface normal (N)
    Vector3D     mView;          // View direction (V)
    Material     *mMaterial;     // Material of object
    . . .
};
```



Main function

- Build a scene
  - Create materials, objects, lights (scene::add)
- Create image buffer
- Setup camera
- Prepare scene (Scene::prepare)
- Create ray tracer object
- Perform ray tracing (Raytracer::computeImage)
- Save output image

Building a Scene

For this assignment we use the scene

```
buildSpheres(Scene &scene);        // located in main.cpp
```

You are encouraged to play around and building your own scene

- Current scene is very *“programmer art”*

Image and Camera classes

An Image is created by the line

```
Image output(512, 512);             // width x height
```

Camera is setup by

```
Camera cam;
Point3D pos       = Point3D(22.0f, 24.0f, 26.0f);
Point3D target    = Point3D(0.0f, 2.0f, 0.0f);
Vector3D up       = Vector3D(0.0f, 1.0f, 0.0f);
float fov         = 52.0f;
cam->setLookAt(pos, target, up, fov);
scene.add(cam);
```

Rendering a Scene

First we must call Scene::prepare()

- Transform to world space
- Set up <<RayAccelerator>>
  - In this lab this is just a list of intersectables

Create <<Raytracer>> object and start rendering

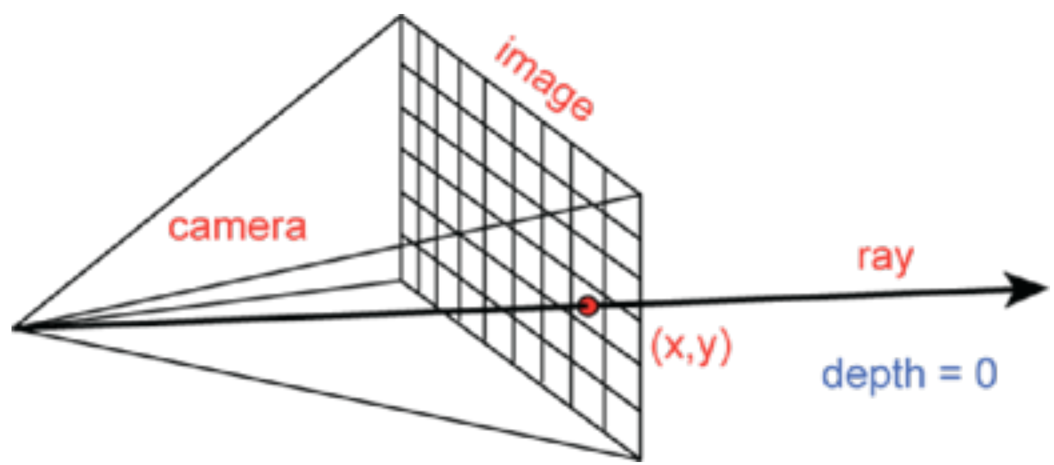
```
WhittedTracer rt(&scene, &output);
rt.computeImage();
output.save("output.exr");
```

Shooting eye rays

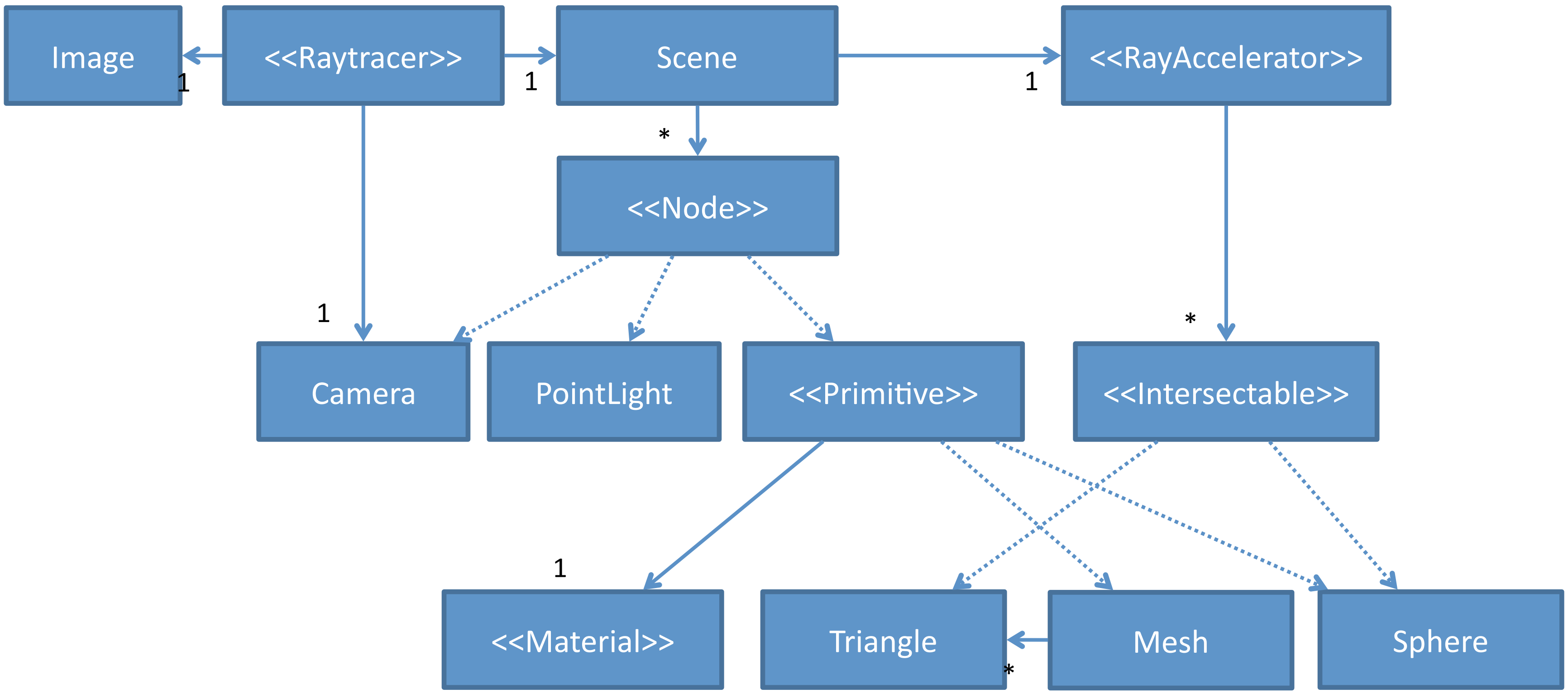
In *WhittedTracer.cpp*

computeImage() calls **tracePixel()** for each pixel (x, y).

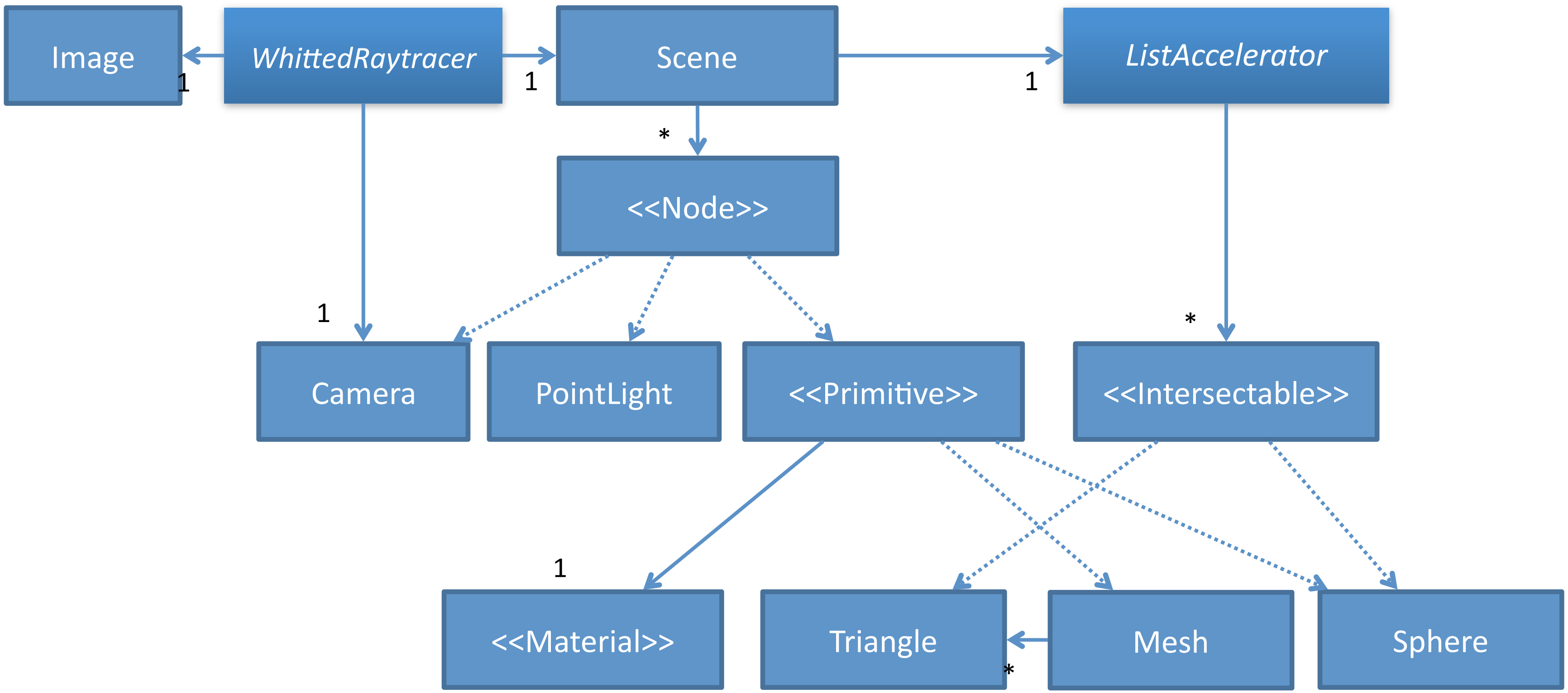
tracePixel() calls **trace()** for rays in the pixel.



prTracer Skeleton Overview



prTracer Skeleton Overview



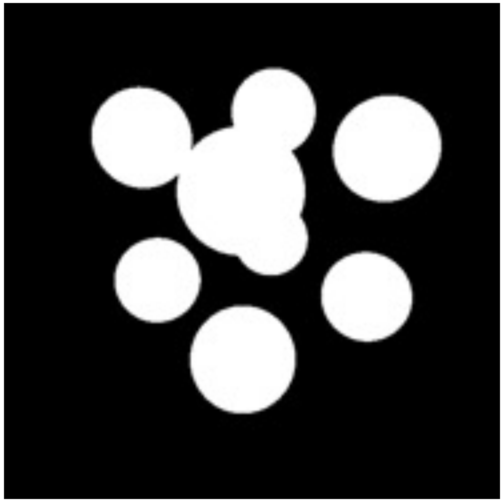
Assignment 1

- Diffuse Reflection
- Whitted ray tracing
  - Shadows
  - Reflections
  - Refractions
- Super sampling
- Blinn-Phong Shading *(Optional)*
- Ray-Triangle Intersection

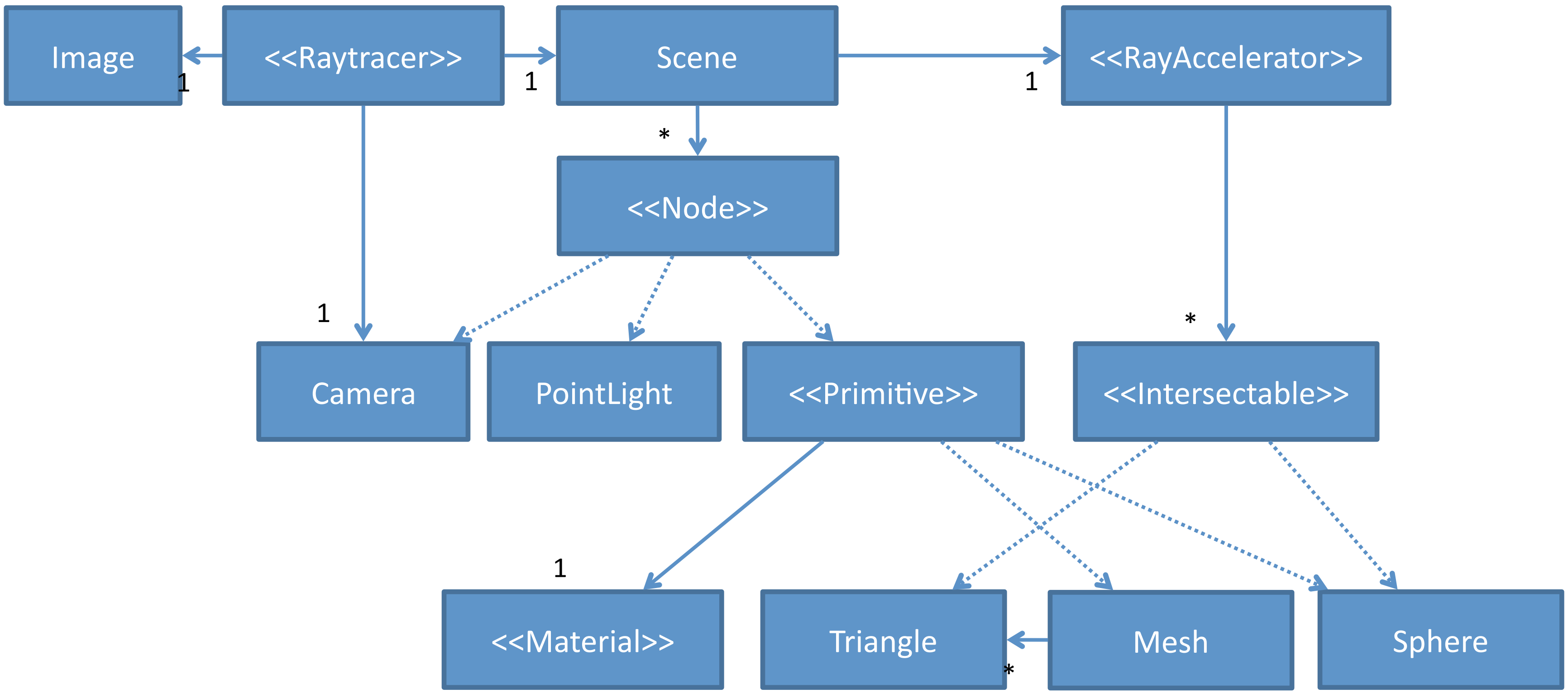
Default Implementation

prTracer embryo

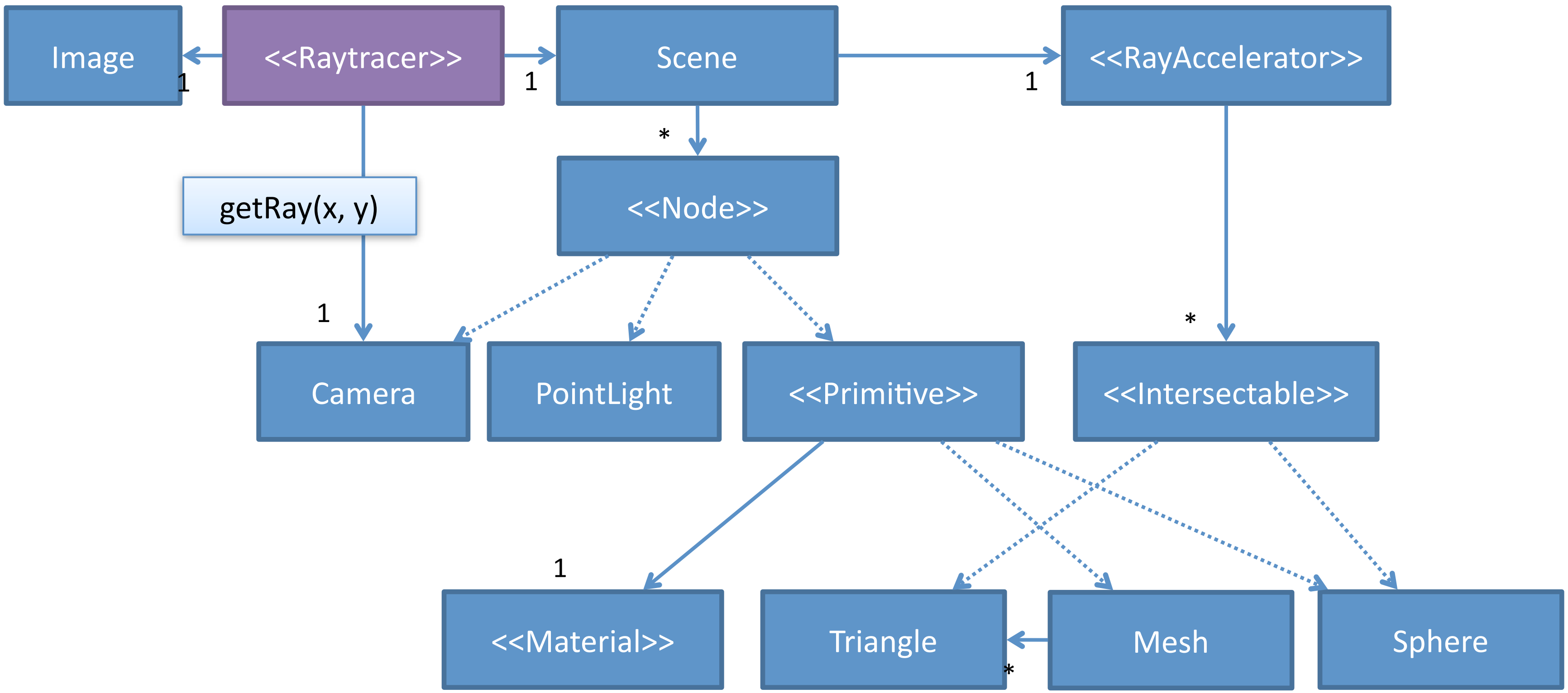
- Colors the pixel white if the ray hits anything, black otherwise
- This is what you should get when you compile and run your code for the first time



Program flow

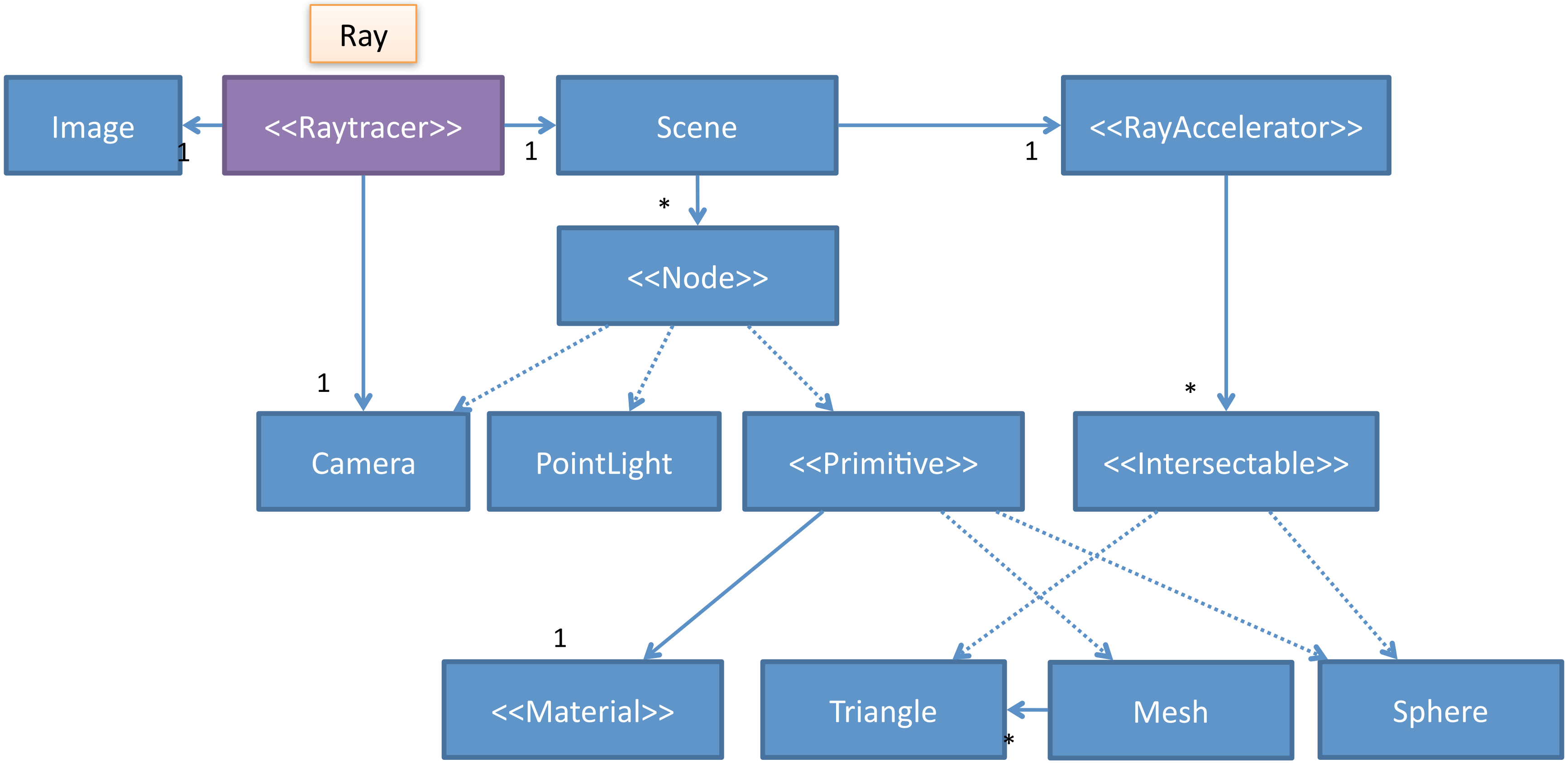


Program flow

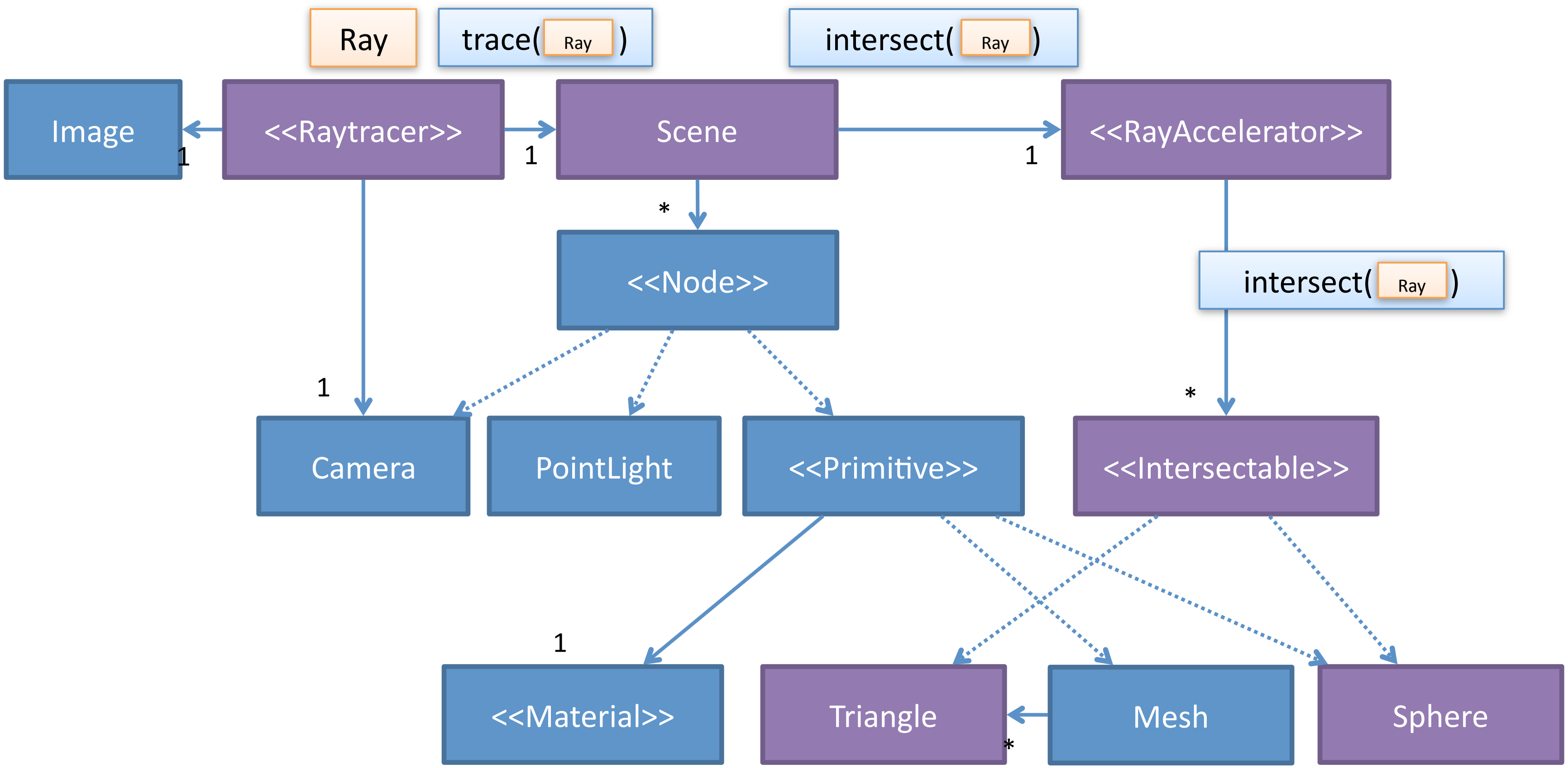


Fetch view ray from camera

Program flow

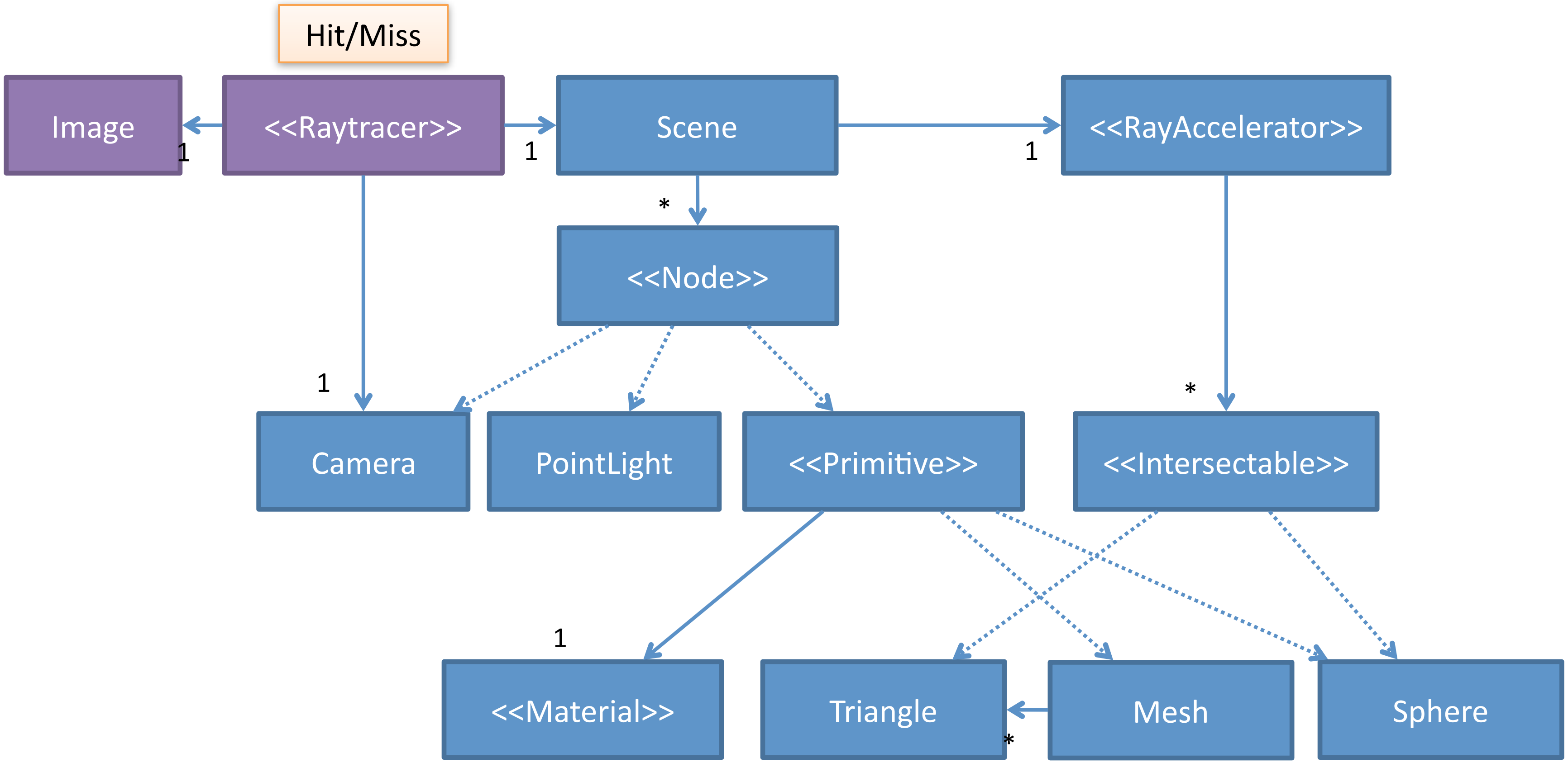


Program flow



Intersect with the scene’s intersectables

Program flow



Hit = write white pixel   Miss = write black pixel

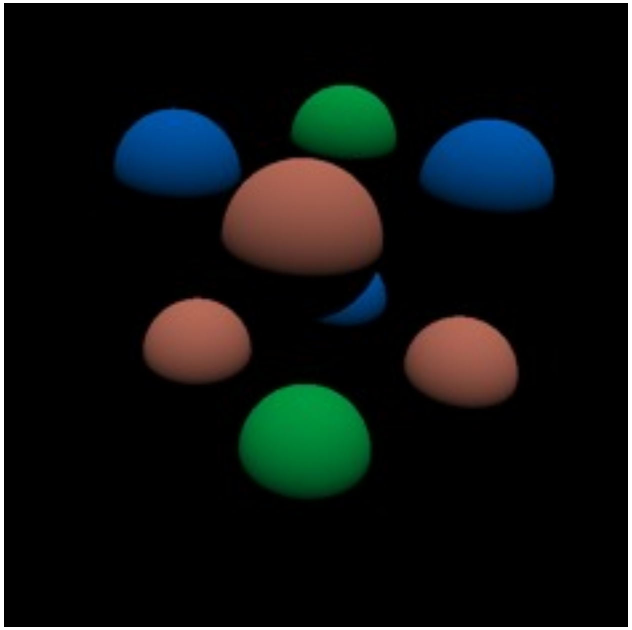
Assignment 1

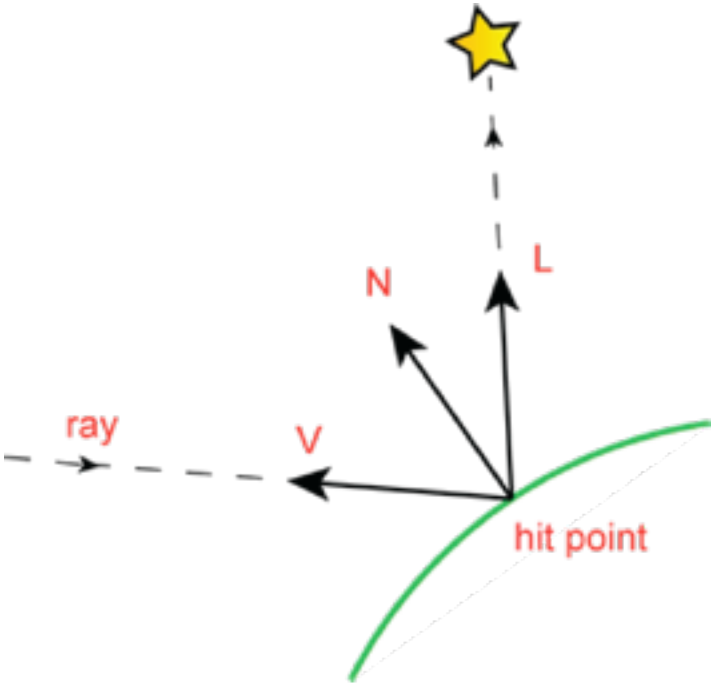
- Diffuse Reflection
- Whitted ray tracing
  - Shadows
  - Reflections
  - Refractions
- Super sampling
- Blinn-Phong Shading *(Optional)*
- Ray-Triangle Intersection

Diffuse Reflection

For this assignment you'll need to modify

```
Color WhittedTracer::trace(const Ray& ray, int depth)
```





Diffuse Reflection

$$L_{out}(x \rightarrow \Theta) = L_{in}(x \leftarrow \Psi) f_r(x, \Psi \leftrightarrow \Theta) cos(N_x, \Psi)$$

- Incoming radiance — light->getRadiance();
- BRDF — is.mMaterial.evalBRDF(is, Ψ);
- Incident angle — max(Ψ.dot(is.mNormal), 0.0f);

Diffuse Reflection

The scene has a number of light sources

```
int n = mScene->getNumberOfLights();
PointLight *light = mScene->getLight(i);
```

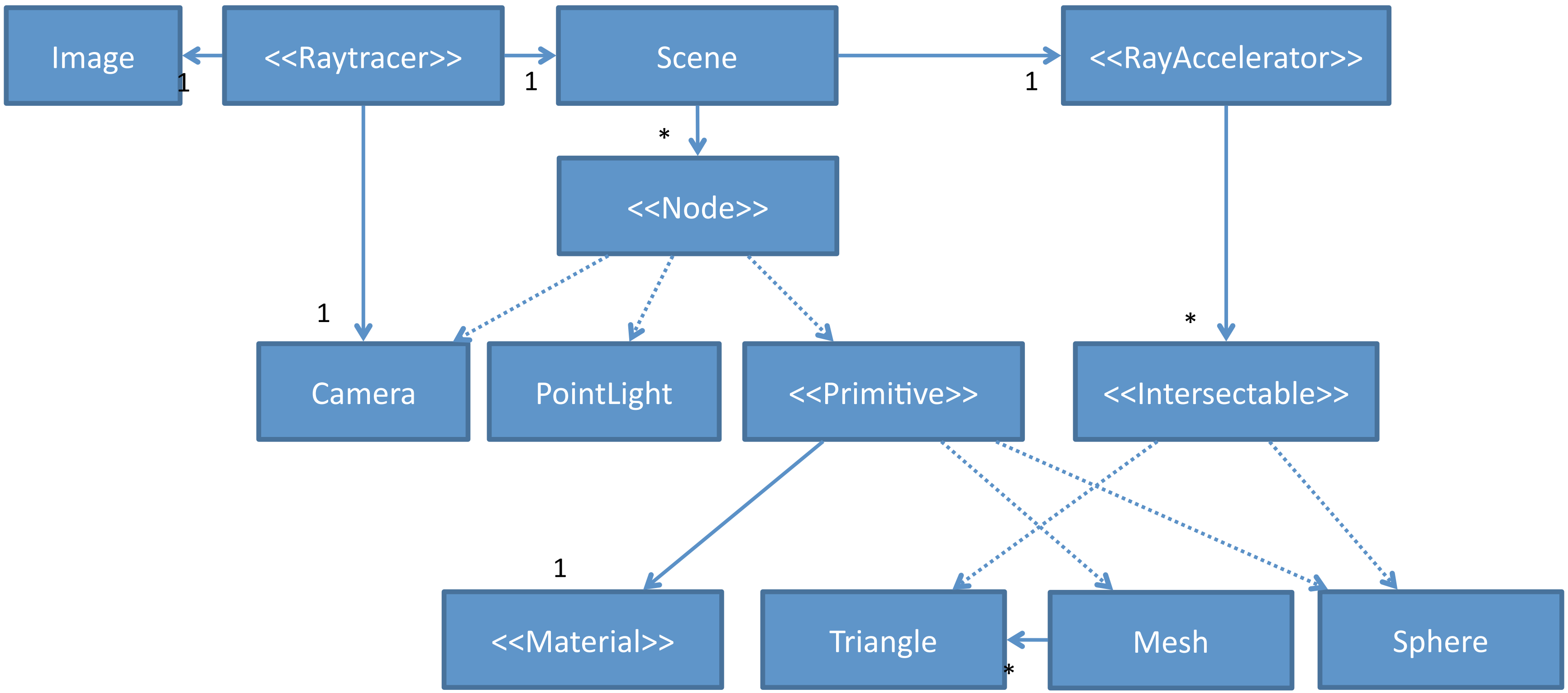
Calculating the light vector (Ψ)

```
Vector3D lightVec = light->getWorldPosition() – is.mPosition;
lightVec.normalize();
```

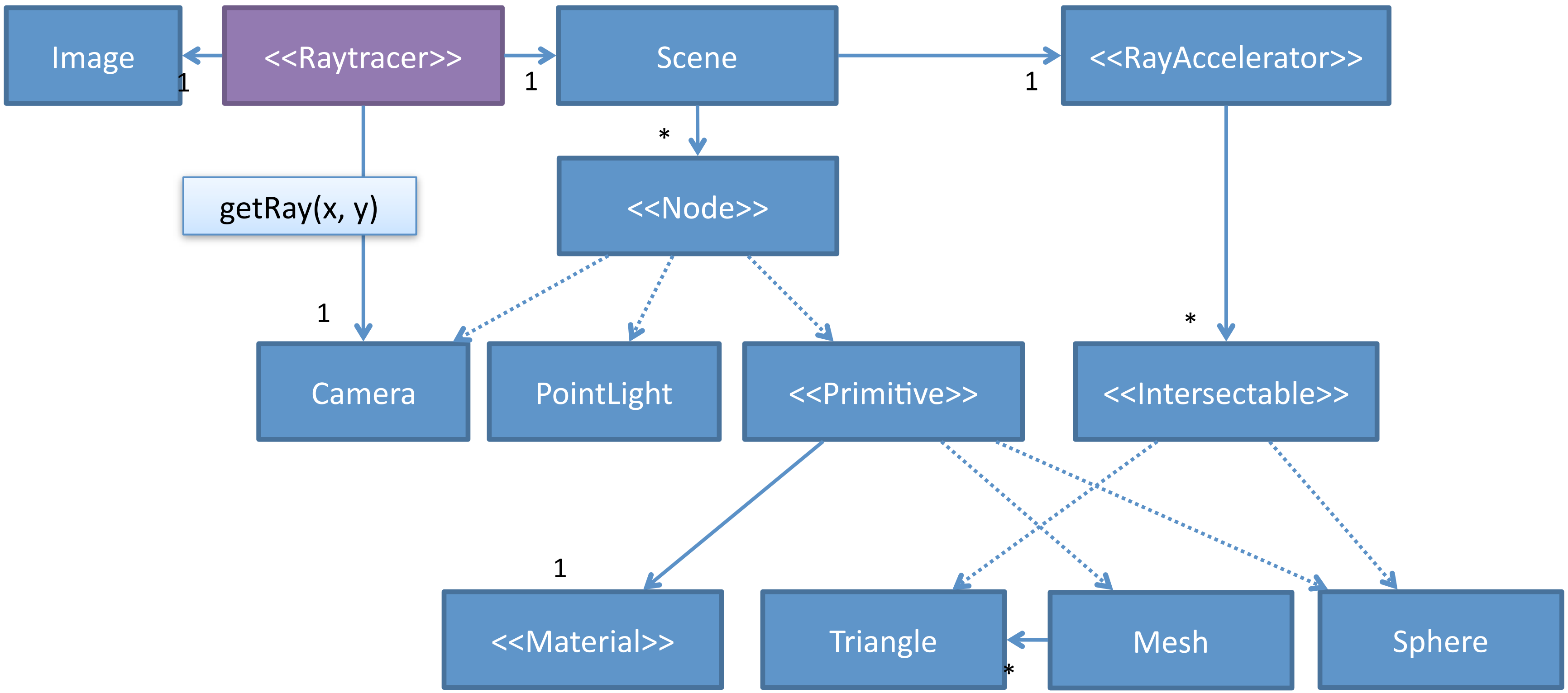
Sum the contribution to get the right result

$$L_{out}(x \to \Theta) = \sum_{i}^{lights} L_i(x \leftarrow \Psi_i) f_r(x, \Psi_i \leftrightarrow \Theta) cos(N_x, \Psi_i)$$

Program flow

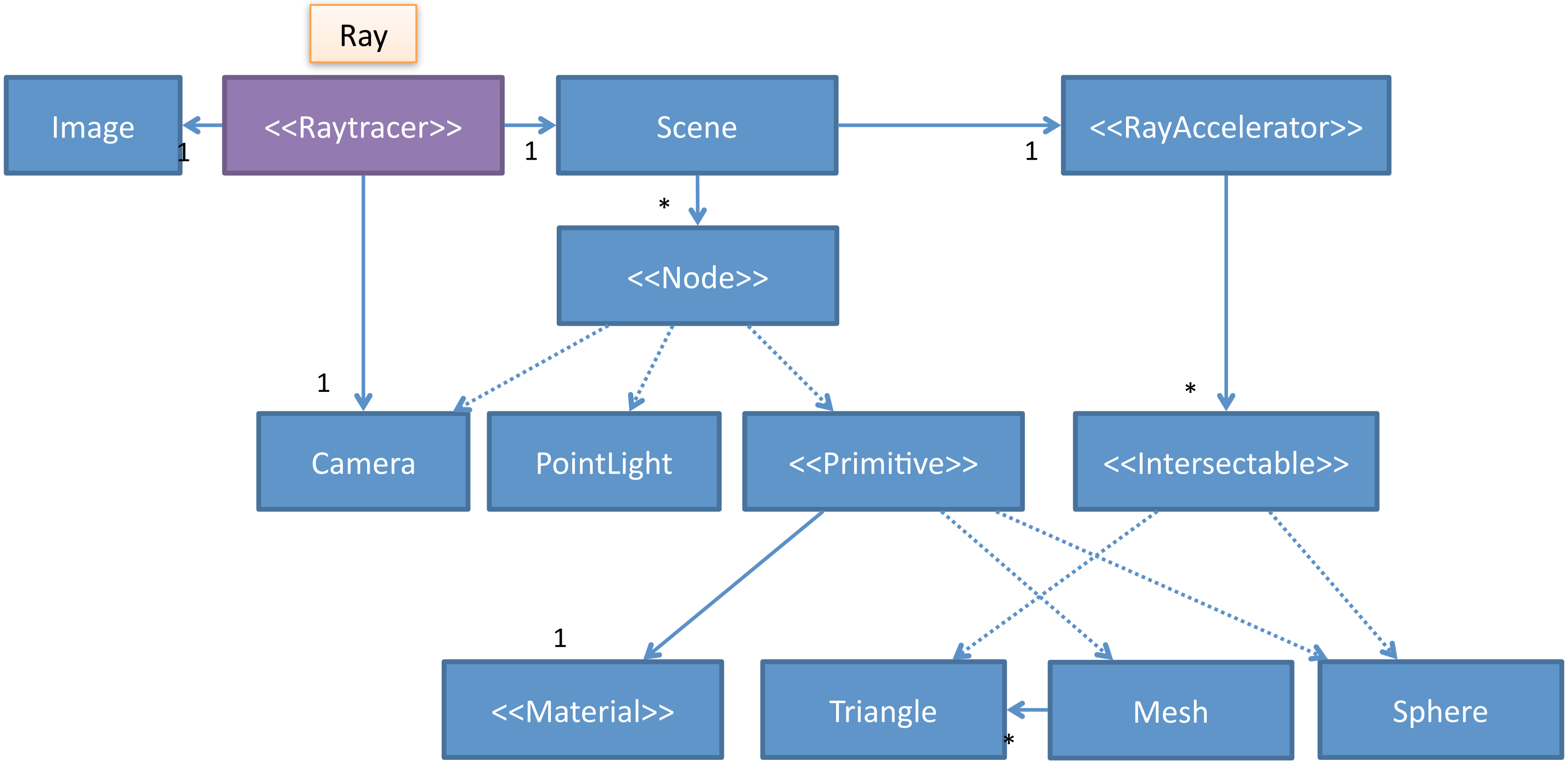


Program flow

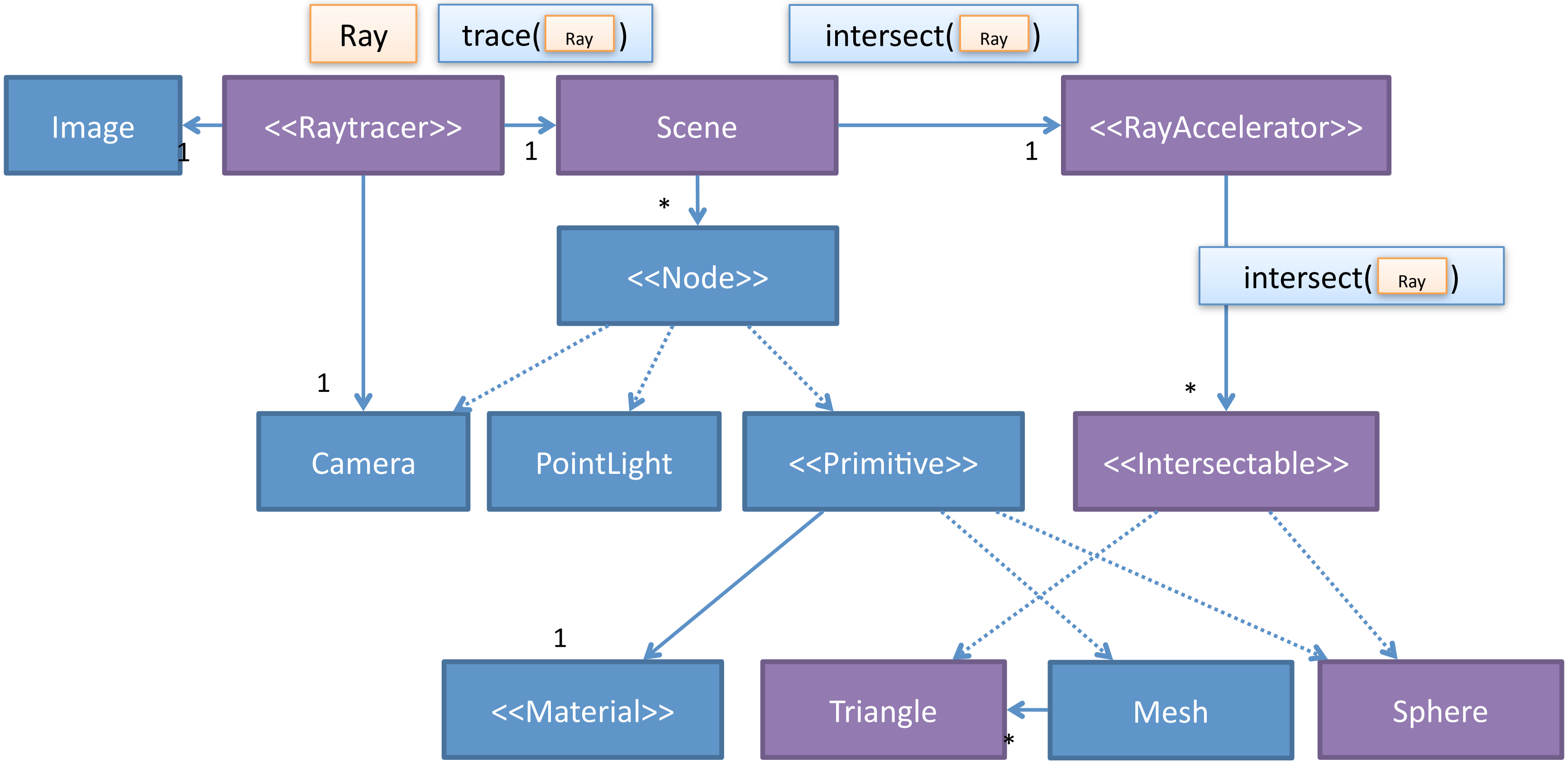


Fetch view ray from camera

Program flow

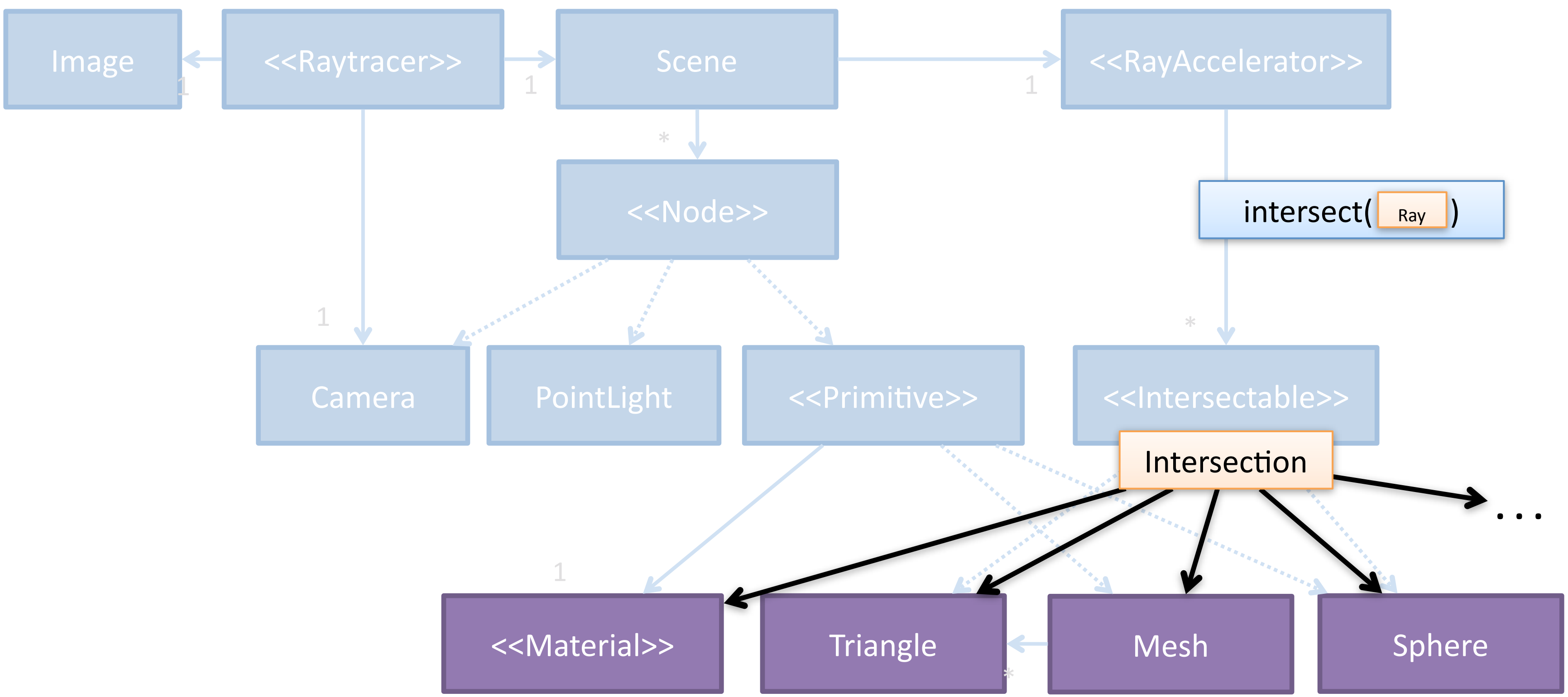


Program flow



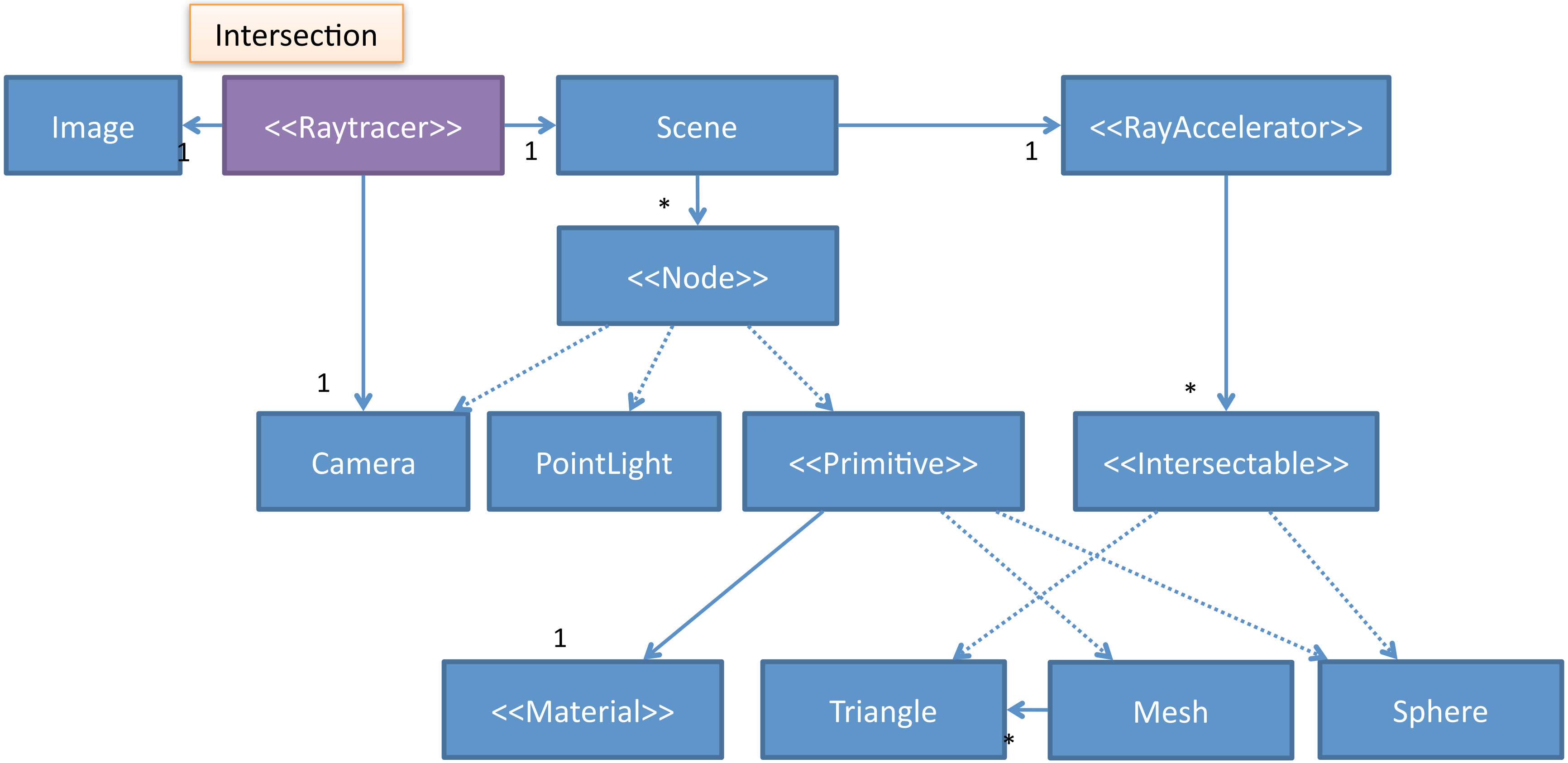
Intersect with the scene’s intersectables

Program flow



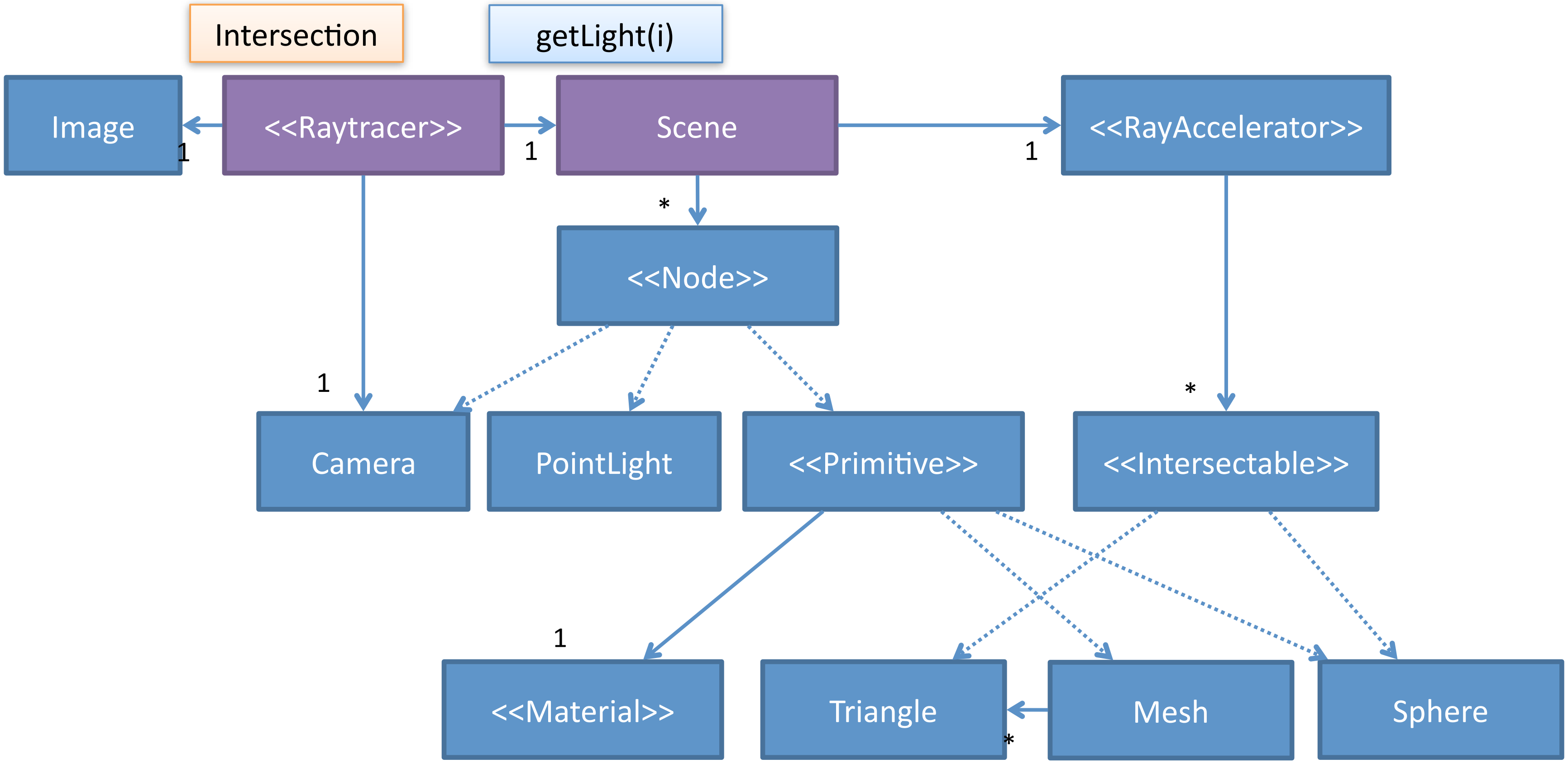
Gather hit point information

Program flow



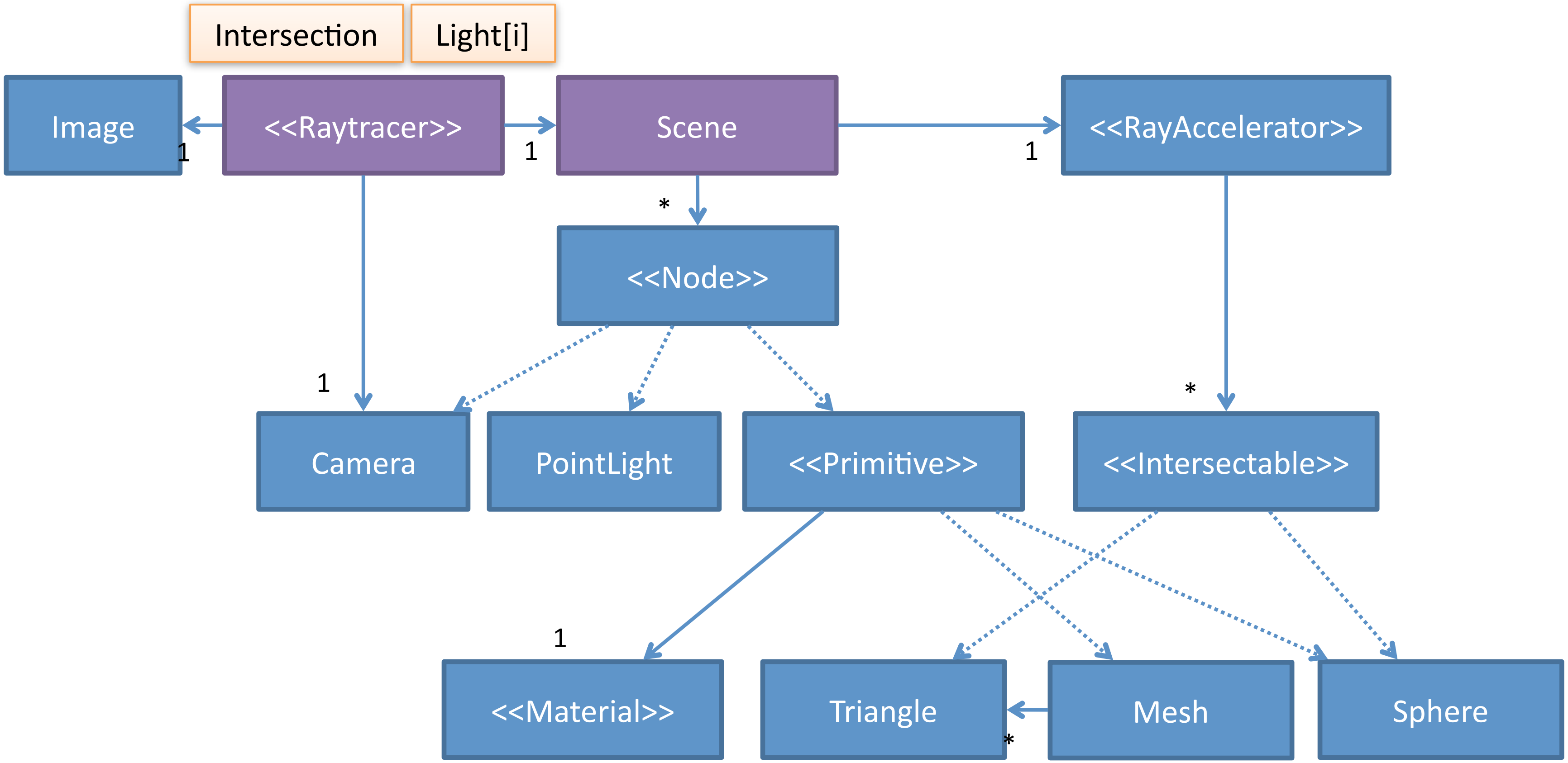
Intersect with the scene’s intersectables

Program flow

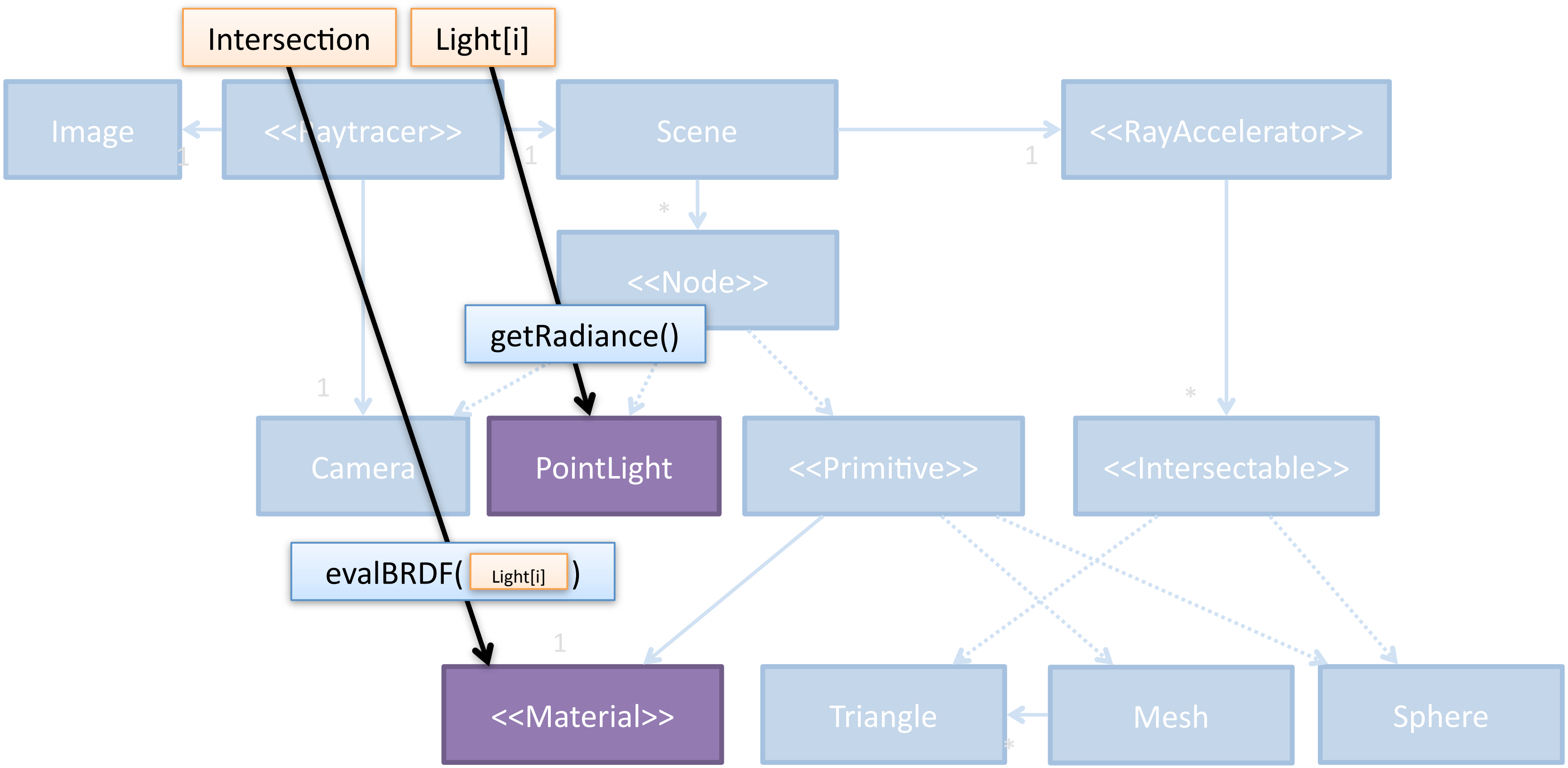


Get light *i* from the scene

Program flow

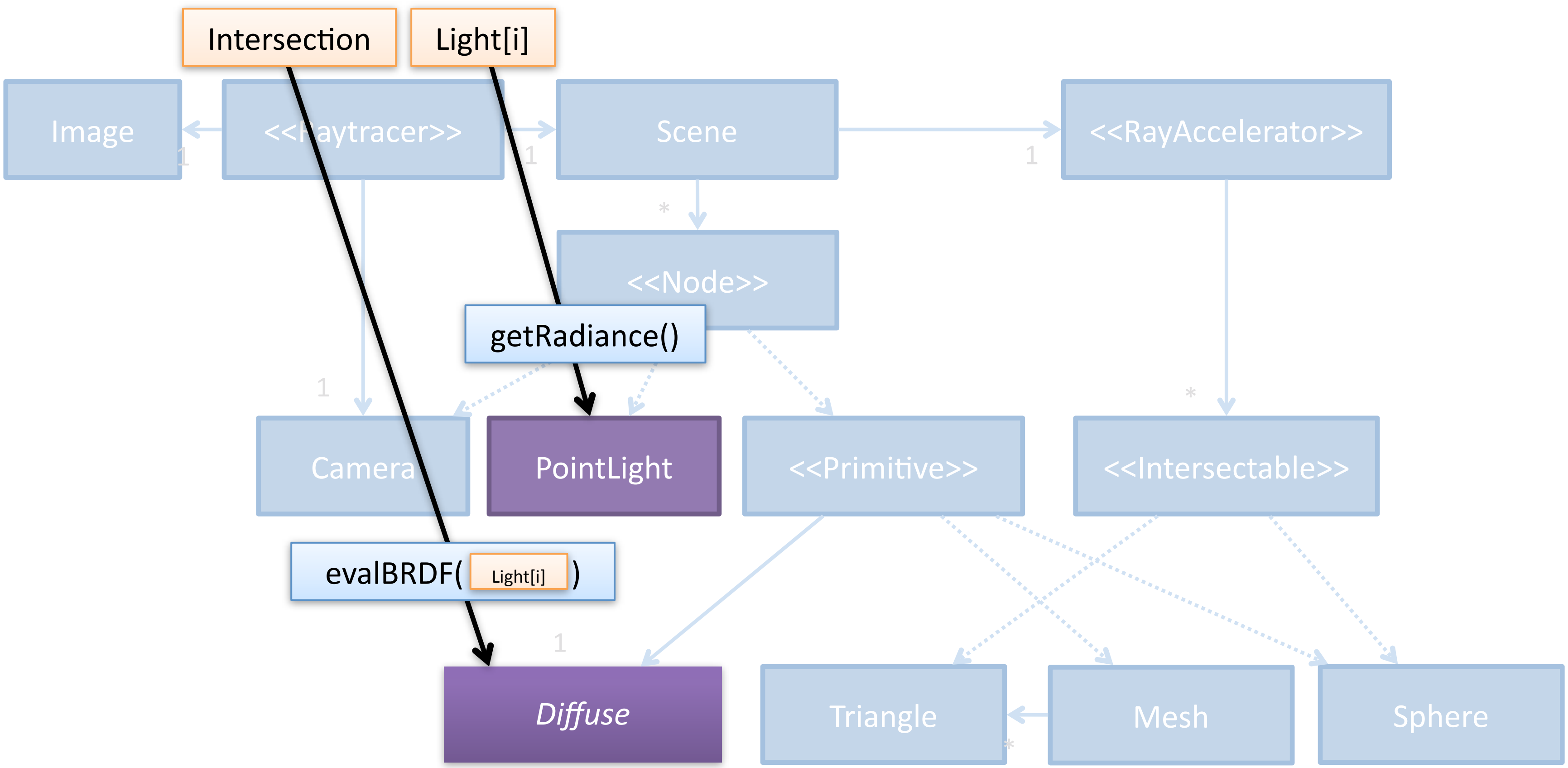


Program flow



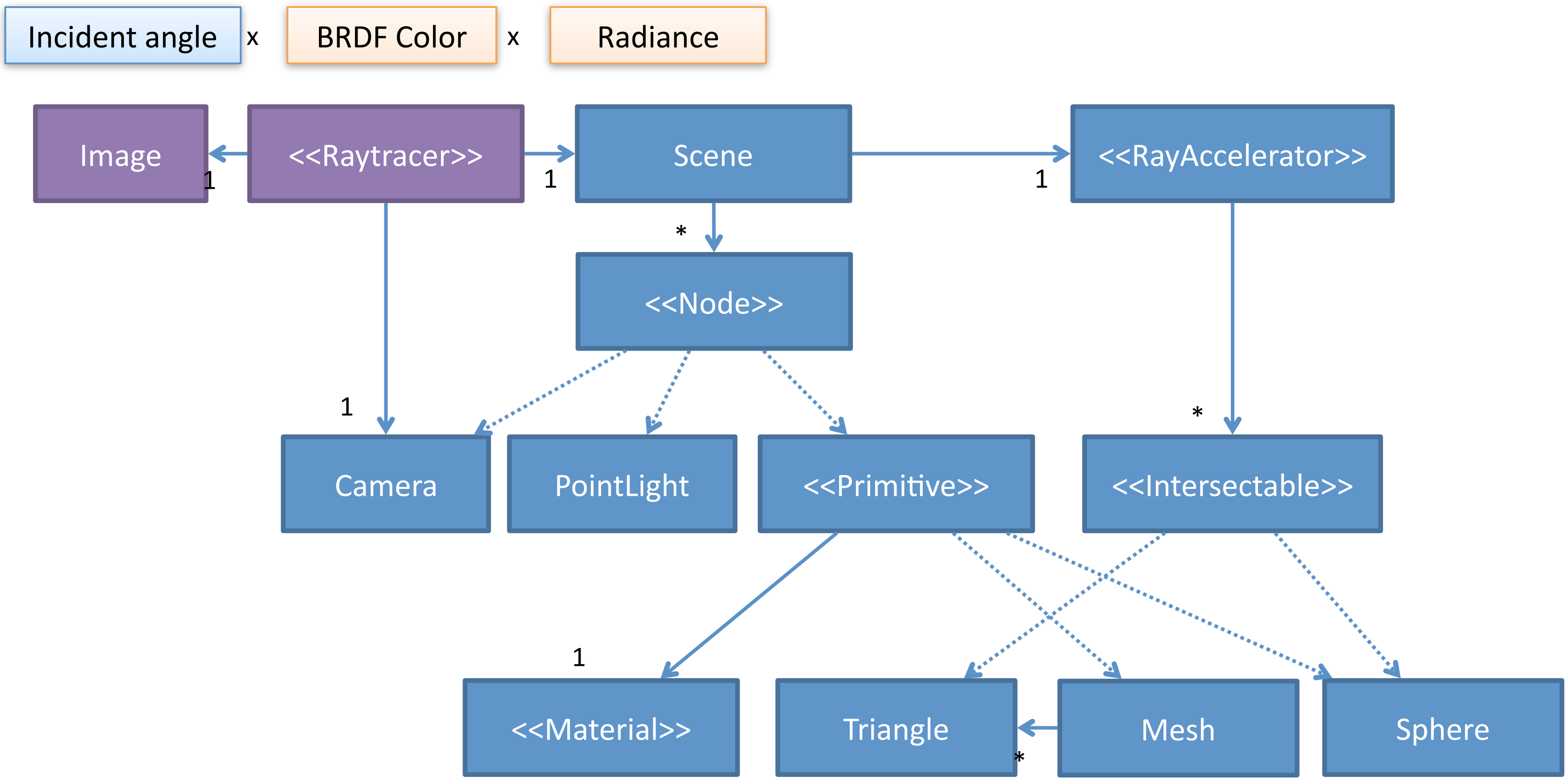
For each light, get light radiance and evaluate material BRDF

Program flow



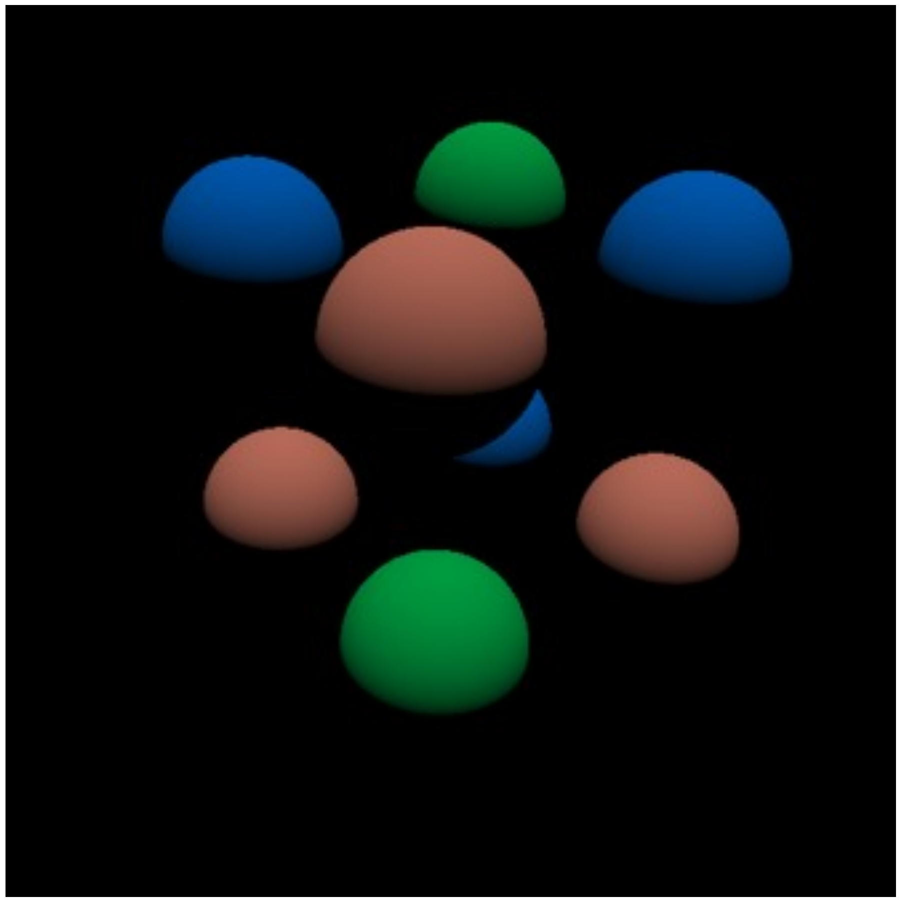
In our case the Material is Diffuse

Program flow



For each light, contribute to the final pixel color

Diffuse Reflection

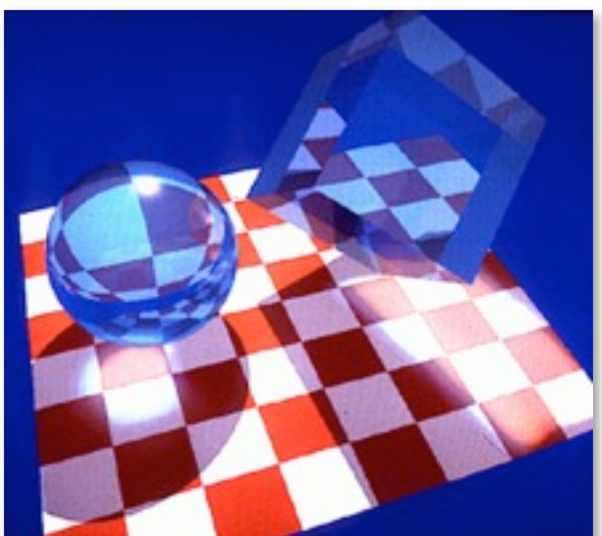


Assignment 1

- Diffuse Reflection
- Whitted ray tracing
  - Shadows
  - Reflections
  - Refractions
- Super sampling
- Blinn-Phong Shading *(Optional)*
- Ray-Triangle Intersection

Whitted Ray Tracing

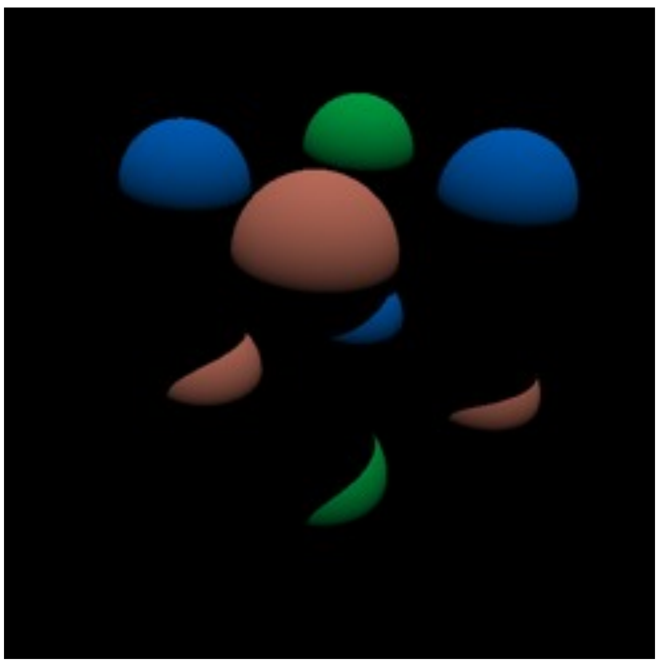
- Introduced by Turner Whitted in 1980
- Added ***shadows*** and recursive ***reflection*** and ***refraction***
- **Not** physically correct!

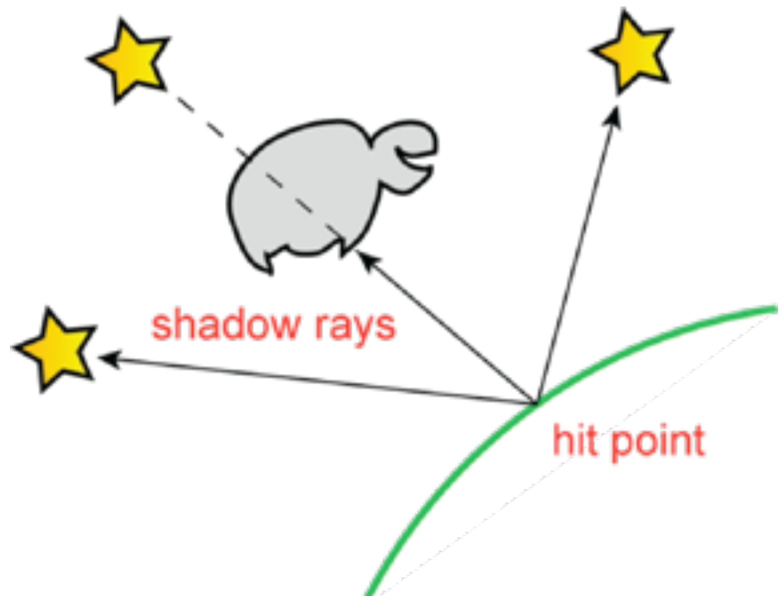


Shadows

For this assignment you will need to modify

```
Color WhittedTracer::trace(const Ray& ray, int depth)
```





Shadows

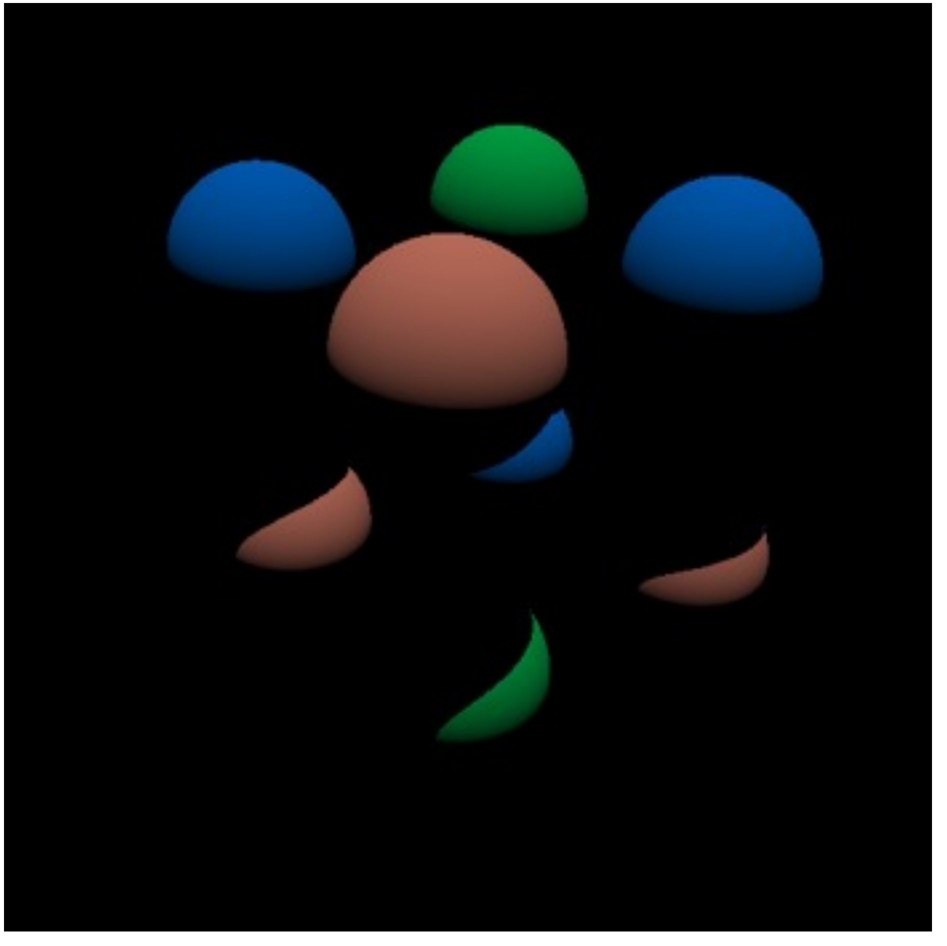
Helper function:

```
PointLight *light = mScene->getLight(i);
Ray r = is.getShadowRay(light);
```

Returns a shadow ray:

- r.orig = is.mPosition + *epsilon* * is.mNormal
- r.dir = direction of light vector
- $r.t_{max}$ = distance from hit point to light source

Shadows

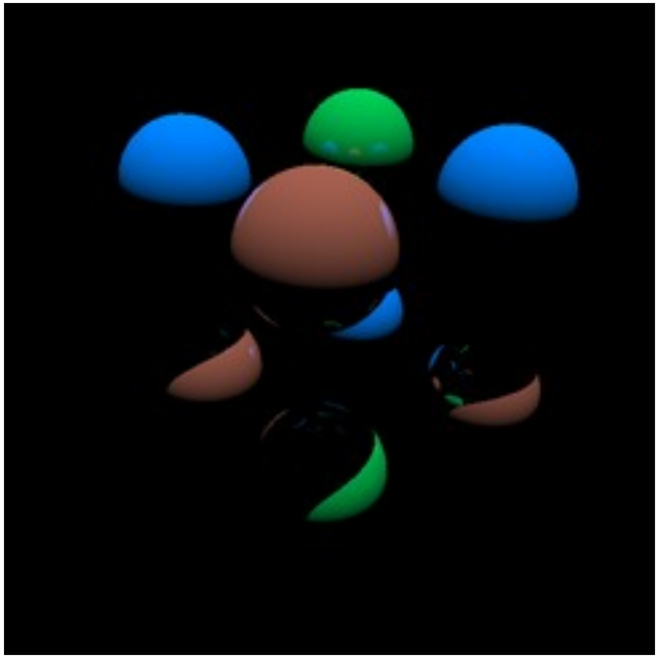


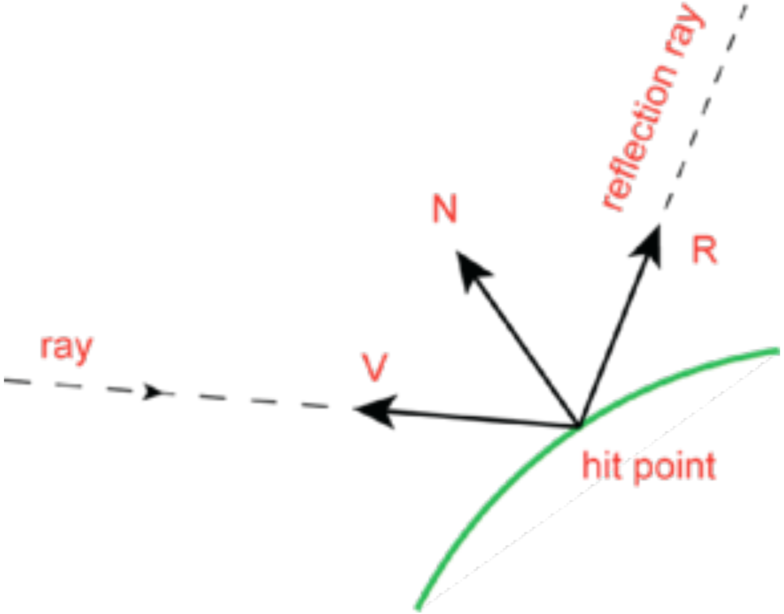
Reflection

For this assignment you will need to modify

Ray Intersection::getReflectedRay() const

Color WhittedTracer::trace(const Ray& ray, int depth)

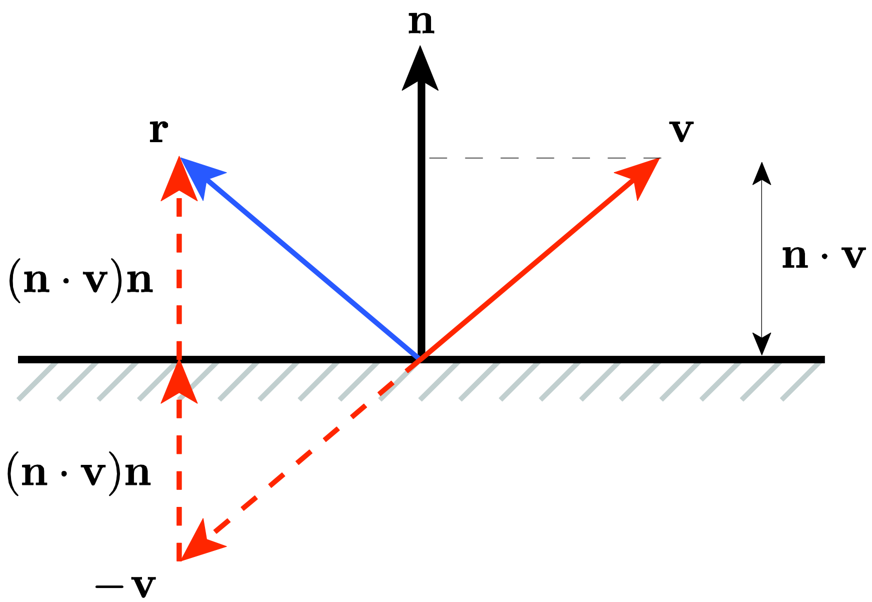




Reflection

First, add proper reflection code to

Ray Intersection::getReflectedRay() const

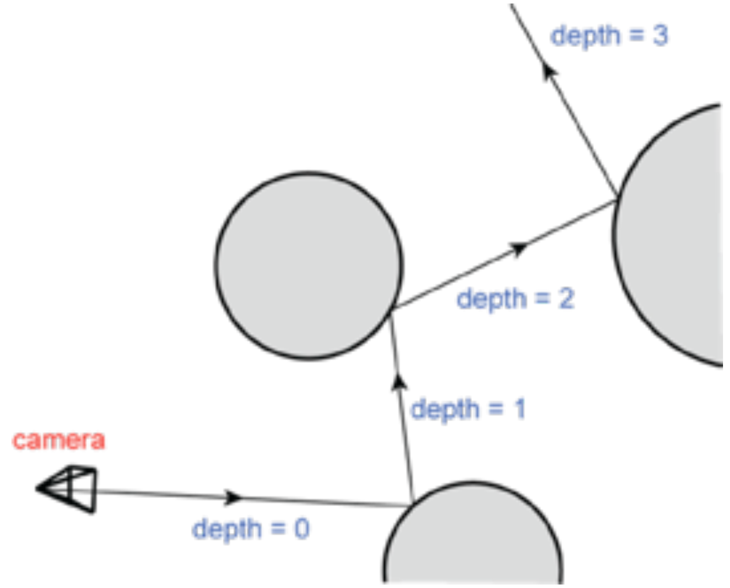


$$\boxed{\mathbf{r} = 2(\mathbf{n} \cdot \mathbf{v})\mathbf{n} - \mathbf{v}}$$

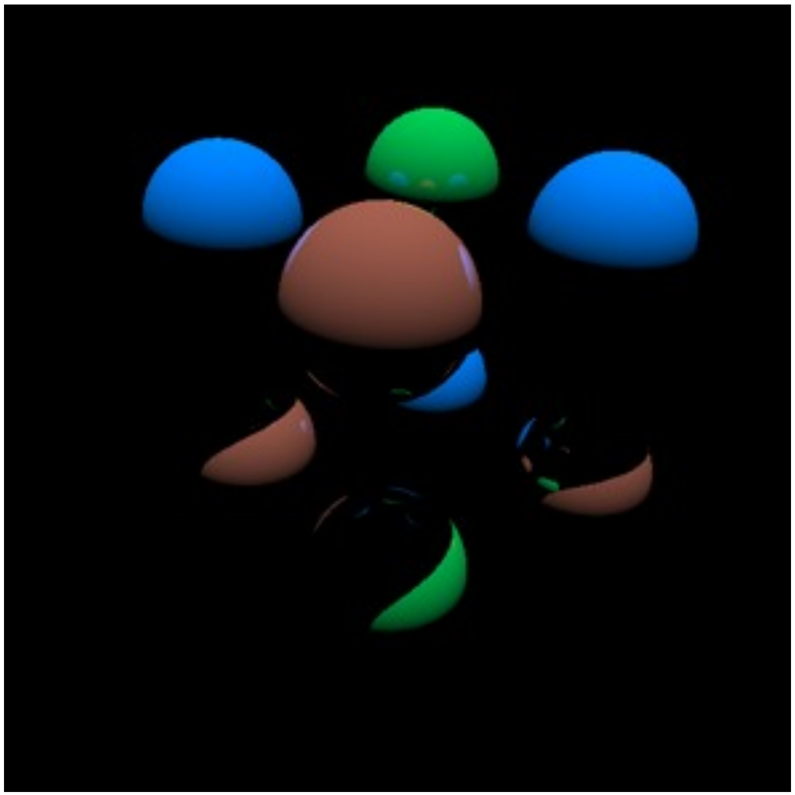
Reflection

Next, modify `Color WhittedTracer::trace(const Ray& ray, int depth)` to follow reflection rays *recursively*.

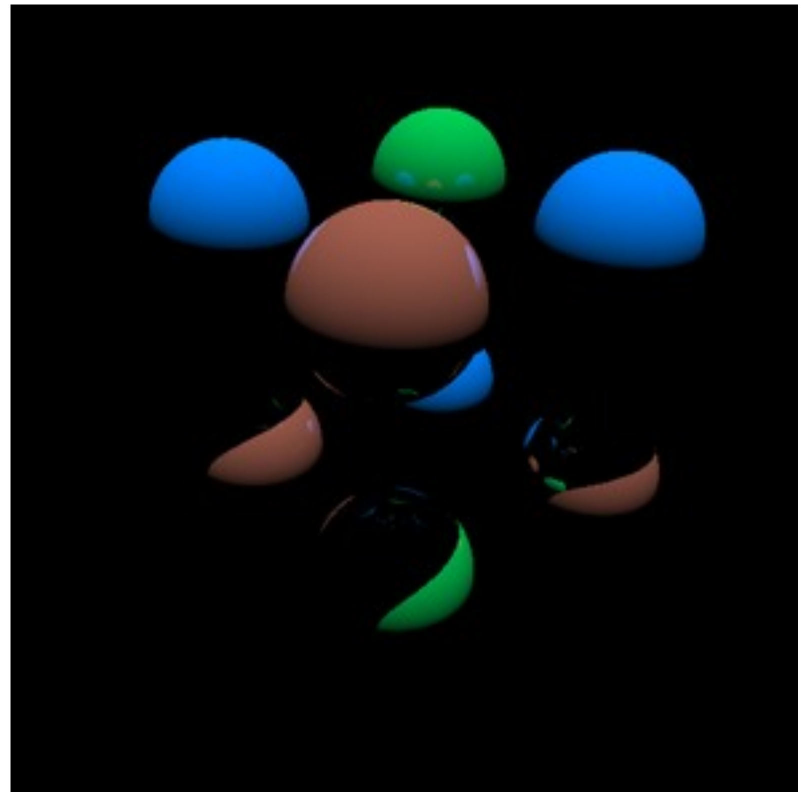
- Use fixed recursion depth (2 or 3 to start with)



Reflection



1 Bounce

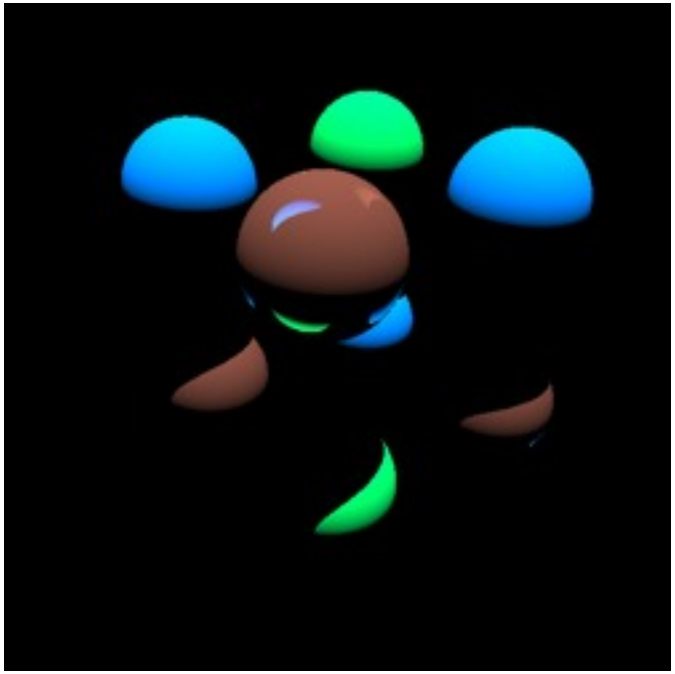


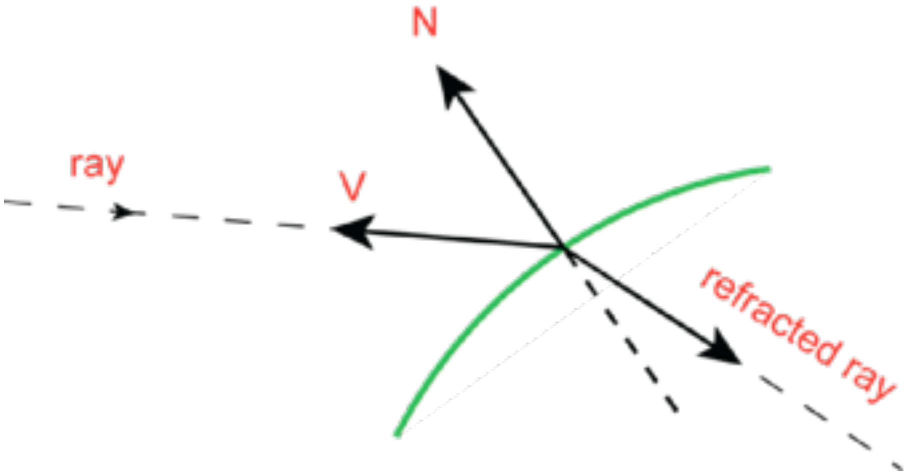
2 Bounces

Refraction

For this assignment you will need to modify

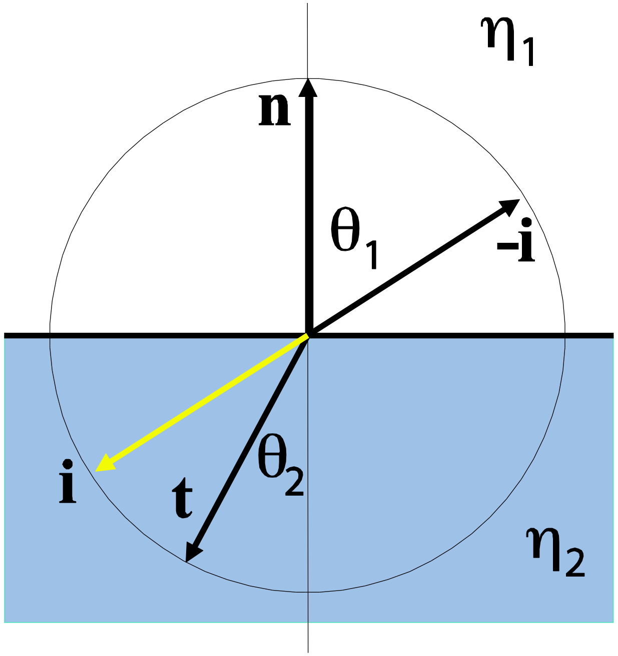
```
Ray Intersection::getRefractedRay() const
Color WhittedTracer::trace(const Ray& ray, int depth)
```



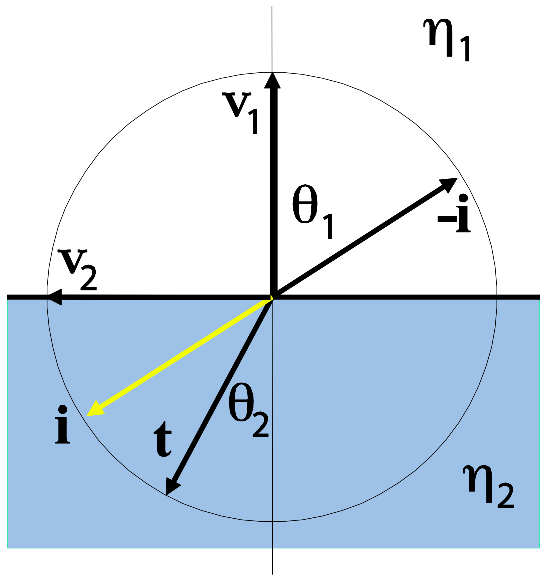


Refraction

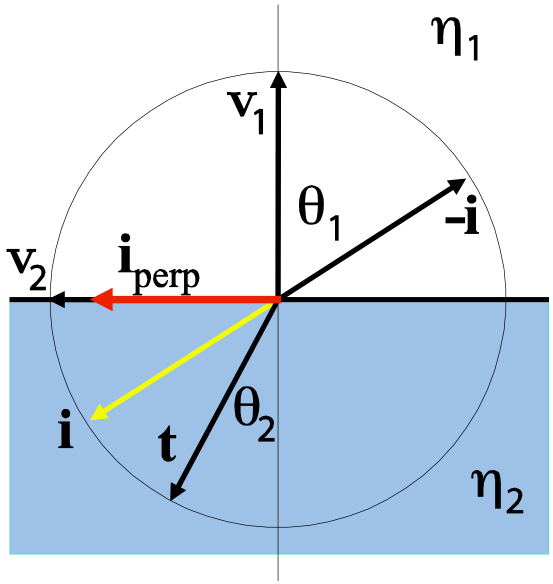
Modify `Ray Intersection::getRefractedRay() const` to properly calculate the refraction vector *t*



Refraction



Find vectors in the refraction plane to express *t* with:

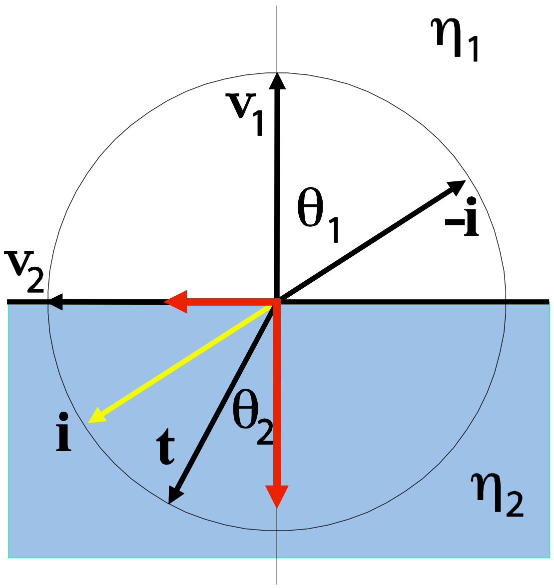


$$\mathbf{v_1} = \mathbf{n}$$

$$\mathbf{i_{perp}} = \mathbf{i} + \cos\Theta_1 \mathbf{n}$$

$$\mathbf{v_2} = \frac{\mathbf{i_{perp}}}{||\mathbf{i_{perp}}||}$$

Refraction



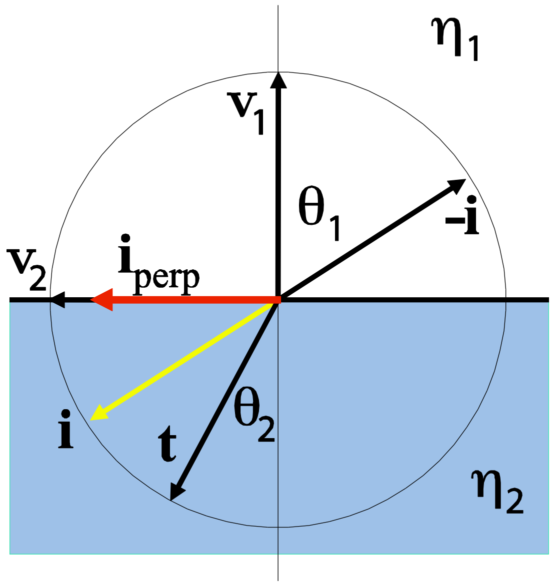
Snell's law:

$$\frac{\sin\Theta_2}{\sin\Theta_1} = \frac{\eta_1}{\eta_2} = \eta$$

We now know enough to solve *t* with

$$\boxed{\mathbf{t} = -\sin\Theta_2 \mathbf{v_1} - \cos\Theta_2 \mathbf{v_2}}$$

Refraction



$$\boxed{\mathbf{t} = \text{-} \sin\Theta_2 \mathbf{v_1} - \cos\Theta_2 \mathbf{v_2}}$$

Simplification!

$$\mathbf{v_2} = \frac{\mathbf{i_{perp}}}{||\mathbf{i_{perp}}||}$$

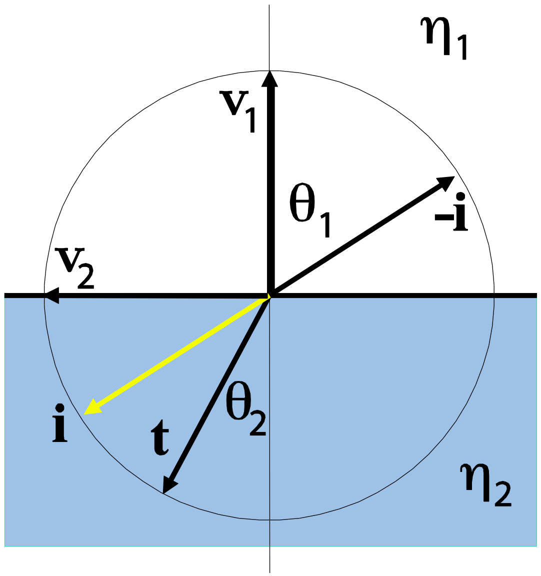
$$\mathbf{i_{perp}} = \mathbf{i} + \cos\Theta_1 \mathbf{n}$$

$$||\mathbf{i_{perp}}|| = \sin\Theta_1$$

=>

$$\boxed{\mathbf{t} = \sin\Theta_2 \frac{\mathbf{i} - \cos\Theta_1 \mathbf{n}}{\sin\Theta_1} - \cos\Theta_2 \mathbf{n}}$$

Refraction



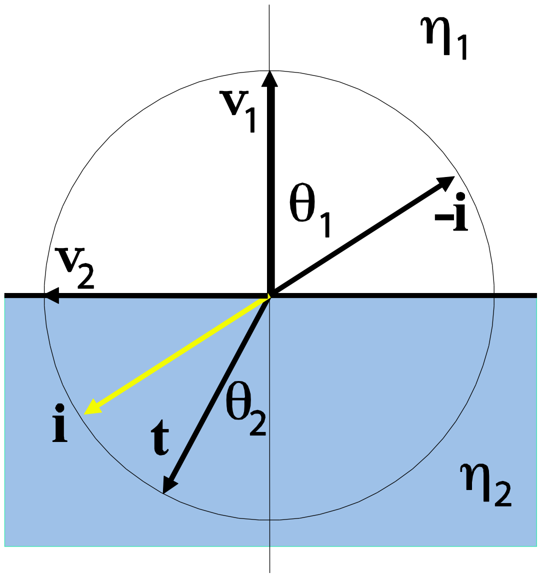
$$\boxed{\mathbf{t} = \sin\Theta_2 \frac{\mathbf{i} - \cos\Theta_1 \mathbf{n}}{\sin\Theta_1} - \cos\Theta_2 \mathbf{n}}$$

$$\frac{\sin\Theta_2}{\sin\Theta_1} = \frac{\eta_1}{\eta_2} = \eta$$

=>

$$\boxed{\mathbf{t} = \eta\mathbf{i} + (\eta\cos\Theta_1 - \cos\Theta_2)\mathbf{n}}$$

Refraction

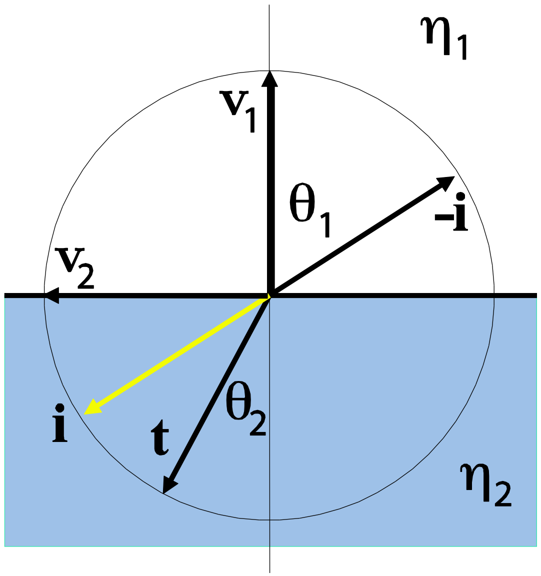


$$\boxed{\mathbf{t} = \eta\mathbf{i} + (\eta\cos\Theta_1 - \cos\Theta_2)\mathbf{n}}$$

$$\cos\Theta_2 = \sqrt{1 - \eta^2(1 - \cos\Theta_1)^2}$$

$$\cos\Theta_1 = -\,\mathbf{i}\cdot\mathbf{n}$$

Refraction



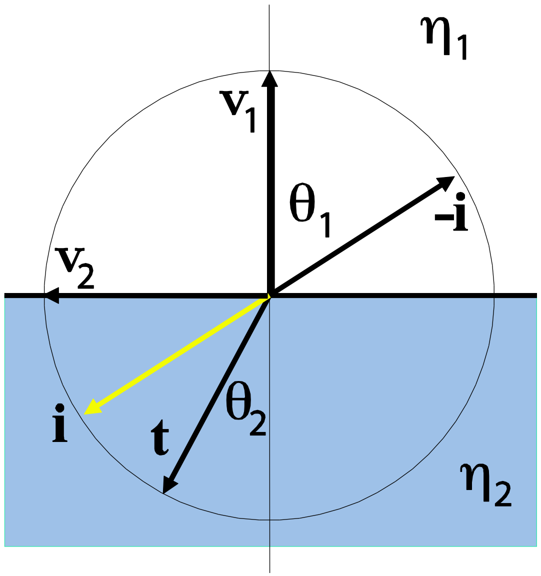
Finally…

$$\mathbf{r} = -\ \mathbf{i} \cdot \mathbf{n}$$

$$\mathbf{c} = 1 - \eta^2 (1 - r^2)$$

$$\mathbf{t} = \eta \mathbf{i} + (\eta r - \sqrt{c}) \mathbf{n}$$

Refraction



Finally…

$$\mathbf{r} = -\ \mathbf{i} \cdot \mathbf{n}$$

$$\mathbf{c} = 1 - \eta^2(1 - r^2)$$

$$\mathbf{t} = \eta\mathbf{i} + (\eta r - \sqrt{c})\mathbf{n}$$

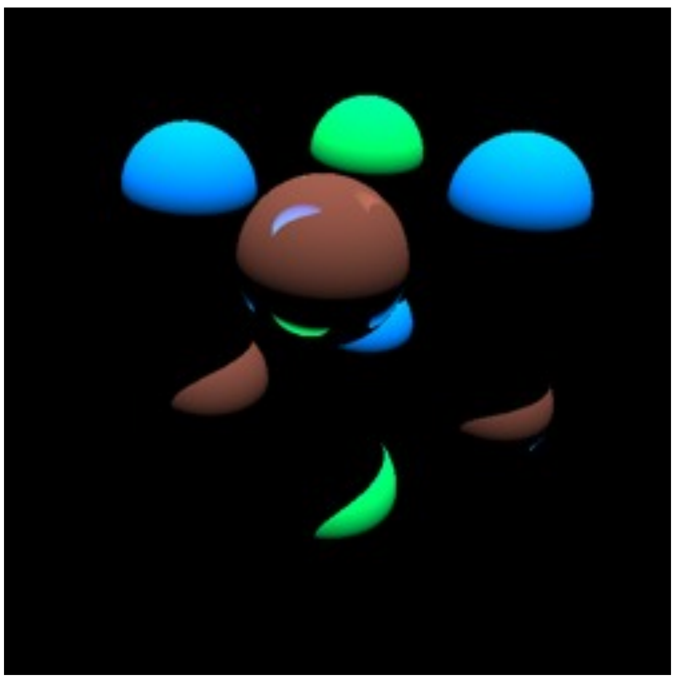
We can't solve for $\mathbf{c} < 0$

– Total internal reflection!

- Return reflection ray instead

Refraction

By now your scene probably looks better than mine…



To get this result I changed to the following in the buildSpheres-function:

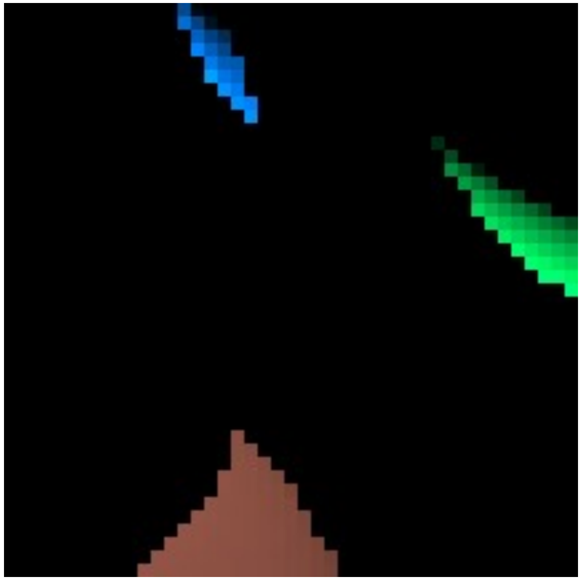
```
material[0]->setReflectivity(0.0f);
material[1]->setReflectivity(0.0f);
material[2]->setReflectivity(0.0f);

material[0]->setTransparency(0.0f);
material[1]->setTransparency(0.9f);
material[2]->setTransparency(0.0f);

material[0]->setIndexOfRefraction(1.0f);
material[1]->setIndexOfRefraction(1.5f);
material[2]->setIndexOfRefraction(1.0f);
```

Assignment 1

- Diffuse Reflection
- Whitted ray tracing
  - Shadows
  - Reflections
  - Refractions
- Super sampling
- Blinn-Phong Shading *(Optional)*
- Ray-Triangle Intersection

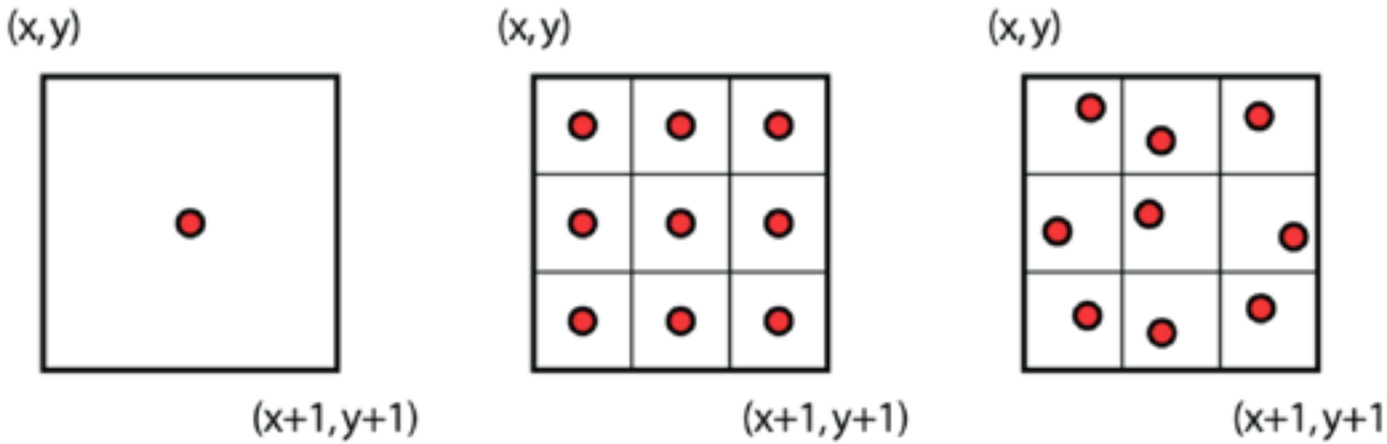
Super Sampling

The images look very jagged so far when we zoom in…



Super Sampling

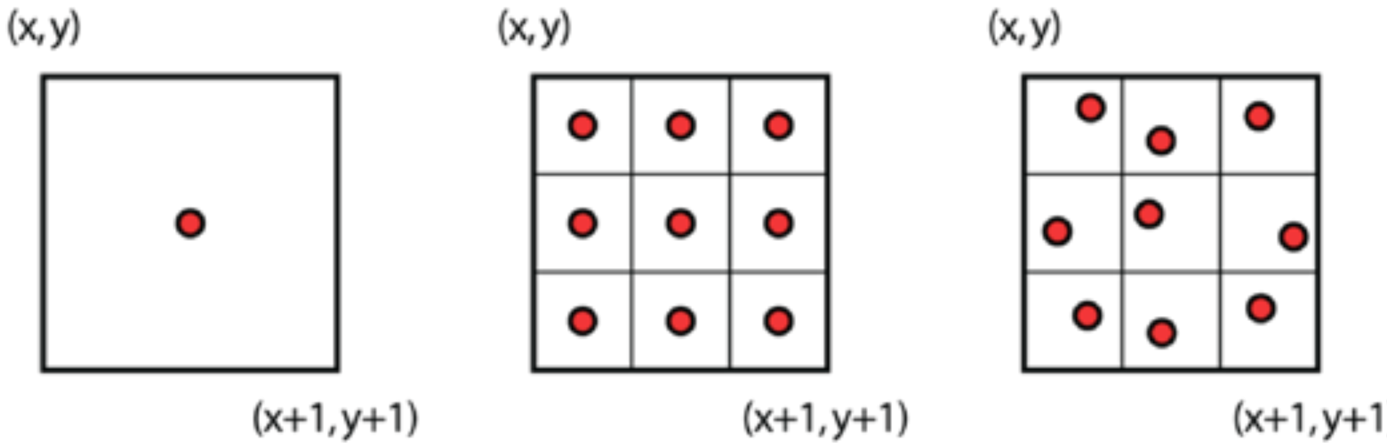
It's because we only use 1 sample/pixel



Instead use NxN stratified samples

Super Sampling

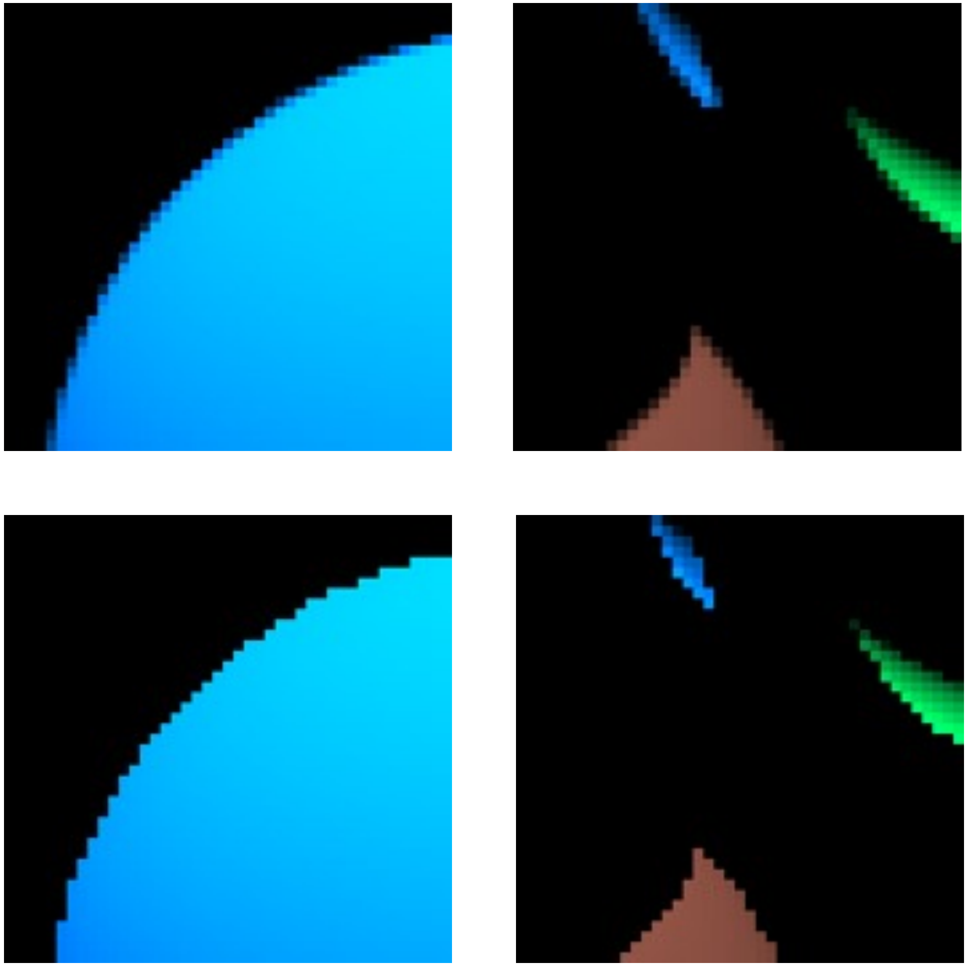
It’s because we only use 1 sample/pixel



Instead use NxN stratified samples

Hint: uniform() *returns a random value [0, 1)*

Super Sampling

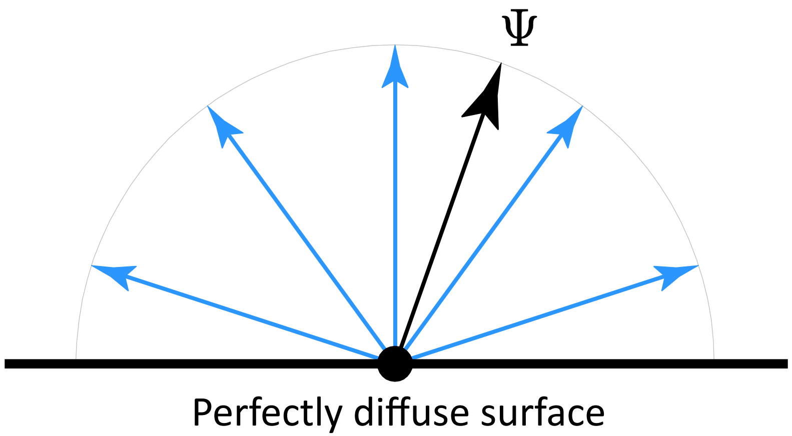


Assignment 1

- Diffuse Reflection
- Whitted ray tracing
  - Shadows
  - Reflections
  - Refractions
- Super sampling
- Blinn-Phong Shading *(Optional)*
- Ray-Triangle Intersection

Diffuse Reflection

Until now we considered a simple BRDF

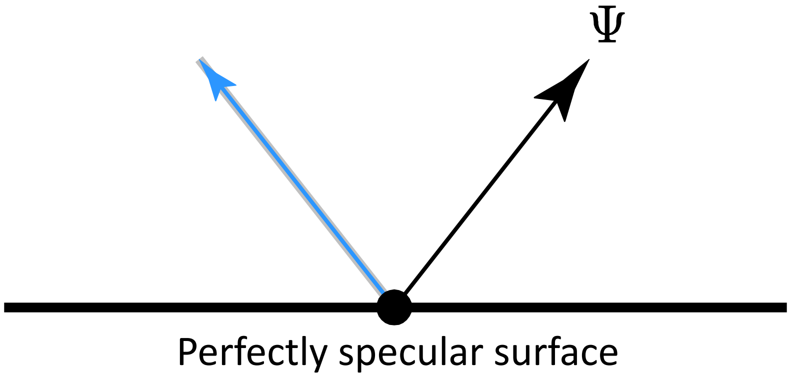


Light is reflected equally in all directions

– *BRDF is constant =* $\boldsymbol{k_d}$

Specular Reflection

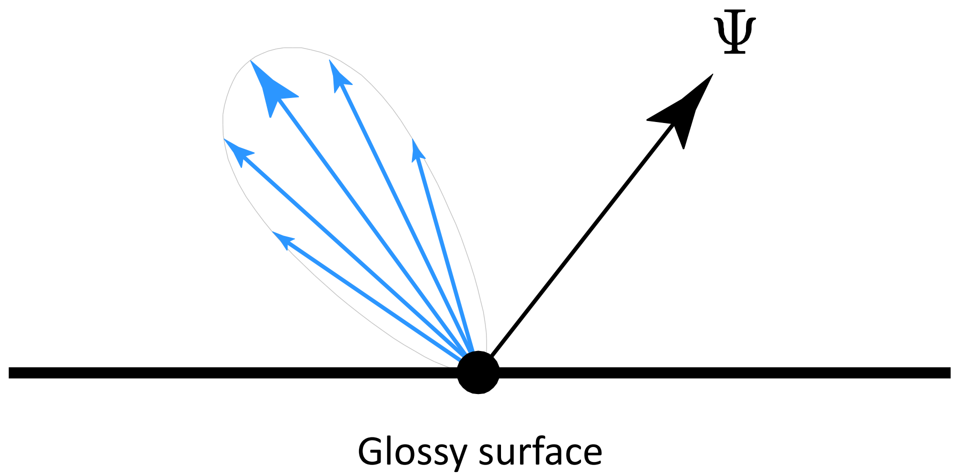
The Whitted tracer has perfectly specular surfaces (perfect mirrors)



Light is reflected in the exact reflection direction only.

Specular Reflection in BRDF

Now we add another glossy specular reflection



Recall

$$L_{out}(x \rightarrow \Theta) = L_{in}(x \leftarrow \Psi) f_r(x, \Psi \leftrightarrow \Theta) cos(N_x, \Psi)$$

- Incoming radiance `light->getRadiance();`
- BRDF `is.mMaterial.evalBRDF(is, `Ψ`);`
- Incident angle `max(`Ψ`.dot(is.mNormal), 0.0f);`

Recall

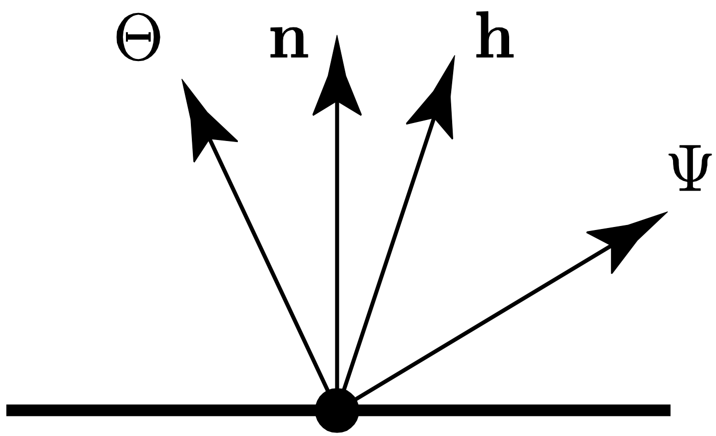
$$L_{out}(x \rightarrow \Theta) = L_{in}(x \leftarrow \Psi) f_r(x, \Psi \leftrightarrow \Theta) cos(N_x, \Psi)$$

- Incoming radiance light->getRadiance();
- BRDF is.mMaterial.evalBRDF(is, Ψ);
- Incident angle max(Ψ.dot(is.mNormal), 0.0f);

Add new sub-class of <<Material>>

Implement your own evalBRDF-function

Blinn-Phong Shading (optional)



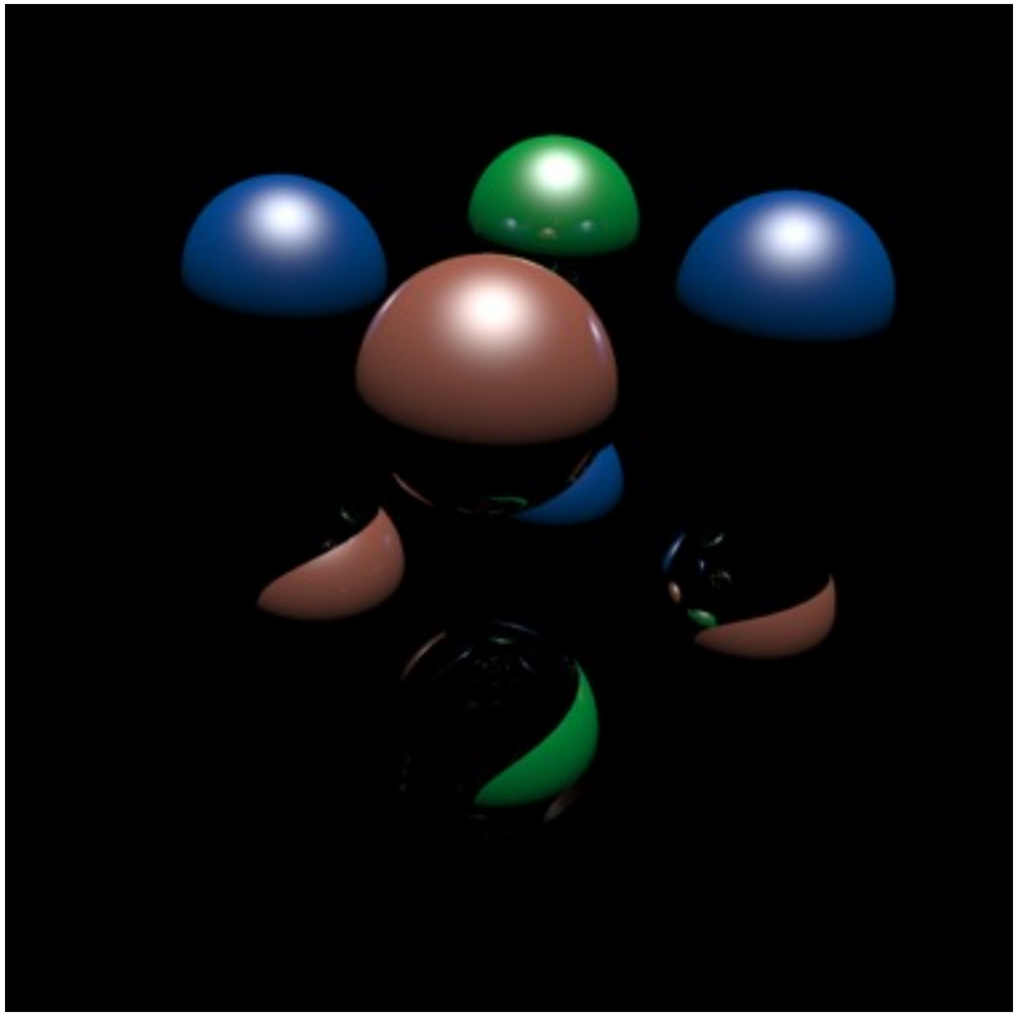
$$\mathbf{h} = \frac{\boldsymbol{\Psi}+\boldsymbol{\Theta}}{|\boldsymbol{\Psi}+\boldsymbol{\Theta}|}$$

$$\text{Diffuse component} = k_d$$

$$\text{Specular component} = k_s(\mathbf{h}\cdot\mathbf{n})^s$$

Result = Specular component + Diffuse Component

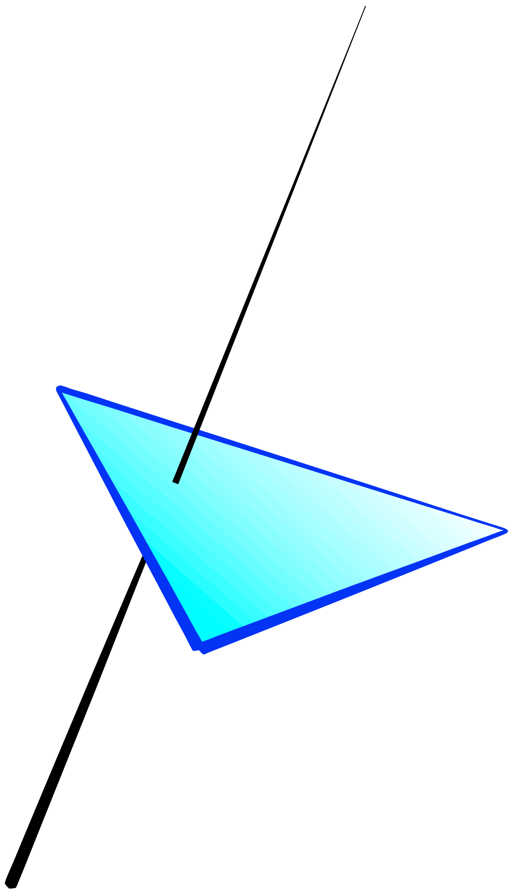
Blinn-Phong Shading (optional)



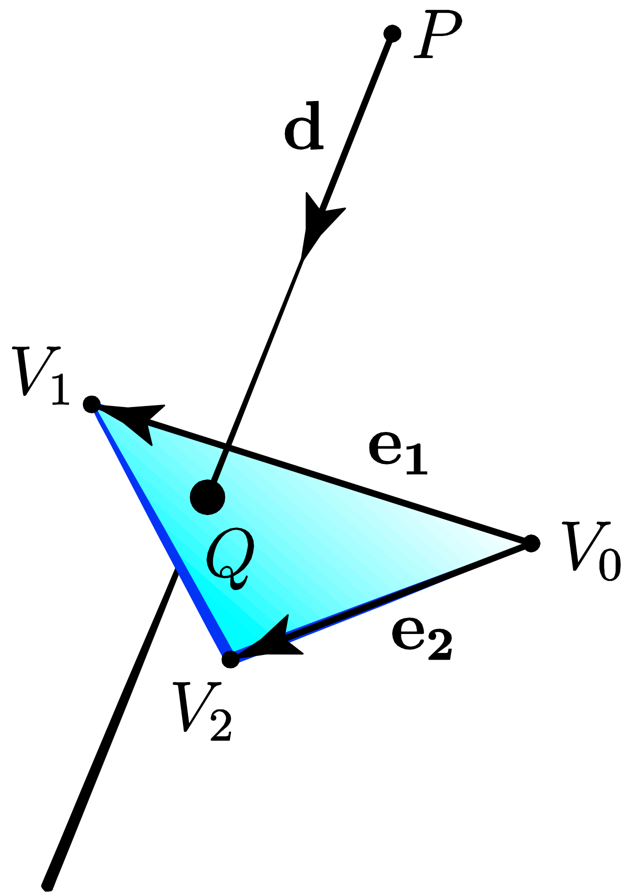
Assignment 1

- Diffuse Reflection
- Whitted ray tracing
  - Shadows
  - Reflections
  - Refractions
- Super sampling
- Blinn-Phong Shading *(Optional)*
- Ray-Triangle Intersection

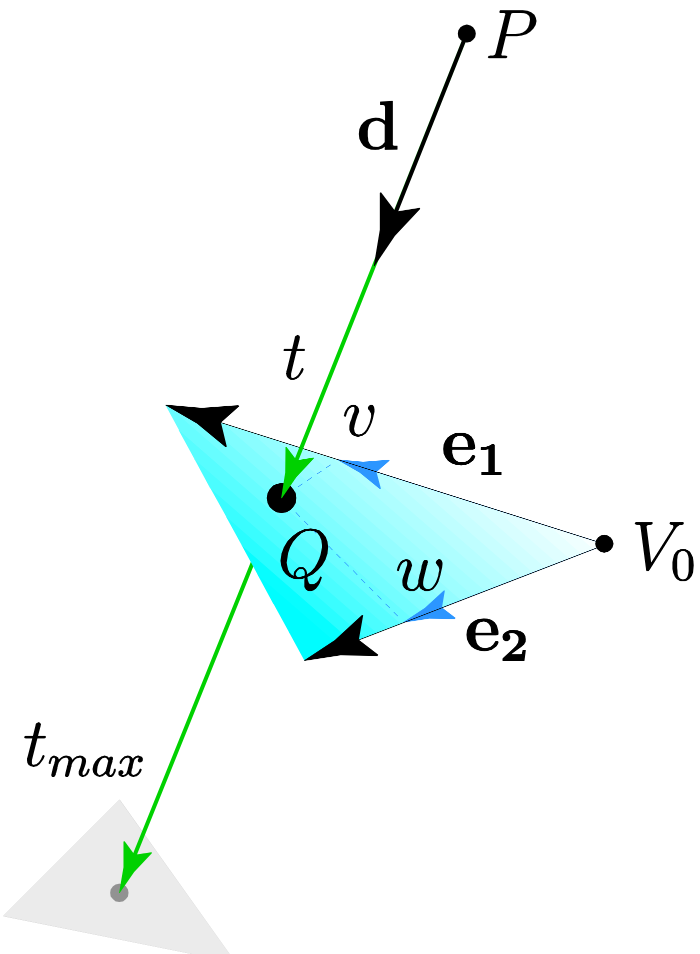
Ray-Triangle Intersection



Ray-Triangle Intersection



Ray-Triangle Intersection



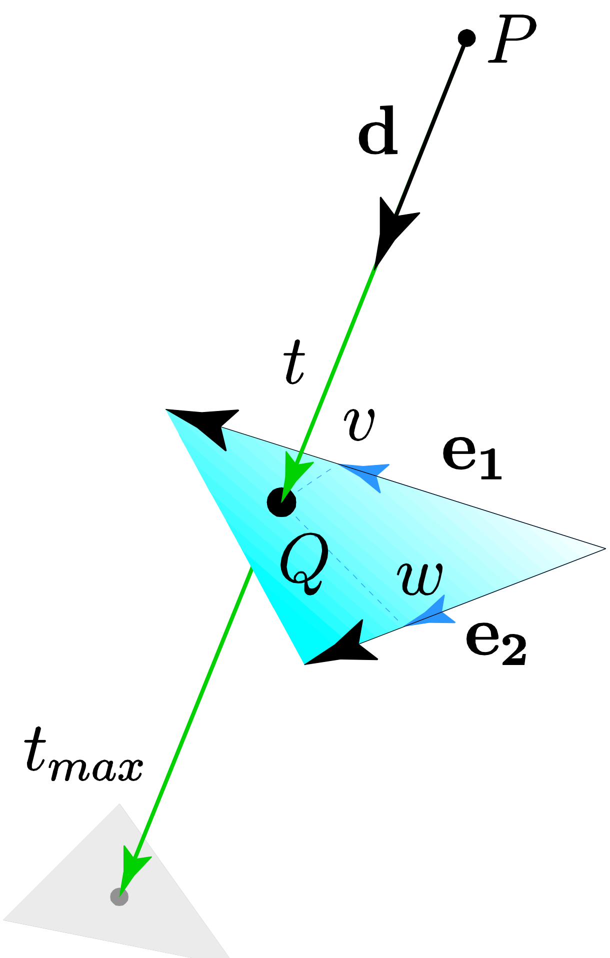
Q is inside the triangle if

$$Q = V_0 + v\mathbf{e}_1 + w\mathbf{e}_2$$
$$v \geq 0$$
$$w \geq 0$$
$$v + w \leq 1$$

Must also make sure that

$$t_{min} < t < t_{max}$$

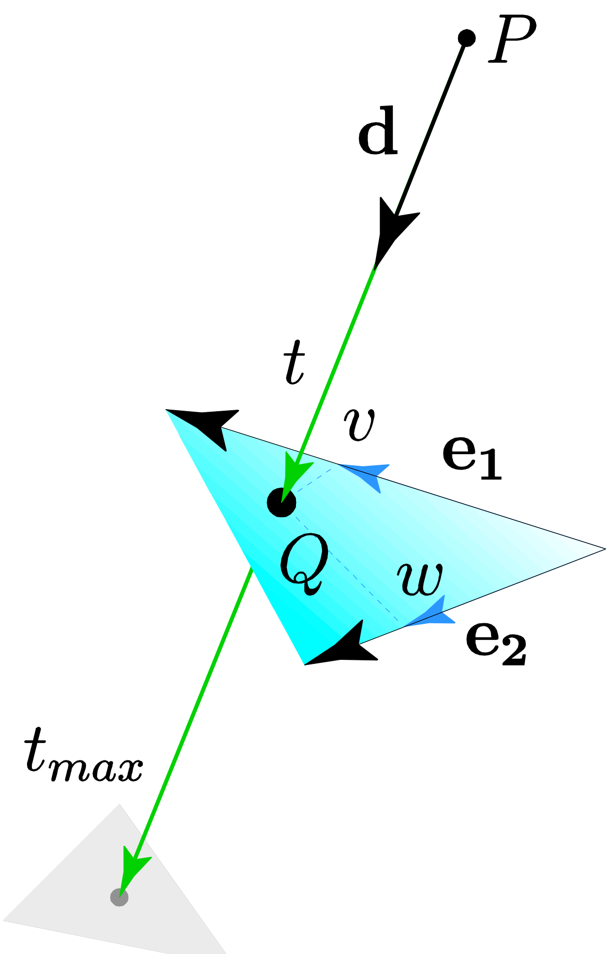
Ray-Triangle Intersection



Substitute *Q* with *ray* and solve for *v, w* and *t*

$$P + t\mathbf{d} = V_0 + v\mathbf{e}_1 + w\mathbf{e}_2$$

Ray-Triangle Intersection



Gather *v, w* and *t* to one side

$$P + t\mathbf{d} = V_0 + v\mathbf{e}_1 + w\mathbf{e}_2$$

=>

$$-t\mathbf{d} + v\mathbf{e}_1 + w\mathbf{e}_2 = P - V_0$$

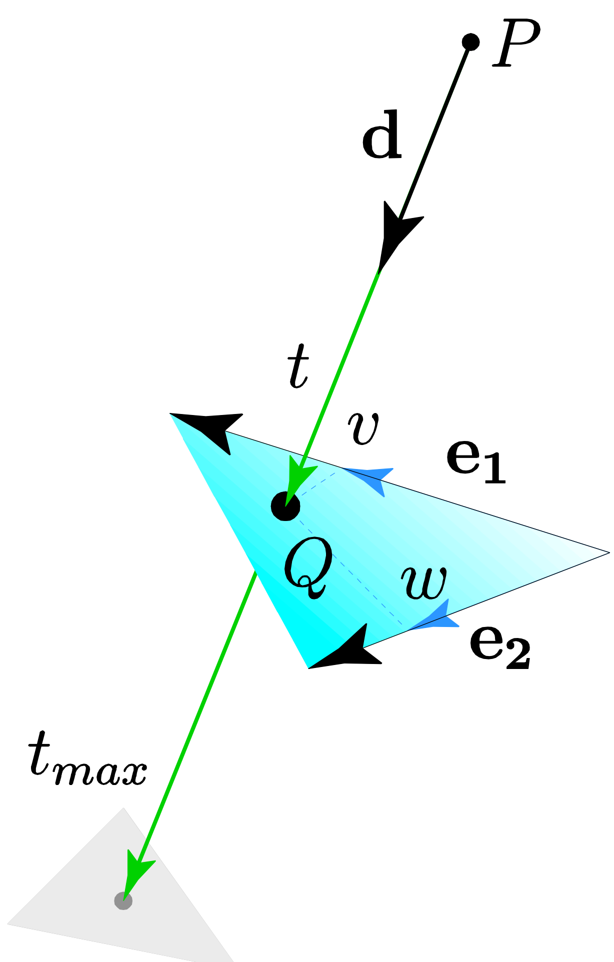
=>

$$\boxed{\begin{pmatrix} -d^x & e_1^x & e_2^x \\ -d^y & e_1^y & e_2^y \\ -d^z & e_1^y & e_2^z \end{pmatrix} \begin{pmatrix} t \\ v \\ w \end{pmatrix} = P - V_0 = \mathbf{r}}$$

Neat, a 3x3 system!

Solve with Cramer's Rule

Ray-Triangle Intersection



Start by calculating the denominator *s*, which is the determinant of the 3x3 matrix

$$\boxed{s = -\mathbf{d} \cdot (\mathbf{e}_1 \times \mathbf{e}_2)}$$

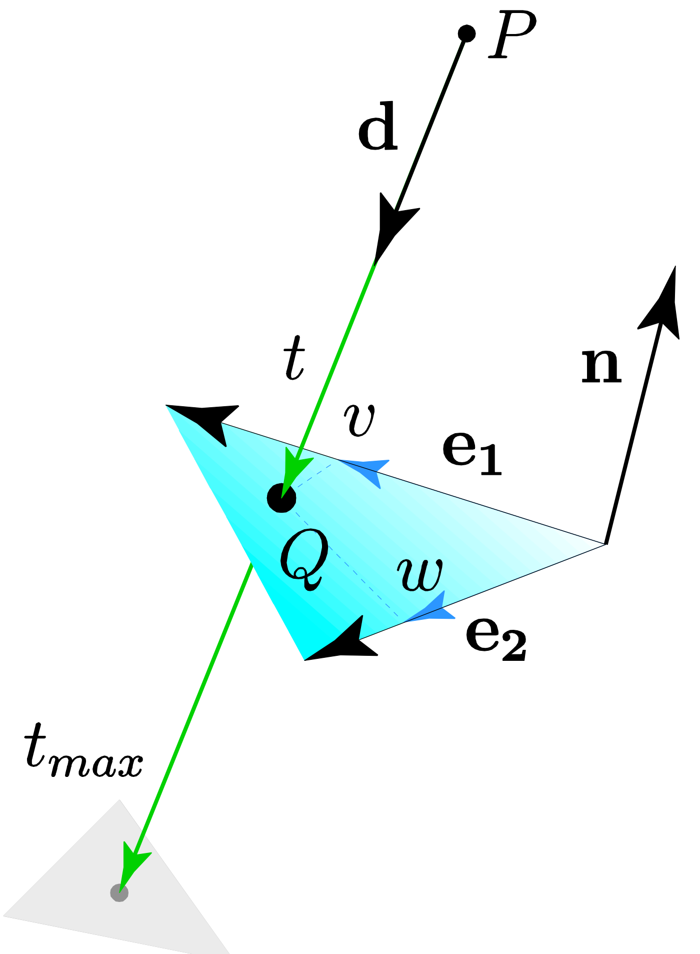
Next we solve for *v, w, t*

$$\begin{aligned} t &= \det\begin{pmatrix} \mathbf{r} & \mathbf{e}_1 & \mathbf{e}_2 \end{pmatrix} / s \\ v &= \det\begin{pmatrix} -\mathbf{d} & \mathbf{r} & \mathbf{e}_2 \end{pmatrix} / s \\ w &= \det\begin{pmatrix} -\mathbf{d} & \mathbf{e}_1 & \mathbf{r} \end{pmatrix} / s \end{aligned}$$

But wait! We can optimize all this by:

- Using scalar triple product magic
- Removing common sub expressions

Ray-Triangle Intersection



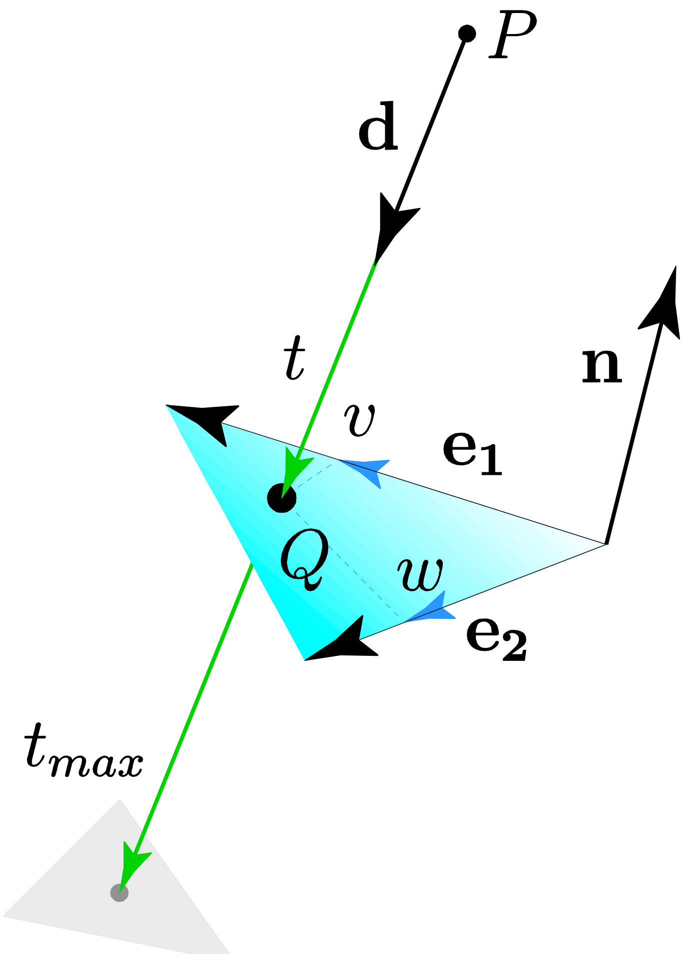
Common sub expressions

$$\mathbf{n} = \mathbf{e}_1 \times \mathbf{e}_2$$
$$\mathbf{q} = -\mathbf{d} \times \mathbf{r}$$

Optimized expressions

$$s = -\mathbf{d} \cdot \mathbf{n}$$
$$t = \frac{\mathbf{r} \cdot \mathbf{n}}{s}$$
$$v = \frac{\mathbf{q} \cdot \mathbf{e}_2}{s}$$
$$w = \frac{-\mathbf{e}_1 \cdot \mathbf{q}}{s}$$

Ray-Triangle Intersection



Putting it all together

$\mathbf{e_1} = \mathbf{v_1} - \mathbf{v_0}$
$\mathbf{e_2} = \mathbf{v_2} - \mathbf{v_0}$

Determinant/facing

$\mathbf{n}$ = CROSS($\mathbf{e_1}$, $\mathbf{e_2}$)
s = DOT(-**d**, **n**)
if (|s| < *epsilon*) return NO INTERSECTION

s_inv = 1 / s

t

r = **p** - $\mathbf{v_0}$
t = DOT(**r**, **n**) * s_inv
if (t <= t_{min} || t >= t_{max}) return NO INTERSECTION

v

q = CROSS(-**d**, **r**)
v = DOT(**q**, $\mathbf{e_2}$) * s_inv

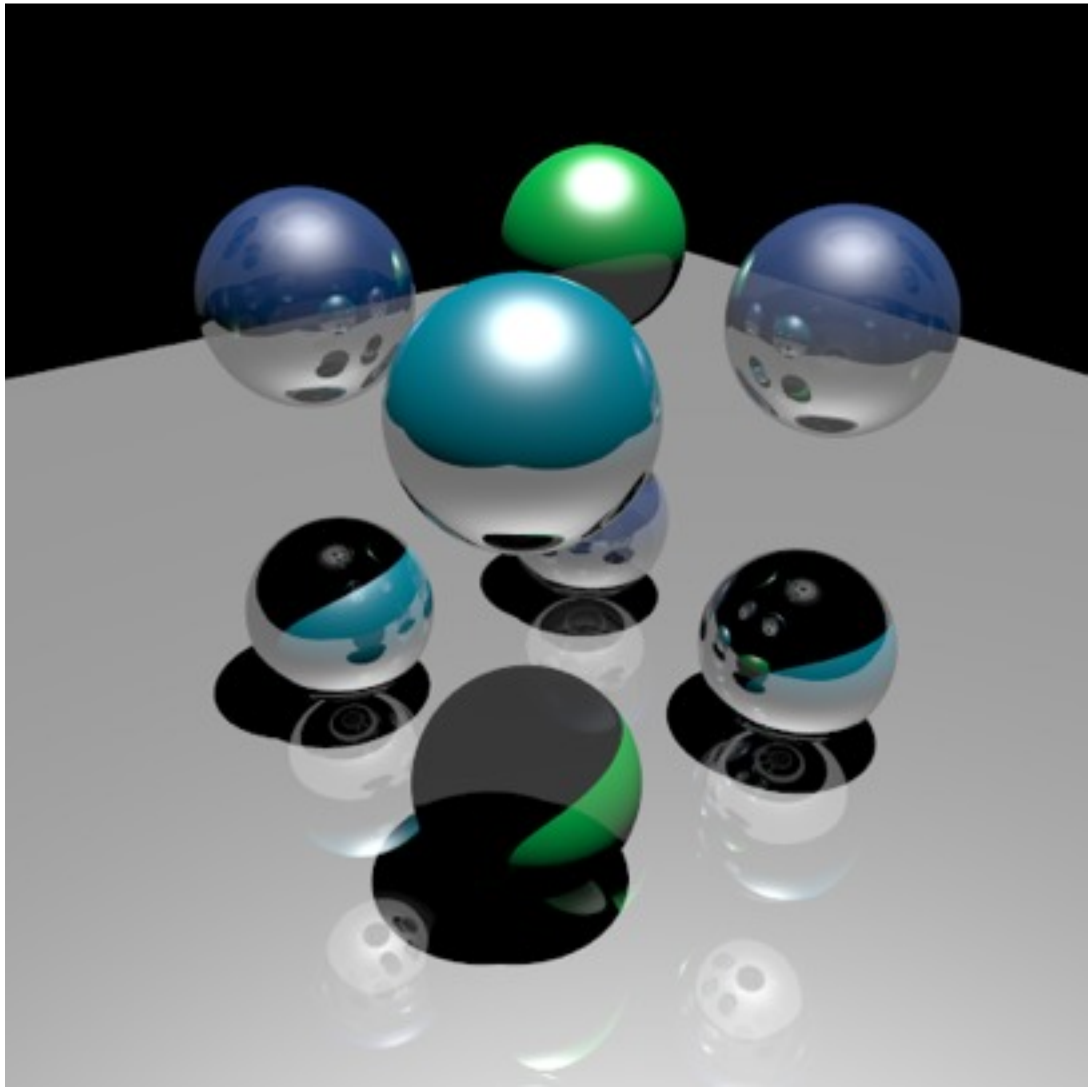
w

if (v < 0) return NO_INTERSECTION

w = DOT(-$\mathbf{e_1}$, **q**) * s_inv
if (w < 0 || v + w > 1) return NO INTERSECTION

(This can be optimized further by pushing the division to the end of the function...)

Finally...



Running on the Lab Machines

Debugging the code:

- Run in debug mode
- Use breakpoints and look at variables
- Still not working? The forum is a good place to find answers and ask questions

Hot tip:

- Debug mode is slow, Release is fast!

That is all.

The first assignment is out now!

We'll be active on the forum, so be sure to check in if you have any comments or questions!

Fin