Computer Graphics

Ray Tracing

P. Willemsen

University of Minnesota Duluth

March 6, 2014

Outline

Objective of Computer Graphics

Goal Create images (of potentially plausible, or realistic, or real-life situations) using automated algorithms.

Focus *3D* - 3-Dimensional computer graphics

Render The process of generating an image, to render a scene.

Today's Goal

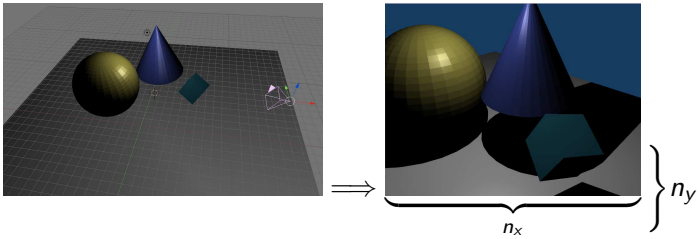
- Understand the basic structure of the ray tracing algorithm
- Become familiar with the ray tracing algorithm components
- Understand the role of the Basis Vectors

What you should be working on now:

- Get my code compiling and look at the test_pngWrite.cpp executable.
- Start constructing your own sets of classes and your own library.
- Build and test your own 3D Basis Vectors class!

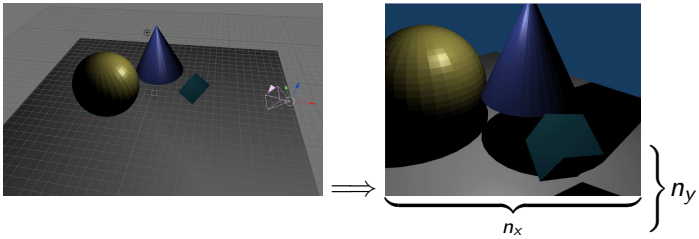
What does 3D mean?

- *Images* are 2-Dimensional...



What does 3D mean?

- *Images* are 2-Dimensional...



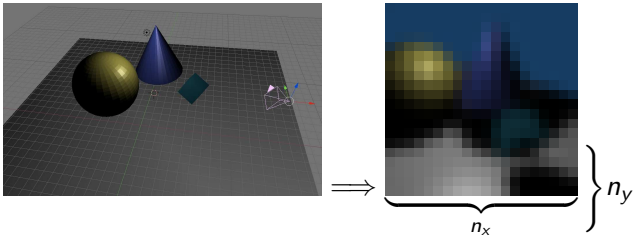
Projection Three-dimensional information captured by two-dimensional image plane of camera. For instance, the light is projected onto the camera.

- 3D Representation - objects, cameras, lights specified in 3D coordinate system
- 2D Image - information presented on a n_x by n_y resolution image

Computing a projection on to an image

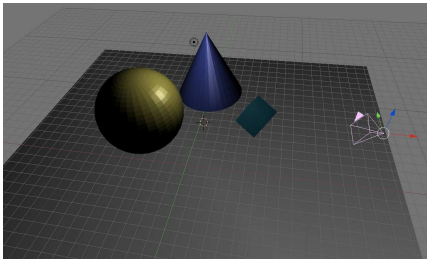
Consider two algorithms:

- Object-order algorithm - outer loop tied to objects in scene
- Image-order algorithm - outer loop tied to pixels in image



Which has higher complexity?

Image Generation - Object Order



im $\leftarrow$ new image($n_x n_y * 3$)
for all $o_k \in$ objects **do**
  $P_{o_k} \leftarrow$ projectOntoImage(o_k)
  for all $p_{i,j} \in P_{o_k}$ **do**
    im$[p_{i,j}] \leftarrow$ shader(o_k, $p_{i,j}$)
  end for
end for

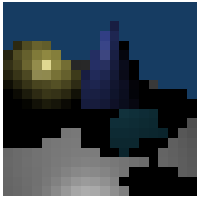
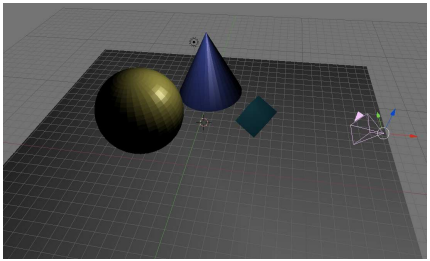
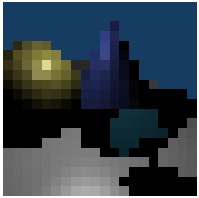


Image Generation - Image Order



im $\leftarrow$ new image($n_x n_y * 3$)
for all $p_{i,j} \in$ im **do**
  ray $\leftarrow$ computeRay($p_{i,j}$)
  $o_k \leftarrow$ findFirstObject(ray)
  im$[p_{i,j}] \leftarrow$ shader(o_k)
end for



What is a ray tracer?

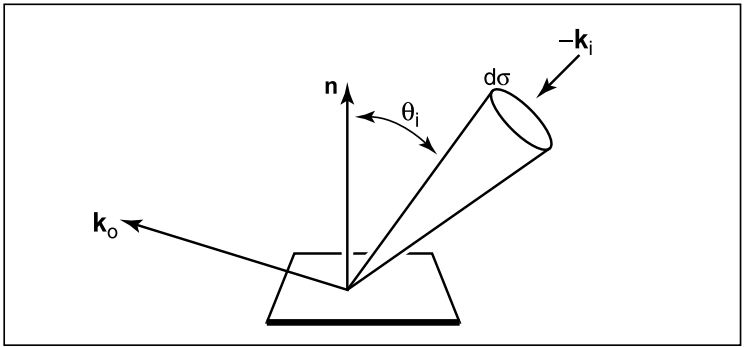
- Mechanism for *sampling* light intensity (or more appropriately, *radiance*)
- *Rays* to *intersect* objects in a scene
- At point of intersection between ray and object, we sample the radiance in a small area

Radiance Characterizes the amount of light energy that is emitted or reflected from a small surface area. This is a view dependent characteristic!

Rendering Equation

Ray tracing samples radiance to approximate the rendering equation [Kajiya, 1986]

$$L_s(\vec{k_o}) = L_e(\vec{k_o}) \int_{\text{all } \vec{k_i}} \rho(\vec{k_i}, \vec{k_o}) L_f(\vec{k_i}) \cos\theta_i d\sigma_i$$



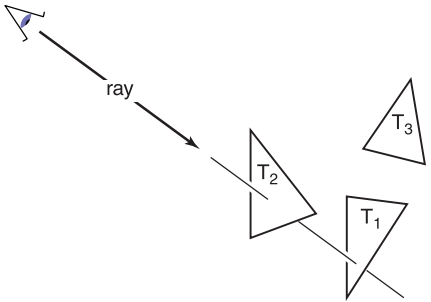
Components for Image Generation

Scene	A basic container for everything
Objects	Objects, such as spheres, triangles, etc... that can be rendered
Shaders	Descriptions of the materials associated with objects in the scene
Camera	Contains information about how the scene should be rendered
Light	How should the scene be lit

Basic Ray Tracer

Has three major parts:

- Ray Generation
- Ray Intersection
- Shading




```
im ← new image( nx ny ∗ 3 )
for all pi,j ∈ im do
    {Compute Viewing Ray}
    rayi,j ← computeRay( i, j )
    {Intersect with Objects}
    ok ← findFirstObject( ray )
    {Set pixel color based on object shader}
    im[pi,j] ← shader(ok)
end for
```

Compute the Viewing Ray

Dependent on the camera's configuration:

- Orthonormal coordinate system: $\vec{U}, \vec{V}, \vec{W}$
- Position: $\vec{e}$
- Image plane width, height, and aspect ratio
- Resolution of final image: n_x by n_y pixels

But, requires the use of a well-constructed 3D Basis...

What's a 3D Basis?

So, what is a 3D basis? Recall,

- A set of three 3D vectors
- Each vector is at right angles to the other vectors
- Each vector is of unit length

You will need to recall some basic Linear Algebra to help you here. As a first start, review vectors!

Vector Review

Some basic, yet important concepts:

- Vectors have both length and direction!
- Vector addition is geometrically intuitive. *Connect the arrows.*
- Vectors are linear combinations of other vectors: $\vec{a} = x_a\vec{X} + y_a\vec{Y} + z_a\vec{Z}$
- Vectors length is determined by: $\|\vec{r}\| = \sqrt{x_a^2 + y_a^2 + z_a^2}$

Vector multiplication is important!

- Dot Product - Scalar product multiplication - computes the cosine of the angle between the vectors!

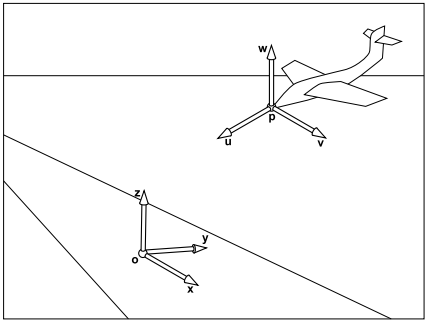
$$\vec{a} \cdot \vec{b} = \|\vec{a}\|\|\vec{b}\| \cos\theta$$

- Dor Product - recall that $\vec{x} \cdot \vec{x} = \vec{y} \cdot \vec{y} = 1$ and $\vec{x} \cdot \vec{y} = 0$
- Cross Product - vector multiplication that returns a 3D vector that is perpendicular to the two input vectors!

$$axb = (y_a z_b - z_a y_b, z_a x_b - x_a z_b, x_a y_b - y_a x_b$$

Coordinate Systems

- Global coordinate system is implicitly defined
  - Origin: $\vec{o}$
  - $\vec{X} = (1, 0, 0)$, $\vec{Y} = (0, 1, 0)$, $\vec{Z} = (0, 0, 1)$
  - No real need to explicit representation
- Every object has its own coordinate system
  - Must represent these explicitly!
  - Referred to as *local coordinate systems*
  - Origin: $\vec{p}$
  - Basis: $\vec{U}, \vec{V}, \vec{W}$



Creating an Orthonormal Basis

- Often need to construct a set of basis vectors
- General mechanism when given a single vector, $\vec{a}$, construct a $\vec{U}, \vec{V}, \vec{W}$ basis
- *Requirement*: create and align $\vec{W}$ with $\vec{a}$ while not caring too much about $\vec{U}$ and $\vec{V}$
  1. $\vec{W} = -\vec{a}/|\vec{a}|$
  2. Pick $\vec{t}$ such that it is NOT colinear with $\vec{W}$
  3. $\vec{U} = \vec{t}x\vec{W}/|\vec{t}x\vec{W}|$
  4. $\vec{V} = \vec{W}x\vec{U}$
- Choose t such that it is W and then modify the smallest magnitude component to 1

Build your own Basis

Try this now. Recall

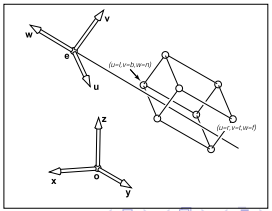
1. $\vec{W} = -\vec{a}/||\vec{a}||$
2. Pick $\vec{t}$ such that it is NOT colinear with $\vec{W}$
3. $\vec{U} = \vec{t}x\vec{W}/||\vec{t}x\vec{W}||$
4. $\vec{V} = \vec{W}x\vec{U}$

- Cross Product
- $\vec{a}x\vec{b} =$
  $(y_a z_b - z_a y_b, z_a x_b - x_a z_b, x_a y_b - y_a x_b$

Constructing Basis from 2 Vectors

- With two vectors, we have more constraints
- Good for setting up a camera's basis for your code
- Input vectors need not be orthonormal
- Notice that $\vec{V}$ will end up being close to $\vec{b}$
- $\vec{b}$ could be set to an idealized Up value, such as $\vec{b} = (0, 1, 0)$ (*Hint:* watch out for $\vec{W}$ being co-linear with $\vec{b}$)

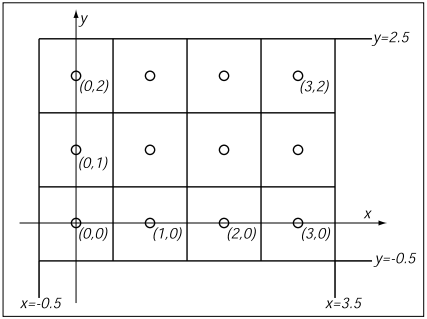
- **Note:** To maintain right-handed system, we will provide an input vector, called $\vec{gaze}$ that will be the direction that the camera is viewing
- Given $\vec{gaze}$ and $\vec{b}$ where $\vec{b}$ guides the selection of $\vec{V}$, we can
  1. $\vec{W} = -\vec{gaze}/||\vec{gaze}||$
  2. $\vec{U} = \vec{b}x\vec{W}/||\vec{b}x\vec{W}||$
  3. $\vec{V} = \vec{W}x\vec{U}$



Compute the Viewing Ray

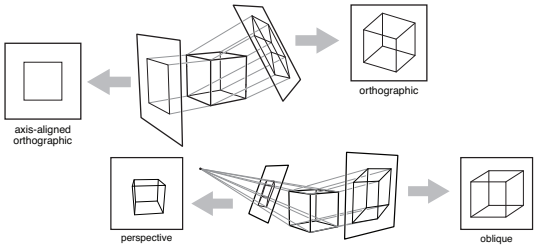
Now that camera's basis is defined, need info on the image

- Image plane width and height
- Resolution of final image: n_x by n_y pixels
- Domain of pixel coords: $[-0.5, n_x - 0.5]x[-0.5, n_y - 0.5]$
- Aspect ratio: width / height



Constructing Views

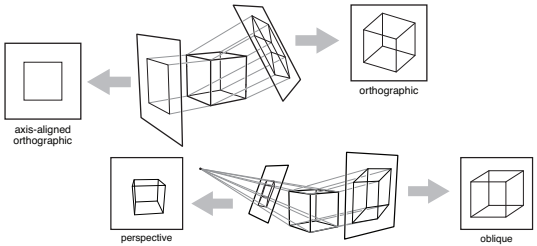
With camera basis and image coordinates, we can generate different views of the 3D representation



- Take 3D representation and "project" it on to 2D plane (the image plane)
- Many different ways to compute this projection
  - Parallel Projection
  - Perspective Projection

Constructing Views

With camera basis and image coordinates, we can generate different views of the 3D representation

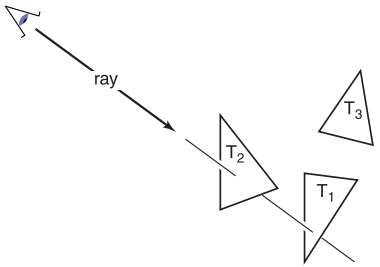


- Take 3D representation and "project" it on to 2D plane (the image plane)
- Many different ways to compute this projection
  - Parallel Projection
  - Perspective Projection
  - Others?

Ray Tracing - Computing Rays

Recall that ray tracers have three major parts:

- Ray Generation
- Ray Intersection
- Shading

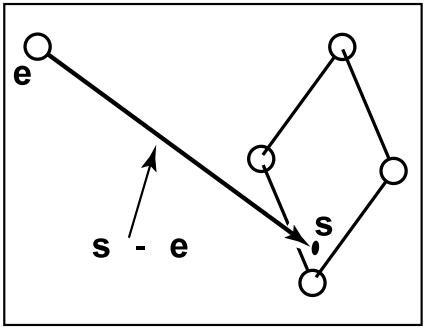



```
im ← new image( n_x n_y * 3 )
for all p_{i,j} ∈ im do
    {Compute Viewing Ray}
    ray_{i,j} ← computeRay( i, j )
    {Intersect with Objects}
    o_k ← findFirstObject( ray )
    {Set pixel color based on object shader}
    im[p_{i,j}] ← shader(o_k)
end for
```

Ray Tracing - Computing Rays

Computing rays will require the parametric line equation

- What is t at different locations?
- Where is s located?




```
im ← new image( n_x n_y * 3 )
for all p_{i,j} ∈ im do
    {Compute Viewing Ray}
    ray_{i,j} ← computeRay( i, j )
    {Intersect with Objects}
    o_k ← findFirstObject( ray )
    {Set pixel color based on object shader}
    im[p_{i,j}] ← shader(o_k)
end for
```

Parametric Line Equation

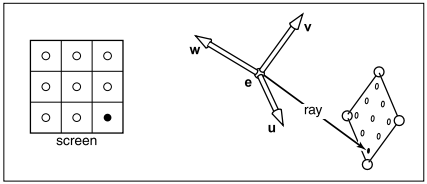
$$\vec{p}(t) = \vec{e} + t(\vec{s} - \vec{e})$$

Compute Viewing Rays

Notice the orthonormal basis of the camera

- Basis: $\vec{U}$, $\vec{V}$, and $\vec{W}$
- Origin: $\vec{e}$
- Often called the camera frame

- Which direction goes toward the image plane?
- How many samples in this example?



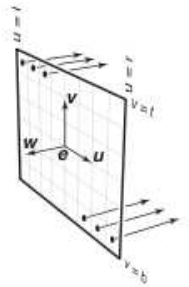
Orthographic Ray Generation

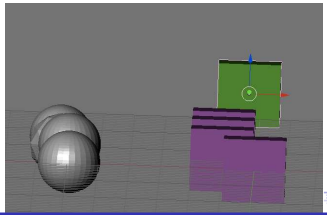
All rays have the same direction: $\vec{W}$
Image plane is defined in terms of

- l - left boundary of image plane
- r - right boundary of image plane
- b - bottom boundary of image plane
- t - top boundary of image plane

Need to fit image of $n_x \times n_y$ pixels into rectangle of size $(r-l)x(t-b)$

- $u = l + (r-l)(i+0.5)/n_x$
- $v = b + (t-b)(j+0.5)/n_y$





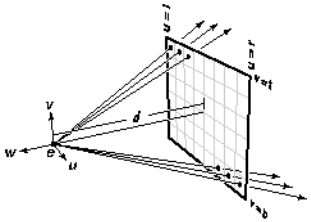
Orthographic Ray Generation

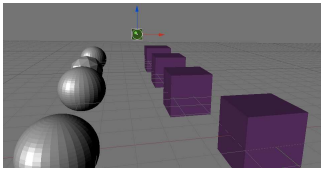
im $\leftarrow$ new image($n_x n_y * 3$)
for all $p_{i,j} \in$ im **do**
  {*Compute Viewing Ray*}
  $u = l + (r - l)(i + 0.5)/n_x$
  $v = b + (t - b)(j + 0.5)/n_y$
  Ray r;
  r.direction $= -\vec{W}$;
  r.origin $= \vec{e} + u\vec{U} + v\vec{V}$;
  {*Intersect with Objects*}
  $o_k \leftarrow$ findFirstObject(r)
  {*Set pixel color based on object shader*}
  im$[p_{i,j}] \leftarrow$ shader(o_k)
end for

Perspective Ray Generation

What's different?

- $u = l + (r - l)(i + 0.5)/n_x$
- $v = b + (t - b)(j + 0.5)/n_y$

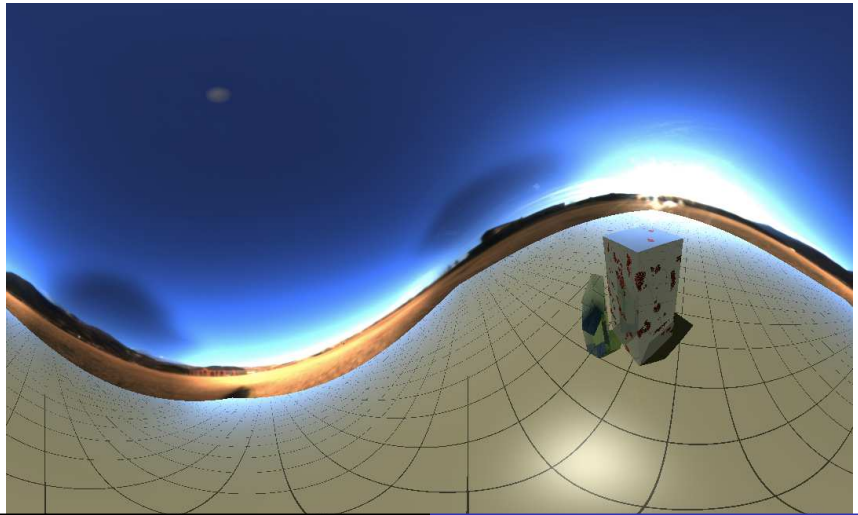




Perspective Ray Generation

im $\leftarrow$ new image($n_x n_y * 3$)
for all $p_{i,j} \in$ im **do**
  {*Compute Viewing Ray*}
  $u = l + (r - l)(i + 0.5)/n_x$
  $v = b + (t - b)(j + 0.5)/n_y$
  Ray r;
  r.direction = $-d\vec{W} + u\vec{U} + v\vec{V}$**;**
  r.origin = $\vec{e}$**;**
  {*Intersect with Objects*}
  $o_k \leftarrow$ findFirstObject(r)
  {*Set pixel color based on object shader*}
  im$[p_{i,j}] \leftarrow$ shader(o_k)
end for

Other types of Ray Generation



Ray Tracing Algorithm

- Have method(s) for generating rays that go through image plane
- Rays sample the light intensity in image plane space
- Next, must compute intersections between *rays* and objects in your scene
- Determines the light intensity that is "behind" the *window* of the pixel that the ray traveled through

im $\leftarrow$ new image($n_x n_y * 3$)
for all $p_{i,j} \in$ im **do**
  {*Compute Viewing Ray*}
  $u = l + (r - l)(i + 0.5)/n_x$
  $v = b + (t - b)(j + 0.5)/n_y$
  Ray r;
  r.direction $= -d\vec{W} + u\vec{U} + v\vec{V}$**;**
  r.origin $= \vec{e}$**;**
  {*Intersect with Objects*}
  $o_k \leftarrow$ findFirstObject(r)
  {*Set pixel color based on object shader*}
  im$[p_{i,j}] \leftarrow$ shader(o_k)
end for

Today's Goal - 02/11/14

Goals for today:

- Class structure ideas
- Ray-Object Intersection
  - Sphere
  - Triangle

Ray Intersection Tests

Why do we compute an intersection between rays and objects

- Determine if ray leaving the eye hits anything
- If it does, then we can set the color of the pixel based on object hit

We will focus on two types of objects initially

1. Sphere
2. Triangle

Other object types are possible, but any objects that will be rendered **must** support answering an intersection with ray query:

```
bool hit = object->intersect( ray );
```

We will first focus on ray-sphere intersection to understand how this will work

Ray Intersection Tests

First, all intersections are between a *parametric line* representation for our ray and typically, an *implicit equation* that describes our object.
What's a **Parametric Line**?

$$\vec{p(t)} = \vec{o} + t * \vec{dir}$$

What's an *implicit equation*?

- A function of some number of parameters that when equal to 0, the parameters define the surface
- More formally, $f(x) = 0$, $f(x, y) = 0$, or $f(x, y, z) = 0$ all represent implicit functions
- For instance, consider the implicit line equation:

$$f(x, y) = (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a = 0$$

Ray Intersection Tests

What's an *implicit equation* cont'd?

$$f(x, y) = (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a = 0$$

Consider the line formed between two points $\vec{a} = (1, 1)$ and $\vec{b} = (3, 3)$:

$$f(x, y) = (1 - 3)x + (3 - 1)y + 1 * 3 - 3 * 1 = -2x + 2y$$

- What is f(0, 0)?

Ray Intersection Tests

What's an *implicit equation* cont'd?

$$f(x, y) = (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a = 0$$

Consider the line formed between two points $\vec{a} = (1, 1)$ and $\vec{b} = (3, 3)$:

$$f(x, y) = (1 - 3)x + (3 - 1)y + 1 * 3 - 3 * 1 = -2x + 2y$$

- What is f(0, 0)?
  - $f(0, 0) = 0$
- What is f(2, 2)?

Ray Intersection Tests

What's an *implicit equation* cont'd?

$$f(x, y) = (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a = 0$$

Consider the line formed between two points $\vec{a} = (1, 1)$ and $\vec{b} = (3, 3)$:

$$f(x, y) = (1 - 3)x + (3 - 1)y + 1 * 3 - 3 * 1 = -2x + 2y$$

- What is f(0, 0)?
  - $f(0, 0) = 0$
- What is f(2, 2)?
  - $f(2, 2) = 0$
- What is f(1, 2)?

Ray Intersection Tests

What's an *implicit equation* cont'd?

$$f(x, y) = (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a = 0$$

Consider the line formed between two points $\vec{a} = (1, 1)$ and $\vec{b} = (3, 3)$:

$$f(x, y) = (1 - 3)x + (3 - 1)y + 1 * 3 - 3 * 1 = -2x + 2y$$

- What is f(0, 0)?
  - $f(0, 0) = 0$
- What is f(2, 2)?
  - $f(2, 2) = 0$
- What is f(1, 2)?
  - $f(1, 2) = -2 + 4 = 2$
- What is f(2, 1)?

Ray Intersection Tests

What's an *implicit equation* cont'd?

$$f(x, y) = (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a = 0$$

Consider the line formed between two points $\vec{a} = (1, 1)$ and $\vec{b} = (3, 3)$:

$$f(x, y) = (1 - 3)x + (3 - 1)y + 1 * 3 - 3 * 1 = -2x + 2y$$

- What is f(0, 0)?
  - $f(0, 0) = 0$
- What is f(2, 2)?
  - $f(2, 2) = 0$
- What is f(1, 2)?
  - $f(1, 2) = -2 + 4 = 2$
- What is f(2, 1)?
  - $f(2, 1) = -4 + 2 = -2$

Ray-Sphere Intersection

The same ideas work for 3D shapes, like spheres:

- Implicit Sphere: $f(x, y, z) = 0$ when a point $\vec{p} = (x, y, z)$ is on the sphere
- if $(f(x, y, z) > 0$ or $f(x, y, z) < 0$ then point $\vec{p}$ is inside or outside the sphere

Parametric Line Equation for Ray

$$p(\vec{t}) = ray\vec{}_{origin} + tray\vec{}_{dir}$$

We have an intersection with the sphere when

- We have found one or two t value(s) that provide a specific point(s) along the ray that cut through the sphere

Ray-Sphere Intersection

Implicit Sphere Equation

$$(x_p - x_{center})^2 + (y_p - y_{center})^2 + (z_p - z_{center})^2 - R^2 = 0$$

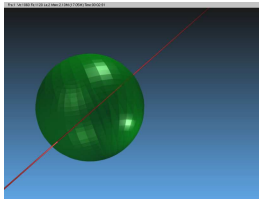
We can transform this into a *vector* equation:

$$(\vec{p} - \vec{center}) \cdot (\vec{p} - \vec{center}) - R^2 = 0$$

So, any point $\vec{p}$ that satisfies this *is on the sphere*!

- Now, just plug in our parametric line equation for $\vec{p}$!
- Recall: $\vec{p(t)} = \vec{r_o} + t\vec{r_d}$

$$(\vec{r_o} + t\vec{r_d} - \vec{center}) \cdot (\vec{r_o} + t\vec{r_d} - \vec{center}) - R^2 = 0$$



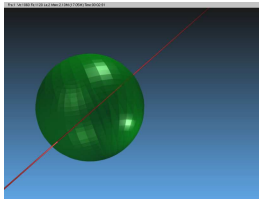
What is t for Ray-Sphere Intersection?

We start with our general form for the ray sphere intersection:

$$(\vec{r_o} + t\vec{r_d} - \vec{center}) \cdot (\vec{r_o} + t\vec{r_d} - \vec{center}) - R^2 = 0$$

Expand this by pushing the dot product through:

$$(\vec{r_d} \cdot \vec{r_d})t^2 + 2\vec{r_d} \cdot (\vec{r_o} - \vec{center})t + (\vec{r_o} - \vec{center}) \cdot (\vec{r_o} - \vec{center}) - R^2 = 0$$



Now, only need to apply the general solution to the *quadratic equation* to solve for t

$$At^2 + Bt + C = 0$$

$$t = \frac{-B \pm sqrt(B^2 - 4AC)}{2A}$$

Solve for t, but check the discriminant! If < 0 then it has no solutions!

Practical Coding - Machine Eps

Smallest Value

What is the smallest value you can add to another floating point number and still get a larger number?

Practical Coding - Machine Eps

Smallest Value

What is the smallest value you can add to another floating point number and still get a larger number?

Here's how you compute it:

```
float b = 1.0f;
float eps = 1.0f;
while ((b + eps) > b) {
  eps = eps * 0.5f;
}
```

Homework: Try it yourself and report your answers on the Moodle forum for Machine Eps.

What about multiple objects?

Multiple Objects

What happens when a ray passes through multiple objects?

What about multiple objects?

Multiple Objects

What happens when a ray passes through multiple objects?

- What are our constraints on the t value?

What about multiple objects?

Multiple Objects

What happens when a ray passes through multiple objects?

- What are our constraints on the t value?
- How can we use the t value to assist the intersection computation?
- What does the computed t value tell us?

What about multiple objects?

Multiple Objects

What happens when a ray passes through multiple objects?

- What are our constraints on the t value?
- How can we use the t value to assist the intersection computation?
- What does the computed t value tell us?

```
bool Shape::intersect( const Ray &r ) = 0;
```

What about multiple objects?

Multiple Objects

What happens when a ray passes through multiple objects?

- What are our constraints on the t value?
- How can we use the t value to assist the intersection computation?
- What does the computed t value tell us?

```
bool Shape::intersect( const Ray &r ) = 0;

bool Shape::intersect( const Ray &r, const float tmin, float &tmax ) = 0;
```

What about multiple objects?

Multiple Objects

What happens when a ray passes through multiple objects?
What's missing from this interface?

```
bool Shape::intersect( const Ray &r, const float tmin, float &tmax ) = 0;
```

What about multiple objects?

Multiple Objects

What happens when a ray passes through multiple objects?
What's missing from this interface?

```
bool Shape::intersect( const Ray &r, const float tmin, float &tmax ) = 0;

bool Shape::intersect( const Ray &r, const float tmin, float &tmax, HitStruct &hit
```

What should HitStruct contain?

Ray Triangle Intersection

What about triangles? First, need mechanism to determine if point is *in* a triangle!

- Algorithm will use barycentric coordinate representation of a triangle
- Barycentric coordinates form a non-orthogonal coordinate system using two of the edges of the triangle to define a point relative to the triangle's plane
- For instance, a point p on a triangle defined by three vertices, $\vec{a}, \vec{b}, \vec{c}$:

$$\vec{p} = \vec{a} + \beta(\vec{b} - \vec{a}) + \gamma(\vec{c} - \vec{a})$$

Barycentric Coordinate Systems

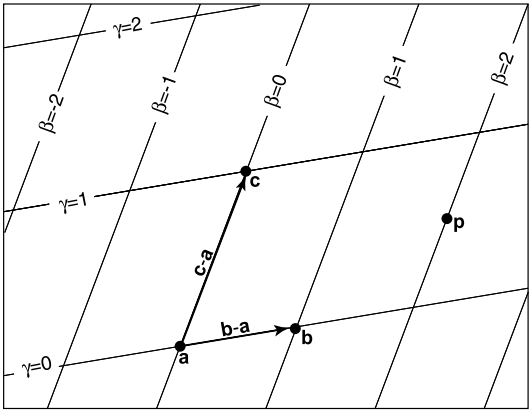
$$\vec{p} = \vec{a} + \beta(\vec{b} - \vec{a}) + \gamma(\vec{c} - \vec{a})$$

and re-working that equation, we get

$$\vec{p} = (1 - \beta - \gamma)\vec{a} + \beta\vec{b} + \gamma\vec{c}$$

let $\alpha = 1 - \beta - \gamma$, which also produces the constraint that $\alpha + \beta + \gamma = 1$

$$\vec{p}(\alpha, \beta, \gamma) = \alpha\vec{a} + \beta\vec{b} + \gamma\vec{c}$$



Barycentric Coordinates

As far as triangles are concerned, barycentric coordinates have a nice property that a point $\vec{p}$ is *inside* a triangle **iff**:

$$0 < \alpha < 1$$

$$0 < \beta < 1$$

$$0 < \gamma < 1$$

For future note, these coefficients can be used to smoothly interpolate across the surface of a triangle.

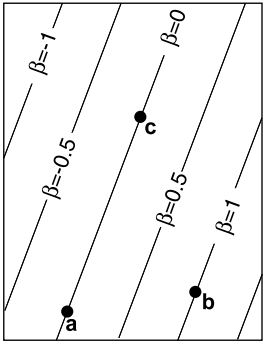
Barycentric Coordinates Defined

- β and γ are easily calculated using the implicit line equation
- Barycentric coordinates represent the signed, scaled distance from the lines on the triangles sides

For instance, we can map β between 0 and 1 across the face of the triangle.

- We would like points on the line $\vec{a} \rightarrow \vec{c}$ to be where $\beta = 0$
- Similarly, at point $\vec{b}$, we would like $\beta = 1$

$$\beta = \frac{f_{ac}(x, y)}{f_{ac}(x_b, y_b)}$$



Barycentric Coordinates Defined

Recall the implicit line equation:

$$f_{ab}(x, y) = (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a = 0$$

which when applied to beta let's us normalize our values into the 0 to 1 range for points on the triangle

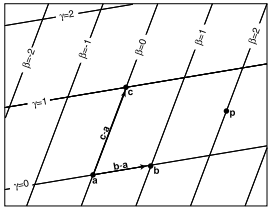
$$\beta = \frac{f_{ac}(x, y)}{f_{ac}(x_b, y_b)}$$

Similarly, γ can be defined:

$$\gamma = \frac{f_{ab}(x, y)}{f_{ab}(x_c, y_c)}$$

with

$$\alpha = 1 - \beta - \gamma$$



Ray Triangle Intersection

We want the point $\vec{p}$ to be the point that our ray intersects the plane of the triangle!

Barycentric coordinate values β and γ can tell us if that point is in the triangle by the following checks:

- iff $\beta > 0$
- iff $\gamma > 0$
- iff $\beta + \gamma < 1$

So, set our ray equal to $\vec{p}$:

$$\vec{r_o} + t\vec{r_d} = \vec{a} + \beta(\vec{b} - \vec{a}) + \gamma(\vec{c} - \vec{a})$$

Ray Triangle Intersection

We know everything except t, β, γ:

$$\vec{r_o} + t\vec{r_d} = \vec{a} + \beta(\vec{b} - \vec{a}) + \gamma(\vec{c} - \vec{a})$$

We can rewrite this *vector* equation into three component-based equations:

$$x_{r_o} + tx_{r_d} = x_a + \beta(x_b - x_a) + \gamma(x_c - x_a)$$

$$y_{r_o} + ty_{r_d} = y_a + \beta(y_b - y_a) + \gamma(y_c - y_a)$$

$$z_{r_o} + tz_{r_d} = z_a + \beta(z_b - z_a) + \gamma(z_c - z_a)$$

Three equations. Three unknowns. We can solve with a linear system solver:

$$Ax = b$$

Ray Triangle Intersection

We know everything except t, β, γ:

$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_{r_d} \\ y_a - y_b & y_a - y_c & y_{r_d} \\ z_a - z_b & z_a - z_c & z_{r_d} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_e \\ y_a - y_e \\ z_a - z_e \end{bmatrix}$$

You can solve this with Cramer's rule using 3x3 matrices and determinants.

However, the book decomposes the above linear system for you into the following using dummy variables a-l to represent the computations:

$$\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} j \\ k \\ l \end{bmatrix}$$

Ray Triangle Intersection

Using Cramer's rule, you can now solve directly for t, β, γ:

$$\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} j \\ k \\ l \end{bmatrix}$$

where

$$a = (x_a - x_b), b = (y_a - y_b), c = (z_a - z_b)$$

$$d = (x_a - x_c), e = (y_a - y_c), f = (z_a - z_c)$$

$$g = x_{r_d}, h = y_{r_d}, i = z_{r_d}$$

$$j = (x_a - x_e), k = (y_a - y_e), l = (z_a - z_e)$$

taken from the standard linear system in the previous slide.

Ray Triangle Intersection

Solve directly for t, β, γ using the following:

$$\beta = \frac{j(ei - hf) + k(gf - di) + l(dh - eg)}{M}$$

$$\gamma = \frac{i(ak - jb) + h(jc - al) + g(bl - kc)}{M}$$

$$t = -\frac{f(ak - jb) + e(jc - al) + d(bl - kc)}{M}$$

```
{Compute t}
if (t < t_min) or (t > t_max) then
    return false;
end if
{Compute γ}
if (γ < 0) or (γ > 1) then
    return false;
end if
{Compute β}
if (β < 0) or (β > 1 − γ) then
    return false;
end if
return true;
```

Recap - 02/13/2013

- Recap the use of the sphere intersection test
- Understand how multiple objects needs to be managed with intersections
- Modify the intersection function to assist with multiple objects
- Add the ray-triangle intersection

What you should be working on now:

- Sphere intersections need to be working
- Output png images
- Start developing triangle support

What's next?

- Start looking at XML parsing example...

Ray Tracing Algorithm

- Using a ray per pixel, find the object closest to the image plane
- Must check all objects
- Next, color the pixel based *shade* of object at point of intersection

im $\leftarrow$ new image($n_x n_y * 3$)
for all $p_{i,j} \in$ im **do**
  {*Compute Viewing Ray*}
  $u = l + (r - l)(i + 0.5)/n_x$
  $v = b + (t - b)(j + 0.5)/n_y$
  Ray r;
  r.direction $= -d\vec{W} + u\vec{U} + v\vec{V}$**;**
  r.origin $= \vec{e}$**;**
  {*Intersect with Objects*}
  for all $o_k \in$ objectList **do**
    $o_k \rightarrow$ intersect(r, t_{min}, t_{max})
  end for
  {*Set pixel color based on object shader*}
  im$[p_{i,j}] \leftarrow$ applyShader($o_{k_{closest}}$)
end for

Shading with Intersection Test

Intersection Calculations

Heart of the ray tracer's computation: *intersection*

- Bulk of operations in your ray tracer will be intersection calculations!
- Ray tracers are not generally CPU-bound, but bandwidth-bound!
- Bandwidth bound refers to access to main memory.
- Memory access plays a huge role in performance!

One question you need to consider is how do you reduce main memory access! How??

Improving Intersection Calculations

Think about data transfers

- When you access the data for an object how will it be used?
- What additional information do you potentially get with that data access?

Improving Intersection Calculations

Think about data transfers

- When you access the data for an object how will it be used?
- What additional information do you potentially get with that data access?
- How many cache lines did you read from main memory?
- Setup your memory so that it sits nicely in caches!
- Carefully align your data to cache lines (should be 64 bytes on current Intel chips)
  - May have to tradeoff memory use to get the correct alignment...
- Arranging data and locality of data
  - Keep data near each other if you need to use it that way
- Keep rays coherent!

Improving Intersection Calculations

Think about data transfers

- When you access the data for an object how will it be used?
- What additional information do you potentially get with that data access?
- How many cache lines did you read from main memory?
- Setup your memory so that it sits nicely in caches!
- Carefully align your data to cache lines (should be 64 bytes on current Intel chips)
  - May have to tradeoff memory use to get the correct alignment...
- Arranging data and locality of data
  - Keep data near each other if you need to use it that way
- Keep rays coherent!
  - Use ray packets to bundle rays together and perform similar intersections!

Next time...

In the following classes we'll get into shading parameters...

- Hit structures and ray payload data
- Lambertian shading
- Phong, and Blinn-Phong shading

The Hit Structure

Important to know *what* was hit, *how* it was hit, and *where* it was hit!

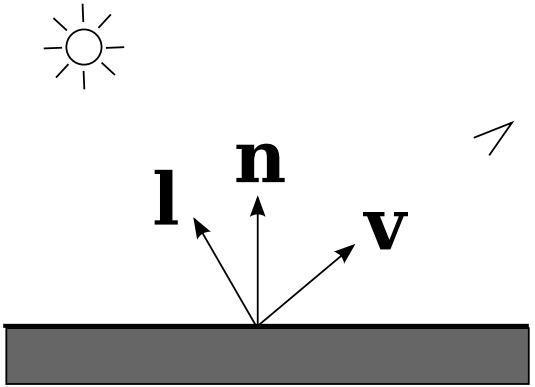
- Can be encoded in class or structure (*HitStructure*)
- Contains what you need for shading operations, some ideas follow:
  - t
  - Normal vector at point of intersection
  - Pointer to object that was intersected
  - Pointer to shader that should be used

```
im ← new image( n_x n_y * 3 )
for all p_{i,j} ∈ im do
    {Compute Viewing Ray}
    u = l + (r − l)(i + 0.5)/n_x
    v = b + (t − b)(j + 0.5)/n_y
    Ray r;
    r.direction = −dW + uU + vV;
    r.origin = e;
    {Intersect with Objects}
    HitStructure h;
    for all o_k ∈ objectList do
        o_k → intersect( r, t_min, t_max, &h )
    end for
    {Set pixel color based on object shader}
    im[p_{i,j}] ← h → applyShader(o_{k_closest})
end for
```

Important Shading Information

What information is necessary and important for shading

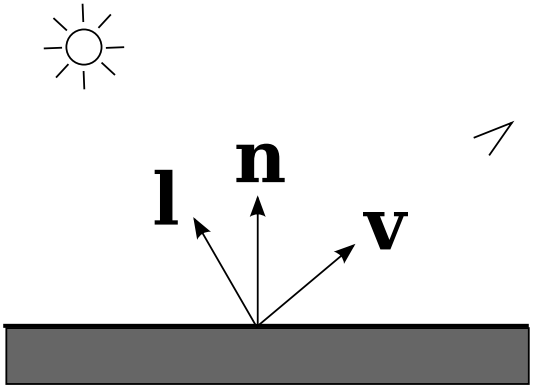
- light direction ($\vec{l}$) - unit vector from point of intersection towards the light
- view direction ($\vec{v}$) - unit vector from point of intersection towards the eye or camera
- surface normal ($\vec{n}$) - unit vector perpendicular to surface at point of intersection
- surface material information - various parameters that describe the properties of the surface materials



Surface Normal

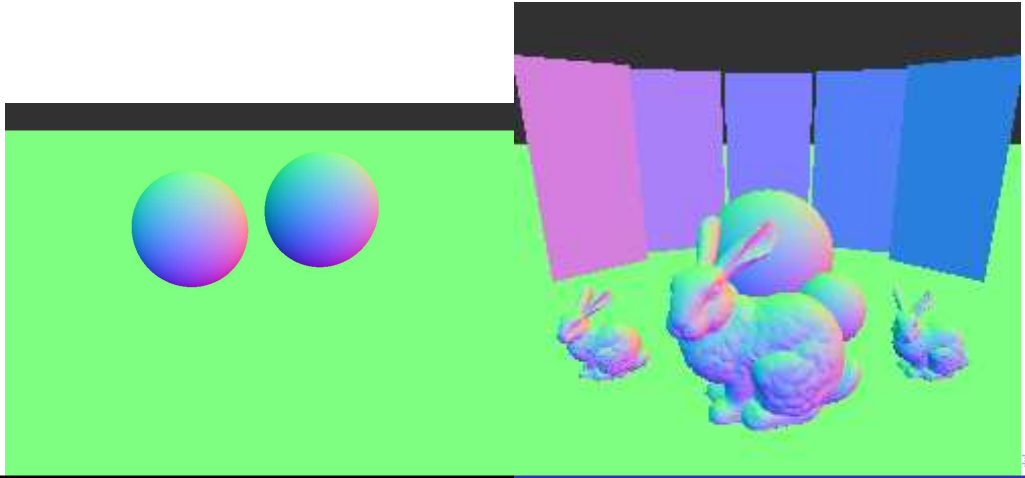
Start with computation of surface normal, $\vec{n}$

- Sphere - unit vector in direction from center to point of intersection
- Triangle - cross two edges of the triangle, then normalize
  - Be careful! Winding (or direction in which you specify vertices) must be consistent!
  - $\vec{n_1} = (\vec{b} - \vec{a}) X (\vec{c} - \vec{a})$
  - $\vec{n_2} = (\vec{c} - \vec{a}) X (\vec{b} - \vec{a})$
  - $\vec{n_1} \neq \vec{n_2}$



Shading - Normal Map

What can you do with normal information?



Shading - Compute the Normal Map

Normal map

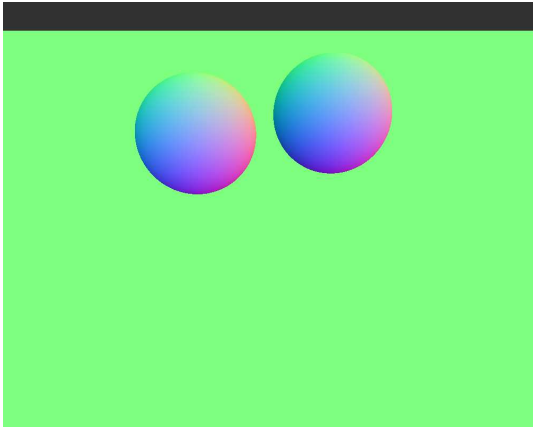
- Visualization of the surface normals as colors
- Good debuggging tool if your shading doesn't look right - check normals!

$$\vec{n} \in [-1, 1]^3$$

$$RGB \in [0, 1]^3$$

$$\vec{n_c} = \vec{n} * 0.5 + 0.5$$

- What colors would you expect to see for certain directions? Are these colors saturated?



A simplified take on the Rendering Equation

Recall that we are really dealing with radiance (or light energy). A simplified look:

$$L(x, \vec{\omega}) = L_e(x, \vec{\omega}) + L_r(x, \vec{\omega})$$

where L_e is the emitted radiance (think lights!) and L_r is reflected radiance. Now, expanded for L_r

$$L_r(x, \vec{\omega}) = \int_{all \vec{\omega}} f_r(\vec{\omega'}, \vec{\omega}) L_i(x, \vec{\omega'}) \cos\theta d\vec{\omega'}$$

What does all this mean???

A simplified take on the Rendering Equation

$$L_r(x, \vec{\omega}) = \int_{all\vec{\omega}} f_r(\vec{\omega'}, \vec{\omega}) L_i(x, \vec{\omega'}) \cos\theta d\vec{\omega'}$$

Will eventually learn, but for now:

- $f_r(\vec{\omega'}, \vec{\omega})$ is known as the BRDF, Bidirectional Reflectance Distribution Function, or in other words, how light energy reflects off our materials and surfaces!
- $L_i(x, \vec{\omega'})$ is the Incident Radiance. For us, this will initially be the radiance received by the light sources in our scene.
- $\cos\theta$ tells us how much energy a specific area receives

Keep this information in the back of your head. We'll return to it at different times.

Shading - Lambertian

Simplest model - based on Lambert's Law

- Amount of energy falling on surface is proportional to the angle of the surface to the light

Think about the cosine function and the dot product!

- $cos(0) = 1$, $cos(90) = 0$, $etc\ldots$

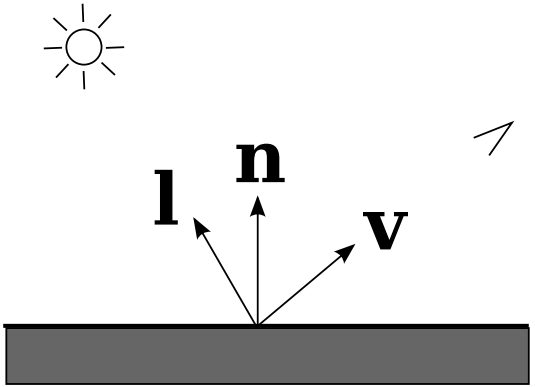
$$L = f_r(\vec{\omega'}, \vec{\omega}) L_i(x, \vec{\omega'}) \cos\theta$$

but we can simplify:

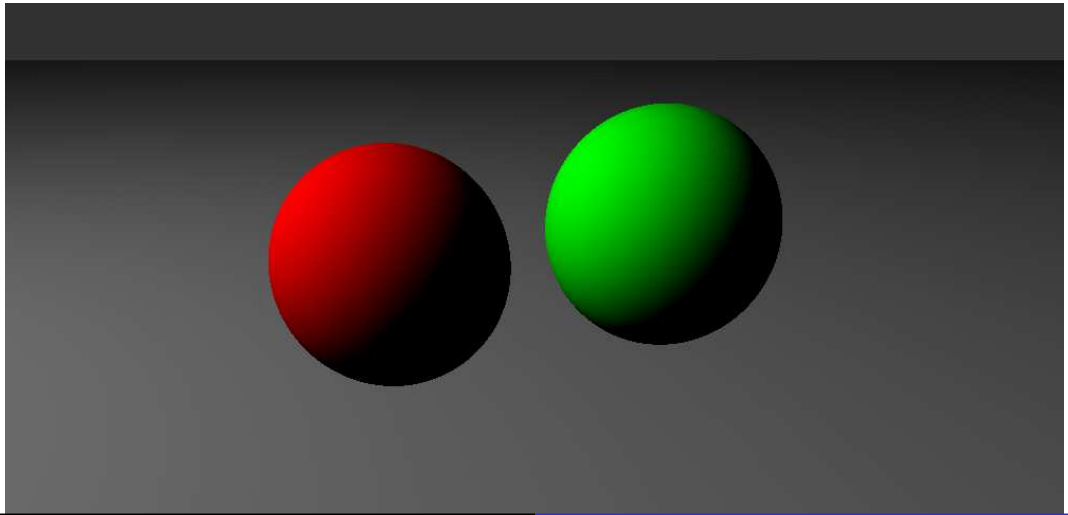
$$L = k_d I_{light} \cos\theta$$

$$L = k_d I_{light} \max(0, \vec{n} \cdot \vec{light_{dir}})$$

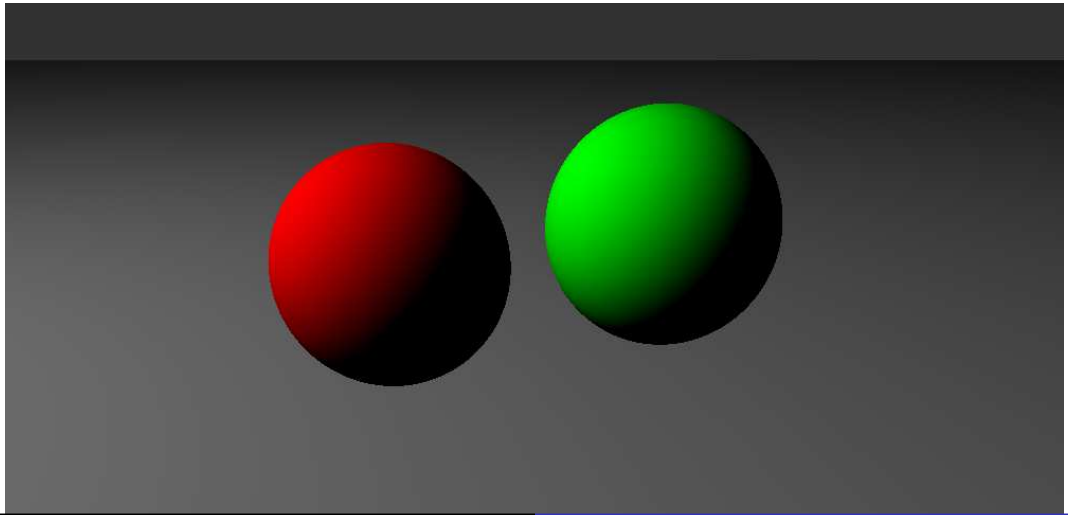
- L is the intensity of the energy or light falling on the surface
- k_d is the diffuse reflectance of the surface
- I_{light} is the intensity of the light source



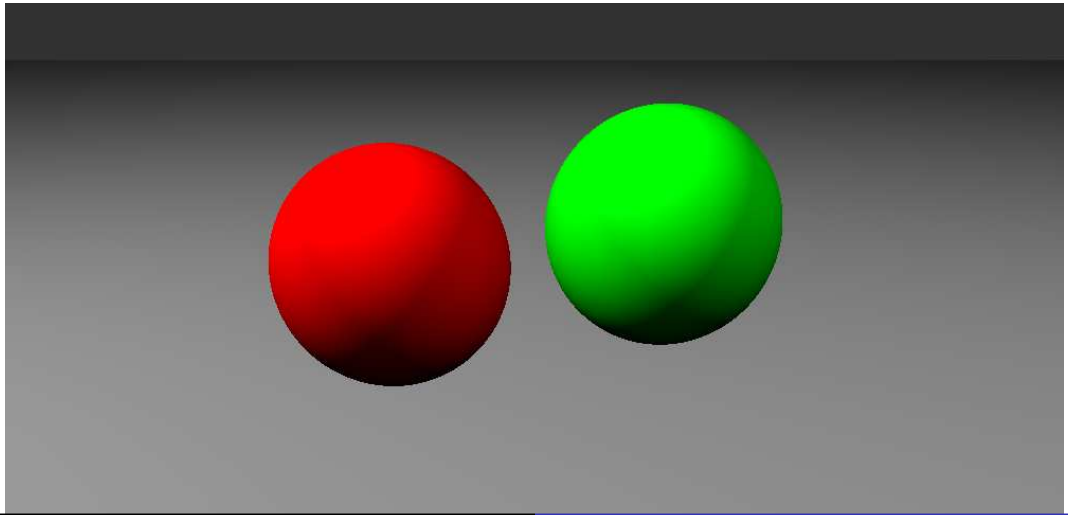
Shading - Lambertian



Shading - One Light



Shading - Multiple Lights

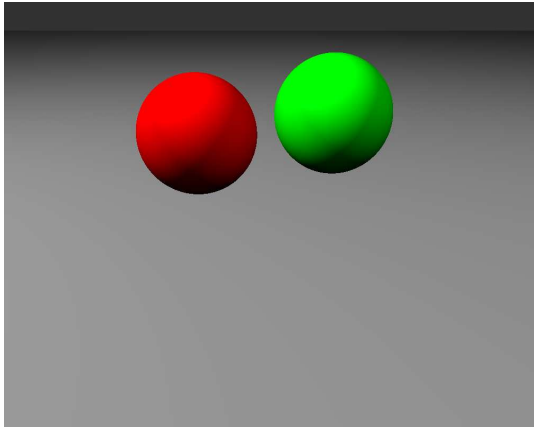


Shading - All the Lights

Shading model must support multiple lights

- Simple in that all shading values from each light are summed together
- Easy to get values greater than 1, so intensities may need to be tweaked
- You may need to clamp values within $[0, 1]$ range!

$$L = \sum_{i=1}^{N} k_d I_{light_i} \max(0, \vec{n} \cdot \vec{light}_{i_{dir}})$$

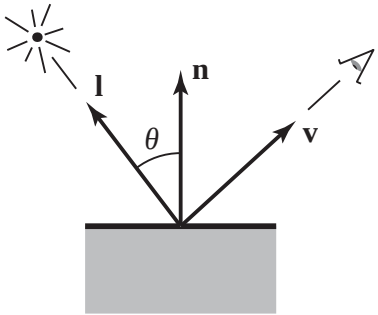


Shading - Specularity

- Not all surfaces are matte (Lambertian)!
- Many have specular highlights or reflections
  - What is a specular highlight?

Shading - Specularity

- Not all surfaces are matte (Lambertian)!
- Many have specular highlights or reflections
  - What is a specular highlight?
  - Your eye observing the reflected light on the object!



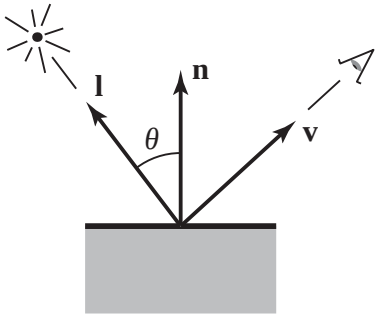
- Simple specular shading model created by Phong (1975) and updated by Blinn (1976)
- Still commonly used!
- *Basic idea*: reflection of light source is brightest when $\vec{l}$ and $\vec{v}$ are symmetric about $\vec{n}$, and hence, the reflection vector $\vec{r}$ coincides with $\vec{v}$

Shading - Phong

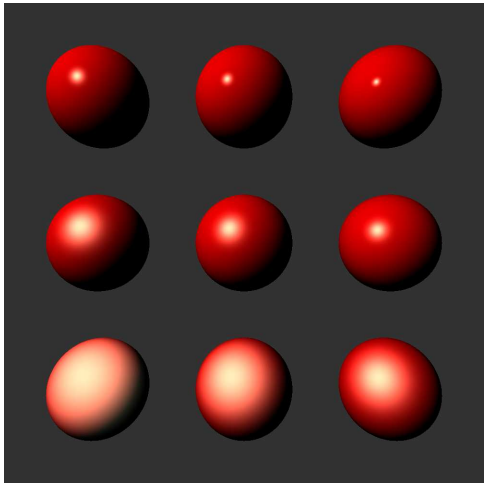
- $\vec{r}$ is reflection of the light from $\vec{l}$ towards $\vec{v}$
- When $\vec{r}$ is symmetric with $\vec{v}$, about $\vec{n}$ we have bright specular highlight

- Phong exponent, p, describes apparent shininess of surface (larger value, more *shininess*)
- k_s is the specular energy of the surface (how surface reflects light)

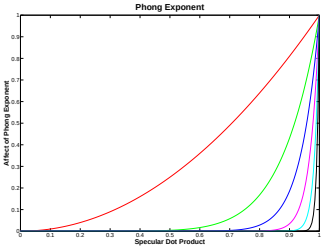
$$L = k_d I \max(0, \vec{n} \cdot \vec{l}) + k_s I \max(0, \vec{v} \cdot \vec{r})^p$$

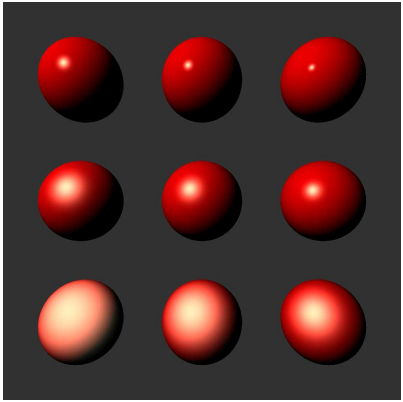


Phong Exponent



Phong Exponent

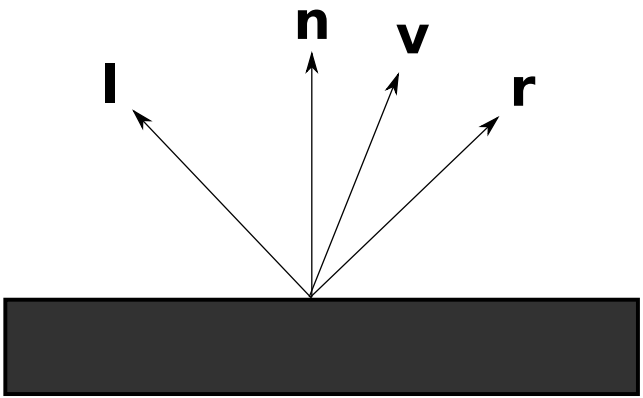




Values relate to shininess:

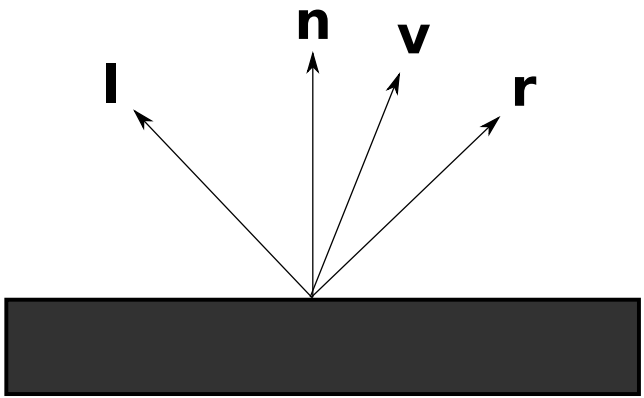
- $p = 10$ - Eggshell
- $p = 100$ - Mildly shiny
- $p = 1000$ - Really glossy

Compute the Reflection Ray



How do you compute $\vec{reflect}_{dir}$ for specular hi-lights?

Compute the Reflection Ray



How do you compute $\vec{reflect}_{dir}$ for specular hi-lights?

$$\vec{reflect}_{dir} = -\vec{l} + 2(\vec{l} \cdot \vec{n})\vec{n}$$

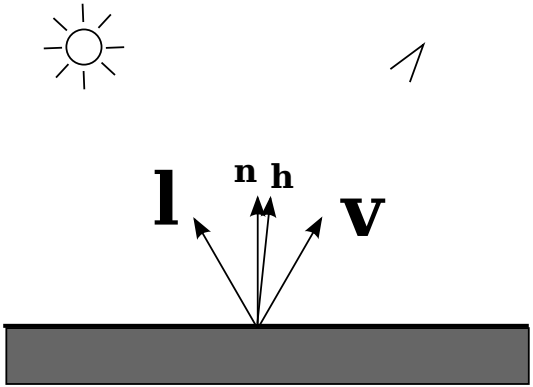
Shading - Blinn Phong Model

Observation from Jim Blinn in 1977 that different model using the half-vector more accurately represents specular hi-lights:

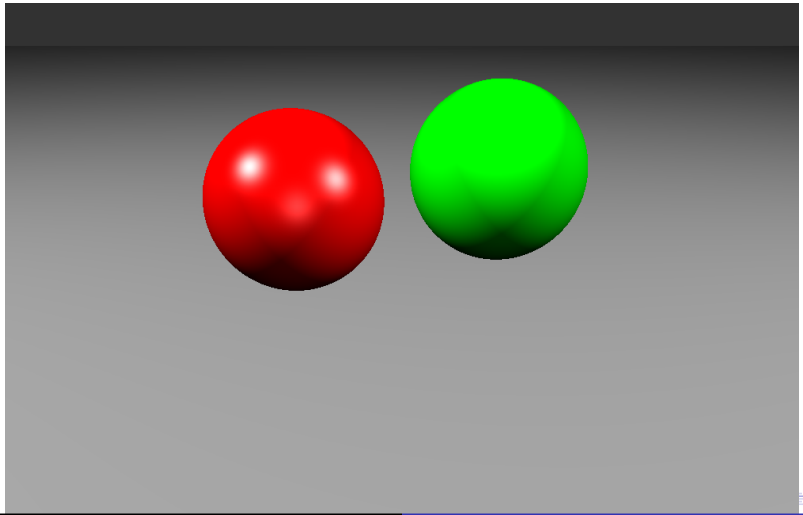
- $\vec{h}$ is bisector between $\vec{l}$ and $\vec{v}$
- When $\vec{h}$ coincident with $\vec{n}$, we have bright specular highlight

$$h = \frac{\vec{v} + \vec{l}}{\left|\left|\vec{v} + \vec{l}\right|\right|}$$

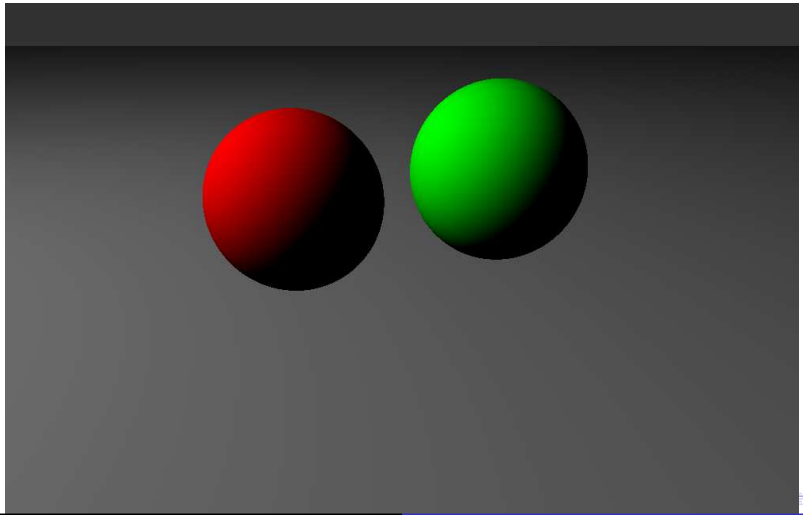
$$L = k_d I \max(0, \vec{n} \cdot \vec{l}) + k_s I \max(0, \vec{n} \cdot \vec{h})^p$$



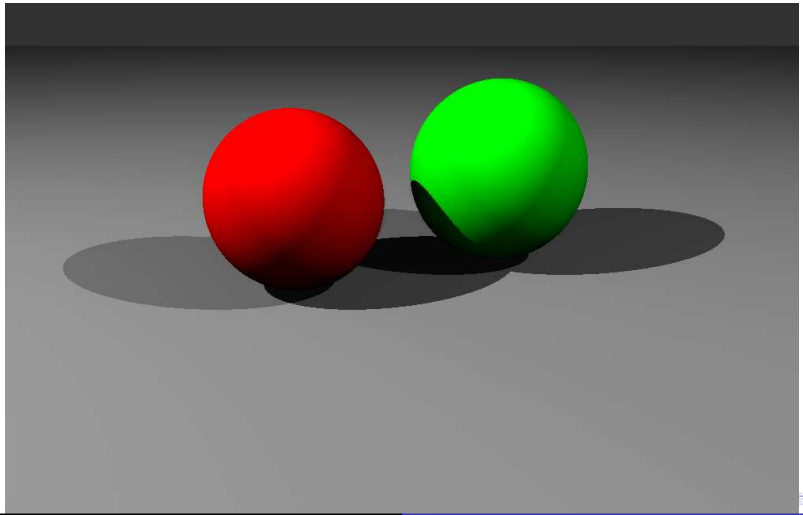
Shading - Blinn-Phong



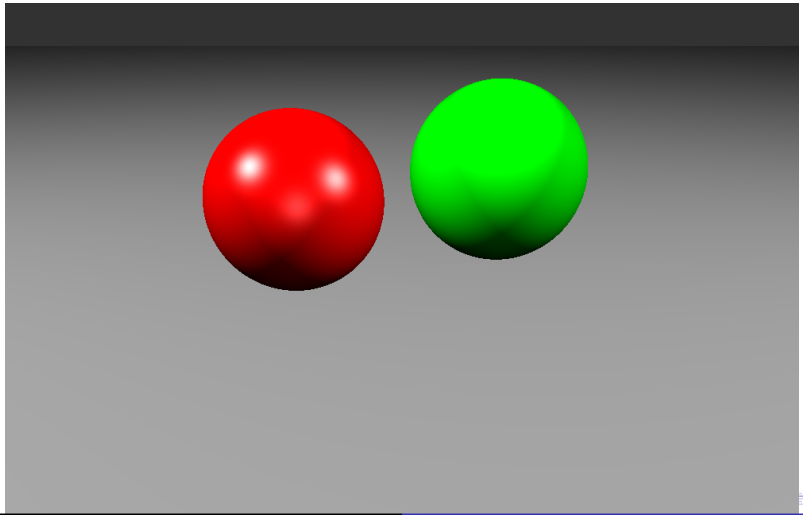
Shading - Lambertian



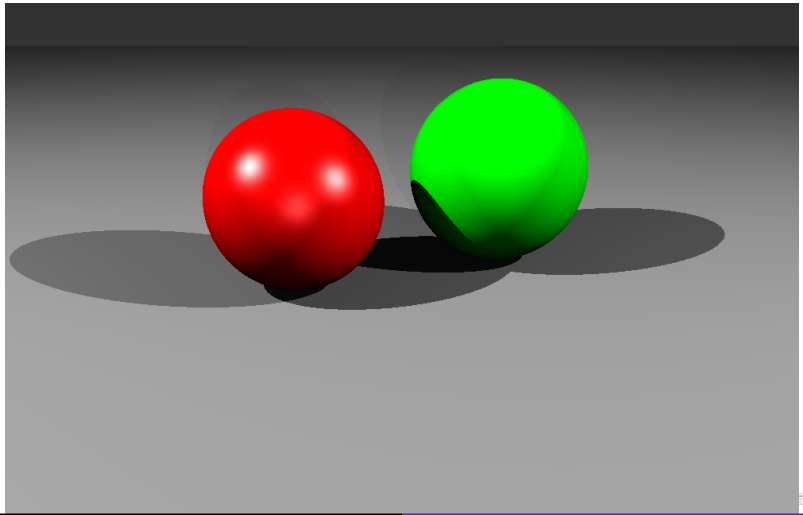
Shading - Lambertian w/ Shadows



Shading - Blinn-Phong



Shading - Blinn-Phong w/ Shadows



Shading - Shadows

Shadows add *much* to the interpretation of a scene!

- For each intersection, send a *shadow* ray towards each light
- If *shadow* ray intersects *any* object, original intersection point is in shadow
- Otherwise, original intersection point can be shaded using the object's material properties

Important to realize:

- t_{min} should NOT be 0.0! Why?

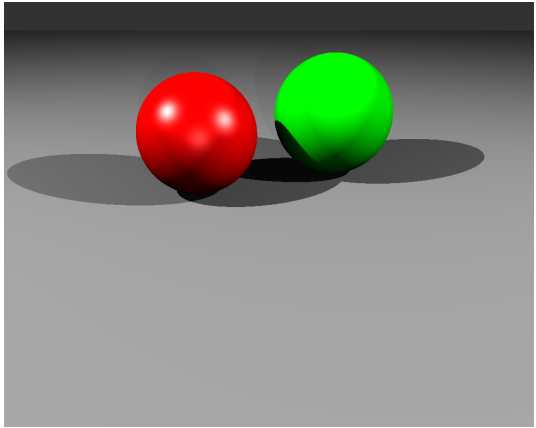
Shading - Shadows

Shadows add *much* to the interpretation of a scene!

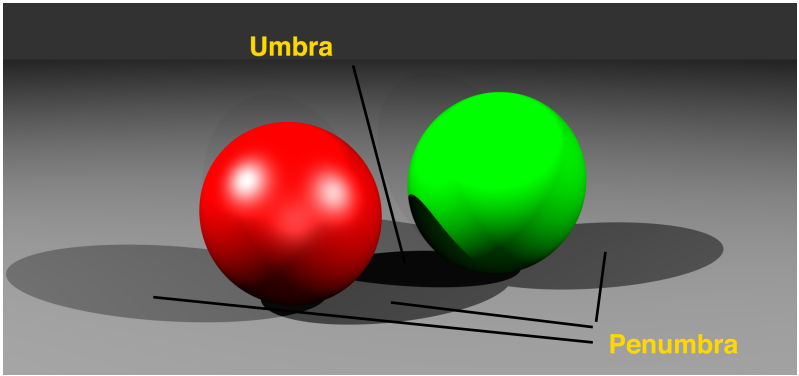
- For each intersection, send a *shadow* ray towards each light
- If *shadow* ray intersects *any* object, original intersection point is in shadow
- Otherwise, original intersection point can be shaded using the object's material properties

Important to realize:

- t_{min} should NOT be 0.0! Why?
- Numerical imprecision may result in object shadowing itself
- Thus, set $t_{min} = eps$ for some small $eps = 1.0e - 3$



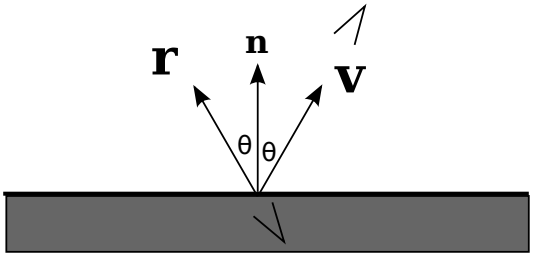
Shadows - Some Definitions



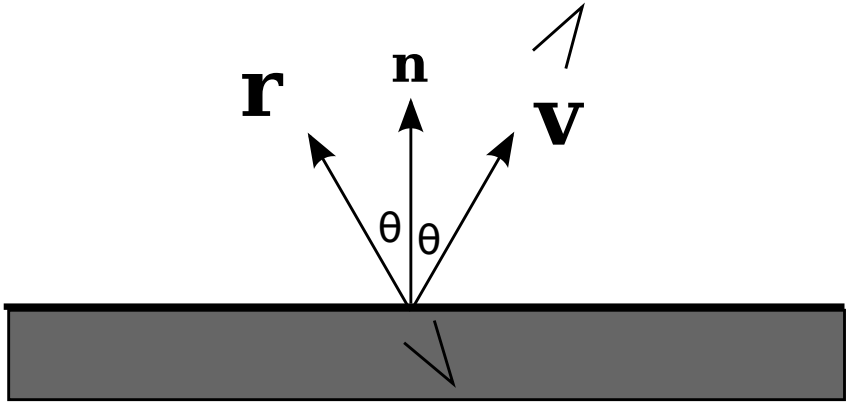
Shading - Ideal Reflections

Adding objects that reflect surroundings - ideal specular reflection

- Also known as a mirror!
- May require use of a mirroring coefficient, k_m to control the amount of mirroring.
- Requires that we calculate the *reflection* ray
- Reflection ray is sent into scene to see what is hit

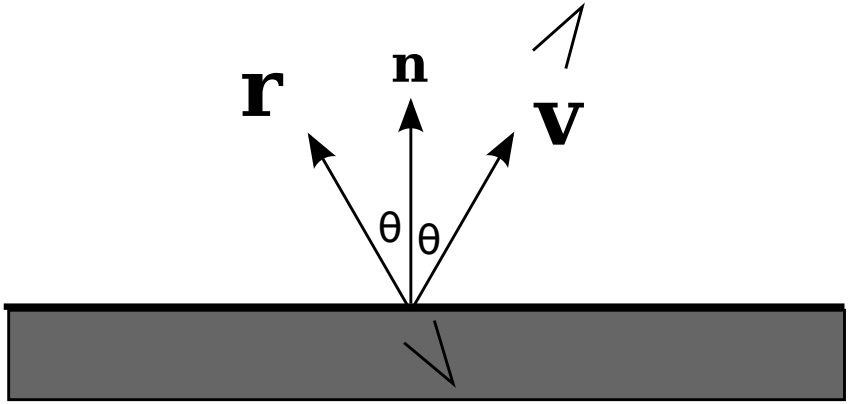


Shading - Compute the Reflection Ray



Recall how do you compute $\vec{r}$ for reflection?

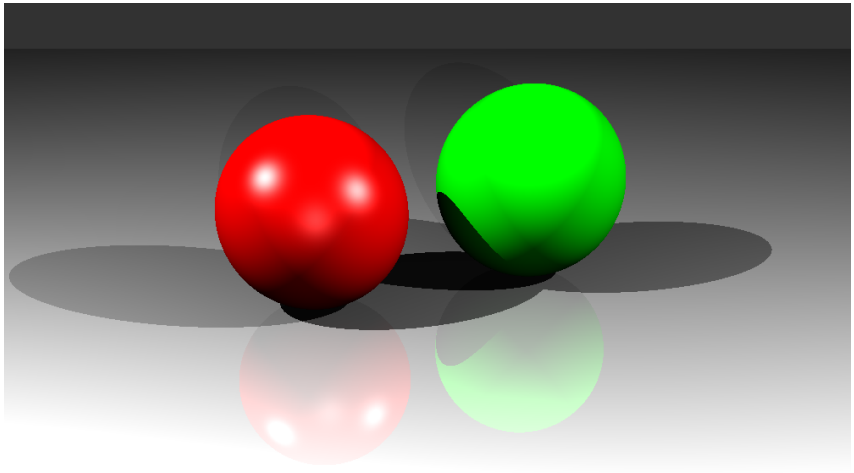
Shading - Compute the Reflection Ray



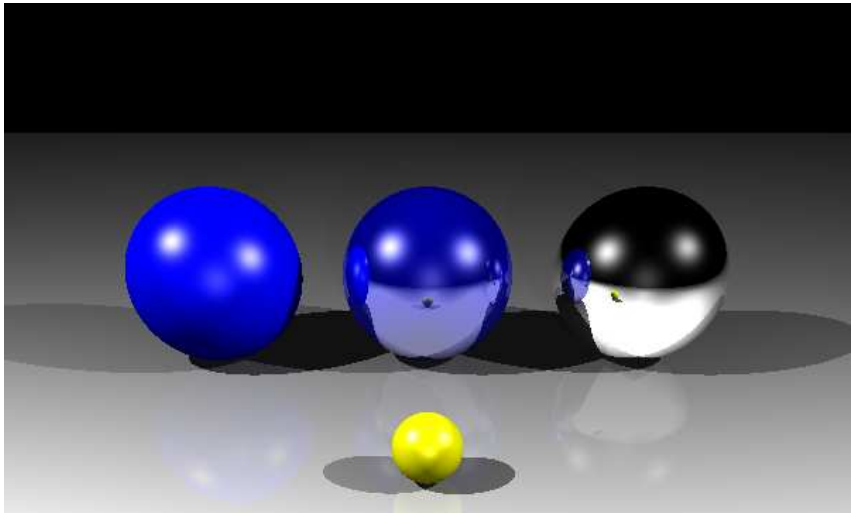
Recall how do you compute $\vec{r}$ for reflection?

$$\vec{r} = -\vec{v} + 2(\vec{v} \cdot \vec{n})\vec{n}$$

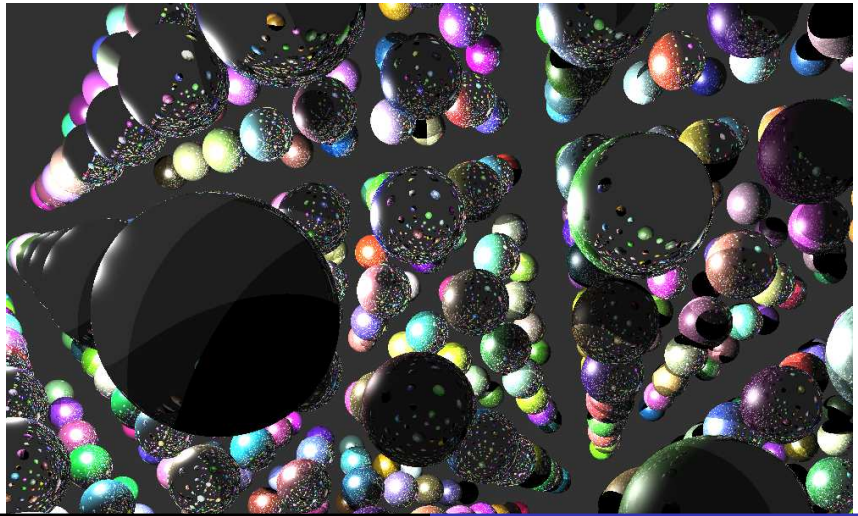
Ideal Specular Reflection



Ideal Specular Reflection



Ideal Specular Reflection



Ideal Specular Reflection

- What changes in your ray tracer for mirror reflections?
- For every intersection point, you already send out *shadow* ray
- Now, you will also send out a *reflection* ray
- What if you hit another reflective object?

- Reflected object color is used to color your original pixel
- *Recursive* ray tracer
- Make sure to clamp the recursion depth!

Basic Mirror Shader

$$c \leftarrow \text{rayColor}(\vec{p} + s\vec{r}, \epsilon, \infty, depth)$$

How do you abstract shaders?

Shader hierarchy...

- You want shaders that can generate their own rays!
- Build good abstractions to access data in your classes, so that shaders can be added without affecting the main ray tracer loop.

Different Shader Ideas

With the introduction of ideal specular reflection, we have some choices:

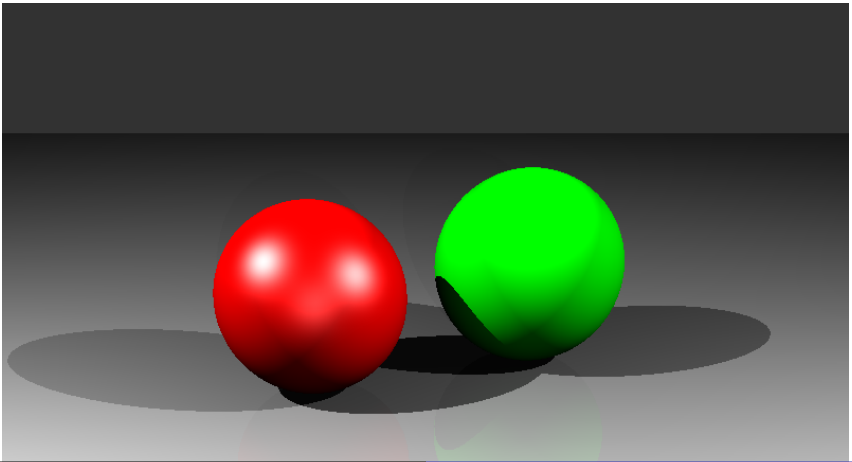
- How do you combine reflected surface color?

You could...

- Have an ideally specular surface - a mirror!
- Mix the shader data from Blinn-Phong with the reflected colors
- Mix the shader data from Lambertian with the reflected colors

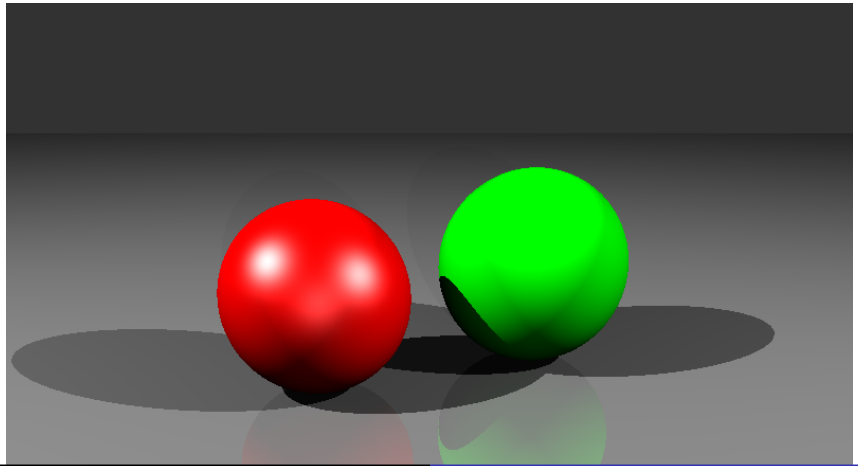
BlinnPhong with Ideal Specular Reflection

$$c \leftarrow (1.0 - k_m)c_{BlinnPhong} + k_m c_{Mirror}$$



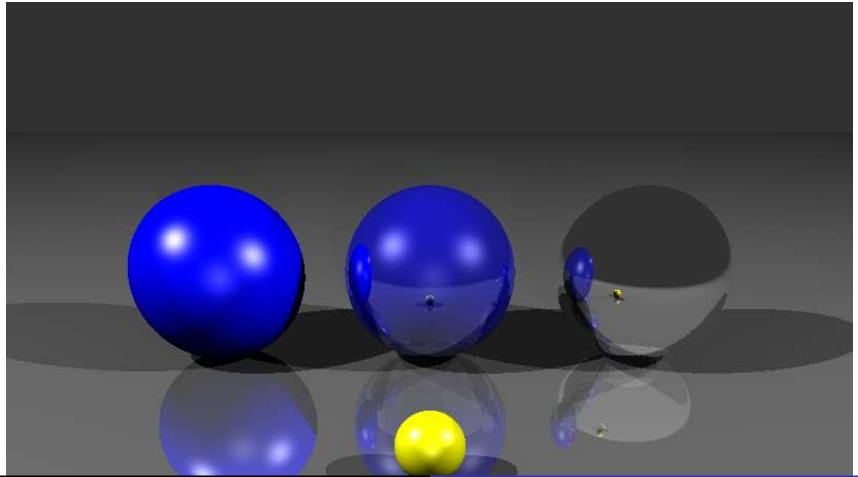
Lambertian with Ideal Specular Reflection - Ceramic Glaze

$$c \leftarrow c_{Lambertian} + k_m c_{Mirror}$$



BlinnPhong with Ideal Specular Reflection

Varying the mirror coefficient from 0.0 to 1.0



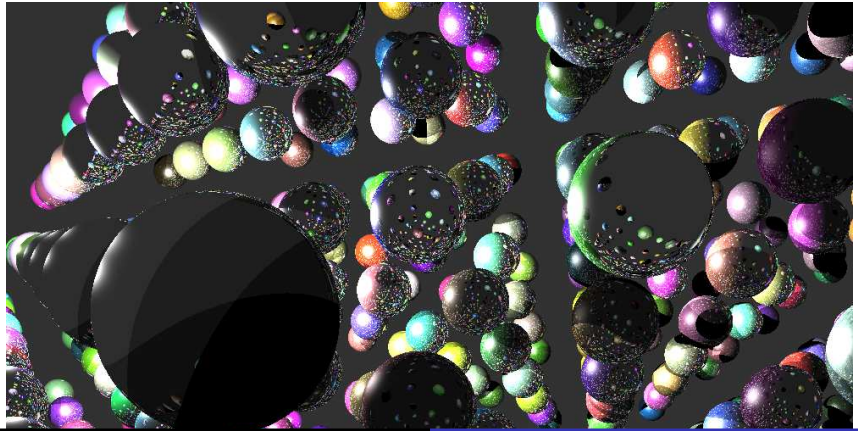
Shader Support

Your code should support the following shader types:

- Lambertian - diffuse shaders that support Lambert's law
- BlinnPhong - supports the diffuse and specular lighting components, but specifying an optional mirrorCoef turns on ideally reflective Blinn-Phong shading
- Glaze - combines the diffuse component directly with the reflective mirroring components
- BlinnPhongMirrored - combines the BlinnPhong shader with ideal specular reflection

What's next?

- That's about it for the first programming assignment...
- What questions do you have?



Ray Hierarchy

So far, three types of rays:

- Viewing ray - what we've mainly dealt with so far
- Reflection Ray - determines ideal specular reflection components
- Shadow Ray - helps determines the amount of light intensity applied to a surface

Recap + Exam 1 Review

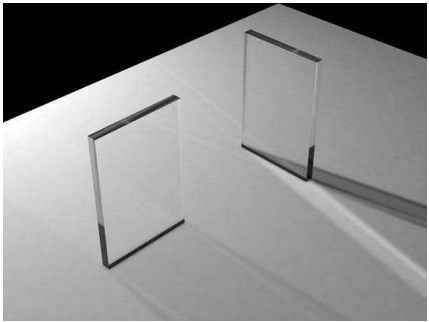
Exam 1 will be held on Tuesday, Feb 21 ???

Refraction Rays and Dielectric Materials

Refraction

- Transparent material that refracts light (diamonds, glass, water, etc...)
- Also filters light, glass is a good example
- As light changes refractive index, it bends, based on Snell's law

$$n\sin\theta = n_t\sin\phi$$



Refraction Rays and Dielectric Materials

Simple Refraction

$t$is refraction ray

- Transparent material that refracts light (diamonds, glass, water, etc...)
- Also filters light, glass is a good example
- As light changes refractive index, it bends, based on Snell's law

$$n\sin\theta = n_t\sin\phi$$