CSC 433/533 Computer Graphics Review 2

Alon Efrat
Credit: Joshua Levine

Vector Math + Coding

Today's Agenda

- Reminders:
  - A02 questions?
- Goals for today:
  - Introduce some mathematics and connect it to code

Vectors

What is a Vector?

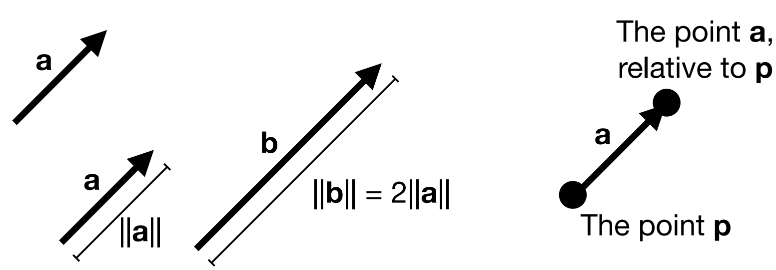
- A **vector** describes a length and a direction
- A vector is also a tuple of numbers
  - But, it often makes more sense to think in terms of the length/direction than the coordinates/numbers
  - And, especially in code, we want to manipulate vectors as objects and abstract the low-level operations
  - Compare with a **scalar**, or just a single number

Properties

- Two vectors, **a** and **b**, are the same (written **a** = **b**) if they have the same length and direction. (other notation: $\bar{a}$, $\overrightarrow{a}$)
- A vector's **length** is denoted with || ||, (sometimes we just denote . When **a** =(x,y), then $|\mathbf{a}| = \sqrt{a.x^2 + a.y^2}$
  - e.g. the length of **a** is ||**a**||
- A **unit vector** has length one
- The **zero vector** has length zero, and undefined direction

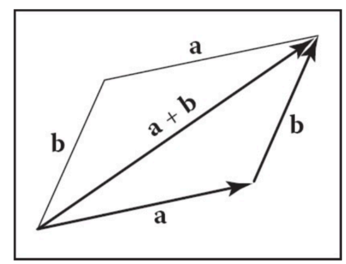
Vectors in Pictures

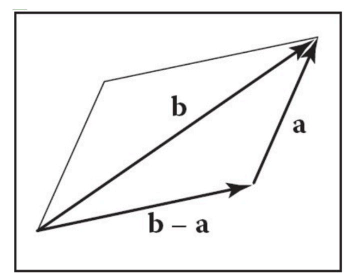
- We often use an arrow to represent a vector
  - The length of the arrow indicates the length of the vector, the direction of the arrow indicates the direction of the vector.
- The position of the arrow is irrelevant!
  - However, we can use vectors to represent positions by describing displacements from a common point



Vector Operations

- Vectors can be added, e.g. for vectors **a**,**b**, there exists a vector **c** = **a**+**b**
  $\mathbf{a} + \mathbf{b} = (a.x + b.x,\ \ a.y + b.y)$
- Defined using the parallelogram rule: idea is to trace out the displacements and produced the combined effect
- Vectors can be negated (flip tail and head), and thus can be subtracted
- Vectors can be multiplied by a scalar, which scales the length but not the direction
  $\beta\mathbf{a} = (\beta a.x,\ \ \beta a.y)$

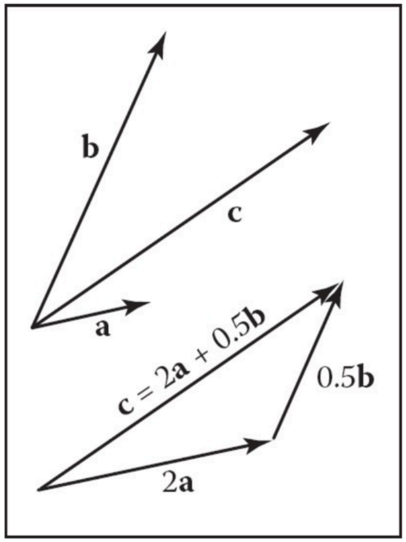




Vectors Decomposition

- By linear independence, any 2D vector can be written as a combination of any two nonzero, nonparallel vectors
- Such a pair of vectors is called a **2D basis**

$$\mathbf{c} = a_c\mathbf{a} + b_c\mathbf{b}$$



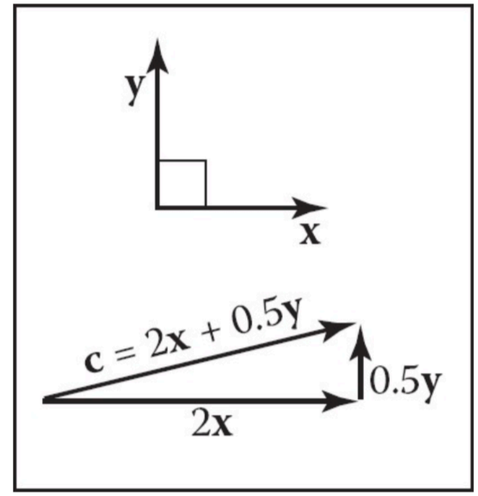
Canonical (Cartesian) Basis

- Often, we pick two perpendicular vectors, **x** and **y**, to define a common **basis**
- Notationally the same,

$$\mathbf{a} = x_a\mathbf{x} + y_a\mathbf{y}$$

- But we often don't bother to mention the basis vectors, and write the vector as **a** = (x_a,y_a), or

$$\mathbf{a} = \begin{bmatrix} x_a \\ y_a \end{bmatrix}$$



Vector Multiplication: Dot Products

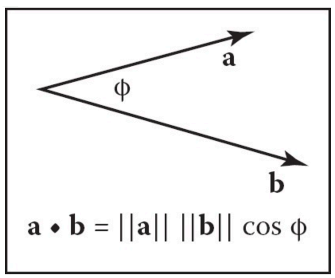
- Given two vectors **a** and **b**, the **dot product**, relates the lengths of **a** and **b** with the angle ϕ between them:

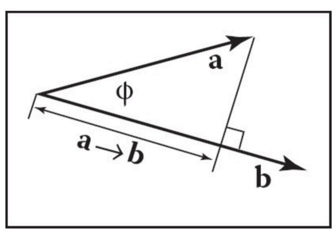
$$\mathbf{a} \cdot \mathbf{b} = (a.x \cdot b.x \ + \ a.y \cdot b.y)$$

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos \phi$$

  - Sometimes called the scalar product, as it produces a scalar value
- Also can be used to produce the **projection**, **a**→**b**, of **a** onto **b**

$$\mathbf{a} \rightarrow \mathbf{b} = \|\mathbf{a}\| \ \cos\phi = \frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{b}\|}$$





Dot Products are Associative and Distributive

$$\mathbf{a}\cdot\mathbf{b} = \mathbf{b}\cdot\mathbf{a},$$
$$\mathbf{a}\cdot(\mathbf{b}+\mathbf{c}) = \mathbf{a}\cdot\mathbf{b} + \mathbf{a}\cdot\mathbf{c},$$
$$(k\mathbf{a})\cdot\mathbf{b} = \mathbf{a}\cdot(k\mathbf{b}) = k\mathbf{a}\cdot\mathbf{b}$$

- And, we can also define them directly if **a** and **b** are expressed in Cartesian coordinates:

$$\mathbf{a}\cdot\mathbf{b} = x_a x_b + y_a y_b$$

3D Vectors

- Same idea as 2D, except these vectors are defined typically with a basis of three vectors
  - Still just a direction and a magnitude
  - But, useful for describing objects in three-dimensional space
- Most operations exactly the same, e.g. dot products:

$$\mathbf{a} \cdot \mathbf{b} = x_a x_b + y_a y_b + z_a z_b$$

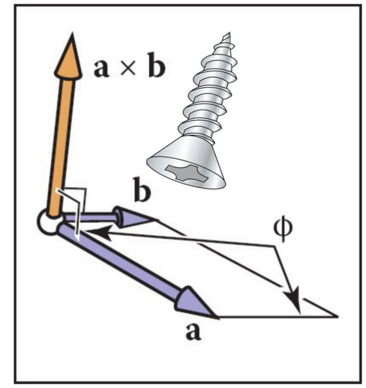
Cross Products

- In 3D, another way to “multiply” two vectors is the **cross product**, **a** × **b**:

  $||\mathbf{a} \times \mathbf{b}|| = ||\mathbf{a}||\ ||\mathbf{b}|| \sin \phi$

- $||\mathbf{a} \times \mathbf{b}||$ is always the area of the parallelogram formed by **a** and **b**, and **a** × **b** is always in the direction perpendicular (two possible answers).
- A screw turned from **a** to **b** will progress in the direction **a** × **b**
- Cross products distribute, but order matters:

$$\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c}$$
$$\mathbf{a} \times (k\mathbf{b}) = k(\mathbf{a} \times \mathbf{b})$$
$$\mathbf{a} \times \mathbf{b} = -(\mathbf{b} \times \mathbf{a})$$



Cross Products

- Since the cross product is always orthogonal to the pair of vectors, we can define our 3D Cartesian coordinate space with it:

$$\begin{aligned} \mathbf{x} &= (1,0,0) \\ \mathbf{y} &= (0,1,0) \\ \mathbf{z} &= (0,0,1) \end{aligned} \qquad \begin{aligned} \mathbf{x} \times \mathbf{y} &= +\mathbf{z}, \\ \mathbf{y} \times \mathbf{x} &= -\mathbf{z}, \\ \mathbf{y} \times \mathbf{z} &= +\mathbf{x}, \\ \mathbf{z} \times \mathbf{y} &= -\mathbf{x}, \\ \mathbf{z} \times \mathbf{x} &= +\mathbf{y}, \\ \mathbf{x} \times \mathbf{z} &= -\mathbf{y}. \end{aligned}$$

- In practice though (and the book derives this), we use the following to compute cross products:

$$\mathbf{a} \times \mathbf{b} = (y_a z_b - z_a y_b,\ z_a x_b - x_a z_b,\ x_a y_b - y_a x_b)$$

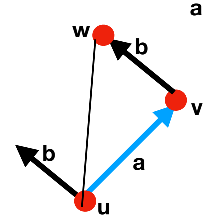
Checking orientation

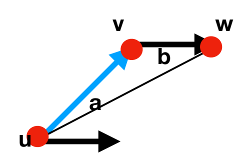
Assume **a, b are in 2D (z=0)**. There are 3 possible scenarios.

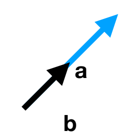
a might be counter-clockwise **(ccw)** of **b**

a might be clockwise **(cw)** of **b**

a is collinear with **b**







$x_a y_b - y_a x_b > 0$ — **a** is counter-clockwise **(ccw)** of **b**

$x_a y_b - y_a x_b < 0$ — **a** is clockwise **(cw)** of **b**

$x_a y_b - y_a x_b = 0$ — **a, b collinear**

This will provide a convenient way to check if a triangle with vertices u,v,w (when vertices are given to us in this order) is CCW or CW

Rendering

What is Rendering?

*“**Rendering** is the task of taking three-dimensional objects and producing a 2D image that shows the objects as viewed from a particular viewpoint”*

Two Ways to Think About Rendering

- Object-Ordered
- Decide, for every object in the scene, its contribution to the image

- Image-Ordered
- Decide, for every pixel in the image, its contribution from every object

Two Ways to Think About Rendering

- Object-Ordered or **Rasterization**

```
for each object {
  for each image pixel {
    if (object affects pixel)
    {
      do something
    }
  }
}
```

- Image-Ordered or **Ray Tracing**

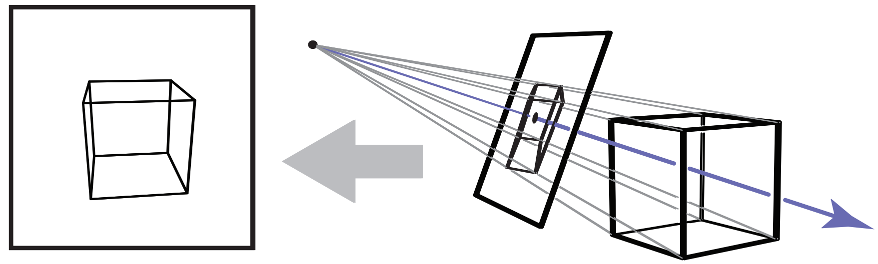
```
for each image pixel {
  for each object {
    if (object affects pixel)
    {
      do something
    }
  }
}
```

TODAY

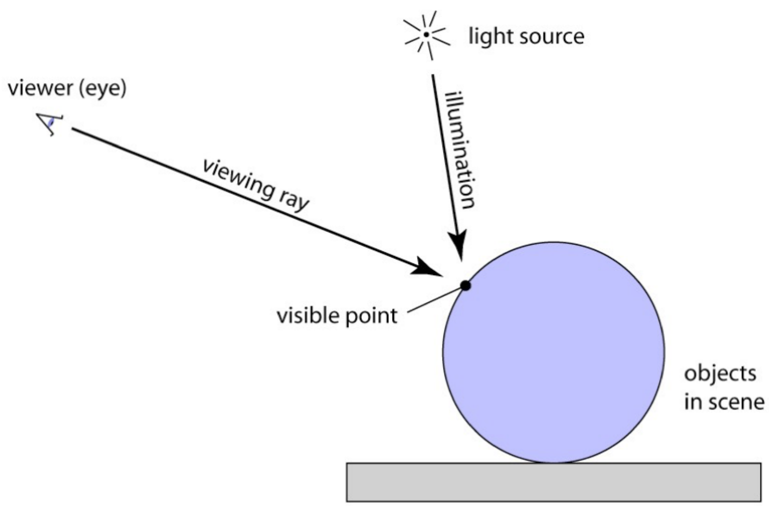
Basics of Ray Tracing

Idea of Ray Tracing

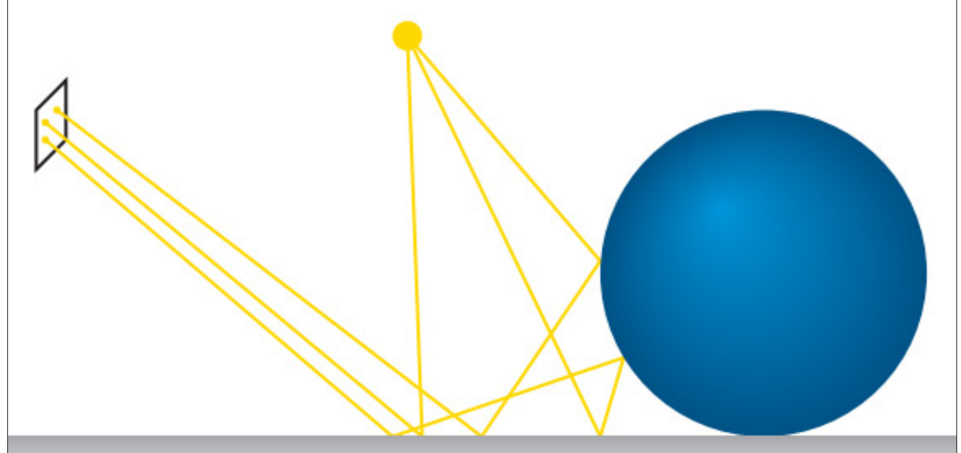
- Ask first, for each pixel: what belongs at that pixel?
- Answer: The set of objects that are visible if we were standing on one side of the image looking into the scene



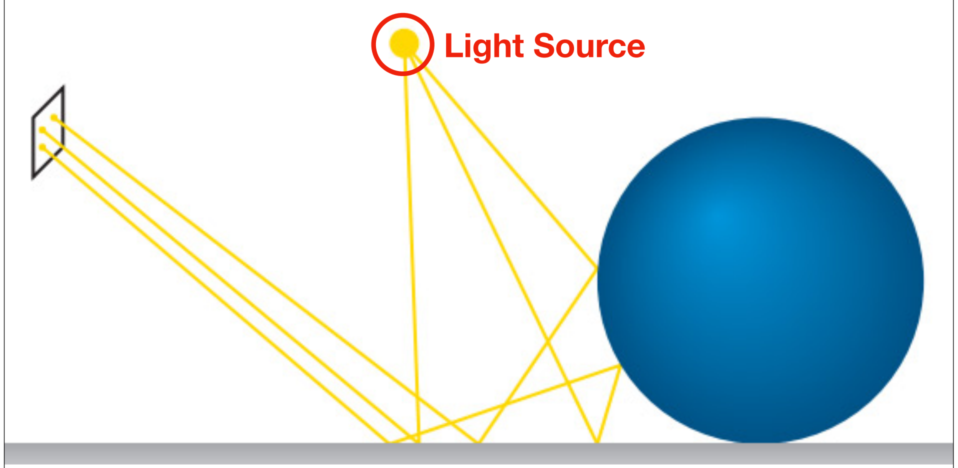
Key Concepts, in Diagram



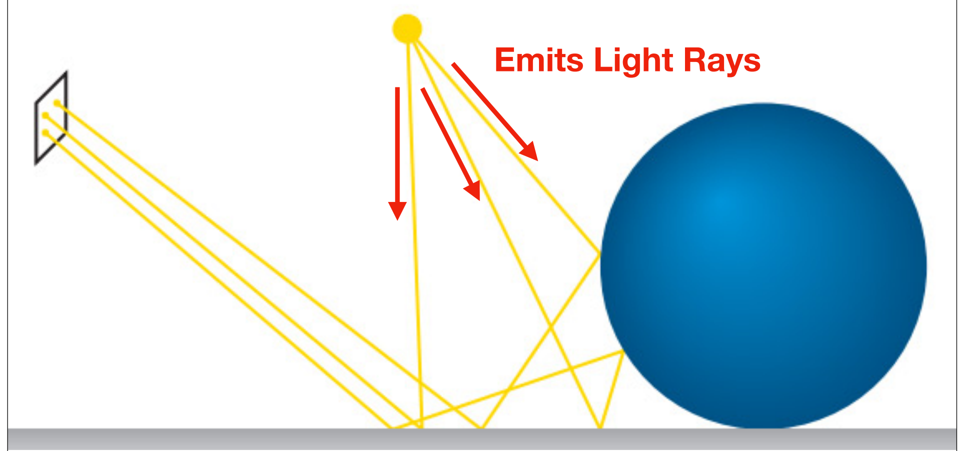
Idea: Using Paths of Light to Model Visibility



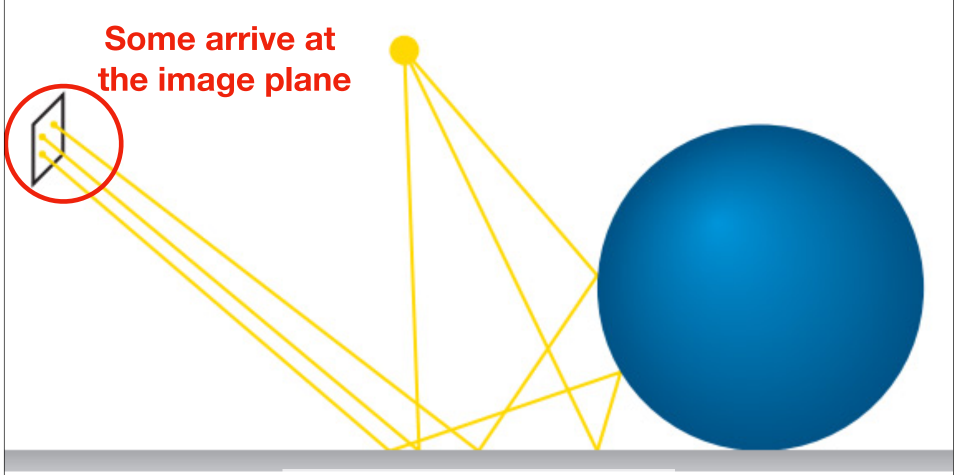
Using Paths of Light to Model Visibility



Using Paths of Light to Model Visibility



Using Paths of Light to Model Visibility



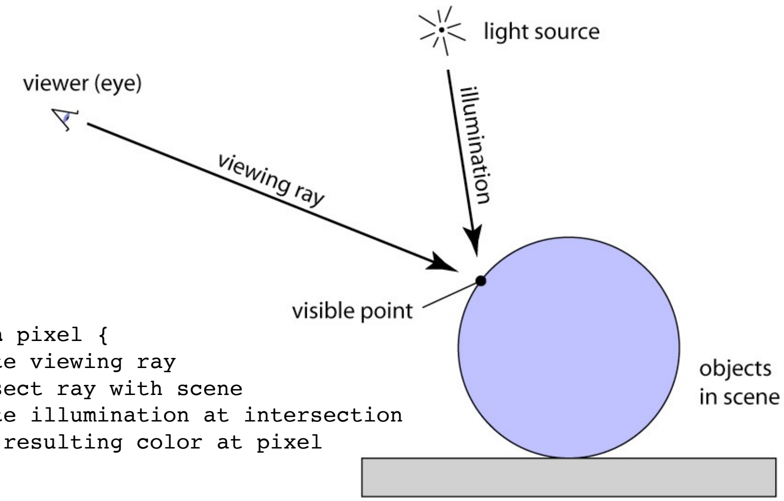
https://software.intel.com/file/37491

Using Paths of Light to Model Visibility

Forwarding vs Backward Tracing

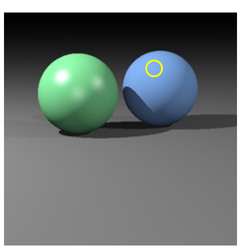
- Idea: Trace rays from light source to image
  - This is slow!
- Better idea: Trace rays **<u>from</u>** image **<u>to</u>** light source

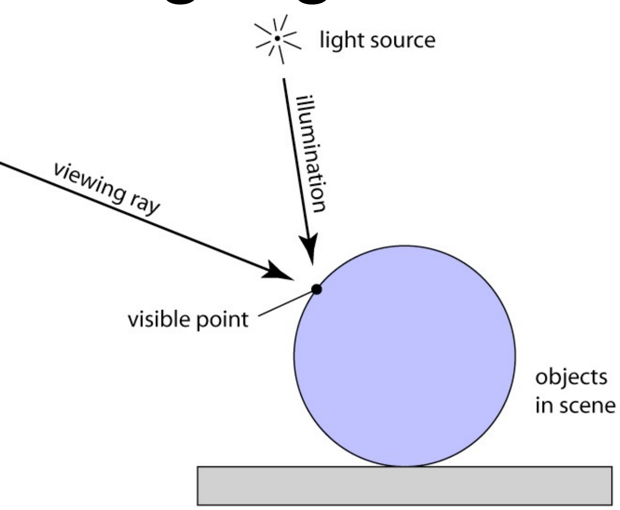
Ray Tracing Algorithm




```
for each pixel {
  compute viewing ray
  intersect ray with scene
  compute illumination at intersection
  store resulting color at pixel
}
```

Ray Tracing Algorithm





Cameras and Perspective

If illumination is uniform and directional-free (ambient light):
for each pixel {
  compute viewing ray
  intersect ray with scene
  copy the color of the object at this point to this pixel.
}

Commonly, we need slightly more involved

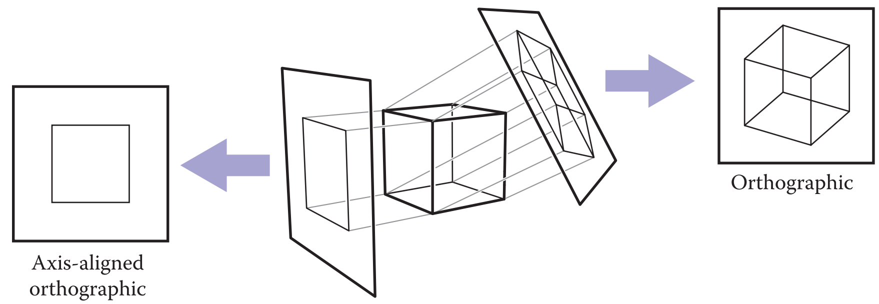
```
for each pixel {
  compute viewing ray
  intersect ray with scene
  compute illumination at intersection
  store resulting color at pixel
}
```

Linear Perspective

- Standard approach is to project objects to an image plane so that straight lines in the scene stay straight lines on the image
- Two approaches:
  - Parallel projection: Results in **orthographic** views
  - Perspective projection: Results in **perspective** views

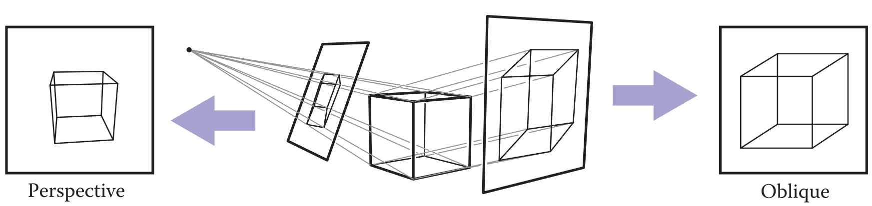
Orthographic Views

- Points in 3D are moved along parallel lines to the image plane.
- Resulting view determined solely by choice of projection direction and orientation/position of image plane



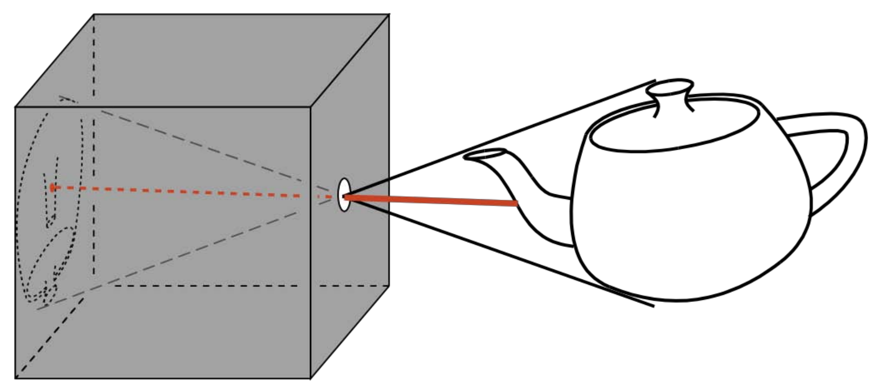
Perspective Views

- But, objects that are further away should look smaller!
- Instead, we can project objects through a single viewpoint and record where they hit the plane.
- Lines which are paper in 3D might be non-parallel in the view



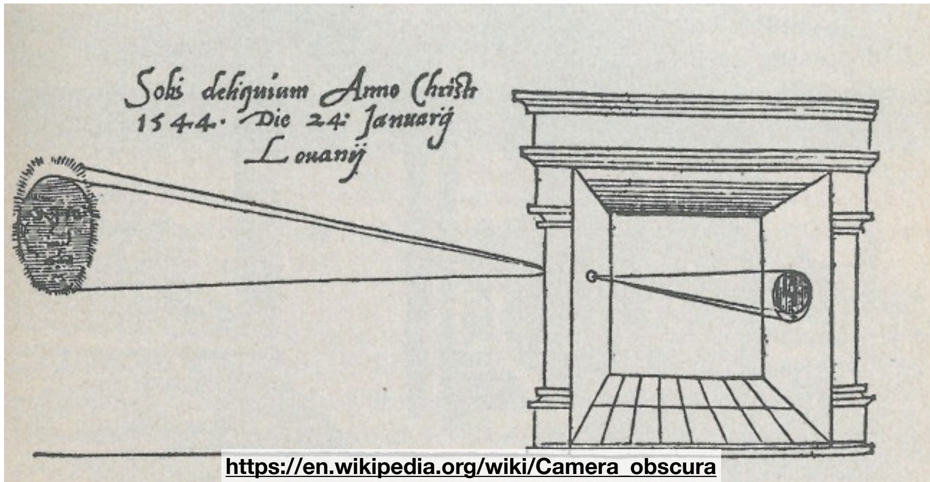
Pinhole Cameras

- Idea: Consider a box with a tiny hole. All light that passes through this hole will hit the opposite side
- Produced image inverts



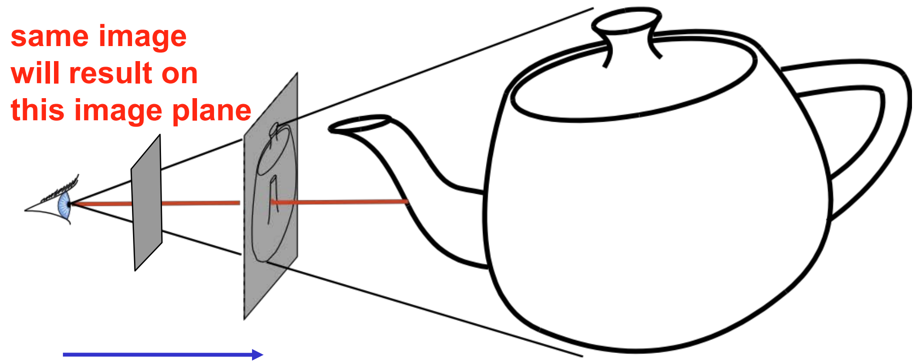
Camera Obscura

- Gemma Frisius, 16th century



Simplified Pinhole Cameras

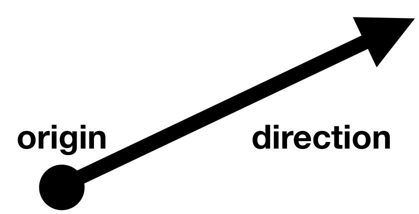
- Instead, we can place the eye at the pinhole and consider the eye-image pyramid (sometimes called **view frustum**)



Defining Rays

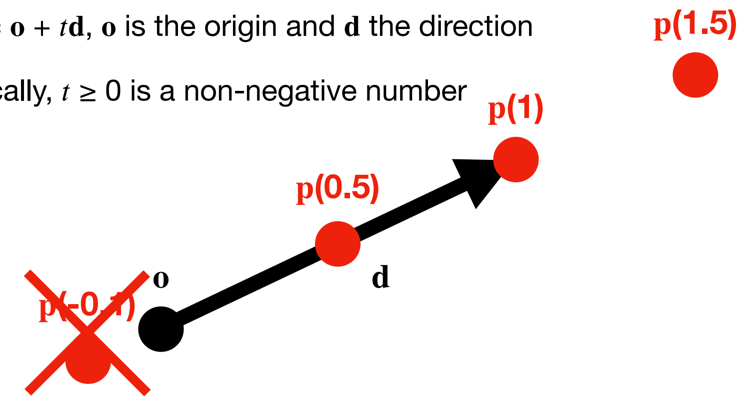
Mathematical Description of a Ray

- Two components:
  - An **origin**, or a position that the ray starts from
  - A **direction**, or a vector pointing in the direction the ray travels
    - Not necessarily unit length, but it's sometimes helpful to think of these as normalized

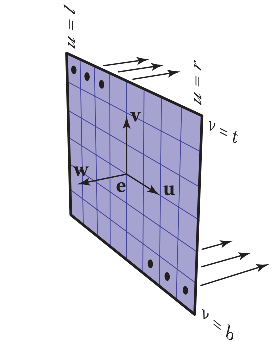


Mathematical Description of a Ray

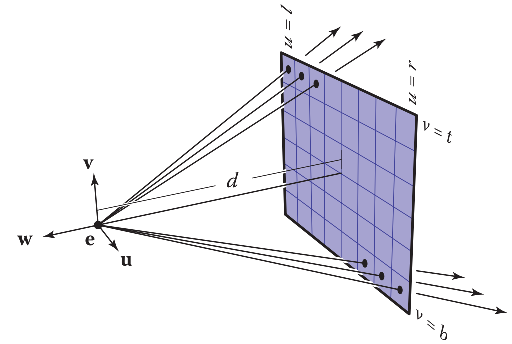
- Rays define a family of points, $\mathbf{p}(t)$, using a **parametric** definition
- $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$, $\mathbf{o}$ is the origin and $\mathbf{d}$ the direction
- Typically, $t \geq 0$ is a non-negative number



Orthographic vs. Perspective Rays



Parallel projection
same direction, different origins



Perspective projection
same origin, different directions

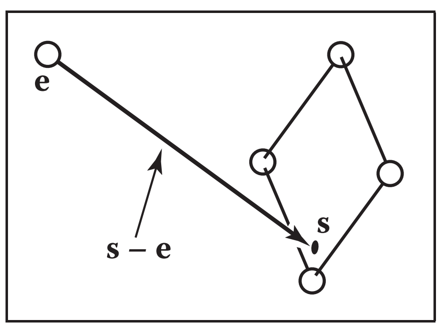
Defining *o* and *d* in Perspective Projection

- Given a viewpoint, $\mathbf{e}$, and a position on the image plane, $\mathbf{s}$

  $\mathbf{o} = \mathbf{e}$

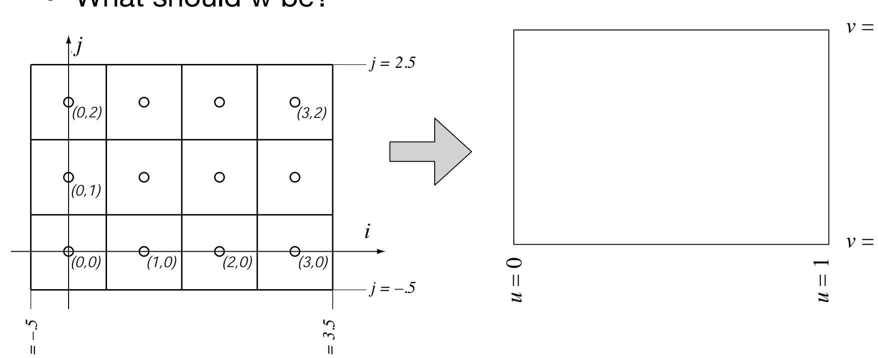
  $\mathbf{d} = \mathbf{s} - \mathbf{e}$

- And thus $\mathbf{p}(t) = \mathbf{e} + t(\mathbf{s} - \mathbf{e})$



Pixel-to-Image Mapping

- Exactly where are pixels located? Must convert from pixel coordinates (i,j) to positions in 3D space (u,v,w)
- What should w be?

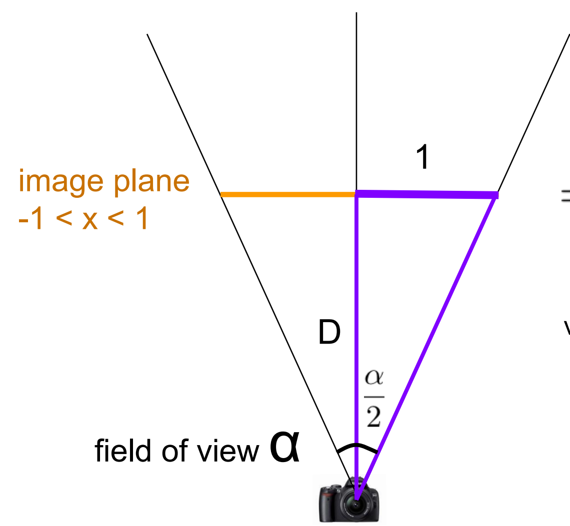


$$u = (i + 0.5)/n_x$$
$$v = (j + 0.5)/n_y$$

Camera Components

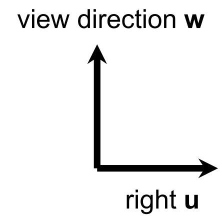
- Definition of an image plane
  - Both in terms of pixel resolution AND position in 3D space or more frequently in **field of view** and/or **distance**
- Viewpoint
- View direction
- Up vector (note that is not necessarily the “up” of the geometric scene

Ray Generation in 2D



$$\tan\frac{\alpha}{2} = \frac{1}{D}$$

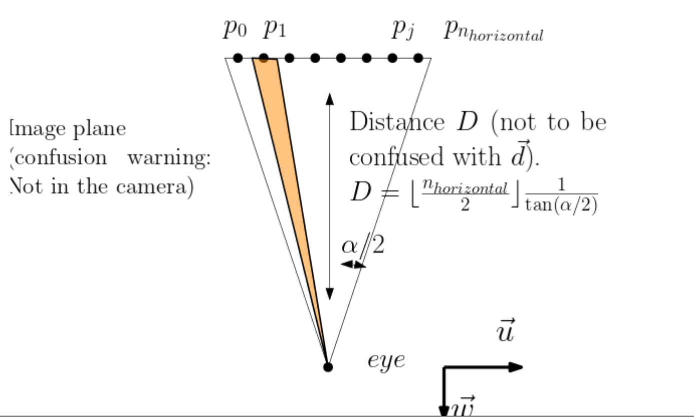
$$\Rightarrow D = \frac{1}{\tan(\alpha/2)}$$



Generating the rays

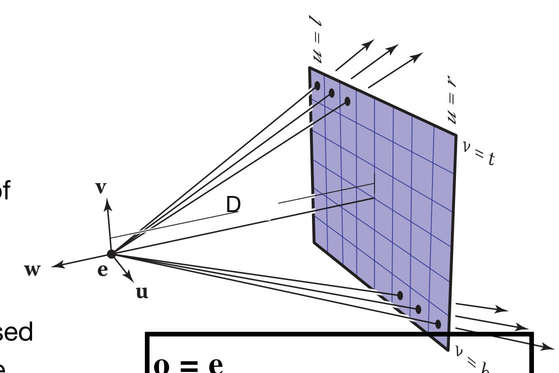
$n_{horizontal}$ is the number of pixels

$$\vec{p}_j = \vec{eye} - D\vec{w} + (j - \lfloor \frac{n_{horizontal}}{2} \rfloor + 0.5)\vec{u}$$



From 2D to 3D

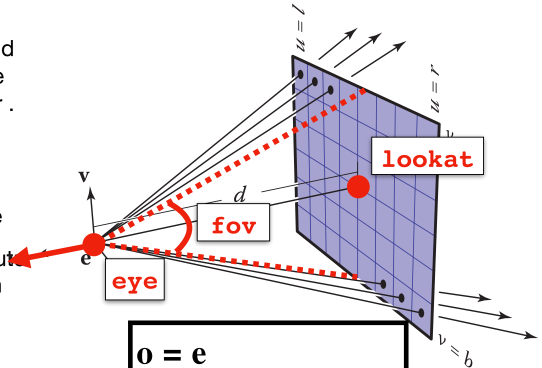
- Moving from 2D to 3D is essentially the same thing once you can define the positions of pixels in uvw space
- Following the convention of the book, **w** is the negated viewpoint vector
- The up vector, **v**, can be used to define a local coordinate space by computing **u**



$$\boxed{\begin{aligned}&\mathbf{o} = \mathbf{e}\\ &\mathbf{d} = -D\overrightarrow{w} + u\,\overrightarrow{u} + v\,\overrightarrow{v}\end{aligned}}$$

In the Assignment

- An `eye` position
- A position to `lookat`, which is centered in the image
  - **w** can be defined using `eye` and `lookat` as well as the distance D, together with the **`up`** vector .
- A `fov_angle` for the vertical FOV
  - The FOV defines the **height** of the image plane in world space
  - You can then use this to compute the **width** of the image plane in world space using the aspect ratio (`rows/columns`) of the image
- Using the number of `rows/columns` you can then sample u, v



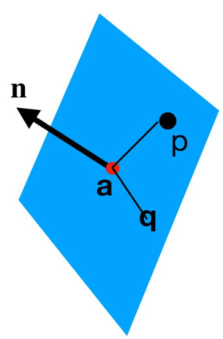
$$\mathbf{o} = \mathbf{e}$$
$$\vec{r} = -D\mathbf{w} + u\mathbf{u} + v\mathbf{v}$$

Intersecting Objects

```
for each pixel {
  compute viewing ray
  intersect ray with scene
  compute illumination at intersection
  store resulting color at pixel
}
```

Defining a Plane

- Let h be a plane with normal **n,** and containing a point **a.** Let p be some other point. Then p is on this plane if and only if (iff) $\mathbf{p}\cdot\mathbf{n} = \mathbf{a}\cdot\mathbf{n}$
- Proof. Consider the segment p-a. p is on the plane iff p-a is orthogonal to n. Using the property of dot product $(\mathbf{p}-\mathbf{a})\cdot\mathbf{n} = |\mathbf{p}-\mathbf{a}||\mathbf{n}|\cos\alpha$
- Here α is the angle between them. Now cos(90)=0. So if p on this plane then $\mathbf{p}\cdot\mathbf{n} = \mathbf{a}\cdot\mathbf{n}$ implying
- If **p n** > **a n** then **p** lives on the "front" side of the plane (in the direction pointed to by the normal
- **p n-an** < 0 means that **p** lives on the "back" side.
- Sometimes used as **f(p)=0** iff ``p on the plane". So the function f(p) is **f(p)=(p-a)n**
- If we have 3 points a,p,q all on the plane, then we can compute a normal $\mathbf{n} = (\mathbf{p}-\mathbf{a})\times(\mathbf{q}-\mathbf{a})$. (cross product).
- Warning: The term "normal" does not mean that it was normalized.



Ray-Plane Intersection

- A ray $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$
- Two conditions must be satisfied:
  - Must be on a ray: $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$
  - Must be on the plane: $f(\mathbf{p}) = (\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0$
- Can substitute the equations and solve for t in $f(\mathbf{p}(t))$:

$$(\mathbf{o} + t\mathbf{d} - \mathbf{a}) \cdot \mathbf{n} = 0$$

- This means that $t_{\text{hit}} = ((\mathbf{a} - \mathbf{o}) \cdot \mathbf{n}) / (\mathbf{d} \cdot \mathbf{n})$. The intersection point is $\mathbf{o} + t_{hit}\mathbf{d}$

Revisiting dot product: projections

Let **u**,**v** be orthonormal vectors **(orthogonal and unit length).** **r** is another vector

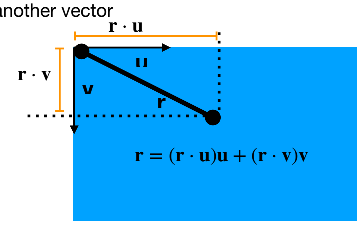
$\mathbf{r} \cdot \mathbf{u} = |\mathbf{r}||\mathbf{u}|\cos\alpha = |\mathbf{r}|\cos\alpha$

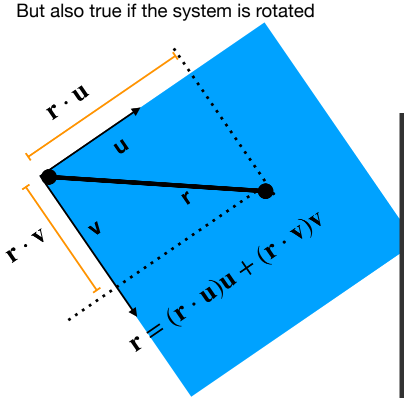
is the projection of **r** on the direction of **u**

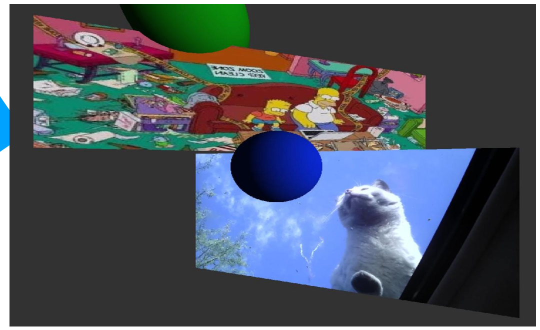
the length of the “shadow” that **r** cast on the line containing **u**

Sounds obvious when the coordinate system is xyz

But also true if the system is rotated





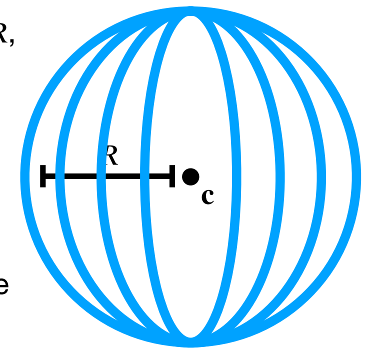


Defining a Sphere

- We can define a sphere of radius R, centered at position $\mathbf{c}$, using the implicit form

$$f(\mathbf{p}) = (\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - R^2 = 0$$

- Any point $\mathbf{p}$ that satisfies the above lives on the sphere



Ray-Sphere Intersection

- Two conditions must be satisfied:
  - Must be on a ray: $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$
  - Must be on a sphere: $f(\mathbf{p}) = (\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - R^2 = 0$
- Can substitute the equations and solve for t in $f(\mathbf{p}(t))$:

$$(\mathbf{o} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{c}) - R^2 = 0$$

- Solving for t is a quadratic equation

Ray-Sphere Intersection

- Solve $(\mathbf{o} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{c}) - R^2 = 0$ for t:
- Rearrange terms:

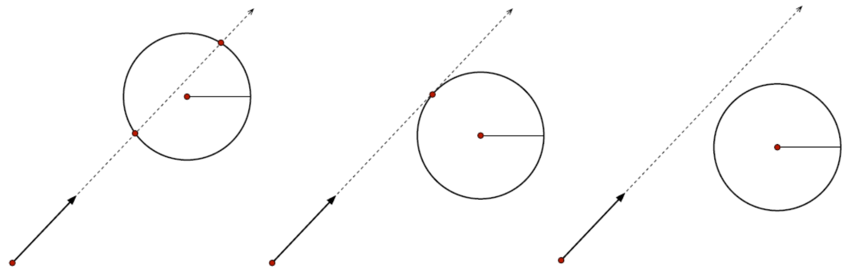
$$(\mathbf{d} \cdot \mathbf{d})t^2 + (2\mathbf{d} \cdot (\mathbf{o} - \mathbf{c}))t + (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - R^2 = 0$$

- Solve the quadratic equation $At^2 + Bt + C = 0$ where
  - $A = (\mathbf{d} \cdot \mathbf{d})$
  - $B = 2^*\mathbf{d} \cdot (\mathbf{o} - \mathbf{c})$
  - $C = (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - R^2$

Discriminant, D = B^2-4*A*C
Solutions must satisfy:
$t = (-B \pm \sqrt{D}) / 2A$

Ray-Sphere Intersection

- Number of intersections dictated by the discriminant
- In the case of two solutions, prefer the one with lower t



Two solutions One solution Imaginary

Geometric Method (instead of Algebraic)

Ray: $P = P_0 + tV$
Sphere: $|P - O|^2 - r^2 = 0$

Geometric Method

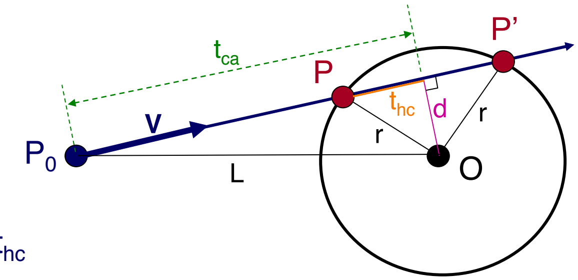
$L = O - P_0$

$t_{ca} = L \bullet V$
if ($t_{ca} < 0$) return 0

$d^2 = L \bullet L - t_{ca}^2$
if ($d^2 > r^2$) return 0

$t_{hc} = \mathrm{sqrt}(r^2 - d^2)$
$t = t_{ca} - t_{hc}$ and $t_{ca} + t_{hc}$

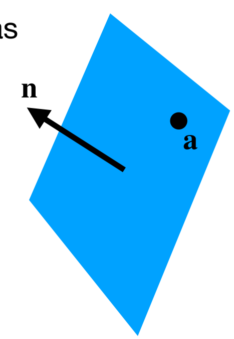
$P = P_0 + tV$



Defining a Plane

- A point $\mathbf{p}$ that satisfies the following implicit form lives on a plane through point $\mathbf{a}$ that has normal $\mathbf{n}$

$$f(\mathbf{p}) = (\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0$$



- $f(\mathbf{p}) > 0$ lives on the "front" side of the plane (in the direction pointed to by the normal
- $f(\mathbf{p}) < 0$ lives on the "back" side

Constructing Orthonormal Bases from a Pair of Vectors

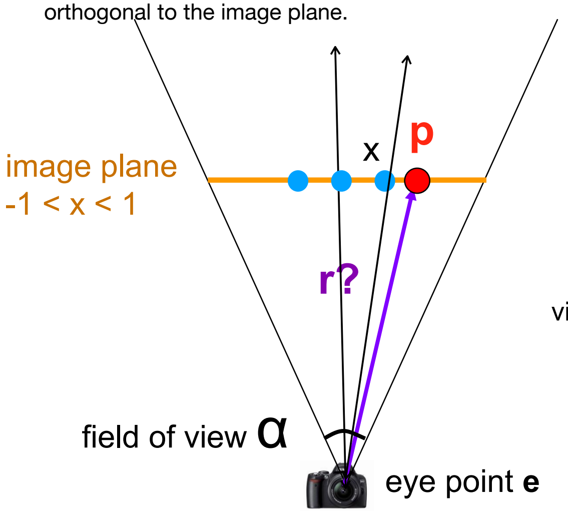
- Given two vectors **a** and **b**, which might not be orthonormal to begin with:

$$\begin{aligned} \mathbf{w} &= \frac{\mathbf{a}}{||\mathbf{a}||}, \\ \mathbf{u} &= \frac{\mathbf{b}\times\mathbf{w}}{||\mathbf{b}\times\mathbf{w}||}, \\ \mathbf{v} &= \mathbf{w} \times \mathbf{u}. \end{aligned}$$

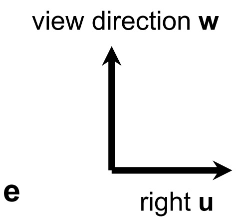
- In this case, **w** will align with **a** and **v** will be the closest vector to **b** that is perpendicular to **w**

Ray Generation in 2D

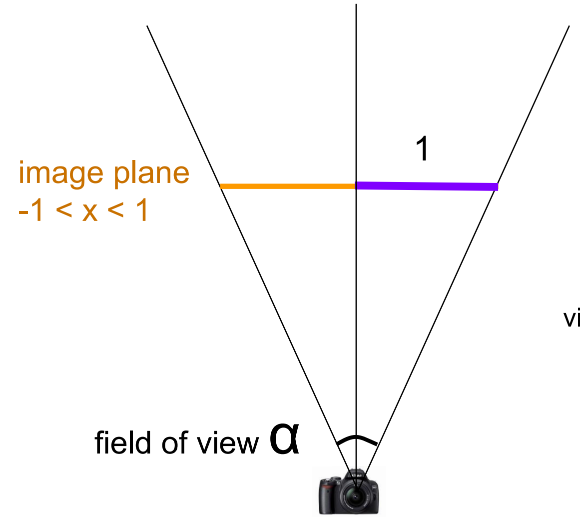
- The image plane (should actually call it ``image line")
- Our algorithm will assign a color to each pixel, by tracing a ray through the pixel and check the color of the object it hits
- User determined the location **e** of the camera, the direction **w** through, and orthogonal to the image plane.



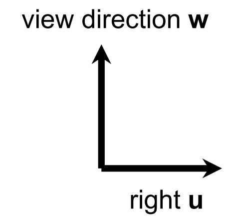
p is point on image plane at coordinate x, we want to know the direction of the ray **r**



Ray Generation in 2D

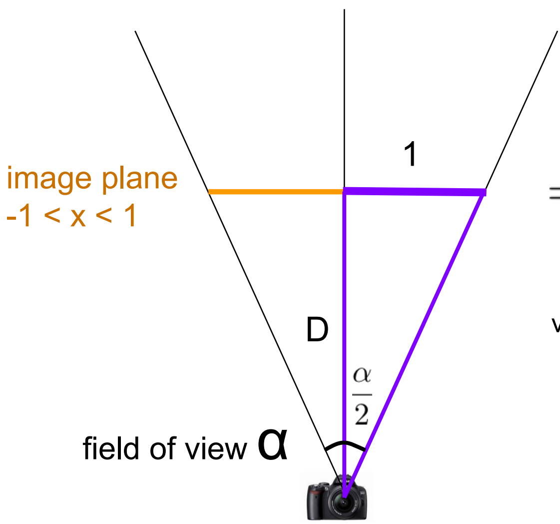


What is the distance D to the screen so that the normalized coordinates go to 1?



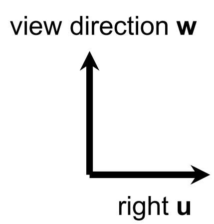
The user specifies the field of view angle `fov_angle.`

`We need to calculate the distance D to the image plane`



$$\tan\frac{\alpha}{2} = \frac{1}{D}$$

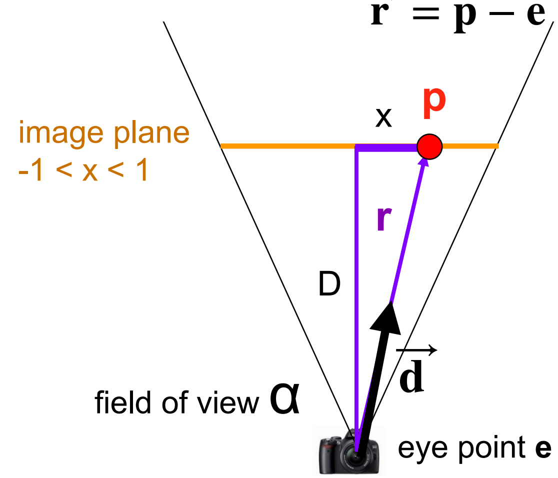
$$\Rightarrow D = \frac{1}{\tan(\alpha/2)}$$



Ray Generation in 2D

$$\vec{\mathbf{p}} = \mathbf{e} + x\vec{\mathbf{u}} + D\vec{\mathbf{w}}$$

$$\vec{\mathbf{r}} = \mathbf{p} - \mathbf{e}$$

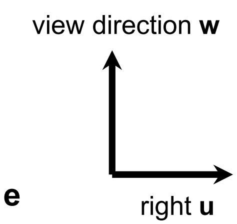


D already computed

Lets normalized $\vec{\mathbf{r}}$

$\vec{\mathbf{d}} = \vec{\mathbf{r}} / |\vec{\mathbf{r}}|$

The ray is $\vec{\mathbf{P}}(\mathbf{t}) = \mathbf{e} + t\vec{\mathbf{r}}$

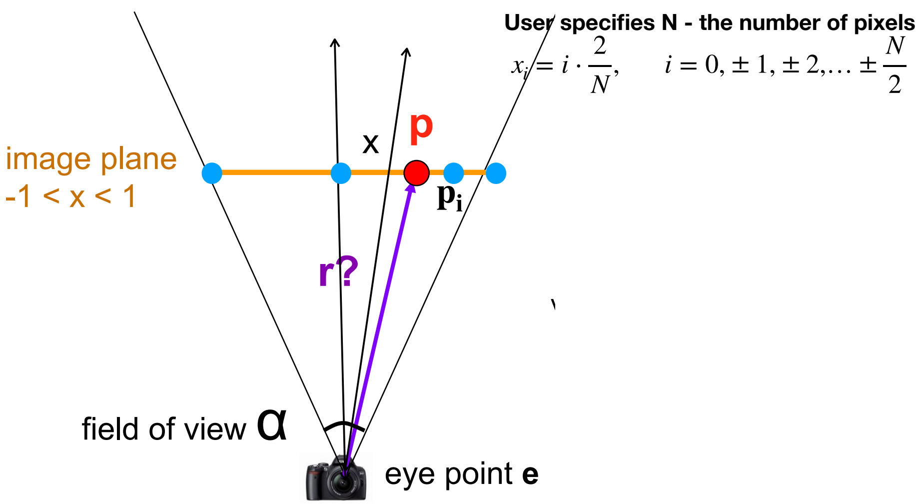


Calculating all the rays

$$\vec{\mathbf{p}}_{\mathbf{i}} = \mathbf{e} + x_i \vec{\mathbf{u}} + D\vec{\mathbf{w}}$$

User specifies N - the number of pixels

$$x_i = i \cdot \frac{2}{N}, \qquad i = 0, \pm 1, \pm 2, \dots \pm \frac{N}{2}$$

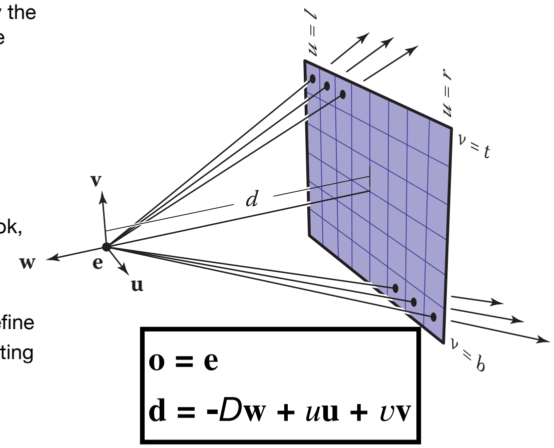


From 2D to 3D

- Moving from 2D to 3D is essentially the same thing once you can define the positions of pixels in uvw space

  from now on, we use *d* to denote distance to image plane

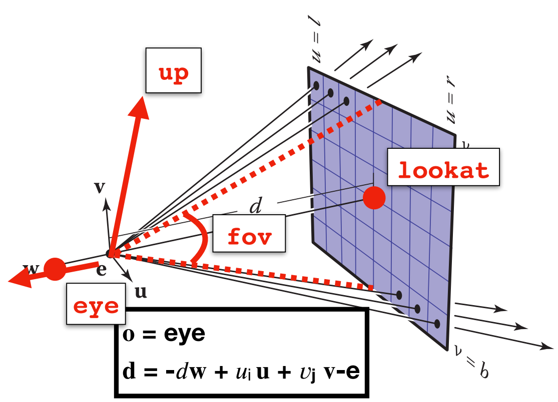
- Following the convention of the book, **w** is the negated viewpoint vector
- The up vector, **v**, can be used to define a local coordinate space by computing **u**



In the Assignment

- An `eye` position
- A position to `lookat`, which is centered in the image
  - **w** can be defined use `eye` and `lookat` as well as *d*
- An `up` vector, not necessarily **v**! (but the vectors v up w in the same plane)
- A `fov_angle` for the **vertical** FOV
- The FOV defines the **height** of the image plane in world space
- Each pixel is a square, but number of pixels rows vs columns might be different
- You can then use this to compute the **width** of the image plane in world space using the aspect ratio (`rows`/`columns`) of the image
- Using the number of `rows`/`columns` you can then sample u, v

We use a terminology that is very common. Not always most intuitive



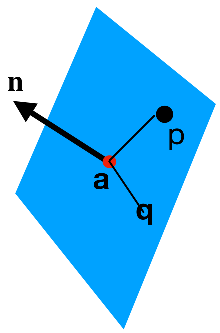
$$u_i = i \cdot 2/N_{cols}, \qquad i = 0, \pm 1, \pm 2, \dots \pm N_{cols}/2$$

Intersecting Objects

```
for each pixel {
  compute viewing ray
  intersect ray with scene
  compute illumination at intersection
  store resulting color at pixel
}
```

Defining a Plane

- Let h be a plane with normal **n,** and containing a point **a.** Let p be some other point. Then p is on this plane if and only if (iff) $\mathbf{p} \cdot \mathbf{n} = \mathbf{a} \cdot \mathbf{n}$
- Proof. Consider the segment p-a. p is on the plane iff p-a is orthogonal to n. Using the property of dot product $(\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = |\mathbf{p} - \mathbf{a}|\,|\mathbf{n}|\cos\alpha$
- Here α is the angle between them. Now cos(90)=0. So if p on this plane then $\mathbf{p} \cdot \mathbf{n} = \mathbf{a} \cdot \mathbf{n}$ implying
- If **p n** > **a n** then **p** lives on the "front" side of the plane (in the direction pointed to by the normal
- **p n-an** < 0 means that **p** lives on the "back" side.
- Sometimes used as **f(p)=0** iff ``p on the plane". So the function f(p) is **f(p)=(p-a)n**
- If we have 3 points a,p,q all on the plane, then we can compute a normal $\mathbf{n} = (\mathbf{p} - \mathbf{a}) \times (\mathbf{q} - \mathbf{a})$. (cross product).
- Warning: The term "normal" does not mean that it was normalized.



Ray-Plane Intersection

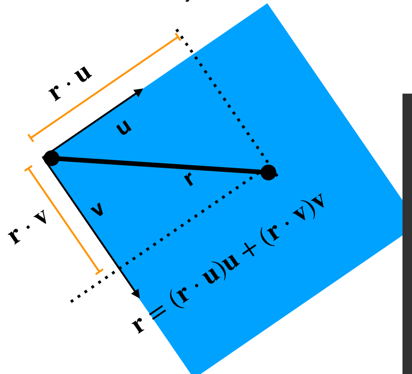
- A ray $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$
- Two conditions must be satisfied:
  - Must be on a ray: $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$
  - Must be on the plane: $f(\mathbf{p}) = (\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0$
- Can substitute the equations and solve for t in $f(\mathbf{p}(t))$:

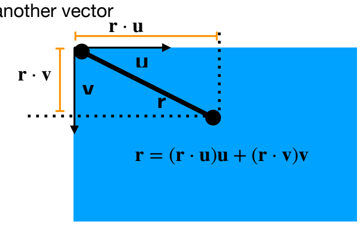
$$(\mathbf{o} + t\mathbf{d} - \mathbf{a}) \cdot \mathbf{n} = 0$$

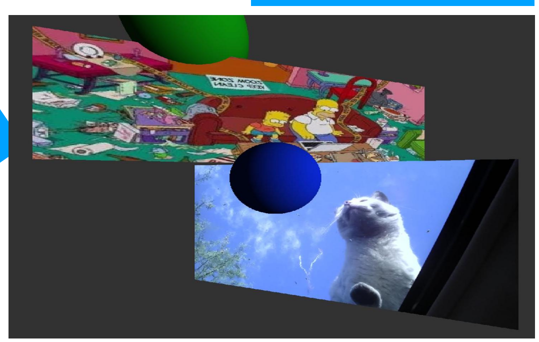
- This means that $t_{\text{hit}} = ((\mathbf{a} - \mathbf{o}) \cdot \mathbf{n}) / (\mathbf{d} \cdot \mathbf{n})$. The intersection point is $\mathbf{o} + t_{hit}\mathbf{d}$

Revisiting dot product: projections

Let **u,v** be orthonormal vectors **(orthogonal and unit length). r** is another vector
$\mathbf{r} \cdot \mathbf{u} = |\mathbf{r}|\,|\mathbf{u}|\cos\alpha = |\mathbf{r}|\cos\alpha$
is the projection of **r** on the direction of **u**
the length of the "shadow" that **r** cast on the line containing **u**
Sounds obvious when the coordinate system is xyz
But also true if the system is rotated







Barycentric Coordinates

- A coordinate system to write all points **p** as a weighted sum of the vertices

  $\mathbf{p} = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$

  $\alpha + \beta + \gamma = 1$,

- Equivalently, α, β, γ are the proportions of area of subtriangles relative total area, A

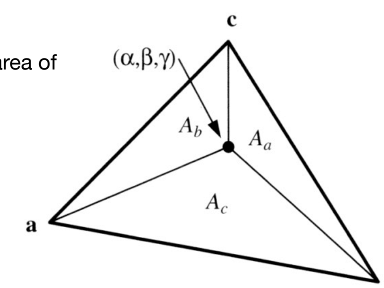
  $A_a / A = \alpha$

  $A_b / A = \beta$

  $A_c / A = \gamma$

- Triangle interior test:

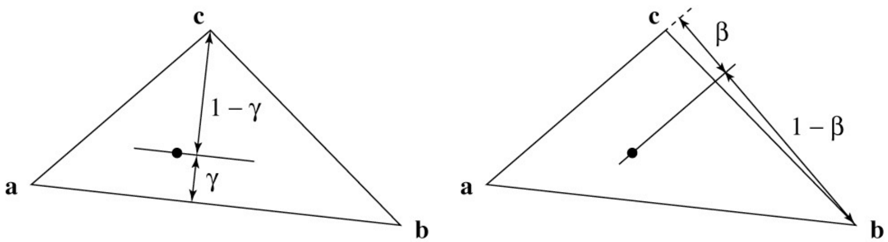
  $\alpha > 0$, $\beta > 0$, and $\gamma > 0$



[Shirley 2000]

Barycentric Coordinates

- Also related to distances



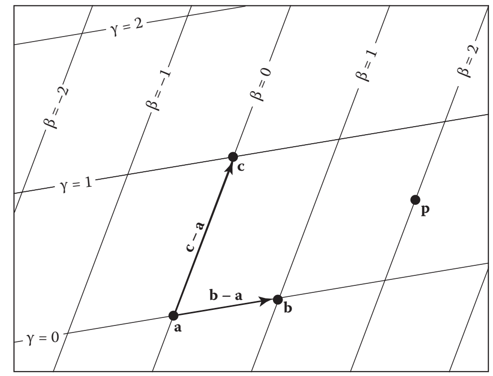
- And, they provide a basis relative to the edge vectors

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

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

Barycentric Coordinates

- This basis defines the plane of the triangle



- In this view, the triangle interior test becomes:

$\beta > 0,\ \gamma > 0,\ \beta + \gamma \leq 1$

Barycentric Ray-Triangle Intersection

- Two conditions must be satisfied:
  - Must be on a ray: $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$
  - Must be in the triangle: $\mathbf{p} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$
- So, set them equal and solve for t, β, γ:

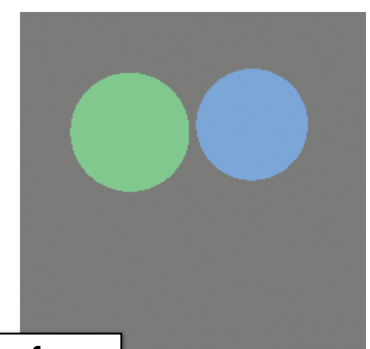
$$\mathbf{o} + t\mathbf{d} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$$

- This is possible to solve because you have 3 equations and 3 unknowns

Our images so far

- With only eye-ray generation and scene intersection

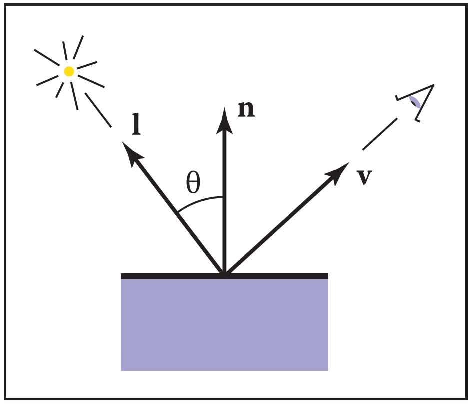
```
for each pixel p in Image {
  let hit_surf = undefined;
  ...

  scene.surfaces.forEach( function(surf) {
    if (surf.intersect(eye, dir, ...)) {
      hit_surf = surf;
      ...
    }
  });

  c = hit_surf.ambient;
  Image.update(p, c);
}
```



Each surface storing a single ambient color

Shading

- Goal: Compute light reflected toward camera
- Inputs:
  - eye direction
  - light direction (for each of many lights)
  - surface normal
  - surface parameters (color, shininess, ...)

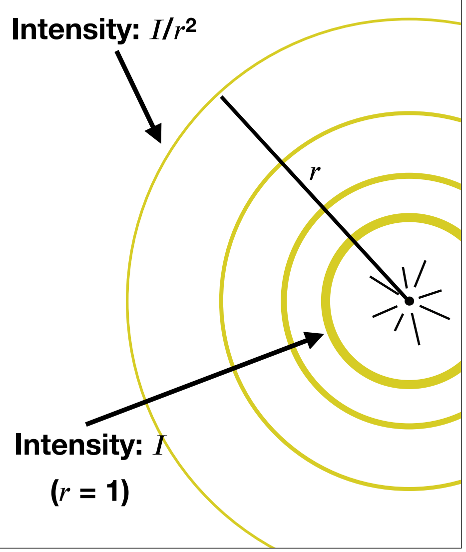


Normals

- The amount of light that reflects from a surface towards the eye depends on orientation of the surface at that point
- A **normal vector** describes the direction that is orthogonal to the surface at that point
- What are normal vectors for planes and triangles?
  - $\mathbf{n}$, the vector we already were storing!
- What are normal vectors for spheres?
  - Given a point $\mathbf{p}$ on the sphere $\mathbf{n} = (\mathbf{p} - \mathbf{c}) / \|\mathbf{p} - \mathbf{c}\|$

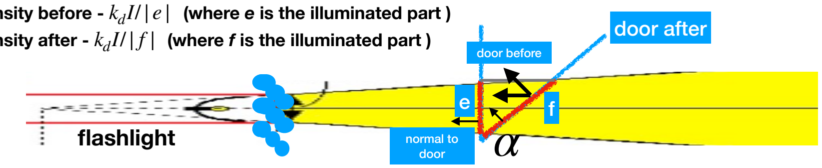
Light Sources

- There are many types of possible ways to model light, but for now we'll focus on **point lights**
- Point lights are defined by a position $\mathbf{p}$ that irradiates equally in all directions
- Technically, illumination from real point sources falls off relative to distance squared, but <u>**we will ignore this for now.**</u>



Lambertian (Diffuse) Shading

- Lets think about the intensity of the light in terms of energy reflected toward the viewer.
- Consider a door illuminated by a flashlight (see below).
- Lets think about the intensity reflected from the door as the door rotates.
- Let I denote the total light energy that the flashlight emits per second (can think about it as #photons / second)
- The Intensity of the light reflected from the door is $\dfrac{k_d \cdot I}{\textit{The area of the illumninated portion}}$.
- Intensity before - $k_d I / |e|$ (where *e* is the illuminated part)
- Intensity after - $k_d I / |f|$ (where *f* is the illuminated part)



$$\frac{|e|}{|f|} = \cos\alpha \quad \text{or} \quad |f| = |e|\frac{1}{\cos\alpha} \quad \text{Implyting that} \quad \frac{k_d I}{|f|} = \frac{k_d I}{|e|\frac{1}{\cos\alpha}} = \frac{k_d I}{|e|}\cos\alpha$$

But I/|e| is the intensity of the reflected light for the before at the "before" stage.

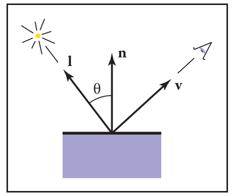
Conclusion - the intensity decrease by a factor of $\cos(\alpha)$

But $\cos(\alpha)$ is just the dot product of two vectors:

1) Normal of the door, and, 2) direction to the light source

Lambertian (Diffuse) Shading

- Simple model: amount of energy from a light source depends on the direction at which the light ray hits the surface
- Results in shading that is *view independent*

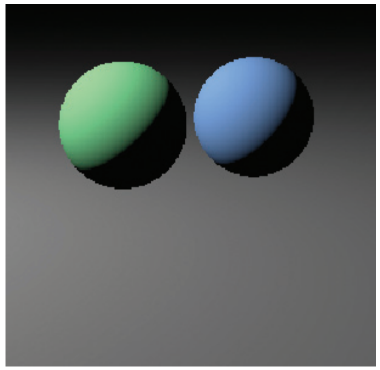


$$L_d = k_d I \max(0, \mathbf{n} \cdot \mathbf{l})$$

diffuse coefficient

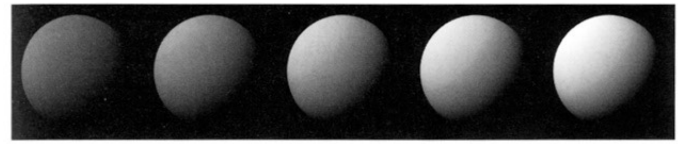
intensity/color of light

cos θ



Lambertian Shading

- k_d is a property of the surface itself (3 constants - one per each color channel)
- Produces matte appearance of varying intensities

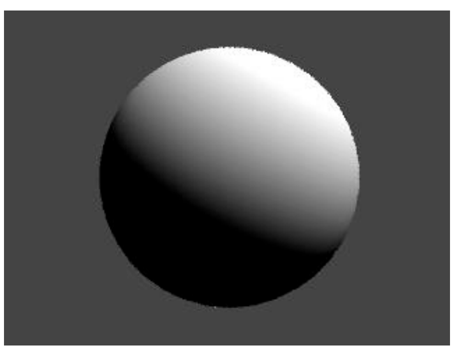


$k_d \longrightarrow$

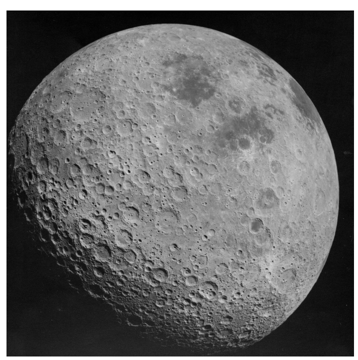
[Foley et al.]

The moon paradox

- why don't we see this gradual shading when looking at the moon ?

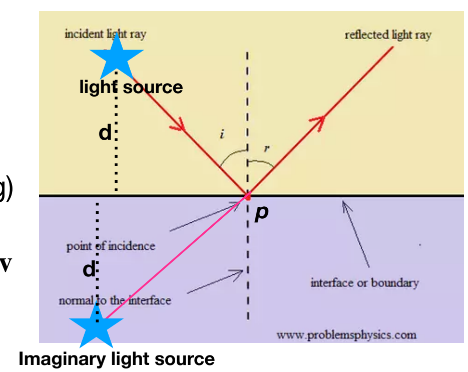


vs

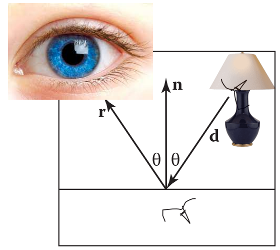


Toward Specular Shading: Perfect Mirror

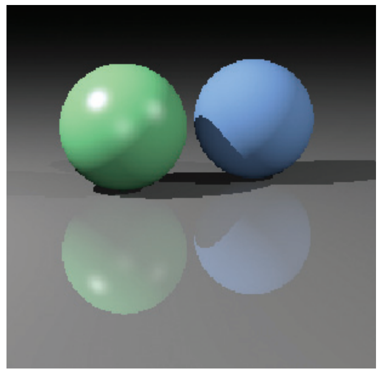
- Many real surfaces show some degree of shininess that produce **specular** reflections
- These effects move as the viewpoint changes (as oppose to diffuse and ambient shading)
- Idea: produce reflection when $\mathbf{v}$ and $\mathbf{l}$ are symmetrically positioned across the surface normal



Reflection

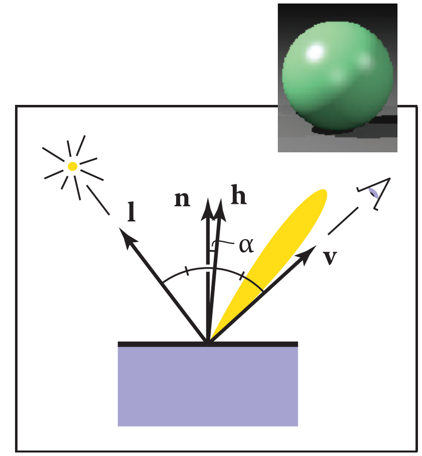


- Ideal **specular** reflection, or mirror reflection, can be modeled by casting another ray into the scene from the hit point
- Direction $\mathbf{r} = \mathbf{d} - 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}$
- **r**- reflected ray toward the eye, **d -** ray from lamp. **n** is a unit vector orthogonal to the plane.
- Proof
  - (handwave) **r**=(r.x, r.y) and **d**=(d.x,d.y)
  - **r** and **d** have the same x-value, but opposite y-value: r.x=d.x and r.y= -r.y = r.y+(-2r.y)= r.y -2 (n · r)
  - $(\mathbf{d} \cdot \mathbf{n})\mathbf{n}$=(0, r.y).
- One can then recursively accumulate some amount of color from whatever object this hits
- color += k_m`*ray_cast()`



Blinn-Phong (Specular) Shading

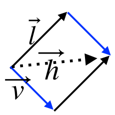
- Many real surfaces show some degree of shininess that produce specular reflections
- These effects move as the viewpoint changes (as oppose to diffuse and ambient shading)
- Idea: produce reflection when $\mathbf{v}$ and $\mathbf{l}$ are symmetrically positioned across the surface normal

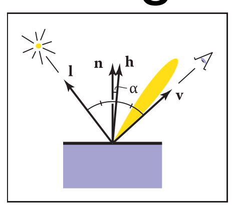


Blinn-Phong (Specular) Shading

- For any two **unit** vectors $\vec{v}, \vec{l}$, the vector $\mathbf{n}$ is a bisector of the angle between these vectors.
- Normalize $\mathbf{v} + \mathbf{l}$

$\mathbf{h} = (\mathbf{v} + \mathbf{l}) / \|\mathbf{v} + \mathbf{l}\|$



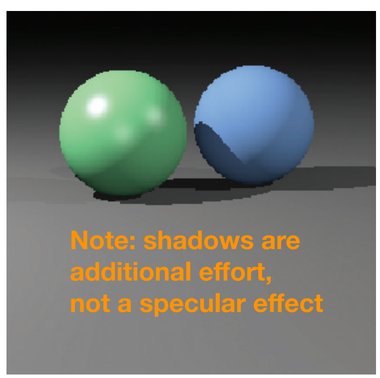


- In a perfect mirror, the 100% of the reflection occurs at the surface point where **h** is the normal **n**
- Diffuse reflection. Reflect large value for points where **h** is ``almost'' **n**
- Phong heuristic:

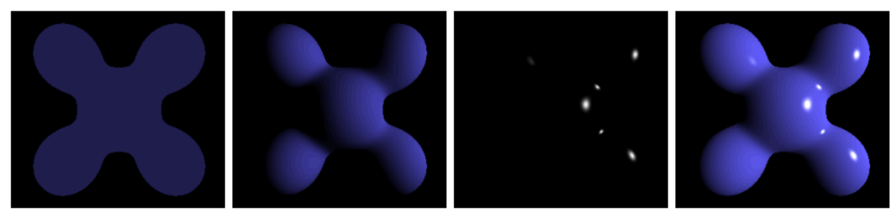
$$L_s = k_s I \max(0, \ (\mathbf{n} \cdot \mathbf{h})^{p} \ )$$

specular coefficient

Phong exponent



Blinn-Phong Decomposed

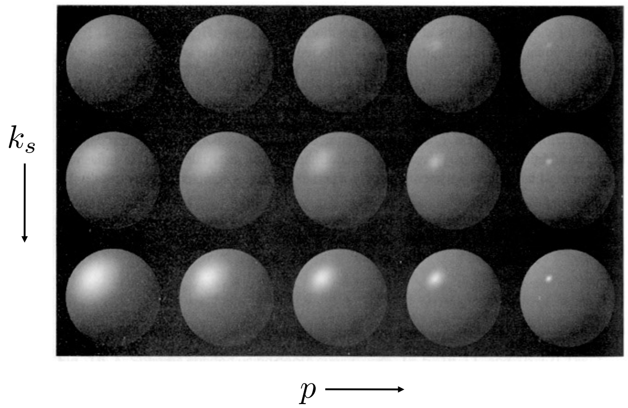


Ambient + **Diffuse** + **Specular** = **Phong Reflection**

https://en.wikipedia.org/wiki/Phong_shading

Blinn-Phong Shading

- Increasing p narrows the lobe
- This is kind of a hack, but it does look good



[Foley et al.]

Putting it all together

- Usually include ambient, diffuse, and specular in one model

$$L = L_a + L_d + L_s$$

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

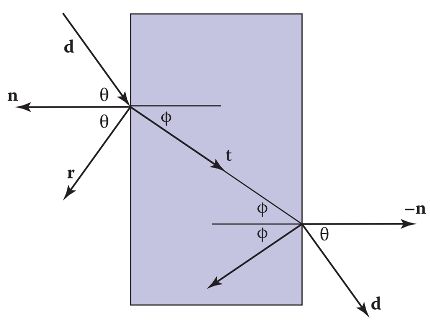
- And, the final result accumulates for all lights in the scene

$$L = k_a I_a + \Sigma_i \left(k_d I_i \max(0, \mathbf{n} \cdot \mathbf{l}_i) + k_s I_i \max(0, \mathbf{n} \cdot \mathbf{h}_i)^p \right)$$

- Be careful of overflowing! You may need to clamp colors, especially if there are many lights.

Snell's Law

- Governs the angle at which a refracted ray bends
- Computation based on *refraction index (confusingly denoted* n_t) of the mediums. The mediums here are air and glass. They air has refraction index n=1, while the glass has refraction index n_t
- Snell law: $n_t \sin\theta = n \sin\phi$

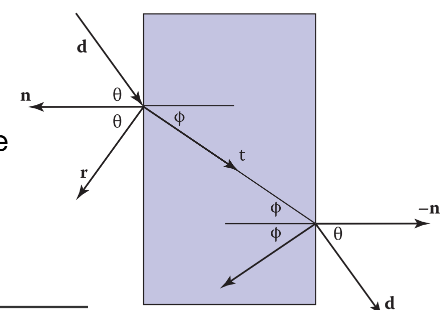


Snell's Law

- Working with cosine's are easier because we can use dot products
- Can derive the vector for the refraction direction $\mathbf{t}$ as

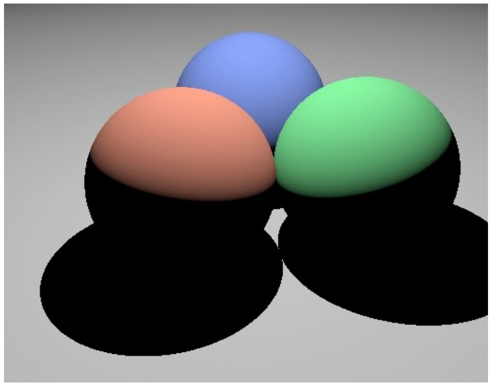
$$\mathbf{t} = \frac{n(\mathbf{d}+\mathbf{n}\cos\theta))}{n_t} - \mathbf{n}\cos\phi$$

$$= \frac{n(\mathbf{d}-\mathbf{n}(\mathbf{d}\cdot\mathbf{n}))}{n_t} - \mathbf{n}\sqrt{1 - \frac{n^2\left(1-(\mathbf{d}\cdot\mathbf{n})^2\right)}{n_t^2}}$$

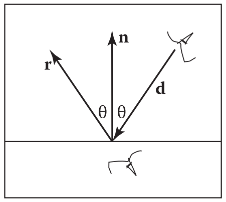


Shadows

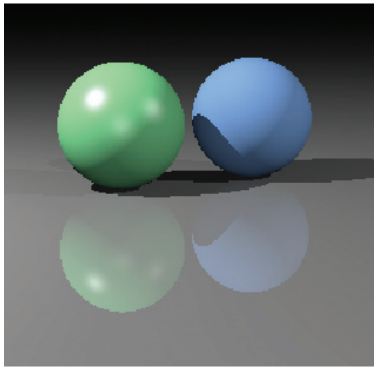
- Idea: after finding the closest hit, cast a ray to each light source to determine if it is visible
- Be careful not to intersect with the object itself. Two solutions:
  - Only check for hits against all other surfaces
  - Start shadow rays a tiny distance away from the hit point by adjusting t_{min}



Reflection

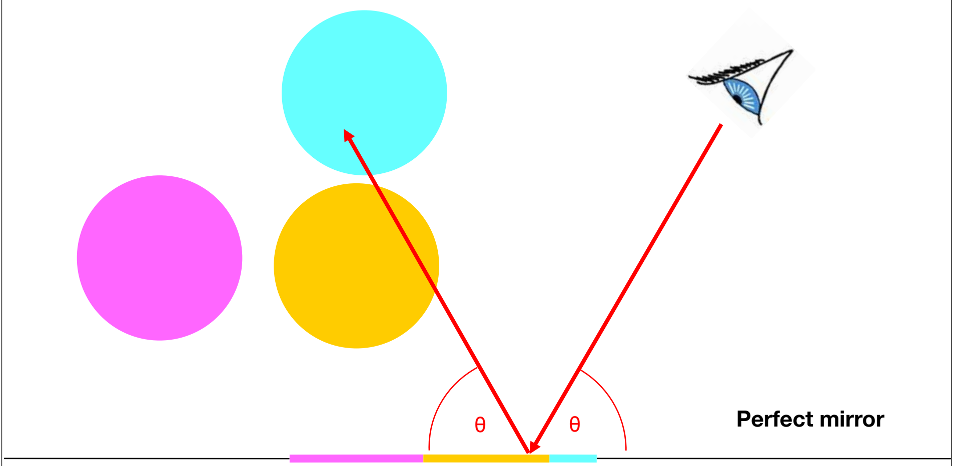


- Ideal **specular** reflection, or mirror reflection, can be modeled by casting another ray into the scene from the hit point
- Direction $\mathbf{r} = \mathbf{d} - 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}$
- One can then recursively accumulate some amount of color from whatever object this hits
- color += k_m`*ray_cast()`

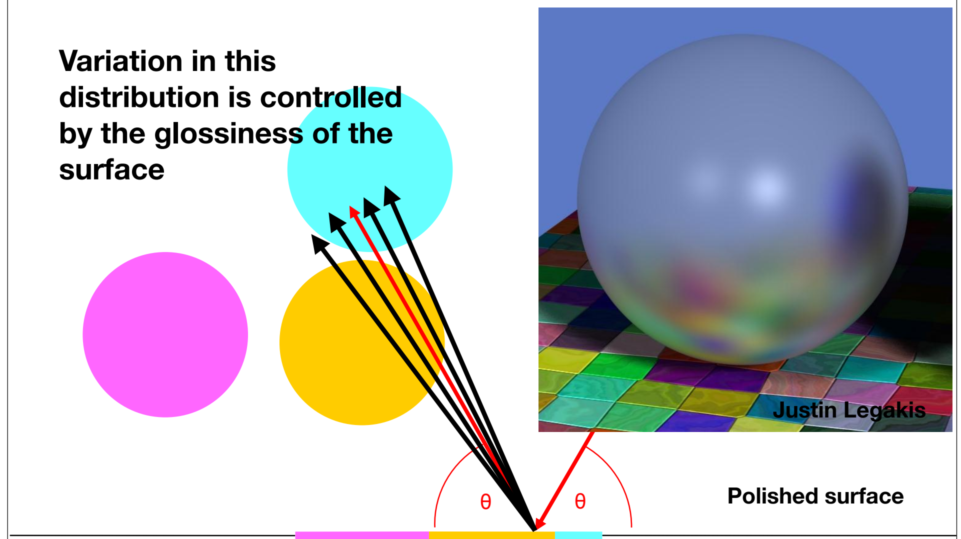


Distribution Ray Tracing

Ideal Reflection: One Ray Per Bounce



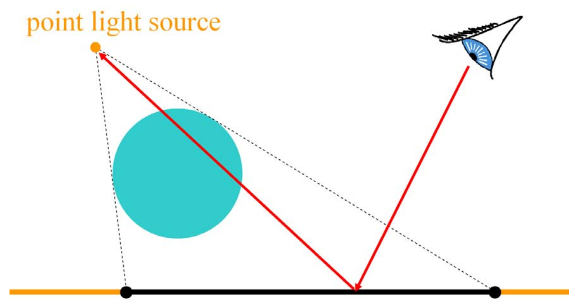
Glossy Reflection: Compute Many Rays per Bounce and Average

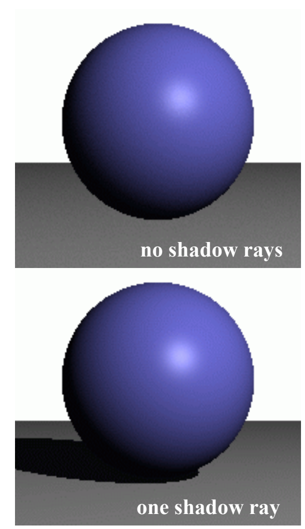


Other Uses of Distribution Ray Tracing

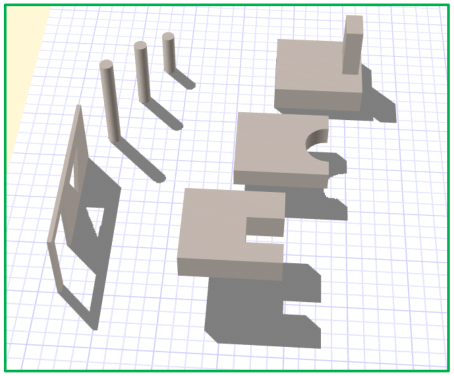
Problem: Hard Shadows

- One shadow ray per intersection per point light source

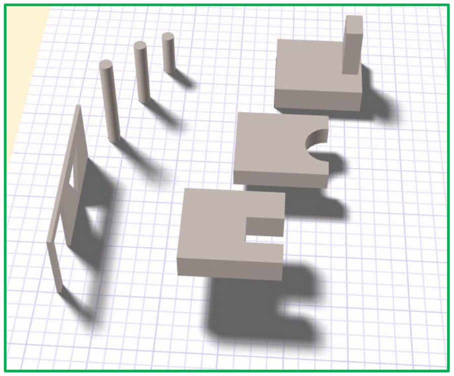




Soft Shadows



Hard shadows

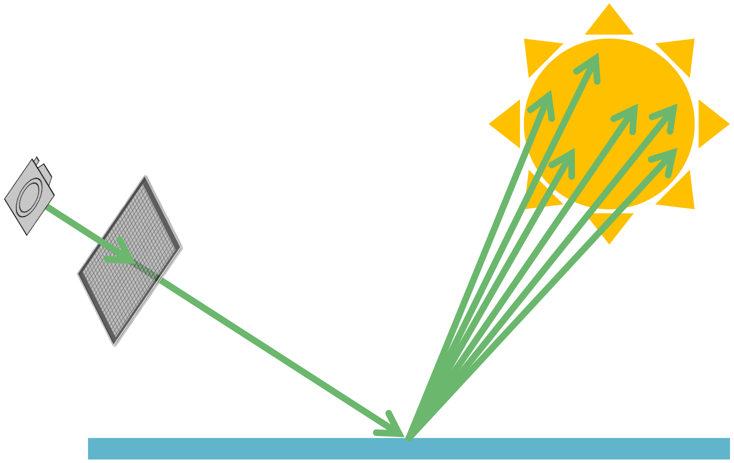


Soft shadows

http://erich.realtimerendering.com/shadow_comparison.html

Soft Shadows

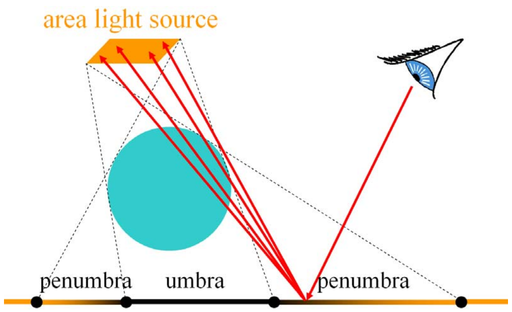
Distribution Soft Shadows

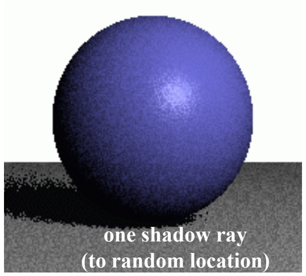


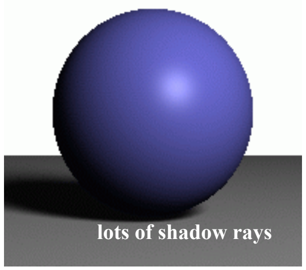
Randomly sample light rays

Computing Soft Shadows

- Model light sources as spanning an area
- Sample random positions on area light source and average rays







Problem: Aliasing
Drawing a black line on a white board

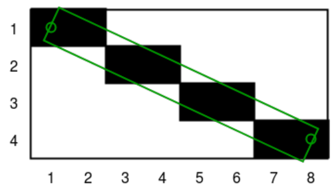
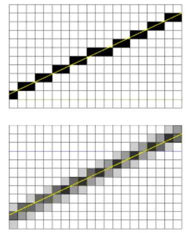


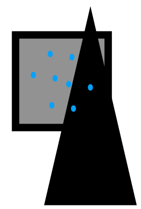
Fig. B: y=f(x) approximation



Some pixels need to be rendered as gray, with gray level=

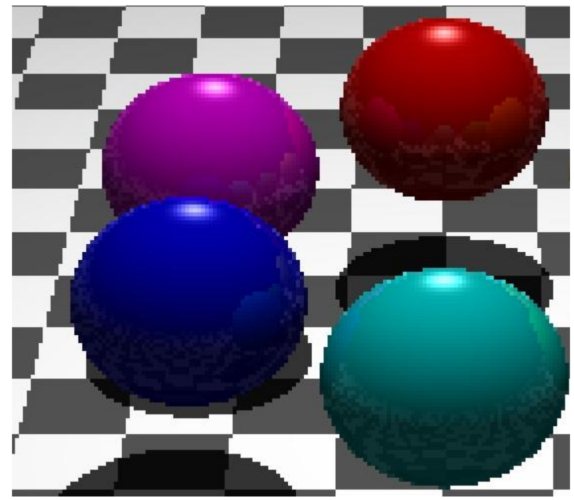
$$\frac{\textbf{Area of black region in pixel}}{\textbf{Area of pixel}}$$

Pixel:



- Problem: Hard to calculate how much of the pixel is covered
- Solution: Random sample points in the pixel.
- Calculate what is the percentage of the point of each color

Problem: Aliasing

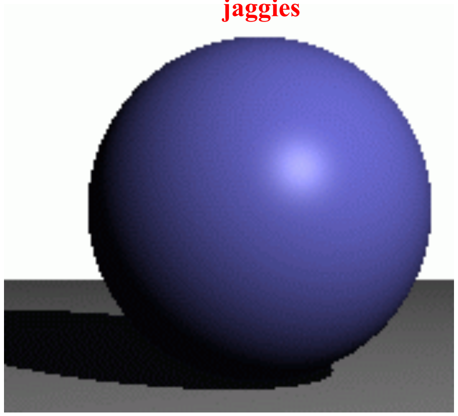


http://www.hackification.com/2008/08/31/experiments-in-ray-tracing-part-8-anti-aliasing/

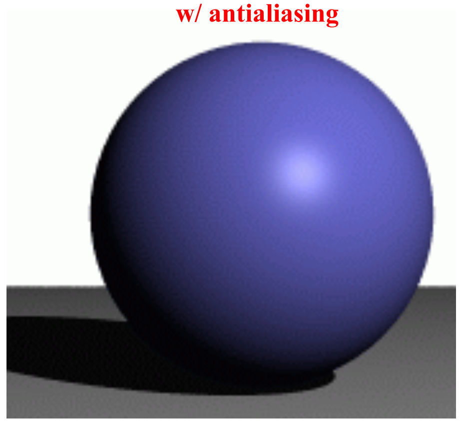
Antialiasing w/ Supersampling

- Cast multiple rays per pixel, average result

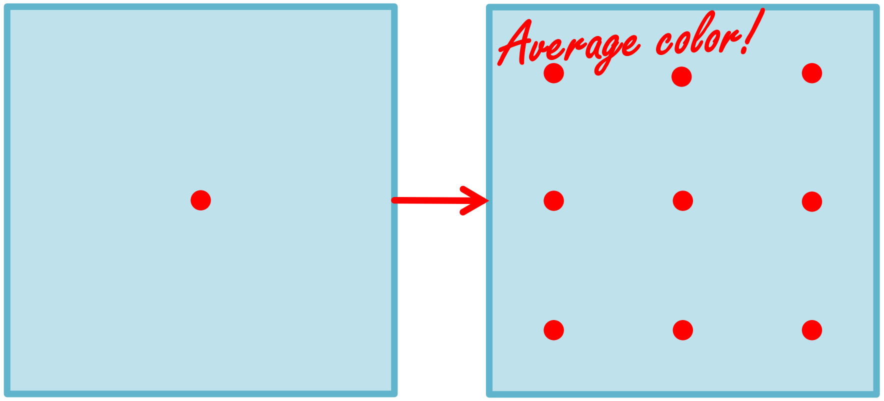
jaggies



w/ antialiasing

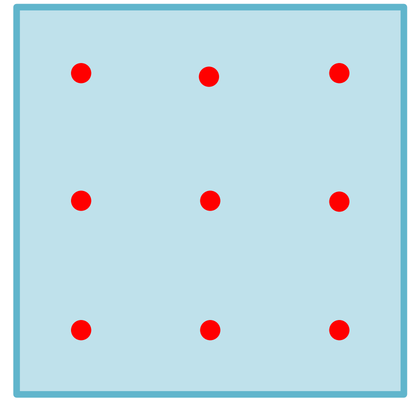


Distribution Antialiasing



Multiple rays per pixel

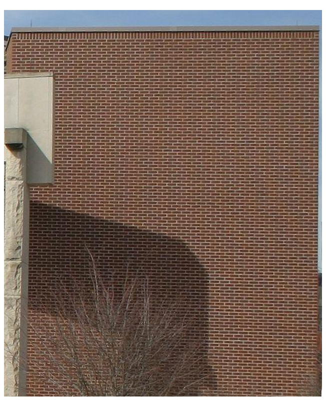
Distribution Antialiasing w/ Regular Sampling

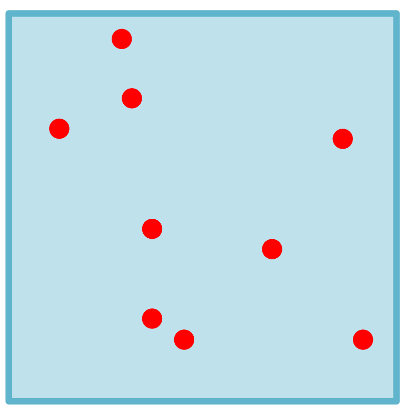


http://upload.wikimedia.org/wikipedia/commons/f/fb/Moire_pattern_of_bricks_small.jpg

Multiple rays per pixel

Distribution Antialiasing w/ Random Sampling

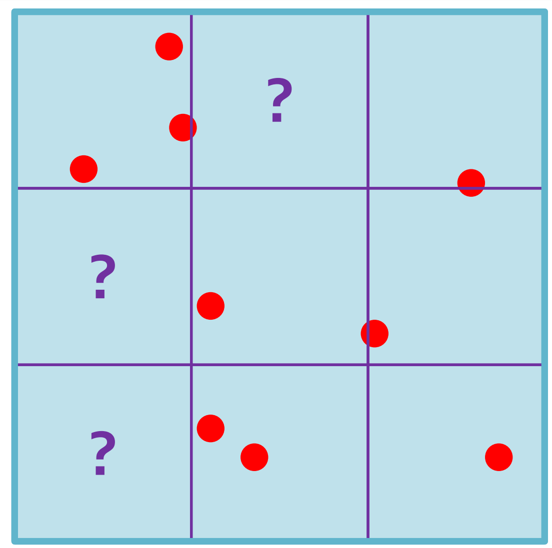




http://en.wikipedia.org/wiki/File:Moire_pattern_of_bricks.jpg

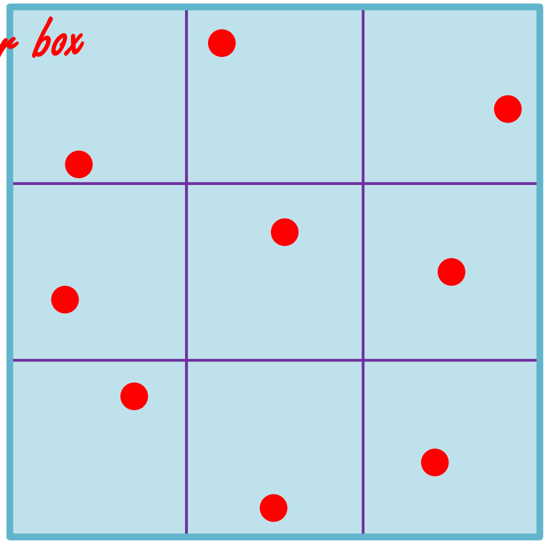
Remove Moiré patterns

Random Sampling Could Miss Regions Without Enough Sampling



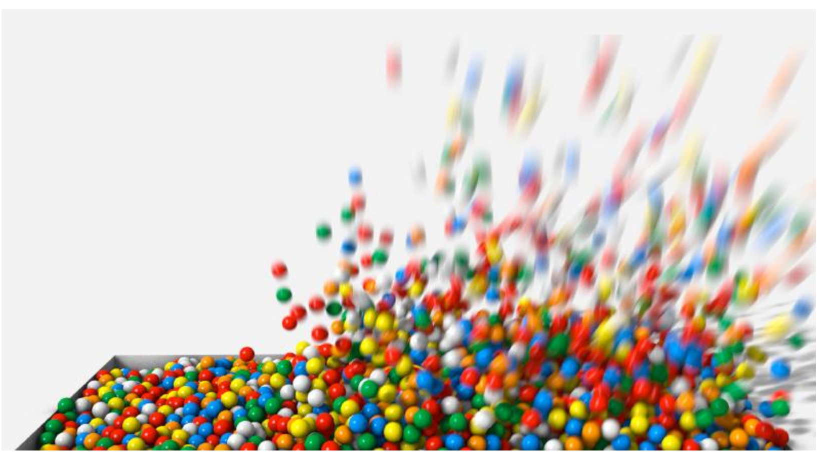
Stratified (Jittered) Sampling

One ray per box



Problem: Exposure Time
Real Sensors Take Time to Acquire

Problem: Exposure Time
Real Sensors Take Time to Acquire



http://www.matkovic.com/anto/3dl-test-balls-01.jpg

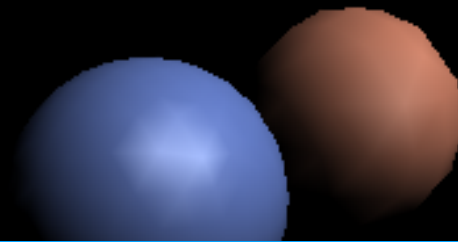
Shading on surfaces

- In practice, we have colors given either to each pixel (texture), or color for each vertex. The discussion below is only about shading
- For simplicity, assume surface has uniform color
- Problem: How could we produce the shading ? Shedding requires normal for each pixels
- If we are happy with a polyhedra surface - just compute for each face the normal.
- If on the other hand, the surface interpolates a smooth surface (e.g. a sphere), we should think about other alternative

Shading on Surfaces

- In practice, we have colors given either to each pixel (texture), or color for each vertex. The discussion below is only about shading
- For simplicity, assume surface has uniform color
- Problem: How could we produce the shading ? Shedding requires normal for each pixels
- If we are happy with a polyhedra surface - just compute for each face the normal.
- If on the other hand, the surface interpolates a smooth surface (e.g. a sphere), we should think about other alternative

Results of Gouraud Shading Pipeline



- Slightly idea. For every vertex v, compute the approximated normal.
- Compute the (shaded) color in each vertex
- Inside each triangle, use barycentric coordinates to interpolate the color.
- When interpolating inside a rectangle (billboard) use the values of α, β as discussed in hw3

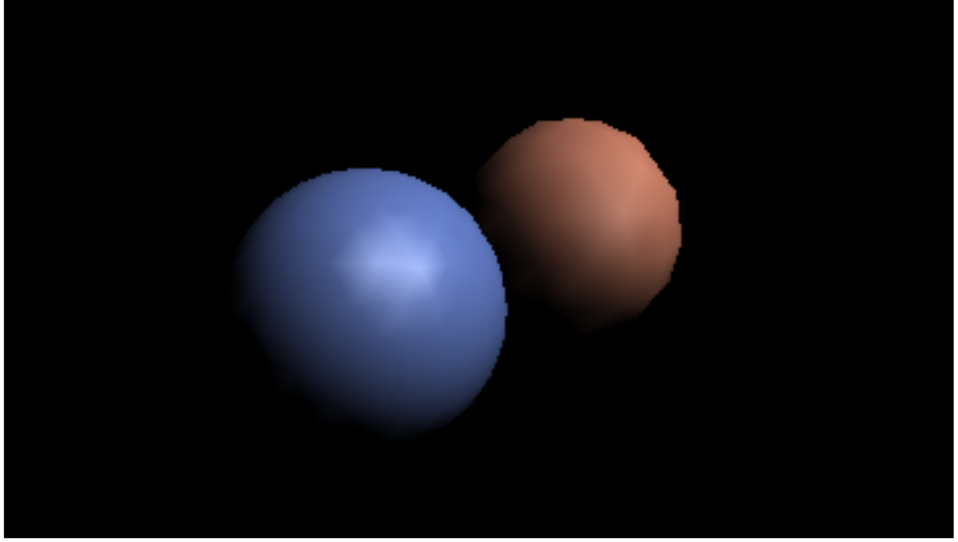
Problems w/ Gouraud Shading

- While you can use any shading model on the vertices (Gouraud shading is just an interpolation method!), typically using only diffuse color works best.
- Results tend to be poor with rapidly-varying models like specular color
  - In particular, when triangles are large relative to how quickly color is changing

Better shading (but slower): Phong Interpolation Shading

- Think about a triangle with vertices v_1, v_2, v_3 or a billboard with corners $p_{LL}, p_{UL}, p_{LR}, p_{UR}$, compute the normals at the corners.
- For each pixel *p* on this triangle or billboard, express p as the convex combination of this corners (needs to compute the weights)
  - (for a triangle) $p = \alpha_1 p_1 + \alpha_2 p_2 + \alpha_3 p_3$ (barycentric coordinates)
  - (for a recitingle)
    $p = \alpha\beta \cdot P_{UL} + (1-\alpha)\beta \cdot \ P_{UR} \ + \ \alpha(1-\beta)P_{LL} + (1-\alpha)(1-\beta)P_{LR}$
- Compute its interpolated normals $\vec{n} = \alpha_1 \vec{n}_1 + \alpha_2 \vec{n}_2 + \alpha_3 \vec{n}_3$
- Normalize its length
- Use this normal (for each pixel) to compute its shading, as if it is the real normal
- See formula on whiteboard
- Caution: interpolated normals must be of unit length
- Caution: Don't confuse with Phong Specular Shading
  - (same person, two different concept)

Results of Gouraud Shading Pipeline



Results of Phong Shading Pipeline