Ray Tracing (Intersection)

CS 4620 Lecture 6

Announcements

- A1 is done
  - Demo slots on Monday evening. Sign up.

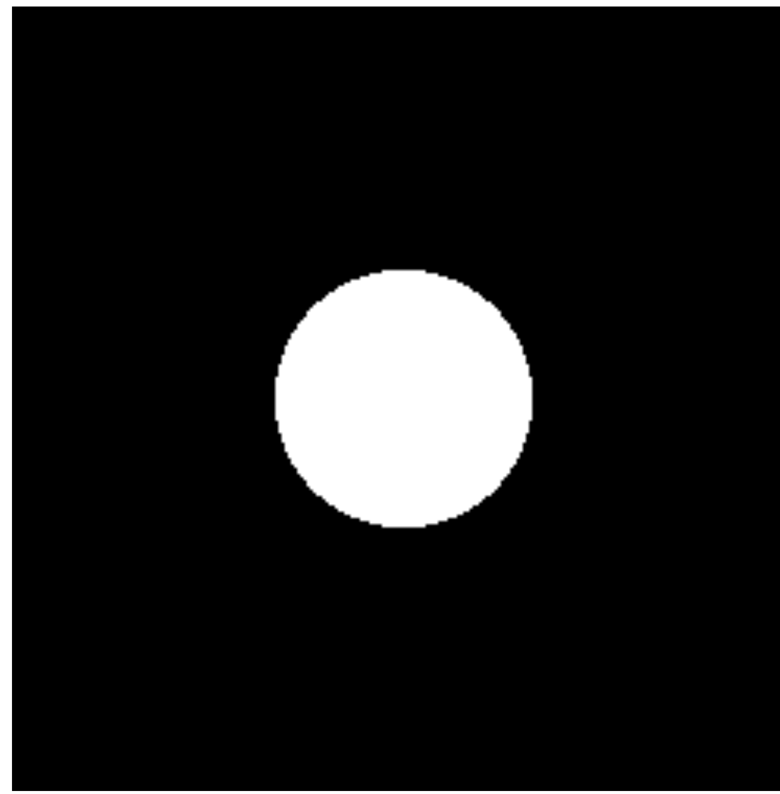
- A2 will be out today

- Updated office hours in a calendar to make sure we are all synced up

Image so far

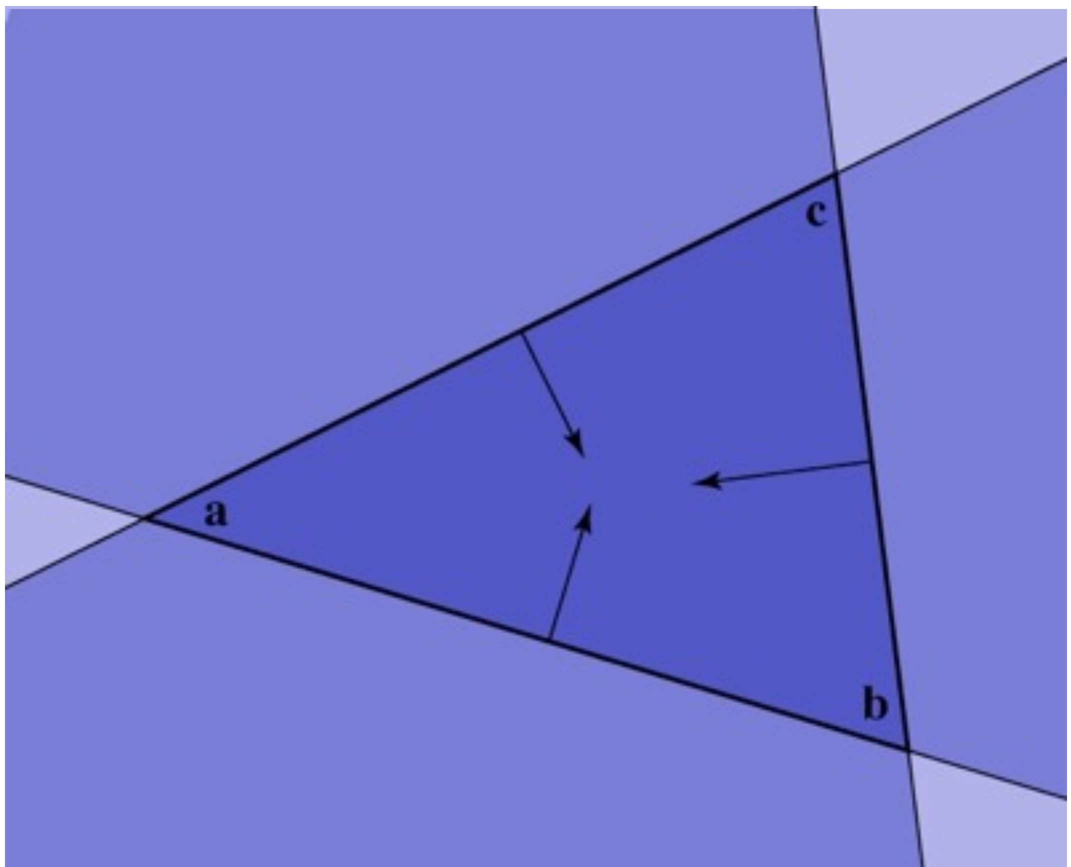
- With sphere intersection

```
Surface s = new Sphere((0.0, 0.0, 0.0), 1.0);
for 0 <= iy < ny
    for 0 <= ix < nx {
        ray = camera.getRay(ix, iy);
        bool didhit = s.intersect(ray, 0, +inf)
        if didhit
            image.set(ix, iy, white);
    }
```



Ray-triangle intersection

- In plane, triangle is the intersection of 3 half spaces



Ray-triangle intersection

- Condition 1: point is on ray

$$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$

- Condition 2: point is on plane

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

- Condition 3: point is on the inside of all three edges
- First solve 1&2 (ray–plane intersection)
  - substitute and solve for *t*:

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

$$t = \frac{(\mathbf{a} - \mathbf{p}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

Deciding about insideness

- Need to check whether hit point is inside 3 edges
  - easiest to do in 2D coordinates on the plane
- Will also need to know where we are in the triangle
  - for textures, shading, etc. … next couple of lectures
- Efficient solution: transform to coordinates aligned to the triangle

Barycentric coordinates

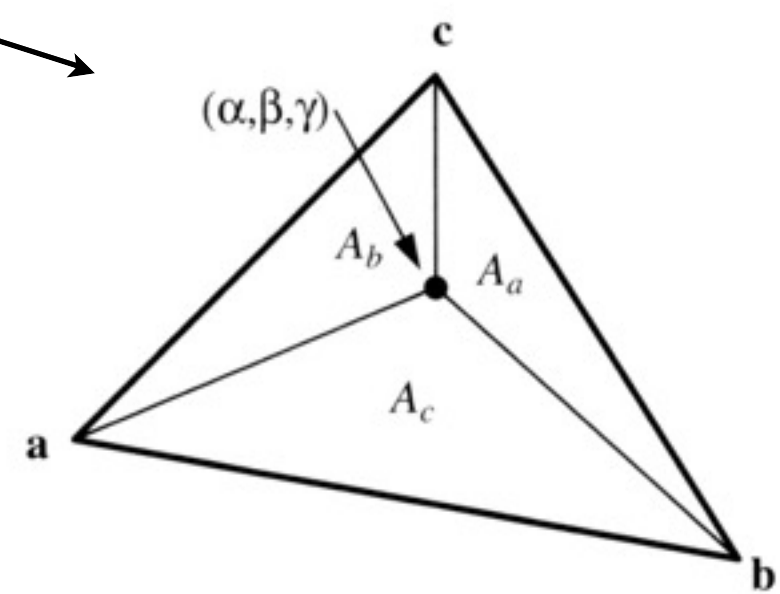
- A coordinate system for triangles
  - algebraic viewpoint:

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

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

  - geometric viewpoint (areas):
- Triangle interior test:

$$\alpha > 0; \quad \beta > 0; \quad \gamma > 0$$

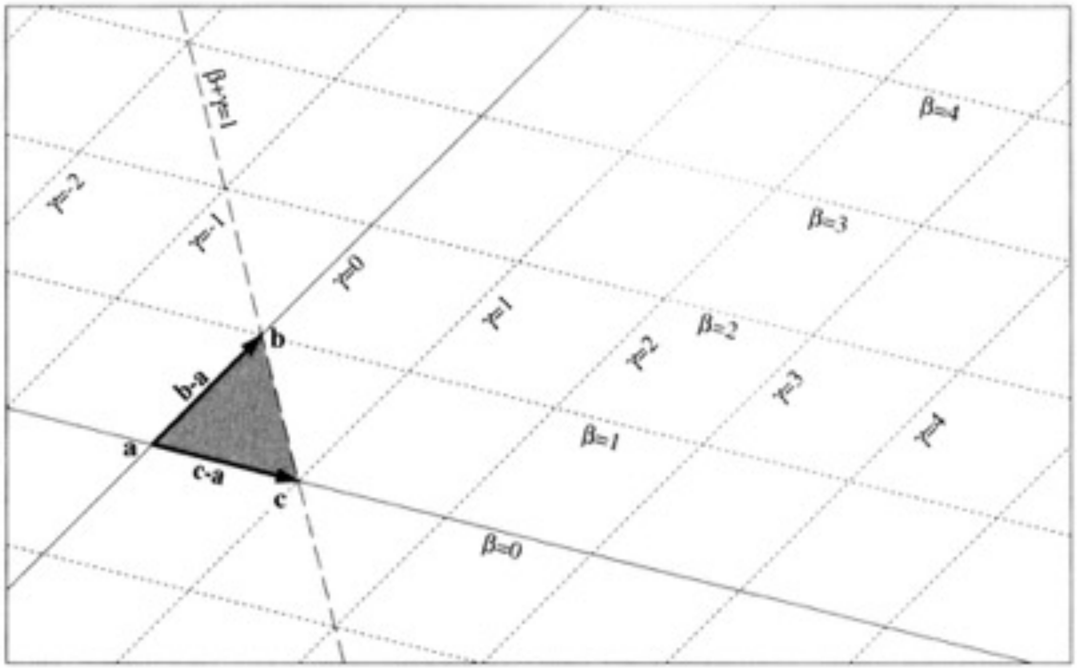


[Shirley 2000]

Barycentric coordinates

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

- Linear viewpoint: basis for the plane



[Shirley 2000]

  – in this view, the triangle interior test is just

$$\beta > 0; \quad \gamma > 0; \quad \beta + \gamma < 1$$

Barycentric ray-triangle intersection

- Every point on the plane can be written in the form:

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

  for some numbers β and γ.

- If the point is also on the ray then it is

$$\mathbf{p} + t\mathbf{d}$$

  for some number t.

- Set them equal: 3 linear equations in 3 variables

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

  …solve them to get t, β, and γ all at once!

Barycentric ray-triangle intersection

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

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

$$\begin{bmatrix} \mathbf{a} - \mathbf{b} & \mathbf{a} - \mathbf{c} & \mathbf{d} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} \mathbf{a} - \mathbf{p} \end{bmatrix}$$

$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_p \\ y_a - y_p \\ z_a - z_p \end{bmatrix}$$

Cramer's rule is a good fast way to solve this system

(see text Ch. 2 and Ch. 4 for details)

Ray intersection in software

- All surfaces need to be able to intersect rays with themselves.

ray to be intersected

```
class Surface {
    ...
    abstract boolean intersect(IntersectionRecord result, Ray r);
}
```

was there an intersection?

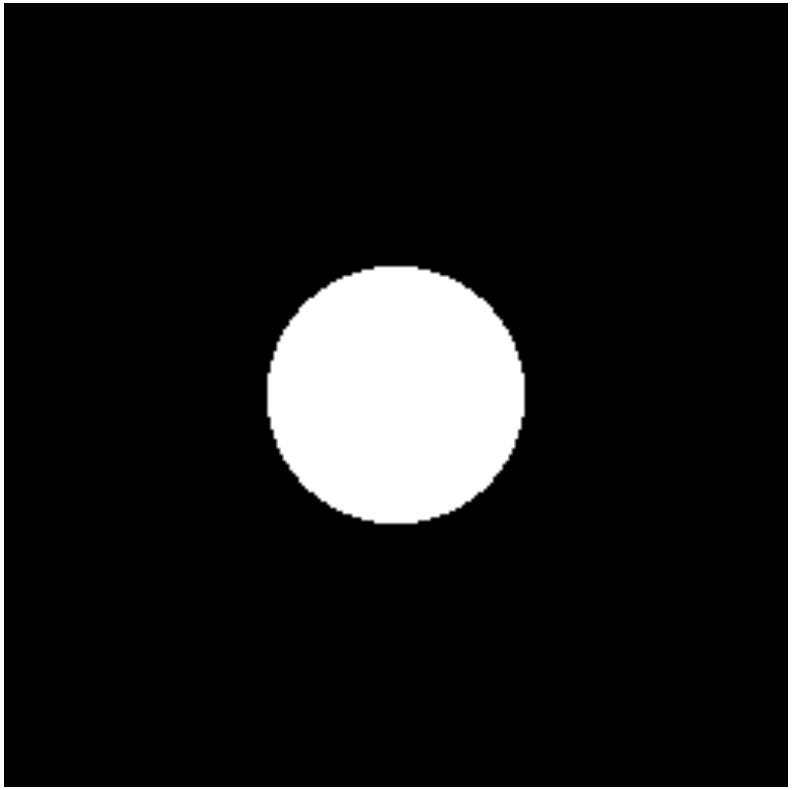
information about first intersection

```
class IntersectionRecord {
    float t;
    Vector3 hitLocation;
    Vector3 normal;
    ...
}
```

Image so far

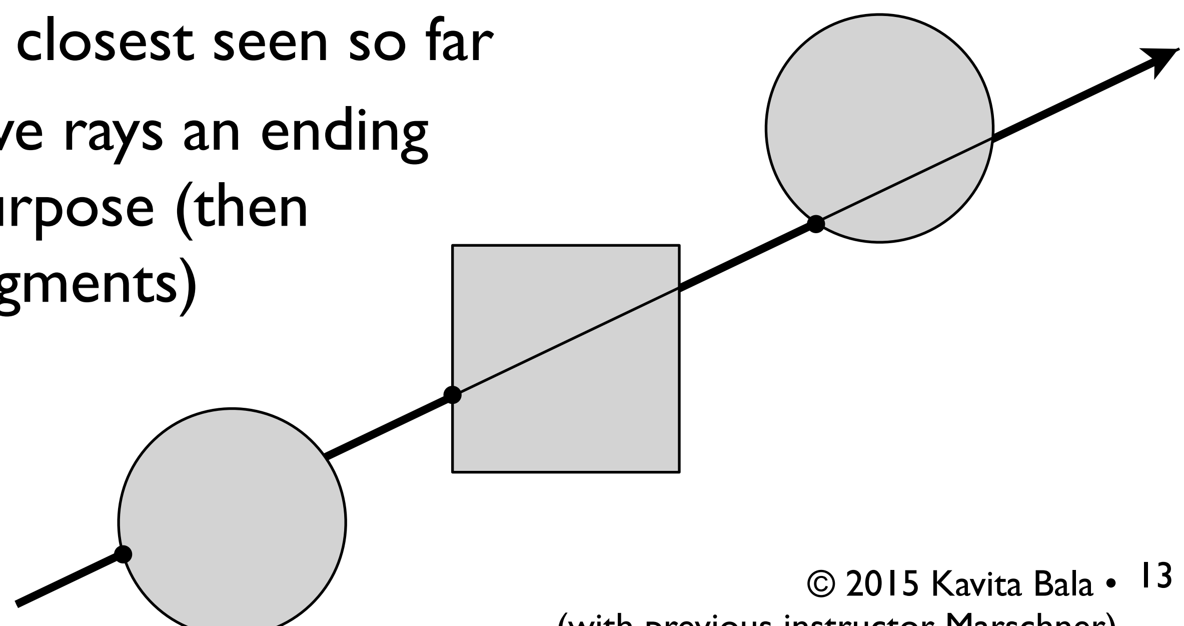
- With eye ray generation and sphere intersection

```
Surface s = new Sphere((0.0, 0.0, 0.0), 1.0);
for 0 <= iy < ny
    for 0 <= ix < nx {
        ray = camera.getRay(ix, iy);
        bool didhit = s.intersect(hit, ray)
        if didhit
            image.set(ix, iy, white);
    }
```



Ray intersection in software

- Scenes usually have many objects
- Need to find the first intersection along the ray
  - that is, the one with the smallest positive t value
- Loop over objects
  - ignore those that don't intersect
  - keep track of the closest seen so far
  - Convenient to give rays an ending t value for this purpose (then they are really segments)



Intersection against many shapes

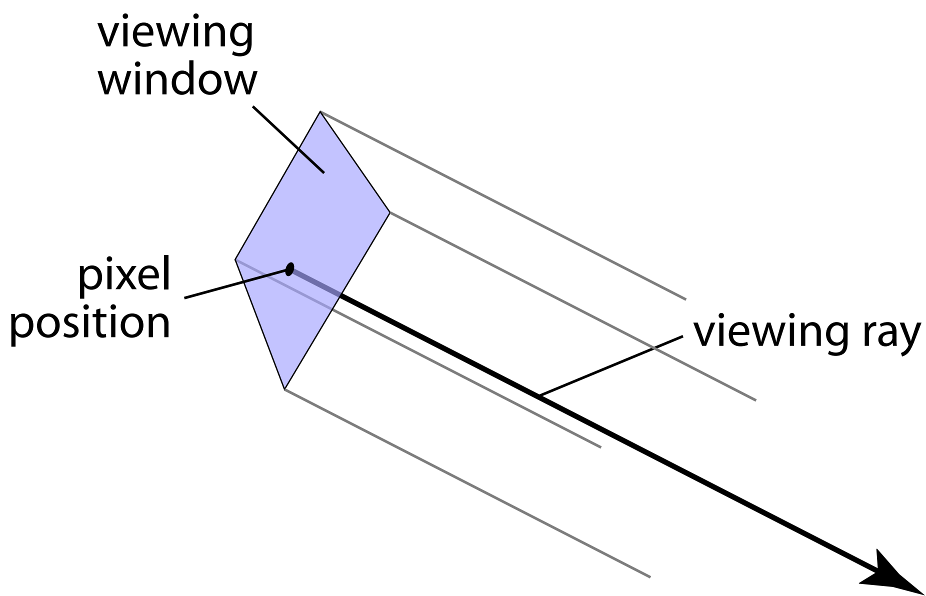
- The basic idea is:

```
intersect (ray, tMin, tMax) {
    tBest = +inf; firstSurface = null;
    for surface in surfaceList {
        bool didhit = surface.intersect(hit, ray, tMin, tBest);
        if didhit {
            tBest = hit.t;
            firstSurface = hit.Surface;
        }
    }
return firstSurface, tBest;
}
```

  - this is linear in the number of shapes
    but there are sublinear methods (acceleration structures)

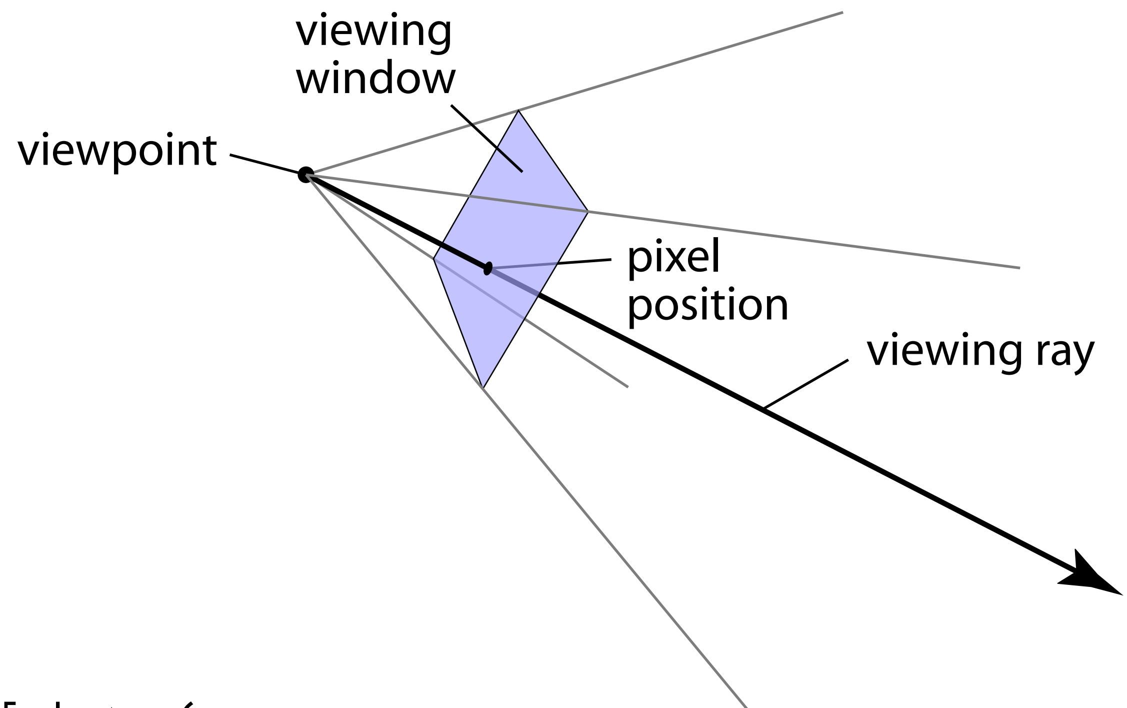
Generating eye rays—planar projection

- Ray origin (varying): pixel position on viewing window
- Ray direction (constant): view direction



Generating eye rays—perspective

- Ray origin (constant): viewpoint
- Ray direction (varying): toward pixel position on viewing window



Software interface for cameras

- Key operation: generate ray for image position

```
class Camera {
    ...
    Ray generateRay(int col, int row);
}
```

← args go from 0, 0 to width – 1, height – 1

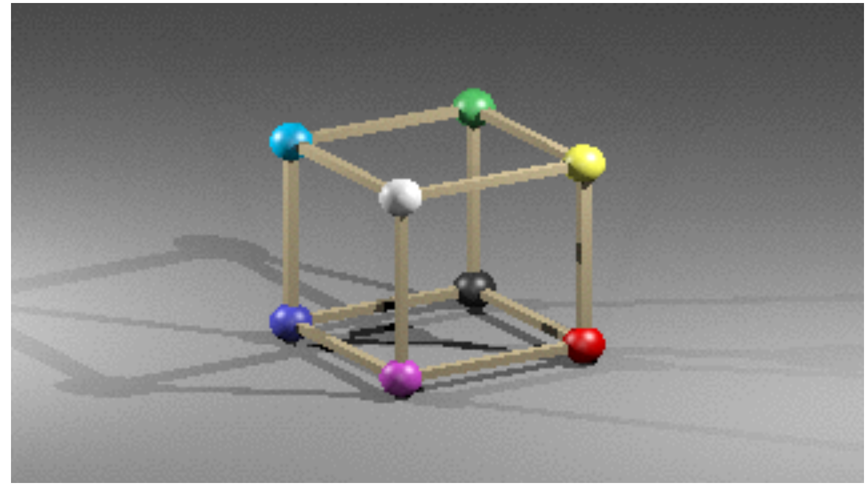
- Modularity problem: Camera shouldn't have to worry about image resolution
  – better solution: normalized coordinates

```
class Camera {
    ...
    Ray generateRay(float u, float v);
}
```

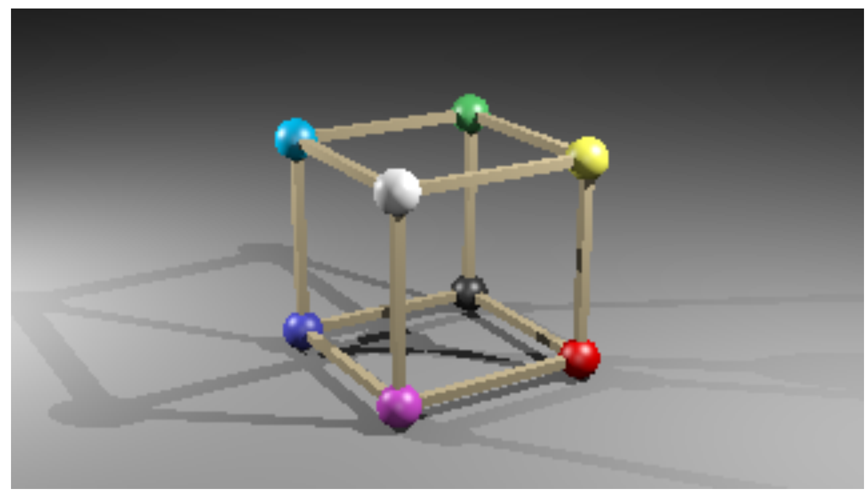
← args go from 0, 0 to 1, 1

Specifying views in Ray 1

```
<camera type="OrthographicCamera">
  <viewPoint>10 4.2 6</viewPoint>
  <viewDir>-5 -2.1 -3</viewDir>
  <viewUp>0 1 0</viewUp>
  <viewWidth>4</viewWidth>
  <viewHeight>2.25</viewHeight>
</camera>
```

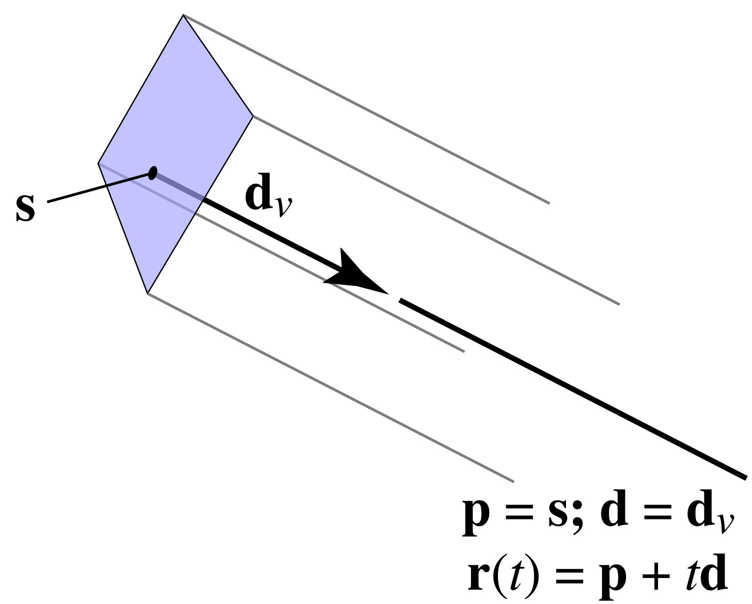


```
<camera type="PerspectiveCamera">
  <viewPoint>10 4.2 6</viewPoint>
  <viewDir>-5 -2.1 -3</viewDir>
  <viewUp>0 1 0</viewUp>
  <projDistance>6</projDistance>
  <viewWidth>4</viewWidth>
  <viewHeight>2.25</viewHeight>
</camera>
```



Generating eye rays—orthographic

- Just need to compute the view plane point **s**:



  – but where exactly is the view rectangle?

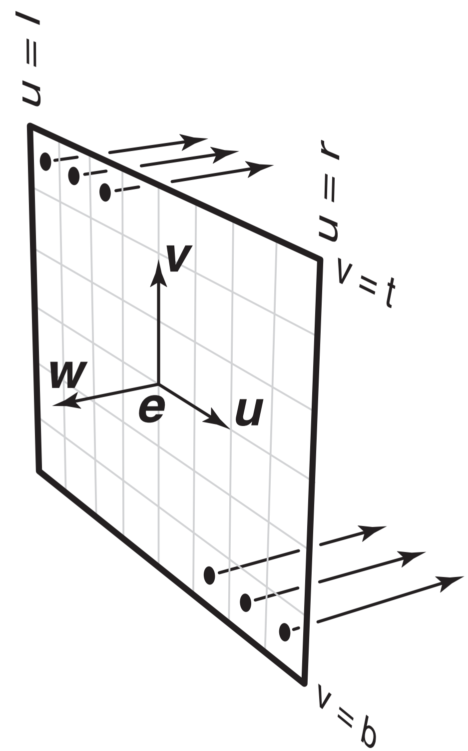
Generating eye rays—orthographic

- Positioning the view rectangle
  - establish three vectors to be *camera basis:* **u**, **v**, **w**
  - view rectangle is in **u**–**v** plane, specified by l, r, t, b
  - now ray generation is easy:

$$\mathbf{s} = \mathbf{e} + u\mathbf{u} + v\mathbf{v}$$

$$\mathbf{p} = \mathbf{s};\ \mathbf{d} = -\mathbf{w}$$

$$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$

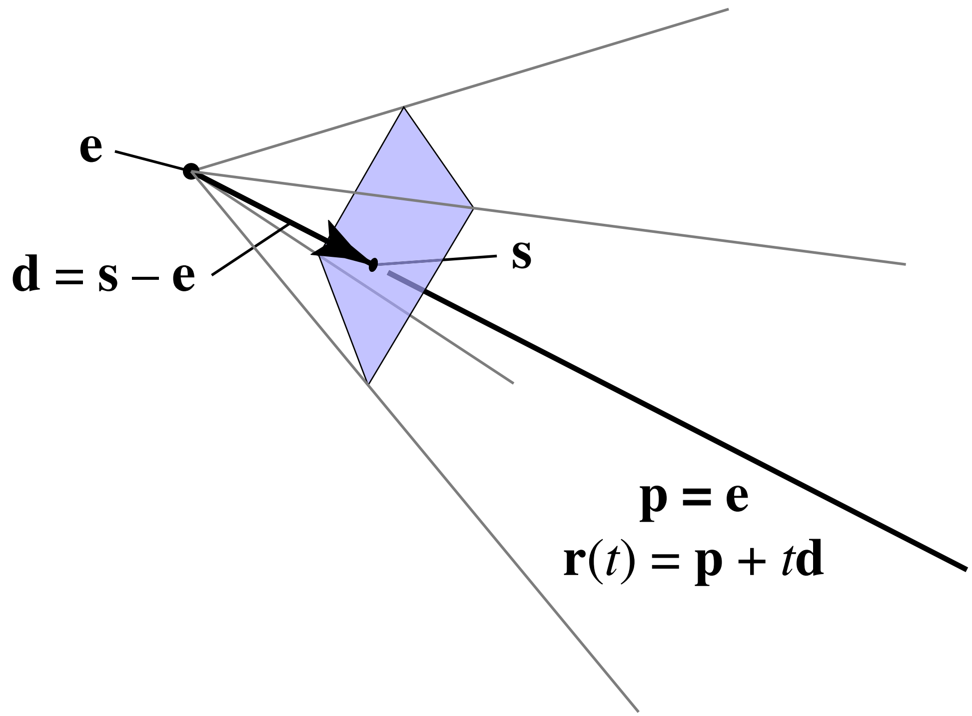


Camera

- Orthonormal bases
  - viewPoint == e
  - viewDir == -w, viewUp == v
    - Compute u from the above

Generating eye rays—perspective

- View rectangle needs to be away from viewpoint
- Distance is important: "focal length" of camera
  - still use camera frame but position view rect away from viewpoint
  - ray origin always **e**
  - ray direction now controlled by **s**



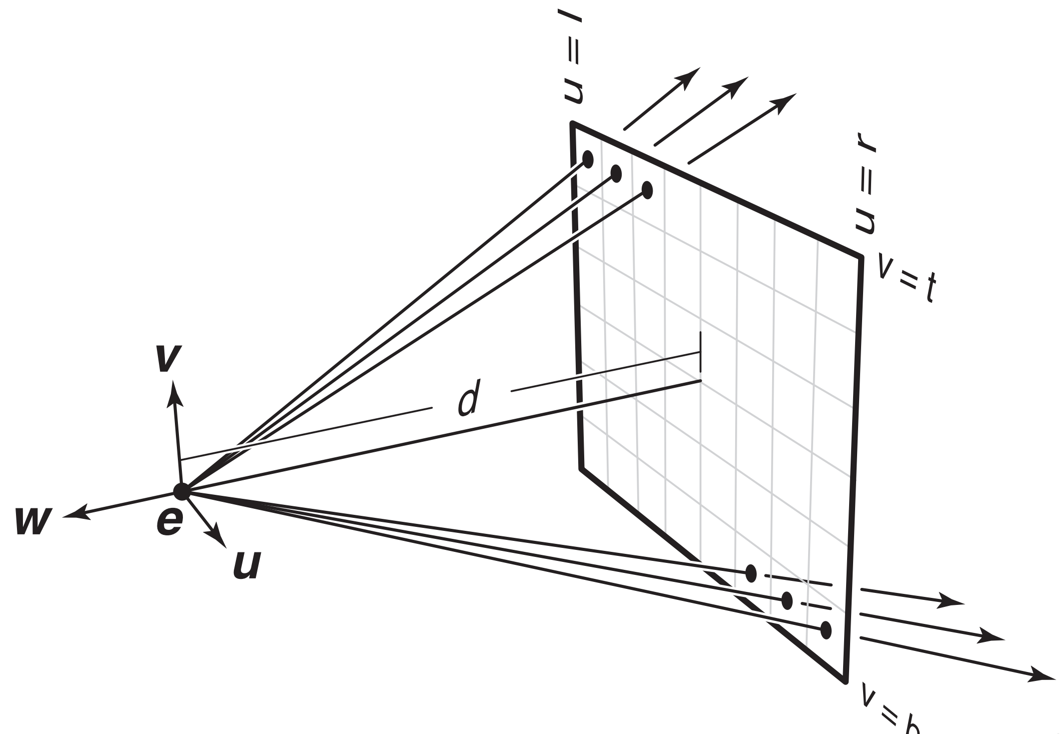
Generating eye rays—perspective

- Compute **s** in the same way; just subtract *d***w**
  - coordinates of **s** are (*u*, *v*, –*d*)

$$\mathbf{s} = \mathbf{e} + u\mathbf{u} + v\mathbf{v} - d\mathbf{w}$$

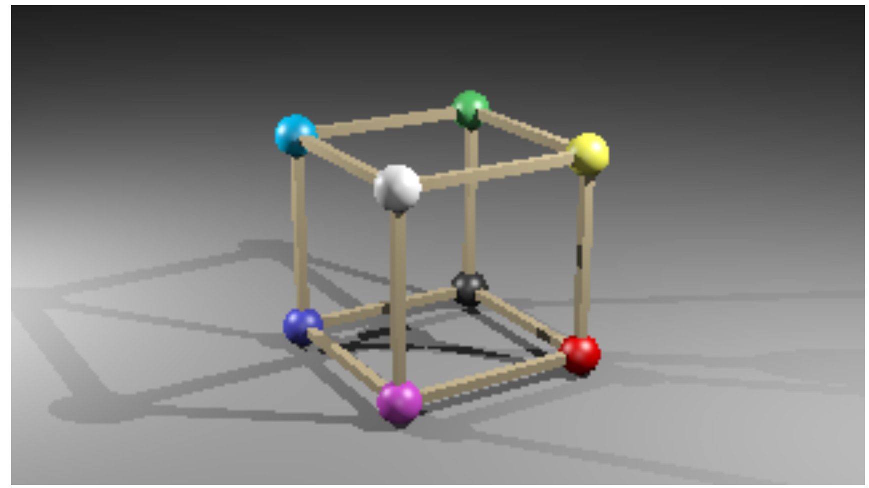
$$\mathbf{p} = \mathbf{e};\ \mathbf{d} = \mathbf{s} - \mathbf{e}$$

$$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$

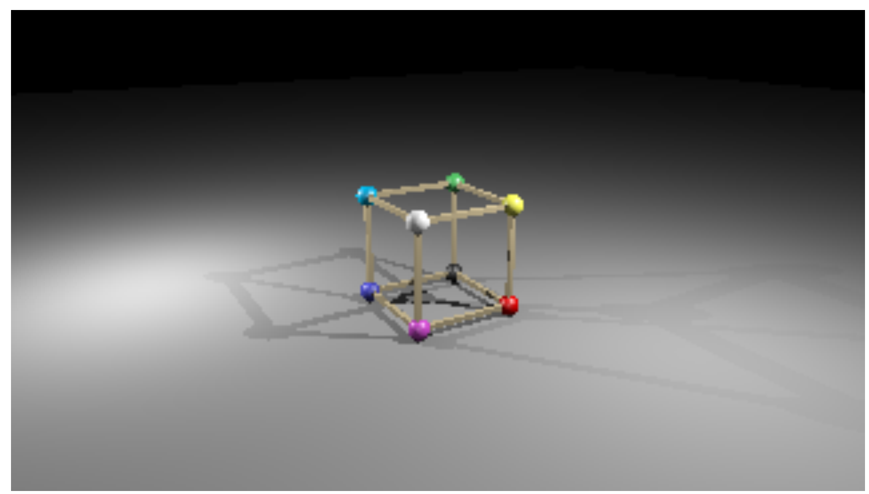


Specifying views in Ray 1

```
<camera type="PerspectiveCamera">
  <viewPoint>10 4.2 6</viewPoint>
  <viewDir>-5 -2.1 -3</viewDir>
  <viewUp>0 1 0</viewUp>
  <projDistance>6</projDistance>
  <viewWidth>4</viewWidth>
  <viewHeight>2.25</viewHeight>
</camera>
```



```
<camera type="PerspectiveCamera">
  <viewPoint>10 4.2 6</viewPoint>
  <viewDir>-5 -2.1 -3</viewDir>
  <viewUp>0 1 0</viewUp>
  <projDistance>3</projDistance>
  <viewWidth>4</viewWidth>
  <viewHeight>2.25</viewHeight>
</camera>
```



Camera

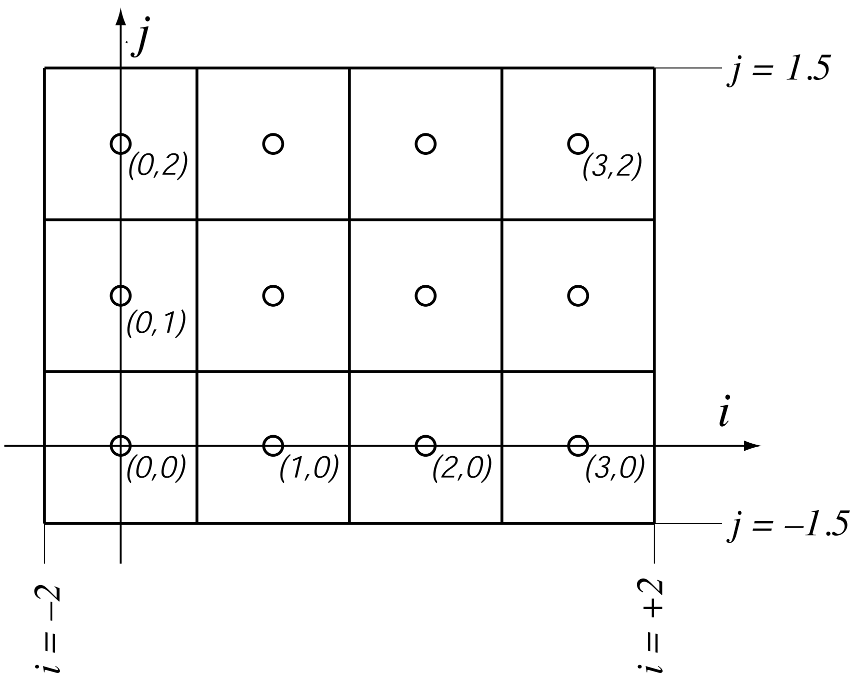
- Orthonormal bases
  - viewPoint == e
  - viewDir == -w, viewUp == v
    - Compute u from the above

l = -viewWidth/2

r = +viewWidth/2

n_x = imageWidth

Where are the pixels located?



$$u = l + (r - l)(i + 0.5)/n_x$$

$$v = b + (t - b)(j + 0.5)/n_y$$