
CSE 472

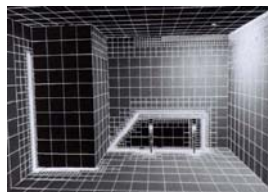
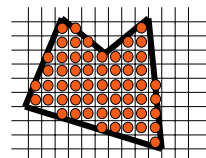
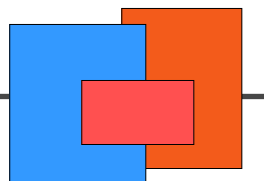
Ray Tracing

CSE 472 Spring 2009

1

Intro

- Scanline converter (+z-buffer)
- Painter's algorithm
- Radiosity
- PRT

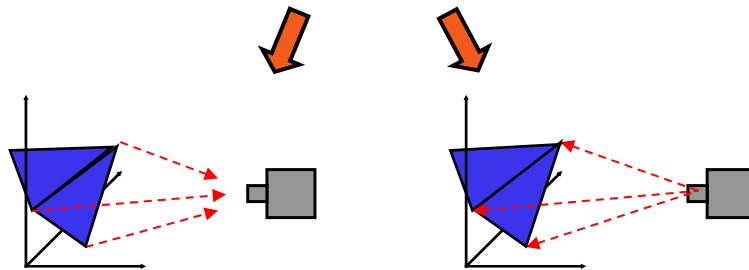


CSE 472 Spring 2009

2

Basic Rendering Model

Models for objects and cameras?



[Slusallek'05]

Rasterization:
Project geometry forward

Ray Tracing:
Project image samples backwards

Raytracing

Reflections (THE point of raytracing)

- Simply compute the color in the reflection direction

What does this do to the running time?

- Was $O(WHP)$



Ray Tracing

Rendering Methods

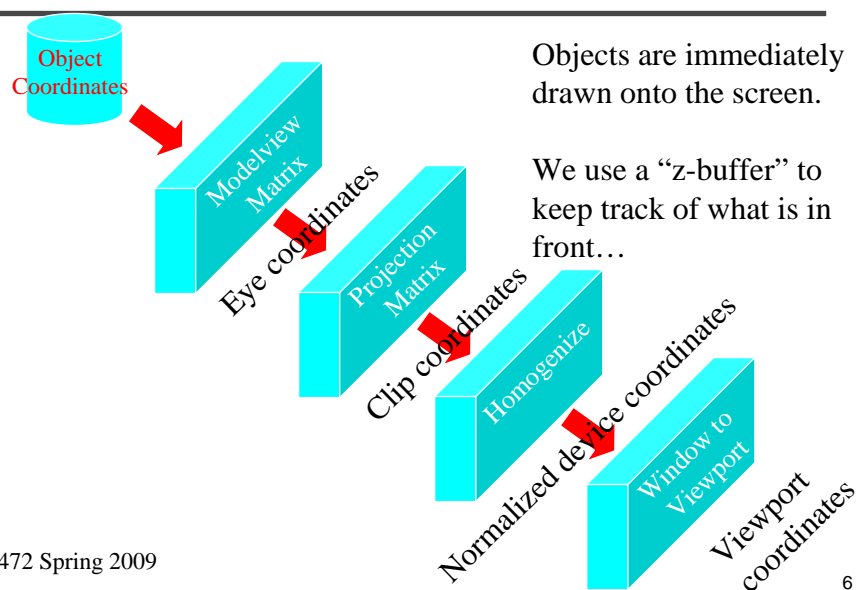
Rays

Rays for screen images

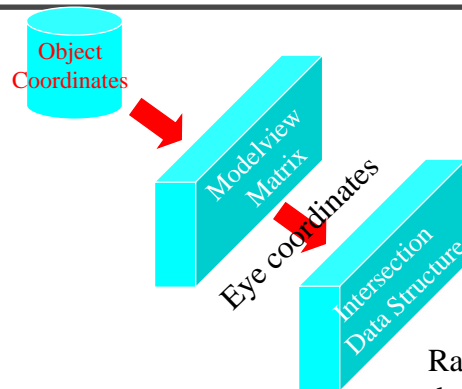
Shadows

Recursion

Z-Buffer Rendering



Visible Surface Raytracing



Raytracing then utilizes this data structure to build the image.

Raytracing

Invented by Arthur Appel

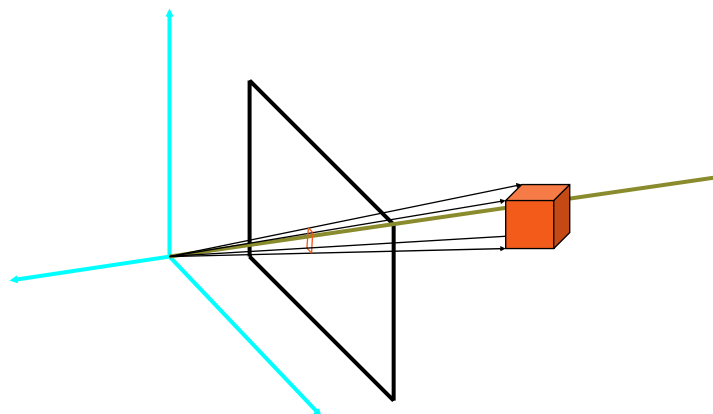
- 1968 IBM T.J. Watson Research Center.
- Not considered a major accomplishment since it was deemed impractical given its enormous computational requirements.
- Languished in obscurity until the late 1970's, when computer and display capabilities began to just barely catch up. SIGGRAPH'77 saw many significant developments, including the first looks at modeling reflection and shadows.

More History

A true recursive ray tracer was presented for the first time by Turner Whitted in the classic article “An improved illumination model for shaded display” in the June, 1980 Communications of the ACM.

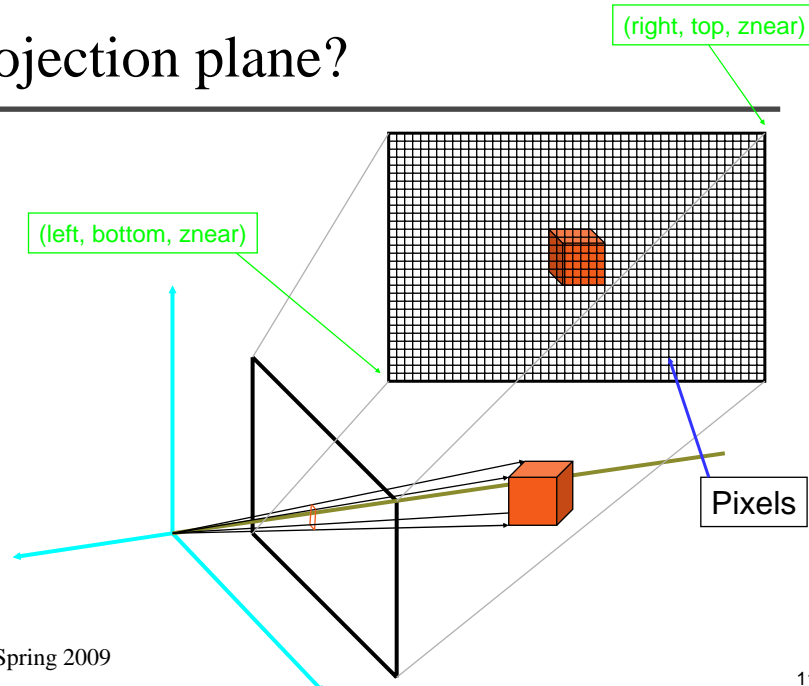
- Note that the article is more concerned with the model for illumination than the ray tracing algorithm.

That Pinhole Camera Model



After eyespace transformation
Projection Plane at $z=-d$

projection plane?



Raytracing

If we shoot a line from the center of projection through the center of a pixel and off into space...

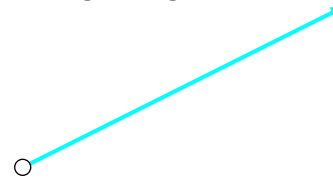
- What does it hit first?
- If it hits an object, compute the color for that point on the object
- That's the pixel color

Rays

A Ray is a *vector* and a *point*

- Point – The starting point of the ray
- Vector – The ray direction
- This describes an infinite line starting at the point and going in the ray direction

Starting point is sometimes called the ray origin



Simple Ray Class

```
class CRay
{
public:
    CRay(const CGrPoint &o, const CGrPoint &d) {m_o=o; m_d=d;}
    CRay() {}
    CRay(const CRay &r) {m_o = r.m_o; m_d = r.m_d;}

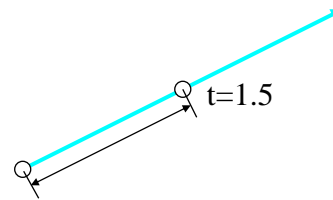
    const CGrPoint &Origin() const {return m_o;}
    const double Origin(int d) const {return m_o[d];}
    const CGrPoint &Direction() const {return m_d;}
    const double Direction(int d) const {return m_d[d];}
    CRay &operator=(const CRay &r) {m_o = r.m_o; m_d = r.m_d; return *this;}

private:
    CGrPoint m_o;           // Ray origin
    CGrPoint m_d;           // Ray direction
};
```

Rays

Ray t values

- A distance along the ray



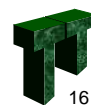
t values

Any point along the ray with origin (x_s, y_s, z_s) and direction (x_d, y_d, z_d) can be described using t:

- $x = x_s + t x_d$
- $y = y_s + t y_d$
- $z = z_s + t z_d$

- Or: $p = o + t d$

In vector math



Simple Ray Class

```
class CRay
{
public:
    CRay(const CGrPoint &o, const CGrPoint &d) {m_o=o; m_d=d;}
    CRay() {}
    CRay(const CRay &r) {m_o = r.m_o; m_d = r.m_d;}

    const CGrPoint &Origin() const {return m_o;}
    const double Origin(int d) const {return m_o[d];}
    const CGrPoint &Direction() const {return m_d;}
    const double Direction(int d) const {return m_d[d];}
    CRay &operator=(const CRay &r) {m_o = r.m_o; m_d = r.m_d; return *this;}

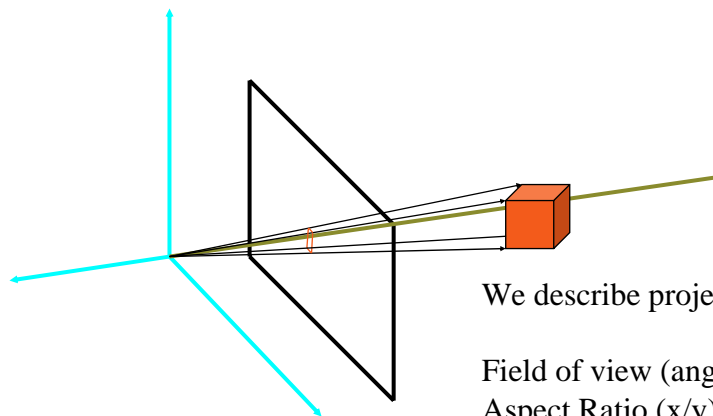
    CGrPoint PointOnRay(double t) const {return m_o + m_d * t;}

private:
    CGrPoint m_o;           // Ray origin
    CGrPoint m_d;           // Ray direction
};
```

CSE 472 Spring 2009

17

viewplane?



We describe projection with:

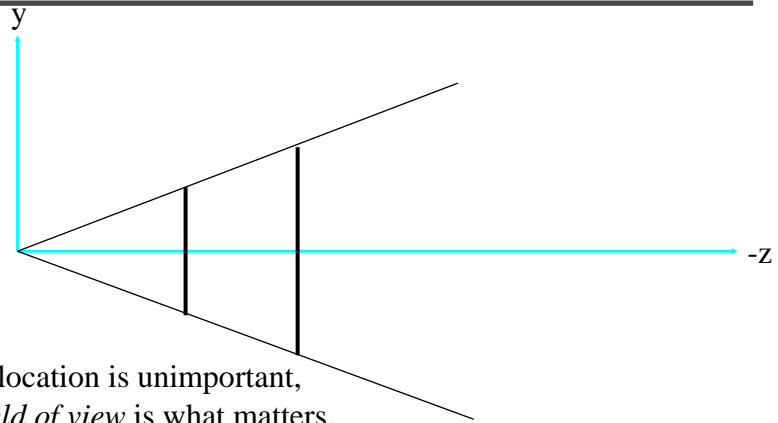
Field of view (angle)
Aspect Ratio (x/y)

Or a Frustum

CSE 472 Spring 2009

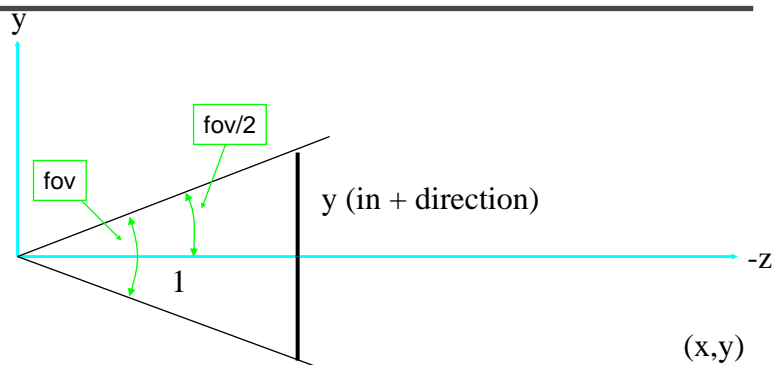
18

Viewplane z location

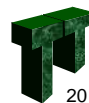


Z location is unimportant,
field of view is what matters.
We'll use $z=-1$
Could also use z_{near}

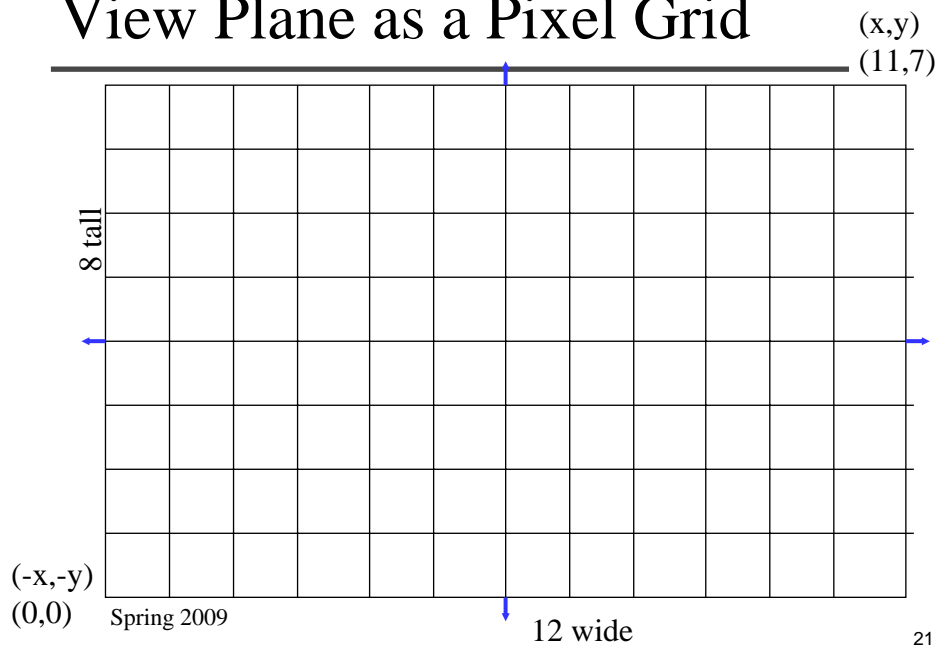
Viewplane size



$y = \tan(\text{fov}/2)$
height = $2y$
 $x = \text{aspect} * y$
width = $2x$



View Plane as a Pixel Grid

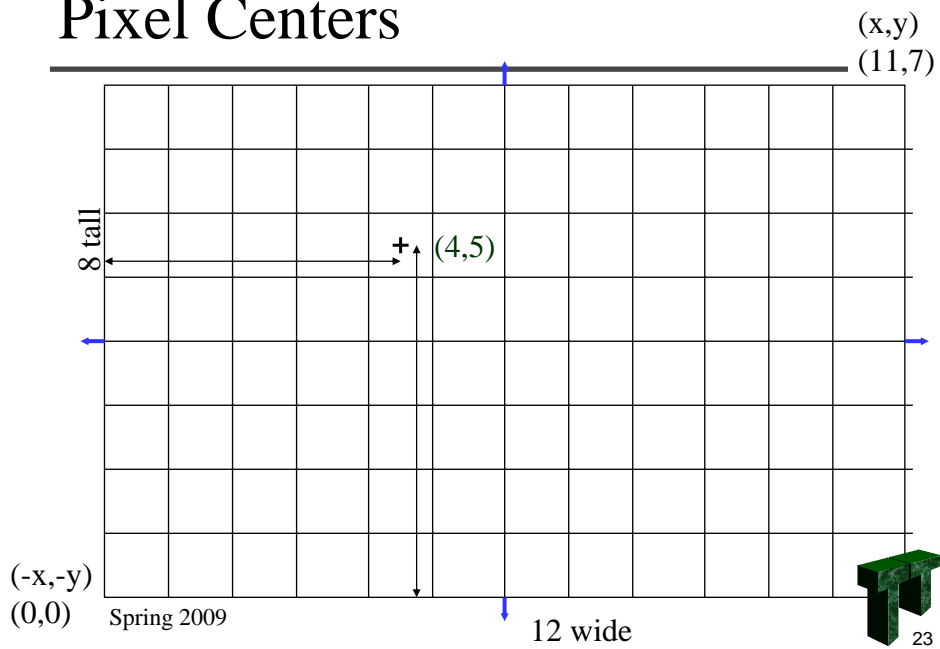


What we want

We want to shoot a ray

- Starting point $(0,0,0)$ (center of projection)
- Direction through the center of a pixel
- What we need is the coordinates of the center of a pixel

Pixel Centers



Visible Surface Raytracing

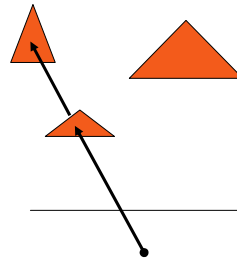
```

// width – Screen width, height – Screen height
// fov – Field of view in degrees, aspect – Aspect ratio
double ey = tan(fov / 2. * DEGTORAD);
double eh=2 * ey;
double ex = ey * aspect;
double ewid = 2 * ex;

for(int r=0; r<height; r++)
  for(int c=0; c<width; c++)
  {
    ray.start.Set(0,0,0);
    ray.dir.Set(-ex + ewid * (c + 0.5) / width,
               -ey + eh * (r + 0.5) / height,
               -1.);
    ray.dir.Normalize();

    // Compute color in ray direction
    // Save color as image pixel
  }

```



what will we hit?

Ray intersection tests

- Reduces to solving for t
 - $x = x_s + t x_d$
 - $y = y_s + t y_d$
 - $z = z_s + t z_d$
- Easiest is the sphere
- $(x-x_0)^2+(y-y_0)^2+(z-z_0)^2=r^2$
- Substitute ray equation in and solve for t

Ray intersection with a sphere

$$t_{1,2} = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

$$A = x_d^2 + y_d^2 + z_d^2$$

$$B = 2(x_d(x_s - x_0) + y_d(y_s - y_0) + z_d(z_s - z_0))$$

$$C = (x_s - x_0)^2 + (y_s - y_0)^2 + (z_s - z_0)^2 - r^2$$

Questions: Why two t values?

How do we tell if there is an intersection?

Which t value do we want?



Intersections with a plane

Plane equation

- $Ax+By+Cz+D=0$
- (A,B,C) is the plane normal
- We can compute the normal and, given a single point, compute D

$$t = -\frac{Ax_s + By_s + Cz_s + D}{Ax_d + By_d + Cz_d}$$

Questions: Can t be negative? Zero?
Why is this “more difficult” than a sphere?

CSE 472 Spring 2009

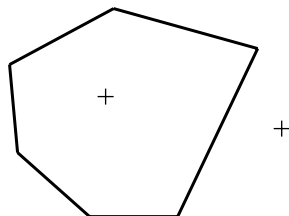


27

Polygon Intersection

Given a plane intersection, the question is:

- Is this point inside the polygon?



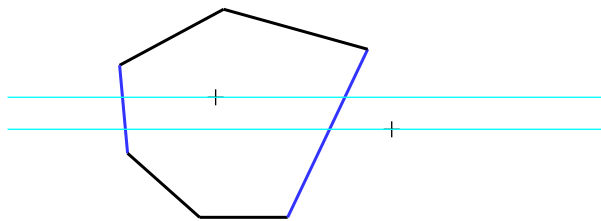
CSE 472 Spring 2009

28

Polygon Interior Tests

In a 2D projection:

- Find all edges that overlap in one dimension
- Determine intersection of a line parallel with an axis and the overlapping edges
- How many intersections are to the left of the point?



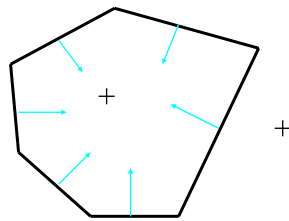
But, what projection?

Project the polygon for the maximum area

- What is the largest component of the normal?
- If x: Use a Y/Z projection
- If y: Use an X/Z projection
- If z: Use an X/Y projection
- Why?

Another Way...

Suppose you know the intersection and know a normal for each edge that points towards the inside of the polygon



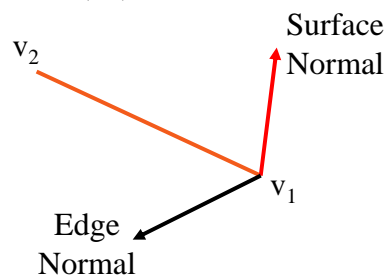
How do we compute these normals?

What is the characteristic of points inside the polygon?

Computing the Normals

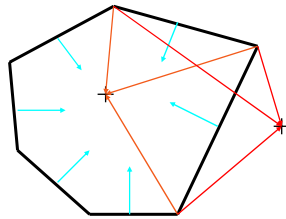
Let v_1, v_2 be two vertices (in counter clockwise order). Then the edge normal is $N \times (v_2 - v_1)$

- Orthogonal to the edge and the polygon surface normal (N).



Interior Test

For an edge v_1, v_2 with edge normal e_1 , the intersection p is on the inside side if $(p-v_1)e_1$ is not negative.



Faster Intersection Tests

Naïve ray tracer: For each ray, test each polygon to see if the ray hits it.

- If you did this, your projects would probably finish running after you graduate

Is there a better way?

- Ideas?

Faster Intersection Tests

Grouping objects into bounding boxes

- If we don't hit the bounding box, don't test inside.

Subdivide space into uniform boxes and walk the ray through the boxes

- Only test what's inside the boxes

Subdivide space into non-uniform boxes and walk the ray through the boxes

- Again, only test what's inside the boxes

Parts of a Ray Tracing System

Pixel loop

- Loops over the pixels and shoots the rays

Ray intersection system

- Determines what a ray hits

Ray color computation

- Determines the color a ray hits

So, how do we do?

$$\begin{aligned} I(\lambda) = & k_{sr} \sum_j I_j(\lambda) F_{sr}(\lambda, \theta_{r,j}) (\cos \theta_{r,j})^n && \text{Specular Reflection} \\ & && \text{from Light Sources} \\ & + k_{st} \sum_j I_j(\lambda) F_{st}(\lambda, \theta_{t,j}) (\cos \theta_{t,j})^{n'} && \text{Specular Transmission} \\ & && \text{from Light Sources} \\ & + k_{dr} \sum_j I_j(\lambda) F_{dr}(\lambda) (\mathbf{N} \cdot \mathbf{L}_j) && \text{Diffuse Reflection} \\ & && \text{from Light Sources} \\ & + k_{sr} I_{sr}(\lambda) F_{sr}(\lambda, \theta_R) T_r^{\Delta sr} && \text{Specular Reflection} \\ & && \text{from other surfaces} \\ & + k_{st} I_{st}(\lambda) F_{st}(\lambda, \theta_T) T_t^{\Delta st} && \text{Specular Transmission} \\ & && \text{from other surfaces} \\ & + k_{dr} I_a(\lambda) F_{dr}(\lambda) && \text{Ambient Light} \end{aligned}$$