Ray Intersection

Steve Rotenberg

CSE168: Rendering Algorithms

UCSD, Spring 2014

Ray Intersection

- At the heart of a ray tracer lies the ray intersection routine (or routines)
- Some ray tracers are based entirely on triangles and don’t support any other rendering primitive
- Even if the ray tracer only renders triangles, it is still likely that it will need to perform other ray intersection tests- for spatial data structures for example
- Therefore, it is practical to examine ray intersection tests with a few common primitives such as spheres, planes, boxes, and triangles

Ray Intersection

- Often when we test a ray with a surface, we don't just want to know if the ray intersects, but we want to know various other data about the surface such as:
  - Where the ray hit (position in space)
  - Distance along the ray (t)
  - Normal of the surface
  - Tangents of the surface
  - Texture coordinates of the surface
  - Material properties, etc.
- Sometimes, however, we perform ray intersects simply to determine if an object is shadowed by another and we don't need any detailed information
- Other times, we perform ray intersections with invisible bounding volumes used in spatial data structures and don't care about normals or textures

Ray Interval

- We think of a ray as starting at an origin and then shooting off infinitely in some direction

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

- Where **r**(t) is a function representing the ray, **p** is the ray origin, **d** is the ray direction, and t is the distance traveled along the ray
- Usually, when we're rendering, we're interested in finding the first surface hit along the ray's travel from the origin, or the intersection with the smallest value of t (but larger than 0)

Ray-Object Intersection

- We will say that our scene is made up of several individual ‘objects’
- For our purposes, we will allow the concept of an object to include primitives such as triangles and spheres, or even collections of primitives or other objects
- In order to be render-able, an object must provide some sort of ray intersection routine
- We will define a C++ base class object as:

```
class Object {
public:
    virtual bool IntersectRay(const Ray &ray, Intersection &hit);
};
```

- The idea is that we can derive specific objects, like triangles, spheres, etc., and then write custom ray intersection routines for them
- The ray intersect routine takes a ray as input, and returns true if the object is hit and false if it is missed
- If the object is hit, the intersection data is filled in into the Intersection class

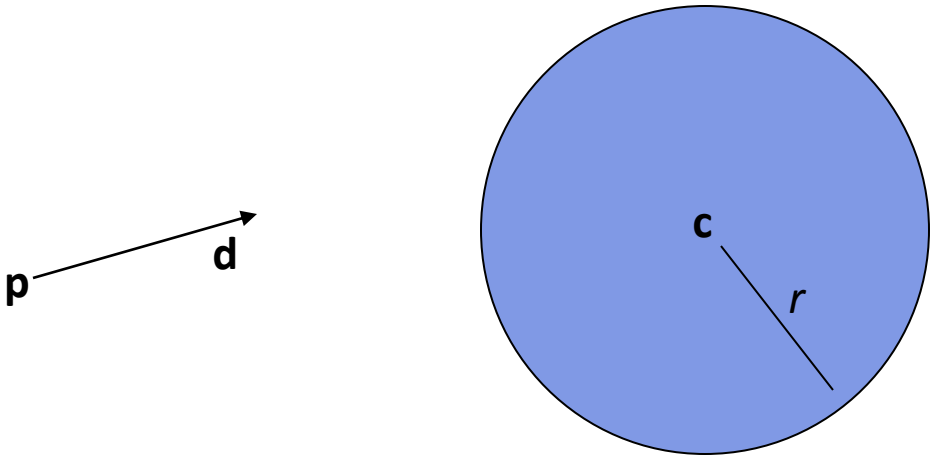
Sphere Intersection

Spheres

- Spheres are sometimes used as rendering primitives, but are most often used as bounding volumes
- At the minimum, a sphere needs a 3D vector for its position and a scalar for its radius

Ray-Sphere Intersection

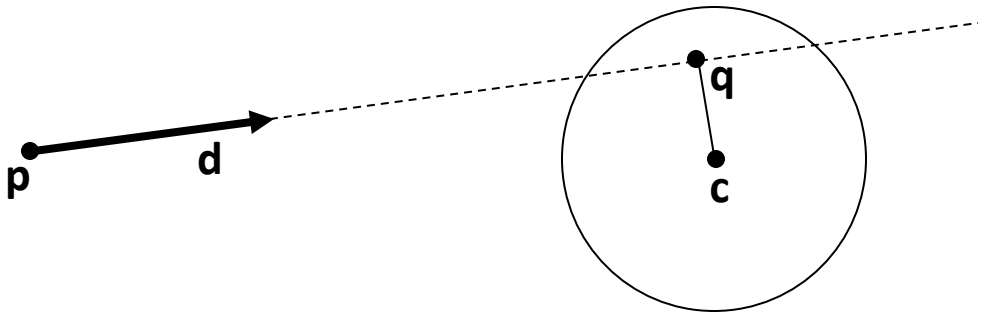
- Let's see how to test if a ray intersects a sphere
- The ray has an origin at point **p** and a unit length direction **d**, and the sphere has a center **c** and a radius *r*



Ray-Sphere Intersection

- The ray itself is the set of points $\mathbf{p}+t\mathbf{d}$, where $t \geq 0$
- We start by finding the point **q** which is the point on the ray-line closest to the center of the sphere
- The line **qc** must be perpendicular to vector **d**, in other words, $(\mathbf{q}-\mathbf{c})\cdot\mathbf{d}=0$, or $(\mathbf{p}+t\mathbf{d}-\mathbf{c})\cdot\mathbf{d}=0$
- We can solve the value of *t* that satisfies that relationship:

  $t=-(\mathbf{p}-\mathbf{c})\cdot\mathbf{d}$, so $\mathbf{q}=\mathbf{p}-((\mathbf{p}-\mathbf{c})\cdot\mathbf{d})\mathbf{d}$

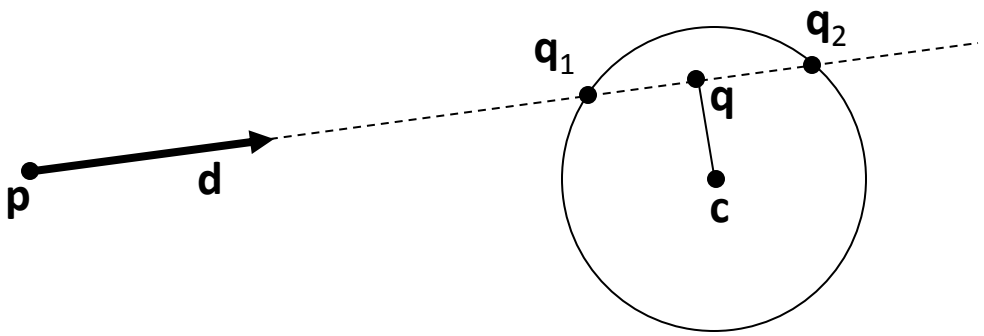


Ray-Sphere Intersection

- Once we have **q**, we test if it is inside the actual sphere or not, by checking if $|\mathbf{q}-\mathbf{c}| \leq r$ (actually its quicker to check if $|\mathbf{q}-\mathbf{c}|^2 \leq r^2$)
- If **q** is outside the sphere, then the ray must miss
- If **q** is inside the sphere, then we need find the actual point on the sphere surface that the ray intersects
- We say that the ray will hit the sphere at two points $\mathbf{q}_1$ and $\mathbf{q}_2$:

$$\mathbf{q}_1=\mathbf{p}+(t-a)\mathbf{d}) \qquad \mathbf{q}_2=\mathbf{p}+(t+a)\mathbf{d}) \qquad \text{where } a=\text{sqrt}(r^2-|\mathbf{q}-\mathbf{c}|^2)$$

- If $t-a \geq 0$, then the ray hits the sphere at $\mathbf{q}_1$, but if it is less than 0, then the actual intersection point lies behind the origin of the ray
- In that case, we check if $t+a \geq 0$ to test if $\mathbf{q}_2$ is a legitimate intersection



Ray-Sphere Intersection

- There are several ways to formulate the ray-sphere intersection test
- This particular method is popular and fast
- As a rule, one tries to postpone expensive operations, such as division and square roots until late in the algorithm when it is likely that there will be an intersection
- Ideally, quick tests can be performed at the beginning that reject a lot of cases where the ray is far away from the object being tested

Sphere Normal

- If we are using the sphere as an invisible bounding volume for a spatial data structure, then all we need to know is if the ray hits
- However, if we are actually rendering the sphere as geometry, then we'll need more information such as a normal and possibly texture coordinates
- To find the normal, we simply generate a vector from the sphere center to the intersection point and then normalize it:

$$\mathbf{n} = \frac{\mathbf{x} - \mathbf{c}}{r}$$

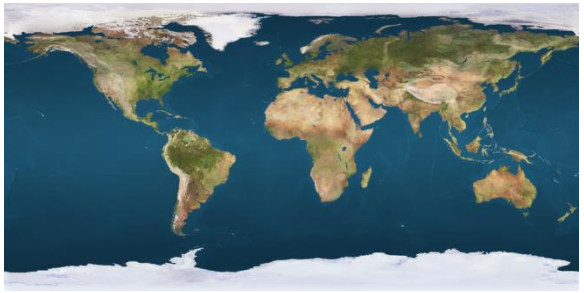
- Where **x** is the intersection point (either $\mathbf{q}_1$ or $\mathbf{q}_2$ from previous slide) and *r* is the sphere radius

Sphere Texture Coordinates

- If we are texture mapping the sphere, then we'll need texture coordinates
- There are many ways to texture map a sphere, but probably the most common is just by latitude and longitude
- We'll assume the 'north pole' is the top point of the sphere in the y direction
- We'll also assume that the 'prime meridian' (0 longitude) is at the –z side of the sphere and increases counterclockwise when looking down from above the sphere
- The texture coordinates (u,v) are:

$$u = \frac{\text{atan2}(n_x, n_z) + \pi}{2\pi}$$

$$v = \frac{\text{asin}(n_y) + 0.5\pi}{\pi}$$



Plane Intersection

Planes

- Infinite planes are useful in simple scenes, but are rarely used in realistic renderings
- Still, they are simple and useful enough to include in a ray tracer
- It is convenient to specify a plane by providing a single point **x** anywhere on the plane and a normal **n**
- This can be reduced to just a normal and a distance from the origin ($d=\mathbf{n}\cdot\mathbf{x}$), but generally users would rather specify an actual point, plus this can be used as a reference point to position a texture map on the plane

Ray-Plane Intersection

- A plane is defined by a normal vector **n** and a signed distance *d*, which is the distance of the plane to the origin
- We test our ray with the plane by finding the point **q** which is where the ray line intersects the plane
- For q to lie on the plane it must satisfy

$$d = \mathbf{q}\cdot\mathbf{n} = (\mathbf{p}+t\mathbf{d})\cdot\mathbf{n} = \mathbf{p}\cdot\mathbf{n} + t\mathbf{d}\cdot\mathbf{n}$$

- We solve for *t:*

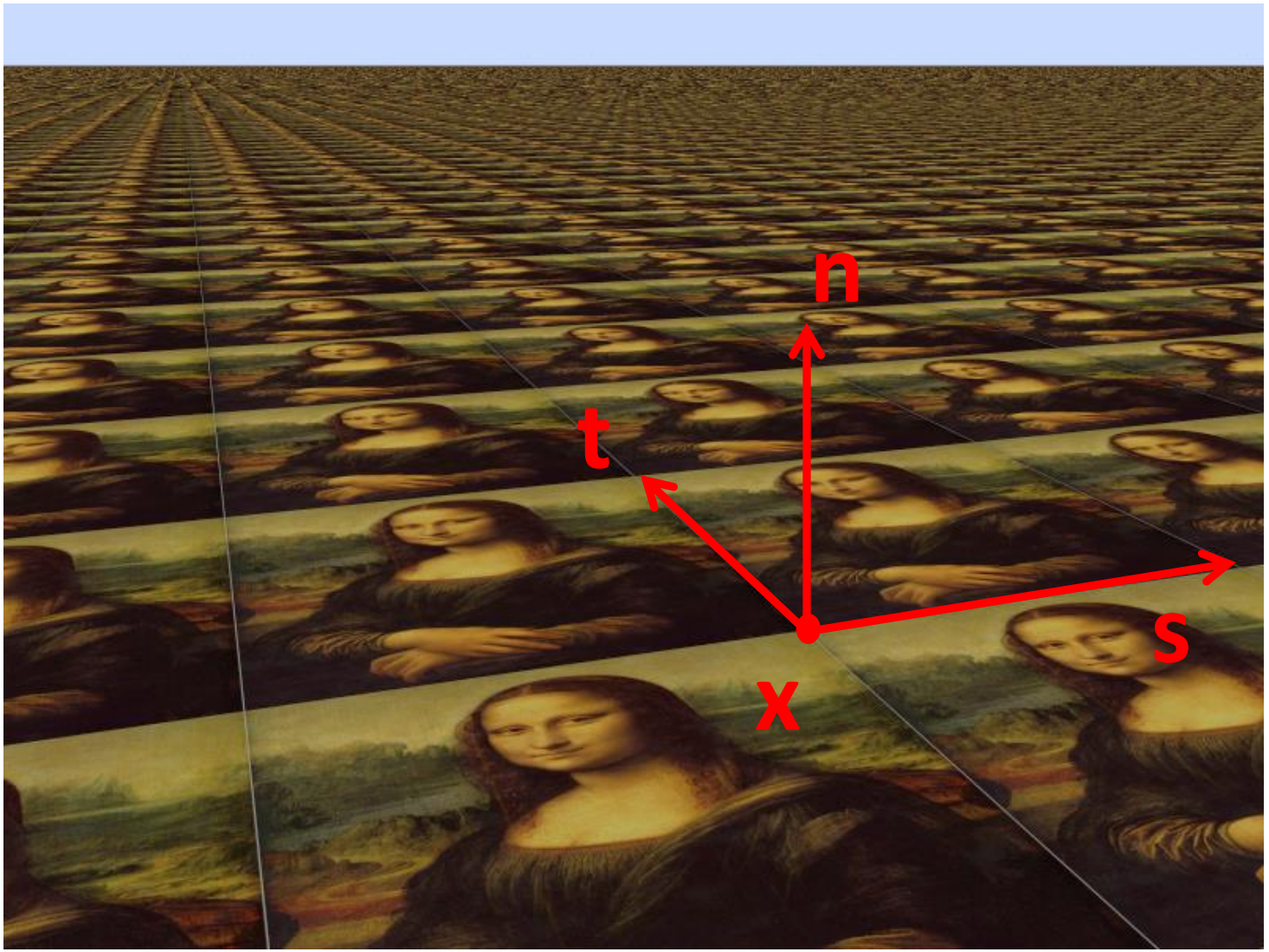
$$t = (d-\mathbf{p}\cdot\mathbf{n}) / (\mathbf{d}\cdot\mathbf{n})$$

- However, we must first check that the denominator is not 0, which would indicate that the ray is parallel to the plane
- If $t \geq 0$ then the ray intersects the plane, otherwise, the plane lies behind the ray, in the wrong direction
- NOTE: remember that *d* is the plane distance and **d** is the ray direction

Plane Texture Coordinates

- We already know the normal to the plane, so that's easy
- How do we get texture coordinates?
- We will assume that the texture tiles infinitely on the plane, but we still need some information about the orientation of the texture
- We will define two *texture basis vectors* **s** and **t** in the plane that orient the texture. Note that **s** and **t** don't have to be unit length or perpendicular to each other
- The reference point x on the plane can be used as the origin of the texture space (0,0)

Plane Texture Basis



Plane Texture Mapping

- We precompute **s**' and **t**':

  $\mathbf{z}=(\mathbf{s} \times \mathbf{t})$

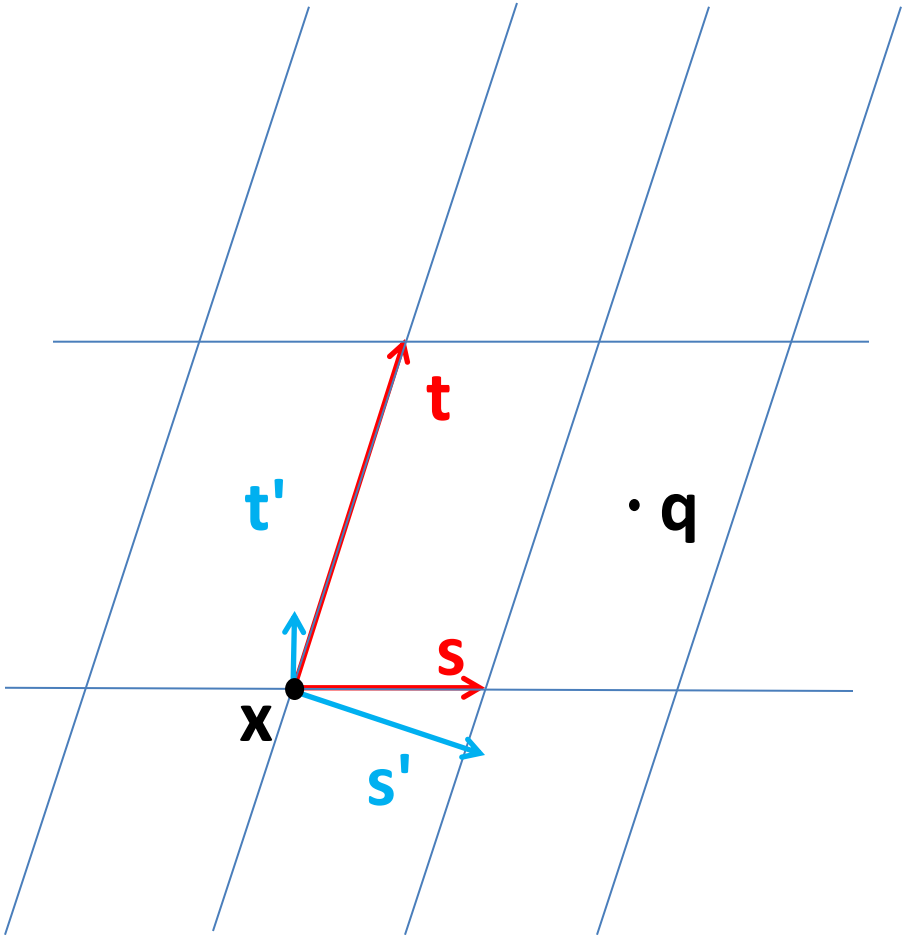
  $\mathbf{s}'=(\mathbf{t} \times \mathbf{z}) / |\mathbf{z}|^2$

  $\mathbf{t}'=(\mathbf{z} \times \mathbf{s}) / |\mathbf{z}|^2$

- We can then find the texture coordinates for a point **q**

  $u=(\mathbf{q}-\mathbf{x}) \cdot \mathbf{s}'$

  $v=(\mathbf{q}-\mathbf{x}) \cdot \mathbf{t}'$

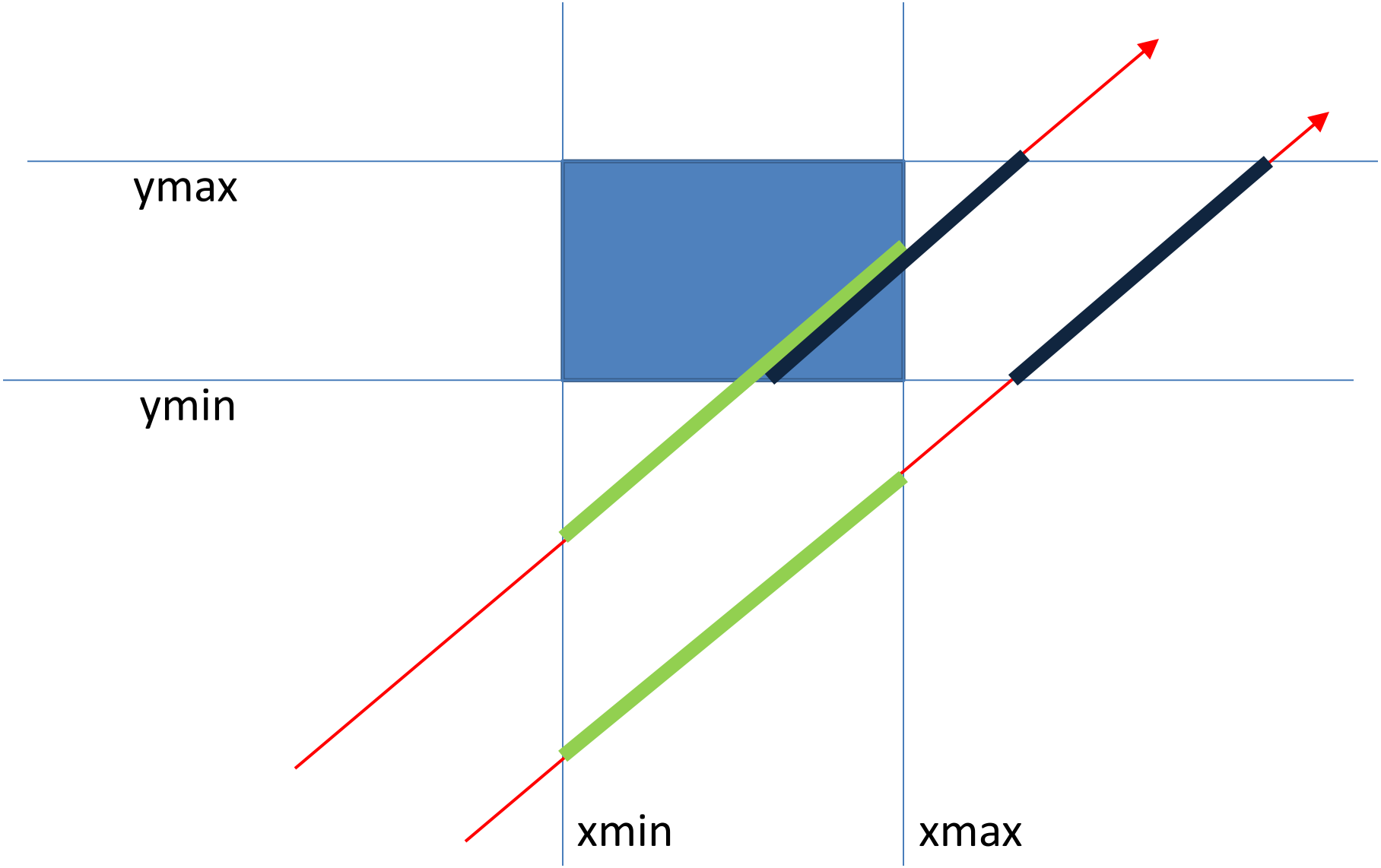


Axis Aligned Box Intersection

Axis Aligned Box

- An axis-aligned box is a box that lines up with the x, y, and z axes
- It can be defined by specifying the minimum and maximum corners (a total of 6 numbers)
- There are various approaches to computing a ray intersection with an axis aligned box
- Popular methods are based on the idea of testing the ray against 3 sets of parallel planes, and determining the interval (along the ray) where the planes are intersected
- If the intervals from x, y, and z overlap, then the ray intersects the box

Ray-AABB



Ray-AABB Test

$$t_{x1} = \frac{a_x - p_x}{d_x}, t_{x2} = \frac{b_x - p_x}{d_x}$$
$$t_{y1} = \frac{a_y - p_y}{d_y}, t_{y2} = \frac{b_y - p_y}{d_y}$$
$$t_{z1} = \frac{a_z - p_z}{d_z}, t_{z2} = \frac{b_z - p_z}{d_z}$$
$$t_{min} = \max\{\min(t_{x1}, t_{x2}), \min(t_{y1}, t_{y2}), \min(t_{z1}, t_{z2})\}$$
$$t_{max} = \min\{\max(t_{x1}, t_{x2}), \max(t_{y1}, t_{y2}), \max(t_{z1}, t_{z2})\}$$

- If $t_{min} \leq t_{max}$ then the ray intersects the box at $t = t_{min}$ (or at $t = t_{max}$ if $t_{min} < 0$). If $t_{max} < 0$, then the box is behind the ray and it's a miss.
- Note: **a** and **b** are the min and max corners of the box. **p** and **d** are the ray origin and direction, as usual

Axis Aligned Box

- Often times, boxes are not used as direct rendering primitives, but instead are used in spatial data structures
- In these cases, it is common that the ray will be tested against several (dozens or more) boxes, all aligned with the same xyz axes
- In these situations, some of the math in the intersection test can be precomputed to improve performance
- As the ray intersection test requires 3 divisions (one for each axis of the ray direction), it is possible to precompute these to save a bit of time
- Some ray tracers store the inverse of the direction vector directly with the ray itself
- The payoff isn't nearly as great on modern hardware as it was in the past, due to improved division hardware, pipelining, and better compiler scheduling

Box Normals and Texture Coordinates

- Most of the time, axis aligned boxes are used in spatial data structures such as box-trees, rather than as actual rendering primitives
- If they are used in spatial data structures, we really only need to know if the box is hit and how far along the ray it was hit- in other words, we don't care about the normal or texture coordinates
- If used for actual rendering primitives, then we would care about the normal and texture coordinates
- The normal is just going to be +/- **x**, **y**, or **z**, depending on which surface corresponds to t_{min} (or t_{max} if the ray starts inside the box)
- From there, we can just base the texture coordinates on the box dimensions

Triangle Intersection

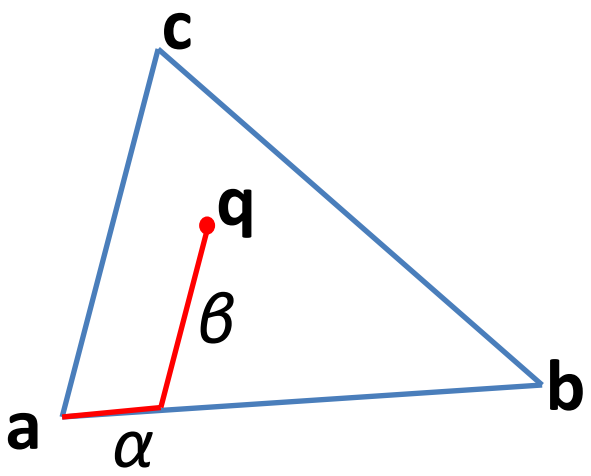
Triangles

- Triangles are the most common geometric primitive used in rendering, so the ray-triangle intersection test is critical
- Sometimes, we can treat triangles as only being one-sided, representing the outside of a surface
- However, if we're rendering transparent objects like glass, we need to treat triangles as two-sided

Barycentric Coordinates

- A triangle is defined by its three vertices: **a**, **b**, and **c**
- We can also define the *barycentric coordinates* α and β which can be used to uniquely specify a point **q** inside the triangle:

$$\mathbf{q} = \mathbf{a} + \alpha(\mathbf{b}-\mathbf{a}) + \beta(\mathbf{c}-\mathbf{a})$$



$0 < \alpha < 1$

$0 < \beta < 1$

$\alpha+\beta < 1$

Ray-Triangle Intersection

$$\mathbf{q} = \mathbf{a} + \alpha(\mathbf{b}-\mathbf{a}) + \beta(\mathbf{c}-\mathbf{a})$$

- To find an intersection, we substitute the ray equation for **q**:

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

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

- This is a linear system $\mathbf{Mx}=\mathbf{s}$ where:

$$\mathbf{M}=[-\mathbf{d}\ \ (\mathbf{b}-\mathbf{a})\ \ (\mathbf{c}-\mathbf{a})]$$

$$\mathbf{s}=[\mathbf{p}-\mathbf{a}]^T$$

$$\mathbf{x}=[t\ \ \alpha\ \ \beta]^T$$

Ray-Triangle Intersection

- M is a 3x3 matrix given by its column vectors. We can solve the linear system using Cramer's rule:

  $\det(\mathbf{M}) = -\mathbf{d} \cdot ((\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a}))$

  $t = (\mathbf{p}-\mathbf{a}) \cdot ((\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})) / \det(\mathbf{M})$

  $\alpha = -\mathbf{d} \cdot ((\mathbf{p}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})) / \det(\mathbf{M})$

  $\beta = -\mathbf{d} \cdot ((\mathbf{b}-\mathbf{a}) \times (\mathbf{p}-\mathbf{a})) / \det(\mathbf{M})$

- The solution is undefined if $\det(\mathbf{M})=0$, which happens when the ray is parallel to the plane
- Once we find t, α, and β, we need to verify that $\alpha>0$, $\beta>0$, $\alpha+\beta<1$ and $t>0$

Triangle Normal

- The triangle normal is just (**b**-**a**)x(**c**-**a**), which is already computed in the previous algorithm. If the ray hits, we can normalize this result
- Often times, we want to *smooth shade* the triangle by defining normals at the vertices and then interpolating them
- In this case:

$$\mathbf{n}=(1-\alpha-\beta)\mathbf{n}_a + \alpha\mathbf{n}_b + \beta\mathbf{n}_c$$

- For either smooth or flat shading, we should check to see that the normal points towards the ray origin. If not, we hit the back of the triangle and should flip the normal

Triangle Texture Coordinates

- We will assume that texture coordinates are defined at the three vertices, and so we can interpolate them the same way we did with the normals:

$$u=(1-\alpha-\beta)u_a + \alpha u_b + \beta u_c$$
$$v=(1-\alpha-\beta)v_a + \alpha v_b + \beta v_c$$

Moller-Trumbore Algorithm

- The triangle intersection we discussed is based on the popular Moller-Trumbore algorithm
- Look it up on the internet for more details
- By the way, Tomas Moller spent 6 months as a visiting researcher at UCSD back in 2004/2005

Ray Types

- Remember that rays tend to fall into different categories:
  - Primary rays (shot directly from the camera)
  - Shadow rays (testing to see if a light is blocked)
  - Reflection rays (additional rays bounced for shading purposes)
- For primary rays and reflection rays, we will want to find the first surface hit and we'll need a normal, texture coords, and possibly more
- For shadow rays, we only need to know if any surface is blocking the light. Once we find a surface we're done (we don't need to know the first surface hit and we don't need normals or anything else)
- Therefore, it is nice to add an additional field to the Ray class to indicate its type. Intersection routines can check this before computing additional information