Scenes & Ray Intersection

Steve Rotenberg
CSE168: Rendering Algorithms
UCSD, Winter 2017
‘Look-At’ Function

• It is often convenient to define the camera matrix using a ‘look-at’ function
• This takes a camera position, a target object position, and a world ‘up’ vector (typically the y-axis), and then builds a matrix:

\[
\begin{align*}
glm::vec3 \ d &= \text{position} \\
glm::vec3 \ c &= \text{glm}::\text{normalize}(d - \text{target}) \\
glm::vec3 \ a &= \text{glm}::\text{normalize}(\text{glm}::\text{cross}(\text{up}, c)) \\
glm::vec3 \ b &= \text{glm}::\text{cross}(c, a) \\
\text{glm}::\text{mat4x4} \ \text{cam}(a.x,a.y,a.z,0, b.x,b.y,b.z,0, c.x,c.y,c.z,0, d.x,d.y,d.z,1)
\end{align*}
\]

• Note: \( b \) will be length 1.0 automatically, being the cross product of two orthogonal unit vectors
• Also note that glm has a lookAt function, but theirs computes the ‘view’ matrix, which is the inverse of the ‘camera’ matrix
When we ray-trace an image, we start by generating rays at the camera and shoot them through each pixel into the scene.

For example, we have a loop something like this:

```cpp
Camera::Render() {
    int x,y;
    for(y=0; y<YRes; y++) {
        for(x=0; x<XRes; x++) {
            RenderPixel(x,y);
        }
    }
}
```
Camera Rays

• GL defines the ‘field of view’ of the camera to be the vertical angle of the view pyramid
• This is not a universal convention, as some people define the FOV as half of the angle, by the horizontal angle, or diagonal angle
• To be consistent, we’ll go with GL’s definition
• We can use this angle to generate our primary (camera) rays
• If we assume that the pixels are square (which is often the case), we can compute the horizontal FOV as:

\[ hfov = 2 \times \tan^{-1}\left( aspect \times \tan\left(\frac{vfov}{2}\right) \right) \]

NOTE: \( aspect \) is XRes/YRes
Camera Rays

- We start by ‘shooting’ rays from the camera out into the scene.
- We can render the pixels in any order we choose (even in random order!), but we will keep it simple and go from top to bottom, and left to right.
- We loop over all of the pixels and generate an initial primary ray (also called a camera ray or eye ray).
- The ray origin is simply the camera’s position in world space.
- The direction is computed by first finding location of a ‘virtual pixel’ on a ‘virtual image plane’, and then computing a normalized direction from the camera position to the virtual pixel.
Camera Rays

- Lets say our vertical resolution is 600 pixels
- The Y pixels are numbered from 0 to 599
- We want to shoot a ray through the center of each pixel, so we’ll add 0.5 to the x and y coordinates and then divide by the resolution to get the 0...1 position of the pixel
- We’ll then subtract 0.5 to get this in the -0.5 to 0.5 range:

  float fx=((float(x)+0.5f) / float(XRes)) – 0.5f;
  float fy=((float(y)+0.5f) / float(YRes)) – 0.5f;
Camera Rays

• We can then use the camera’s \(a\), \(b\), \(c\), & \(d\) vectors to compute the position of the virtual pixel, and then the ray direction

\[
\begin{align*}
a &= \text{glm}::\text{vec3}(\text{cam}[0]); & \text{// NOTE: glm stores matrices as column-major} \\
b &= \text{glm}::\text{vec3}(\text{cam}[1]); \\
c &= \text{glm}::\text{vec3}(\text{cam}[2]); \\
d &= \text{glm}::\text{vec3}(\text{cam}[3]); \\
scaleX &= 2\times\tan(hfov/2); \\
scaleY &= 2\times\tan(vfov/2); \\
\text{Ray.Origin} &= \mathbf{d}; \\
\text{Ray.Direction} &= \text{glm}::\text{normalize}\left(fx\times\text{scaleX}\times\mathbf{a} + fy\times\text{scaleY}\times\mathbf{b} - \mathbf{c}\right);
\end{align*}
\]
Scenes
Scenes

• The scene contains *lights* and *objects*
• Objects include *geometry, spatial data structures*, and *scene graph components*
• Geometry includes *triangles* (and/or other primitives) and *materials*
class Scene {
public:
    Scene();
    void AddObject(const Object &obj) {Objects.push_back(&obj);} 
    void AddLight(const Light &lgt) {Lights.push_back(&lgt);}  
    void SetSkyColor(const Color sky) {SkyColor=sky;} 

private:
    std::vector<const Object*> Objects;
    std::vector<const Light*> Lights;
    Color SkyColor;
};
Lights
Lights

• Lots of different things can emit light in the real world
• For rendering, we often start with simplified virtual lights such as point light sources or infinite directional lights
• For photoreal rendering, we can add area light sources, or allow light to be emitted off of triangles and arbitrary geometry
• Because we will want to support several types of light sources, it is a nice opportunity to use derived classes and virtual functions for lights
• We will create a base Light class and derive a PointLight and a DirectLight for now. Later, we will add more types
class Light {
public:
    Light() {
        Intensity=1.0;
        BaseColor=Color::White;
    }
    void SetBaseColor(const Color &col) {BaseColor=col;}
    void SetIntensity(float i) {Intensity=i;}
    virtual float Illuminate(const glm::vec3 &pos, Color &col, glm::vec3 &toLight, glm::vec3 &ltPos)=0;
protected:
    float Intensity;
    Color BaseColor; // Actual color is Intensity*BaseColor
};
Light Class Details

- It is convenient to assign a color and intensity as properties of the base Light.
- However, this doesn’t really restrict us to having light sources with a single color or intensity, since we can derive other types later.
- The BaseColor values are kept in the $0...1$ range, but the Intensity has no upper limit (it does have a lower limit of 0 though). The actual ‘color’ of the light is the product of the two ($\text{Intensity} \times \text{BaseColor}$), but it is nice to keep them separate to allow for easier tuning.
- The ‘Illuminate()’ function is the main function for a light source. It takes an input target position ‘pos’, and determines the intensity and color of light arriving at that position. The final color is returned in the ‘col’ field and limited to the $0...1$ range, while the intensity is the return value of the function itself.
- The Illuminate() function also sets a vec3 &ltPos, which is the position of the light source, and a normalized vec3 &toLight which points from pos to ltPos. Later, when we add support for shadows and area lights, the necessity for these will become apparent.
Point Lights

• We can start by creating a simple point light source
• We will derive PointLight off of the base Light
• The only new information it needs is a 3D position
• The point light will behave according to physics and will have an inverse square falloff of intensity with respect to distance
PointLight Class

class PointLight: public Light {
public:
    PointLight();
    float Illuminate(const glm::vec3 &pos, Color &col, glm::vec3 &toLight, glm::vec3 &ltPos) {
        toLight = Position - pos;
        float bright = Intensity / glm::length2(toLight); // Inverse square falloff
        toLight = glm::normalize(toLight);
        col = BaseColor;
        ltPos = Position;
        return bright;
    }

private:
    glm::vec3 Position;
};
Directional Lights

• We will also add a directional light to simulate very distant light sources such as the sun
• As the distance is assumed to be very large, we can ignore the effects of inverse square falloff, and we can have the intensity (and direction) be the same everywhere
class DirectLight: public Light {

public:
    DirectLight();
    float Illuminate(const glm::vec3 &pos, Color &col, glm::vec3 &toLight, glm::vec3 &ltPos) {
        toLight = -Direction;
        col = BaseColor;
        ltPos = pos - (1000000.0 * Direction); // Create virtual position
        return Intensity;
    }
    void SetDirection(glm::vec3 &dir) { Direction = glm::normalize(dir); }

private:
    glm::vec3 Direction;
};
Other Light Types

• Historically in computer graphics, people have defined several common light types such as point, directional, spot, area, projections, and others

• We could derive new light types if we want, but for now, we will stick with point & directional

• Later in the course, we will discuss area lights and some others
Objects
Objects

• The concept of an ‘Object’ is intended to include all of the things we might need to intersect rays with
• This includes geometry such as triangles and/or other primitives like spheres and planes
• It could potentially include more complex surface types such as curved surfaces, NURBS, subdivision surfaces, etc.
• Objects can also include spatial data structures used for optimization
• Objects can also include other scene management components, such as instances, which allow copies of other objects to be positioned and oriented with a matrix
class Object {
    public:
        virtual ~Object();
        virtual bool Intersect(const Ray &ray, Intersection &hit)=0;
};

NOTE: The virtual destructor ~Object() is necessary in order to allow for derived objects to have destructors. For example, if we derive a MeshObject that has to allocate arrays for vertices & triangles, it will need a destructor to delete that memory. Creating a virtual destructor in the base class allows higher level classes to overload the destruction routine.
Meshes

• Rather than derive the Triangle off of Object directly, we will use a MeshObject that contains a mesh of Triangles, as well as the supporting Vertexes and Materials

• This is also more efficient as it allows the Triangle::Intersect() routines to be called within a tight loop, and not have to use virtual function calls

• A mesh class will also allow us to add convenient routines such as MakeBox(), and LoadFile()
class MeshObject:public Object {
public:

    MeshObject();
    bool Intersect(const Ray &ray, Intersection &hit);

    bool Load(const char *filename);
    void MakeBox(float x, float y, float z);

private:

    int NumVertexes, NumTriangles, NumMaterials;
    Vertex *Vertexes;
    Triangle *Triangles;
    Material *Materials;
};
Materials

- Any actual rendered geometry (such as triangles) needs to have some sort of material information associated with it.
- Materials may have a wide variety of optical appearances from shiny, dull, transparent, opaque, translucent, multicolored, etc.
- In order to allow for a wide variety of material types, we will want to define a base class material and allow specific types to be derived from it.
- We will talk more about Materials in the next couple lectures.
Spatial Data Structures

• Earlier, we stated that Objects include geometry, spatial data structures, and scene graph data
• Of those, only geometry is actually visible
• Spatial data structures are invisible geometric data structures build around the geometry for the main purpose of speeding up ray intersection tests
• Some graphics people call them *acceleration structures*, but *spatial data structure* is a more general purpose term within computer science
• We will have a lecture on spatial data structures soon
Scene Graph Components

• Objects can also include scene graph components
• These are other components that describe how a scene is organized
• Ray tracers might use a couple different types of scene graph components, but the most important type is an instance
Instancing

• An *instance* is essentially a copy of an object
• For example, if we wanted to render a room with 100 chairs, we would store the chair model one time and build a scene containing 100 instances of that model
• An instance would typically contain a pointer to the object being instanced and a matrix to position the object in space
InstanceObject Class

Class InstanceObject: public Object {
public:

   InstanceObject();
   bool Intersect(const Ray &ray, Intersection &hit);
   void SetChild(Object &obj);
   void SetMatrix(glm::mat4x4 &mtx);

private:

   glm::mat4x4 Matrix;
   glm::mat4x4 Inverse;  // Pre-computed inverse of Matrix
   Object *Child;
};
Instancing

- Instancing works out particularly well with ray tracers
- Instead of using the instance matrix to transform the object into world space, we use the inverse of the matrix to transform the ray into the object’s space
- If we find an intersection (in object space), then we use the instance matrix to transform that intersection back into world space
- This way, we do a couple quick computations per ray rather than having to transform several copies of objects into world space
- To avoid having to generate a matrix inverse every time, we simply precompute that before we start rendering and store it with the instance

```cpp
bool Instance::Intersect(const Ray &ray, Intersection &hit) {
    Ray ray2;
    ray2.Origin = glm::vec3(MtxInverse * glm::vec4(ray.Origin, 1));
    ray2.Direction = glm::vec3(MtxInverse * glm::vec4(ray.Direction, 0));
    if (ChildObject->Intersect(ray2, hit) == false) return false;
    hit.Position = glm::vec3( Matrix * glm::vec4(hit.Position, 1) );
    hit.Normal = glm::vec3( Matrix * glm::vec4(hit.Normal, 0));
    hit.HitDistance = glm::distance(ray.Origin, hit.Position);  // Correct for any scaling
    return true;
}
```
Ray Intersection
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
  – Tangent vectors 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

\[ \mathbf{r}(t) = \mathbf{p} + td \]

• Where \( \mathbf{r}(t) \) is a function representing the ray, \( \mathbf{p} \) is the ray origin, \( \mathbf{d} \) 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)
Floating Point Roundoff

- It is common to spawn new rays off of an existing surface (for example, shadow rays, reflections, etc.)
- If the origin of the ray is ‘on’ the surface, we could run into some floating point roundoff issues, where the origin is actually a tiny amount above or below the surface
- If it is below the surface, then the spawned ray might actually intersect with the original surface itself, which is bad
- To fix this problem, it is common to ignore any ray intersections that are extremely close to the origin
- Therefore, if the ‘t’ value (distance) of the intersection is below some threshold (like 0.001), then we treat it as a ‘miss’, rather than a ‘hit’
- We’ll look at this again in a couple lectures when we talk about shadows
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 \( p + td \), 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, \((q-c) \cdot d = 0\), or \((p+td-c) \cdot d = 0\)
• We can solve the value of \( t \) that satisfies that relationship:
  \[
  t = -(p-c) \cdot d, \text{ so } q = p - ((p-c) \cdot d)d
  \]
Ray-Sphere Intersection

- Once we have \( q \), we test if it is inside the actual sphere or not, by checking if \( |q-c| \leq r \) (actually its quicker to check if \( |q-c|^2 \leq r^2 \))
- 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 \( q_1 \) and \( q_2 \):

\[
q_1 = p + (t-a)d \\
q_2 = p + (t+a)d \\
\text{where } a = \sqrt{r^2 - |q-c|^2}
\]

- If \( t-a \geq 0 \), then the ray hits the sphere at \( q_1 \), 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 \( q_2 \) 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
• 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 \( \mathbf{x} \) is the intersection point (either \( q_1 \) or \( q_2 \) 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:

\[
\begin{align*}
  u &= \frac{\text{atan2}(n_x,n_z)+\pi}{2\pi} \\
  v &= \frac{\text{asin}(n_y)+0.5\pi}{\pi}
\end{align*}
\]
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 \( \mathbf{x} \) anywhere on the plane and a normal \( \mathbf{n} \).
• 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
  \[
d = q \cdot n = (p + td) \cdot n = p \cdot n + td \cdot n
\]
- We solve for \( t \):
  \[
t = \frac{(d-p \cdot n)}{(d \cdot n)}
\]
- 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 \( t' \):
  
  \[
  z = (s \times t) \\
  s' = (t \times z) / |z|^2 \\
  t' = (z \times s) / |z|^2
  \]

- We can then find the texture coordinates for a point \( q \):
  
  
  
  \[
  u = (q-x) \cdot s' \\
  v = (q-x) \cdot t'
  \]
Axis Aligned Box Intersection
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

\[ t_{x1} = \frac{a_x - p_x}{d_x}, \quad t_{x2} = \frac{b_x - p_x}{d_x} \]

\[ t_{y1} = \frac{a_y - p_y}{d_y}, \quad t_{y2} = \frac{b_y - p_y}{d_y} \]

\[ t_{z1} = \frac{a_z - p_z}{d_z}, \quad t_{z2} = \frac{b_z - p_z}{d_z} \]

\[ t_{\text{min}} = \max\{\min(t_{x1}, t_{x2}), \min(t_{y1}, t_{y2}), \min(t_{z1}, t_{z2})\} \]

\[ t_{\text{max}} = \min\{\max(t_{x1}, t_{x2}), \max(t_{y1}, t_{y2}), \max(t_{z1}, t_{z2})\} \]

- If \( t_{\text{min}} \leq t_{\text{max}} \) then the ray intersects the box at \( t = t_{\text{min}} \) (or at \( t = t_{\text{max}} \) if \( t_{\text{min}} < 0 \)). If \( t_{\text{max}} < 0 \), then the box is behind the ray and it’s a miss.
- Note: \( \mathbf{a} \) and \( \mathbf{b} \) are the min and max corners of the box. \( \mathbf{p} \) and \( \mathbf{d} \) 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* \( \alpha \) and \( \beta \) which can be used to uniquely specify a point \( q \) inside the triangle:

\[
q = a + \alpha(b-a) + \beta(c-a)
\]

\( 0 < \alpha < 1 \)
\( 0 < \beta < 1 \)
\( \alpha + \beta < 1 \)
Ray-Triangle Intersection

\[ q = a + \alpha (b-a) + \beta (c-a) \]

- To find an intersection, we substitute the ray equation for \( q \):
  \[ p + td = a + \alpha (b-a) + \beta (c-a) \]
  \[ p - a = -td + \alpha (b-a) + \beta (c-a) \]

- This is a linear system \( \mathbf{Mx = s} \) where:
  \[ \mathbf{M} = \begin{bmatrix} -d & (b-a) & (c-a) \end{bmatrix} \]
  \[ \mathbf{s} = [p-a]^T \]
  \[ \mathbf{x} = [t \ \alpha \ \beta]^T \]
Ray-Triangle Intersection

- M is a 3x3 matrix given by its column vectors. We can solve the linear system using Cramer’s rule:
  \[ \det(M) = -d \cdot ((b-a) \times (c-a)) \]
  \[ t = (p-a) \cdot ((b-a) \times (c-a)) / \det(M) \]
  \[ \alpha = -d \cdot ((p-a) \times (c-a)) / \det(M) \]
  \[ \beta = -d \cdot ((b-a) \times (p-a)) / \det(M) \]
- The solution is undefined if \( \det(M) = 0 \), which happens when the ray is parallel to the plane.
- Once we find \( t, \alpha, \) and \( \beta \), we need to verify that \( \alpha > 0, \beta > 0, \alpha + \beta < 1 \) and \( t > 0 \).
Triangle Normal

• The triangle normal is just \((\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})\), 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

• I will post a link to the original paper on the class web page

• 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
Project 1
Project 1

• For project 1, create a basic ray tracer and render some boxes lit with some lights
• It should be able to run the sample code on the next slide, as well as some more complex samples listed on the web page (future projects will also run similar samples)
• It should be able to generate an exact match of the sample image in a few seconds
• I suggest you start with the base code provided, but you don’t have to if you don’t want to
• Compile with optimization on (in VisualStudio, this means setting it to release build)
• Due on Wednesday 4/12 by 5:00 pm
Sample Program

```c
main() {
    // Set up scene
    Scene scn;
    scn.SetSkyColor(Color(0.8, 0.9, 1.0));

    // Create box
    MeshObject box;
    box.MakeBox(0.5,0.2,0.3);
    scn.AddObject(box);

    // Create light
    PointLight redlgt;
    redlgt.SetBaseColor(Color(1.0, 0.2, 0.2));
    redlgt.SetIntensity(10.0);
    redlgt.SetPosition(Vector3(1.0, 2.0, 1.5));
    scn.AddLight(redlgt);

    // Create camera
    Camera cam;
    cam.LookAt(glm::vec3(0,2,5),glm::vec3(0,0,0),glm::vec3(0,1,0));
    cam.SetFOV(50.0); // NOTE: this is in degrees for UI purposes. Internally, it should be stored as radians
    cam.SetAspect(1.33);
    cam.SetResolution(800,600);

    // Render image
    cam.Render(scn);
    cam.SaveBitmap(“test.bmp”);
}
```