

Contents

1	Ray Tracing	3
1.1	Overview of Ray Tracing	3
1.2	The Ray	3
1.3	View Setup	4
1.4	Intersection Computations	5
1.4.1	Ray-Plane Intersection	6
1.4.2	Ray-Sphere Intersection	6
1.4.3	Ray-Triangle Intersection	9
1.4.4	Ray-Box Intersection	10
1.4.5	Triangle-Box Intersection	11
1.5	Advanced Geometry	13
1.5.1	Transformed Geometry	13
1.5.2	Subdivision Surfaces	13
1.5.3	Bicubic Patches	13
1.5.4	NURBS	13
1.6	Acceleration Structures	13
1.6.1	Bounding Box Hierarchies	14
1.6.2	Grids	14
1.6.3	BSP Trees	14
1.6.4	Other Acceleration Structures	15
1.7	Shading	15
1.7.1	A Basic Shading Model	16
1.7.2	Direct Illumination	16
1.7.3	Diffuse Reflection	16
1.7.4	Specular Reflection	16
1.7.5	Specular Refraction	16
1.7.6	Texturing	16
1.7.7	Bump Mapping	16
1.7.8	Displacement Mapping	16
1.7.9	Environment Mapping	16
	Index	18

Chapter 1

Ray Tracing

By Henrik Wann Jensen, revision April 27, 2004

Note that this text is work in progress. Please report errors to henrik@cs.ucsd.edu

Intro...

1.1 Overview of Ray Tracing

Diagram / flowchart

Camera

Primary rays

Shadow rays

Reflected rays

Refracted rays

1.2 The Ray

A ray, $\vec{r}(t)$, is specified by an origin, \vec{o} , and a direction vector, \vec{d} :

$$\vec{r}(t) = \vec{o} + t\vec{d}. \quad (1.1)$$

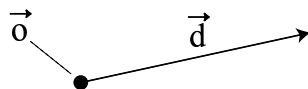


Figure 1.1: A ray is defined by its origin, \vec{o} , and a direction vector, \vec{d} .

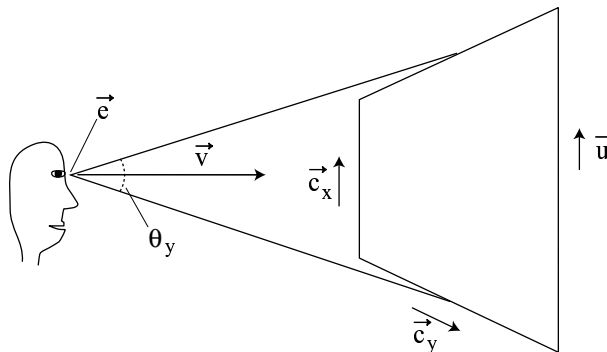


Figure 1.2: The view setup in ray tracing.

The free parameter, t , defines points along the ray. The direction vector, \vec{d} , is often normalized and then t is the distance to a point along the ray to the origin. In most situations we only care about information “in front” of the ray when $t > 0$. Figure 1.1 shows a ray.

1.3 View Setup

The most common viewing model in ray tracing is the pinhole camera. The pinhole camera assumes that all light passing through an infinitesimal hole is recorded onto a film. The pinhole camera handles perspective projection naturally, and one of the nice aspects of ray tracing is that the everything can be handled directly in world space without any transformations to camera space.

The view setup in ray tracing is needed to generate *primary rays* that start at the observer (camera or eye) position, \vec{e} , and move through a given location (pixel) in a viewing plane. This is illustrated in Figure 1.2.

To generate primary rays for a given pixel location we need a number of parameters specifying the view:

Eye position, \vec{e}

The location of the observer.

Viewing direction, \vec{v}

The direction that the observer is looking.

Up vector, \vec{u}

The orientation (up direction) of the view.

Field of view, θ

The angular spread of the view in the horizontal direction. Note that Figure 1.2 illustrates the field of view (θ_y) in the vertical direction.

and the parameters for the image are:

Width, w

The width in pixels of the image.

Height, h

The height in pixels of the image.

From these parameters we can compute the location of a virtual image plane in front of the observer as shown in Figure 1.2. This view is only used to compute ray directions and it does not restrict the visibility of objects to objects in front of the image plane. Normally, we use $t > 0$, which means that all objects directly in front of the observer location will be visible. The two vectors \vec{c}_x and \vec{c}_y that span the image plane in the horizontal respectively the vertical direction are computed as:

$$\vec{c}_x = \frac{\vec{v} \times \vec{u}}{\|\vec{v} \times \vec{u}\|_2} \quad (1.2)$$

$$\vec{c}_y = \vec{c}_x \times \vec{v}. \quad (1.3)$$

Note, that both \vec{c}_x and \vec{c}_y are normalized. We also need to compute the width, c_w , and the height, c_h , of the image plane:

$$c_w = 2 \tan\left(\frac{\theta}{2}\right) \quad (1.4)$$

$$c_h = c_w / a. \quad (1.5)$$

Here, $a = \frac{w}{h}$ is the aspect ratio of the image plane. Dividing c_h by a is necessary to ensure that pixels are square shaped in world space — if this is not the case then a rendered sphere will look like an ellipsoid rather than a round sphere.

The parameters for the viewing plane gives the necessary information to compute a primary ray, $\vec{r}_p(t)$, for a given pixel position (p_x, p_y) :

$$\vec{o} = \vec{e} \quad (1.6)$$

$$\vec{d} = \vec{v} + \left(\frac{p_x}{w} - \frac{1}{2}\right) c_w \vec{c}_x + \left(\frac{p_y}{h} - \frac{1}{2}\right) c_h \vec{c}_y \quad (1.7)$$

$$(1.8)$$

In addition it is often advantageous to normalize the ray direction before tracing the ray. The reason for this is that many ray intersection calculations can be optimized to take advantage of a unit length ray direction vector.

Mention orthographic projection as well....

1.4 Intersection Computations

The computation of intersections between a ray and objects in the scene is a fundamental operation in ray tracing. Visibility and shading is decided based on the intersection of rays with the scene, and ray intersections often account

for more than 90% of the compute time in a ray tracer, and it is important to ensure that the specific techniques used to compute ray intersections are efficient to prevent the ray tracer from being slow.

A general technique for computing the intersection between a ray, $\vec{r}(t)$ and an object is to define the object implicitly as:

$$f(\vec{x}) = 0 , \quad (1.9)$$

where, f is a function describing the object, and $f(\vec{x}) = 0$ for all points \vec{x} on the surface of the object. To find the intersection(s) with a ray we can substitute the ray equation, $\vec{r}(t)$, for \vec{x} and solve for t :

$$f(\vec{o} + t\vec{d}) = 0 , \quad (1.10)$$

If this equation has a valid solution, t_h , where $t_h > 0$ — meaning a valid intersection in front of the ray — then we can compute the hit location, \vec{h} , as:

$$\vec{h} = \vec{o} + t_h \vec{d} . \quad (1.11)$$

Furthermore, we often need the normal, \vec{n} , for shading and \vec{n} is computed based on the type of object that is intersected.

1.4.1 Ray-Plane Intersection

A plane is defined as

$$(\vec{p} - \vec{x}) \cdot \vec{n} = 0 , \quad (1.12)$$

where \vec{p} is a point in the plane, and \vec{n} is the normal to the plane. This is equation is true for all points \vec{x} in the plane.

To compute the intersection with the ray we substitute the ray equation for \vec{x} :

$$(\vec{p} - \vec{o} - t\vec{d}) \cdot \vec{n} = 0 . \quad (1.13)$$

Rearranging terms we get:

$$t = \frac{\vec{p} \cdot \vec{n} - \vec{o} \cdot \vec{n}}{\vec{d} \cdot \vec{n}} \quad (1.14)$$

Note, that t is undefined if the ray is parallel to the plane, since $\vec{d} \cdot \vec{n} = 0$. For the plane the hit location is computed using Equation 1.11 and the normal, \vec{n} , at the hit is always the normal to the plane.

1.4.2 Ray-Sphere Intersection

A sphere is defined by its center, \vec{c} , and the radius of the sphere R . All points \vec{x} on the surface of the sphere are given implicitly by the following equation:

$$(\vec{c} - \vec{x})^2 - R^2 = 0 , \quad (1.15)$$

where the square of the vector denotes a dot-product of the vector with itself. In the following we will describe two methods for computing the intersection(s) between a ray and a sphere.

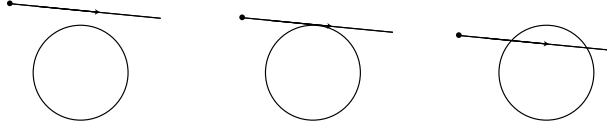


Figure 1.3: A ray can intersect a sphere 0 (left), 1 (middle), or 2 (right) times.

Algebraic method

The algebraic method uses the equations for the sphere and the ray to solve for an intersection between the two. This technique uses the implicit equation for the sphere and proceeds as previously described by inserting the equation for the ray in the equation for the sphere:

$$\begin{aligned} 0 &= (\vec{c} - \vec{o} - t\vec{d})^2 - R^2 \\ &= (\vec{c} - \vec{o}) \cdot (\vec{c} - \vec{o}) + (\vec{d} \cdot \vec{d})t^2 - 2((\vec{c} - \vec{o}) \cdot \vec{d})t - R^2. \end{aligned} \quad (1.16)$$

The resulting equation is quadratic and of the form:

$$at^2 + bt + c = 0, \quad (1.17)$$

where

$$a = \vec{d} \cdot \vec{d} \quad (1.18)$$

$$b = -2((\vec{c} - \vec{o}) \cdot \vec{d}) \quad (1.19)$$

$$c = (\vec{c} - \vec{o}) \cdot (\vec{c} - \vec{o}) - R^2, \quad (1.20)$$

This equation has either zero, one or two solutions depending on whether the ray misses the sphere, just strikes the sphere, or hits the sphere. This is shown in Figure 1.3. The solution for t is:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (1.21)$$

The number of intersections are:

$$\text{intersections} = \begin{cases} 0 & \text{if } b^2 - 4ac < 0 \\ 1 & \text{if } b^2 - 4ac = 0 \\ 2 & \text{if } b^2 - 4ac > 0 \end{cases} \quad (1.22)$$

In the case of two intersections are found then the one with the smallest positive t value is used. This can be the second root of the quadratic equation when the ray starts inside the sphere.

Once a hit, t_h , has been found we can compute the hit location, \vec{h} , using Equation 1.11 and the normal, \vec{n} , at \vec{h} is computed as:

$$\vec{n} = \frac{\vec{h} - \vec{c}}{R} \quad (1.23)$$

Geometric method

Besides solving the equations directly as described in the previous section it is possible to determine the intersection of a ray and a sphere by analyzing the geometric configuration as shown in Figure 1.4.

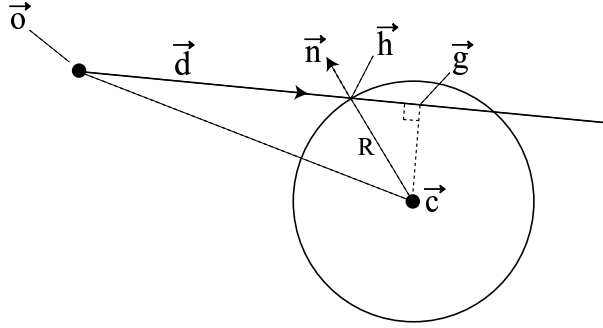


Figure 1.4: The geometry for the ray-sphere intersection method.

First we compute the vector, \vec{oc} , from the ray origin to the sphere center.

$$\vec{oc} = \vec{c} - \vec{o} \quad (1.24)$$

The squared distance between the ray and the sphere is:

$$\|oc\|^2 = \vec{oc} \cdot \vec{oc} \quad (1.25)$$

If $\|oc\|^2 > R^2$ then the ray origin is outside the sphere, otherwise it is inside. Next we compute the location, t_g , of the point on the ray \vec{g} that is closest to the center of the sphere:

$$t_g = \vec{oc} \cdot \vec{d} \quad (1.26)$$

This value gives us the first opportunity to check if the ray misses the sphere. If $t_g < 0$ and the ray is outside the sphere then the sphere is behind the ray, and there is no intersection. Otherwise, we continue and compute the squared distance, $\|gc\|^2$, from the point, \vec{g} , on the ray to the sphere center using Pythagoras rule (as shown in Figure 1.4):

$$\|gc\|^2 = \|oc\|^2 - t_g^2 \quad (1.27)$$

Next we can use Pythagoras rule on the smaller triangle shown in Figure 1.4 to compute the squared distance, $\|hg\|^2$, from \vec{g} to the hit location, \vec{h} :

$$\|hg\|^2 = R^2 - \|gc\|^2 \quad (1.28)$$

If $\|hg\|^2 < 0$ then the ray misses the sphere, otherwise t is computed as:

$$t = t_g - \sqrt{\|hg\|^2}, \quad (1.29)$$

and we can use Equations 1.11 and 1.23 to compute the hit location, \vec{h} and the normal \vec{n} .

1.4.3 Ray-Triangle Intersection

The computation of intersections between a ray and a triangle is probably one of the most frequently used components of most ray tracers. Many ray tracers only support triangles as a geometric primitive in order to make the ray tracer more robust and efficient. The technique presented in this section is a variation of the method outlined in [?], which has the advantage that it does not require any extra storage beyond the actual triangle data.

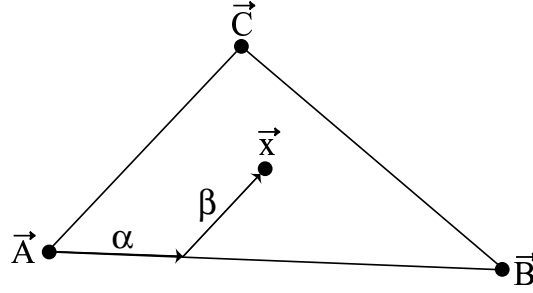


Figure 1.5: A point inside any triangle is uniquely defined by the barycentric coordinates (α, β) .

A triangle is specified by the three vertices \vec{A} , \vec{B} , and \vec{C} . In addition it is common to associate values such as normals and texture coordinates with the vertices — these values are linearly interpolated over the triangle once a hit location is determined. For this purpose we need to introduce the concept of barycentric coordinates. The two barycentric coordinates α and β for a triangle are used to uniquely specify a point inside the triangle. As shown in Figure 1.5 this is done using the two edges of the triangle α specifies the distance from \vec{A} along the \vec{AB} edge and β specifies the distance from \vec{A} along the \vec{AC} edge. In this way any point, \vec{x} , inside the triangle is given as:

$$\vec{x} = \vec{A} + \alpha\vec{AB} + \beta\vec{AC}, \quad (1.30)$$

where $\vec{AB} = \vec{B} - \vec{A}$, $\vec{AC} = \vec{C} - \vec{A}$, and where $\alpha > 0$, $\beta > 0$, and $\alpha + \beta < 1$.

To compute the intersection between a ray and the triangle we substitute the ray equation for \vec{x} and this gives:

$$\begin{aligned} \vec{o} + t\vec{d} &= \vec{A} + \alpha\vec{AB} + \beta\vec{AC} \\ \updownarrow & \end{aligned} \quad (1.31)$$

$$\vec{o} - \vec{A} = -t\vec{d} + \alpha\vec{AB} + \beta\vec{AC} \quad (1.32)$$

This is a linear system, $\mathbf{A}\vec{x} = \vec{b}$, with three equations and three unknowns where

$$\mathbf{A} = [-\vec{d} \quad \vec{AB} \quad \vec{AC}] \quad (1.33)$$

$$\vec{b} = [\vec{o} - \vec{A}]^T \quad (1.34)$$

$$\vec{x} = [t \quad \alpha \quad \beta]^T. \quad (1.35)$$

Here, \mathbf{A} is a 3×3 matrix given by its three column vectors. We can solve this system using Cramer's rule and we get:

$$\det(\mathbf{A}) = -\vec{d} \cdot (\vec{AB} \times \vec{AC}) \quad (1.36)$$

$$t = \frac{\det(\vec{b} \ \vec{AB} \ \vec{AC})}{\det(\mathbf{A})} = \frac{(\vec{o} - \vec{A}) \cdot (\vec{AB} \times \vec{AC})}{\det(\mathbf{A})} \quad (1.37)$$

$$\alpha = \frac{\det(-\vec{d} \ \vec{b} \ \vec{AC})}{\det(\mathbf{A})} = \frac{-\vec{d} \cdot ((\vec{o} - \vec{A}) \times \vec{AC})}{\det(\mathbf{A})} \quad (1.38)$$

$$\beta = \frac{\det(-\vec{d} \ \vec{AB} \ \vec{b})}{\det(\mathbf{A})} = \frac{-\vec{d} \cdot (\vec{AB} \times (\vec{o} - \vec{A}))}{\det(\mathbf{A})} . \quad (1.39)$$

The solution is undefined if $\det(\mathbf{A}) = \vec{d} \cdot (\vec{AB} \times \vec{AC}) = 0$. Since the triangle normal is equal to $\vec{AB} \times \vec{AC}$ it can be seen that this only happens when the ray is parallel to the triangle. Once a solution is found we need to check if the triangle was intersected by the ray. This is the case if:

$$\alpha \geq 0 \text{ and } \beta \geq 0 \text{ and } \alpha + \beta \leq 1 . \quad (1.40)$$

Once a hit with a triangle has been found we can compute the hit location using Equation 1.11 and the triangle normal, \vec{n} , at the hit is given as is given as:

$$\vec{n} = \vec{AB} \times \vec{AC} . \quad (1.41)$$

Note, that this vector has been computed already as part of the evaluation of $\det(\mathbf{A})$.

For triangles it is common to associate values with the vertices. This can either be normals or texture coordinates that are linearly interpolated over the triangle once a hit is found. If three normals \vec{n}_A , \vec{n}_B , and \vec{n}_C are given for the three vertices A , B , and C then the interpolated normal, \vec{n} , is computed as:

$$\vec{n} = (1 - \alpha - \beta)\vec{n}_A + \alpha\vec{n}_B + \beta\vec{n}_C . \quad (1.42)$$

Similarly, if texture coordinates (u_A, v_A) , (u_B, v_B) and (u_C, v_C) are given at the triangle vertices then the interpolated texture coordinates (u, v) are computed as:

$$\begin{aligned} u &= (1 - \alpha - \beta)u_A + \alpha u_B + \beta u_C \\ v &= (1 - \alpha - \beta)v_A + \alpha v_B + \beta v_C . \end{aligned} \quad (1.43)$$

1.4.4 Ray-Box Intersection

This section describes a method for intersecting a ray with an axis-aligned box. The axis aligned box is specified by the location of two opposit corners \vec{a} and \vec{b} of the box. To compute the intersection of the box with a ray we use the

slab method introduced by Kay and Kajiya [?]. Their method first computes an intersection between the ray and the three pairs of planes (x , y , and z):

$$\begin{aligned} t_{x1} &= \frac{\vec{a}_x - \vec{o}_x}{\vec{d}_x} \quad , \quad t_{x2} = \frac{\vec{b}_x - \vec{o}_x}{\vec{d}_x} \\ t_{y1} &= \frac{\vec{a}_y - \vec{o}_y}{\vec{d}_y} \quad , \quad t_{y2} = \frac{\vec{b}_y - \vec{o}_y}{\vec{d}_y} \\ t_{z1} &= \frac{\vec{a}_z - \vec{o}_z}{\vec{d}_z} \quad , \quad t_{z2} = \frac{\vec{b}_z - \vec{o}_z}{\vec{d}_z} \end{aligned} \quad (1.44)$$

The ray only intersects the box if the intervals $[t_{x,min}, t_{x,max}]$, $[t_{y,min}, t_{y,max}]$, and $[t_{z,min}, t_{z,max}]$ overlap. We can compute the extents of the common interval $[t_{min}, t_{max}]$ as follows:

$$t_{min} = \max \{ \min(t_{x1}, t_{x2}) , \min(t_{y1}, t_{y2}) , \min(t_{z1}, t_{z2}) \} \quad (1.45)$$

$$t_{max} = \min \{ \max(t_{x1}, t_{x2}) , \max(t_{y1}, t_{y2}) , \max(t_{z1}, t_{z2}) \} \quad (1.46)$$

If $t_{min} \leq t_{max}$ then the ray intersects the box at $t = t_{min}$ (or at $t = t_{max}$ if the ray originates inside the box). The ray can also be in front of the box if $t_{max} < 0$.

To find the normal, \vec{n} , at the hit location we compute a vector \vec{p} from the center of the box to the hit location. The next step is finding the largest absolute value in the vector \vec{p} and setting the other elements of \vec{p} to 0. The resulting vector is the normal \vec{n} , and it should be normalized.

1.4.5 Triangle-Box Intersection

The ability to decide if a triangle intersects a box is not a fundamental component of the basic ray tracing, but later we will see how this algorithm becomes very important for building acceleration structures in complex scenes (with millions of triangles). For this purpose it is necessary to compute if a triangle intersects, or more precisely, if any part of a triangle is within a given axis-aligned box. In principle this computation is of importance for other objects such as spheres, squares etc. but in practice triangles are the most important primitive in complex scenes, and we therefore restrict our description to the triangle box test.

Given a box described by the minimum corner \vec{a} and the maximum corner \vec{b} (i.e. $\vec{b}_x > \vec{a}_x$, $\vec{b}_y > \vec{a}_y$, and $\vec{b}_z > \vec{a}_z$), the task is to decide if a triangle specified by the vertices \vec{A} , \vec{B} , and \vec{C} is intersecting the box volume (the triangle can also be fully contained within the box). For this computation it is necessary to perform a number of steps that can be used to either decide that the triangle is inside or completely outside the box. The tests progress from simple and quick tests to more complex intersection computations.

Vertex in box test: if any of the triangle vertices are inside the box then the triangle intersects the box:

$$\text{intersection if } \left\{ \begin{array}{l} \vec{a}_x \leq \vec{A}_x \leq \vec{b}_x \text{ and } \vec{a}_y \leq \vec{A}_y \leq \vec{b}_y \text{ and } \vec{a}_z \leq \vec{A}_z \leq \vec{b}_z \\ \text{or} \\ \vec{a}_x \leq \vec{B}_x \leq \vec{b}_x \text{ and } \vec{a}_y \leq \vec{B}_y \leq \vec{b}_y \text{ and } \vec{a}_z \leq \vec{B}_z \leq \vec{b}_z \\ \text{or} \\ \vec{a}_x \leq \vec{C}_x \leq \vec{b}_x \text{ and } \vec{a}_y \leq \vec{C}_y \leq \vec{b}_y \text{ and } \vec{a}_z \leq \vec{C}_z \leq \vec{b}_z \end{array} \right.$$

Triangle outside box test: if all of the triangle vertices are outside one of the planes defining the sides of the box then the triangle can be trivially rejected as outside:

$$\text{no intersection if } \left\{ \begin{array}{l} \vec{A}_x < \vec{a}_x \text{ and } \vec{B}_x < \vec{a}_x \text{ and } \vec{C}_x < \vec{a}_x \\ \text{or} \\ \vec{A}_y < \vec{a}_y \text{ and } \vec{B}_y < \vec{a}_y \text{ and } \vec{C}_y < \vec{a}_y \\ \text{or} \\ \vec{A}_z < \vec{a}_z \text{ and } \vec{B}_z < \vec{a}_z \text{ and } \vec{C}_z < \vec{a}_z \\ \text{or} \\ \vec{A}_x > \vec{b}_x \text{ and } \vec{B}_x > \vec{b}_x \text{ and } \vec{C}_x > \vec{b}_x \\ \text{or} \\ \vec{A}_y > \vec{b}_y \text{ and } \vec{B}_y > \vec{b}_y \text{ and } \vec{C}_y > \vec{b}_y \\ \text{or} \\ \vec{A}_z > \vec{b}_z \text{ and } \vec{B}_z > \vec{b}_z \text{ and } \vec{C}_z > \vec{b}_z \end{array} \right.$$

Triangle edge - box intersection test: if one of the three triangle edges intersects the box then the triangle intersects the box. For this purpose, we define three rays from \vec{A} to \vec{B} , from \vec{A} to \vec{C} , and from \vec{B} to \vec{C} . Each of these rays are tested for intersection with the bounding box using the method described in section 1.4.4.

Box diagonal - triangle intersection test: if one of the four box diagonals intersects the triangle then the triangle intersects the box. As before define four different rays from each of the four corners on one side of the box to the four opposite corners on the other side of the box. Each ray is tested against the triangle using the method described in section 1.4.3.

Done : if none of the last two tests succeeded then the triangle does not intersect the box.

1.5 Advanced Geometry

1.5.1 Transformed Geometry

Normal Transformation

1.5.2 Subdivision Surfaces

Loop subdivision

Catmull-Clark subdivision

1.5.3 Bicubic Patches

1.5.4 NURBS

1.6 Acceleration Structures

The basic ray tracing algorithm requires an intersection computation with all objects in the scene for every ray, and the complexity is $O(N_{rays} \cdot N_{objects})$. In complex scenes with millions of objects (and millions of rays) this operation becomes a major bottleneck and ray tracing is not practical without the use of acceleration structures. The purpose of an acceleration structure is to reduce the number of ray-object intersections in order to speedup ray tracing.

The key idea behind acceleration structures is that the objects in the scene should be encapsulated using a structure that is simple to ray trace. There are several different types of acceleration structures, and they each have different advantages depending on the type of scene that is being ray traced. The distribution of objects in the scene is particularly important for the behaviour of most acceleration structures. The key types of acceleration structures are:

- Bounding volume hierarchies
 - Bounding spheres
 - Bounding boxes
 - Slabs
- Spatial subdivision
 - Grids (uniform, hierarchical, and adaptive)
 - Octrees
 - BSP trees
 - 5D trees

In the following sections the most commonly used types of acceleration structures will be described. Each acceleration structure has two components: how to build it, and how to traverse it efficiently.

1.6.1 Bounding Box Hierarchies

1.6.2 Grids

1.6.3 BSP Trees

The BSP tree is a very practical and frequently used acceleration structure for ray tracing. BSP stands for Binary Space Partitioning, and this is exactly how BSP trees work. First a bounding box is computed for the entire scene. This bounding box is then partitioned into two boxes along a *splitting plane* - for ray tracing this splitting is almost always axis-aligned since it makes the algorithm much simpler, and in the following we will only consider axis-aligned splitting planes.

```
struct BSP_node {
    int axis;
    float plane;
    BSP_node *left, *right;
    bool is_leaf;
    Object *objects;
}
```

Figure 1.6: A BSP tree node can be either a leaf node with a pointer to an object list, or an internal node containing information about the splitting plane and pointers to the left and right nodes.

The nodes of a BSP tree can be either internal nodes that point to other BSP tree nodes or leaf nodes that may contain objects. An example of a BSP node is shown in Figure 1.6 — note that this node is rather large, in practice several of the elements are combined in order to reduce the memory requirements for the BSP tree.

The root node of the BSP tree is the bounding box of all the objects in the tree. This root node can be represented as shown in Figure 1.6. The BSP tree is constructed by recursively splitting the nodes of the BSP tree until a certain stopping criteria is reached. The criteria can be until the number of objects in the node is below a certain threshold, or a maximum tree level has been reached. Each internal node is split using an axis-aligned splitting plane. The location of the splitting plane is often the center of the node although any position within the node can be used. The splitting axis can arbitrary as well although it is common to alternate the choice of axis between levels (e.g. x at the root, y at the next level, then z, x, y, z, ...). The pseudocode for recursively constructing a BSP tree is shown in Figure 1.7.

Tracing a ray through a BSP tree can be done efficiently by recursively picking the left and right nodes of the tree in the order in which they are intersected by the ray. The pseudocode for the traversal algorithm is shown in Figure 1.8. This algorithm keeps track of a valid interval $[t_{min}, t_{max}]$ in which

```

subdivide_node( node, objlist, level )
{
    if ( objlist.size < maxobj || level == maxlevel ) {
        node->objects = objlist;
        node->is_leaf = true
    } else {
        node->is_leaf = false
        select axis and splitting plane position

        objleft = objects on left side of splitting plane
        subdivide_node( node->left, objleft, level+1 )
        objright = objects on right side of splitting plane
        subdivide_node( node->right, objright, level+1 )
    }
}

```

Figure 1.7: The BSP tree is build by recursively splitting a node until it contains sufficiently few objects or until a maximum tree level has been reached.

the ray is inside the current bounding node. This interval can then be used to avoid intersected both child nodes in case the ray only passes through one of the nodes. This is illustrated in Figure 1.9. In practice the BSP tree traversal algorithm is often implemented using an explicit stack on which the far node is pushed in order to avoid the overhead of the recursive function calls.

1.6.4 Other Acceleration Structures

Octrees

Adaptive grids

Slabs

5D trees

1.7 Shading

Shading is the process of computing the radiance leaving a surface in a given the direction (most often in the direction of the incoming ray). The key parameters of most shading calculations are:

- The direction of the incident ray, \vec{d}
- The hit location, \vec{h}
- The normal, \vec{n} , at \vec{h}
- The local material parameters

In this section, we will describe a basic shading model for ray tracing that makes it possible to compute shading from point lights as well as diffuse shading, specular reflections, and specular refractions.

1.7.1 A Basic Shading Model

1.7.2 Direct Illumination

1.7.3 Diffuse Reflection

1.7.4 Specular Reflection

1.7.5 Specular Refraction

1.7.6 Texturing

1.7.7 Bump Mapping

1.7.8 Displacement Mapping

1.7.9 Environment Mapping


```

intersect_bsp()
{
    [t_min,t_max] = intersect bounding box
    intersect_node( root, t_min, t_max )
}

intersect_node( node, t_min, t_max )
{
    if (node->is_leaf) {
        t = intersect node->objects
        if (t < t_max)
            done
    } else {
        if (ray.direction[ node->axis ] > 0)
            near = node->left; far = node->right
        else
            near = node->right; far = node->left;

        t = intersect node->plane
        if ( t > t_max )
            intersect_node( near, t_min, t_max )
        else if ( t < t_min )
            intersect_node( far, t_min, t_max )
        else {
            intersect_node( near, t_min, t )
            intersect_node( far, t, t_max )
        }
    }
}

```

Figure 1.8: A ray is traced through the BSP tree by recursively selecting the right or left node in the order in which they are intersected by the ray. This continues until a leaf node is reached at which point the objects in the node are checked for intersections. The traversal is stopped when an intersection within the current interval is found.

A ray passing through a node can hit both, or just the left and the right node.

Figure 1.9: Ray node intersection modes....

Index

- Acceleration Structures, 13
- Barycentric Coordinates, 9
- Box Intersection, 10
- BSP tree, 14
 - Building, 14
 - Nodes, 14
 - Ray traversal, 14
- Bump mapping, 16
- Camera Setup, 4
- Displacement mapping, 16
- Environment mapping, 16
- Intersection, 5
 - BSP tree, 14
 - Ray-Box, 10
 - Ray-Plane, 6
 - Ray-Sphere, 6
 - Ray-Triangle, 9
 - Triangle-Box, 11
- Normal Transformation, 13
- Plane Intersection, 6
- Ray Tracing, 3
 - Overview, 3
 - Ray Structure, 3
- Shading, 15
- Shading model, 16
- Sphere Intersection, 6
- Texturing, 16
- Transformations, 13
- Transforming Geometry, 13
- Triangle Intersection, 9
- Triangle-Box Intersection, 11
- View Setup, 4