Notes for CS155/Spr07 Computer Graphics

Timothy J. Hickey

November 8, 2007

Contents

1 Introduction **5**
1.1 Ray Tracing 5

2 Points and Rays **7**
2.1 3D coordindates 7
2.1.1 Proof of the Pythagorean Theorem 8
2.1.2 Proof of the Generalized Pythagorean Theorem 8
2.2 Dot products, norms, and vector lengths 9
2.2.1 Geometric Interpretation of the Dot Product 10
2.2.2 Dot Product as Matrix Multiplication 12
2.3 The Cross Product 12
2.3.1 Proof that the cross product $\mathbf{u} \times \mathbf{v}$ is perpendicular to $\mathbf{u}$ and $\mathbf{v}$ 13
2.3.2 The Matrix representation of cross product 13
2.4 Representing Points in Java 13
2.5 Representing rays 14
2.6 Exercises 15

3 Basic Ray Tracing **17**
3.1 Objects, Object Groups and RayHits 17
3.2 Finding the intersection of a ray and a sphere 19
3.2.1 Representing spheres 19
3.2.2 Intersecting a ray with a sphere 20
3.2.3 Corner cases 20
3.2.4 Finding the intersection point 21
3.3 RayTracing a sphere using vector notation 21
3.3.1 Intersecting a ray with a sphere in vector notation 21
3.4 The Sphere3D class 22

4 More Objects **25**
4.1 Adding planes to the ray tracer 25
4.1.1 Constraint for a point to be in a plane 25
4.1.2 Intersecting a ray with a plane in vector notation 25
4.2 Adding cylinders to the ray tracer 25

4.2.1 Intersecting the cylinder with a ray 27
4.2.2 Finding the normal to a cylinder at a point 28

5 Lighting Models 31
5.1 Ambient illumination 31
5.2 Diffuse illumination 31
5.3 Specular illumination 32
5.4 Lights 33

6 Implementing a Basic Ray Tracer 35
6.1 Cameras 35
6.2 The Scene class 36
6.3 The Canvas3D class 36
6.4 The RayTracer3D class 37
6.5 A Simple GUI 38
6.6 Examples 38

7 Expanding the Lighting Model 41
7.1 Color 41

8 Reflection and Refraction 43
8.1 Reflection 43
8.2 Refraction 44
8.3 Changes to RayTracer3D 45

9 Transforms 47
9.1 Fundamental Transforms 47
9.1.1 Translation 47
9.1.2 Scaling 48
9.1.3 Rotation 48
9.1.4 Compound Transformations 49
9.1.5 The general rotation around a vector v 50
9.1.6 Proof of the general rotation formula 51
9.2 Applying a transform to a ray 52
9.3 Intersecting a ray with a transformed object 52
9.4 Exercises 53
9.5 Changes to RayTracer3D 53

10 Textures 55
10.1 Textures for a cube 55
10.2 Texture mapping for the plane 55
10.3 Texture mapping for the cylinder 56
10.4 Texture Coordinates for a sphere 56
10.5 Texture transforms 57

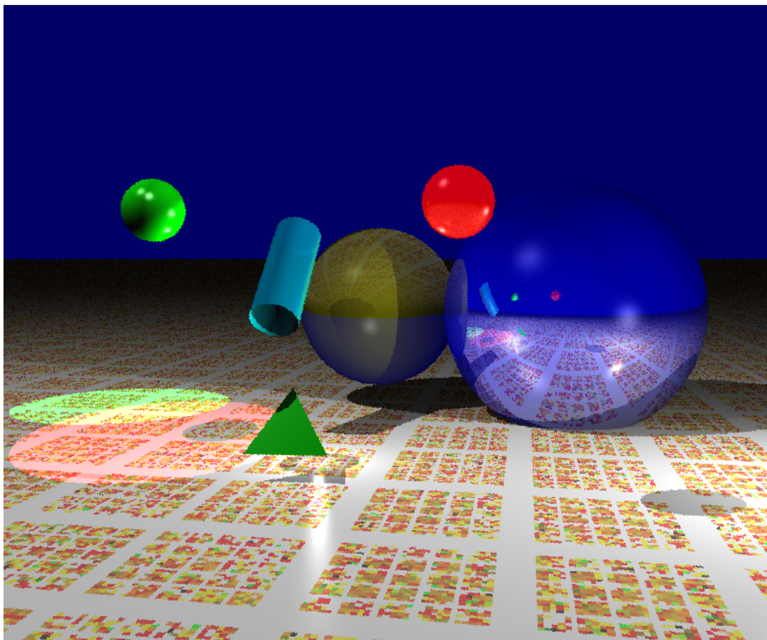
11 Scene Graphs **59**
11.1 JScheme as a Scene Description Language 59
11.2 Sample Scenes 59

A Solutions to Exercises **61**
A.1 Chapter 2 61
A.2 Chapter 9 63

Chapter 1

Introduction

These notes provide an introduction to RayTracing for use by the CS155: Computer Graphics course at Brandeis University, taught during the Fall 2007 semester.



1.1 Ray Tracing

Ray Tracing is a method of generating realistic 3D images from a mathematical description of a scene. The basic idea is that one creates a 3d scene consisting

of a camera, lights, and various 3d objects, such as sphere, planes, cylinders, or polyhedra. Next one generates a 2d image on the screen that represents the view of that 3d scene from the camera by associating to each pixel in the screen window a ray that starts at the camera and moves out in a straight line into the scene. Using simple (and fast) mathematical tests, one determines the first object that the ray intersects and then determines the color of that pixel by using the properties of the lights and materials. One can obtain more realistic images by letting the ray reflect off of the partly mirrored object and refract through the partly transparent objects. The reflected and refracted light must then be combined in a weighted average with the already computed material color.

In these notes we will describe the mathematics and algorithms behind ray tracing and in the process develop a simple ray tracer in Java. The image below is an example of our ray tracer can produce:

We begin the notes with some preliminaries about points and rays in 3d space.

Chapter 2

Points and Rays

2.1 3D coordindates

Any point **a** in 3d space can be uniquely described by giving its 3d coordinates $\mathbf{a} = (a_x, a_y, a_z)$ which describe the distance to a in each of the three directions corresponding to the x, y, and z axes.

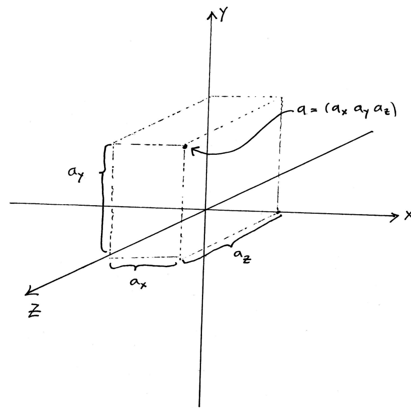


Figure 2.1: 3d coordinates

Each point **a** uniquely defines a directed line segment from the origin $(0, 0, 0)$ to **a**. We call such directed line segments vectors and we will often switch back and forth between these two interpretations of **a**.

For example, the distance d of a point **a** from the origin $(0, 0, 0)$ is also the length of the vector **a** which we denote $|\mathbf{a}|$ and which is computed using a generalized pythagorean theorem:

$$|\mathbf{a}| = \sqrt{a_x^2 + a_y^2 + a_z^2} \tag{2.1}$$

2.1.1 Proof of the Pythagorean Theorem

Lets prove the generalize pythagorean theorem (Eq 2.1) by starting with a simple geometric proof of the 2 dimensional Pythagorean theorm. Remember, this states that if a right triangle has sides of length A, B, and C with c being the hypotenuse, then $A^2 + B^2 = C^2$.

This can be easily demonstrated using the diagram in Figure 2.2.

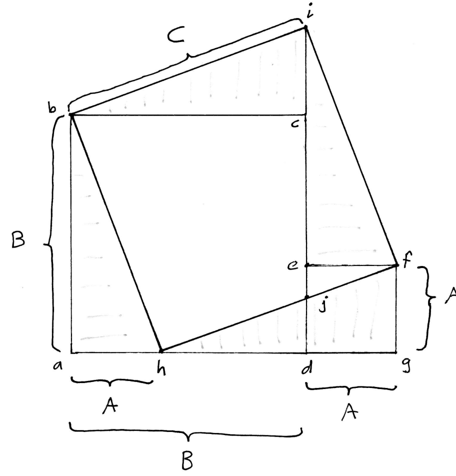


Figure 2.2: Pythagorean Theorem

Observe that A^2+B^2 is the area of the two squares $abcd$ and $defg$. The clever idea here is to cut out two right triangles $\triangle ahb$ and $\triangle gfh$ with sides A, B, C from these two squares and to move them to $\triangle efi$ and $\triangle ib$ respectively. Moving these two triangles changes the shape from the union of two squares ($abcd$ and $defg$) into the single quadrilateral $bhfi$ whose sides are all of length C. If we can prove that this quadilateral is a square, then its area will be C^2 and we will have shown that $A^2+B^2 = C^2$ for a right triangle of sides A, B and hypotenuse C. So we need to prove that each of the four angles of the quadrilateral are right angles, but this follows easily from the fact that the sum of the angles of a triangle is 180 degrees and hence the sum of the two non-right angles in 90 degrees. For example,

$$\angle fib = \angle fic + \angle cib = \angle ibc + \angle cib = 90^\circ$$

and we leave the other three angles as geometry exercises. Q.E.D.

2.1.2 Proof of the Generalized Pythagorean Theorem

We can prove the 3 dimensional pythagorean theorem by applying the 2d version twice. Indeed, consider the figure below. We let $\mathbf{p} = (p_x, p_y, p_z)$ be an arbitrary

3d point and let $\mathbf{q} = (p_x, 0, p_z)$ be its projection into the xz plane. Observe that the line segment from the origin to $\mathbf{q}$ is the hypotenuse of a right triangle in the xz-plane whose sides have lengths p_x and p_z. Thus we can apply the 2d pythagorean theorem to determine the length h_1 of the vector $\mathbf{q}$:

$$h_1 = p_x^2 + p_z^2$$

Now consider the three points p, q, and the origin o. These three points uniquely determine a plane and its clear that then angle $\angle oqp$ is a right angle (since the y-axis is perpendicular to the xz plane). Thus, we can again apply the 2d pythagorean theorem and we obtain the length h_2 of the vector $\mathbf{p}$

$$h_2 = h_1^2 + p_y^2 = p_x^2 + p_z^2 + p_y^2 = p_x^2 + p_y^2 + p_z^2$$

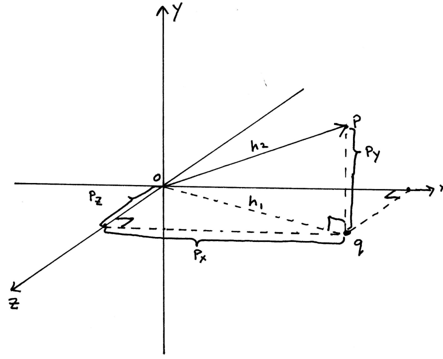


Figure 2.3: Length of a vector in 3d

A similar proof would work in dimensions 4 and more by inductively projecting down 1 dimension, applying the $n-1$ dimensional pythagorean theorem and then completing the induction step using the 2d pythagorean theorem in the 2d plane consisting of the origin, the point, and its projection into an $n-1$ dimensional subspace.

2.2 Dot products, norms, and vector lengths

Given two vectors $\mathbf{p} = (p_x, p_y, p_z)$ and $\mathbf{q} = (q_x, q_y, q_z)$. Their dot-product is denoted $\mathbf{p} \cdot \mathbf{q}$ and is defined by

$$\mathbf{p} \cdot \mathbf{q} = (p_x * q_x + p_y * q_y + p_z * q_z)$$

The norm of a vector is defined to be its dot product with itself:

$$\|\mathbf{p}\| = \mathbf{p} \cdot \mathbf{p} = (p_x^2 + p_y^2 + p_z^2)$$

As we have seen above, the norm has the property that its square root is the length of $\mathbf{p}$ and we often use single bars to denote the length of the vector:

$$|\mathbf{p}| = \sqrt{\|\mathbf{p}\|}$$

A vector whose length is 1 is called "normalized" Any vector $\mathbf{v}$ can be normalized by dividing by its length:

$$\frac{\mathbf{v}}{|\mathbf{v}|}$$

2.2.1 Geometric Interpretation of the Dot Product

In this section, we show that if $\mathbf{u}$ and $\mathbf{v}$ are two vectors, then

$$\mathbf{u} \cdot \mathbf{v} = \cos(\theta)|\mathbf{u}||\mathbf{v}| \tag{2.2}$$

where θ is the angle between $\mathbf{u}$ and $\mathbf{v}$.

We can get a geometric interpretation of this property by considering the figure below. Let u and v be two vectors and assume that u is normalized (i.e. $|\mathbf{u}| = 1$). Then as the figure shows the projection of v onto u is a vector of length $\cos(\theta)|\mathbf{v}|$.

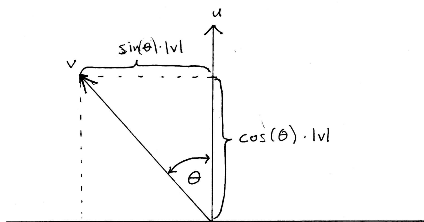


Figure 2.4: $|v|\ \cos(\theta)$ is the projection of $\mathbf{v}$ on $\mathbf{u}$

To prove the property of the dot product in Eqn 2.2, we need to first recall and prove the Law of Cosines which is a generalization of the pythagorean theorem for general triangles.

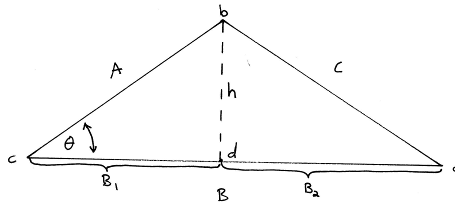


Figure 2.5: Law of Cosines: $C^2 = A^2 + B^2 - 2AB\cos(\theta)$

Let $\triangle abc$ be a general triangle as in Figure 2.5 and let θ be the angle $\angle abc$. Let A, B, C be the the lengths of the edges opposite the sides a, b, c respectively, as shown in the following figure:

The Law of Cosines states that the following relationship holds for any such triangle:

$$C^2 = A^2 + B^2 - 2AB\cos(\theta)$$

To prove this relation, we decompose the triangle $\triangle abc$ into two right triangles $\triangle adb$ and $\triangle cdb$ and let h be the length of the common side bd and B_1 and B_2 the lengths cd and da respectively. Then, by the definition of the cosine we know that

$$B_1 = A * \cos(\theta)$$

and from the Pythagorean Theorem we know that

$$\begin{aligned} A^2 &= B_1^2 + h^2 \\ C^2 &= B_2^2 + h^2 \end{aligned}$$

and since $B = B_1 + B_2$ we see that

$$\begin{aligned} B^2 &= (B_1 + B_2)^2 \\ &= B_1^2 + 2B_1B_2 + B_2^2 \end{aligned}$$

So

$$\begin{aligned} C^2 - (A^2 + B^2) &= B_2^2 + h^2 - (B_1^2 + h^2 + B_1^2 + 2B_1B_2 + B_2^2) \\ &= B_2^2 + h^2 - B_1^2 - h^2 - B_1^2 - 2B_1B_2 - B_2^2) \\ &= -2B_1^2 - 2B_1B_2) \\ &= -2B_1(B_1 + B_2) \\ &= -2B_1B \\ &= -2(A\cos(\theta))B \\ &= -2AB\cos(\theta) \end{aligned}$$

and so

$$C^2 = A^2 + B^2 - 2AB\cos(\theta)$$

To apply the law of cosines to the dot product, observe that if $\mathbf{u}$ and $\mathbf{v}$ are two vectors then the three points $\mathbf{u}$, $\mathbf{v}$, and the origin $\mathbf{o}$ all lie in a plane and form a triangle in that plane. We can let A be the length of $\mathbf{u}$ and B be the length of $\mathbf{v}$ and let C be the length of the third side $\mathbf{w} = \mathbf{v} - \mathbf{u}$ and let θ be the angle between $\mathbf{u}$ and $\mathbf{v}$.

By the Law of Cosines we know that:

$$C^2 = A^2 + B^2 - 2AB\cos(\theta)$$

and since $A^2 = \mathbf{u}\cdot\mathbf{u}$, $B^2 = \mathbf{v}\cdot\mathbf{v}$, $C^2 = \mathbf{w}\cdot\mathbf{w}$ we have

$$\mathbf{w}\cdot\mathbf{w} = \mathbf{u}\cdot\mathbf{u} + \mathbf{v}\cdot\mathbf{v} - 2AB\cos(\theta)$$

Substituting in $\mathbf{w} = \mathbf{v} - \mathbf{u}$ and using the distributive property of the dot product we find that

$$\begin{aligned} \mathbf{u}\cdot\mathbf{u} + \mathbf{v}\cdot\mathbf{v} - 2AB\cos(\theta) &= \mathbf{w}\cdot\mathbf{w} \\ &= (\mathbf{v}-\mathbf{u})\cdot(\mathbf{v}-\mathbf{u}) \\ &= \mathbf{v}\cdot\mathbf{v} - \mathbf{u}\cdot\mathbf{v} - \mathbf{v}\cdot\mathbf{u} + \mathbf{u}\cdot\mathbf{u} \end{aligned}$$

and so

$$-2AB\cos(\theta) = -2\mathbf{u}\cdot\mathbf{v}$$

from which the dot product property follows immediately from $A = |\mathbf{u}|$ and $B = |\mathbf{v}|$:

$$\mathbf{u}\cdot\mathbf{v} = |\mathbf{u}|\ |\mathbf{v}|\ \cos(\theta)$$

2.2.2 Dot Product as Matrix Multiplication

If we think of $\mathbf{p}$ and $\mathbf{q}$ as being column matrices

$$p = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \quad q = \begin{pmatrix} q_x \\ q_y \\ q_z \end{pmatrix}$$

then we can also write the dot product as a matrix multiplication of a (1x3) and a (3x1) matrix to get a 1x1 matrix:

$$p\cdot q = p^T * q = (p_x p_y p_z) * \begin{pmatrix} q_x \\ q_y \\ q_z \end{pmatrix} = p_x q_x + p_y q_y + p_z q_z$$

and where M^T denotes the transpose of the matrix M, that is the matrix whose rows and columns are reversed:

$$M^T_{i,j} = M_{j,i}$$

2.3 The Cross Product

The cross product is a very useful operation which allows one to take any two vectors $\mathbf{u}$ and $\mathbf{v}$ and to generate a new vector $\mathbf{u}\times\mathbf{v}$ called the cross product of $\mathbf{u}$ and $\mathbf{v}$ which has the property that it is perpendicular to both u and v. Equivalently, if u and v are not multiples of each other, then they define a 2d plane in 3d space, and w will be perpendicular to that plane.

The cross product is relatively easy to compute using the following formula:

$$\mathbf{u}\times\mathbf{v} = (u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_z v_x)$$

and it satisfies the following properties:

$$(\mathbf{u}\times\mathbf{v})\cdot u = 0 \ \ (\mathbf{u}\times\mathbf{v})\cdot v = 0$$

2.3.1 Proof that the cross product $\mathbf{u} \times \mathbf{v}$ is perpendicular to $\mathbf{u}$ and $\mathbf{v}$

To prove this we need some linear algebra. Lets assume that $\mathbf{u}$ and $\mathbf{v}$ do not lie on the same line in 3d space, then they must be linearly independent and hence they uniquely define a 2d plane P. Recall that if $\mathbf{w}$ is any vector, then $\mathbf{w}$ lies in the plane defined by $\mathbf{u}$ and $\mathbf{v}$ if and only if the determinant with rows $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ is zero, that is

$$\begin{vmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{vmatrix} = 0$$

We can compute the determinant by expanding the minors along the bottom row to get

$$\begin{aligned} \begin{vmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{vmatrix} &= w_x(u_y v_z - u_z v_y) - w_y(u_x v_z - u_z v_x) + w_z(u_x v_y - u_y v_x) \\ &= (w_x, w_y, w_z) \cdot (u_y v_z - u_z v_y, -(u_x v_z - u_z v_x), u_x v_y - u_y v_x) \\ &= \mathbf{w} \cdot (\mathbf{u} \times \mathbf{v}) \end{aligned}$$

Thus, we see that $\mathbf{w} \cdot (\mathbf{u} \times \mathbf{v})$ is zero if and only if w is in the same plane P as u and v, and so we conclude that

$$\mathbf{u} \cdot (\mathbf{u} \times \mathbf{v}) = 0 \quad \mathbf{v} \cdot (\mathbf{u} \times \mathbf{v}) = 0$$

2.3.2 The Matrix representation of cross product

The cross product $\mathbf{u} \times \mathbf{v}$ can also be represented using matrix multiplication. Indeed, let $u^\flat$ denote the following matrix constructed from a vector u:

$$u^\flat = \begin{pmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 0 \end{pmatrix}$$

and observe that $(\mathbf{u} \times \mathbf{v}) = \mathbf{u}^\flat * \mathbf{v}$

2.4 Representing Points in Java

In this section we provide Java classes that represent points and rays in 3D space. The Point3D class represents 3d points and vectors while Ray3D represents rays (as pairs of Point3D objects). The code is in Figures 2.6 and 2.7 and it is a fairly straightforward encoding of the operations we have introduced in the previous chapters (adding and scaling points, dot product, cross product, length, norm).

We represent the coordinates of a Point3D object by three public fields x, y, z of doubles. There is also an additional field w which we will need later when we introduce transforms. Observer for now that it is not used by any of the methods

```
package cs155.jray;
/**   A Point3D is a triple of doubles that can represent a point or a vector.
**/
public class Point3D
{   public double x=0.0, y=0.0, z=0.0, w=1.0;

    public Point3D(double x, double y, double z) {
        this.x = x;  this.y = y;  this.z = z;}
    public Point3D(Point3D m){ this(m.x,m.y,m.z); }
    public String toString() { return "point3d("+x+","+y+","+z+")"; }
    public Point3D subtract(Point3D q) { return new Point3D(x-q.x, y-q.y,z-q.z);  }
    public Point3D add(Point3D q) {
        return new Point3D(x+q.x, y+q.y,z+q.z);   }
    public Point3D scale(double a) {return new Point3D(x*a, y*a, z*a);   }
    public Point3D translate(double x_dist, double y_dist, double z_dist) {
        return new Point3D (x + x_dist, y + y_dist, z + z_dist); }
    public double norm(Point3D q) { return q.dot(q);     }
    public double length(Point3D q) { return Math.sqrt(q.dot(q));    }
    public double dot(Point3D q){ return  x*q.x+y*q.y+z*q.z; }
    public Point3D cross(Point3D q){
        return new Point3D(y*q.z - q.y*z, q.x*z - x*q.z, x*q.y - q.x*y); }
    public Point3D normalize() { return this.scale(1/this.length()); }
    public double length() { return Math.sqrt((x*x) + (y*y) + (z*z)); }
}
```

Figure 2.6: Point3D.java

in this class. We will use the w parameter for now to indicate whether a Point3D object represents a point in space (w=1) or a direction (w=0). This is just a convention for now, but will play a critical role when we add transformations to the ray tracer.

2.5 Representing rays

A ray is represented by an origin vector $\mathbf{p} = (p_x, p_y, p_z)$ and a direction vector $\mathbf{d} = (d_x, d_y, d_z)$. The parametric form of a ray is shown below.

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

It is a function of a single variable t and can be thought of as returning the position $\mathbf{r}(t)$ of a point that starts at the origin $\mathbf{p}$ and moves in direction $\mathbf{d}$ at a constant rate of $|d|$ units per time unit. We will usually assume that the direction d of a ray has length 1 so we don't have to normalize it in the formula for $\mathbf{r}(t)$ as above.

```
package cs155.jray;
/**   A Ray3D consists of a point and a (normalized) direction. **/
public class Ray3D
 {  public Point3D p,d;
  /**
    This represents a 3D ray with a specified origin point p and direction d.
    The direction of a ray is a normalized vector.
  **/
  public Ray3D(Point3D p, Point3D d) {
    this.p = p;  this.d = d.normalize(); this.d.w=0;  }

  /**   This returns the point along the ray t units from its origin p   **/
  public Point3D atTime(double t){
    return new Point3D((p.x+t*d.x), p.y + t*d.y, p.z+t*d.z);   }
}
```

Figure 2.7: Ray3D.java

Equivalently, we can compute the component functions for **r** (assuming **d** has length 1).

$$\mathbf{r}(t) = (r_x(t), r_y(t), r_z(t))$$

$$\begin{aligned} r_x(t) &= p_x + t * d_x \\ r_y(t) &= p_y + t * d_y \\ r_z(t) &= p_z + t * d_z \end{aligned}$$

A ray is represented in our Ray3D class by two points p and d, where p is the origin of the ray and d is the direction of the ray. This is a very simple class which allows one to construct a ray from a point and a direction.

When tracing rays, we think of light particles moving out along the ray until they hit an object (which is of course the reverse of what actually happens!). The method `r.atTime(t)` returns the point on the ray whose distance from the origin is given by `t`. If we think of the light moving along the ray at a speed of 1 unit per second, then `r.atTime(t)` returns the location after `t` seconds.

2.6 Exercises

Consider the following three points

$$\begin{aligned} a &= (1, 2, 2) \\ b &= (7, -2, 4) \\ c &= (0, 4, -2) \\ d &= (1, 1, 1) \end{aligned}$$

Let u be the vector from b to a, and let v be the vector from b to c. Let P be the plane that passes through d and is normal to a.

1. How long is the vector u?
2. Is the angle between u and v less than 90 degrees, equal to 90 degrees, or greater than 90 degrees? How do you know?
3. Calculate the normalization of the vector a, i.e. the unit vector pointing in the same directions as a.
4. How far away is the point b from the plane P?
5. Find the point p which is the projection of b onto the plane P.
6. Use the cross product to find a vector w which is perpendicular to u and v.
7. Find the point e that you get by rotating the point c around the x axis by 90 degrees.

Chapter 3

Basic Ray Tracing

In this chapter we develop the fundamental ray tracing algorithms and provide a Java implementation for the corresponding objects.

3.1 Objects, Object Groups and RayHits

We represent an object by an abstract class which a method for intersecting that object with a ray. The class also provides an inner and outer material for the object (so its insides and outsides can be rendered with different colors, textures, and other material properties like shininess, reflectivity, refractivity).

For the moment we will provide a very simple representation for Material which specifies the color of the material and provides a "hardness" value that represents the shininess of the material and which we will discuss in more length in a later chapter when we cover the lighting model.

Objects also can be intersected with a ray and the result is a RayHit object. The code for Object3D is in Figure 3.1. We will show how to extend this class

```
package cs155.jray;
public abstract class Object3D {
    public static final double epsilon = 0.00001;
    public Material insideMat = Material.defaultMat,
                    outsideMat= Material.defaultMat;

    /**  rayIntersect(r) returns the intersection
         of the object with a ray as a RayHit object,
    **/
    public abstract RayHit rayIntersect (Ray3D ray);
}
```

Figure 3.1: Object3D.java

```
package cs155.jray;
public class Material {
    public static Material defaultMat = new Material();
    double hardness = 70d;
    Color3D color = Color3D.WHITE;
}
```

Figure 3.2: Material.java

```
package cs155.jray;
/** Record properties of intersection of a ray and an object **/
public class RayHit {
    public static RayHit NO_HIT = new RayHit(Double.POSITIVE_INFINITY,null, null);

    public double distance; // distance along ray to the first intersection
    public Point3D normal; // normal of the object at the intersection point
    public Object3D obj; // innermost primitive object that this ray hits...

    public RayHit(double distance,Point3D normal, Object3D obj) {
this.distance=d; this.normal=n;this.obj = obj);    }
    public String toString() {
return "rayhit("+distance+","+normal+","+obj+")";    }
}
```

Figure 3.3: RayHit.java

for Spheres, Cylinders, and Planes as we develop the ray intersection algorithms for these objects. The Object3D class contains a parameter `epsilon` which is used to deal with roundoff error as we will see later.

The RayHit object (in Figure 3.3 stores several bits of information about the intersection:

- the distance along the ray at which the intersection takes place (and `Double.POSITIVE_INFINITY` if there is no intersection of the ray and the object.
- the normal vector at the intersection point
- the object that the ray intersects (which is useful when you intersect a ray with a group of objects).

We also define a class `Group3D` (in Figure 3.4) which represents a group of `Object3D` objects and is itself an extension of the `Object3D` class. This class implements the `rayIntersect(r)` method by looping through all objects in the group, finding the intersection points, and keeping the closest one. The class also provide helper methods that allow one to add objects to the group

```
package cs155.jray;
import java.util.ArrayList;
/** This class represents a group of Objects. **/
public class Group3D extends Object3D {
    private ArrayList<Object3D> objs = new ArrayList<Object3D>();
    public Group3D() {; }
    public Group3D(Object3D[] objs) {this.add(objs); }
    public void clear() {objs = new ArrayList<Object3D>(); }
    public void add(Object3D obj) {objs.add(obj); }
    public void add(Object3D[] obj) {
for(int i=0; i<obj.length; i++) objs.add(obj[i]);  }
    public RayHit rayIntersect (Ray3D ray){
        RayHit closestHit = RayHit.NO_HIT;
        for(int i=0; i<objs.size(); i++) {
    Object3D obj = objs.get(i);
    RayHit hit = obj.rayIntersect(ray);
    if (hit.distance < closestHit.distance) {
closestHit = hit; }}
return closestHit;    }
}
```

Figure 3.4: Group3D.java

and to clear the group. We currently implement a Group3D object using a java.util.ArrayList of Object3D elements.

3.2 Finding the intersection of a ray and a sphere

In this section, we show how to compute the interesection of a ray and a sphere using the 3d coordinates directly. In the next section, we provide a simpler approach using vector algebra, and this will be the approach we use with all later objects.

3.2.1 Representing spheres

A sphere consists of all 3D points which are a constant distance r from the center of the sphere $\mathbf{c} = (c_x, c_y, c_z)$. The equation that determines whether a point (x, y, z) is on the sphere of radius r and center $\mathbf{c}$ is

$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2$$

This is an implicit representation of the sphere and can be thought of as a test of whether a point is on the sphere.

3.2.2 Intersecting a ray with a sphere

To determine the intersection of a ray $r_{\mathbf{p},\mathbf{d}}$ with a sphere $S_{\mathbf{c},r}$.

$$\begin{aligned} r^2 &= (r_x(t) - c_x)^2 + (r_y(t) - c_y)^2 + (r_z(t) - c_z)^2 \\ &= (d_x t + p_x - c_x)^2 + (d_y t + p_y - c_y)^2 + (d_z t + p_z - c_z)^2 \\ &= d_x^2 t^2 + 2d_x(p_x - c_x) + (p_x - c_x)^2 \\ &\quad + d_y^2 t^2 + 2d_y(p_y - c_y) + (p_y - c_y)^2 \\ &\quad + d_z^2 t^2 + 2d_z(p_z - c_z) + (p_z - c_z)^2 \end{aligned}$$

So, grouping the terms by powers of t we find that $\mathbf{r}(t)$ is on the sphere if and only if

$$At^2 + Bt + c = 0$$

where

$$\begin{aligned} A &= d_x^2 + d_y^2 + d_z^2 \\ B &= 2(d_x(p_x - c_x) + d_y(p_y - c_y) + d_z(p_z - c_z)) \\ C &= (p_x - c_x)^2 + (p_y - c_y)^2 + (p_z - c_z)^2 - r^2 = 0 \end{aligned}$$

and we can solve this using the quadratic equation. We do this by first evaluating the discriminant $D = B^2 - 4AC$.

If $D < 0$ then there are no solutions, so the ray does not intersect the Sphere.

If $D = 0$, then there is one solution $T = -B/(2A)$ and if this is positive, then the ray intersects the sphere with a glancing blow at exactly one point and does not pierce the interior of the sphere

If $D > 0$ then there are two solutions $t_1 < t_2$

$$\begin{aligned} t_1 &= (-B - sqrt(D))/(2A) \\ t_2 &= (-B + sqrt(D))/(2A) \end{aligned}$$

If both are negative, then the ray does not intersect the sphere. If one is negative and one is positive, the ray is inside the Sphere and it intersect the sphere at time t_2. If both are positive, then it is outside the sphere and intersects it at time t_1.

3.2.3 Corner cases

We've left a few tricky cases out of the analysis above. For example, suppose the discriminant D is positive, so there are two solutions. What does it mean if one of them is zero? Also, what does it mean of the two intersection points t_1 and t_2 are very close (say less than 0.000000001 units apart?) We will need to deal with these "corner cases" in any practical ray tracer we write.

3.2.4 Finding the intersection point

Once we have found the "time" t_0 of the earliest intersection of the ray with the sphere, we can find the intersection point by evaluating $\mathbf{r}$ at that time.

$$\begin{aligned} \mathbf{r}(t_0) &= (r_x(t_0), r_y(t_0), r_z(t_0)) \\ &= (p_x + t_0 * d_x, p_y + t_0 * d_y, p_z + t_0 * d_z) \end{aligned}$$

3.3 RayTracing a sphere using vector notation

Now we reproduce the same derivations as in the previous section, but using vector algebra instead. You will see that the calculations are considerably simpler.

A sphere consists of all 3D points which are a constant distance r from the center of the sphere $\mathbf{c} = (c_x, c_y, c_z)$. The equation that determines whether a point $\mathbf{v}$ is on the sphere of radius r and center $\mathbf{c}$ is

$$\|\mathbf{v} - \mathbf{c}\| = r^2$$

This is an implicit representation of the sphere and can be thought of as a test of whether a point is on the sphere.

3.3.1 Intersecting a ray with a sphere in vector notation

To determine the intersection of a ray $r_{\mathbf{p},\mathbf{d}}$ with a sphere $S_{\mathbf{c},r}$. We look for points $\mathbf{v}$ on the Sphere which are also on the ray, and hence have the form: $\mathbf{p} + t\mathbf{d}$

$$\begin{aligned} r^2 &= \|\mathbf{v} - \mathbf{c}\| \\ &= (\mathbf{v} - \mathbf{c}) \cdot (\mathbf{v} - \mathbf{c}) \\ &= (t\mathbf{d} + \mathbf{p} - \mathbf{c}) \cdot (t\mathbf{d} + \mathbf{p} - \mathbf{c}) \\ &= \|\mathbf{d}\|t^2 + 2\mathbf{d} \cdot (\mathbf{p} - \mathbf{c})t + \|\mathbf{p} - \mathbf{c}\| \end{aligned}$$

So, grouping the terms by powers of t we find that $\mathbf{r}(t)$ is on the sphere if and only if

$$At^2 + Bt + C = 0$$

where

$$\begin{aligned} A &= \|\mathbf{d}\| \\ B &= 2\mathbf{d} \cdot (\mathbf{p} - \mathbf{c}) \\ C &= \|\mathbf{p} - \mathbf{c}\| \end{aligned}$$

and we can solve this using the quadratic equation as before:

We do this by first evaluating the discriminant $D = B^2 - 4AC$.

If $D < 0$ then there are no solutions, so the ray does not intersect the Sphere.

If $D = 0$, then there is one solution $T = -B/(2A)$ and if this is positive, then the ray intersects the sphere with a glancing blow at exactly one point and does not pierce the interior of the sphere

If $D > 0$ then there are two solutions $t_1 < t_2$

$$\begin{aligned} t_1 &= (-B - sqrt(D))/(2A) \\ t_2 &= (-B + sqrt(D))/(2A) \end{aligned}$$

If both are negative, then the ray does not intersect the sphere. If one is negative and one is positive, the ray is inside the Sphere and it intersect the sphere at time t_2. If both are positive, then it is outside the sphere and intersects it at time t_1.

3.4 The Sphere3D class

We can encapsulate the algorithms of the previous section into the Sphere3D class, which represents a sphere and provides methods for intersecting a ray with a sphere (and in the process finding the distance from the origin of the ray to the sphere and the normal to the sphere at the intersection point). The code is in Figure **??**.

```
package cs155.jray;
import java.awt.Color;
/** class represents a 3D sphere **/
public class Sphere3D extends Object3D
{
    public Point3D center;
    public double radius=0.0;

    public Sphere3D (Point3D center, double radius, Material m) {
      super();  this.center = center;  this.radius = radius;
      this.insideMat = this.outsideMat = m;     }

    public RayHit rayIntersect(Ray3D r) {
      Point3D P = r.p, D=r.d;
      Point3D PC = P.subtract(center);
      double t0=-1,  A= D.dot(D),  B= 2*D.dot(PC),
             C= PC.dot(PC) - radius*radius;
      double Discr = (B*B-4*A*C);
      boolean has_solution = (Discr>=0);
      if (!(has_solution)) return RayHit.NO_HIT;

      double SqrtDisc = Math.sqrt(Discr);
      double t1=(-B-SqrtDisc)/(2 *A), t2=(-B+SqrtDisc)/(2*A);
      // note t1<=t2 always
      if (t1>= epsilon) t0=t1;  // ray hits the outside of the sphere
      else if (t2 >= epsilon) t0=t2; // ray starts inside the sphere
      else return RayHit.NO_HIT; // sphere is behind the ray!

      Point3D p0 = r.atTime(t0);
      Point3D n0 = p0.subtract(center).normalize();
      return new RayHit(t0,n0,this);
     }
}
```

Figure 3.5: Sphere3D.java

Chapter 4

More Objects

4.1 Adding planes to the ray tracer

4.1.1 Constraint for a point to be in a plane

A plane P can be specified by a point $\mathbf{q}$ and a normal $\mathbf{m}$ to that plane. A point $\mathbf{v}$ is in the plane if the vector $\mathbf{v} - \mathbf{q}$ is perpendicular to $\mathbf{m}$. That is if

$$(\mathbf{v} - \mathbf{q}) \cdot \mathbf{m} = 0$$

4.1.2 Intersecting a ray with a plane in vector notation

' To determine the intersection of a ray $r_{\mathbf{p},\mathbf{d}}$ with a plane passing through $\mathbf{q}$ and with normal $\mathbf{m}$, we look for points $\mathbf{v}$ on the plane which are also on the ray, and hence have the form: $\mathbf{p} + t\mathbf{d}$

$$\begin{aligned} 0 &= (\mathbf{v} - \mathbf{q}) \cdot \mathbf{m} \\ &= (t\mathbf{d} + \mathbf{p} - \mathbf{q}) \cdot \mathbf{m} \\ &= t\mathbf{d} \cdot \mathbf{m} \; + \; (\mathbf{p} - \mathbf{q}) \cdot \mathbf{m} \end{aligned}$$

So, solving for t we get

$$t = \frac{(\mathbf{q} - \mathbf{p}) \cdot m}{\mathbf{d} \cdot \mathbf{m}}$$

provided $\mathbf{d} \cdot \mathbf{m} \neq 0$. If $t > 0$, then the ray interesects the plane.

4.2 Adding cylinders to the ray tracer

In this section, we specify a cylinder by two vectors and two positive numbers.

- a point $\mathbf{q}$ at the center of the base of the cylinder

```
package cs155.jray;
/** class represents a 3D plane  */
public class Plane3D extends Object3D
{
    public Point3D center,normal;

    public Plane3D (Point3D center, Point3D normal, Material m) {
        super();
this.center=center;
        this.normal=normal.normalize();
        this.insideMat = this.outsideMat = m; }

    private Point3D project(Point3D q) {
    return q.subtract(normal.scale(normal.dot(q)));   }

    public RayHit rayIntersect(Ray3D r) {
      Point3D P = r.p, D=r.d;
      Point3D PC = P.subtract(center);
      double  dn = normal.dot(D),
             cpn = normal.dot(PC),
               t;
      if (dn==0.0) return RayHit.NO_HIT;
      double t0 = -cpn/dn;
      if (t0 < epsilon)
  return RayHit.NO_HIT;
      else {
          Point3D q = r.atTime(t0);
          Point3D q0 = project(q);
  return new RayHit(t0,normal,this);
      }}
}
```

Figure 4.1: Plane3D.java

- a vector **n** that defines the central axis of the cylinder
- the radius r of the cylinder, and
- the height h of the cylinder

We first find a formula for testing whether a point v is on the cylinder. The idea is to project the point v onto a the plane P through q with normal m. This gives a point v_2 on the plane P. If the distance between v_2 and q is r then the point is on the infinite tube containing the cylinder in question:

$$\begin{aligned}
\mathbf{v}_1 &= \mathbf{v} - \mathbf{q} \\
\mathbf{v}_2 &= \mathbf{v}_1 - (\mathbf{v}_1 \cdot \mathbf{m})\mathbf{m} \\
\|\mathbf{v}_2\| &= r^2
\end{aligned}$$

To see if it is on the cylinder we must do two more tests using $\mathbf{v}_1$:

$$\begin{aligned}
\mathbf{v}_1 \cdot \mathbf{m} &\geq 0 \\
\mathbf{v}_1 \cdot \mathbf{m} &\leq h
\end{aligned}$$

4.2.1 Intersecting the cylinder with a ray

We now plug in $\mathbf{v} = t\mathbf{d} + \mathbf{p}$ into the cylinder test above and solve for t. This will give a quadratic equation that we can solve for two roots t_1 and t_2. Then we find the smallest of these two roots whose corresponding point v also satisfies the two dot product constraints.

$$\begin{aligned}
\mathbf{v} &= t\mathbf{d} + \mathbf{p} \\
\mathbf{w} &= \mathbf{p} - \mathbf{q} \\
\mathbf{v}_1 &= t\mathbf{d} + \mathbf{w} \\
\mathbf{v}_1 \cdot \mathbf{m} &= t(\mathbf{d} \cdot \mathbf{m}) + \mathbf{w} \cdot \mathbf{m} \\
\mathbf{v}_2 &= \mathbf{v}_1 - (\mathbf{v}_1 \cdot \mathbf{m})\mathbf{m} \\
&= t\mathbf{d} + \mathbf{w} - (t\mathbf{d} \cdot \mathbf{m} + \mathbf{w} \cdot \mathbf{m})\mathbf{m} \\
&= t(\mathbf{d} - (\mathbf{d} \cdot \mathbf{m})\mathbf{m}) + \mathbf{w} - (\mathbf{w} \cdot \mathbf{m})\mathbf{m} \\
&= t\alpha + \beta
\end{aligned}$$

where

$$\begin{aligned}
\alpha &= \mathbf{d} - (\mathbf{d} \cdot \mathbf{m})\mathbf{m} \\
\beta &= \mathbf{w} - (\mathbf{w} \cdot \mathbf{m})\mathbf{m}
\end{aligned}$$

so the point $\mathbf{v}$ is on the cylinder if and only if

$$\begin{aligned}
r^2 &= \|\mathbf{v}_2\| \\
&= t^2\|\alpha\| + t2\alpha \cdot \beta + \|\beta\|
\end{aligned}$$

which is a quadratic in the variable t

$$A^2 + Bt + C = 0$$

This quadratic has a solution iff the discriminant $D = B^2 - 4AC$ is non-negative where

$$\begin{aligned} A &= \|\alpha\| \\ B &= 2\alpha \cdot \beta \\ C &= \|\beta\| - r^2 \end{aligned}$$

The quadratic formula gives two solutions t_1 and t_2 which corresponds to two points $\mathbf{u}_1 = t_1\mathbf{d} + \mathbf{p}$ and $\mathbf{u}_2 = t_2\mathbf{d} + \mathbf{p}$. We look for the smallest t_i which is positive and for which $(\mathbf{u}_i - \mathbf{q}) \cdot \mathbf{m} \geq 0$ and $(\mathbf{u}_i - \mathbf{q}) \cdot \mathbf{m} \leq h$. If there are no such t_i then the ray does not intersect the cylinder; otherwise the selected u_i is the intersection point.

4.2.2 Finding the normal to a cylinder at a point

To find the normal $\mathbf{n}$ at a point v on a cylinder, we just project it into the plane perpendicular to the central axis $\mathbf{m}$ of the cylinder.

$$\begin{aligned} \mathbf{v1} &= \mathbf{v} - \mathbf{q} \\ \mathbf{n} &= \mathbf{v1} - (\mathbf{v1} \cdot m)\mathbf{m} \end{aligned}$$

```
package cs155.jray;
/**  this class represents a 3D cylinder with open ends  */

public class Cylinder3D extends Object3D
{
    public Point3D center;
    public Point3D direction;

    public double radius=0.0;
    public double height=0.0;

    public Cylinder3D (Point3D c, Point3D d, double r, double h, Material m)
     { center = c; direction = d.normalize();
       radius = r; height = h; insideMat = outsideMat = m;     }

     public RayHit rayIntersect(Ray3D r) {
      Point3D P = r.p, D=r.d;
      Point3D PC = P.subtract(center);
      Point3D a = D.subtract(direction.scale(direction.dot(D)));
      Point3D b = PC.subtract(direction.scale(direction.dot(PC)));
      double  A= a.dot(a),  B= 2*a.dot(b), C= b.dot(b) - radius*radius;
       double Discr = (B*B-4*A*C);
       boolean has_solution = (Discr>=0);
       if (!(has_solution)) return RayHit.NO_HIT;
       double SqrtDisc = Math.sqrt(Discr);
       double t1=(-B-SqrtDisc)/(2 *A), t2=(-B+SqrtDisc)/(2*A);
       // note t1<=t2 always
       Point3D p,V2;
       double h;
       if (t1>= epsilon) {
           p = r.atTime(t1); V2 = p.subtract(center); h = direction.dot(V2);
   if ((h>= 0) && (h <= height)) return new RayHit(t1,normal(p),this); }
       if (t2 >= epsilon) {
           p = r.atTime(t2); V2 = p.subtract(center); h = direction.dot(V2);
   if ((h>= 0) && (h <= height)) return new RayHit(t2,normal(p),this); }
       return RayHit.NO_HIT;
     }

      private Point3D normal(Point3D p) {
Point3D p1 = p.subtract(center);
return  p1.subtract(direction.scale(direction.dot(p1))).normalize();}
}
```

Figure 4.2: Cylinder3D.java

Chapter 5

Lighting Models

In the previous two chapters we developed several classes representing 3d objects. In this chapter we add lights to the program and introduce the three main lighting effects used in 3d graphics engines: ambient, diffuse, and specular illumination.

5.1 Ambient illumination

The ambient lighting model assigns a fixed intensity of light to any visible part of an object. This ambient intensity does not depend on the presence of any lights in the scene. It models the general light that bounces off the walls of a room and fills the space uniformly with a low level of light. Ambient light is handled in two places. First, it is a parameter of the scene and provides a light level that is used to illuminate all objects in the scene. Second, each light has an ambient parameter, which provides a low level of light for any objects that are visible to the light.

5.2 Diffuse illumination

Diffuse illumination models light that hits a rough surface and is scattered equally in all directions. The intensity of the light depends on the angle at which the light hits the surface. Thus light directly overhead will maximize the diffuse illumination whereas light coming from a point just over the horizon will be very weak.

To compute the diffuse illumination at a point $\mathbf{p}$ on a Sphere with center $\mathbf{c}$. We first compute the normal $N = \mathbf{p} - \mathbf{c}$ and then the vector which points from the point ot the light $\mathbf{L}$ which is $\mathbf{L} - \mathbf{p}$. Then we normalize both and take their dot product

$$\frac{(\mathbf{p} - \mathbf{c}) \cdot (\mathbf{L} - \mathbf{p})}{(|\mathbf{p} - \mathbf{c}|\ |\mathbf{L} - \mathbf{p}|)}$$

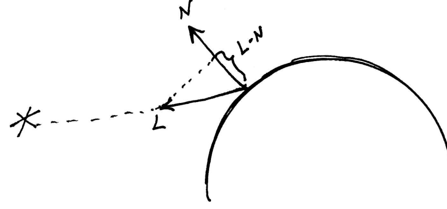


Figure 5.1: Diffuse illumination is the dot product of two normalized vectors

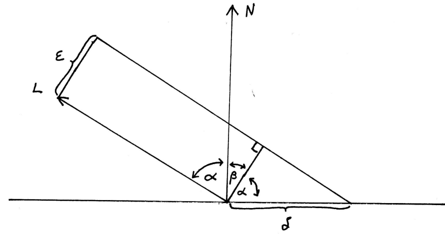


Figure 5.2: Diffuse illumination is smaller when one unit of light is spread over a larger region

5.3 Specular illumination

Specular illumination models the shininess of objects and is responsible for the bright spot on a shiny apple or the glint off the chrome bumper of a car. It is largest when the light bouncing off of the surface (assuming it was mirrored) would bounce directly into the camera and it falls off rapidly as the camera moves away from that point of maximum specular intensity. The rate at which it falls of is called the hardness. Lets now look at one way to model specular illumination.

Let $\mathbf{v}$ be a point on an object with normal $\mathbf{n}$ and let u be the unit vector that points to ward a light L and e the unit vector that points from $\mathbf{v}$ to the camera. Let P be the plane passing through $\mathbf{v}$ and with normal $\mathbf{n}$.

The Phong model of specular illumination is computed by taking the cosine of the angle between $\mathbf{e}$ and the reflection $\mathbf{u}'$ of in P. To compute u' we first project u onto $\mathbf{n}$ to get $\mathbf{u}_n$ and also into a vector u_P lying in P:

$$\begin{aligned}
\mathbf{u}_1 &= (\mathbf{u} \cdot \mathbf{n})\mathbf{n} \\
\mathbf{u}_P &= \mathbf{u} - (\mathbf{u}_1) \\
\mathbf{u}' &= \mathbf{u} - 2\mathbf{u}_P \\
&= 2(\mathbf{u} \cdot \mathbf{n})n - \mathbf{u}
\end{aligned}$$

The specular reflection, s, is then computed by calcuting $cos(\alpha)^n$ where α is the

angle between u' and e and h is the "hardness" of the material:

$$s = (\mathbf{u}' \cdot \mathbf{e})^h$$

The value h is typically between 0 and 128.

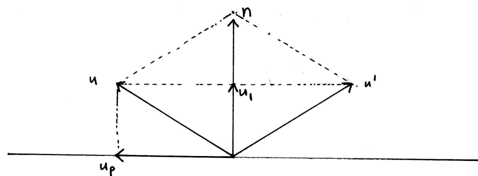


Figure 5.3: Calculating the reflection of **u** relative to the normal

The Blinn-Phong model is Another way to calculate the specular reflection. In this approach we find the vector w halfway between u and e and then to use the angle β between w and the normal n rather than α. This gives as value s' which is another type of specular illumination:

$$\begin{aligned} w &= \frac{\mathbf{u}+\mathbf{e}}{|\mathbf{u}+\mathbf{e}|} \\ s' &= (\mathbf{w} \cdot \mathbf{n})^h \end{aligned}$$

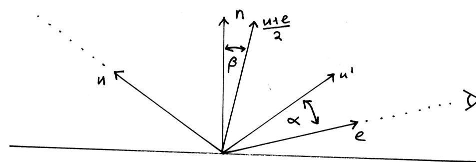


Figure 5.4: Specular illumination

5.4 Lights

The lights in this version of the ray tracer are specified by giving their positions and intensities. They also have methods to compute the diffuse and specular lighting intensities. The diffuse intensity depends on the normal to the object and the vector from the intersection point to the light. The specular intensity also needs to vector from the intersection point to the camera and the harndess coefficient of the material.

```
package cs155.jray;
import java.awt.*;
import javax.swing.*;
/**   This represents simple  3D lights with location and intensity **/

public class Light3D {
    public Point3D location = new Point3D(0d,0d,0d);
    public double intensity = 1.0;

    public Light3D( Point3D new_location ){ this(new_location,1d);  }
    public Light3D( Point3D new_location, double intensity ){
        location= new Point3D(new_location); this.intensity = intensity;}

    /**   calculate the specular intensity of the light    **/
    public double specular(Point3D lightVec, Point3D normal,
                           Point3D eyeVec, int hardness) {
        Point3D u = lightVec.normalize();
        Point3D e = eyeVec.normalize();
        Point3D w = u.add(e).normalize();
        if (u.dot(normal) > 0)
           return Math.pow(w.dot(normal), hardness);
        else  return 0.0;    }

    /**  calculate the diffuse intensity    **/
    public double diffuse (Point3D lightVec, Point3D normal){
      double cosA = normal.dot(lightVec);
      if (cosA < 0) return 0.0;  else return (cosA);   }
}
```

Figure 5.5: Light3D.java

Chapter 6

Implementing a Basic Ray Tracer

In this chapter we show how to implement a simple ray tracer based on the concepts presented in the previous chapters. This raytracer will allow us to generate images of sphere, cylinders, and planes with diffuse illumination providing depth cues. In later chapters we will add color and textures which greatly add to the realism. We will also, introduce transforms to that the lights and camera can easily be positioned about the scene and the objects can be transformed in various ways. Our first raytracer will consist of the following classes, several of which we have described in previous chapters.

```
Point3D.java  - a point or vector in 3D space
Ray3D.java - a ray consisting of an origin point and a direction vector
Camera3D.java - a camera that converts pixel coordinates into rays
Light3D.java - a light with a specified position
Object3D.java - this is an abstract class for all objects
Group3D.java - a group of objects
Scene3D.java - a scene consisting of a camera, several lights, and a group of objects
Sphere3D.java - the Sphere subtype of Object3D
Cylinder3D.java - the Cylinder subtype of Object3D
Plane3D.java - the Plane subtype of Object3D
Material3D.java -  material properties (color, reflectivity, etc.)
RayTracer3D.java - the main class raytracing class
```

In this chapter we will present and describe the code for these classes..

6.1 Cameras

The camera in this version of the program simply points down the z-axis from the origin and is used to generate rays corresponding to virtual screen positions. Indeed, the method `generateRay(u,v)` is given two doubles `u,v` and returns

```
package cs155.jray;
/**  This class represents a camera at (0,0,0) facing toward (0,0,-1). **/
public class Camera3D {
    private Point3D origin = new Point3D(0d,0d,0d);
    public double screenDist = -2d;

    public Camera3D () {;}

    /**  Convert screen coordinates (represented as two doubles) into a ray **/
    public Ray3D generateRay(double u, double v) {
      return  new Ray3D(origin, new Point3D(u,v,screenDist));  }
}
```

Figure 6.1: Camera3D.java

the ray centered at the origin and passing through the point `(u,v,-D)` where `D` is the distance of the virtual screen from the camera eye, and is intially 2. Increasing D will zoom the camera in, while moving it toward zero produces a fisheye effect.

6.2 The Scene class

The Scene3D class represents a scene as consisting of three components:

- a group `objs` of objects represented by a `Group3D` element
- an array of lights
- a camera

It also encapsulates several parameters describing the scene:

- a background color, for rays that do not intersect any objects
- an oversampling parameter, OSI, which is used for antialiasing
- an ambient color, representing the general "directionless" light in the scene that is not produced by any positioned light

This class also provides methods for adding cameras, objects, and lights to the scene and for clearing the scene of all elements.

6.3 The Canvas3D class

Once we have defined a scene, we need a place to draw the 2d projection of the scene. This role will be filled by any object that implements the Canvas3D

```
package cs155.jray;
/**  A 3d scene that can be drawn on a Canvas3D object. **/
public class Scene3D {
    /** Here we create the scene elements as instance variables **/
    public Group3D objs = new Group3D();
    public int numObj = 0;
    public Light3D light[] =new Light3D[100];
    public int numLights=0;
    public Color3D ambient = new Color3D(0.1,0.1,0.1);
    public Color3D backgroundColor = new Color3D(0d,0d,0.4d);
    public Camera3D camera = new Camera3D();
    public int OSI = 1;
    public Scene3D() {; }
    public void add(Camera3D x) {camera = x;  }
    public void add(Object3D x) {ojs.add(x);  }
    public void add(Light3D x) {light[numLights++]=x;  }
    public void clear() {objs.clear(); numLights=0;  }
}
```

Figure 6.2: Scene3D.java

interface in Figure 6.3. In the appendix we provide an implementation of this interface, `NewRayCanvas3D`, but since its implementation has little to do with 3D graphics we do not go into those details here.

The Canvas3D interface provides methods for finding the height and width of the canvas, for drawing a color on a particular pixel, and for refreshing the canvas which makes all changes to the canvas appear on the screen.

6.4 The RayTracer3D class

The RayTracer3D class is where everything comes together. It provides a method for drawing a 2D projection of a 3D scene on a Canvas3D object. In

```
package cs155.jray;
/** a canvas is any Java object one can draw Pixels on **/
public interface Canvas3D {
    public int height();
    public int width();
    public void drawPixel(int i, int j, java.awt.Color c);
    public void refresh();
}
```

Figure 6.3: Canvas3D.java

our implementation we decompose it into a few methods as shown in the figures below.

6.5 A Simple GUI

Briefly describe the NewRayCanvas class and show how it can be embedded in a frame.

6.6 Examples

```
package cs155.jray;
import java.awt.*;
import javax.swing.*;


/**   This consists of static methods for drawing a scene on a canvas. **/

public class RayTracer3D {
    public static int recursionDepth=4;
    public static RayHit intersectScene(Scene3D s, Ray3D r) {
return s.objs.rayIntersect(r);    }

  public static void drawScene(Scene3D s,Canvas3D canvas) {
        double
          h = canvas.height(), w = canvas.width();

        for(int i=0;i<w; i++) {
          for(int j=0; j<h; j++)  {
            double u,v;
            u = (double)((i-w/2)/(w/2));
            v = (double)((h/2-j)/(w/2));

            // calculate the color of the pixel corresponding to that ray
            Color3D pixelColor = Color3D.BLACK;
            for (int k=0; k< s.OSI; k++) {
Ray3D r1 =
                   s.camera.generateRay(u+(Math.random()-0.5)/(w/2),
                                        v+(Math.random()-0.5)/(h/2));
Color3D localColor = computeColor(r1,s,recursionDepth);
pixelColor = pixelColor.add(localColor);
            }
            pixelColor = pixelColor.scale(1.0/s.OSI);
    canvas.drawPixel(i,j,pixelColor.toColor());
         } // close for i
       } // close for j
     } // close drawScene
```

Figure 6.4: RayTracer.java

```
    /**  Compute the color corresponding of the pixel  **/
    public static Color3D computeColor(Ray3D r, Scene3D s, int depth) {
      // first we intersect the ray with the scene
      RayHit hit = intersectScene(s,r);
      Point3D n = hit.normal;
      Object3D obj = hit.obj;

      // if the ray hits no object, return the background color
      if (hit.distance == Double.POSITIVE_INFINITY)  return s.backgroundColor;
      Point3D p = r.atTime(hit.distance);

      // if the ray hits the outside of the object
      // then switch the normal to point inward
      boolean outside = (n.dot(r.d) < 0);
      if (!outside) n = n.scale(-1d);  // flip the normal when the ray hits on the insi
      Material m = (outside?obj.outsideMat:obj.insideMat);

      // calculate the initial pixel color
      Color3D pixelColor = Color3D.BLACK;

      for (int k = 0; k<s.numLights; k++) {
          Color3D localColor = Color3D.BLACK;
          Light3D light = s.light[k];
          Point3D lightPos = light.location;
          Point3D lightVec = lightPos.subtract(p);
          double lightDist = lightVec.length(); // distance from point to the light
          lightVec = lightVec.normalize(); //vector from point to the light
          Ray3D LR = new Ray3D(p,lightVec);

          // check for shadows
          RayHit lightHit = intersectScene(s,LR);
          if (lightHit.distance < lightDist-Object3D.epsilon) continue;

          // calculate the local diffuse pixel color contribution
          double diffuseIntensity = light.diffuse(lightVec,n);
          localColor  = (m.color).times(light.color).scale(diffuseIntensity);

          // calculate the local specular pixel color contribution
          Point3D eyeVec = r.p.subtract(p).normalize();
          double specularIntensity = light.specular(lightVec,n,eyeVec,m.hardness);
          Color3D specularColor  =
            localColor.add((m.color).times(light.color).scale(specularIntensity));
          localColor = localColor.add(specularColor);
          localColor = localColor.scale(light.intensity);
          // finally add this local pixel contribution to the accumulated pixel color
          pixelColor = pixelColor.add(localColor);
       }
        return pixelColor;
    }
}
```

Figure 6.5: RayTracer.java

Chapter 7

Expanding the Lighting Model

THIS SECTION IS UNDER CONSTRUCTION ...

7.1 Color

Discuss OpenGL color model: ambient, diffuse, specular colors for the lights and materials. emissive colors for the materials, attenuation, spot light effects, ambient light of the scene itself. Reference the OpenGL manual.

Show how to change the code for Material, Light3D, and RayTracer to allow these features.

Chapter 8

Reflection and Refraction

8.1 Reflection

To find the reflection of a ray $\mathbf{d}$ at a point $\mathbf{q}$ of a plane P with normal $\mathbf{n}$. We first project $\mathbf{d}$ onto the normal to get

$$\mathbf{d}_n = (\mathbf{d} \cdot \mathbf{n})\mathbf{n}$$

This vector represent how far beneath the plane the ray would go if it passed straight through the plane. To get the reflection we add twice $\mathbf{d}_n$ to $\mathbf{d}$ and get the reflection direction:'

$$\mathbf{d}' = \mathbf{d} - 2\mathbf{d}_n$$

The reflecting ray starts at point $\mathbf{q}$ and goes in direction $\mathbf{d}'$. Note that this

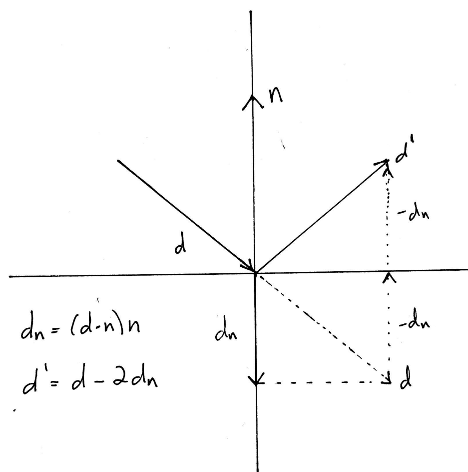


Figure 8.1: Reflection of a Ray in a Surface

formula is different than the one we obtained in the previous section when

computing the specular reflection, because in that case we used a vector pointing toward the light which was in the same direction as the normal. In this section we are considering a vector pointing into the plane in the opposite direction of the normal, if you let $\mathbf{e} = -\mathbf{d}$, and substitute into the formula in this section, you'll get the same formula as in the previous section.

8.2 Refraction

Light travels at different speeds in different media. The index of refraction (IOR) of a material is a number related to the speed of light in that material. hen light passes from one region to another with a different index of refraction (IOR) the light ray is bent in a way that depends on the two IORs. Indeed, if we let a_1 be the angle between the entering ray and the outward facing normal $\mathbf{N}$, and let a_2 be the angle between the exiting ray and the inward facing normal, and if we let n_1 and n_2 be the two indices of refraction, then Snell's Law (shown below) describes the relationship between these quantities:

$$n_1 \sin(a_1) = n_2 \sin(a_2)$$

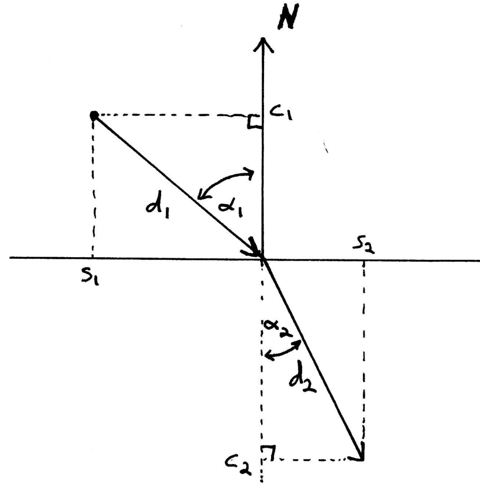


Figure 8.2: Refraction

Let $\mathbf{d}_1$ be the direction of the entering ray and $\mathbf{d}_2$ be the direction of the exiting ray. If the IORs are equal, then $\mathbf{d}_1 = \mathbf{d}_2$. Otherwise, we will usually know $\mathbf{d}_1$ and want to compute $\mathbf{d}_2$. The key to deriving this relation is by observing that we can compute $c_1 = \cos(a_1)$ as a dot product between $\mathbf{d}_1$ and $-\mathbf{n}$. This allows us to compute $s_1 = \sin(a_1)$ since $s_1^2 + c_1^2 = 1$

$$c_1 = \mathbf{d}_1 \cdot -\mathbf{N}$$

$$s_1 = \sqrt{1 - c_1^2}$$

Next, let $s_2 = \sin(a_2)$ and $c_2 = \cos(a_2)$ and $n = n_1/n_2$, then $s_2^2 + c_2^2 = 1$ and from Snell's Law we know that $s_2 = ns_1$, so we can calculate s_2 and c_2 as

$$\begin{aligned} s_2 &= ns_1 \\ c_2 &= \sqrt{1 - s_2^2} \\ &= \sqrt{1 - n^2(1 - c_1^2)} \\ &= \sqrt{1 - n^2 + n^2 c_1^2)} \end{aligned}$$

We can now express the refracted array in terms of two components. The component in the normal direction is $-c_2\mathbf{n}$ and the other part is the component in the plane P. To get this we must first project $\mathbf{d}_1$ into P to get a vector $\mathbf{f}_1$ and normalize it to get $\mathbf{f}_1/s_1$ of length one:

$$\begin{aligned} \mathbf{f}_1 &= \mathbf{d}_1 - (\mathbf{d}_1 \cdot (-\mathbf{N}))(-\mathbf{N}) \\ &= \mathbf{d}_1 - (\mathbf{d}_1 \cdot \mathbf{N})\mathbf{N} \\ &= \mathbf{d}_1 + c_1\mathbf{N} \end{aligned}$$

and so

$$\begin{aligned} \mathbf{d}_2 &= c_2(-\mathbf{N}) + s_2\mathbf{f}_1/s_1 \\ &= -c_2\mathbf{N} + s_2/s_1(\mathbf{d}_1 + c_1\mathbf{N}) \\ &= -c_2\mathbf{N} + n\mathbf{d}_1 + nc_1\mathbf{N} \\ &= n\mathbf{d}_1 + (nc_1 - c_2)\mathbf{N} \end{aligned}$$

This gives us our final formula for the refraction direction $\mathbf{d}_d$ of a ray with direction $\mathbf{d}_1$

$$\mathbf{d}_2 = n\mathbf{d}_1 + (nc_1 - c_2)\mathbf{n}$$

8.3 Changes to RayTracer3D

Show the minimal changes we need to make to Ray Tracer3D to implement reflections and refactions.

Chapter 9

Transforms

One of the most useful tools in computer graphics is the notion of a transform and the observation that one can easily find the intersection of ray with the transform of an object, by intersecting the original object with the inverse transform of the ray, and then applying the transform to the resulting point. In this section we show how to represent a wide class of transforms as 4x4 matrices. In the next section, we'll show how to compute the intersection of a ray with a transform of an object, and to complete the normal at that point as well.

9.1 Fundamental Transforms

We will look at three classes of transforms: translations, rotations, and scalings, and we will show how to represent all of these transforms as 4x4 matrices. Composing these transforms can be done by multiplying their corresponding matrices and this gives us a compact way of representing arbitrary combinations of translations, rotations, and scales. The class of transforms that can be represented by 4x4 matrices are called the affine transforms and they are heavily used in Computer Graphics.

9.1.1 Translation

For any point p in 3D space, the translation T_p is a function that maps a point q to $p+q$:

$$T_p(q) = p + q$$

or in terms of vectors if $p = (a, b, c)$ then

$$T_p(x, y, z) = (x + a, y + b, z + c)$$

We represent T_p by a 4x4 matrix:

$$T_p = \begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

To apply this matrix to a point (x, y, z) we embed it in 4d space by adding a 1 in the fourth coordinate and then apply the 4x4 transformation matrix to this 4-tuple as usual:

$$T_p(x, y, z) = \begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + a \\ y + b \\ z + c \\ 1 \end{pmatrix}$$

To apply this matrix to a direction (x, y, z) such as the direction **d** in a ray, we embed it in 4d space by adding a 0 in the fourth coordinate and then apply the 4x4 transformation matrix to this 4-tuple as usual:

$$T_p(x, y, z) = \begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix}$$

and we see that translating a direction in space does not change its value, which is as we would expect.

9.1.2 Scaling

For any triple $p = (a, b, c)$, the scaling matrix S_p is the following function

$$S_p(x, y, z) = (a * x, b * y, c * z)$$

We represent S_p by a 4x4 matrix:

$$S_p = \begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

9.1.3 Rotation

For any angle a, let $s = \sin(a)$ and $c = \cos(a)$. Then the rotation transformation $R_{z,a}$ that rotates the point q around the z-axis by the angle a is given by

$$R^z_a(x, y, z) = (c * x - s * y, s * x + c * y, z)$$

which is represented by the following 4x4 matrix:

$$R_a^z = \begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Similary we can rotate around the y and x axes:

$$R_a^y = \begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and

$$R_a^x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

9.1.4 Compound Transformations

A general 4x4 matrix T represents a transform on both points $p = (x, y, z, 1)$ in 3D space and directions $p = (x, y, z, 0)$ as follows. First we apply T to the point $p = (x, y, z, w)$ where w is 1 for a point and 0 for a direction. This gives us some vector $q = (a, b, c, d)$. If d is non-zero then

$$T(x, y, z) = (a/d, b/d, c/d, 1)$$

otherwise $T(p)$ is a direction which we can think of as the "point at infinity in the direction $(a, b, c, 0)$".

This space consisting of both regular points $p = (x, y, z, 1)$ and points at infinity $p' = (x, y, z, 0)$ is called Projective 3 space. It can also be represented as the equivalence classes of points in 4D space where two points are equivalent if one is a non-zero multiple of another.

$$(x, y, z, w) \equiv (\lambda x, \lambda y, \lambda z, \lambda w)$$

for all real numbers $\lambda \neq 0$. The representation of a 3D point or direction as a 4-tuple of numbers is called homogeneous coordinates.

For example, lets consider the operation that rotates a point p through the vector parallel to the z-axis that passes through the point q. We can represent this as a composition of three operators. First translate by $-q$. This moves q to the origin. Then rotate about the $z - axis$, then translate by q which moves the origin back to q. Thus, this operation is

$$M_{q,a}^z = T_q R_a^z T_{-q}$$

and if $q = (u, v, w)$ and $s = \sin(a)$ and $c = \cos(a)$, then we have

$$\begin{aligned} M^z_{q,a} &= \begin{pmatrix} 1 & 0 & 0 & u \\ 0 & 1 & 0 & v \\ 0 & 0 & 1 & w \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -u \\ 0 & 1 & 0 & -v \\ 0 & 0 & 1 & -w \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} c & -s & 0 & (1-c)u + sv \\ s & c & 0 & -su + (1-c)v \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

and this gives a matrix representation for a fairly complex compound operation.

9.1.5 The general rotation around a vector v

Let $\mathbf{v} = (v_x, v_y, v_z)$ be a non-zero vector and let a be an angle. We want to find a matrix that rotates points around an axis specified by the vector v. For example, when $v = (0, 0, 1)$ this should just be the R^z_a matrix that rotates around the z-axis.

There is a very clever way of constructing this matrix which we describe below. Lets think of v as a column vector:

$$\mathbf{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$

and lets normalize $\mathbf{v}$ so that it has length 1. That is $\mathbf{v}^T * \mathbf{v} = v_x^2 + v_y^2 + v_z^2 = 1$. Next let $v^\flat$ be the matrix such that for any vector $\mathbf{w}$, we have $v^\flat \cdot w = v \times w$, that is

$$\mathbf{v}^\flat = \begin{pmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{pmatrix}$$

Next, let $R = v * v^T$ be the dense 3x3 matrix obtained by multiplying $\mathbf{v}$ with itself

$$R = v * v^T = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} * (v_x v_y v_z) = \begin{pmatrix} v_x^2 & v_x v_y & v_x v_z \\ v_y v_x & v_y^2 & v_y v_z \\ v_z v_x & v_z v_y & v_z^2 \end{pmatrix}$$

and let I be the identity matrix as usual (i.e. the matrix with 1's down the diagonal and 0's elsewhere:

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Finally, let $c = \cos(a)$ and $s = \sin(a)$ and define M to be the following matrix:

$$M = v * v^T + c(I - v * v^T) + s * v^\flat$$

and so $M =$

$$
\begin{aligned}
&= v * v^T + c(I - v * v^T) + s * v^\flat \\
&= \begin{pmatrix} v_x^2 & v_x v_y & v_x v_z \\ v_y v_x & v_y^2 & v_y v_z \\ v_z v_x & v_z v_y & v_z^2 \end{pmatrix} + \begin{pmatrix} c(1 - v_x^2) & -c v_x v_y & -c v_x v_z \\ -c v_y v_x & c(1 - v_y^2) & -c v_y v_z \\ -c v_z v_x & -c v_z v_y & c(1 - v_z^2) \end{pmatrix} \\
&\quad + \begin{pmatrix} 0 & -s v_z & s v_y \\ s v_z & 0 & -s v_x \\ -s v_y & s v_x & 0 \end{pmatrix} \\
&= \begin{pmatrix} c + (1 - c) v_x^2 & (1 - c) v_x v_y - s v_z & (1 - c) v_x v_z + s v_y \\ (1 - c) v_y v_x + s v_z & c + (1 - c) v_y^2 & (1 - c) v_y v_z - s v_x \\ (1 - c) v_z v_x - s v_y & (1 - c) v_z v_y + s v_x & c + (1 - c) v_z^2 \end{pmatrix}
\end{aligned}
$$

and this gives a relatively simple matrix representation for a general rotation about a general non-zero vector v.

9.1.6 Proof of the general rotation formula

Now let p be any point, and let w be the projection of p into the plane perpendicular to v. Then $p = v + w$ and $v^T * w = 0$. Finally, let $u = v \times w$. We will see that

$$M(v + w) = v + cw + su$$

which shows that M rotates p by an angle a around v.

Proof: First observe that $M(v) = v$ using $v * v^T * v = v * 1 = v$ and $v^\flat * v = v \times v = 0$. Indeed,

$$
\begin{aligned}
M(v) &= (v * v^T + c(I - v * v^T) + s v^\flat) * v \\
&= (v * v^T) * v + c(I - v * v^T) * v + s v^\flat * v \\
&= v * (v^T * v) + cI * v - cv * v^T * v + s v^\flat * v \\
&= v * (v \cdot v) + (cv - cv * (v \cdot v)) + s(v \times v) \\
&= v + (cv - cv) + 0 \\
&= v
\end{aligned}
$$

Next observe that $M(w) = cw + su$ using $v^T * w = 0$ and $v^\flat * w = v \times w = u$. Indeed,

$$
\begin{aligned}
M(w) &= (v * v^T + c(I - v * v^T) + s v^\flat) * w \\
&= v * v^T * w + cw - cv * v^T * w + s v^\flat * w \\
&= v * (v \cdot w) + cw - cv * (v \cdot w) + s(v \times w) \\
&= v * (0) + cw - cv * (0) + su \\
&= cw + su
\end{aligned}
$$

as we set out to show.

9.2 Applying a transform to a ray

The use of 4x4 matrices allows us to represent translations, rotations, and scaling operations as matrices provided we represent points in 3d space as 4-tuples where the last coordinate is 1. This representation is called homogeneous coordinates. It has another advantage and that is that by setting the last coordinate to 0, we can represent directions, and these are then handled correctly by the usual matrix operations, that is, rotations and scaling operations modify the direction, but translations have no effect. We can then apply a transform to a ray by simply representing the point p and the direction d of the ray in homogeneous coordinates and then applying the transform directly to p and d respectively.

So if $R = \mathbf{p}t + \mathbf{d}$ is a ray with $\mathbf{p} = (x, y, z, 1)$ and $\mathbf{d} = (a, b, c, 0)$, and T is an affine transform, then $T(R) = T(\mathbf{p})t + T(\mathbf{d})$.

9.3 Intersecting a ray with a transformed object

Given an object X and a ray R with origin p and direction d, let $f_X(R) = t$ be the distance from p to the 1st intersection of R with X, and let $n_X(R)$ be the normal to the surface of the object X at that intersection point.

If T is a transform, then $T(X)$ is the set of all points $\{T(x) : x \in X\}$.

For example, if X is a sphere of radius 1 centered at the origin, then $R(X) = X$ for any rotation R. If $T = T_p$ is a translation, then $T_p(X)$ is the sphere of radius 1 centered at p. If $S_{(a,b,c)}$ is a scaling operator, then $S_p(X)$ is an ellipsoid whose x,y, and z axes have lengths a, b, c respectively.

To compute the intersection of a ray R with $T(X)$ we can generate the transformed ray $R' = T(R)$ and calculate the distance $t' = f_X(R')$ but this will not in general be equal to $f_{T(X)}(R)$.

For example, consider the scaling operator that multiplies all coordinates by e, that is $S_e(x, y, z) = (ex, ey, ez)$. The inverse operation is $S_{1/e}$. Let R be the ray starting at the origin and extending down the z-axis. Let X be the plane perpendicular to the ray which is 1 unit away from the origin. Then $S_e(X)$ is the plane passing through $(0, 0, -e)$, but $S_{1/e}(R) = R$ because scaling doesn't change the point $(0, 0, 0)$ and it doesn't change the direction (it just lengthens the direction vector by a factor of e). Thus, $f_{S_e(X)}(R) = e$ but $f_X(S_{1/e}(R)) = 1$.

The correct formula is

$$f_{T(X)}(R) = f_X(T^{-1}(R)) * \frac{|R.d|}{|T^{-1}(R.d)|}$$

where $R.d$ is the direction vector for the ray R.

The calculation of the normal is also somewhat complex

$$n_{T(X)}(R) = ((T^{-1})^t) * n_X(T^{-1}(R))$$

where M^t is the transpose of the matrix M. To see where this comes from, let n be the normal to X at a point p and let v be a vector perpendicular to n at p. Then v is in the tangent plane P and as one takes smaller and

smaller neighborhoods near the point p, the surface X more and more closely approximates that tangent plane. If T is a transform of the object, then $T(P)$ is the tangent plane to $T(X)$ at $T(p)$, but the normal to that plane is not $T(n)$. Indeed, since we know that the dot product of n and any vector v in P is zero as the tangent plane is perpendicular to n, so

$$n^t * v = 0$$

From this it follows that

$$n^t * I * v = n^t * T^{-1} * T * v = 0$$

as $T^{-1} * T = I$ and $I * v = v$. Now, since $n^t * T^{-1} = \left((T^{-1})^t * n\right)^t$ we see that

$$\left((T^{-1})^t * n\right)^t * (T * v) = 0$$

for all v in the tangent space to P. Thus, the normal to $T(P)$ must be given by

$$n_{T(X)}(R) = (T^{-1})^t * n_X(T(R))$$

9.4 Exercises

Let $h = \frac{\sqrt{2}}{2}$, so $h = \sin(\pi/4)$.

1. Write out the transformation matrix R_z which rotates the world around the z axis by 45 degrees.
2. Write out the transformation matrix R'_z which rotates the world around the z axis by -45 degrees.
3. Write out the transformation matrix R_y which rotates the world around the y axis by 45 degrees.
4. Write out the transformation matrix T which translates the origin to the point $(1, 2, 3)$
5. Write out the transformation matrix S which scales (x, y, z) to $(2x, 2y, 3z)$
6. Compute the matrix product $R_z R'_z$.
7. Compute the matrix products $R_y R_z$ and also $R_z R_y$? Are they equal? If not, how are they different?
8. Compute the matrix products $R_z S$ and $S R_z$? Are they equal?
9. Apply the matrix ST to the vector $(-2, 3, 1)$

9.5 Changes to RayTracer3D

Chapter 10

Textures

Photorealism is greatly enhanced by adding textures to the geometric objects and meshes in our raytracer. The approach described here is to associate a texture coordinate with each intersection point of a ray and a surface. The texture coordinate consists of a pair of floating point numbers which can then be mapped to a color based on the particular texture associated with the material.

The standard approach is to defining a texture map on the plane which associates an RGB color to any pair of floatin point numbers (x, y). We can then get a texture on an object by mapping each point $\mathbf{p}$ on the object to a point (x, y) in texture space, and using the texture color for that point.

10.1 Textures for a cube

The simplest way to create a texture map for a cube is to define the cube as a group of transformed unit squares $([0,1],[0,1],0)$ in the $z = 0$ plane. The texture coordinate for a point $(x, y, 0)$ is just (x, y)

10.2 Texture mapping for the plane

For the plane $z = 0$ we can easily map an intersection point $(x, y, 0)$ to the natural texture coordinate (x, y). For a general plane that passes through a point $\mathbf{q}$ with normal $\mathbf{n}$ we can get the same effect by specifying two orthogonal unit vectors in the plane (call them $\mathbf{r}$ for right and $\mathbf{u}$ for up and then mapping a point $\mathbf{p}$ in the plane to its coordinates relative to the right and up vectors:

$$\mathbf{p} \mapsto ((\mathbf{p} - \mathbf{q}) \cdot \mathbf{r}, (\mathbf{p} - \mathbf{q}) \cdot \mathbf{u})$$

We can construct $\mathbf{r}$ by projecting some vector, say $(1, 0, 0)$ into the plane through $(0, 0, 0)$ with normal $\mathbf{n}$ to get $\mathbf{r}$. We can then get $\mathbf{u}$ using the cross product

$$\mathbf{u} = \mathbf{n} \times \mathbf{r}$$

as this will be orthogonal to $\mathbf{n}$ and hence in the plane, and will be orthogonal to $\mathbf{r}$ so the texture will be applied without any slant! The one case where this approach may fail is when $(1, 0, 0)$ is nearlly perpendicular to the plane, but in this case we can project $(0, 1, 0)$ instead as the *right* vector.

10.3 Texture mapping for the cylinder

A cylinder is defined by the base plane (a point $\mathbf{q}$ and a normal $\mathbf{n}$) together with a radius R and a height H.

We can easily get a texture coordinate for an intersection point $\mathbf{p}$ on the cylinder by projecting $\mathbf{p}$ into the base plane of the cylinder to get a point $\mathbf{p}'$.

$$\mathbf{p}' = \mathbf{p} - ((\mathbf{p} - \mathbf{q}) \cdot \mathbf{n})\mathbf{n}$$

Since the projection is in the plane, it has a texture coordinate (x, y) where

$$\begin{aligned} x &= (\mathbf{p} - \mathbf{q}) \cdot \mathbf{r} \\ y &= (\mathbf{p} - \mathbf{q}) \cdot \mathbf{u} \end{aligned}$$

where $\mathbf{r}$ and $\mathbf{u}$ are the `right` and `up` vectors in the plane that define the texture of the plane as in the previous section. We can use the `atan2` Java method to convert this into an angle α between $-\pi$ and π radians.

$$\alpha = \texttt{Math.atan2}(y, x)$$

To get the natural texture coorindates for the cylinder, we can also compute the distance

$$h = (\mathbf{p} - \mathbf{q}) \cdot \mathbf{n}$$

between the two points $\mathbf{p}$ and $\mathbf{p}'$, and then we get the texture coordinate map

$$\mathbf{p} \mapsto (h/H, 1/2 + \alpha/(2\pi))$$

which has the effect of unrolling the cylinder onto the plane and rescaling so it fits in the rectangle $[0, 1] \times [0, 1]$.

10.4 Texture Coordinates for a sphere

For simplicity, lets first consider a sphere centered at the origin. The easiest way to get texture coordinates for the sphere is to circumscribe the sphere with a cylinder parallel to the y-axis and then project the sphere onto the cylinder using rays perpendicular to the y-axis. For example, let S be a sphere with center $\mathbf{q} = (0, 0, 0)$ and radius R and let $\mathbf{p} = (x, y, z)$ be a point on S. Then $x^2 + y^2 + z^2 = R^2$ and the projection of this point onto the cylinder of radius R perpendicular to the y axis is

$$(x/d, y/d, z)$$

where $d = \sqrt{x^2 + y^2}/R$ as then we have

$$(x/d)^2 + (y/d)^2 = (x^2 + y^2)/d^2 = R^2$$

We can then compute the texture coordinates for this point on the cylinder.

Another approach is to enclose the sphere in a box and project the sphere on the box using the normal vectors of the sphere (actually this works with any geomatric shape that can be enclosed in a box). A texture for the box then maps to a texture for the sphere.

10.5 Texture transforms

We have shown how to define an initial texture mapping for an object, but in practice one often wants to transform the texture mapping. For example, shifting it, rotating it, scaling it, shearing it. This can easily be implemented by defining a texture transform object which is a 2D affine transform that is applied to the texture coordinates before the color is computed.

Chapter 11

Scene Graphs

11.1 JScheme as a Scene Description Language

11.2 Sample Scenes

Appendix A

Solutions to Exercises

A.1 Chapter 2

Consider the following three points

$$\begin{aligned} a &= (1,2,2) \\ b &= (7,-2,4) \\ c &= (0,4,-2) \\ d &= (1,1,1) \end{aligned}$$

Let u be the vector from b to a, and let v be the vector from b to c. Let P be the plane that passes through d and is normal to a.

1. How long is the vector u?

$$\begin{aligned} u = a - b &= = (1,2,2) - (7,-2,4) = (-6,4,-2) \\ |u| &= \sqrt{|u \cdot|} = \sqrt{36+16+4} = \sqrt{56} \end{aligned}$$

2. Is the angle between u and v less than 90 degrees, equal to 90 degrees, or greater than 90 degrees? How do you know?
   Let θ be the angle between $\mathbf{u}$ and $\mathbf{v}$, and recall tha the dot product formula expresses the dot product of two vectors in term of their lengths and the cosine of the angle between them:

$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}|\ |\mathbf{v}| \cos(\theta)$$

   If $\cos(\theta) > 0$, then θ must be less than 90 degrees and if $\cos(\theta) < 0$ then θ is greater than 90 degrees. So we compute:

$$\begin{aligned} u &= (-6,4,-2) \\ v &= c - b = (0,4,-2) - (7,-2,4) = (-7,6,-6) \\ u \cdot v &= 42 + 24 + 12 = 78 > 0 \end{aligned}$$

and see that the angle is less than 90 degrees.

3. Calculate the normalization of the vector a, i.e. the unit vector pointing in the same directions as a.
Since $\mathbf{a} = (1,2,2)$, $|\mathbf{a}| = \sqrt{1+4+4} = 3$ so the normalization $\tilde{\mathbf{a}}$ is

$$\tilde{\mathbf{a}} = \frac{a}{3} = (\frac{1}{3}, \frac{2}{3}, \frac{2}{3})$$

4. How far away is the point b from the plane P?
To do this, we take the dot product of $w = \mathbf{b} - \mathbf{d}$ with $\tilde{\mathbf{a}}$

$$w = (7,-2,4) - (1,1,1) = (6,-3,3)$$

So

$$\mathbf{w} \cdot \tilde{\mathbf{a}} = (6,-3,3) \cdot (\frac{1}{3}, \frac{2}{3}, \frac{2}{3}) = 2 + (-2) + 2 = 2$$

So $\mathbf{b}$ is 2 units above P.

5. Find the point p which is the projection of b onto the plane P.
We have already calculated that $\mathbf{b}$ is 2 units above P where the notion of up and down is specified by the normal $\tilde{\mathbf{a}}$. Thus if we subtract t times the normal to P from b we will move it directly into the plane, and this is the projection:

$$\mathbf{p} = \mathbf{b} - ((\mathbf{b} - \mathbf{d}) \cdot \tilde{\mathbf{a}})\tilde{\mathbf{a}} = (7,-2,4) - 2(\frac{1}{3}, \frac{2}{3}, \frac{2}{3}) = (6\frac{1}{3}, -3\frac{1}{3}, 2\frac{2}{3})$$

6. Use the cross product to find a vector w which is perpendicular to u and v.

$$\begin{aligned} u &= (-6,4,-2) \\ v &= (-7,6,-6) \\ u \times v &= (u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_z v_x) \\ &= (-24 - -12, 14 - 36, -36 - -28) = (-12,-22,-8) \end{aligned}$$

and we can then check that $(4,2,-50)$ is perpendicular to u and v:

$$\begin{aligned} (-6,4,-2) \cdot (-12,-22,-8) &= 72 - 88 + 16 = 0 \\ (-7,6,-6) \cdot (-12,-22,-8) &= 84 - 132 + 48 = 0 \end{aligned}$$

7. Find the point e that you get by rotating the point c around the x axis by 90 degrees.
Rotating $c = (0,4,-2)$ around the x axis will keep the x coordinate the same but will move the y axis onto the z axis, and the z axis onto the $-y$ axis, thus it should go to $(0,2,4)$.

A.2 Chapter 9

Let $h = \frac{\sqrt{2}}{2}$, so $h = \sin(\pi/4)$.

1. Write out the transformation matrix R^z_{45} which rotates the world around the z axis by 45 degrees.

$$R^z_{45} = \begin{pmatrix} h & -h & 0 & 0 \\ h & h & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2. Write out the transformation matrix R^z_{-45} which rotates the world around the z axis by -45 degrees.

$$R^z_{-45} = \begin{pmatrix} h & h & 0 & 0 \\ -h & h & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3. Write out the transformation matrix R^y_{45} which rotates the world around the y axis by 45 degrees.

$$R^y_{45} = \begin{pmatrix} h & 0 & h & 0 \\ 0 & 1 & 0 & 0 \\ -h & 0 & h & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

4. Write out the transformation matrix T which translates the origin to the point $(1, 2, 3)$

$$T = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

5. Write out the transformation matrix S which scales (x, y, z) to $(2x, 2y, 3z)$

$$S = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

6. Compute the matrix product $R^z_{45} R^z_{-45}$.

$$R^z_{45} * R^z_{-45} = \begin{pmatrix} h & -h & 0 & 0 \\ h & h & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} h & h & 0 & 0 \\ -h & h & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} h^2+h^2 & h^2-h^2 & 0 & 0 \\ h^2-h^2 & h^2+h^2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

since $h^2 = \frac{1}{2}$

7. Compute the matrix products $R^y_{45} R^z_{45}$ and also $R^z_{45} R^y_{45}$? Are they equal? If not, how are they different?

$$R^y_{45} * R^z_{45} = \begin{pmatrix} h & 0 & h & 0 \\ 0 & 1 & 0 & 0 \\ -h & 0 & h & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} h & -h & 0 & 0 \\ h & h & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} h^2 & -h^2 & h & 0 \\ h & h & 0 & 0 \\ -h^2 & h^2 & h & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

while

$$R^z_{45} * R^y_{45} = \begin{pmatrix} h & -h & 0 & 0 \\ h & h & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} h & 0 & h & 0 \\ 0 & 1 & 0 & 0 \\ -h & 0 & h & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} h^2 & -h & h^2 & 0 \\ h^2 & h & h^2 & 0 \\ -h & 0 & h & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The diagonal values are the same but the off diagonals are transposed and possibly negated..

8. Compute the matrix products $R^z_{45} S$ and $S R^z_{45}$? Are they equal? They are both equal to

$$\begin{pmatrix} 2h & -2h & 0 & 0 \\ 2h & 2h & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

9. Apply the matrix ST to the vector $(-2, 3, 1)$

Lets first compute the matrix ST:

$$S * T = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 & 2 \\ 0 & 2 & 0 & 4 \\ 0 & 0 & 3 & 9 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Next, we apply this matrix to $(-2, 3, 1)$

$$\begin{pmatrix} 2 & 0 & 0 & 2 \\ 0 & 2 & 0 & 4 \\ 0 & 0 & 3 & 9 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} -2 \\ 3 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} -4 + 2 \\ 6 + 4 \\ 3 + 9 \\ 1 \end{pmatrix} = \begin{pmatrix} -2 \\ 10 \\ 12 \\ 1 \end{pmatrix}$$

Equivalently, we could translate the point $(-2, 3, 1)$ by $(1, 2, 3)$ to get $(-1, 5, 4)$ and then scale it by $(2, 2, 3)$ to get $(-2, 10, 12)$.