Computer Graphics: An Implementor's Guide

Luther A. Tychonievich

May 5, 2008 – March 8, 2010

Preface

Although only marginally interested in this subject, I was a teaching assistant for Brigham Young University's senior-level computer graphics class for four years, working with two professors and several hundred students. That experience taught me that most students misunderstand the same things. This booklet is designed to present ideas in such a way that those particular problems will not arise.

I am aware that the pedagogy and terminology in this booklet are not always those commonly in use. There are topics that people ought to know which are not here, and others that are breezed over rather glibly without citing the people who spent years of their lives developing the techniques discussed. This is intentional; my goal is to present ideas in a way that is accessible and leads to implementations which are easy to understand and maintain. I find that citing papers makes simple ideas seem less accessible to most undergraduates, and the most optimal solutions possible is usually a lot more brittle and difficult to get working than are slightly less optimal but much cleaner versions.

It is also possible that some things I say are just plain false. As I said, I am only marginally interested in graphics and have not researched much of this. I have tried most of it out myself and proven many facts to my own satisfaction, but anyone who still thinks proofs or testing validates an algorithm has not yet written enough algorithms. That said, I am not intentionally misleading and will happily correct errors or clarify misleading points if they are identified to me.

—Luther Tychonievich
M.S. in Computer Science, BYU, 2008
lty@cs.byu.edu

Contents

Chapter 1

Rasterization: Idea → Screen

1.1 Introduction to Rasterization

A raster display is a grid of pixels. Nearly all computers use raster displays, and generally when people talk about computer graphics they are talking about raster computer graphics; other types of computer graphics do exist but will not be discussed here.

Computer graphics is a family of techniques that fill a raster display with color values to create a picture of some "thing" defined inside the computer. Defining the "thing"s is a bit tricky. It is presently not practically possible to express the vast quantity of atoms that make up each object and then simulate the optics that make those objects visible. Instead, computer graphics works backwards: we figure out what we *can* draw, and then approximate objects using those building blocks. Because of this, we will discuss how to draw things you might not think you wanted to draw, such as triangles and conic sections; figuring out how to use these elements to create nice pictures will come later.

There are two major and several minor means of drawing something onto a raster display. While you could argue that "rasterization" ought to be a general term for all such techniques (and indeed some authors use it that way), in practice it is often used as the name of one of the two major methods. This method takes as input a description of an "object" to draw and produces, as output, a list of pixels covered by that object.

1.2 Normalized Device Coordinates → Device Coordinates

"Device coordinates" identify each pixel by a pair of integers; hence the display surface usually runs from $(0, 0)$ to $(width, height)$, though this may vary depending on the details of your screen. "Normalized device coordinates" identify each pixel by a pair of floating-point values such that the corners of the display surface have the same coordinates no matter what the surface is. Unfortunately, how the screen is normalized varies to some degree: some go from $(0, 0)$ to $(1, 1)$, others from $(-1, -1)$ to $(1, 1)$; directions also vary, each normalization using its own corner for $(1, 1)$. OpenGL uses $(-1, -1)$ in the bottom left by default.

You can get OpenGL emulate device coordinate by calling something like the following anytime the window changes size:

```
void resize(width, height) {
    glViewport(0, width, 0, height);
    glMatrixMode(GL_PROJECTION); glLoadIdentity();
    glMatrixMode(GL_MODELVIEW); glLoadIdentity();
    glOrtho(0, width, 0, height, -1, 1);
}
```

Then you can access a pixel x, y with color r, g, b (where 1 is full intensity) by `glColor3f(r, g, b); glVertex2f(x, y);`.

If you are using a 2D graphics library, there is probably some function like `setPixel(x,y,color)` which also accepts device coordinates.

If y_n is a normalized coordinate, y_{s_0} is the minimal pixel y (usually 0), and y_{s_1} is the maximal pixel y (usually the height of the window), the point-point line formula allows us to convert between normalized and non-normalized coordinates:

$$(y_s - y_{s_0})(y_{n_1} - y_{n_0}) = (y_n - y_{n_0})(y_{s_1} - y_{s_0}).$$

1.3 Terminology: vector $\neq$ vector

Computer graphics uses several different sub-disciplines of mathematics, which creates a few namespace collisions. Some of these I'll cleverly ignore, as the contexts are sufficiently different not to cause confusion. One we need to make explicit if the term "vector." In linear algebra a vector is an element of a vector space and, for our purposes, can be expressed as a finite ordered list of real numbers:

$$\begin{aligned}
\vec{x} &\triangleq (x_1, x_2, \ldots, x_n) \\
\vec{x} + \vec{y} &\triangleq (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n) \\
\vec{x} - \vec{y} &\triangleq (x_1 - y_1, x_2 - y_2, \ldots, x_n - y_n) \\
\vec{x} \otimes \vec{y} &\triangleq (x_1 \times y_1, x_2 \times y_2, \ldots, x_n \times y_n) \\
s\vec{x} \triangleq \vec{x} \times s &\triangleq (x_1 \times s, x_2 \times s, \ldots, x_n \times s) \\
\vec{x} \div s &\triangleq (x_1 \div s, x_2 \div s, \ldots, x_n \div s)
\end{aligned}$$

In geometry, there are several different uses of mathematical vectors, unfortunately including both Euclidean vectors and homogeneous vectors, all commonly called "vectors", as well as Euclidean and homogeneous points. There is, to my knowledge, no commonly accepted notation for distinguishing between these, so I will introduce my own: $\overrightarrow{x}$ for a Euclidean vector, $\overrightarrow{x}^h$ for a homogeneous vector, $\mathbf{x}$ for a Euclidean point, and $\mathbf{x}^h$ for a homogeneous point.

Throughout this chapter (except as noted), everything we know about a point we will dump into one big vector; for example, if we know position x, y, z, normal n_x, n_y, n_z, color r, g, b, opacity α, and texture coordinates u, v we have a 12-vector for the point $\vec{p} = (x, y, z, n_x, n_y, n_z, r, g, b, \alpha, u, v)$.

1.4 Rasterizing a Line Segment

Suppose you have a line from $\vec{p}_0$ to $\vec{p}_1$ that you want to display. The mathematically correct solution to this problem would be a blank screen—mathematical lines have no width and cannot be seen. What we want instead is either (1) a contiguous set of pixels which approximate the line or (2) the set of all the pixels through which the line passes. The first problem is easier, so we'll address it first.

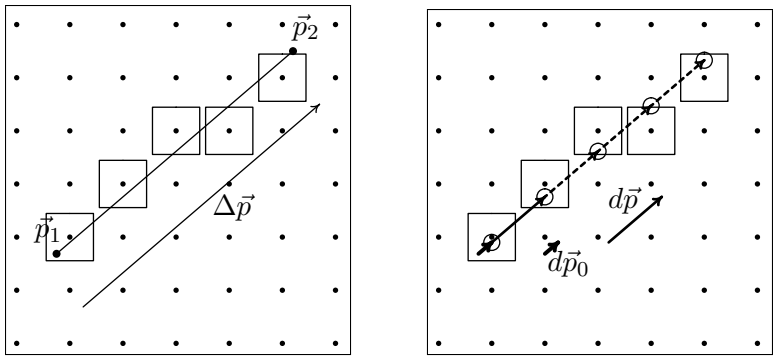

Figure 1.1: Rasterization of a line. Pixel locations are represented by dots; filled pixels by squares.

1.4.1 DDA Line Drawing

We here present a version of the Digital Differential Analyzer (DDA) line drawing algorithm, which draws a contiguous set of pixels approximating a line. By contiguous we mean that, for mostly horizontal lines, there is one pixel per column of pixels; for mostly vertical lines we want one pixel per row of pixels instead. This leads to the following algorithmic outline:

1. figure out if we step columns or rows,
2. find an increment to get from one column/row to the next,
3. find the first column/row, and
4. use the increment and rounding to generate all the pixels.

This process is illustrated at a high level in Figure 1.1, given formally in Algorithm 1.1, and discussed in more detail below.

Let's assume we have the two endpoints $\vec{p}_1$ and $\vec{p}_2$. First, we find the vector separating the endpoints, $\Delta\vec{p} = \vec{p}_2 - \vec{p}_1$. If $\Delta\vec{p}$'s x has a larger absolute value than its y, we step columns; otherwise we step rows. Let i be the index of that value; that is $\Delta p_i = x$ if we are stepping columns, $\Delta p_i = y$ if we are stepping rows. Let j be the index of the other xy coordinate.

If Δp_i is negative, swap $\vec{p}_1$ and $\vec{p}_2$, which causes every element in $\Delta\vec{p}$ to be negated.

The increment to move between pixels, $d\vec{p}$, must have the same direction as $\Delta\vec{p}$, but should be length 1 in the i direction. This is trivially created by dividing by the length we have:

$$d\vec{p} = \frac{\Delta\vec{p}}{\Delta p_i}$$

To find the first pixel, we want to round the ith component of $\vec{p}_1$ to the nearest pixel value. If we do this directly, however, we will generally change what line we draw, so we need to tweak the other coordinates as well. This is easy to do, however; if we want to change $\vec{p}_{1\,i}$ by r we can simply add $rd\vec{p}$ to $\vec{p}_1$. This observation results in

$$d\vec{p}_0 = (\lceil p_{1\,i} \rceil - p_{1\,i})\, d\vec{p}.$$

Now you want to draw the nearest pixel to the points $\vec{p}_1 + d\vec{p}_0$, $\vec{p}_1 + d\vec{p}_0 + d\vec{p}$, $\vec{p}_1 + d\vec{p}_0 + 2d\vec{p}$, etc., up to but not including $\vec{p}_2$. If you do include $\vec{p}_2$ then two lines sharing a common end-point might both fill that point, which is slightly inefficient and can create problems with transparency.

1.4.2 DDA Grid Walking

Sometimes we want to find all the pixels a line passes through instead of just an approximation. Given a particular $\vec{q}$, the line may pass through both $\lceil q_j \rceil$ and $\lfloor q_j \rfloor$, or it may pass through only one of them. If it passes through two then the point where the line passes between the two pixels is given

Algorithm 1.1 Basic DDA Algorithm

Input: points $\vec{p}_1$ and $\vec{p}_2$ in device coordinates
Purpose: draw a line conneting $\vec{p}_1$ and $\vec{p}_2$

1: $\Delta\vec{p} \Leftarrow \vec{p}_2 - \vec{p}_1$
2: **if** $|\Delta p_x| > |\Delta p_y|$ **then** — *step in columns*
3: $i \Leftarrow \text{indexof}(x)$
4: $j \Leftarrow \text{indexof}(y)$
5: **else** — *step in rows*
6: $i \Leftarrow \text{indexof}(y)$
7: $j \Leftarrow \text{indexof}(x)$
8: **if** $\Delta p_i < 0$ **then**
9: swap $\vec{p}_1$ and $\vec{p}_2$
10: $\Delta\vec{p} \Leftarrow -\Delta\vec{p}$
11: $d\vec{p} \Leftarrow \Delta\vec{p} \div \Delta p_i$
12: $d\vec{p}_0 = (\lceil p_{1\,i} \rceil - p_{1\,i})\, d\vec{p}$
13: $\vec{q} \Leftarrow \vec{p}_1 + d\vec{p}_0$
14: **while** $q_i < p_{2\,i}$ **do** — *note "<", not "≤"*
15: — *note q_i is already an integer; only q_j needs to be rounded...*
16: plotPixel(round(q_x), round(q_y), q_{color})
17: $\vec{q} \Leftarrow \vec{q} + d\vec{p}$

by the intersection of $j = \lfloor q_j \rfloor + 0.5$ and the line:

$$q_i + \frac{0.5 - q_j + \lfloor q_j \rfloor}{dq_j};$$

if this number is more than 0.5 away from q_i then the line only passes through one pixel.

This observation creates the following modification of Algorithm 1.1: replace line 16 with

offset $\Leftarrow \frac{0.5 - q_j + \lfloor q_j \rfloor}{dq_j}$
if |offset| < 0.5 **then**
plotPixel($\lfloor q_x \rfloor$), $\lfloor q_y \rfloor$, q_{color})
plotPixel($\lceil q_x \rceil$), $\lceil q_y \rceil$, q_{color})
else
plotPixel(round(q_x), round(q_y), q_{color})

You can also use the offset information to created an anti-aliased line; I'll leave this as an exercise for the reader.

1.5 Polygon Contents

To fill the pixels contained within a polygon, we use the DDA algorithm first in y along the edges, then in x between them. This really is the entirety of the polygon fill, as illustrated in Figure 1.2 and made formal in Algorthim 1.2.

Note that rasterizing a triangle works very nicely, but rasterizing other shapes is a bit of a misnomer. You can use the double DDA technique listed here for any polygon—we'll discuss how momentarily—but if not all the vertices are the same color this can create some really strange looking results.

That said, people often ask for arbitrary polygons to be filled. For rasterizing, either split them into triangles or use the DDA technique along with what is called the "edge table." An edge table is just a list of edges, where an edge is a

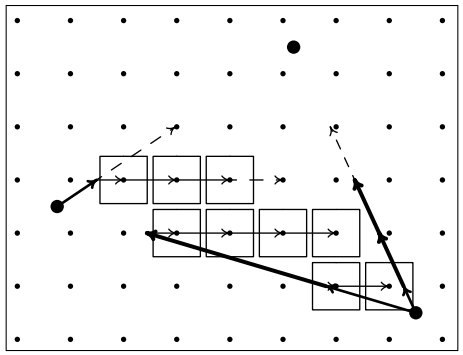

Figure 1.2: Rasterization of a polygon by first DDA-stepping up the edges, then DDA-stepping between the edges. In Algorithm 1.2, this image corresponds to being halfway through the DDA-line call on the first iteration of the loop at line 9.

pair of connected vertices. The "global edge table" is the list of all edges in the polygon. The "active edges" for a given value of y are the edges which cross that y. If you consider an edge to include its bottom endpoint's y but not its top, and ignore horizontal edges completely, then every y must have an even number of active edges.

The algorithm then runs as follows.

1. Begin at the smallest y in the polygon.
2. DDA up all active edges, of which there are always an even number.
3. At each y, DDA will give you a point for each active edge. Sort these in x, then DDA between pairs of points (1^{st} and 2^{nd}, 3^{rd} and 4^{th}, etc.).

Algorithm 1.2 Double-DDA Triangle Fill

Input: points $\vec{p}_b$, $\vec{p}_m$, and $\vec{p}_t$ in device coordinates
Purpose: draw all pixels inside the triangle

```
1: assert(p_b y ≤ p_m y ≤ p_t y)
2: Find dp⃗ and initial q⃗ for (p⃗_b, p⃗_m); call them dq⃗_a and q⃗_a
3: Find dp⃗ and initial q⃗ for (p⃗_b, p⃗_t); call them dq⃗_c and q⃗_c
4: while q⃗_a y < p_m y do
5:     DDA-Line(q⃗_a, q⃗_c)
6:     (q⃗_a, q⃗_c) ⇐ (q⃗_a + dq⃗_a, q⃗_c + dq⃗_c)
7: Find dp⃗ and initial q⃗ for (p⃗_m, p⃗_t); call them dq⃗_e and q⃗_e
8: while q⃗_e y < p_t y do
9:     DDA-Line(q⃗_e, q⃗_c)
10:    (q⃗_e, q⃗_c) ⇐ (q⃗_e + dq⃗_e, q⃗_c + dq⃗_c)
```

1.6 Extensions of DDA

The basic DDA-based line and polygon drawing routines are elegant and efficient, but are rather restricted in their application. This section lists a number of extensions, some very common, others more unusual.

1.6.1 Perspective-correct Interpolation

The basic technique discussed above interpolates everything linearly, which is great for 2D drawing, but for perspective drawing it does not preserve interiors of objects correctly (see Figure 1.3). Fortunately, there is a simple fix which uses linear interpolation to create perspective-correct results. The theory behind this method was first published in Jim Blinn's 1992 "Hyperbolic Interpolation" and is cleanly explained in Kok-Lim Low's 2002 paper "Perspective-Correct Interpolation," which is freely available online. I present only the

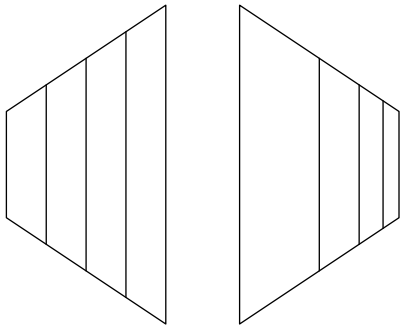

Figure 1.3: Equal divisions of a trapezoid, found by linear interpolation, and of a rectangle seen in perspective, found by hyperbolic interpolation.

details for implementation.

The standard perspective projection involves division by some depth value, which I will call w, such that we plot point x, y at screen position $\frac{x}{w}, \frac{y}{w}$. The trick to hyperbolic interpolation is to divide everything in the point by w except w itself, which we divide by w^2 instead. Thus if we had the point

$$\vec{p} = [x, y, z, w, n_x, n_y, n_z, r, g, b, u, v]$$

we would interpolate linearly using the point

$$\vec{p'} = \left[\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, \frac{1}{w}, \frac{n_x}{w}, \frac{n_y}{w}, \frac{n_z}{w}, \frac{r}{w}, \frac{g}{w}, \frac{b}{w}, \frac{u}{w}, \frac{v}{w}\right].$$

Then, at each point on the screen we would undo the division by w by dividing by the interpolated $\frac{1}{w}$.

To help clarify this process, consider a simple example of drawing a line with points in (x, y, w, r, g, b)-format.

$\vec{p}_0$:	$(-1, 0, 1, 1, 1, 0)$
$\vec{p}_1$:	$(3, 0, 3, 1, 0, 1)$
mapped to hyperbolic space	
$\vec{p_0}'$:	$(-1, 0, 1, 1, 1, 0)$
$\vec{p_1}'$:	$(1, 0, \frac{1}{3}, \frac{1}{3}, 0, \frac{1}{3})$
step in x	
$\Delta\vec{p}$:	$(2, 0, \frac{-2}{3}, \frac{-2}{3}, -1, \frac{1}{3})$
$d\vec{p}$:	$(1, 0, \frac{-1}{3}, \frac{-1}{3}, \frac{-1}{2}, \frac{1}{6})$
$d\vec{p}_0$:	$(0, 0, 0, 0, 0, 0)$
interpolated pixels	
$\vec{p_0}' + d\vec{p}_0$:	$(-1, 0, 1, 1, 1, 0)$
$\vec{p_0}' + d\vec{p}_0 + d\vec{p}$:	$(0, 0, \frac{2}{3}, \frac{2}{3}, \frac{1}{2}, \frac{1}{6})$
mapped back to Euclidean space	
	$(-1, 0, 1, 1, 1, 0)$
	$(0, 0, 1, \frac{3}{4}, 0, \frac{1}{4})$

Note that the resulting middle color is significantly more similar to the near point than it is to the far point, as we expect for perspective interpolation

1.6.2 DDA-like technique for Polynomials

DDA is based on the observation that for any linear function $f(t)$, $f(t+dt) - f(t)$ is the same value for all ts. We can write this as a difference table, where each element in the last row is the difference of the two elements above it, as follows:

t	0	1	2	3	4	5
$f(t) = 2t + 1$	1	3	5	7	9	11
$f(t+1) - f(t)$		2	2	2	2	2

For a quadratic function, the first difference row is itself linear and a second difference row will be constant:

t	1	2	3	4	5	6
$f(t) = t^2 - 1$	0	3	8	15	24	35
f_t 1		3	5	7	9	11
f_{tt}			2	2	2	2

In general, for a polynomial of order n, the n^{th} differences will be constant.

This observation allows us to draw any polynomial function using a DDA-like technique. For example, to plot $y = ax^3 + bx^2 + cx + d$ stepping in x from x_s to x_e we do the following:

1: Set $y_{i\in\{0,1,2,3\}} = a(x_s + i)^3 + b(x_s + i)^2 + c(x_s + i) + d$
2: Set $y'_{i\in\{0,1,2\}} = y_{i+1} - y_i$
3: Set $y''_{i\in\{0,1\}} = y'_{i+1} - y'_i$
4: Set $y'''_0 = y''_{i+1} - y''_i = 6a$
5: Set $x = x_s$
6: **while** $x < x_e$ **do**
7: Plot x, y_0
8: Add 1 to x
9: Add y'_0 to y_0
10: Add y''_0 to y'_0
11: Add y'''_0 to y''_0

There is also a similar tool for plotting implicit polynomial equations, as we shall see in the next section.

1.6.3 Drawing Implicit Curves

We just saw how to plot $y = P(x)$ by stepping in x, which is somewhat nice, but we can also plot implicit polynomial equations of the form $P(y, x) = 0$. The first thing we need to do this is to observe that the finite differences we have used are the discrete analog of the derivative and may be written out functionally; for example, the $f(t) = t^2 - 1$ noted above has a first difference of

$$f_t = f(t+1) - f(t) = ((t+1)^2 - 1) - ((t)^2 - 1) = t^2 + 2t + 1 - 1 - t^2 + 1 = 2t + 1$$

and a second difference of

$$f_{tt} = f_t(t+1) - f_t(t) = (2(t+1)+1) - (2(t)+1) = 2t + 2 + 1 - 2t - 1 = 2.$$

Because of this we can also define mixed differences as $f_{xy}(x, y) = f_x(x, y + 1) - f_x(x, y)$. To help solidify this idea, consider the differences for $f(x, y) = 2x^2y$:

$$\begin{aligned}
f_x &= 2y(x+1)^2 - 2y(x)^2 && = 4xy + 2y \\
f_y &= 2x^2(y+1) - 2x^2(y) && = 2x^2 \\
f_{xx} &= (4(x+1)y + 2y) - (4(x)y + 2y) && = 4y \\
f_{xy} &= (4x+2)(y+1) - (4x+2)(y) && = 4x + 2 \\
f_{yy} &= 2x^2 - 2x^2 && = 0 \\
f_{xxy} &= 4(y+1) - 4(y) && = 4 \\
f_{xxx} &= 4y - 4y && = 0
\end{aligned}$$

Note that the ordering of the differences does not matter; f_{yx} will always be the same as f_{xy}.

To plot implicit equations, we begin by evaluating the function and all of its differences at some point x, y. We can now find the value of the function at an adjacent point using DDA additions: To increment y we add f_{*y} to f_* in increasing order (e.g., f_x += f_{xy} before f_{xy} += f_{xyy}); to decrement it we subtract f_{*y} from f_* in decreasing order (e.g., f_{xy} += f_{xyy} before f_x += f_{xy}). Similarly, to increment x we add f_{*x} to f_* in increasing order, while to decrement it we subtract f_{*x} from f_* in decreasing order.

Plotting a curve then amounts to moving in x and y in order to keep f and $f + f_x$ (or f and $f + f_y$) of different signs. There are some optimizations and pathological cases

as well; see Van Aken & Novak, "Curve-drawing algorithms for raster displays" ACM TOG (April 1985) pp. 147–169 for a discussion.

1.6.4 Just Integers, No Division

We can pose the DDA technique without any division, and if we do we can also draw shapes with integer inputs using only integer math. To do this, notice for a DDA line stepping in y, the denominator of all coordinates is Δy. Thus, given $x = a\frac{b}{\Delta y}$ on one row, where $a = \lfloor x \rfloor$, on the next row it is $x' = a + \frac{b+\Delta x}{\Delta y}$. Let $i = \Delta x \div \Delta y$ and $r = \Delta x \pmod{\Delta y}$ and we can rewrite that as $x' = (a+i) + \frac{b+r}{\Delta y}$. This allows the following method for incrementing y:

1: Set $b = b + r$
2: Set $a = a + i$
3: **if** $b \geq \Delta y$ **then**
4: Set $a = a + 1$
5: Set $b = b - \Delta y$

If no division is available, we can also write

1: Set $a = a + \Delta x$
2: **while** $b \geq \Delta y$ **do**
3: Set $a = a + 1$
4: Set $b = b - \Delta y$

For software running on modern PC hardware, this method of avoiding divisions has little perceptible runtime difference from the straight DDA on floating-point data. However, for simpler embedded processors or custom-made hardware it is noticeably faster, and in all cases it avoids the roundoff error present in floating-point DDA.

1.6.5 About Names

We have discussed the DDA algorithm in vector form because that seems to be the cleanest version to understand. In graphics literature, DDA is usually presented in an element-by-element form instead; DDA-based polygon fill is called linear scan-line interpolation (and often presented without that $d\vec{p}_0$s, an omission that causes neighboring polygons to overlap), and the integer-only technique for lines is called the Bresenham line algorithm. The implicit curve plotting routine was published as the midpoint algorithm, named after an optimization where the next pixel is analytically reduced to one of two pixels and the sign at the midpoint of the two is used to pick which one to plot.

1.6.6 Example: Midpoint Circle Algorithm

This example walks through using the implicit DDA in the integer-only form with some additional optimizations, together known as the midpoint circle algorithm.

The goal is to draw a circle, centered at the origin, with radius r. We first note that it suffices to draw an eighth of the circle, since by symmetry we know that if we plot pixel (x, y) we will also plot pixels $(-x, y)$, $(-x, -y)$, $(x, -y)$, (y, x), $(-y, x)$, $(-y, -x)$, and $(y, -x)$. We will pick, arbitrarily, to generate points on the top half of the left side of the circle, where $0 \leq y \leq -x$; the rest of the circle we will create using symmetry.

The implicit equation of a circle is $P(x, y) = x^2 + y^2 - r^2 = 0$. We will start at the point $(-r, 0)$, since we know it is one we

want to plot, and find the following initial finite differences:

$$\begin{aligned} P(-r,0) &= 0 \\ P_x(-r,0) &= [2x+1]_{x=-r} = -2r+1 \\ P_y(-r,0) &= [2y+1]_{y=0} = 1 \\ P_{xx} = P_{yy} &= 2; \end{aligned}$$

all the other differences are zero. This is enough to write a basic form of the algorithm, Algorithm 1.3: Note that we

Algorithm 1.3

Input: Integer radius r
Purpose: Plot the interior border of the circle

```
1: (x, y) ⇐ (−r, 0)
2: P ⇐ 0
3: (P_x, P_y) ⇐ (−2r + 1, 1)
4: (P_xx, P_xy, P_yy) ⇐ (2, 0, 2)
5: while y < −x do
6:    Plot the eight points (±x, ±y),(±y, ±x)
7:    Increment y
8:    Add P_y to P
9:    Add P_yy to P_y
10:   if P > 0 then
11:      Increment x
12:      Add P_x to P
13:      Add P_xx to P_x
```

do not need a **while** loop for incrementing x because in the region we selected, the slope never drops below one, so for each step x either stays the same or increments exactly once.

This code is both efficient and accurate as long as you only wish to color pixels inside the circle. Often, however, people want to color the pixels that are as close to the boundary as possible; this is done using the "midpoint" version of the algorithm. The idea is simple; instead of finding the initial value of P at the first pixel, we find the value at the midpoint of the first pixel and the next pixel outward from it; that is, $P(-r-\frac{1}{2},0) = -r-\frac{1}{4}$ and $P_x(-r-\frac{1}{2},0) = -2r$. We can get rid of the $\frac{1}{4}$ by observing that if $P = 0$ then $4P$ must also equal zero, giving us initial values of

$$\begin{aligned} P &= -r-1 \\ P_x &= -8r \\ P_y &= 4 \\ P_{xx} = P_{yy} &= 8. \end{aligned}$$

If we plug these directly into the algorithm presented earlier, we will be evaluating the polynomial at the midpoint between two pixels (hence the name "midpoint algorithm") and plot those pixels closest to the actual circle itself.

1.7 Fragment Shading: All the Per-Pixel Details

So far we have discussed how to use the mathematical description of an object to generate a set of pixels, each with a color, position, normal, and whatever else you want to interpolate. In real-time graphics each of these pixels is called a fragment and the code that translates them into color on the screen is called the fragment shader. Fragment shaders are nice because their runtime performance is dictated by the number of pixels on the screen rather than the number of polygons displayed.

The simplest fragment shader sets the color of the pixel on

the screen to be the color interpolated to the fragment.

Usually, people use depth buffering to draw only the closest fragment at each pixel. Do do this, simply create store a depth value for each pixel. Then, for each fragment, if its depth is closer than the stored depth, draw it and update the stored depth; otherwise, discard it. This is called depth-buffering or z-buffering.

You can also light each pixel individually. This process is called Phong shading (not to be confused with Phong lighting) or per-pixel lighting. Phong shading takes as input the interpolated object color, pre-device-coordinate-transform vertex position, and vertex normal of a fragment. These are combined using your favorite lighting model (see Section 2.5) to find the color to plot. Note that applying a lighting model to the vertices and interpolating the lit color is called Gouraud shading; most graphics cards default to Gouraud shading.

Texture mapping is done by interpolating texture coordinates to each fragment and looking up the appropriate color from a texture image. Typically, texture coordinates are specified in the range $[0, 1)$ so you'll want to do a device-coordinate-like transformation before doing the texel lookup. There is no reason to restrict textures to storing color information; bump mapping, stores an offset for the normal in the texture (which only works if you do Phong shading, since Gouraud ignores per-fragment normals). Procedural textures generate texture valued through a function rather than via an image lookup.

Mip-mapping is a way of improving textures by having a number of versions of the texture stored at different resolutions. The resolution selected is based on the magnitude of the texture coordinate elements of $d\vec{p}$: larger values means neighboring texels are farther apart in texture coordinate space and calls for a lower-resolution texture image. Often several neighboring texels in several neighboring levels of the mip map are combined via some sort of weighted average to create a smoother looking scene.

Fog is just a function of the depth of a pixel, like the depth buffer. More distant fragments have more of their own color replaced by the fog color.

It is possible to do transparency in a rasterizing system; just keep part of the existing color on the screen, and blend it with the new fragment's color. This requires that fragments are generated in the right order. Typically, this is done by drawing all opaque objects first, then sorting the transparent polygons based on their depth and drawing the most distant ones first. Alternately, you can store a linked list of all the fragments on a particular pixel, then sort them and draw them as a post-processing step. Neither of these techniques is particularly fast.

1.8 Rasterization-time clipping

You never want to draw pixels off of the screen, both for memory security and speed reasons. There are lots of ways to do clipping, but rasterization-time clipping is the one that most graphics cards do.

The naïve version of this is to generate all fragments for all polygons but only plot those that lie within the screen bounds. This is a bad idea; perspective projection often results in polygons hundreds of times larger than the screen, and generating all of those fragments could take a very very long time; in rare cases floating point roundoff could even cause $\vec{p} + \vec{dp}$ to be exactly equal to $\vec{p}$, resulting in an infinite

loop.

The correct solution here is to modify $d\vec{p}_0$ to step not just to the first row or column of pixels, but all the way to the first row or column on the screen. This means we redefine

$$d\vec{p}_0 = \begin{cases} (\lceil p_{1\,i} \rceil - p_{1\,i})\, d\vec{p} & \text{if } p_{1\,i} > 0 \\ (0 - p_{1\,i})\, d\vec{p} & \text{otherwise} \end{cases}$$

Similarly, as soon we DDA to the edge of the screen, we can quit. Thus, the adjusted algorithm to only creates on-screen fragments with very little extra work.

Chapter 2

Raytracing: Screen → Idea

2.1 Introduction to Raytracing

Raytracing tries to solve the same problem as rasterization—that is, create a set of pixel colors to represent a mathematical description of stuff—but goes about it in reverse. Where rasterization asks the questions "what pixels are contained within this object" raytracing asks instead "what objects are visible within this pixel?" It forms the second major technique of drawing on raster displays.

Raytracing is currently slower than rasterization, though it admits almost unlimited amounts of parallelism causing some people think it will become faster in years to come. Currently its primary advantage is that the process does not depend on pixel locality and so can easily model reflection, transparency, and similar optical properties.

The speed of a raytracing system depends largely on the quality of the spatial hierarchy used. Conceptually, the idea here is to collect a group of nearby objects and find a bounding box for them; if a box is not visible from a particular pixel, none of the objects within it are either. I will not discuss these hierarchies further, beyond stating that k-d trees are one of the more popular spatial hierarchies and many discussions of k-d trees (and other hierarchies) are available online.

2.2 Primary and Secondary Rays

A ray is a semi-infinite line; it is typically stored as a point called the ray origin, $\mathbf{r_o}$; and a direction vector $\overrightarrow{r}_d$. Then the ray itself is the set of points $\{\mathbf{r_o} + t\overrightarrow{r}_d \mid t \in [0, \infty)\}$, and the goal of raytracing is to find the point in the ray contained within another object which is closest to the ray origin (that is, with minimal t).

Ray tracing creates one or more rays per pixel. Those rays are intersected with objects in the scene and then, generally, several secondary rays are generated from those intersection points, and intersected with the scene again, and then more are generated, etc., until you get tired of shooting rays, at which point you do some sort of direct illumination (see 2.5) and call it good.

There are a number of ways to generate rays from pixels.

For all of them we need the camera position and orientation, given by the eye position $\mathbf{e}$ and the forward, up, and right directions $\overrightarrow{f}$, $\overrightarrow{u}$, and $\overrightarrow{r}$; the ray origin is generally just $\mathbf{e}$. The most common choice for ray direction is designed to replicate the standard rasterization perspective look: given normalized device coordinates for a pixel (x, y), its ray direction is $\overrightarrow{f}\cos(\theta) + (x\overrightarrow{r} + y\overrightarrow{u})\sin(\theta)$, where θ is the field of view. Fish-eye $\left(x\overrightarrow{r} + y\overrightarrow{u} + \sqrt{1-x^2-y^2}\overrightarrow{f}\right)$ and cylindrical $\left(y\overrightarrow{u} + \cos(x)\overrightarrow{f} + \sin(x)\overrightarrow{r}\right)$ projections are also used in some settings.

Sections 2.6 and 2.6.4 discusses some of the techniques used to generate secondary rays.

2.3 Rays and Ray-* Intersections

In general, the intersection of a ray $\mathbf{r_o} + t\overrightarrow{r}_d$ with some object $g(\mathbf{p}) = 0$ is found as the minimal t for which $g(\mathbf{r_o} + t\overrightarrow{r}_d) = 0$. Constrained minimization of this type is a widely-studied problem in numerical analysis, optimization, and many branches of engineering and science. However, most raytracers use only three particular solutions: ray-plane, ray-AABB, and ray-sphere intersections.

2.3.1 Ray-Plane Intersection

Planes can be described in a number of ways; principle among them are implicit $Ax + By + Cz + D = 0$, point-and-normal $(\overrightarrow{n}, \mathbf{p})$, and three-point $(\mathbf{p_0}, \mathbf{p_1}, \mathbf{p_2})$. We will use the point-normal version, noting that

$$\overrightarrow{n} \equiv (A, B, C) \equiv (\overrightarrow{\mathbf{p_1} - \mathbf{p_0}}) \times (\overrightarrow{\mathbf{p_2} - \mathbf{p_0}})$$

and that the point $\left(-\frac{D}{A}, 0, 0\right)$ is on the plane.

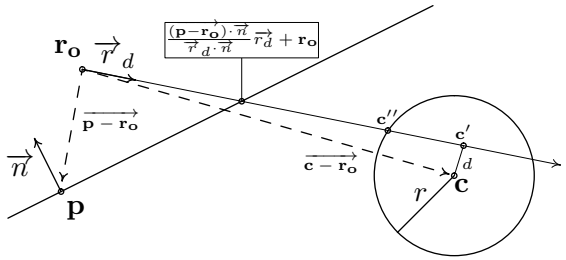

Figure 2.1: Some of the geometry used in the ray-plane and ray-sphere intersection routines.

The distance between the ray origin $\mathbf{r_o}$ and a plane is $(\overrightarrow{\mathbf{r_o} - \mathbf{p}}) \cdot \overrightarrow{n}\frac{1}{\|\overrightarrow{n}\|}$. The distance the ray travels toward the plane per unit t is $\overrightarrow{r}_d \cdot \overrightarrow{n}\frac{-1}{\|\overrightarrow{n}\|}$. Setting these equal to one another we get $t = \frac{(\overrightarrow{\mathbf{p}-\mathbf{r_o}})\cdot\overrightarrow{n}}{\overrightarrow{r}_d\cdot\overrightarrow{n}}$. That's all there is to ray-plane intersection.

2.3.2 Ray-Sphere Intersection

Given a sphere with center $\mathbf{c}$ and radius r, we first evaluate if the ray originates inside the sphere ($\|\overrightarrow{\mathbf{c} - \mathbf{r_o}}\|^2 < r^2$) or not. We then find the t value of the point where the ray comes closest to the center of the sphere, $t_c = \frac{(\overrightarrow{\mathbf{c}-\mathbf{r_o}})\cdot\overrightarrow{r}_d}{\|\overrightarrow{r}_d\|}$. If the ray origin is outside and its value is negative, there is no intersection. Otherwise we proceed to find the squared distance of closest approach $d^2 = \|\mathbf{r_o} + t_c\overrightarrow{r}_d - \mathbf{c}\|^2$. If $d^2 > r^2$, which can only happen if the ray originates outside the sphere, then there is no intersection; otherwise we find how far from the point of closest approach the point of intersection is as $t_{\text{offset}} = \frac{\sqrt{r^2-d^2}}{\|\overrightarrow{r}_d\|}$. If the origin is inside, the point of intersection is $\mathbf{r_o} + (t_c + t_{\text{offset}})\overrightarrow{r}_d$; otherwise, it is $\mathbf{r_o} + (t_c -$

$t_{\text{offset}})\overrightarrow{r}_d$. This is formalized in Algorithm 2.1

Algorithm 2.1 Ray-Sphere Intersection

Input: Ray $(\mathbf{r_o}, \overrightarrow{r_d})$ and sphere $(\mathbf{c}, r)$
Purpose: Find t of intersection, or fail.

$inside \Leftarrow \left(\|\overrightarrow{\mathbf{c}-\mathbf{r_o}}\|^2 < r^2\right)$
$t_c \Leftarrow \frac{(\overrightarrow{\mathbf{c}-\mathbf{r_o}})\cdot\overrightarrow{r}_d}{\|\overrightarrow{r}_d\|}$
if not $inside$ and $t_c < 0$ **then**
  return no intersection
$d^2 \Leftarrow \|\mathbf{r_o} + t_c\overrightarrow{r}_d - \mathbf{c}\|^2$
if not $inside$ and $d^2 > r^2$ **then**
  return no intersection
$t_{\text{offset}} \Leftarrow \frac{\sqrt{r^2-d^2}}{\|\overrightarrow{r}_d\|}$.
if $inside$ **then**
  return $t = t_c + t_{\text{offset}}$
else
  return $t = t_c - t_{\text{offset}}$

2.3.3 Ray-AABB Intersection

It is rare to want to create images of **a**xis-**a**ligned **b**ounding **b**oxes (AABBs), but it is easy to find ray-AABB intersections and easy to find AABBs for most objects, so most raytracers try AABB intersections before trying the intersections of the objects within the AABB.

AABBs consist of six axis-aligned planes. For the axis-aligned case, the ray-plane intersection becomes quite simple because the normal has only one non-zero element; thus, for the plane, e.g., $x = a$, the t value of intersection is $t_{x=a} = \frac{a-\mathbf{r_o}_x}{\overrightarrow{r}_{dx}}$. The ray then intersects the AABB if and only if there is some positive t between all six planes; for minimum point (a, b, c) and maximum point (A, B, C), we have

$$[0, \infty) \cap [t_{x=a}, t_{x=A}] \cap [t_{y=b}, t_{y=B}] \cap [t_{z=c}, t_{z=C}] \neq \emptyset.$$

It is usually not important to know the t-value for an AABB intersection, but if needed it is simply the smallest t in the interval defined above.

2.4 Inverse Mapping and Barycentric Coordinates

Once you have found an intersection with some object it is generally desirable to know where you hit it so that you can apply texture mapping, normal interpolation, or the like. This process is called "inverse mapping."

2.4.1 Inverse Sphere Mapping

For a sphere, if the point of intersection is $\mathbf{p}$ then the normal simply points from the center to the point of intersection, $\overrightarrow{n} = \frac{1}{r}\overrightarrow{\mathbf{p}-\mathbf{c}}$. From that it is easy to derive that the longitude is $\texttt{atan2}(n_x, n_z)$ and the latitude is $\texttt{atan2}(n_y, \sqrt{n_x^2 + n_z^2})$.

2.4.2 Inverse Triangle Mapping

For a triangle, the typical inverse mapping gives you the Barycentric coordinates of the point of intersection. Barycentric coordinates are three numbers, one per vertex of the triangle, which state how close to each of the three vertices the point in question is. In particular, given an intersection point of $\mathbf{p}$ and vertices $\mathbf{p_0}$, $\mathbf{p_1}$, and $\mathbf{p_2}$, the barycentric coordinates (b_0, b_1, b_2) satisfy the two properties that (a) they sum to 1, and (b) $\mathbf{p} = b_0\mathbf{p_0} + b_1\mathbf{p_1} + b_2\mathbf{p_2}$. Every point in the same

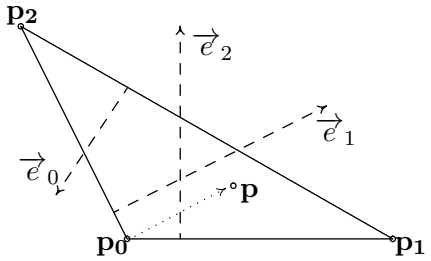

Figure 2.2: Finding Barycentric coordinates. If $\vec{e_2} \cdot (\mathbf{p_2} - \mathbf{p_0}) = 1$, then $b_2 = \vec{e}_2 \cdot (\mathbf{p} - \mathbf{p_0})$, and similarly for b_1. Since $b_0 + b_1 + b_2 = 1$, b_0 is simply $1 - b_1 - b_2$. Thus, assuming that correctly-scaled $\vec{e}_1$ and $\vec{e}_2$ are precomputed, we can compute the barycentric coordinates using just six multiplies and nine adds.

plane as the triangle has a unique set of Barycentric coordinates, and all three coordinates are positive if and only if the point is within the triangle's bounds.

There are many techniques for inverse mapping a triangle. The one I present here is not the most common, but is easy and efficient as long as you compute and store the information only once. First, observe that since b_i is the "nearness" to point $\mathbf{p_i}$, it is also the distance from the edge joining the other two points. This distance can be found directly by using a dot product with a vector perpendicular to this edge. This process is illustrated in Figure 2.2; in that image, $b_1 = \vec{e}_1 \cdot (\mathbf{p} - \mathbf{p_0})$ because $\vec{e}_1$ points directly away from the edge between $\mathbf{p_{i \neq 1}}$. Similarly, $b_2 = \vec{e}_2 \cdot (\mathbf{p} - \mathbf{p_0})$ and $b_0 = 1 - b_1 - b_2$. It thus suffices to find $\vec{e}_1$ and $\vec{e}_2$ in order to find the barycentric coordinates. This may be done as

$$\begin{aligned}
\vec{a}_1 &= \overrightarrow{\mathbf{p_2} - \mathbf{p_0}} \times \vec{n} \\
\vec{a}_2 &= \overrightarrow{\mathbf{p_1} - \mathbf{p_0}} \times \vec{n} \\
\vec{e}_1 &= \frac{1}{\vec{a}_1 \cdot \overrightarrow{\mathbf{p_1} - \mathbf{p_0}}} \vec{a}_1 \\
\vec{e}_2 &= \frac{1}{\vec{a}_2 \cdot \overrightarrow{\mathbf{p_2} - \mathbf{p_0}}} \vec{a}_2.
\end{aligned}$$

These two $\vec{e}_i$ vectors can be precomputed and stored along with the normal $\vec{n}$ in each triangle data structure to allow rapid ray-triangle intersections. Note that this also suffices for the inside-outside test needed to turn a ray-plane intersection into a ray-triangle intersection; a point is inside a triangle if and only if all three barycentric coordinates are between zero and one.

Because $\mathbf{p} = b_0\mathbf{p_0} + b_1\mathbf{p_1} + b_2\mathbf{p_2}$, we can use the barycentric coordinates to find all the information stored in each vertex interpolated to any point on the interior of the triangle: $\vec{p} = b_0\vec{p_0} + b_1\vec{p_1} + b_2\vec{p_2}$. You can then use that information just as you would in rasterization (see Section 1.7).

2.5 Direct Illumination

Once we have a point in space mapped to a pixel, one of the most common tasks is to approximate the color and lighting at that point Typically computer-faked light is divided into three parts: ambient, diffuse, and specular. Each of these will generate a color vector for each light source; these color vectors are then multiplied element-wise by the color of the

object. which may be different for each type of light:

$$\left(\vec{m}_a \otimes \sum_{l \in L} \vec{l_c} \times l_a\right) + \left(\vec{m}_d \otimes \sum_{l \in L} \vec{l_c} \times l_d\right) + \left(\vec{m}_s \otimes \sum_{l \in L} \vec{l_c} \times l_s\right).$$

A note about notation in this section: $\hat{n}$ is the unit-length normal vector, $\hat{e}$ a unit-length vector that points towards the eye of the viewer, $\hat{\ell}$ a unit-length vector that points towards the light source, and l_a, l_d, and l_s are the ambient, diffuse, and specular light intensities, respectively. I also use $\llbracket a \rrbracket$ to mean $\max(0, a)$ in this section for compactness of notation.

In all cases the discussion below assumes no attenuation of light with distance or angle; to add attenuation, simply multiply the results below by an attenuation factor (ex: since point light falls with the square of the distance, we would simply multiply all the results below by $\frac{1}{d^2}$ to model its attenuation).

2.5.1 Ambient Light

Ambient light is assumed to come from everywhere and reach everywhere equally. Thus, the ambient light color is independent of the object; l_a is simply a constant. There is no ambient light in the real world; instead, it is a substitute for tracing light that reaches an object by first bouncing off other objects. In general, keep the ambient light small, no more than 20% of the total light possible.

2.5.2 Diffuse Light

Diffuse light is the main component we think of when considering a matte object. There are several different models for generating it.

In the Lambert model, $l_d = \llbracket \hat{n} \cdot \hat{\ell} \rrbracket$. This is what would happen if every photon bounced in a completely random direction on a smooth surface.

The Minnaert model extends the Lambert model by biasing the light to bounce away from the surface; the bias is given by a constant $k \in [0, 1]$, and the formula is $l_d = \llbracket \hat{n} \cdot \hat{l} \rrbracket^k \llbracket \hat{n} \cdot \hat{e} \rrbracket^{1-k}$. This was created to model the appearance of the moon; the lower k the brighter the edges of an object will appear.

The Oren-Nayer models assumes an object is made out of a sub-pixel-resolution bumps and crevices. It is quite complicated, but very close to what real-world objects look like. Let $\sigma \in [0, 1]$ be the roughness of the surface. Then the Oren-Nayer model is

$$\begin{aligned}
\hat{f}_e &= \frac{\hat{e} - (\hat{e} \cdot \hat{n})\hat{n}}{\|\hat{e} - (\hat{e} \cdot \hat{n})\hat{n}\|} \qquad \hat{f}_\ell = \frac{\hat{\ell} - (\hat{\ell} \cdot \hat{n})\hat{n}}{\left\|\hat{\ell} - (\hat{\ell} \cdot \hat{n})\hat{n}\right\|} \\
\theta_\ell &= \cos^{-1}(\hat{\ell} \cdot \hat{n}) \qquad \theta_e = \cos^{-1}(\hat{e} \cdot \hat{n}) \\
\alpha &= 0.45 \frac{\sigma^2}{\sigma^2 + 0.09} \sin(\max(\theta_\ell, \theta_e)) \tan(\min(\theta_\ell, \theta_e)) \\
l_d &= \llbracket \hat{n} \cdot \hat{\ell} \rrbracket \left(1 - 0.5 \frac{\sigma^2}{\sigma^2 + 0.33} + \alpha \llbracket \hat{f}_e \cdot \hat{f}_l \rrbracket\right)
\end{aligned}$$

Any of these can be turned into a toon shader by simply picking a cutoff value of l_d and clamping numbers above it to 1, numbers below it to 0.

2.5.3 Specular Light

The specular highlight of an object (shiny spot) has even more versions than does diffuse lighting. Most of versions rely on a hardness number $n \geq 1$ which makes the shine spot small, on the reflection of the light off the surface $\hat{r} =$

$2(\hat{n}\cdot\hat{\ell})\hat{n}-\hat{\ell}$, and/or on the vector halfway between the light and the eye $\hat{h}=\frac{\hat{\ell}+\hat{e}}{\left\|\hat{\ell}+\hat{e}\right\|}$.

The Phong model is $l_s=\llbracket\hat{r}\cdot\hat{e}\rrbracket^n$ and the Blinn-Phong is $l_s=\llbracket\hat{h}\cdot\hat{n}\rrbracket^n$. They are similar in look and are easy to compute; generally Blinn-Phong is used in conjunction with a single approximate $\hat{e}$ and $\hat{\ell}$ while Phong is used if $\hat{\ell}$ and/or $\hat{e}$ vary across the scene.

The Gaussian model is more accurate, but likewise more expensive to compute: $l_s=e^{-m\cos^{-2}\llbracket\hat{n}\cdot\hat{h}\rrbracket}$ (note e is Euler's number, $\hat{e}$ is the vector to the eye). Better and more expensive is the Beckmann distribution

$$l_s=\frac{m}{\llbracket\hat{n}\cdot\hat{h}\rrbracket^4}e^{-m\tan^2(\cos^{-1}\llbracket\hat{n}\cdot\hat{h}\rrbracket)}.$$

Fresnel's law specifies how much light penetrates a transparent object; this can be combined with the Beckmann distribution to get the Cook-Torrance model. Let λ be the Fresnel factor and β the computed Beckmann distribution; then Cook-Torrance gives

$$l_s=\beta\frac{(1+\llbracket\hat{e}\cdot\hat{n}\rrbracket)^\lambda}{\llbracket\hat{e}\cdot\hat{n}\rrbracket}\min\left(1,\frac{2\left(\hat{h}\cdot\hat{n}\right)2\left(\hat{e}\cdot\hat{n}\right)}{\hat{e}\cdot\hat{h}},\frac{2\left(\hat{h}\cdot\hat{n}\right)2\left(\hat{\ell}\cdot\hat{n}\right)}{\hat{e}\cdot\hat{h}}\right).$$

There are also a variety of anisotropic models (notably Heidrich-Seidel and Ward) which depend additionally upon the principle tangent vector of the surface and can give the appearance of brushed metal, hair, and the like, but which are beyond the scope of this booklet.

2.6 Secondary Rays

Shadows, reflection, and transparency are easily achieved using secondary rays: Once you find an intersection point $\mathbf{p}$ you then generate a new ray with $\mathbf{p}$ as its origin and intersect that ray with the scene. Care should be taken that roundoff errors in storing $\mathbf{p}$ do not cause the the secondary ray to intersect the object from which it originates.

2.6.1 Shadows

A point is in shadow relative to a particular light source if the ray $(\mathbf{p},\hat{\ell})$ intersects anything closer than the light source itself.

2.6.2 Reflection

A mirrored object's color is given by the ray tracing result of the ray $(\mathbf{p},2(\hat{n}\cdot\hat{e})\hat{n}-\hat{e}))$. A partially mirrored object mixes that color result with a standard lighting computation at $\mathbf{p}$. One reflection ray might generate another; a cutoff number of recursions is necessary to prevent infinite loops.

2.6.3 Transparency

Transparency is somewhat more complicated, relying on Snell's Law. For it to make sense, every surface needs to be a boundary between two materials, which is not trivially true in the case of triangles nor intersecting spheres. However, assuming that we know that the ray is traveling from a material with index of refraction n_1 for index of refraction n_2, we can derive a rule for finding the transmitted ray.

The cosine of the entering ray is $(\hat{e}\cdot\hat{n})$, meaning that it's sine is $\sqrt{1-(\hat{e}\cdot\hat{n})^2}$, or $\|-\hat{e}-(\hat{e}\cdot\hat{n})\hat{n}\|_2$. The sine of the out-

going vector thus needs to be $\frac{n_1}{n_2}\sqrt{1-(\hat{e}\cdot\hat{n})}$; if that is greater than 1, we have total internal refraction and use the reflection equation instead; otherwise, the cosine of the outgoing ray is $\sqrt{1-(\frac{n_1}{n_2})^2(1-(\hat{e}\cdot\hat{n}))}$. Putting this all together, we have

1: $a \Leftarrow \hat{e}\cdot\hat{n}$
2: $b \Leftarrow \frac{n_1}{n_2}\sqrt{1-(\hat{e}\cdot\hat{n})}$
3: **if** $b \geq 1$ **then**
4: **return** $2(\hat{n}\cdot\hat{e})\hat{n}-\hat{e})$
5: $c \Leftarrow \sqrt{1-\frac{n_1^2}{n_2^2}(1-(\hat{e}\cdot\hat{n}))}$
6: **return** $\frac{n_1}{n_2}(-\hat{e}-(\hat{e}\cdot\hat{n})\hat{n})-c\hat{n}$

2.6.4 Global Illumination

The standard lighting models pretend like the world is divided into a small number of light emitters and a vast supply of things that emit no light. This is obviously not true; if nothing but light sources gave off photons, we could only see the light sources themselves. With global illumination, you try to discover the impact of light bouncing off of the wall, floor, and other objects by creating a number of secondary rays sampling the diffuse reflection of each object. Ideally, the distribution of rays should parallel the chosen model of diffuse lighting (see Section 2.5.2) and should be so dense as to be intractable for any reasonable scene. Much of the research in photo-realistic rendering is devoted to finding shortcuts and techniques that make this process require fewer rays for the same visual image quality.

2.7 Photon Mapping and Caustics

Photon mapping is raytracing run backwards: instead of shooting rays from the eye, you shoot them from the lights. The quantity of light reaching each point in the scene is recorded and used when the scene is rendered, either by raytracing or rasterizing. Since many photons leaving a light source never reach the eye, this is an inefficient way of creating a single picture; however, it allows lens and mirror-bounced light (called caustics) to be rendered, and it can be more efficient than viewer-centric global illumination if many images are to be made of the same static scene.

2.8 Sub-, Super-, and Importance-Sampling

Sub-sampling is shooting fewer rays than you have pixels, interpolating the colors to neighboring pixels. Super-sampling is shooting several rays per pixel, averaging colors to create an anti-aliased image. Importance sampling shoots fewer rays per pixel into "boring" areas and more into "important" areas. Image-space importance sampling shoots more rays into areas of the scene where neighboring pixels differ in color; scene-space importance sampling shoots more rays toward particular items.