Computer Graphics Notes

Ethan Cheng

Abstract

Computer Graphics Programming, taught by Jon-Alf Dyrland-Weaver

Contents

1 Feb. 3, 2016 - Image File Formats

We will be choosing an easy-to-work-with file format for images we're generating... because this a graphics course. We will be making graphics. That are images. Yes.

1.1 Compressed vs Uncompressed

- Compressed
  - Performs an algorithm on an uncompressed image to reduce file size
  - Smaller
  - Often less information
- Uncompressed
  - Full map of pixel values
  - Raw data

1.1.1 Compressed Image Formats

- png
- gif
- jpeg/jpg

1.1.2 Uncompressed Image Formats

- bmp (Bitmap)
- tiff
- svg
- raw

1.2 Lossy vs Lossless

- Lossy
  - Compression algorithms lose some of the original information
- Lossless
  - Contains all the original information

All **uncompressed** image formats are lossless. Duh. **JPEG** files are lossy compressed. However, **PNG** files are lossless compressed.

1.3 Run Length Encoding

We can very easily make a lossless compression algorithm:

```
12B: GGGGGYYYRRRR
6B:  5G3Y4R
```

However, this is a bad compression algorithm for real-life images (i.e. from a camera), since adjacent pixels will almost NEVER be the same. This is, however, what the PNG format uses, and is used for flat, "minimalist" computer generated images.

1.4 Raster vs Vector

- Raster
  - Image is represented as a grid of pixels
- Vector
  - Image is represented as a list of drawing instructions
  - Scale well
  - SVG stands for "Scalable Vector Graphics"

1.5 So... what are we using? netpbm!

`netpbm` is a very simple, raster, lossless, uncompressed format. It is extremely simple: space separated values of triplet integers from [0,255].

2 Feb. 4, 2016 - Yay! NetPBM!

2.1 NetPBM File Format

NetPBM files are `.ppm`, plaintext files, interpretted as images.

2.1.1 File Header

- File Header : `P3`
- `XRES` : x-resolution as an integer value of pixels
- `YRES` : y-resolution as an integer value of pixels
- `MAX_COLOR_VALUE` : Maximum color value of a pixel

2.1.2 File Data

- R_n : ASCII representation of red value of pixel n (i.e. 255)
- G_n : ASCII representation of green value of pixel n (i.e. 255)
- B_n : ASCII representation of blue value of pixel n (i.e. 255)

These values can be separated by **any** type of whitespace:

- space
- tab
- newline

Here is a sample file: `yellow.ppm` (a 5x5 pixel square of yellow):

```
P3
5 5
255 255 0 255 255 0 255 255 0 255 255 0 255 255 0
255 255 0 255 255 0 255 255 0 255 255 0 255 255 0
255 255 0 255 255 0 255 255 0 255 255 0 255 255 0
255 255 0 255 255 0 255 255 0 255 255 0 255 255 0
255 255 0 255 255 0 255 255 0 255 255 0 255 255 0
```

3 Feb. 5 - 10, 2016 - Bresenham's Line Algorithm

3.1 Computer Graphics Basics: The Octal Cartesian Plane

In math, we usually split the Cartesian Plane into **quadrants**, along intervals of 90°, going radially.

In computer graphics, it is much easier to subdivide the quadrants into **octants**, along intervals of 45°, going radially.

3.2 Computer Graphics Basics: Lines with Pixels

In computer graphics, drawing a straight line is a bit strange. Because pixels are integer values, we don't care about floating point numbers. This is why graphics cards exist: they are processors that only operate with integers, and are well optimized and fast.

What about drawing the line? We can come up with a few ideas:

1. Pick a random point, and those that are on the line, we keep it
2. Calculate the "delta" value of x and y based on the slope of the line, given by the two endpoints
3. Using some forms of the line equations from math
4. Some combination with some clever optimizations of the 3

3.3 Plotting a line without floats

We have our standard line equation in point–slope form. We can manipulate this to **not** use division, which prevents floating point numbers

$$\begin{aligned} y &= mx + b \\ y &= \frac{\Delta y}{\Delta x}x + b \\ y\Delta x &= x\Delta y + b\Delta x \\ 0 &= x\Delta y - y\Delta x + b\Delta x \\ A &= \Delta y \\ B &= -\Delta x \\ C &= b\Delta x \\ 0 &= Ax + By + C = f(x, y) \end{aligned}$$

We end up with our standard line equation from math, which deals strictly with INTEGER VALUES, since x and y are pixel values, which are integers.

We can simply evaluate $f(x, y)$, and if the result equals 0, then (x, y) is on the line.

Let's split up the 8 octants when we consider with pixels when we use our function $F(x, y)$

Note that we can split this into cases!

$$f(x, y) = Ax + By + C = \begin{cases} = 0 \rightarrow (x, y) \text{ is on the line} \\ < 0 \rightarrow (x, y) \text{ is above the line} \\ > 0 \rightarrow (x, y) \text{ is below the line} \end{cases}$$

We can get the relations with the line using $B = -\Delta x$.

From this piece-wise evaluation: we only have 2 options when we plot a point from one endpoint to another:

1. $(x + 1, y + 1)$
2. $(x + 1, y)$

Rather than looking at both of those, we can instead look at the midpoint $(x+1, y+\frac{1}{2})$.

This gives us the following relation:

$$f(x+1, y+\tfrac{1}{2}) = \begin{cases} < 0 \rightarrow \text{ midpoint is above the line. Draw the lower pixel: } (x+1, y) \\ 0 \rightarrow \text{ midpoint is on the line. Draw either pixel} \\ > 0 \rightarrow \text{ midpoint is below the line. Draw the higher pixel: } (x+1, y+1) \end{cases}$$

3.4 Algorithm Pseudocode

We now have our algorithm, so we can write a crude version in pseudocode, drawing a line from $(x_0, y_0 \rightarrow (x_1, y_1)$.

```
1 : // Check if we are in quadrant I
2 : @assert(0 < m and m < 1)
3 : x = x0
4 : y = y0
5 : while (x <= x1)
6 :     plot(x, y)
7 :     // Calculate the delta using given function f()
8 :     d = f(x + 1, y + (1 / 2))
9 :     if (d > 0)
10:         y = y + 1
11:     x = x + 1
```

However, on line 8, we are recalculating d every single iteration of the loop. We can bring it out of the loop. We can set the initial value:

$$\begin{aligned} d &= f(x_0 + 1, y_0 + \frac{1}{2}) \\ &= A(x_0 + 1) + B(y_0 + \frac{1}{2}) + C \\ &= Ax_0 + A + By_0 + \frac{1}{2}B + C \\ &= (Ax_0 + By_0 + C) + A + \frac{1}{2}B \\ &= f(x_0, y_0) + A + \frac{1}{2}B \\ \Delta d &= A + \frac{1}{2}B \end{aligned}$$

From this, we can improve our pseudocode:

```
1 : // Check if we are in quadrant I
2 : @assert(0 < m and m < 1)
3 : x = x0
4 : y = y0
5 : d = A + B / 2
6 : while (x <= x1)
7 :     plot(x, y)
8 :     if (d > 0)
9 :         y = y + 1
10:     x = x + 1
11:     d = f(x + 1, y + (1 / 2))
```

This isn't very helpful, since we are still calling `f()` every single iteration. Let us make a table:

$d < 0$	$d > 0$

$x \to x + 1, y \to y$
`f(x+1, y)`
`d = d + A`
	$x \to x + 1, y \to y + 1$
	`f(x+1, y+1)`
	`d = d + A + B`

This let's us reduce calculating $f()$ to a simple addition.

```
1 : // Check if we are in quadrant I
2 : @assert(0 < m and m < 1)
3 : x = x0
4 : y = y0
5 : d = A + B / 2
6 : while (x <= x1)
7 :     plot(x, y)
8 :     if (d > 0)
9 :         y = y + 1
10:         d = d + B
10:     x = x + 1
11:     d = d + A
```

There is one more flaw: we are dividing by 2 on line 5. We can get rid of this by doing the following:

$$d = A + \frac{1}{2}B$$
$$2d = 2A + B$$

This gives us $2d$ instead of d. We can change our incrementor statements:

$$\begin{aligned} d = d + A \quad &\rightarrow 2d \quad &= 2d + 2A \\ d = d + A + B \quad &\rightarrow 2d \quad &= 2d + 2A + 2B \end{aligned}$$

We can patch our implementation:

```
1 : // Check if we are in quadrant I
2 : @assert(0 < m and m < 1)
3 : x = x0
4 : y = y0
5:  A = y1 - y0
6:  B = -(x1 - x0)
7 : d = 2A + B
8 : while (x <= x1)
9 :     plot(x, y)
10:     if (d > 0)
11:         y = y + 1
12:         d = d + 2B
13:     x = x + 1
14:     d = d + 2A
```

This gives us a valid algorithm for octant 1. Let us extend this to octant 2:

3.4.1 Extending the algorithm to other quadrants

We now have a new rule for m: $1 < m$. Like in octant 1, where we had:

1. $(x + 1, y + 1)$

2. $(x+1, y)$

In octant 2, we have:

1. $(x, y+1)$
2. $(x+1, y+1)$

This gives us a midpoint of $(x+\frac{1}{2}, y+1)$, giving us an initial point of $f(x_0+\frac{1}{2}, y_0+1))$, and $d = \frac{1}{2}A + B$.

Using this, we can adapt our algorithm to octant 2:

```
1 : // Check if we are in quadrant I
2 : @assert(0 < m and m < 1)
3 : x = x0
4 : y = y0
5:  A = y1 - y0
6:  B = -(x1 - x0)
7 : d = A + 2B
8 : while (y <= y1)
9 :     plot(x, y)
10:     if (d < 0)
11:         x = x + 1
12:         d = d + 2A
13:     y = y + 1
14:     d = d + 2B
```

Octants 5 and 6 are the same as 1 and 2! Likewise, Octants 3 and 4 are the same as 7 and 8, albeit going between the pairs of pairs requires flipping the signs.

In Octant 8, we have

1. $(x+1, y-1)$
2. $(x+1, y)$

This gives us a midpoint of $(x+1, y-\frac{1}{2})$, giving us an initial point of $f(x_0+1, y-\frac{1}{2}))$, and $d = \frac{1}{2}A + B$.

4 Feb. 23 - 24, 2016 - Matrix Math Review

In order to implement a different method of representing a picture, we have to review some matrix math:

4.1 Scalar Multiplcation

$$s \cdot \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} sa & sb \\ sc & sd \end{bmatrix}$$

Simple!

4.2 Matrix Multiplication

Now let's move on to something harder. Matrix Multiplication must satisfy some requirements:

Given $M_0 M_1$

- $M_0 M_1 \neq M_1 M_0$
- The number of columns of M_0 must equal the number of rows in M_1

For instance,

$$\begin{bmatrix} a & b & c \end{bmatrix}$$

has the shape 1×3. If this matrix is M_0, M_1 must have the shape $3 \times n$. For instance:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

If we multiply these, we will get:

$$\begin{bmatrix} a & b & c \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 1a + 2b + 3c \end{bmatrix}$$

From this we see: if M_0 is a matrix of shape $a \times b$ and M_1 is a matrix of shape $b \times c$, $M_0 \cdot M_1$ has the shape $a \times c$.

To find the $(x \times y)$th element is the product of the xth row of M_0 and the yth row of M_1.

For practice:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} = \begin{bmatrix} 1a+2b+3c & 4a+5b+6c \\ 1d+2e+3f & 4d+5e+6f \\ 1g+2h+3i & 4g+5h+6i \end{bmatrix}$$

4.3 Multiplicative Identity Matrix

There is a Multiplicative Identity Matrix that can have a variable shape, but must satisfy the following conditions:

- Square shape (number of rows is equal to number of columns)
- Diagonal values (items (n,n) for $0 \leq n \leq$Number of rows) are all 1
- All other values are 0

4.4 Transforming Matrices

4.4.1 Scaling

We can apply this to a simple scaling algorithm:

$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax \\ by \\ cz \\ 1 \end{bmatrix}$$

This scales the x, y, and z dimensions by factors of a, b, and c respectively.

4.4.2 Translating

To translate (x, y, z) by (a, b, c) WITHOUT adding the matrices (we want to keep everything in terms of matrix multiplication.

$$\begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \\ z+c \\ 1 \end{bmatrix}$$

4.4.3 Rotation

To rotate a point (x, y, z), we must supply a rotation axis and an angle.

Without loss of generality, let's rotate about he z axis:

$$(x, y, z) \to (?, ?, z)$$

To find the two question marks, we must do some trig: If we convert to polar, we have:

$$\begin{aligned} x &= r\cos(\phi) \\ y &= r\sin(\phi) \end{aligned}$$

If we rotate an angle θ, we rotate to:

$$\begin{aligned} x_0 &= r\cos(\phi + \theta) \\ y_0 &= r\sin(\phi + \theta) \end{aligned}$$

Expanding this out with angle addition formulas, we have:

$$\begin{aligned} x_0 &= r\cos(\phi + \theta) \\ &= r\cos(\phi)\cos(\theta) - r\sin(\phi)\sin(\theta) \\ &= x\cos(\theta) - y\sin(\theta) \end{aligned}$$

$$\begin{aligned} y_0 &= r\sin(\phi + \theta) \\ &= r\sin(\phi)\cos(\theta) + r\cos(\phi)\sin(\theta) \\ &= y\cos(\theta) + x\sin(\theta) \end{aligned}$$

5 Feb. 25, 2016 - Rotation Continued

Now that we have our identities for rotation for the z axis, we can convert it to rotation about the x and y axis:

- **x-axis**

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$(x, y, z) \rightarrow (x, y\cos\theta - z\sin\theta, y\sin\theta + z\cos\theta)$

- **y-axis**

$$\begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$(x, y, z) \rightarrow (x\cos\theta - z\sin\theta, y, x\sin\theta + z\cos\theta)$

- **z-axis**

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$(x, y, z) \rightarrow (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta, z)$

5.1 Applying Transformations

Note: The transformations we are doing are called **affined transformations**.

Let E_0 be our edge matrix, which we have added edges into. Let T be our translate matrix, S be our scale matrix, and R be our rotate matrix.

$T \cdot E_0 = E_1$ Translated
$S \cdot E_1 = E_2$ Translated, then Scaled
$R \cdot E_2 = E_3$ Translated, then Scaled, then Rotated

Note that $E_3 = R \cdot S \cdot T \cdot E_0$. Matrix multiplication is not commutative, but it **is** *associative*! We can simply group the transformations before applying them on E_0, which would improve efficiency. Because our translation matrices are 4×4, but E_0 is $4 \times n$ where n is an arbitrary number, if n is large, multiplying by a $4 \times n$ 3 times is quite consuming.

$(R \cdot S \cdot T) \cdot E_0$ will only multiply by a $4 \times n$ once, saving runtime and resources.

6 Mar. 7, 2016 - Parametric Equations

Given x and y equations, we can let x and y be equations of t to create parametric equations.

Given our line equation that takes (x_0, y_0) to (x_1, y_1), we can create parametric equations:

$$f(t) = x_0 + t(\Delta x)$$
$$g(t) = y_0 + t(\Delta y)$$

We can simply define parametrics for a circle and draw the circle with iterations of a small amount to iterate t:

```
double x(double t) {
    return 25 * cos(2 * M_PI * t) + XRES / 2;
}

double y(double t) {
    return 25 * sin(2 * M_PI * t) + YRES / 2;
}

double step, x0, y0, x, y;

double CEILING = 1.0;

x0 = x(0);
y0 = y(0);

for (double t = step; t < CEILING; t += step) {
    x = x(t);
    y = y(t);
```

```
    draw_line(x0, y0, x, y);

    x0 = x;
    y0 = y;
}
```

However, due to floating point imprecision, we should make `CEILING` a bit higher than 1.0, so a value like 1.0001.

7 Mar. 8, 2016 - Splining

Hermite Curves

Given $f(t) = at^3 + bt^2 + ct + d$, we can find its derivative $f^{'}(t) = 3at^2 + 2bt + c$. When $t = 0$, $f(t) = d$ and $f^{'}(t) = c$. When $t = 1$, $f(t) = a + b + c + d$ and $f^{'}(t) = 3a + 2b + c$.

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} d \\ a+b+c+d \\ c \\ 3a+2b+c \end{bmatrix} = \begin{bmatrix} P_0 \\ P_1 \\ R_0 \\ R_1 \end{bmatrix}$$

To obtain $\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$, We need the inverse matrix of $\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}$ which just so happens to be: $\begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$

7.1 Bezier Curves

We need 4 endpoints! There are some cool animations on wikipedia that can explain:

`https://en.wikipedia.org/wiki/Bezier_curve`.

8 Mar. 10, 2016 - Bezier Curves Continued

A linear Bezier Curve from P_0 to P_1 is denoted by the parametric equation

$$P(t) = (1-t)P_0 + tP_1$$

A quadratic Bezier Curve with 3 points P_0 to P_2, with P_1 doing the "tugging" is denoted by the parametric equation:

$$R(t) = (1-t)^2P_0 + 2t(1-t)P_1 + t^2P_2$$

A cubic Bezier Curve with 4 points P_0 to P_3, with P_1 and P_2 tugging, is denoted by:

$$Q(t) = (1-t)^3P_0 + 3t(1-t)^2P_1 + 3t^2(1-t)P_2 + t^3P_3$$

9 Mar. 22, 2016 - 3 Dimensionality

Now that we have lines, hermite curves, bezier curves, and circles, we want to be able to generate pseudo-3D images with 2D representations.

9.1 Spheres

Let's make a sphere. As we know from Blender, there is the icosphere (formed by triangles) and the UV sphere (formed by layers of circles).

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\phi & -sin\phi & 0 \\ 0 & sin\phi & cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} r\cos\theta \\ r\sin\theta \\ 0 \\ 1 \end{bmatrix} = \begin{cases} x = r\cos\theta \\ y = r\sin\theta\cos\phi \\ z = r\sin\theta\sin\phi \end{cases}$$

We have θ as the circle creation, and ϕ as the angle of circle rotation. We need to check ranges between 0 and 2π and between 0 and π for both θ and ϕ.

Pseudocode for sphere points:

```
for p from 0.0 to 1.0:
for t from 0.0 to 1.0:
x = r * cos(pi * t)
z = r * sin(pi * t) cos(2 * pi * p)
z = r * sin(pi * t) sin(2 * pi * p)
```

Now... onto the tastiest of shapes!

9.2 The Torus. AKA The Tasty Donut

If we think about it we can generate a torus by translating about the y axis and rotating in the x axis. This allows to rotate the circle about an external point, generating each "sector" or "slice" of the torus.

We can modify our original matrix operation to translate / rotate our circle to generate a torus of radius R:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\phi & -sin\phi & 0 \\ 0 & sin\phi & cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} r\cos\theta \\ r\sin\theta + R \\ 0 \\ 1 \end{bmatrix} = \begin{cases} x = r\cos\theta \\ y = \cos\phi(r\sin\theta + R) \\ z = \sin\phi(r\sin\theta + R) \end{cases}$$

10 Mar. 29, 2016 - Wireframe and Polygon Meshes

From our code for adding spheres and tori, here:

```
addSphere(Double cx, Double cy, Double cz, Double radius) {
    double x, y, z;

    for (double p = 0; p < CEILING; p += STEP) {
        for (double t = 0; t < CEILING; t += STEP) {
            x = radius * cos(t * PI) + cx;
            y = radius * sin(t * PI) * cos(p * 2 * PI) + cy;
            z = radius * sin(t * PI) * sin(p * 2 * PI) + cz;
            addPoint(new Point((int) x, (int) y, (int) z));
        }
    }
}

addTorus(Double cx, Double cy, Double cz, Double r1, Double r2) {
    double x, y, z;

    for (float p = 0; p < CEILING; p += STEP) {
        for (float t = 0; t < CEILING; t += STEP) {
            x = r1 * cos(t * 2 * PI) + cx;
            y = cos(p * 2 * PI) * (r1 * sin(t * 2 * PI) + r2) + cy;
            z = sin(p * 2 * PI) * (r1 * sin(t * 2 * PI) + r2) + cz;
            addPoint(new Point((int) x, (int) y, (int) z));
        }
    }
}
```

We see that the points that we insert are always a **set amount**. This means for a sphere of radius 5, we may be plotting 4000 points, and eventually, it just looks like a dot, destroying the 3D aspect.

We move onto wireframe and polygon meshes.

10.1 Wireframe Meshes

Wireframe meshes are 3D objects defined by the edges that connect the vertices or points.

It uses the same edge matrix concepts that we have been dealing with throughout previous assignments.

10.2 Polygon Meshes

Polygon meshes are 3D objects defined by the surfaces (typically triangles or quadrilaterals, as we've seen in Blender) that cover the object.

However, we need to change our polygon mesh to use **polygon matrices** rather than edge matrices.

This allows us to draw faces and surfaces, as well as remove hidden faces and surfaces, which saves much computation later on down the line.

10.3 Polygon Matrices

Like we had with an edge matrix, where we stored 2 points (start and end), in a polygon matrix of triangles, we store 3 points: the vertices of a triangle. We are drawing **3** lines per polygon.

Edge Matrix	Polygon Matrix
`plot`	`plot`
`drawLine`	`drawLine`
`drawLines`	`drawPolygons`
`addPoint`	`addPoint`
`addEdge`	`addPolygon`

NOTE that in `addPolygon`, we must add the 3 points in COUNTERCLOCKWISE fashion.

11 April 5, 2016: Backface Culling

Backface culling is a technique to render only forward facing surfaces. The surface normal, $\vec{N}$, is a vector perpendicular to a plane.

We can compare $\vec{N}$ to the view vector (camera view).

If we look towards an object, then $\vec{N}$ must point towards you. Thus, if the angle between $\vec{N}$ and the view vector $\vec{V}$ is θ, then $90 \leq \theta \leq 270$ will create a front facing surface.

Thus, we must use the following algorithm:

1. Calculate $\vec{N}$

   Cross product of 2 vectors that share endpoint and go in different directions:

   $$\begin{aligned}\vec{N} &= \vec{A} \times \vec{B} \\ &=< A_yB_z - A_zB_y, A_zB_x - A_xB_z, A_xB_y - A_yB_x >\end{aligned}$$

2. Find θ between $\vec{N}$ and $\vec{V}$

   We can just use a default view vector: $\vec{V} =< 0, 0, -1 >$. We can use the formula:

   $$\cos\theta = \frac{N_xV_x + N_yV_y + N_zV_z}{|\vec{N}||\vec{V}|}$$

3. If $90 \leq \theta \leq 270$, draw the surface.

12 April 12, 2016: Relative Coordinate System

Currently all of our objects are drawn with respect to the same origin / coordinate system that is **global**. In a relative coordinate system! Each object could have its own origin / coordinate system.

We will be using a stack to store the coordinate systems. Our drawing framework will be as follows:

1. Transformations are applied to the current top of the stack
2. The stack is pushed and popped as needed.
3. Drawing pipeline:
   (a) Generate the points or polygons and add them to a matrix
   (b) Multiply the points by the current stack top
   (c) Draw the points to the screen
   (d) Clear the point matrix

13 April 19 - 21, 2016: Compiler

Using this stack structure requires a bit of a change in our parser. Our current interpreter is too simple. We need... a *compiler*.

The compilation process looks like this:

1. Source code
2. Compiler
   (a) Lexer
   - Performs lexical analysis
   - Knows the valid symbols of your language
   - Generate a list of the tokens in the source code
   - Given the following code:

```
int main() {
    long x = 5 + 6;
    printf("hi");
    return 0;
}
```

This will turn into a list of tokens (keywords):

```
int
IDENT main
(
)
{
    long
    IDENT x
    =
    5
    +
    6
    ;
}
```

(b) Parser

- Performs syntax analysis
- Knows the grammar of your language
- Generates a syntax tree

(c) Semantic analyzer

- Takes a syntax tree as input
- Knows how to map tokens to operations or variables
- Note that functions are variables! They are variables that define a sequence of operations!
- `return` is an operation!
- Knows how to navigate the syntax tree
- Has a symbol table to map things
- List of operations in order to be performed
- Outputs the operation list (pseudo machine code)

(d) Optimizer

- Magic :D

(e) Code generator

- Knows what the operations mean in assembly (machine code)
- Generates the machine code program file

3. Machine Code

Each step in a compiler has 3 things that it **must** have:

- Input
- What must it know beforehand
- Output

So what are we going to use to compile??

13.1 Our tools!

`flex` - A command line utility to take source code and output a token list.

`bison` - A parser and semantic analyzer

```
sudo apt-get install flex bison
```

And in the case of Java, `javacc`:

```
javacc - Parser generator for use with Java
javacc-doc - Documentation for the JavaCC Parser Generator
jtb - syntax tree builder and visitors generator for JavaCC
libjavacc-maven-plugin-java - maven plugin which uses JavaCC to process JavaCC
grammar files
```

And our code generator? Well, we'll be writing that ourselves.

14 May 3 - 4, 2016: Animation

The principle of animating a translation is to perform a certain percentage of the translation successively. If we want to translate 400 pixels along the x axis, then we can perform it in increments of 25%, moving 100 pixels at a time.

We can use a potentiometer, I mean, a *knob*, that indicates a **factor**. We also need a function called **vary** that takes 4 parameters: the name of the knob, the start and end frames, and the start and end values.

In order to implement these frames and knobs, we need to do the following process:

1. Pass 1
   - Check for animation commands (`frame`, `basename`, `vary`)
   - Check for basic animatino command errors
     - Check if `vary` keeps within the bounds set by `frame`
     - Check if `frame` exists if `vary` exists
   - Set the number of frames
   - If `basename` is not set, then use a default value and alert the user.
2. Pass 2 (Only runs if Pass 1 finds animation commands)
   - Calculate and store knob values
   - Create an array / list where each index maps to a frame and contains a list of knob names and values for that frame

   For example:

   ```
   vary k 0 10 0 1
   vary x 3  6 1 0
   ```

3. Pass 3
   - If no animation code, draw a single image and exit.
   - If there is animation code, for each frame:
     (a) Set all knob values in symbol table based on the the list from Pass 2
     (b) Go through all the drawing commands and apply knob values when appropriate
     (c) Save the image as `basename + frame_no`

Frame	k	x
0	0.0	1.0
1	0.1	1.0
2	0.2	1.0
3	0.3	1.0
4	0.4	0.67
5	0.5	0.33
6	0.6	0.0
7	0.7	0.0
8	0.8	0.0
9	0.9	0.0
10	1.0	0.0

15 May 12, 2016 - Filling in shapes, and preparing for lighting!

Now that we have a decent interpretter / scripting type language, it makes our lives a lot easier to test, draw images, etc. Now, we can focus on improving our single frame rendering now that we have a ton of overhead out of the way. One of the things we need to do is to **fill** in polygons. There are a few ways to do this:

- Flood fill
- Sweeping out the triangles with a ton of lines
- Using a bounding rectangle and checking each pixel
- Dilating triangles from size multiplier 1.0 to 0 and drawing each out
- and many many many more

Out of these ideas, we want the *least memory intensive one*, as this is going to a very common operation. Because our line algorithm is so optimized already, we can use the "sweeping lines" idea, combined with a "horizontal line" flood fill, known as the **scanline conversion**.

15.1 Scanline Conversion

Scanline conversion is where we start from a vertex, and draw vertical OR horizontal lines in the direction of the other 2 vertices to fill in the triangle.

Some characteristics of scanline conversion:

- Filling in a polygon with a series of horizontal or vertical lines
- Need to identify the top, bottom, and middle vertices for each polygon.
- We will denote B as the bottom vertex, M as the middle vertex, and T as the top vertex, using their y axis coordinates

We have $(x_0 y_0)$ that is always on $\bar{BT}$, and (x_1, y_1) that will either be on $B\bar{M}$ or $\bar{MT}$.

The horizontal line will have $y_0 = y_1$, and will move upwards from B to T.

1. Our first line is $\bar{BB}$. Simply, `x0 = x1 = B.x` and `y0 = y1 = B.y`
2. Our second line goes from $(x_B + \Delta X_1, y_B + 1)$ to $(x_B + \Delta X_2, y_B + 1)$.

   To find ΔX_1, we know that the first point is on $\bar{BT}$. Using some simple coordinate geometry and the line equation, we can see:

$$\begin{aligned} \Delta X_1 &= \tfrac{X_T - X_B}{Y_T - Y_B} \\ \Delta X_2 &= \tfrac{X_M - X_B}{Y_M - Y_B} \text{ on } B\bar{M} \\ \Delta X_2 &= \tfrac{X_T - X_M}{Y_T - Y_M} \text{ on } \bar{MT} \end{aligned}$$

However, these Δ values are floating point values. This is not very convenient for us, since we want to use integers for pixel locations. Another issue is division by 0. We will always have a "true" top point and a "true" bottom point, but the middle point may lie on the same horizontal as B or T, which causes the denominator being 0. We could simple add checks beforehand, use division by 0 = infinity, and division by infinity = 0 hackery, or any other such solution. Preventing this is trivial.

16 May 18, 2016 - Z Buffering

Now that we are filling in faces, if we try multiple figures, some pixels might be overwritten. We can solve this "overlapping" problem using the z-buffering technique.

In Z Buffering, we track a 2D array with the same dimensions as our canvas. Each index stores a floating point value that represents a Z coordinate. We should ONLY draw things with greater Z coordinate values than those that currently exist. For example, drawing in index (250, 150), if we first draw a green pixel with Z value 23, it is drawn in index (250, 150) since there is nothing

yet there. If we later try to draw a blue pixel at index (250, 150) with a Z value of 8, 8 is less than 23 so we do NOT draw it.

As such, we should use `Double.MIN_VALUE` as default values, as this is the least possible element in this Z buffer. To implement this in our current graphics engine, we need to change our `plot` function, `draw_line` and `draw_polygon` function. For example, if we need to draw faces, we should pass that boolean as a parameter to plot, so plot can track a Z buffer. This also requires passing a Z value to `draw_line`. Anything else that uses `draw_line` will also need to be modified.

17 May 23, 2016 - Modeling Real Lighting

Colors are calculated by looking at:

1. The reflective properties of each object
2. The properties of the light hitting each object

There is, of course, a standard lighting equation in computer graphics, derived after many years of computer graphics research:

- We want this equation to generate a 3 number color code value for each polygon / pixel.
- If you want a grayscale image, you only need to calculate the color once per polygon / pixel.
- If you want a color image, you must calculate each separate color value for red, gree, and blue for each polygon.

Let I denote "illumination".

$$I = I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}}$$

Ambient, **diffused**, and **specular** light are different kinds of reflected light.

Ambient light is also known as *background* light. Ambient light is more or less evenly distributed (it hits all objects equally), and comes from all directions.

Another form of light source is **point light** or **spotlight** sources. These are lights that come from a specific point and are focused onto a particular area, being extremely intense in those areas,

and extremely dim elsewhere. Their strength is therefore determined by an objects position away from the source. **Diffuse** lighting and **specular** lighting deals with point light sources.

I_{ambient}: a combination of the color of ambient light and the amount and color that the object reflects.

I_{diffuse}: Diffuse reflection reflects point lights back equally in all directions. It is based on the locations of the light and the object.

I_{specular}: Comes from a point light source, and is reflected back at a specific direction. Thus we need to worry about what angle we are looking at the object as well.

17.1 Representing Illumination in Code

There are now 2 things that affect our code. **Quality of Light** and **Reflective Properties**.

17.1.1 Quality of Light

For an ambient light, we will represent the quality of light (color value) as a number between 0 and 255.

For a point light, we need to represent it as a color value and a 3 dimensional point.

17.1.2 Reflective Properties

Represented as the percentage of light reflected back as constants: k_{ambient}, k_{diffuse}, and k_{specular}.

This will also satisfy the relation:

$$0 \leq k_a, k_d, k_s$$
$$k_a + k_d + k_s = 1$$

18 May 26, 2016: The Lighting Equation

$$I = I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}}$$

Ambient reflection requires us to have variables for:

- Color value (C_a)
- Constant of ambient reflection (K_a)

$$I_a = C_a K_a$$

Diffuse reflection requires us to have variables for:

- Color point source (C_p)
- Constant of diffuse reflection (K_d)
- The angle of incidence (θ)

$$I_d = C_p K_d \cos\theta$$

In specular reflection, we have the vector of incoming light, the color of the point source, and the view angle. We also have the constant of specularity K_s.

Note that $\vec{L}$ is the vector of incoming light, and $\vec{N}$ is the normal vector. $\vec{L} \cdot \vec{N} = \|\vec{L}\|\|\vec{N}\| \cos\theta$

We will have to turn $\vec{L}$ and $\vec{N}$ into unit vectors via normalizing!

There is one more thing we need to keep track of: the view vector: $\vec{V}$. Let α be the angle between $\vec{V}$ and the reflected ray $\vec{R}$. To find $\vec{R}$. We can project $\vec{L}$ onto $\vec{N}$ and then use that to create $\vec{R}$.

Note that $\vec{L}$ and $\vec{N}$ are *normalized*. Therefore their magnitudes are 1. We can use the relation:

$$\vec{P} = \|\vec{N}\| \cos\theta = \|\vec{N}\|(\vec{L} \cdot \vec{N})$$

Letting $\vec{S}$ be the perpendicular from $\vec{L}$ to $\vec{N}$, we know that $\vec{S} = \vec{N}(\vec{L} \cdot \vec{N}) - \vec{L}$, which we can plug into $\vec{R} = \vec{P} + \vec{S}$. Thus, $\vec{R} = 2 * \vec{P} - \vec{L}$.

Using the dot product of $\vec{V}$ and $\vec{R}$ to get the angle α between them.

$$I_s = C_p K_s \cos\alpha$$

We can also raise the $\cos\alpha$ term to some power n to tune the value.

18.1 Issues with calculating color values

Color values can become negative after it is computed! Basically turn everything less than 0 into 0. If a color value is higher than 255, then make it 255.

18.2 Shading Models

- Flat shading: Calculate I once per polygon.
- Goroud shading: vertex normals for each polygon. Combine surface normals of all polygons that share a vertex

19 May 31, 2016 - Shading models

Gouraud, Goroud, whatever, shading involves:

- Calculate I once per vertex normal (normalized sum of the surface normals that share a vertex).
- Interpolate I during scanline conversion and during drawing lines

In the Phong Shading model:

- Calculate vertex normals
- Interpolate the normal in scanline conversion and drawline
- Recalculate I for each PIXEL