Brad Burkman's Notes for
CSCE 515 Principles of Computer Graphics
Dr. Christoph Borst, CBDI co-PI, OLVR 342, `cbx99999@louisiana.edu`
Fall 2018
`bburkman@lsmsa.edu`
`C00412257@louisiana.edu`

Grader: Jason Woodworth, `downtothelastpixel@gmail.com`

Contents

1 Tuesday 21 August: Introduction

Ray Tracing

OpenGL 3.3

Couldn't find a textbook that did both the math and OpenGL 3.3 well.

Cross Product v/s Dot Product of Vectors

Start with how lines and triangles are rendered.

Acknowledge sources I used in homework and projects.

Exam will be hard.

Lagniappe - Come to VR lab to do a project.

2 Math Review: Dot Product and Cross Product

Example of Dot Product and Cross Product:
$\vec{u} = \langle 1, 2, 3 \rangle$
$\vec{v} = \langle 4, 5, 6 \rangle$

$\vec{u} \cdot \vec{v} = 1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 4 + 10 + 18 = 32$

$\vec{w} = \vec{u} \times \vec{v} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{vmatrix} = \begin{vmatrix} 2 & 3 \\ 5 & 6 \end{vmatrix} \vec{i} - \begin{vmatrix} 1 & 3 \\ 4 & 6 \end{vmatrix} \vec{j} + \begin{vmatrix} 1 & 2 \\ 4 & 5 \end{vmatrix} \vec{k} = -3\vec{i} + 6\vec{j} - 3\vec{k} =$
$\langle -3, 6, -3 \rangle$

Is the cross product commutative?

$\vec{v} \times \vec{u} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{vmatrix} = \begin{vmatrix} 5 & 6 \\ 2 & 3 \end{vmatrix} \vec{i} - \begin{vmatrix} 4 & 6 \\ 1 & 3 \end{vmatrix} \vec{j} + \begin{vmatrix} 4 & 5 \\ 1 & 2 \end{vmatrix} \vec{k} = 3\vec{i} - 6\vec{j} + 3\vec{k} =$
$\langle 3, -6, 3 \rangle = -\vec{w}$

No, it's anti-commutative, but that's okay. The cross product gives a vector, $\vec{w}$, orthogonal to both $\vec{u}$ and $\vec{v}$, and a constant multiple of $\vec{w}$ is still orthogonal to both other vectors.

The cross product is also not associative, but satisfies the Jacobi identity.

$$a \times (b \times c) + b \times (c \times a) + c \times (a \times b) = 0 \ \forall a, b, c \in V$$

3 Math Review: Algebra

A **group** is a set S with an operation ("+") such that:

The set S is *closed* under the operation, meaning that if $a, b \in S$, then $a + b \in S$.

The operation is associative, meaning that if $a, b, c \in S$, then $a + (b + c) = (a + b) + c$.

The set S contains a unit element ("0"), such that $a + 0 = 0 + a = a \ \forall \ a \in S$.

For every element $a \in S$, there is an inverse element, $-a$, such that $a + (-a) = -a + a = 0$.

If the operation is commutative, meaning $a + b = b + a \ \forall \ a, b, \in S$, then S is called an **Abelian group.**

A **ring** is a set R with two operations, $+$ and $\cdot$, such that:

The set R is an Abelian group.

The set R is closed under multiplication.

Multiplication is associative.

The distributive laws hold:

$a \cdot (b + c) = a \cdot b + a \cdot c$

$(b + c) \cdot a = b \cdot a + c \cdot a$

A ring with a multiplicative identity ("1") is called a **ring with unit**.

If the ring has the property that if $a \cdot b = 0$ then $a = 0$ or $b = 0$, it is called a **domain**.

If multiplication is commutative, then the ring is called a **commutative ring**.

If a domain is commutative, then it is called an **integral domain**.

A **field** is a commutative ring with unit element ("1") such that every nonzero element has an inverse.

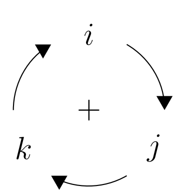
Quaternions are not a field because multiplication of quaternions is not commutative; however, every nonzero quaternion has a multiplicative inverse, so they are called a **division ring**.

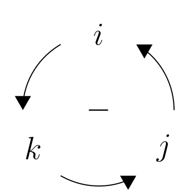
4 Math Review: Quaternions

Complex numbers, $a + bi$ where a and b are real numbers and $i = \sqrt{-1}$, are two-dimensional numbers. Quaternions, $a + bi + cj + dk$, are four-dimensional numbers. Just as you can think of complex numbers being two-dimensional vectors having the basis vectors $1 = \langle 1, 0 \rangle$ and $i = \langle 0, 1 \rangle$, you can think of quaternions as four-dimensional vectors having the basis vectors $1 = \langle 1, 0, 0, 0 \rangle$, $i = \langle 0, 1, 0, 0 \rangle$, $j = \langle 0, 0, 1, 0 \rangle$, $k = \langle 0, 0, 0, 1 \rangle$,

While complex numbers have the basis elements 1 and i with, $i^2 = -1$, quaternions have this multiplication scheme for their basis elements. Note that $i^2 = j^2 = k^2 = ijk = -1$, but they are anti-commutative, with, for example, $ij = -ji$.

$\times$	1	i	j	k
1	1	i	j	k
i	i	-1	k	$-j$
j	j	$-k$	-1	i
k	k	j	$-i$	-1





The inverse of a quaternion is given by:

$$(a+bi+cj+dk)^{-1} = \frac{a-bi-cj-dk}{a^2+b^2+c^2+d^2}$$

Vector Form of Quaternions

Think of $a+bi+cj+dk$ as the pair, $(a, \langle b,c,d\rangle)$, with a scalar part and a vector part.
Then quaternion addition is $(r_1,\vec{v}_1)+(r_2,\vec{v}_2) = (r_1+r_2, \vec{v}_1+\vec{v}_2)$.
Vector multiplication is $(r_1,\vec{v}_1)(r_2,\vec{v}_2) = (r_1r_2 - \vec{v}_1\cdot\vec{v}_2,\ r_1\vec{v}_2 + r_2\vec{v}_1 + \vec{v}_1\times\vec{v}_2)$

4.1 Quaternions as Rotations

We start with a vector, $\vec{p} = (p_x, p_y, p_z)$, written as a quaternion with real coordinate zero,

$$p = p_x\mathbf{i} + p_y\mathbf{j} + p_z\mathbf{k}$$

We want a rotation of p through an angle of θ about the axis defined by a unit vector

$$\vec{u} = (u_x, u_y, u_z) = u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k}$$

In two dimensions, Euler's Formula, $e^{i\theta} = \cos\theta + i\sin\theta$, gives a counterclockwise rotation of θ. We can extend it to three dimensions as

$$q = e^{\frac{\theta}{2}(u_x\mathbf{i}+u_y\mathbf{j}+u_z\mathbf{k})} = \cos\frac{\theta}{2} + (u_x\mathbf{i}+u_y\mathbf{j}+u_z\mathbf{k})\sin\frac{\theta}{2}$$

The rotation of $\vec{p}$ about $\vec{u}$ is given by

$$p' = qpq^{-1}$$

Going back to using Euler's Formula for a rotation, let's look at the multiplicative inverse of q and see that it's consistent with previous knowledge. By a previous formula,

$$a^{-1} = (a_w + a_x\mathbf{i} + a_y\mathbf{j} + a_z\mathbf{k})^{-1} = \frac{1}{a_w^2+a_x^2+a_y^2+a_z^2}(a_w - a_x\mathbf{i} - a_y\mathbf{j} - a_z\mathbf{k})$$

The vector $\vec{u} = u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k}$ is a unit vector with real part zero, so the denominator is 1.

$$(\vec{u})^{-1} = (u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k})^{-1} = -u_x\mathbf{i} - u_y\mathbf{j} - u_z\mathbf{k} = -\vec{u}$$

Applying the extension of Euler's Formula,

$$q = e^{\frac{\theta}{2}(u_x\mathbf{i}+u_y\mathbf{j}+u_z\mathbf{k})} = \cos\frac{\theta}{2} + (u_x\mathbf{i}+u_y\mathbf{j}+u_z\mathbf{k})\sin\frac{\theta}{2}$$

and the inverse formula

$$a^{-1} = (a_w + a_x\mathbf{i} + a_y\mathbf{j} + a_z\mathbf{k})^{-1} = \frac{1}{a_w^2+a_x^2+a_y^2+a_z^2}(a_w - a_x\mathbf{i} - a_y\mathbf{j} - a_z\mathbf{k})$$

we get

$$\begin{aligned}
q^{-1} &= \frac{1}{\cos^2\frac{\theta}{2} + u_x^2\sin^2\frac{\theta}{2} + u_y^2\sin^2\frac{\theta}{2} + u_z^2\sin^2\frac{\theta}{2}}\left(\cos\frac{\theta}{2} - (u_x\mathbf{i}+u_y\mathbf{j}+u_z\mathbf{k})\sin\frac{\theta}{2}\right) \\
&= \frac{1}{\cos^2\frac{\theta}{2} + (u_x^2+u_y^2+u_z^2)\sin^2\frac{\theta}{2}}\left(\cos\left(-\frac{\theta}{2}\right) + (u_x\mathbf{i}+u_y\mathbf{j}+u_z\mathbf{k})\sin\left(-\frac{\theta}{2}\right)\right) \\
&= \frac{1}{\cos^2\frac{\theta}{2} + \sin^2\frac{\theta}{2}}\left(\cos\left(-\frac{\theta}{2}\right) + (u_x\mathbf{i}+u_y\mathbf{j}+u_z\mathbf{k})\sin\left(-\frac{\theta}{2}\right)\right) \\
&= e^{\frac{\theta}{2}(-u)} = \left(e^{\frac{\theta}{2}u}\right)^{-1}
\end{aligned}$$

5 Thursday 23 August: Raster Display v/s Vector Display

Ray Tracing: Which objects intersect the line?

Polygon Projection (Polygon Rendering)
Triangles on the screen
Vertex coordinates
Multiply by matrices to convert to screen's coordinate system

Modeling Coordinates
→ World Coordinates
→ Viewing and Projection Coordinates (Camera View)
→ Normalized Coordinates
→ Device Coordinates

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \rightarrow \begin{array}{l}\text{Model}\\\text{Martrix}\end{array} \rightarrow \begin{array}{l}\text{Projection}\\\text{Matrix}\\\text{(Camera}\\\text{View)}\end{array} \rightarrow \begin{array}{l}\text{Perspective}\\\text{Division}\end{array} \rightarrow \begin{array}{l}\text{Viewpoint}\\\text{Transformation}\end{array}$$

2D Scan Conversion: Converts 2D object description to pixel values.
"Pixel" Picture Element
Visible Surface Determination: Occlusion

Direct Illumination v/s Global Illumination: Shadows not completely dark
Curves and surfaces give a smooth alternative to polygonal representation of objects.

Lots of student projects have dealt with **Particle Dynamics**

Early Video Games Nimatron (1940) First "video" game used an oscilloscope as its screen.

Raster Display - Image broken into pixels
Vector Display - Smooth curves, like moving lasers or electron beam.

Black & white **bitmap** has one bit per pixel

Raster Displays
More computationally expensive
Requires more memory
Constant refresh rate
Supports area fills
Won over vector after memory got cheap.

Frame buffers are getting more complex.
- Double buffering so screen doesn't refresh in the middle of a memory move
- Left and right buffers for stereoscopic rendering
- Depth buffer for occlusion

Know from Today's Lesson
Pixel, raster, bitmap, frame buffer, *aliasing*
Know the difference between raster and vector.

Aliasing - in computer graphics, the jagged, or saw-toothed appearance of curved or diagonal lines on a low-resolution monitor.

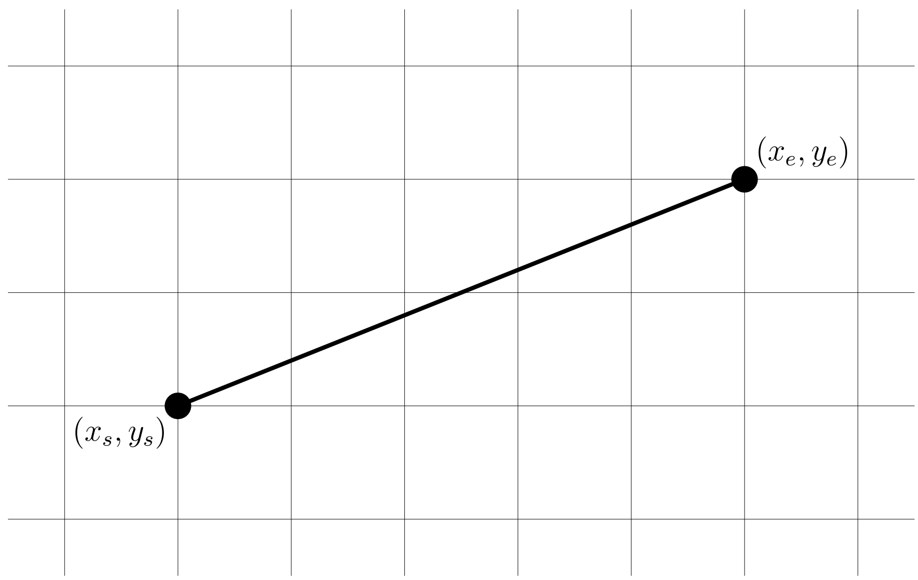
6 Tuesday 28 August: Scan Conversion of a Line Segment

Simplifying Assumptions

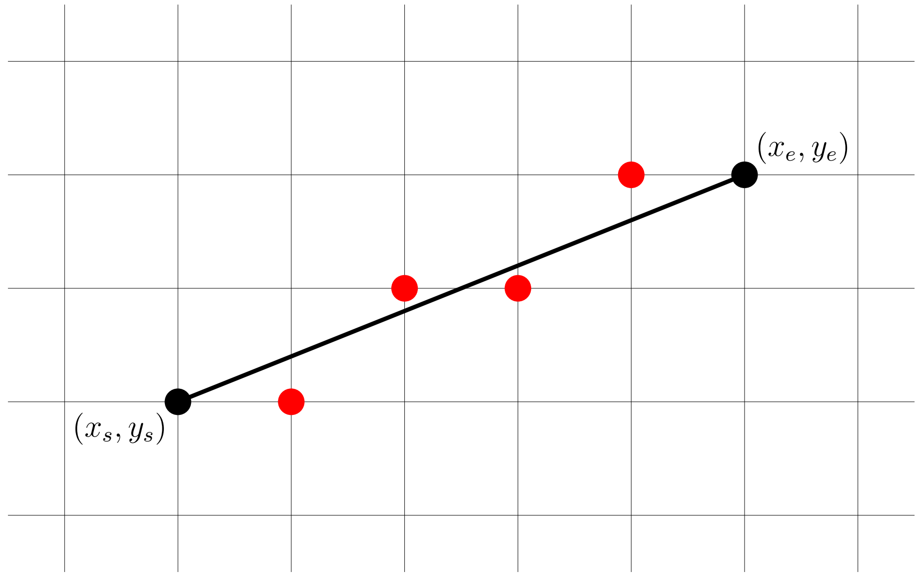
1-pixel-thick line on a B&W display (monochrome)

$m \in [-1, 1]$. More elaborate code can deal with other slopes.

Line segment described by two integer endpoints (endpoints fall exactly on pixels) (x_s, y_s), (x_e, y_e)



Main Idea: Activate one pixel per column from x_s to x_e.



Simple, but Slow, Algorithm
$m = (y_e - y_s)/(x_e - x_s)$
$b = y_s - m \cdot x_s$
for x from x_s to x_e *(inclusive)*
    color pixel at $(x, \lfloor m \cdot x + b + 0.5 \rfloor)$

The problem with this algorithm that is the $m \cdot x$, floating-point multiplication, is really computationally expensive, usually six cycles, while addition is relatively cheap, usually one cycle.

Rendering algorithms have to be as fast as possible, because they run billions of times.

Basic Incremental Algorithm (DDA, Digital Differential Analyzer)
$m = (y_e - y_s)/(x_e - x_s)$
$b = y_s - m \cdot x_s$
$x_0 = x_s$
$y_0 = y_s$
for i from 1 to $(x_e - x_s)$
    $(x_{i+1}, y_{i+1}) = (x_i + 1, y_i + m)$
    color pixel at $(x_i + 1, \lfloor y_{i+1} + 0.5 \rfloor)$

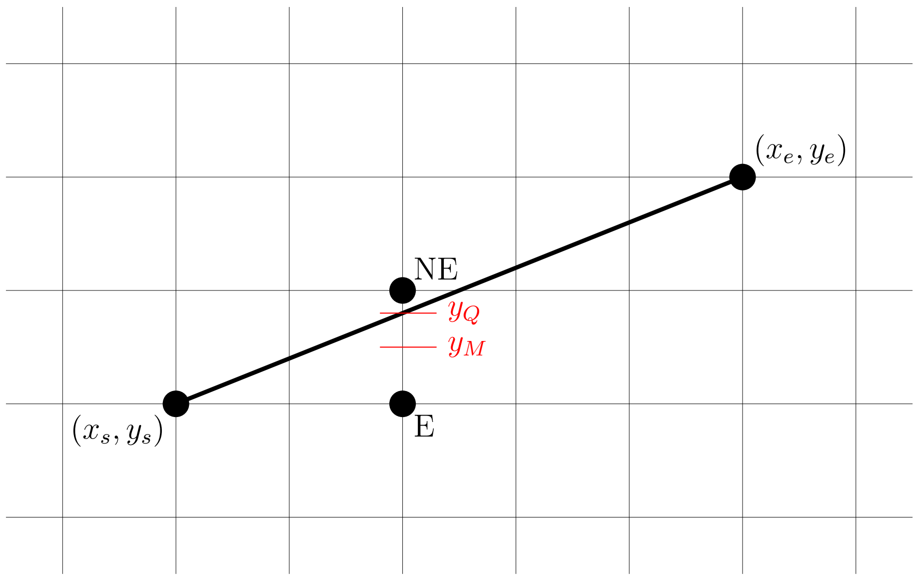
Midpoint Algorithm (Besenham)
Developed for pen plotter.
Bottleneck was algorithm and computational speed, not hardware speed.

Simplifying assumption: $m \in [0, 1]$

Main idea

Move either E or NE in each move.

Choose based on value of decision variable, d, which lets us choose betwen E and NE.



Notation: Q for *crossing*, M for *midpoint*.

Algorithm

$d = y_Q - y_M$.

Look at sign of d.

Move NE when d is positive.

Move E otherwise.

Computing d

Initialize $d = m - 0.5$

Increment:

for E moves: $d = d + m$

for NE moves: $d = d + m - 1$

Here's how Midpoint is cheaper than DDA: We can change everything to integers and not have any floats.

Make Everything Integers

Scale everything by $2\Delta x$, so that $m - 0.5$ becomes an integer.
$d = 2\Delta y - \Delta x$
East increment: $d = d + 2\Delta y$
NE increment: $d = d + 2\Delta y - 2\Delta x$

These methods aren't what we use today. Probably triangle scan conversion.

7 Thursday 30 August: Scan Conversion of a Triangle

7.1 Homework 1

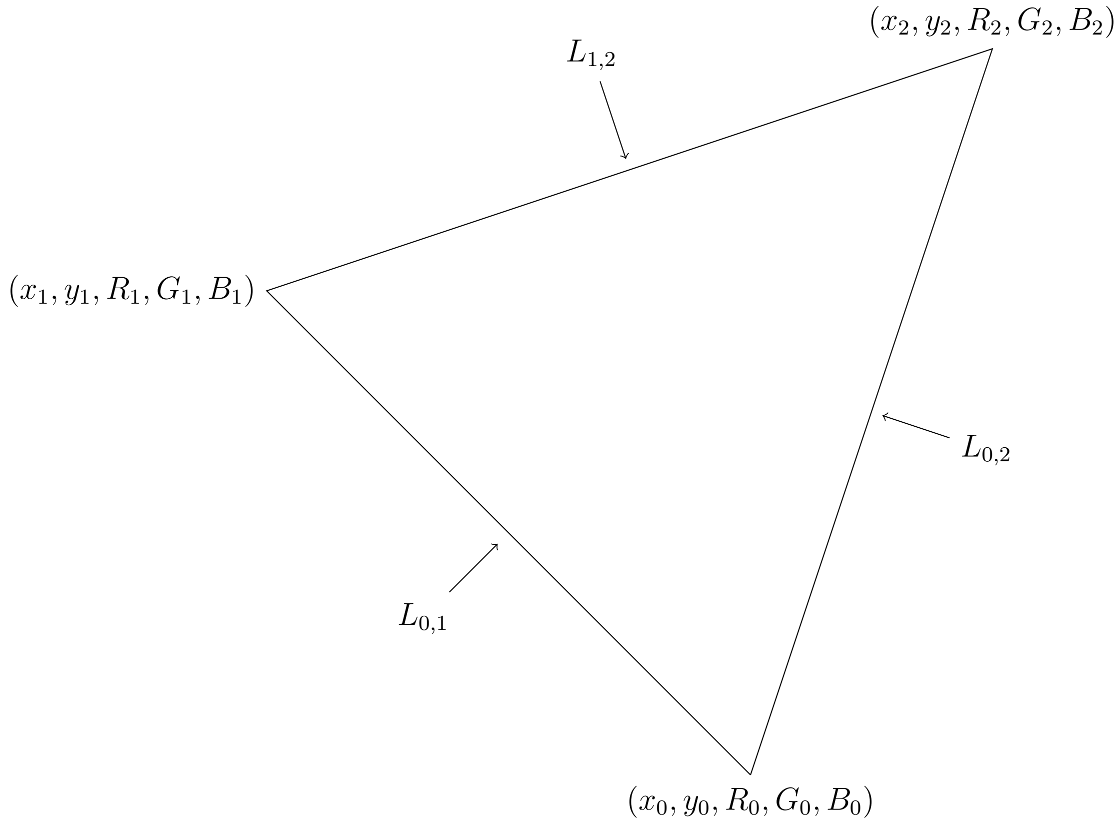
due 13 September
Callback Do sth when sth happens
Will have global variables
Assignment is given as a triangle with a certain order of x_0, x_1, and x_2. Extra part of assignment is to account for different orders.

7.2 Midpoint Algorithm (Besenham) Pseudocode

Initialize integers Δ_E, Δ_{NE}, d, x, and y.
Set pixel at (x, y) (first pixel)

while $x <$ last column (given by an endpoint)
{
    increment x by 1
    if $(d < 0)$
        add Δ_E to d
    else
    {
        increment y by 1
        add Δ_{NE} to d
    }
    set pixel at (x, y)
}

7.3 Naming Conventions



Simplifying Assumptions:

Vertices have integer coordinates

$y_0 \leq y_1 \leq y_2$

Idea

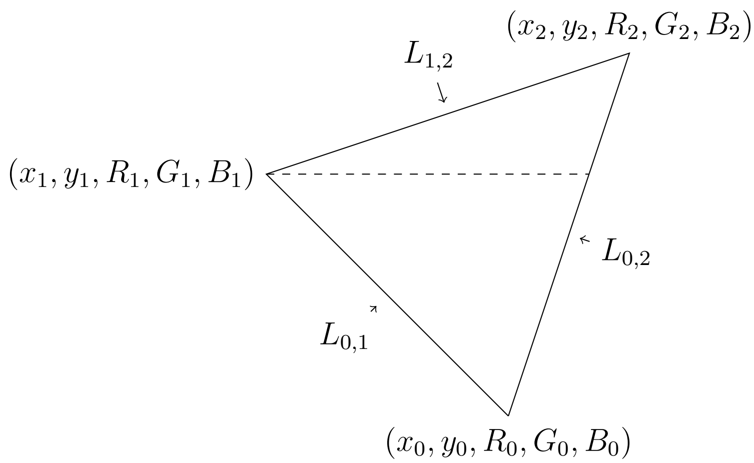
We want to interpolate the colors to get shading.

Look at the triangle one scan line at a time, looping bottom to top, left to right.

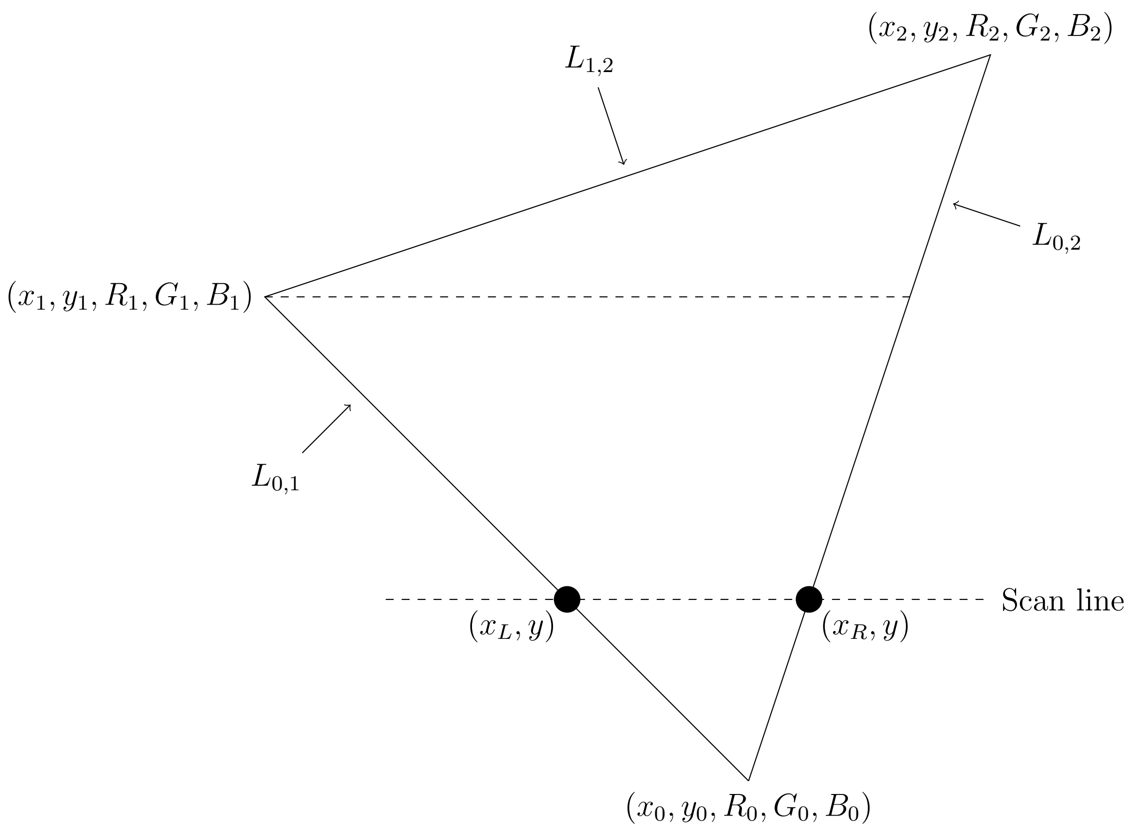
Two loops

One for the bottom

One for the top



Scan Line



7.4 Line Scan Pseudocode

for y from y_0 to $(y_1 - 1)$

{

Calculate coordinates where scan line intersects $L_{0,1}$ and $L_{0,2}$.

(*i.e.* Calculate x_L and x_R incrementally)

Color the pixels from $(\lceil x_L \rceil, y)$ to $(\lfloor x_R \rfloor, y)$

}

for y from y_1 to y_2

{

Calculate coordinates where scan line intersects $L_{1,2}$ and $L_{0,2}$.

(*i.e.* Calculate x_L and x_R incrementally)

Color the pixels from $(\lceil x_L \rceil, y)$ to $(\lfloor x_R \rfloor, y)$

}

7.5 Special Cases

Watch for division by zero!

Horizontal lines, either between V_0 and V_1 or between V_1 and V_2.

7.6 Calculating x_L and x_R Incrementally

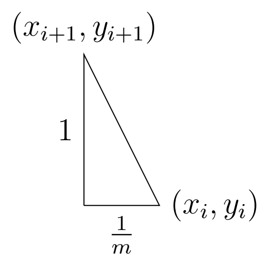
Initialize $x_L = x_0$, $x_R = x_0$, $y = y_0$

Note that y is being incremented by 1.

$$x_{i+1} = x_i + \frac{1}{m} = x_i + \frac{x_1 - x_0}{y_1 - y_0}$$

$$x_L = x_L + +\frac{x_1 - x_0}{y_1 - y_0}$$

$$x_R = x_R + +\frac{x_2 - x_0}{y_2 - y_0}$$



7.7 Color

Notes on this section:

Make sure we're consistent about when to use $x_L - x_R$ and when to use $\lceil x_L \rceil - \lfloor x_R \rfloor$.

Computing color incrementally (Here considering only R, red)

Initialize $R_L = R_0$, $R_R = R_0$

For each scan line:

$$R_L = R_L + \frac{R_1 - R_0}{y_1 - y_0}$$

$$R_R = R_R + \frac{R_2 - R_0}{y_2 - y_0}$$

Within a scan line, between $\lceil x_L \rceil$ and $\lfloor x_R \rfloor$

Initialize $R = R_L + \dfrac{R_L - R_R}{x_L - x_R}(\lceil x_L \rceil - x_L)$
Color first pixel
for R from R_L to R_R
{
$$R = R + \frac{R_L - R_R}{\lceil x_L \rceil - \lfloor x_R \rfloor}$$
[Do the same for G and B.]
Color pixel (x, y, R, G, B)
}

8 Running OpenGL on a Mac

8.1 Installing OpenGL

You don't have to. It comes in the box.

8.2 Visual Studio

`https://social.msdn.microsoft.com/Forums/en-US/ef99e9f5-2a48-423b-b6c0-fa5617d7c63d/how-do-i-get-c-to-work-on-visual-studio-for-mac?forum=visualstudiogeneral`
This post was from March 2017, but I think it's still true. The Microsoft person says that Visual Studio for Mac is designed for building mobile apps, not for general computing. It does not support `C++`. She kindly recommends that you run Windows.

8.3 How Brad Does It

- Install GLEW and GLFW. The easiest way to do it is with Homebrew, which is one of the programs you can install that lets you unleash the Linux power of your Mac.

  ```
  brew install glew
  ```

  Another way to do it is to download the .zip files from the GLEW and GLFW websites and build. GLEW comes with a `Makefile`, but for GLFW you have to use CMake.

- Link your code to `glew.h` and `glfw3.h`.

  One way to link your code to the library is to replace the

```
#include <GLFW/glfw3.h>
```

with the path to the `glfw3` file on your machine, something like

```
#include </usr/local/Cellar/glfw/3.2.1/include/GLFW/glfw3.h>
```

- Tell OpenGL which version you want to use.

  If you want to run an earlier version of OpenGL, as in Assignment 1 (so you can use `glDrawPixels`, which was removed from the language in Version 3.2), ignore this step, and it will default to 2.1.

  This trick only works on my Mac if I specify version 3.3.

  Put this code in your `main()` function.

  ```
  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
  glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
  glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
  ```

- Compile and Link.

  I use `g++`, the GNU compiler for `C++`. I use it not just because I'm old (which I am), but because I often run my big jobs in parallel on remote UNIX clusters, and they don't have integrated development environments (IDE's) like Visual Studio. You have to do everything from a UNIX command line. Since I have to be proficient in those tools anyway, I use them on my local computer.

  Here's the command I use to compile and link my code.

  ```
  g++ Assignment1.cpp -framework OpenGL -lGLEW -lGLFW
  ```

8.4 How Other People Do It

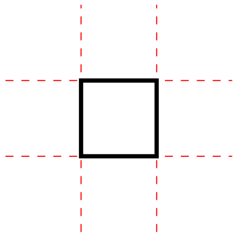
Send me your stories and I'll put them in.

9 Tuesday 4 September: Clipping

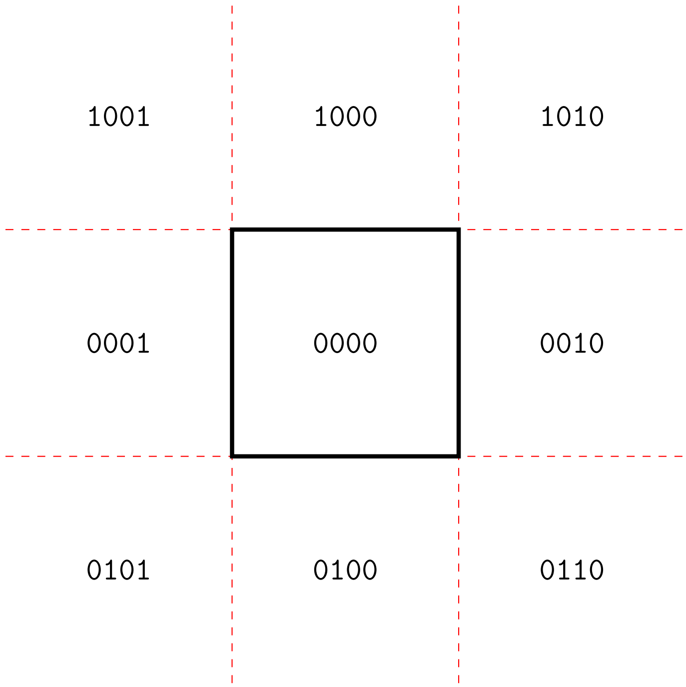
9.1 Cohen-Sutherland Line Clipper

We don't want to scan-convert things outside the viewing window. Throw away the part of the line outside the window.

Culling is different. That's throwing out objects that do not intersect the window at all.



Each of these red dashed lines divides the plane into "inner" and "outer" sections.



Encoding
Region out-codes

Pick an order (and be consistent)

Dr. Borst's order in today's class:
Top, Bottom, Right, Left

Encode each sector with a four-digit binary code in which each digit tells whether the points are inside (0) or outside (1) the boundary.

"On the line" counts as inside.

Iterative Algorithm

Assign codes to segment endpoints.

IF the segment is entirely in the window (both codes 0000)

Accept the segment

ELIF both endpoints are outside a common boundary, *e.g.* both to the left, *i.e.* logical AND of codes is nonzero,

Reject the segment.

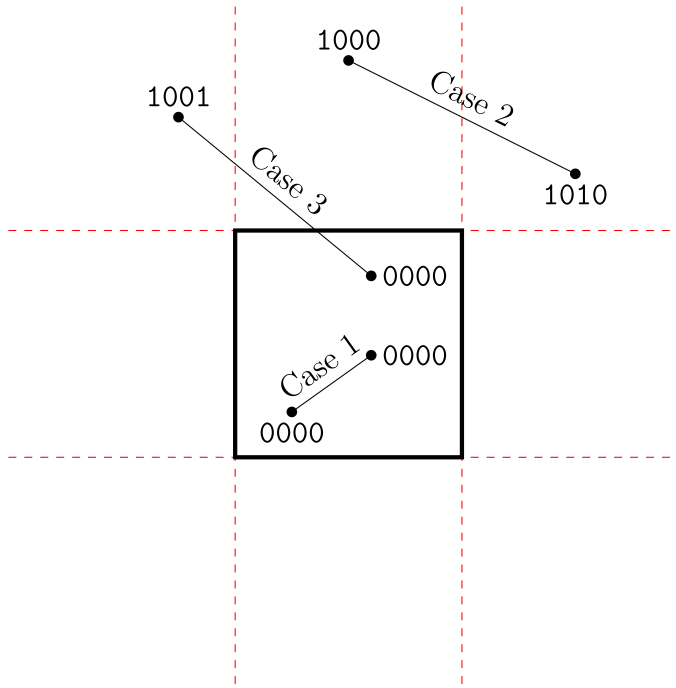
(Same as culling.)

ELSE

Cut segment at a boundary

Discard the outer part

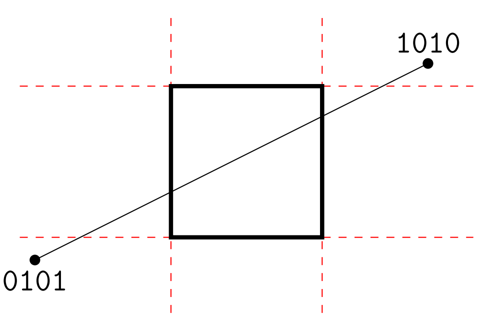
Run C-S Clipper on the inner part.



To Cut the Line

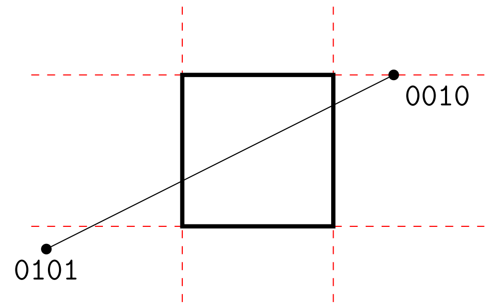
Pick an outside (not 0000) endpoint.
Walk through the four digits left to right.
Identify the first nonzero bit in the code.
Cut the line at the boundary that corresponds to that bit.
Lather, rinse, repeat.

1. Scan the point codes left to right and pick the first one with a 1. There will be only one, because if they both had a 1 in the same position, they would have already been eliminated as being outside the same boundary. Choose 1010.



2. In `1010`, scan left to right to find the first `1`, which here represents “Top.” Cut at the top boundary.

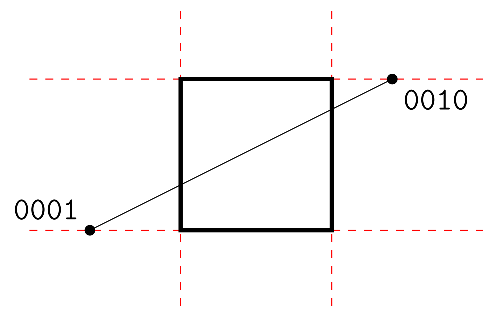
3. Check to see whether both points have encoding `0000`. If they do, accept and end. If not, check to see whether the encodings of the two points share a `1`. If they do, reject the line.



4. Scan the point codes left to right and pick the first one with a `1`. Choose `0101`.

5. In `0101`, scan left to right to find the first `1`, which here represents “Bottom.” Cut at the bottom boundary.

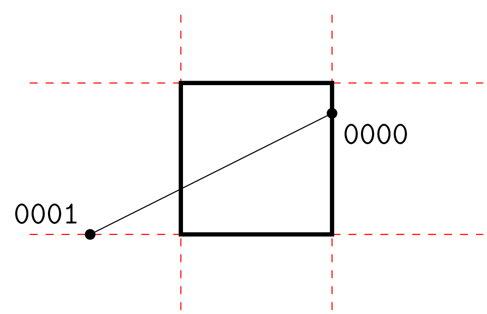
6. Check to see whether both points have encoding `0000`. If they do, accept and end. If not, check to see whether the encodings of the two points share a `1`. If they do, reject the line.



7. Scan the point codes left to right and pick the first one with a `1`. Choose `0010`.

8. In `0010`, scan left to right to find the first `1`, which here represents “Right.” Cut at the right boundary.

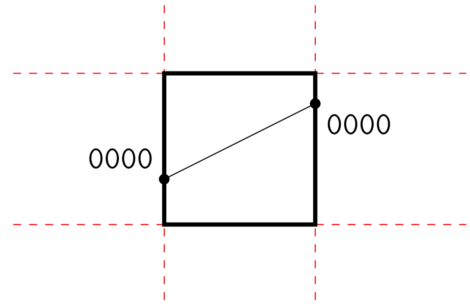
9. Check to see whether both points have encoding `0000`. If they do, accept and end. If not, check to see whether the encodings of the two points share a `1`. If they do, reject the line.



10. Scan the point codes left to right and pick the first one with a `1`. Choose `0001`.

11. In `0001`, scan left to right to find the first `1`, which here represents "Left." Cut at the left boundary.

12. Check to see whether both points have encoding `0000`. Since they do, accept and end.

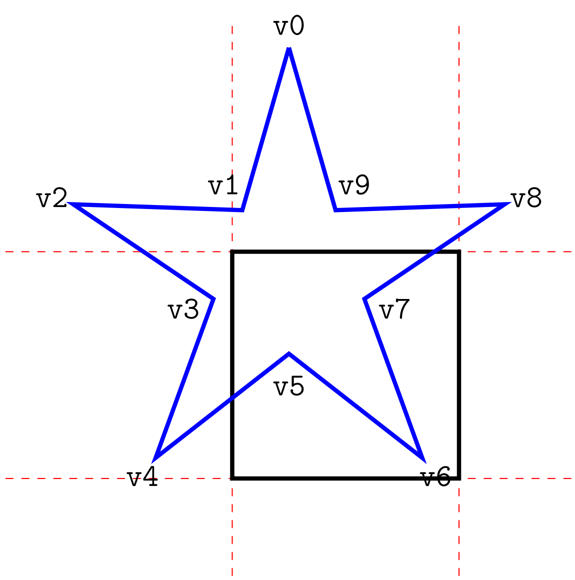


Generalizes to 3D.

9.2 Clipping Polygons

Not clear about what to do with each vertex in the algorithm, whether to discard it or not.

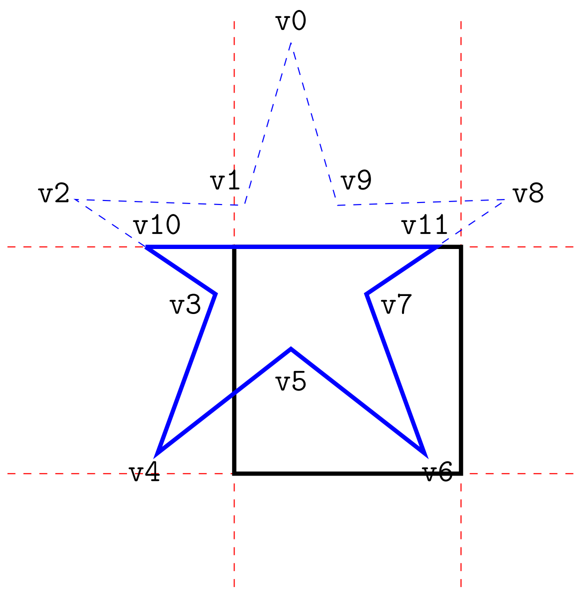
The math is a little more tedious, but not much.
Window must be convex.
Convex casual definition: If a bug walking around the perimeter only turns one way, it's convex.



Algorithm
Pick a boundary line.
Walk around in one direction, slicing the segments that intersect the boundary line.

Top
`v0 -- v1` is outside. Discard `v0`.
`v1 -- v2` is outside. Discard `v1`.
`v2 -- v3` goes outside to inside. Make `v10` at intersection. Discard `v2`.
`v3 -- v4` is inside. Keep these vertices.
⋮
`v7 -- v8` intersects. Make `v11` at intersection.

New polygon is `v10 -- v3 -- v4 -- v5 -- v6 -- v7 -- v11`



Repeat for other three boundary lines.

Four cases moving from one vertex to the next.

1. Completely inside. Add one vertex
2. Crosses inside to outside. Add one vertex.
3. Completely outside. Do nothing.
4. Cross outside to inside. Add two vertices.

10 Thursday 6 September: Math Review

Here I'm going to put things that Dr. Borst talked about that weren't in Jason's excellent talk, plus some thoughts that came to my mind.

10.1 Matrix Notation and Matrix-Matrix Multiplication (MMM)

Let $B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$ and $C = \begin{bmatrix} 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 \end{bmatrix}$

B is a "three by two" matrix, and C is a "two by four"matrix. The first number is the number of rows, and the second the columns.

$B_{i,j}$ is the *element* of B in the i^{th} row and the j^{th} column, so $B_{2,3} = 6$. Note that we're counting the rows and columns starting at 1, not 0.

To multiply the matrices, $A = B \times C$, we're using the dot product. The value of $A_{i,j}$ is the dot product of row i of B and column j of C. Note that this multiplication can't happen if the number of columns of B doesn't match the number of rows of C.

$$A_{2,3} = [3,4] \cdot \begin{bmatrix} 9 \\ 13 \end{bmatrix} = 3 \cdot 9 + 4 \cdot 13 = 27 + 52 = 79$$

$$A = B \times C = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \cdot \begin{bmatrix} 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 \end{bmatrix}$$

$$= \begin{bmatrix} 1 \cdot 7 + 2 \cdot 11 & 1 \cdot 8 + 2 \cdot 12 & 1 \cdot 9 + 2 \cdot 13 & 1 \cdot 10 + 2 \cdot 14 \\ 3 \cdot 7 + 4 \cdot 11 & 3 \cdot 8 + 4 \cdot 12 & 3 \cdot 9 + 4 \cdot 13 & 3 \cdot 10 + 4 \cdot 14 \\ 5 \cdot 7 + 6 \cdot 11 & 5 \cdot 8 + 6 \cdot 12 & 5 \cdot 9 + 6 \cdot 13 & 5 \cdot 10 + 6 \cdot 14 \end{bmatrix} = \begin{bmatrix} 29 & 32 & 35 & 38 \\ 65 & 72 & 79 & 86 \\ 101 & 112 & 123 & 134 \end{bmatrix}$$

In mathic notation, $A_{i,j} = \sum_{k=1}^{2} B_{i,k} \times C_{k,j}$

In `C++`,

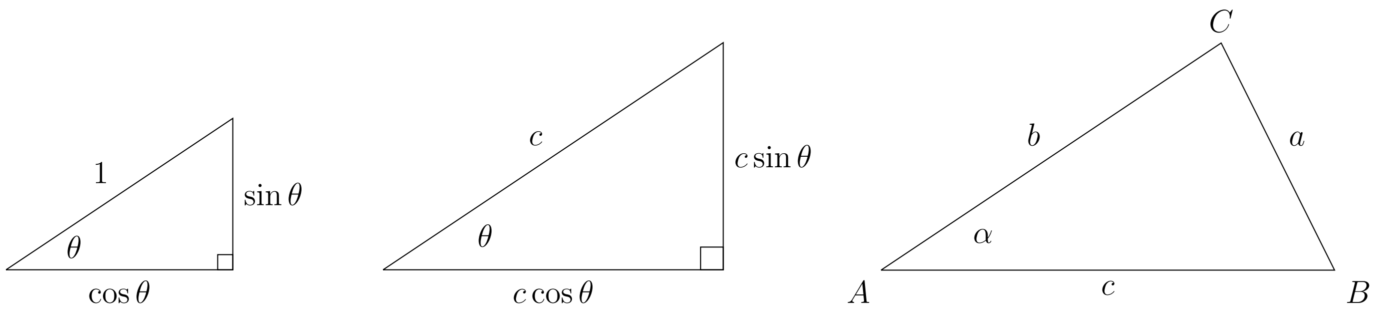
Listing 1: Matrix-Matrix Multiplication

```
int function()
{
    A[3][4] = {};
    B[3][2] = {{1,2},{3,4},{5,6}};
    C[2][3] = {{7,8,9,10},{11,12,13,14}};
    for (i=0; i<3; i++)
    {
        for (j=0; j<4; j++)
        {
            for (k=0; k<2; k++)
            {
                A[i][j] = A[i][j] + B[i][k] * C[k][j];
            }
        }
    }
}
```

10.2 Dot Product: Derivation and as a Projection

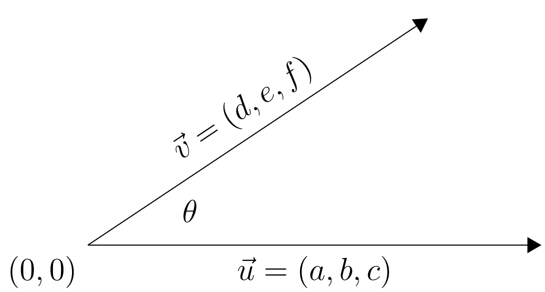
10.2.1 Trig Review

First, a Trig review. The triangle on the right gives a right-triangle definition of sine and cosine. The triangle in the center is a similar triangle adaptation. The triangle on the left is for the Law of Cosines.

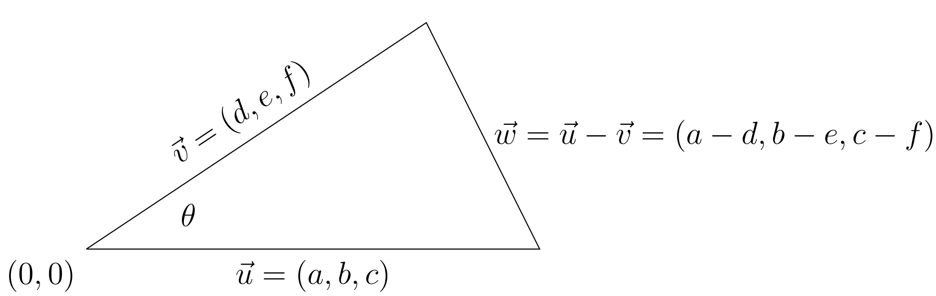


The Law of Cosines says that $a^2 = b^2 + c^2 - 2bc\cos\alpha$.

10.2.2 Derivation of $\vec{u} \cdot \vec{v} = |\vec{u}||\vec{v}|\cos\theta$

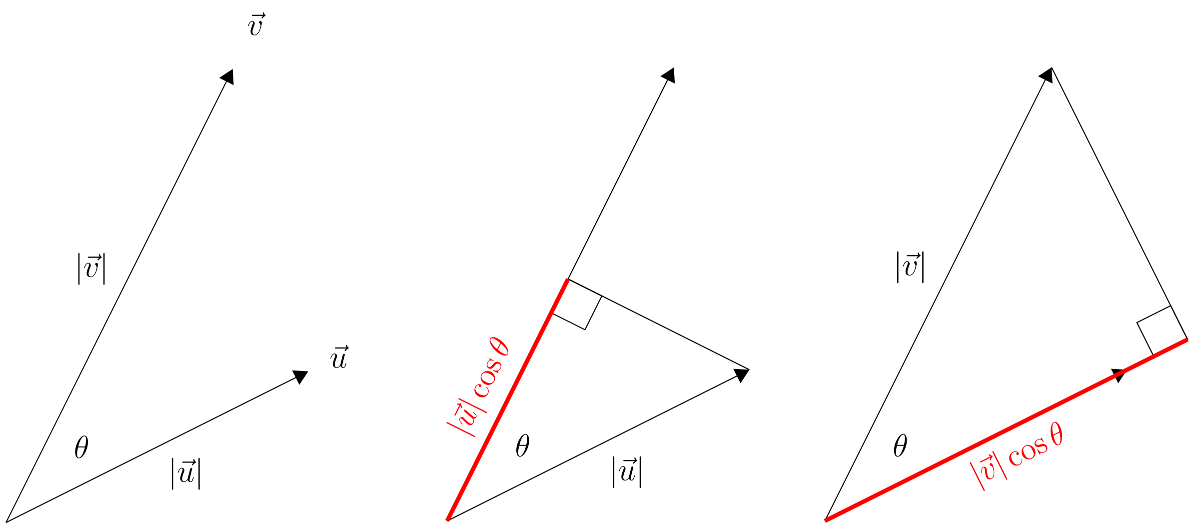


Make it a triangle and apply the Law of Cosines.



$$\begin{aligned}
|\vec{w}|^2 &= |\vec{u}|^2 + |\vec{v}|^2 - 2|\vec{u}||\vec{v}|\cos\theta \\
(a-d)^2 + (b-e)^2 + (c-f)^2 &= (a^2+b^2+c^2) + (d^2+e^2+f^2) - 2|\vec{u}||\vec{v}|\cos\theta \\
a^2 - 2ad + d^2 + b^2 - 2be + e^2 + c^2 - 2cf + f^2 &= (a^2+b^2+c^2) + (d^2+e^2+f^2) - 2|\vec{u}||\vec{v}|\cos\theta \\
-2ad - 2be - 2cf &= -2|\vec{u}||\vec{v}|\cos\theta \\
ad + be + cf &= |\vec{u}||\vec{v}|\cos\theta \\
\vec{u}\cdot\vec{v} &= |\vec{u}||\vec{v}|\cos\theta \qquad Q.E.D.
\end{aligned}$$

10.2.3 Dot product as a projection



The segment of length $|\vec{u}|\cos\theta$ is the *projection* of $\vec{u}$ onto $\vec{v}$.
The segment of length $|\vec{v}|\cos\theta$ is the *projection* of $\vec{v}$ onto $\vec{u}$.

In the middle triangle, you can visualize the dot product, $\vec{u}\cdot\vec{v} = |\vec{u}||\vec{v}|\cos theta$, as the product of the length of the projection onto $\vec{v}$ and the length of $\vec{v}$. Similarly in the left triangle, you can visualize the dot product as the product of the length of the projection onto $\vec{u}$ and the length of $\vec{u}$.

10.3 Cross Product

10.3.1 Finding the Cross Product using the Determinant

The *determinant* of a matrix is a scalar that embodies many mysterious properties of the matrix.

To find the cross product of $\vec{u} = (a, b, c)$ and $\vec{v} = (d, e, f)$, let $\vec{i} = (1, 0, 0)$, $\vec{j} = (0, 1, 0)$, and $\vec{k} = (0, 0, 1)$ be the unit vectors in the x, y and z directions.

$$\vec{u} \times \vec{v} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a & b & c \\ d & e & f \end{vmatrix} = \begin{vmatrix} b & c \\ e & f \end{vmatrix} \vec{i} - \begin{vmatrix} a & c \\ d & f \end{vmatrix} \vec{j} + \begin{vmatrix} a & b \\ d & e \end{vmatrix} \vec{k}$$

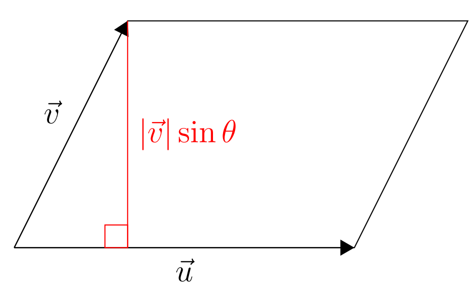
$$= (bf - ce)\vec{i} - (af - cd)\vec{j} + (ae - bd)\vec{k} = \begin{bmatrix} bf - ce \\ cd - af \\ ae - bd \end{bmatrix}$$

The derivation of the identity that the magnitude of the cross product is the product of the magnitudes of the vectors and the sine of the angle between them is to long to fit in the margin of this paper.

I believe it comes from $\cos\theta = \dfrac{\vec{u} \cdot \vec{v}}{|\vec{u}||\vec{v}|}$ and $\sin^2\theta = 1 - \cos^2\theta$.

10.3.2 Cross Product as the Area of the Parallelogram

The area of a parallelogram is base $\times$ height, where the height is perpendicular to the base. The area of this parallelogram is $|\vec{u}||\vec{v}|\sin\theta$, which is the magnitude of the cross product.



11 Monday 11 September: Color Models

11.1 Anti-Aliasing

Oblique lines of the same length have less color intensity because the pixels are more scattered.
Anti-aliasing can be fixed with greyscale.
Grey level intensity proportional to the area covered.
Randomizing subpixels (Sampling error)

11.2 Character Generation

Bitmap fonts v/s outline fonts
Ideal: Start with outline fonts, render into font cache.

11.3 Color Models

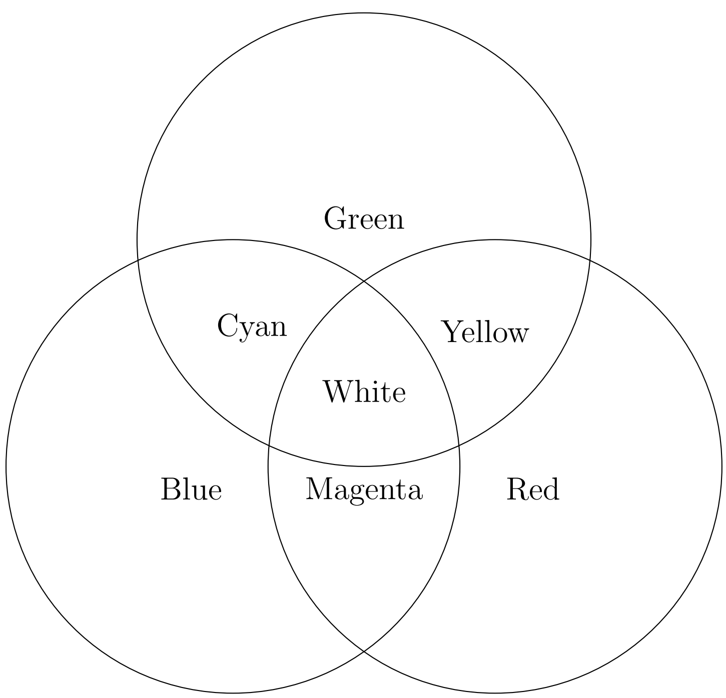
Parts of a color model:

- Hue. Distinguishes b/n colors, relates to "dominant wavelength"
- Saturation. How far a color is from grey. "Excitation purity."
- Light. Perceived intensity, reflected. "Luminescence."

Humans are less sensitive to variations in blue than variations in red or green.

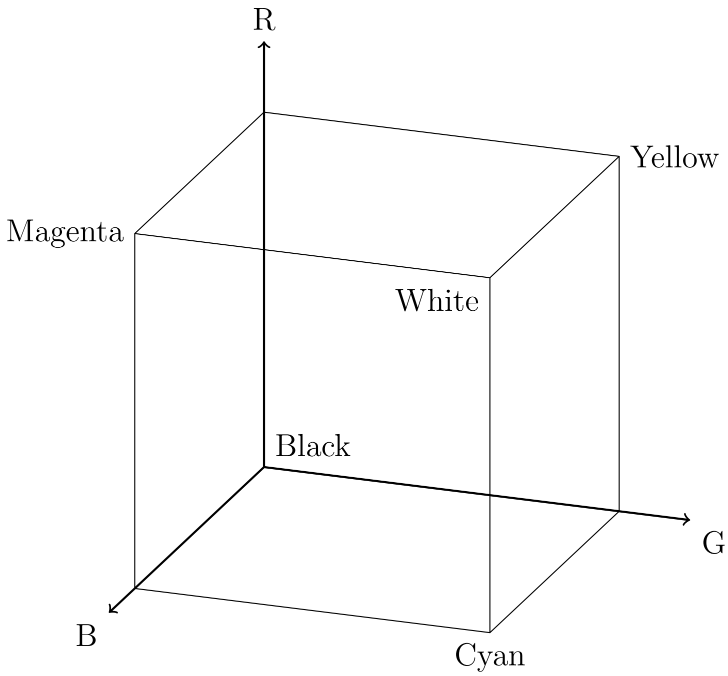
11.3.1 RGB: "Emitting Light"

RGB is device-oriented.



There is no wavelength that gives "magenta." we perceive it as the mixture of two colors.

11.3.2 RGB Cube



11.3.3 CMY: "Absorbing Light."

Printers use CMY (Cyan, Magenta, Yellow)

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

11.3.4 HSV Model

Hue, Saturation, Value
Cone shaped
More artist-oriented.

11.4 Modeling the World

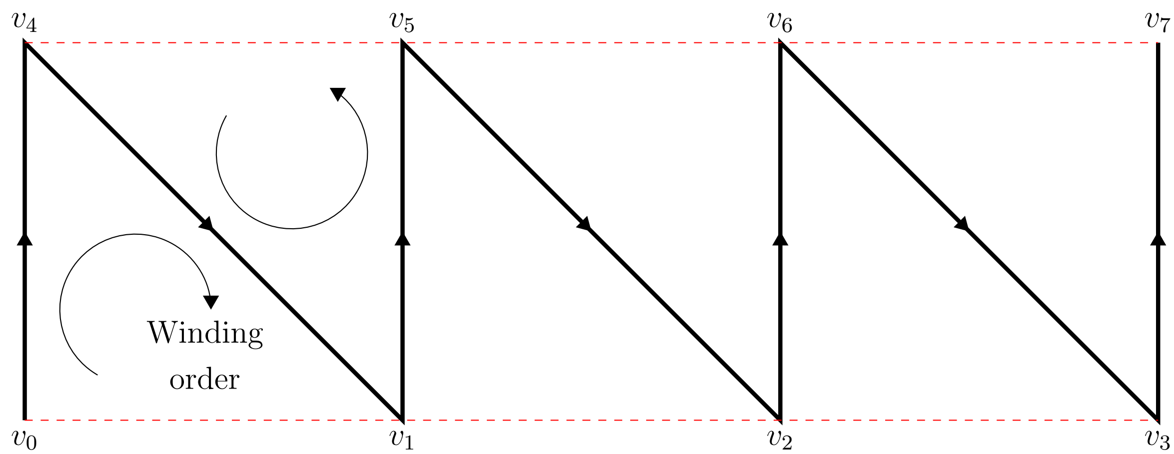
Surface modeling, Materials properties, Scene graph organization
Surface representation techniques:
    Polygon meshes
    Triangle meshes (in this class)
Polygon rendering:
    Consider each polygon

Project onto a 2D viewing screen.

12 Wednesday 13 September: Triangle Mesh

HW #2 Triangle Mesh
Move around based on user clicks.

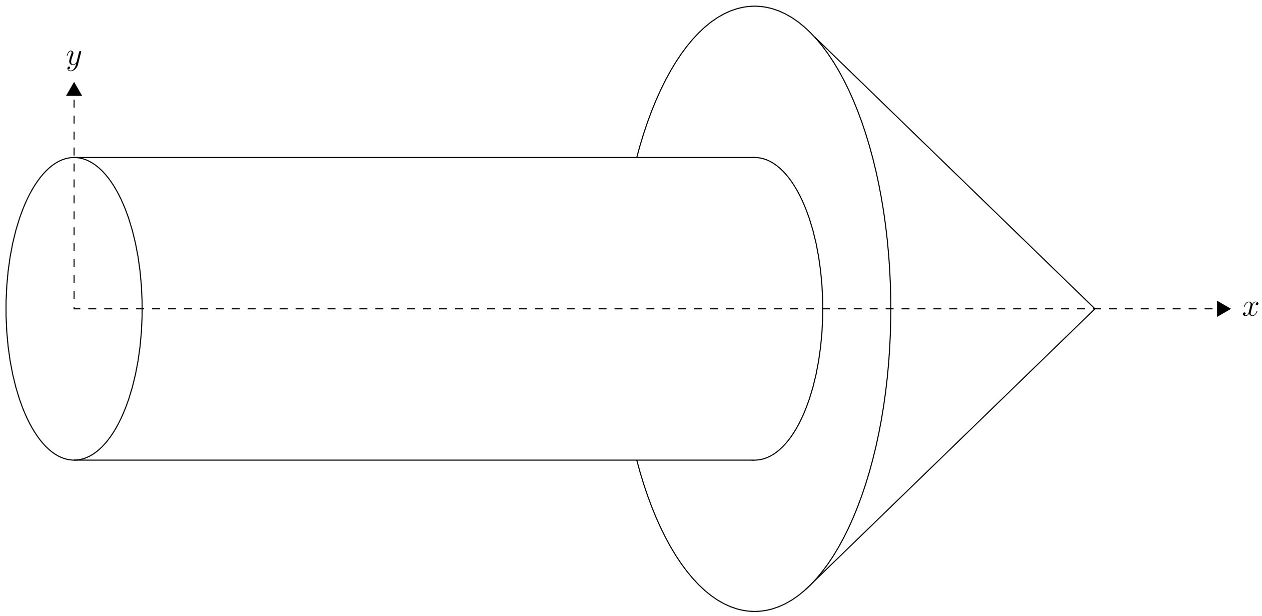
Primitive called OpenGL Triangle Strip

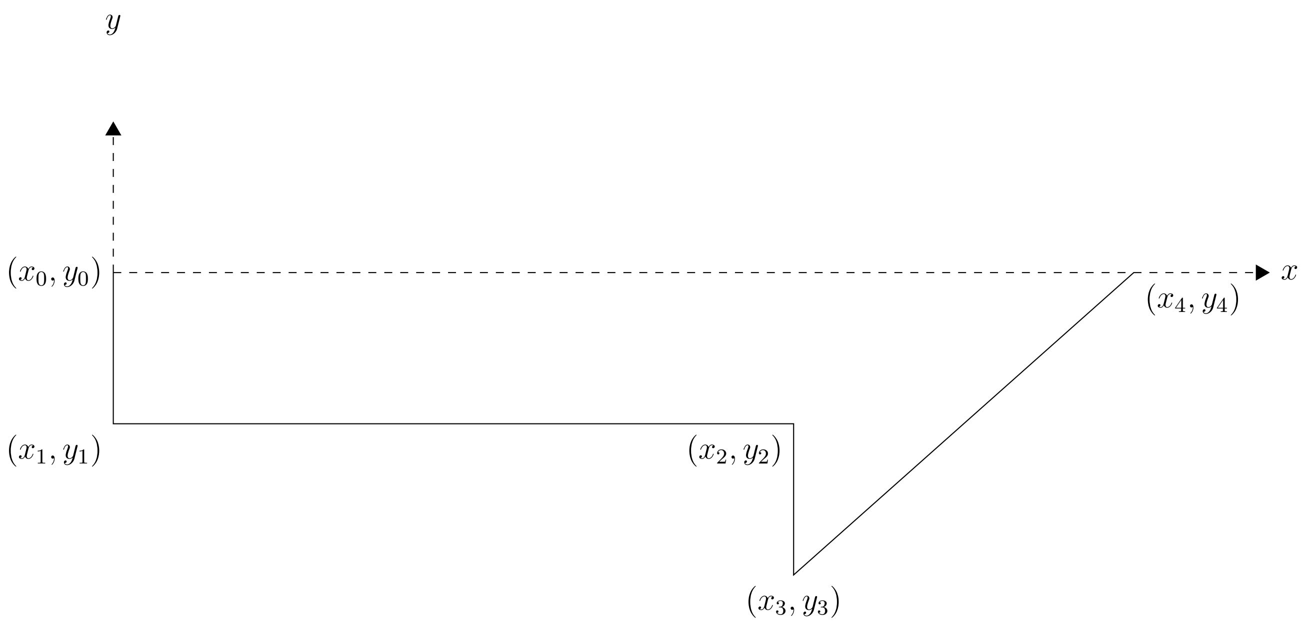


Vertex List Array with vertices or vertex array.
$\{\{x_0, y_0, z_0\}, \{x_1, y_1, z_1\}, \ldots, \{x_7, y_7, z_7\}\}$

Index List Order in which the vertices form triangles.
First three vertices define first triangle, next vertex creates another triangle.
$\{0, 4, 1, 5, 2, 6, 3, 7\}$

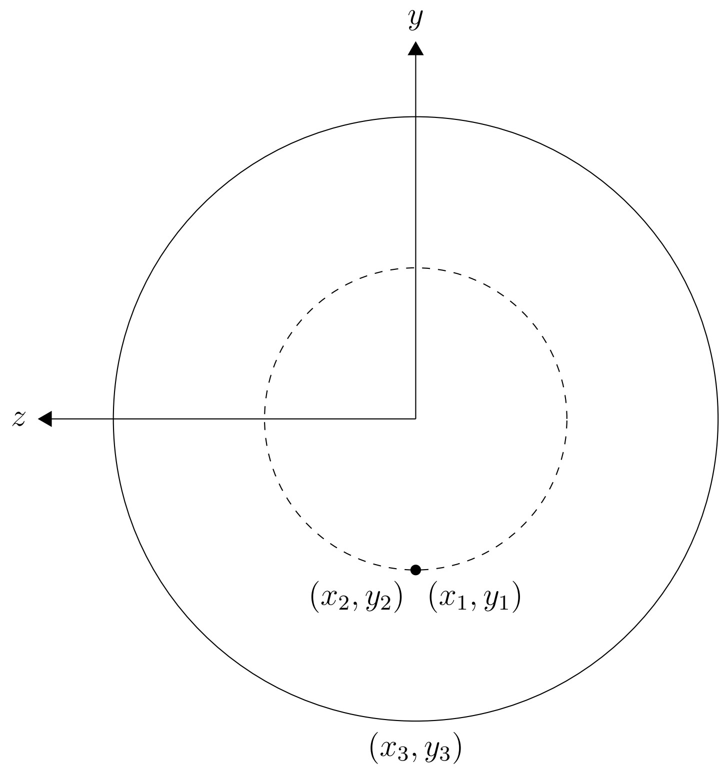
Vertex list gives *geometry*, index list gives *topology* – connectivity with neighbors.





Rotate about x-axis.

Front (tip) view.

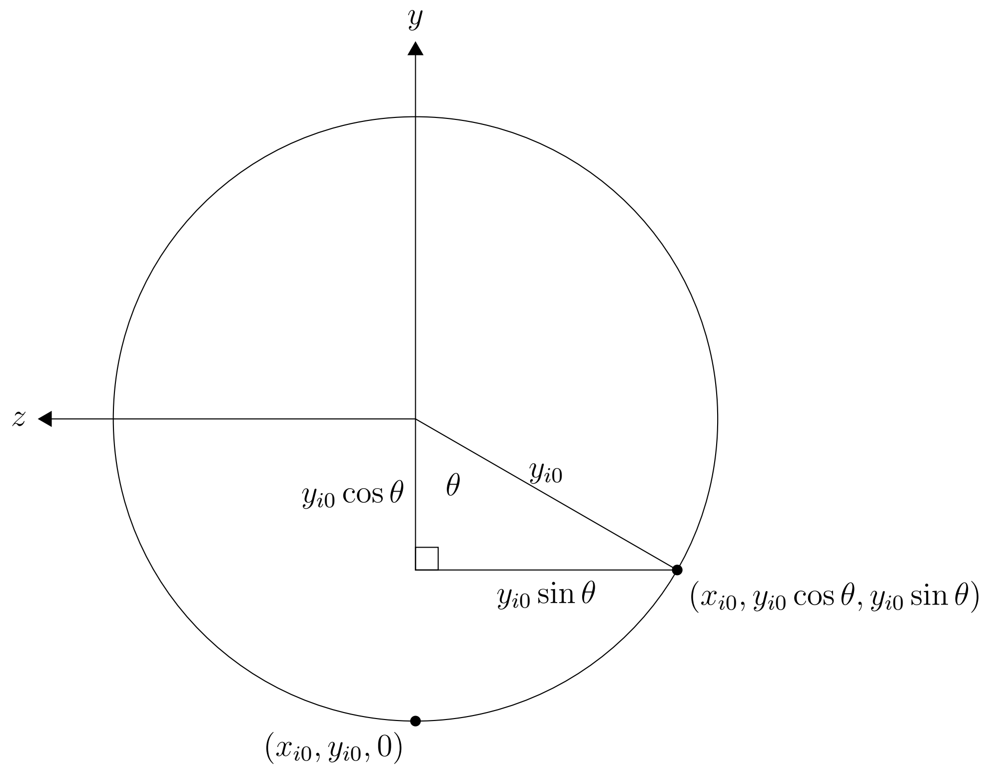


Positive rotation is counter-clockwise when the axis is pointing towards you.

Let np be the number of points to be rotated, and nm be the number of steps of the rotation.

Vertex list:

$$(x_{ij}, y_{ij}, z_{ij}) = (x_{i0}, \cos\theta y_{i0}, \sin\theta y_{i0}) \text{ with } \theta = \frac{2\pi}{n} j \text{ for } 0 \le i \le np \text{ and } 0 \le j \le nm.$$



y
z
$y_{i0}\cos\theta$
θ
y_{i0}
$y_{i0}\sin\theta$
$(x_{i0}, y_{i0}\cos\theta, y_{i0}\sin\theta)$
$(x_{i0}, y_{i0}, 0)$