Intro to OpenGL II

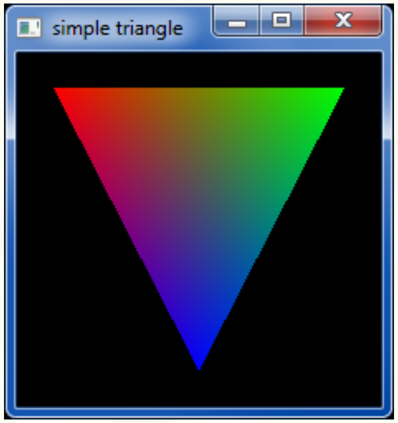
Don Fussell
Computer Science Department
The University of Texas at Austin

Where are we?

- Last lecture, we started the OpenGL pipeline with our example code
- This lecture we'll continue that

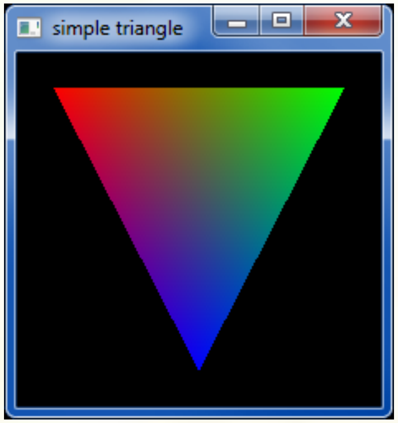
OpenGL API Example

```
glShadeModel(GL_SMOOTH);  // smooth color interpolation
glEnable(GL_DEPTH_TEST);  // enable hidden surface removal

glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glBegin(GL_TRIANGLES);  // every 3 vertexes makes a triangle
  glColor4ub(255, 0, 0, 255);    // RGBA=(1,0,0,100%)
  glVertex3f(-0.8,  0.8,  0.3);  // XYZ=(-8/10,8/10,3/10)

  glColor4ub(0, 255, 0, 255);    // RGBA=(0,1,0,100%)
  glVertex3f( 0.8,  0.8, -0.2);  // XYZ=(8/10,8/10,-2/10)

  glColor4ub(0, 0, 255, 255);    // RGBA=(0,0,1,100%)
  glVertex3f( 0.0, -0.8, -0.2);  // XYZ=(0,-8/10,-2/10)
glEnd();
```



GLUT API Example

```
#include <GL/glut.h>  // includes necessary OpenGL headers

void display() {
  // << insert code on prior slide here >>
  glutSwapBuffers();
}
void main(int argc, char **argv) {
  // request double-buffered color window with depth buffer
  glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
  glutInit(&argc, argv);
  glutCreateWindow("simple triangle");
  glutDisplayFunc(display); // function to render window
  glutMainLoop();
}
```



Simplified Graphics Pipeline

Application

Vertex batching & assembly

OpenGL API

Clipping

NDC to window space

Rasterization

Fragment shading

Depth testing ↔ Depth buffer

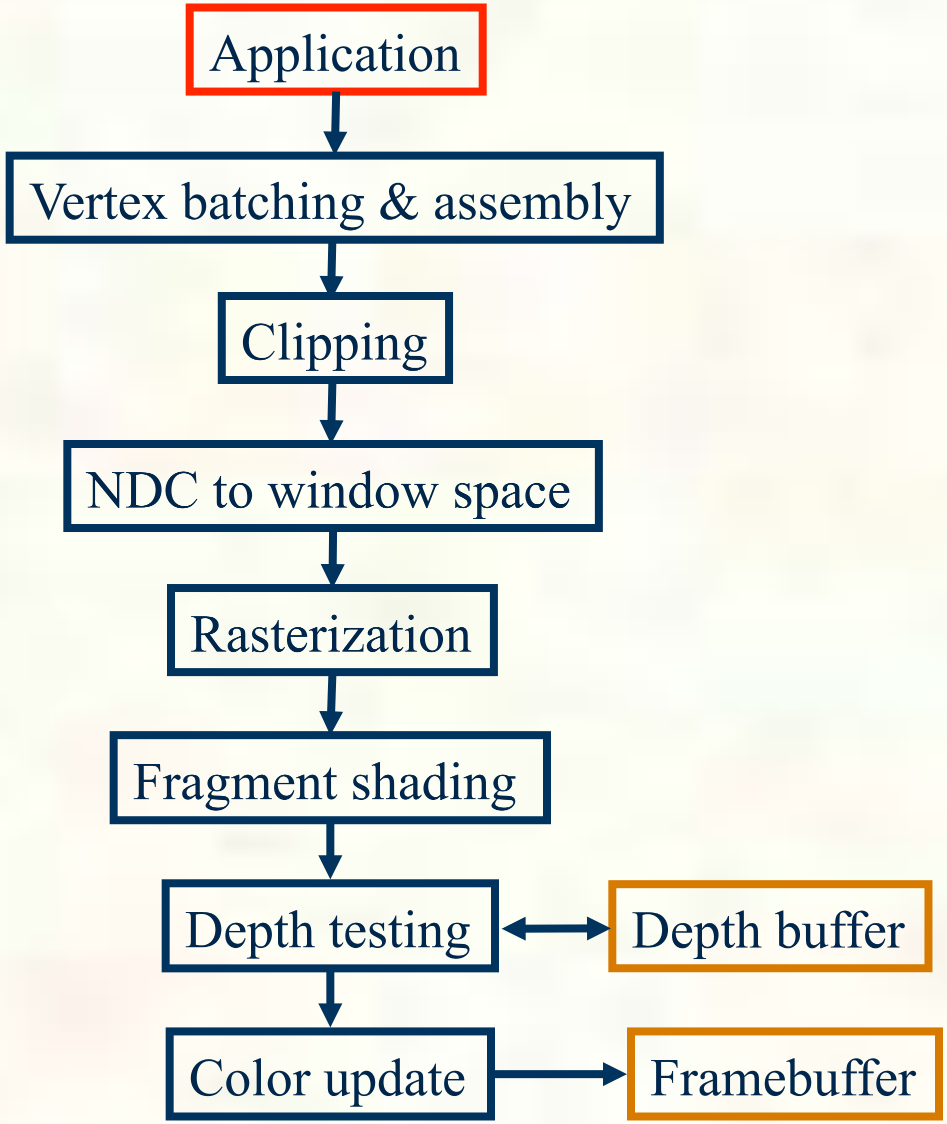
Color update → Framebuffer →

Really lots more steps than this but these are the non-trivial operations in our simple triangle example

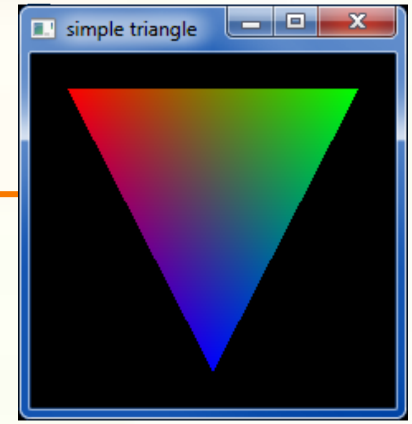
NDC = Normalized Device Coordinates, this is a $[-1,+1]^3$ cube

Application

- What's the app do?
  - Running on the CPU
- Initializes app process
  - Creates graphics resources such as
    - OpenGL context
    - Windows
- Handles events
  - Input events, resize windows, etc.
  - Crucial event for graphics: **Redisplay**
    - Window needs to be drawn—so do it
    - GPU gets involved at this point



App Stuff



- GLUT is doing the heavy lifting
  - Talking to Win32, Cocoa, or Xlib for you
  - Other alternatives: SDL, etc.

```
#include <GL/glut.h>  // includes necessary OpenGL headers

void display() {
  // << insert code on prior slide here >>
  glutSwapBuffers();
}

void main(int argc, char **argv) {
  // request double-buffered color window with depth buffer
  glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
  glutInit(&argc, argv);
  glutCreateWindow("simple triangle");
  glutDisplayFunc(display); // function to render window
  glutMainLoop();
}
```

display function is being registered as a "callback"

Rendering - the *display* Callback

```
glShadeModel(GL_SMOOTH);   // smooth color interpolation
glEnable(GL_DEPTH_TEST);   // enable hidden surface removal
```

Graphics state setting

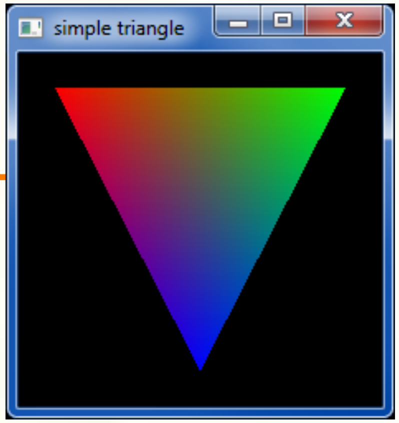
```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

Framebuffer buffer clearing

```
glBegin(GL_TRIANGLES); {   // every 3 vertexes makes a triangle
  glColor4ub(255, 0, 0, 255);     // RGBA=(1,0,0,100%)
  glVertex3f(-0.8,  0.8,  0.3);   // XYZ=(-8/10,8/10,3/10)

  glColor4ub(0, 255, 0, 255);     // RGBA=(0,1,0,100%)
  glVertex3f( 0.8,  0.8, -0.2);   // XYZ=(8/10,8/10,-2/10)

  glColor4ub(0, 0, 255, 255);     // RGBA=(0,0,1,100%)
  glVertex3f( 0.0, -0.8, -0.2);   // XYZ=(0,-8/10,-2/10)
} glEnd();
```

Triangle rendering

Graphics State Setting



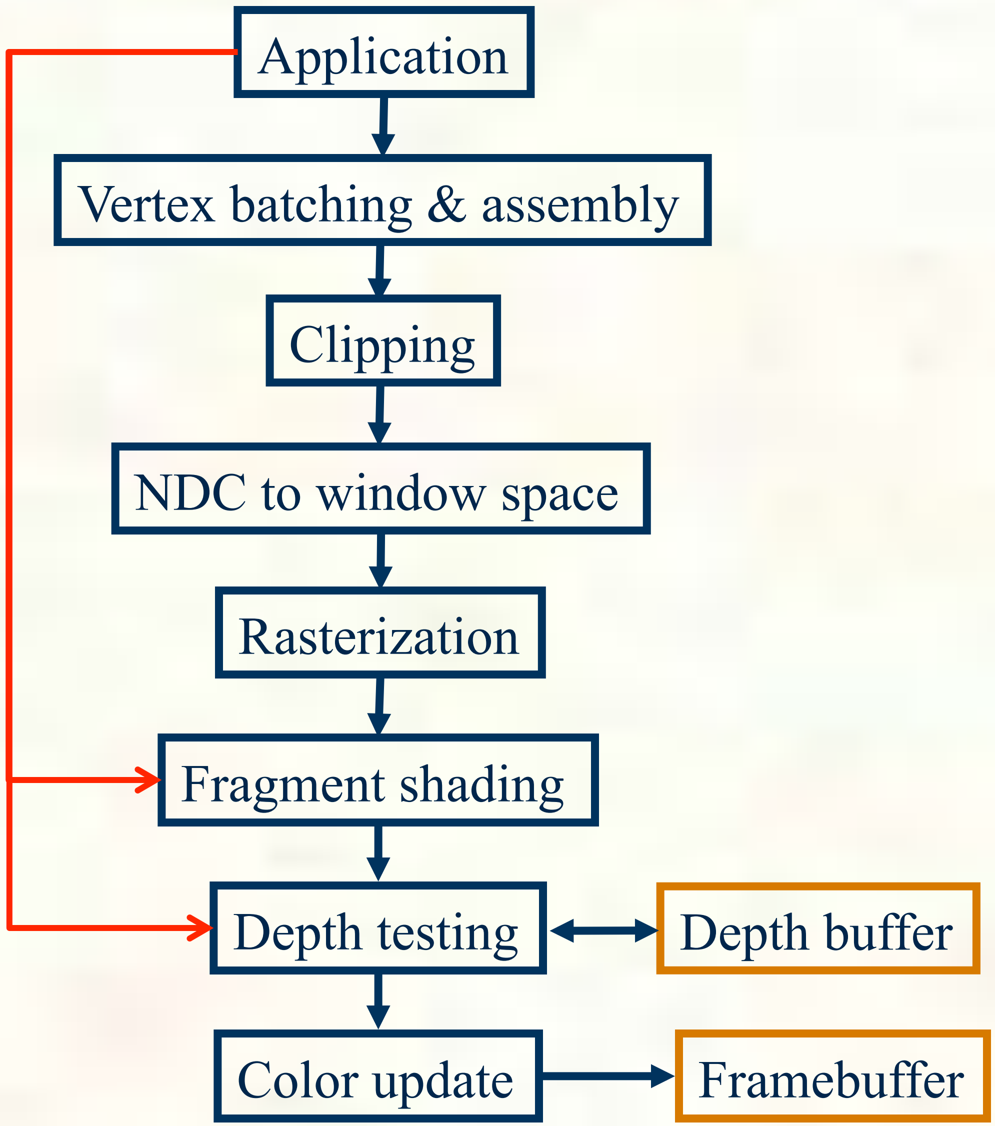
- Within the draw routine

```
glShadeModel(GL_SMOOTH);   // smooth color interpolation
glEnable(GL_DEPTH_TEST);   // enable hidden surface removal

glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glBegin(GL_TRIANGLES); {  // every 3 vertexes makes a triangle
  glColor4ub(255, 0, 0, 255);     // RGBA=(1,0,0,100%)
  glVertex3f(-0.8,  0.8,  0.3);  // XYZ=(-8/10,8/10,3/10)

  glColor4ub(0, 255, 0, 255);     // RGBA=(0,1,0,100%)
  glVertex3f( 0.8,  0.8, -0.2);  // XYZ=(8/10,8/10,-2/10)

  glColor4ub(0, 0, 255, 255);     // RGBA=(0,0,1,100%)
  glVertex3f( 0.0, -0.8, -0.2);  // XYZ=(0,-8/10,-2/10)
} glEnd();
```

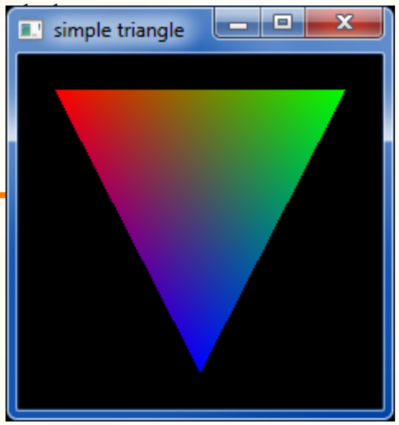
graphics context state is "stateful" (sticky) so technically doesn't need to be done every time display is called

State Updates

- ShadeModel(SMOOTH) requests smooth color interpolation
  - changes fragment shading state
  - alternative is "flat shading"
- Enable(DEPTH_TEST) enables depth buffer-based hidden surface removal algorithm
- State updates happen in command sequence order
- In fact, all OpenGL commands are in a stream that must complete in order



Clearing the buffers

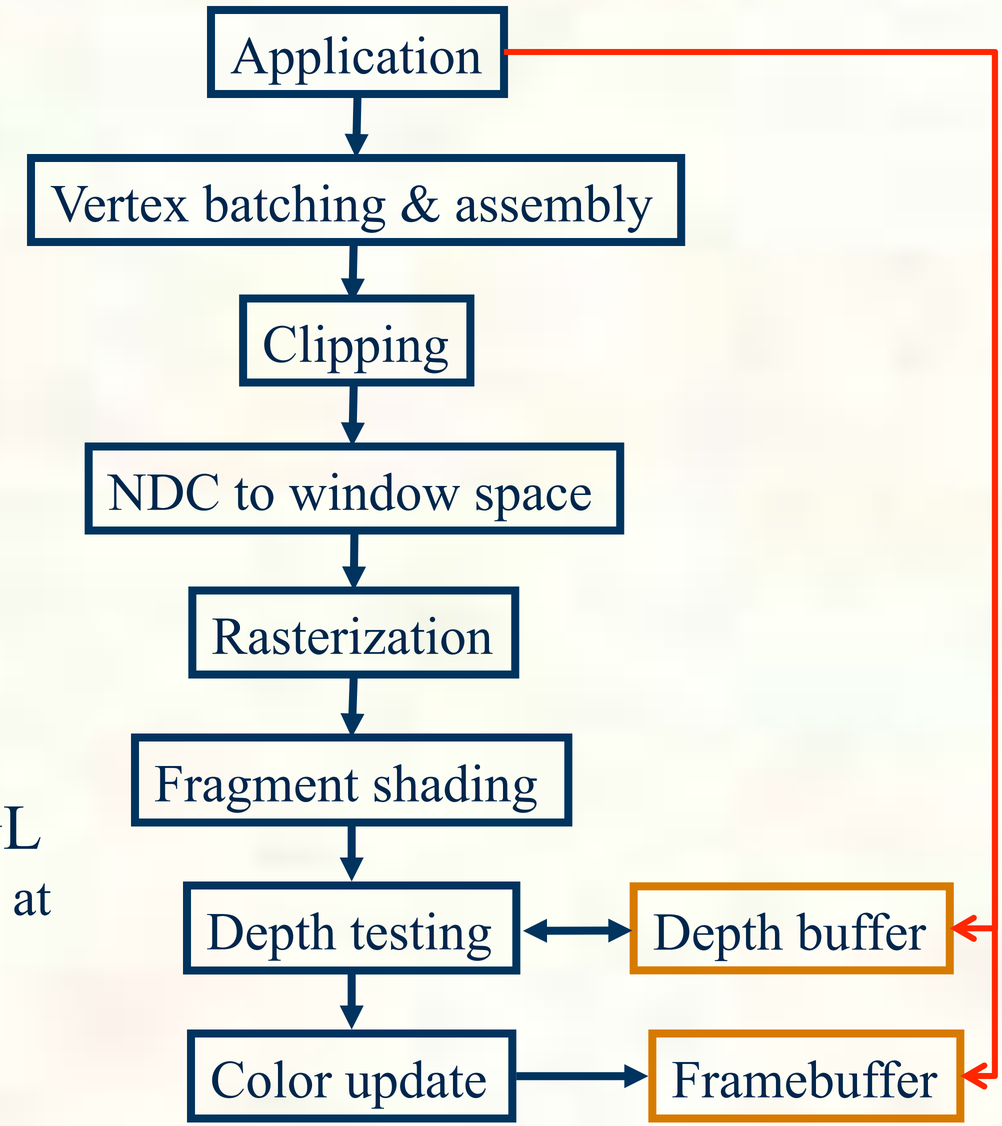


- Within the draw routine

```
glShadeModel(GL_SMOOTH);   // smooth color interpolation
glEnable(GL_DEPTH_TEST);   // enable hidden surface removal

glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glBegin(GL_TRIANGLES);    // every 3 vertexes makes a triangle
  glColor4ub(255, 0, 0, 255);     // RGBA=(1,0,0,100%)
  glVertex3f(-0.8,  0.8,  0.3);  // XYZ=(-8/10,8/10,3/10)

  glColor4ub(0, 255, 0, 255);     // RGBA=(0,1,0,100%)
  glVertex3f( 0.8,  0.8, -0.2);  // XYZ=(8/10,8/10,-2/10)

  glColor4ub(0, 0, 255, 255);     // RGBA=(0,0,1,100%)
  glVertex3f( 0.0, -0.8, -0.2);  // XYZ=(0,-8/10,-2/10)
glEnd();
```

Buffer Clearing

- New frame needs to reset entire color buffer to “background” or “clear” color
  - Avoids having remnants of prior frame persist
    - Needed if can’t guarantee every pixel is touched every frame
- Depth buffer needs to be cleared to “farthest value”
  - More about depth buffering later
- Special operation in OpenGL
  - Hardware wants clears to run at memory-saturating speeds
  - Still in-band with command stream

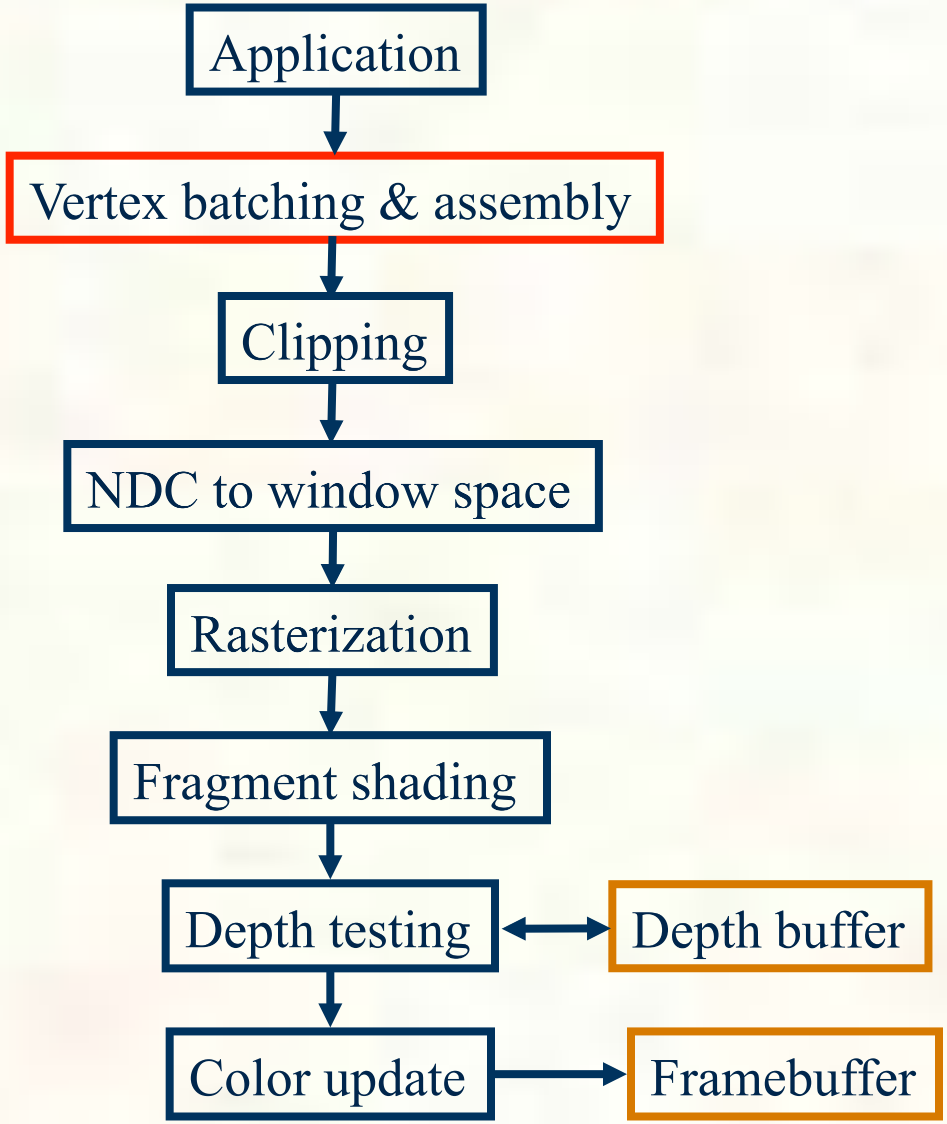


Clear Values and Operations

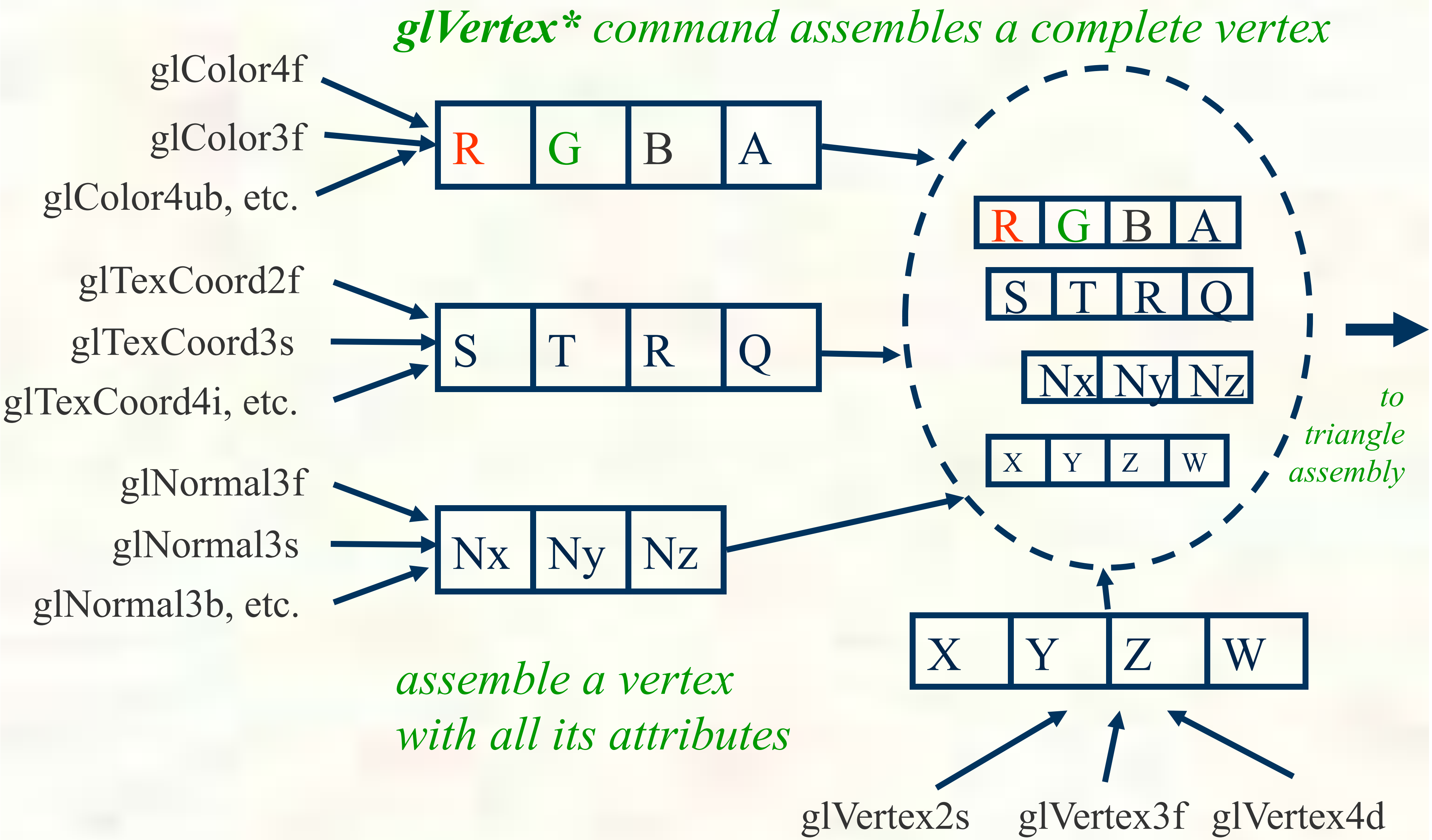
- OpenGL commands to set clear values
  - glClearColor for RGBA color buffers
    - **Example:** glClearColor(0,0,0,1);
      - Clear to black with 100% opacity
      - Initial clear value is (0,0,0,0) so black with 0% opacity
  - glClearDepth for depth buffers
    - **Example**: glClearDepth(1.0);
      - Clear to farthest depth value, for [0,1] range
      - Initial depth clear value is 1.0 so farthest depth value
  - Neither commands does the actual clear operation…
- That's done by glClear(mask)
  - Mask parameter indicates buffers to clear
    - GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT
    - Bitwise-OR (|) them together
    - Also GL_STENCIL_BUFFER_BIT, GL_ACCUM_BUFFER_BIT
  - Allows multiple buffers (e.g. depth <u>&</u> color) to be cleared in single operation, possibly in parallel

Batching and Assembling Vertices

- glBegin and glEnd designate a batch of primitives
  - Begin mode of GL_TRIANGLES means every 3 vertexes = triangle
- Various vertex attributes
  - Position attribute sent with glVertex* commands
  - Also colors, texture coordinates, normals, etc.
- glVertex* assembles a vertex and puts it into the primitive batch
  - Other vertex attribute commands such as glColor* have their attributes "latched" when glVertex* assembles a vertex



Assembling a Vertex



Vertex Attribute Commands

- OpenGL vertex attribute commands follow a regular pattern
  - gl-prefix :: common to all OpenGL API calls
  - Vertex, Normal, TexCoord, Color, SecondaryColor, FogCoord, VertexAttrib, etc.
    - Name the semantic meaning of the attribute
    - VertexAttrib is for generic attributes
      - Used by vertex shaders where the shader determines "meaning" of attributes
      - Attribute zero & Vertex are "special"—they latch the assembly of a vertex
  - 1, 2, 3, 4 :: Number of components for the attribute
    - For an attribute with more components than the number, sensible defaults apply
      - For example, 3 for Color means Red, Green, Blue & Alpha assumed 1.0
  - f, i, s, b, d, ub, us, ui
    - Type of components: float, integer, short, byte, double, unsigned byte, unsigned short, unsigned integer
  - v :: means parameters are passed by a pointer
    - Instead of immediate values

Example

- Consider glColor4ub and glVertex3fv

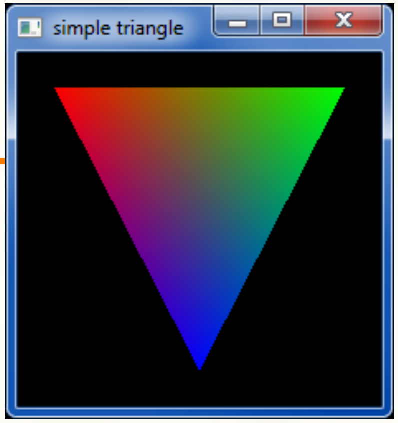
glColor4ub(red, green, blue, alpha);

Belongs to OpenGL *Meaning of attribute* *Number of components* *Type of components*

glVertex3fv(const GLfloat v[3]);

Vector arguments

Assemble a Triangle



- Within the draw routine

```
glBegin(GL_TRIANGLES);

  glColor4ub(255, 0, 0, 255);
  glVertex3f(-0.8,  0.8,  0.3);

  glColor4ub(0, 255, 0, 255);
  glVertex3f( 0.8,  0.8, -0.2);

  glColor4ub(0, 0, 255, 255);
  glVertex3f( 0.0, -0.8, -0.2);

glEnd();
```

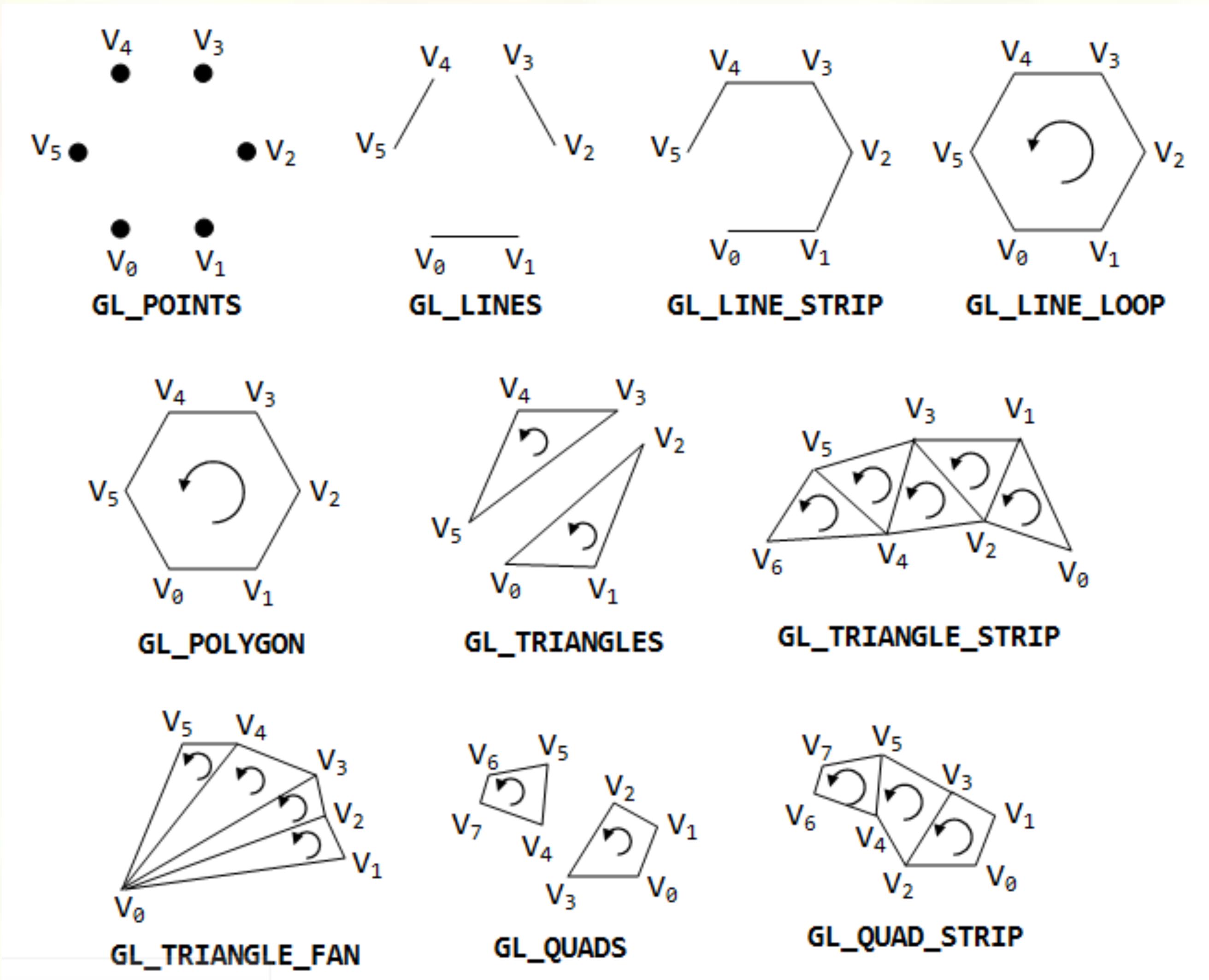
First vertex

Second vertex

Third vertex

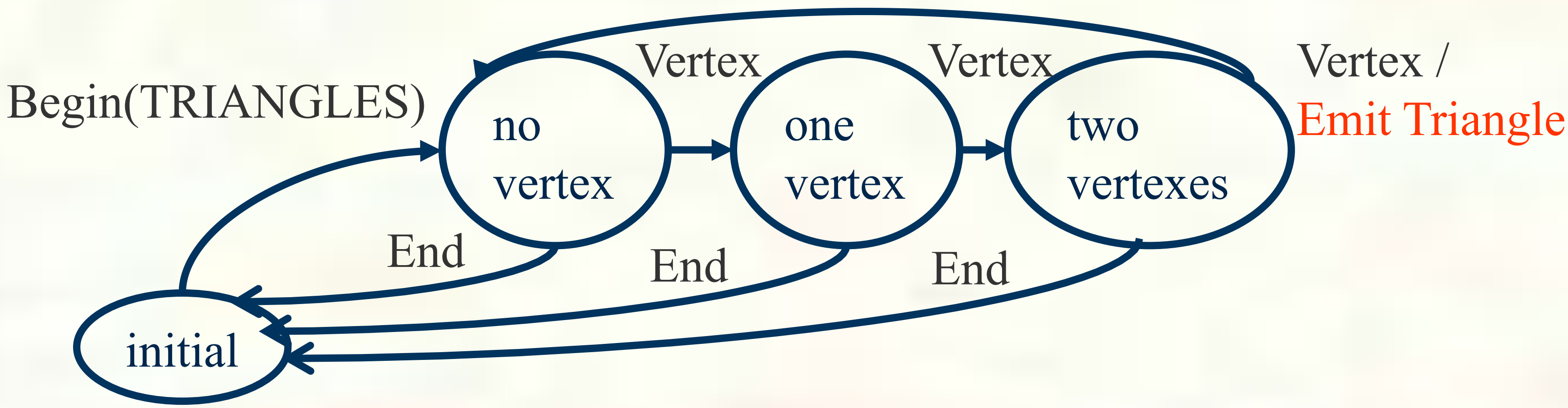
First triangle

glBegin Primitive Batch Types

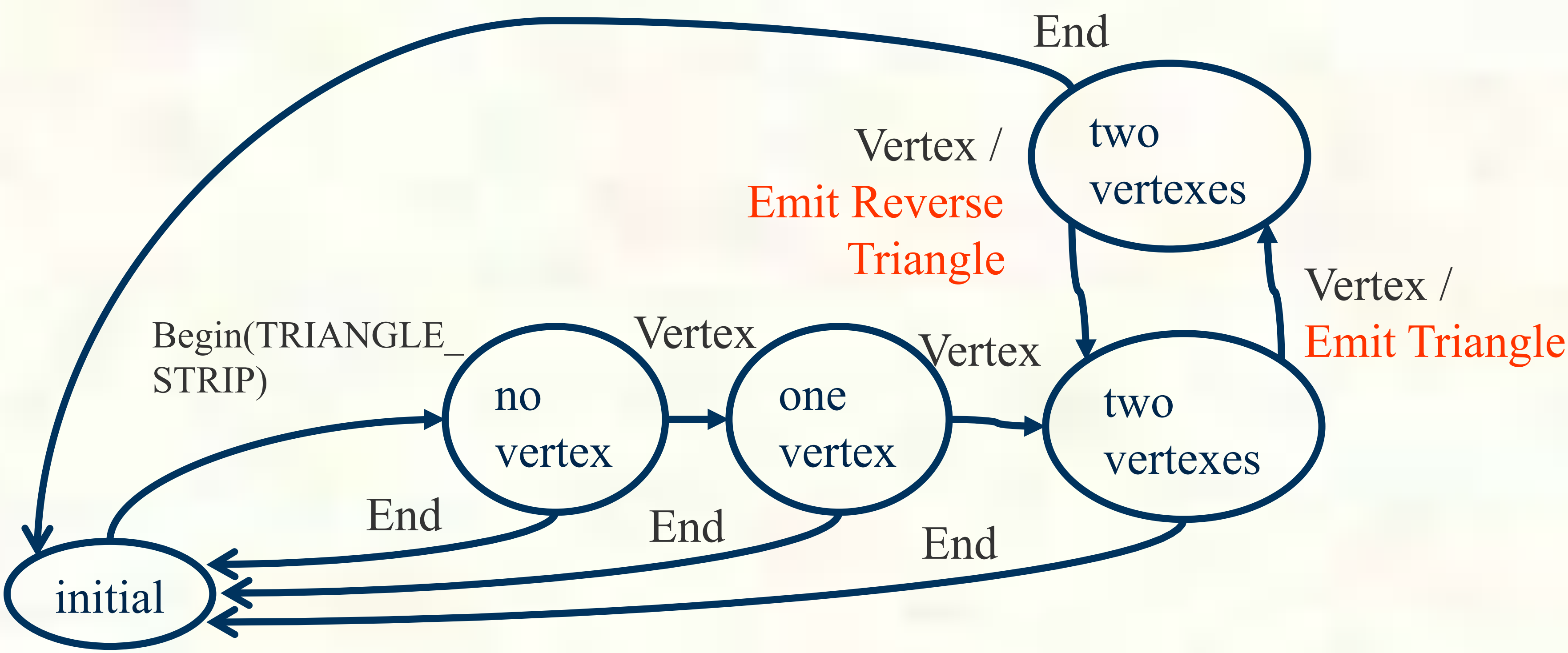


Assembly State Machines

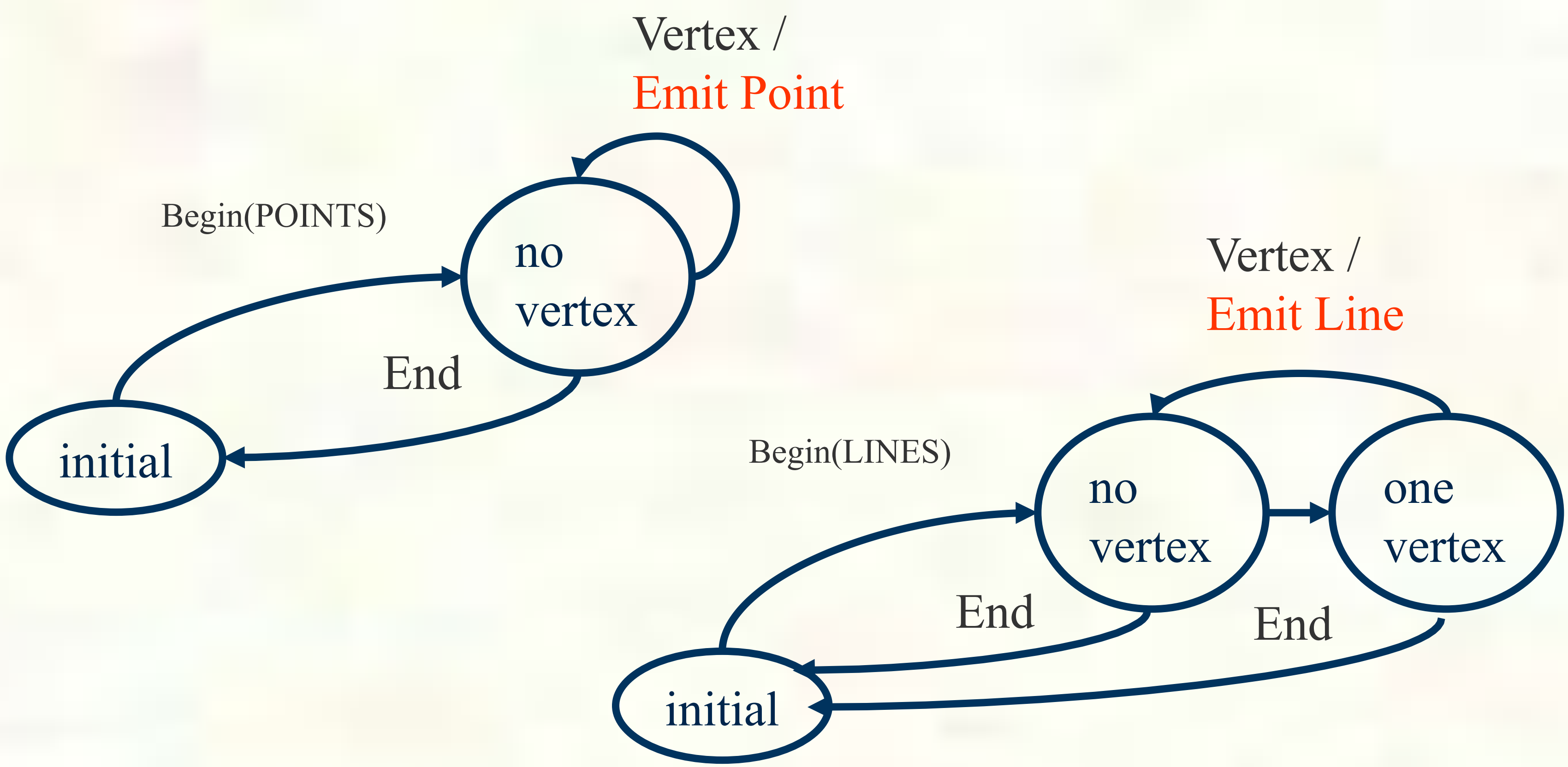
- Fixed-function hardware performs primitive assembly
  - Based on glBegin's mode
- State machine for GL_TRIANGLES



GL_TRIANGLE_STRIP



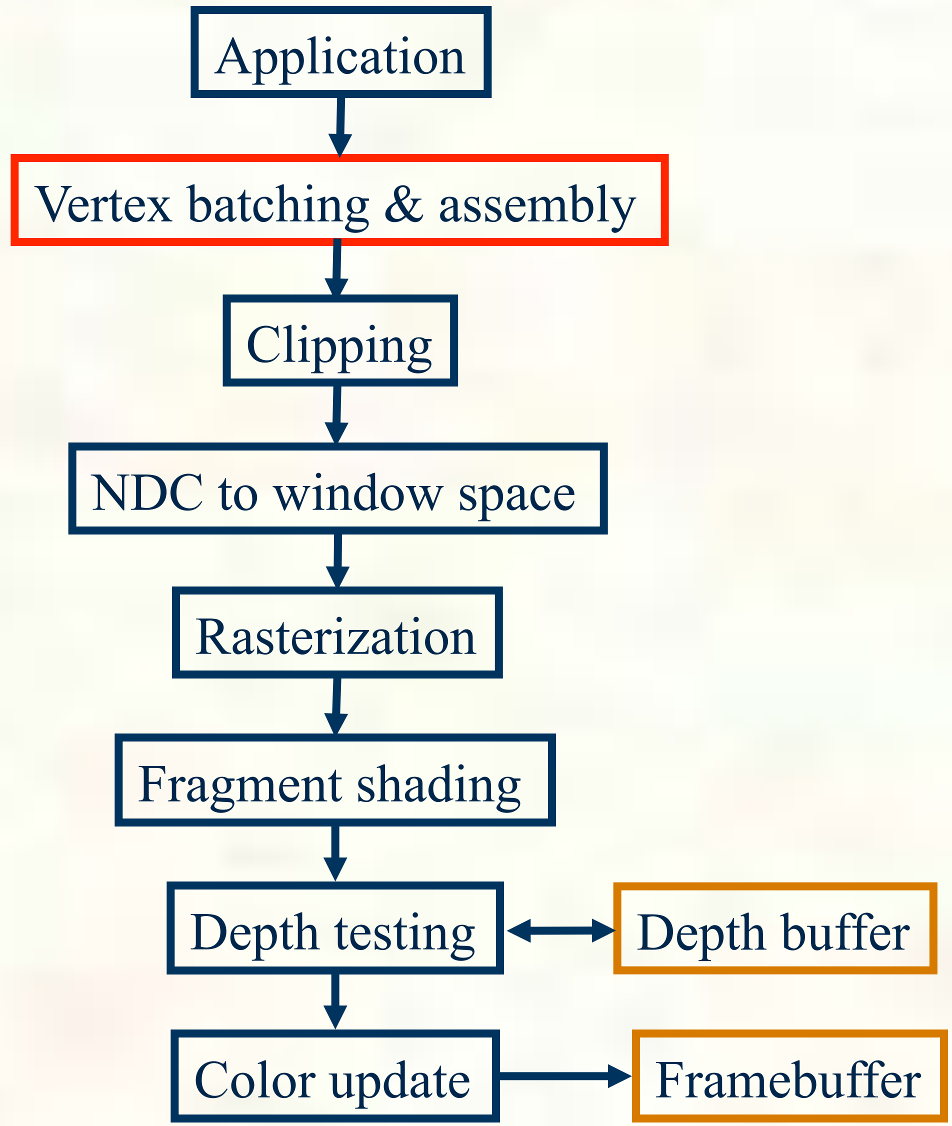
GL_POINTS and GL_LINES



Actual hardware state machine handles all OpenGL begin modes, so rather complex

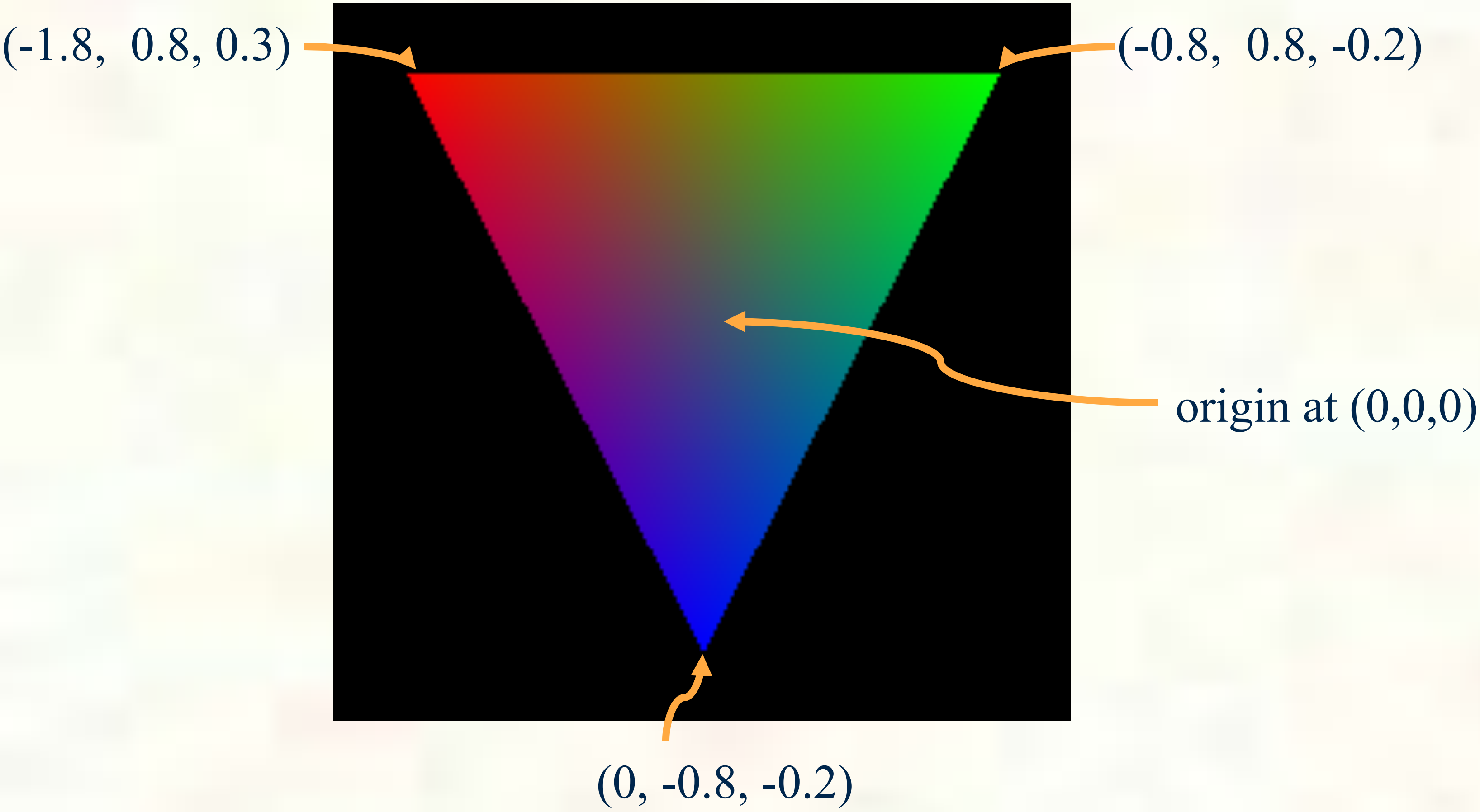
Triangle Assembly

- Now we have a triangle assembled
- Later, we'll generalize how the vertex positions get transformed
  - And other attributes might be processed too
- For now, just assume the XYZ position passed to glVertex3f position is in NDC space



Our Newly Assembled Triangle

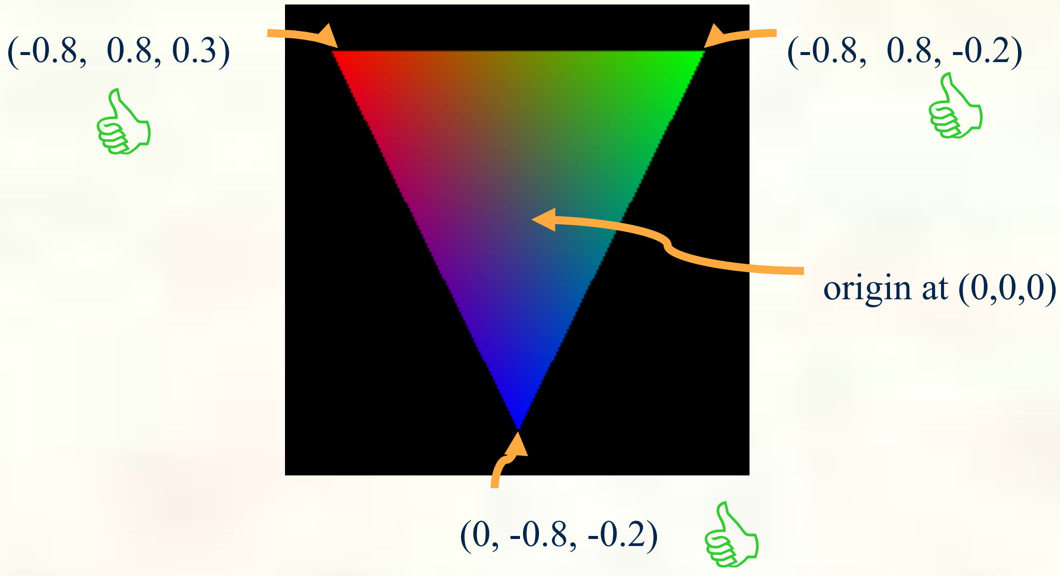
- Think of drawing into a $[-1,+1]^3$ cube



Clipping

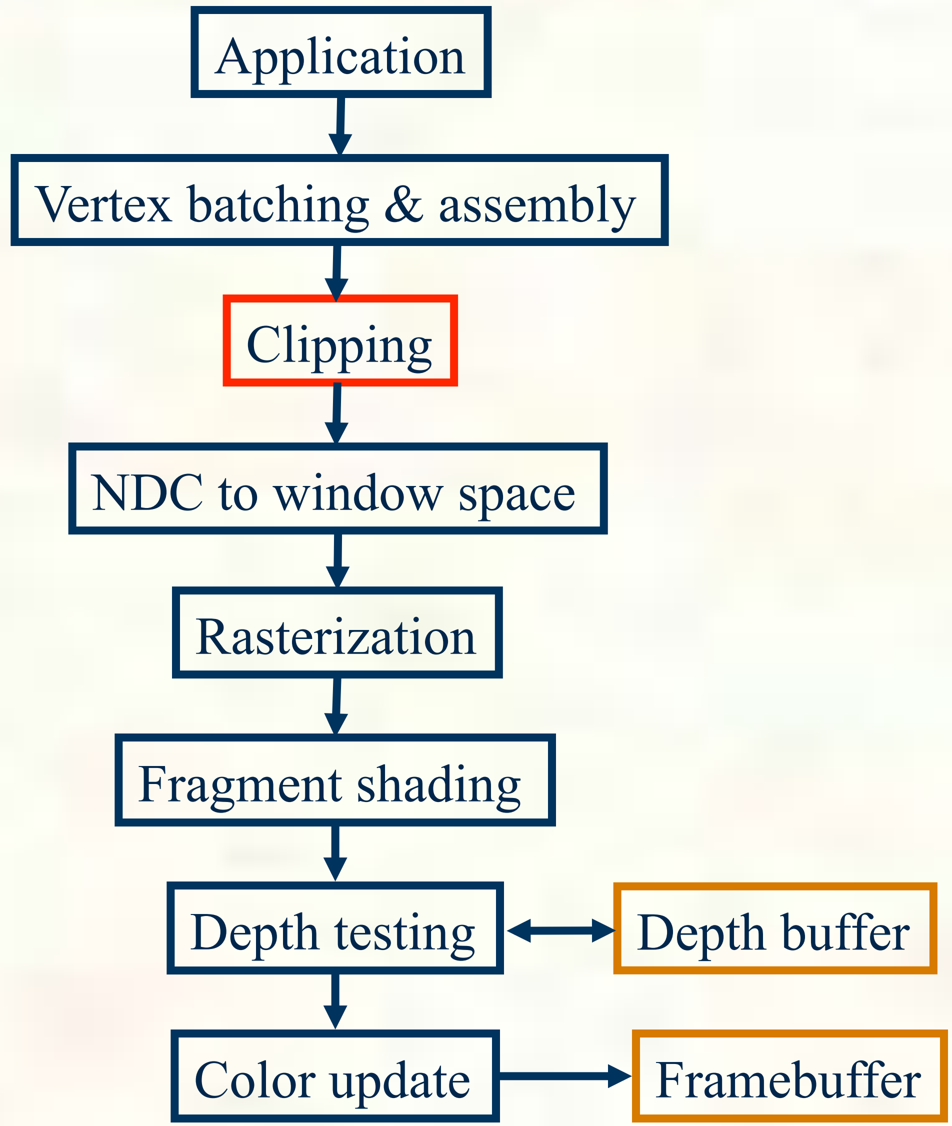
- What if any portion of our triangle extended beyond the NDC range of the $[-1,+1]^3$ cube?
  - Only regions of the triangle $[-1,+1]^3$ cube should be rasterized!
- No clipping for our simple triangle
  - This situation is known as "trivial accept"
  - Because all 3 vertices in the $[-1,+1]^3$ cube

Triangles are convex, so entire triangle must also be in the cube if the vertexes are



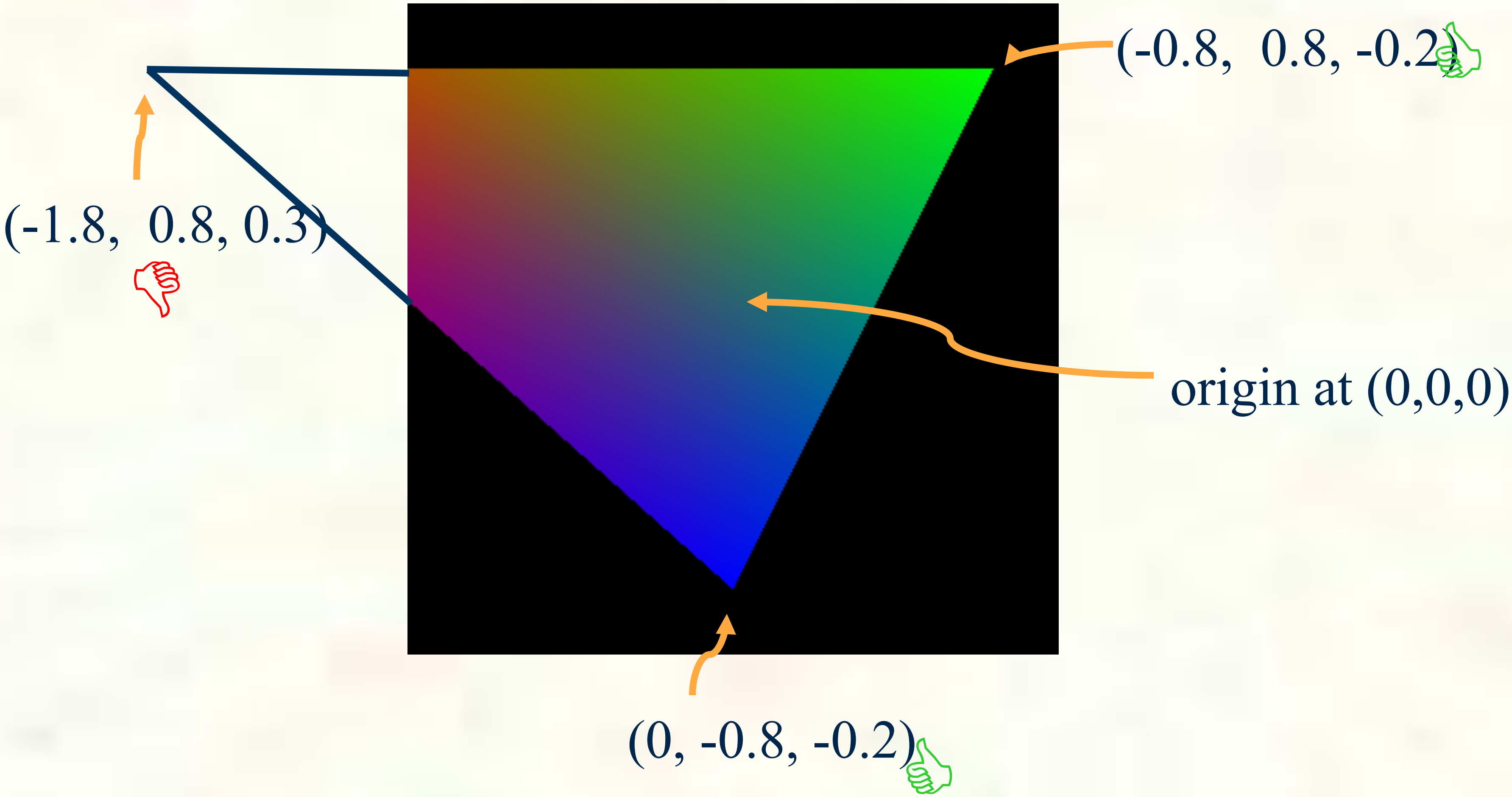
Triangle Clipping

- Triangles can straddle the NDC cube
  - Happens with lines too
- In this case, we must “clip” the triangle to the NDC cube
  - This is an involved process but one that must be done

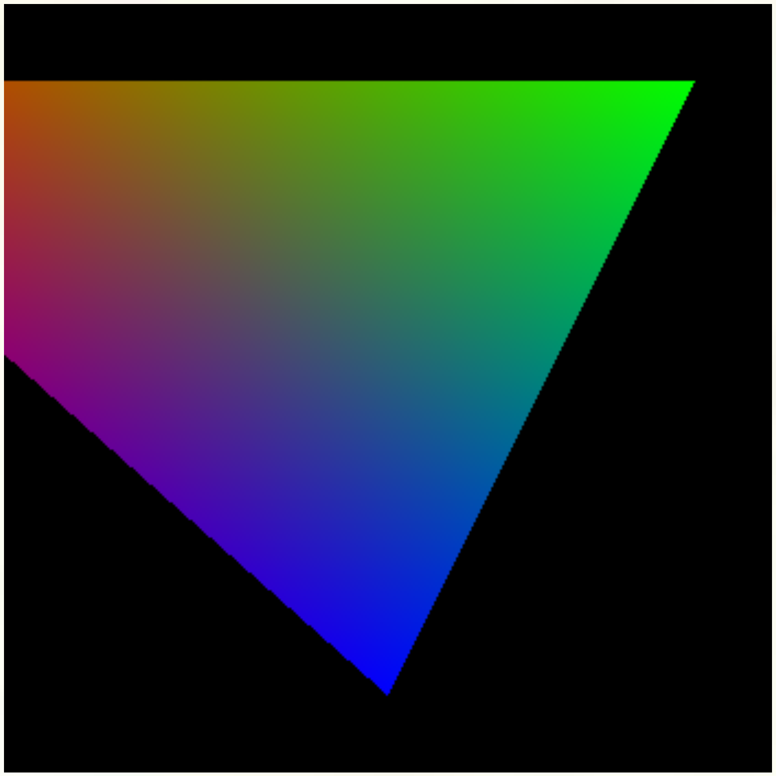


Consider a Different Triangle

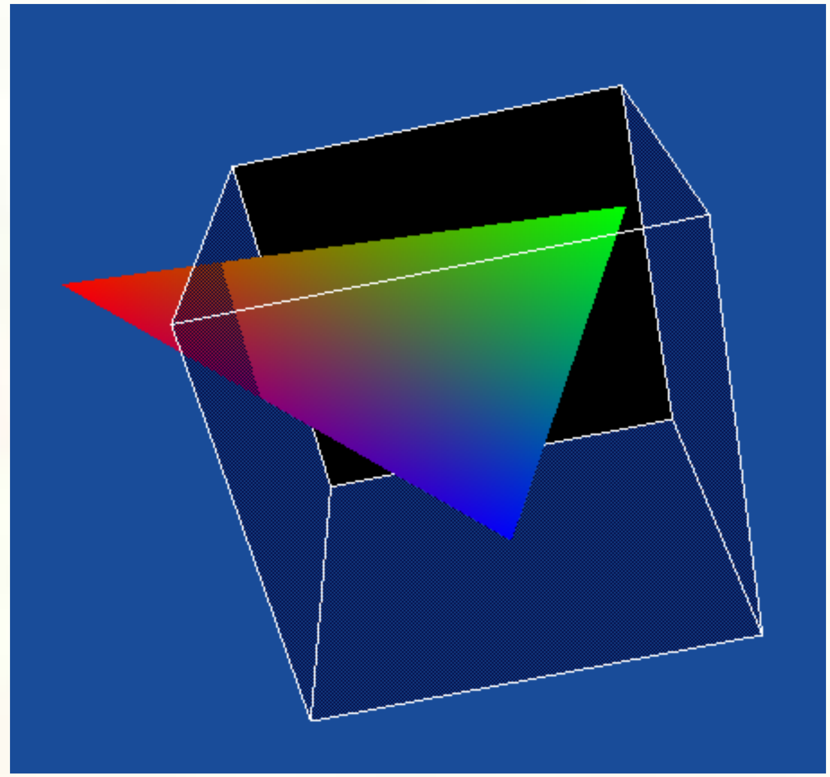
- Move left vertex so it's X = -1.8
  - Result is a <u>clipped</u> triangle



Clipped Triangle Visualized



Clipped and Rasterized Normally



Visualization of NDC space

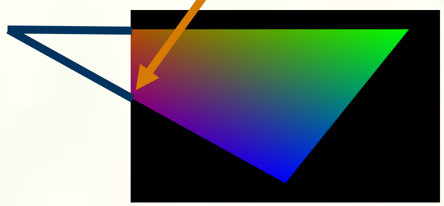
Notice triangle is "poking out" of the cube; this is the reason that should be clipped

Clipping Complications

- Given primitive may be clipped by multiple cube faces
  - Potentially clipping by all 6 faces!
- Approach
  - Four possibilities
    - Face doesn’t actually result in any clipping of a triangle
      - Triangle is unaffected by this plane then
    - Clipping eliminates a triangle completely
      - All 3 vertices on “wrong” side of the face’s plane
    - Triangle “tip” clipped away
      - Leaving two triangles
    - Triangle “base” is clipped away
      - Leaving a single triangle
  - **Strategy**: implement <u>recursive</u> clipping process
    - “Two triangle” case means resulting two triangles must be clipped by all remaining planes

Attribute Interpolation

- When splitting triangles for clipping, must also interpolate new attributes
  - For example, color
  - Also texture coordinates
- Back to our example
  - BLUE×0.8/1.8 + RED×1/1.8
    - (0,0,1,1)×0.8/1.8 + (1,0,0,1)×1/1.8
    - (0.444,0,.555,1) or MAGENTA

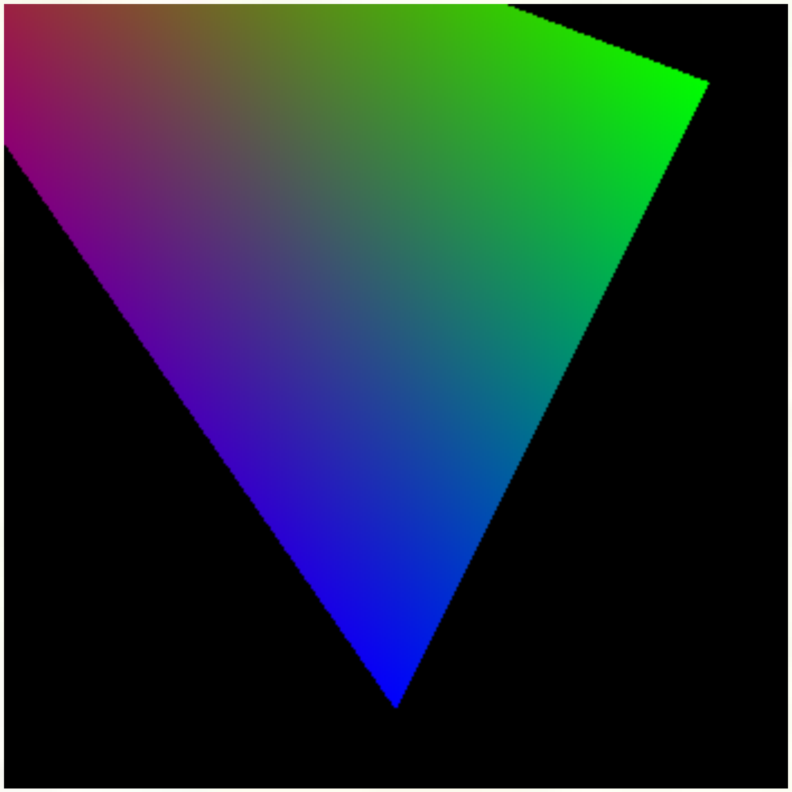


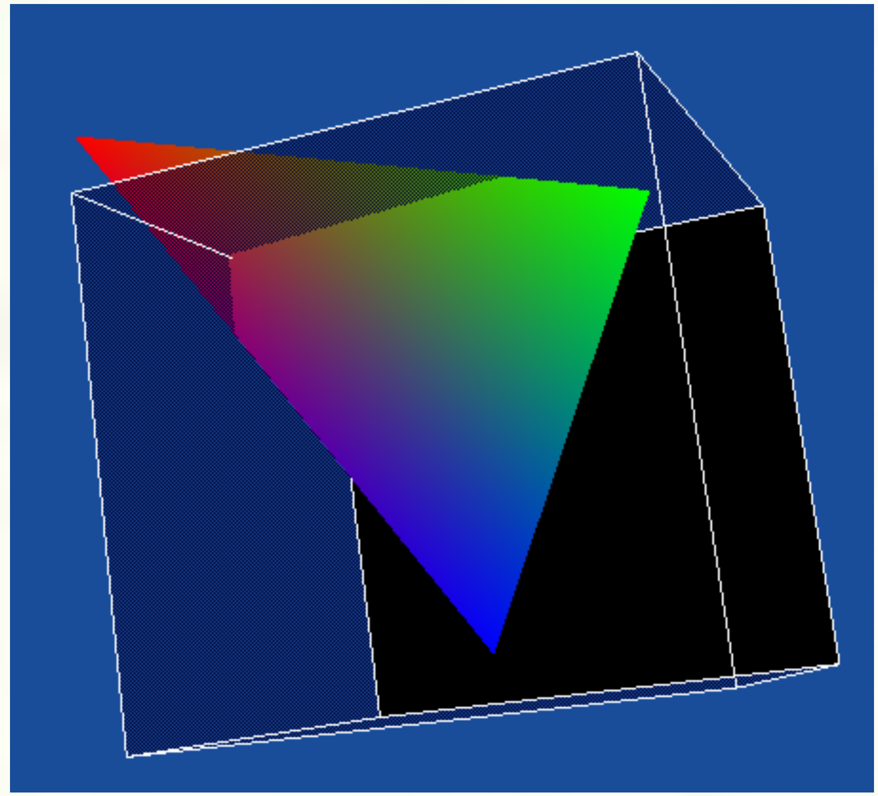
Weights:
1/1.8
0.8/1.8, sum to 1

What to do about this?

- Several possibilities
  - Require applications to never send primitives that require clipping
    - Wishful thinking
    - And a cop-out—makes clipping their problem
  - Rasterize into larger space than normal and discard pixels outsize the NDC cube
    - Increases useless rasterizer work
      - Requires additional math precision in the rasterizer
    - Worse, creates problems when rendering into a projective clip space (needed for perspective)
      - Something for a future lecture
  - Break clipped triangles into smaller triangles that tessellate the clipped region…

Triangle clipped by Two Planes

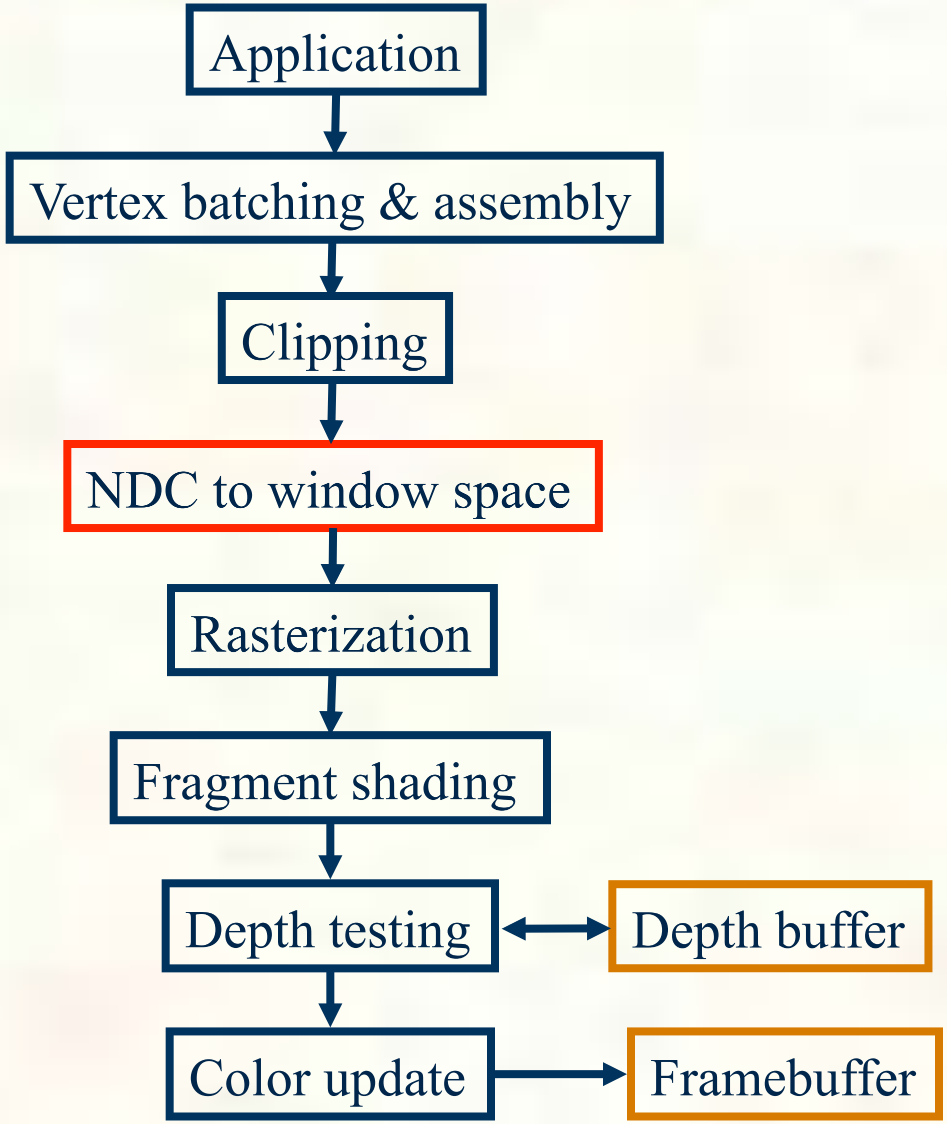




Recursive process can make 4 triangles
And it gets worse with more non-trivial clipping

NDC to Window Space

- NDC is "normalized" to the $[-1,+1]^3$ cube
  - Nice for clipping
  - But doesn't yet map to pixels on the screen
- **Next:** a transform from NDC space to window space



Viewport and Depth Range

- OpenGL has 2 commands to configure the state to map NDC space to window space
  - glViewport(GLint vx, GLint vy, GLsizei w, GLsizei h);
    - Typically programmed to the window's width and height for *w* & *h* and zero for both *vx* & *vy*
    - **Example:** glViewport(0, 0, window_width, window_height);
  - glDepthRange(GLclampd n, GLclampd f);
    - *n* for near depth value, *f* for far depth value
    - Normally set to glDepthRange(0,1)
      - Which is an OpenGL context's initial depth range state
- The mapping from NDC space to window space depends on *vx*, *vy*, *w*, *h*, *n*, and *d*

OpenGL Data Type Naming

- The OpenGL specification allow an implementation to specify how language data types map to OpenGL API data types
  - GLfloat is usually typedef'ed to float but this isn't necessarily true
    - Same for GLint, GLshort, GLdouble
    - But is true in practice
  - GLbyte is byte-sized so expected it to be a char
  - GLubyte, GLushort, and GLuint are unsigned versions of GLbyte, GLshort, and GLint
- Certain names clue you into their parameter usage
  - GLsizei is an integer parameter that is not allowed to be negative
    - An GL_INVALID_VALUE is generated if a GLsizei parameter is ever negative
  - GLclampd and GLclampf are the same as GLfloat and GLdouble, but indicate the parameter will be clamped automatically to the [0,1] range
- Notice
  - glViewport uses GLsizei for width and height
  - glDepthRange uses GLclampd for near and far

OpenGL Errors

- OpenGL reports asynchronously from your commands
  - Effectively, you must explicitly call glGetError to find if any prior command generated an error or was otherwise used incorrectly
  - glGetError returns GL_NO_ERROR if there is no error
    - Otherwise an error such as GL_INVALID_VALUE is returned
- Rationale
  - OpenGL commands are meant to be executed in a pipeline so the error might not be identified until after the command's function has returned
    - Errors might be detected by hardware that isn't actually the CPU
  - Also forcing applications to check return codes of functions is slow
    - It's inappropriate for a high-performance API such as OpenGL
- So if you suspect errors, you have to poll for them
  - Learn to do this while you are debugging your code
  - If something fails to happen, suspect there's an OpenGL errors
- Also commands that generated an error are ignored
  - The only exception is GL_OUT_OF_MEMORY which results in undefined state

Mapping NDC to Window Space

- Assume (x,y,z) is the NDC coordinate that's passed to glVertex3f in our simple_triangle example
- Then window-space (w_x, w_y, w_z) location is
  - $w_x = (w/2) \times x + v_x + w/2$
  - $w_y = (h/2) \times y + v_y + h/2$
  - $w_z = [(f-n)/2] \times z + (n+f)/2$

× means scalar multiplication here

Where is glViewport set?

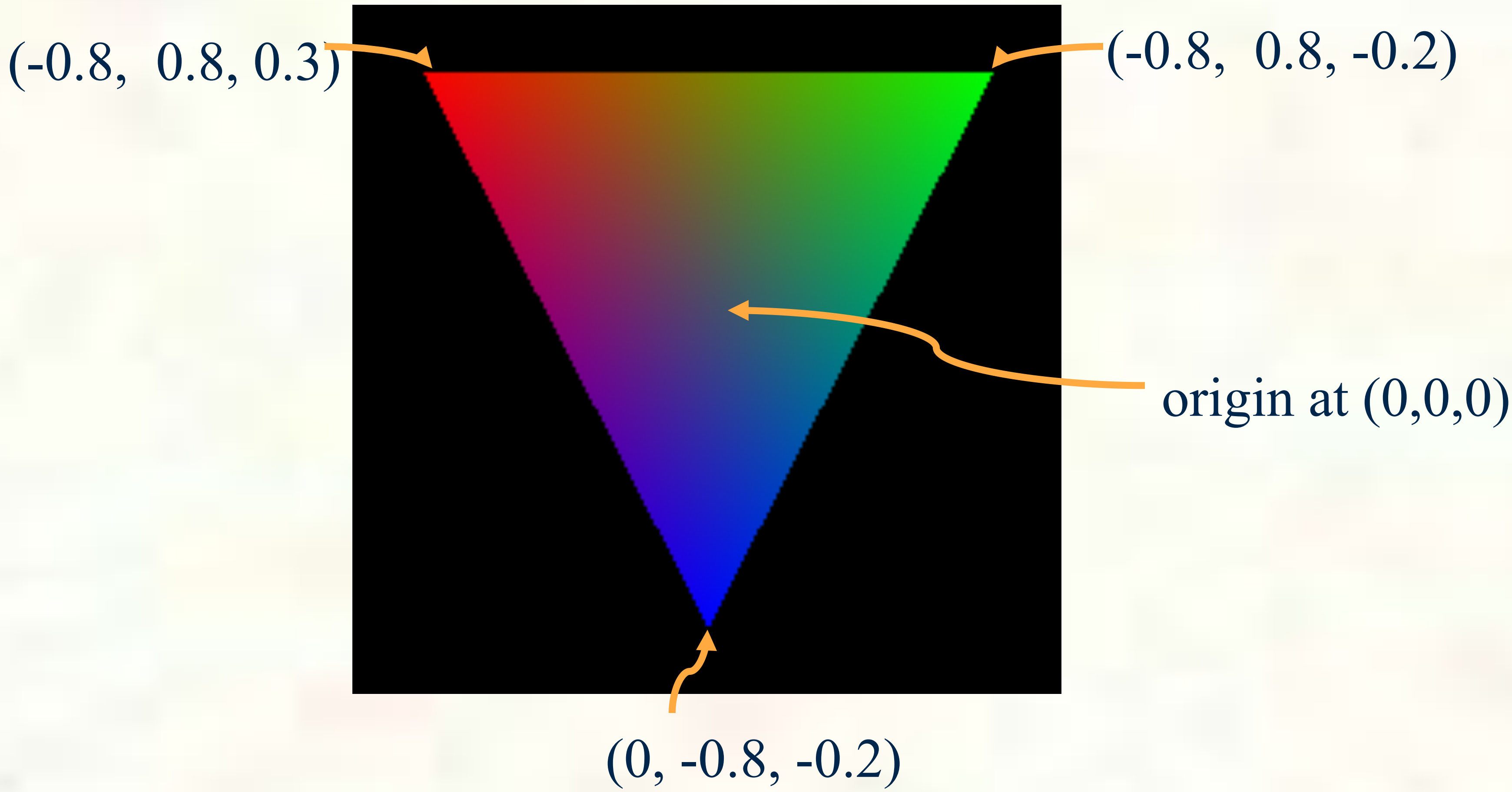
- The simple_triangle program never calls glViewport
  - That's OK because GLUT will call glViewport for you if you don't register your own per-window callback to handle when a window is reshaped (resized)
  - Without a reshape callback registered, GLUT will simply call glViewport(0, 0, window_width, window_height);
- Alternatively, you can use glReshapeFunc to register a callback
  - Then calling glViewport or otherwise tracking the window height becomes your application's responsibility
  - Example reshape callback:
    ```
    void reshape(int w, int h) {
      glViewport(0, 0, w, h);
    }
    ```
  - Example registering a reshape callback:
    glReshapeFunc(reshape);
- **FYI**: OpenGL maintains a lower-left window-space origin
  - Whereas most 2D graphics APIs use upper-left

What about glDepthRange?

- Simple applications don't normally need to call glDepthRange
  - Notice the simple_triangle program never calls glDepthRange
- Rationale
  - The initial depth range of [0,1] is fine for most application
  - It says the entire available depth buffer range should be used
- When the depth range is [0,1] the equation for window-space z simplifies to $wz = \frac{1}{2} \times z + \frac{1}{2}$

Triangle Vertices in Window Space

- Assume the window is 500x500 pixels
  - So glViewport(0,0,500,500) has been called

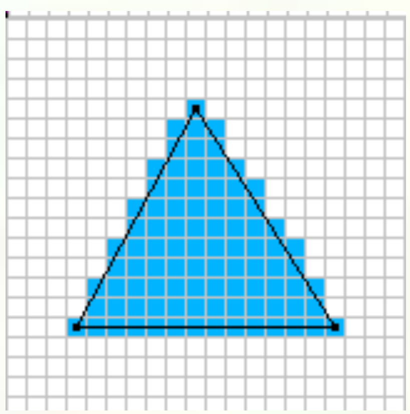


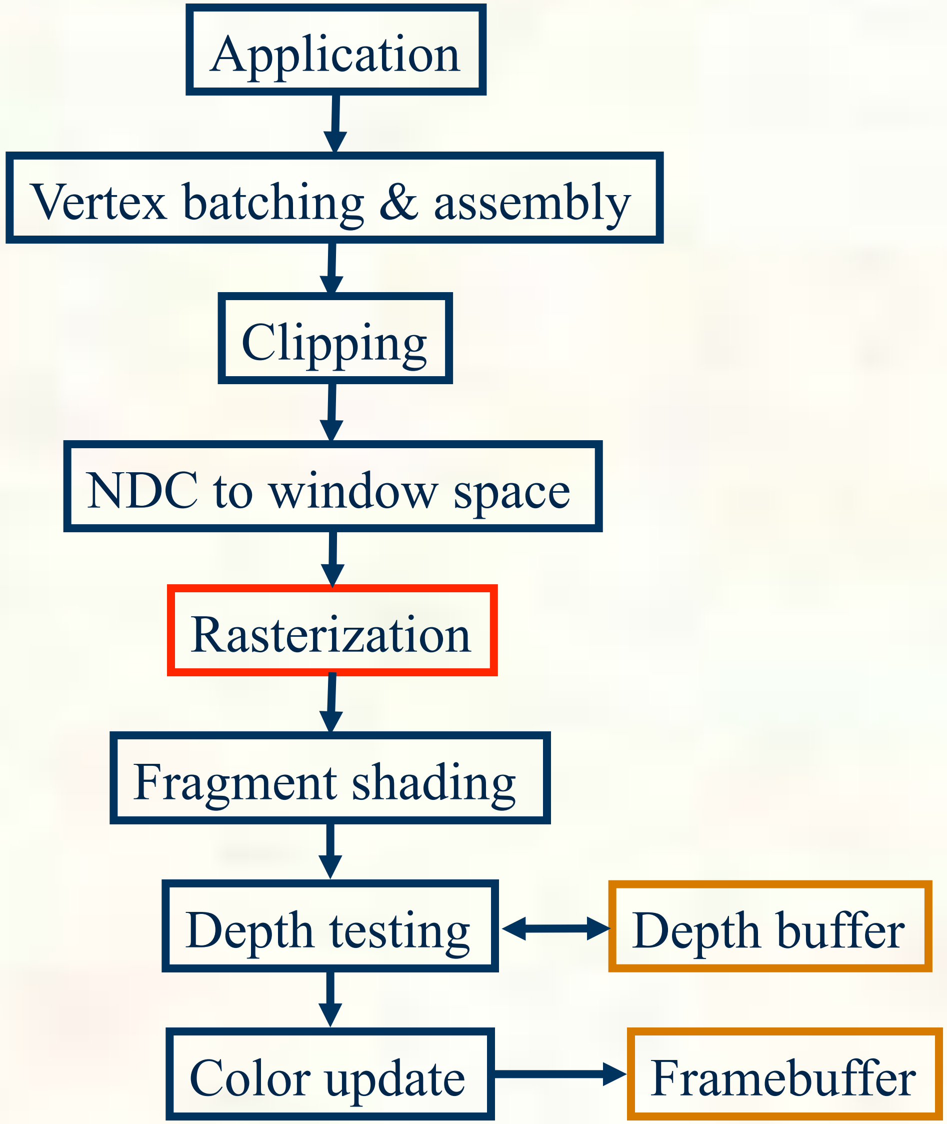
Apply the Transforms

- First vertex :: (-0.8, 0.8, 0.3)
  - $w_x = (w/2)\times x + v_x + w/2 = 250\times(-0.8) + 250 = 50$
  - $w_y = (h/2)y + v_y + h/2 = 250\times(0.8) + 250 = 450$
  - $w_z = [(f-n)/2]\times z + (n+f)/2 = 0.65$
- Second vertex :: (0.8, 0.8, -0.2)
  - $w_x = (w/2)\times x + v_x + w/2 = 250\times(-0.8) + 250 = 50$
  - $w_y = (h/2)y + v_y + h/2 = 250\times(0.8) + 250 = 450$
  - $w_z = [(f-n)/2]\times z + (n+f)/2 = 0.4$
- Third vertex :: (0, -0.8, -0.2)
  - $w_x = (w/2)\times x + v_x + w/2 = 250\times 0 + 250 = 250$
  - $w_y = (h/2)y + v_y + h/2 = 250\times(-0.8) + 250 = 50$
  - $w_z = [(f-n)/2]\times z + (n+f)/2 = 0.4$

Rasterization

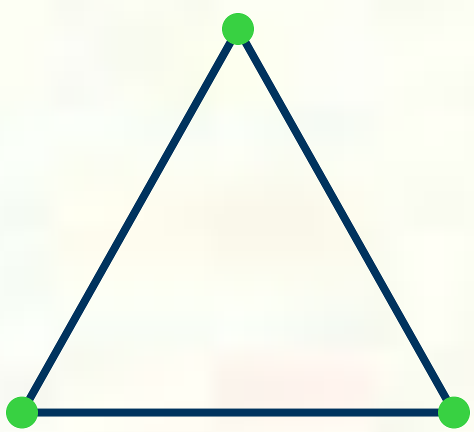
- Process of converting a clipped triangle into a set of sample locations covered by the triangle
  - Also can rasterize points and lines



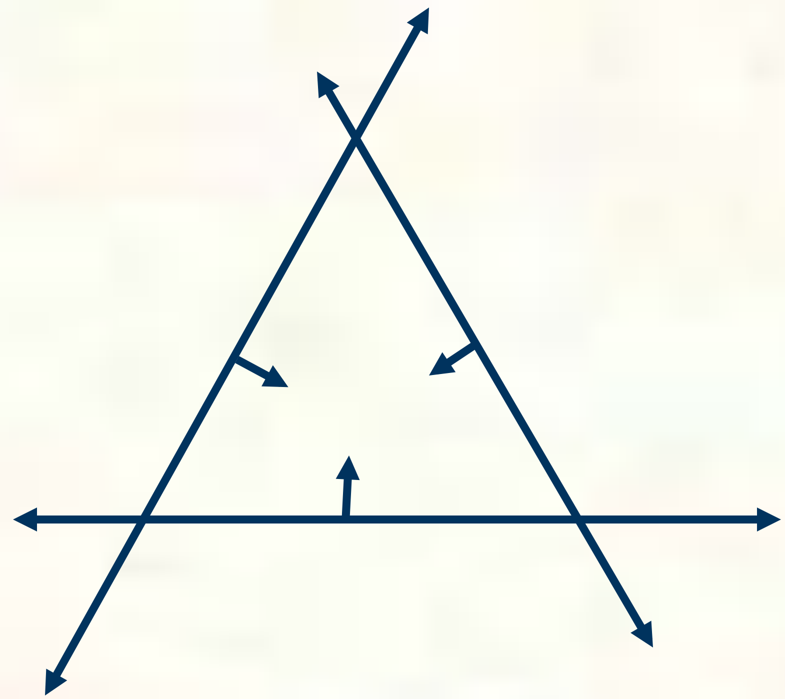


Determining a Triangle

- **Classic view**: 3 points determine a triangle
  - Given 3 vertex positions, we determine a triangle
  - Hence glVertex3f/ glVertex3f/glVertex3f

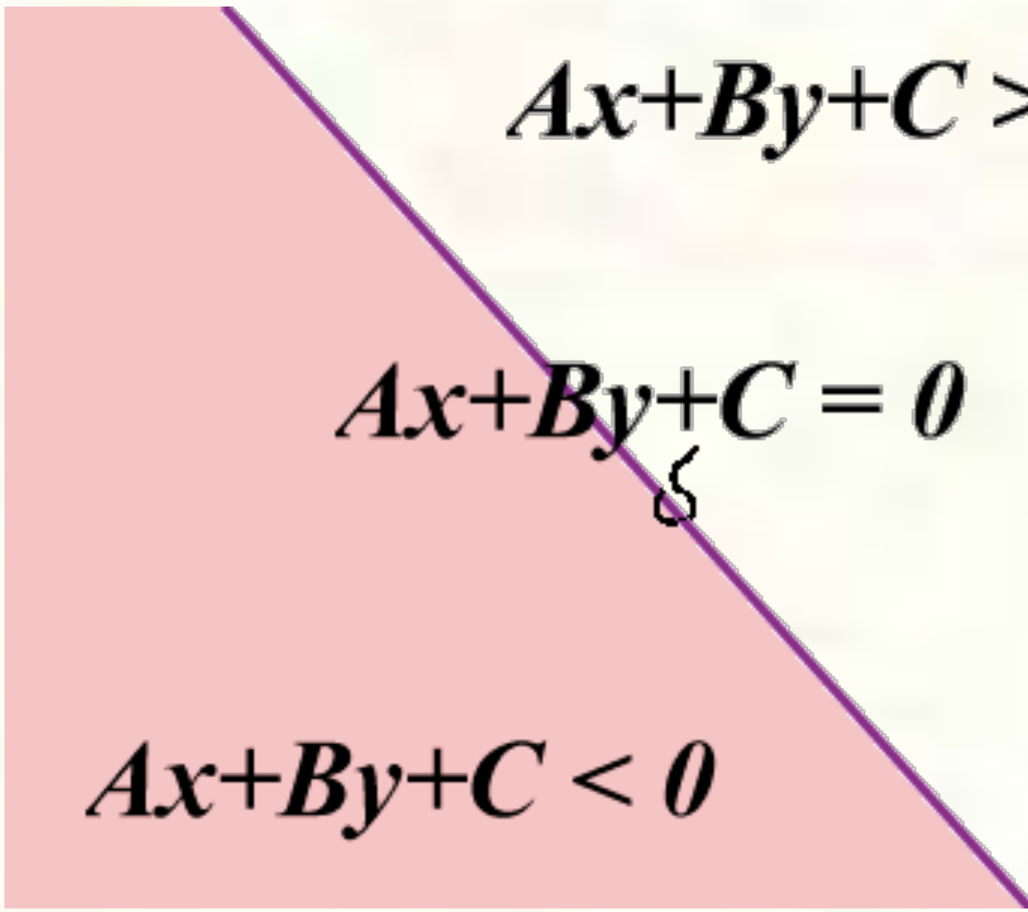


- **Rasterization view**: 3 oriented edge equations determine a triangle

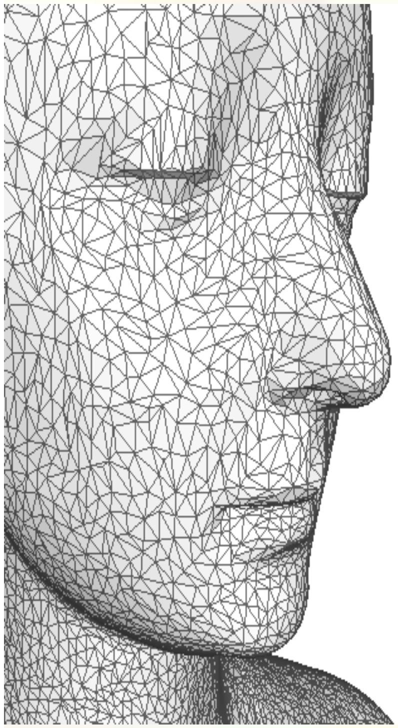


Each oriented edge equation in form:
$A*x + B*y + C \geq 078$

Oriented Edge Equations



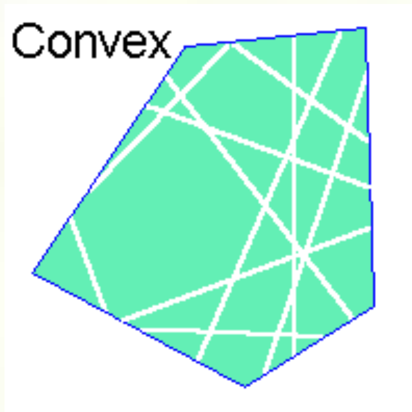
Step back: Why Triangles?

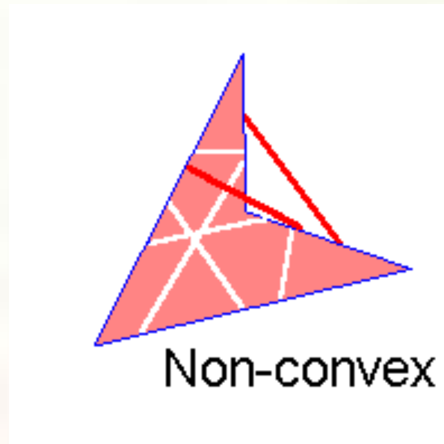


Face meshed with triangles

- Simplest linear primitive with area
  - If it got any simpler, the primitive would be a line (just 2 vertexes)
  - Guaranteed to be planar (flat) and convex (not concave)
- Triangles are compact
  - 3 vertexes, 9 scalar values in affine 3D, determine a triangle
  - When in a mesh, vertex positions can be "shared" among adjacent triangles
- Triangles are simple
  - Simplicity and generality of triangles facilitates elegant, hardware-amenable algorithms
- Triangles lacks curvature
  - BUT with enough triangles, we can piecewise approximate just about any manifold
- We can subdivide regions of high curvature until we reach flat regions to represent as a triangle

Concave vs. Convex

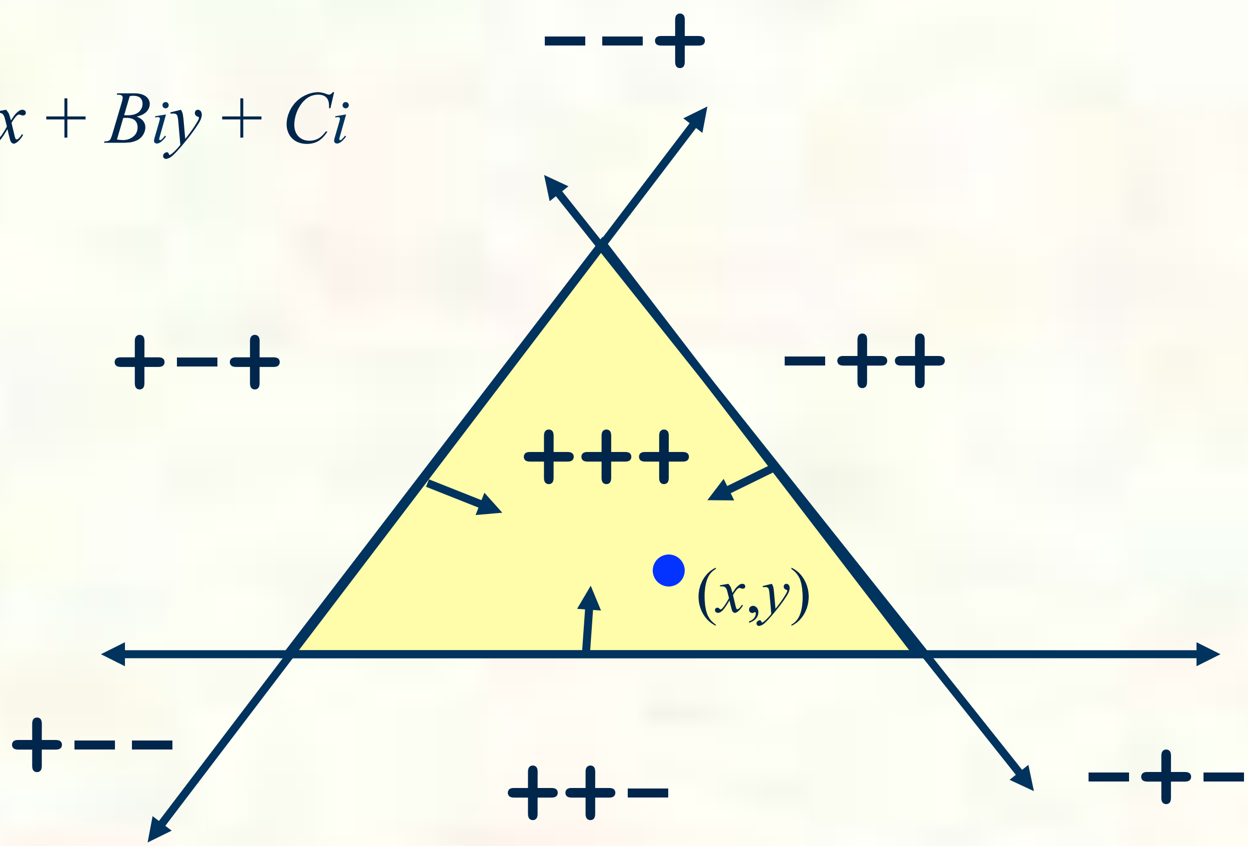




- Region is convex if any two points can be connected by a line segment where all points on this segment are also in the region
  - Opposite is non-convex
- Concave means the region is connected but NOT convex
  - Connected means there's some path (not necessarily a line) from every two points in the region that is entirely in the region

7 Cases

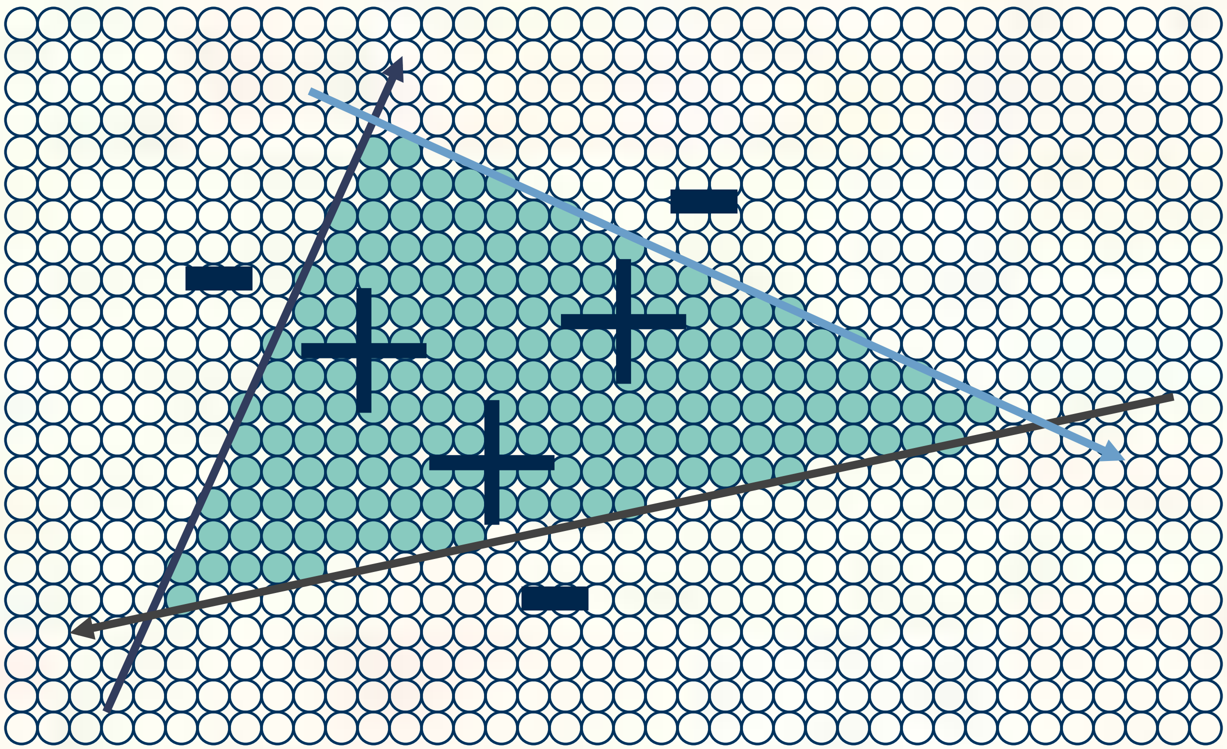
$$E_i(x,y) = Aix + Biy + Ci$$



Inside Triangle Test

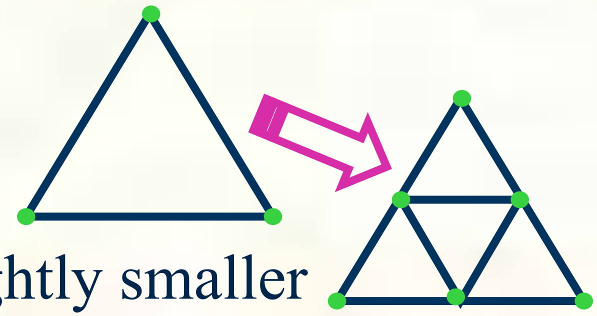
- Evaluate edge equations at grid of sample points
  - If sample position is "inside" all 3 edge equations, the position is "within" the triangle
  - Implicitly parallel—all samples can be tested at once

- Good for hardware implementation
  - Pixel-planes
  - Pineda tiled extension

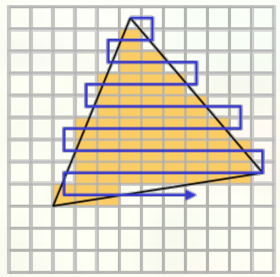


Other Rasterization Approaches

- Subdivision approaches
  - Easy to split a triangle into 4 triangles
  - Keep splitting triangles until they are slightly smaller than your samples
    - Often called micro-polygon rendering
    - Chief advantage is being able to apply displacements during the subdivision

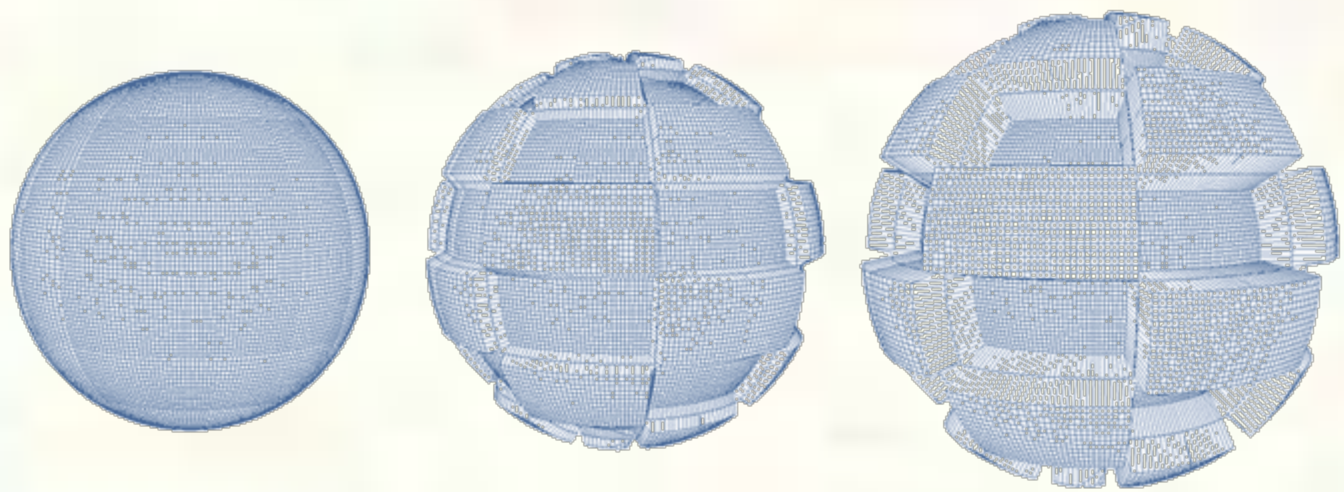


- Edge walking approaches
  - Often used by CPU-based rasterizers
  - Much more sequential than Pineda approach
  - Work efficient and amendable to fixed-point implementation



Micropolygons

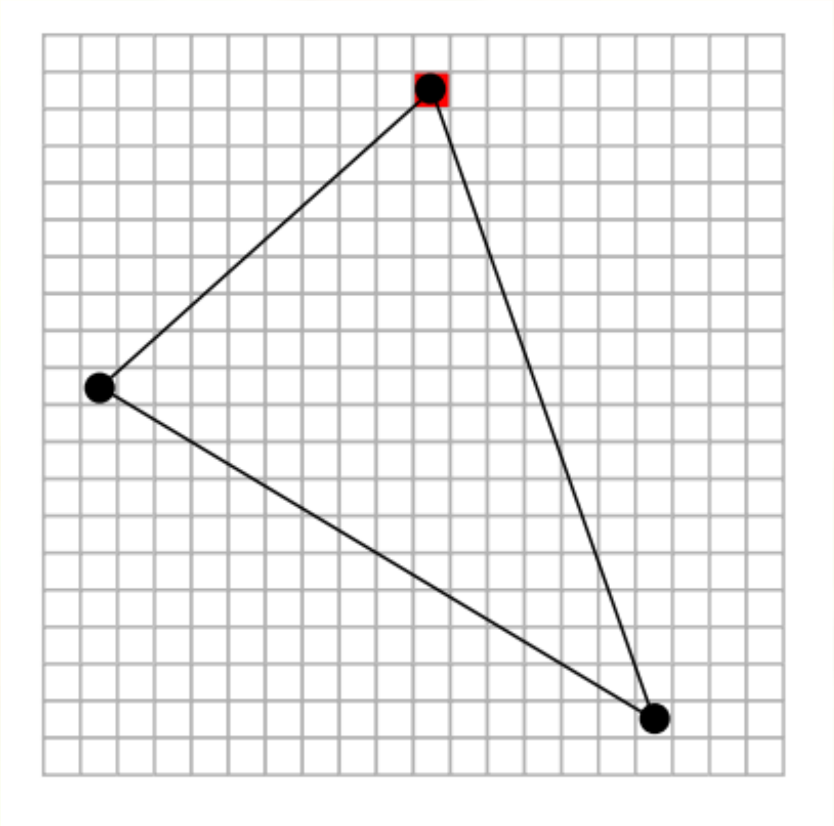
- Rasterization becomes a geometry dicing process
  - Approach taken by Pixar
    - For production rendering when scene detail and quality is at a premium; interactivity, not so much
  - High-level representation is generally patches rather than mere triangles



Displacement mapping of a meshed sphere [Pixar, RenderMan]

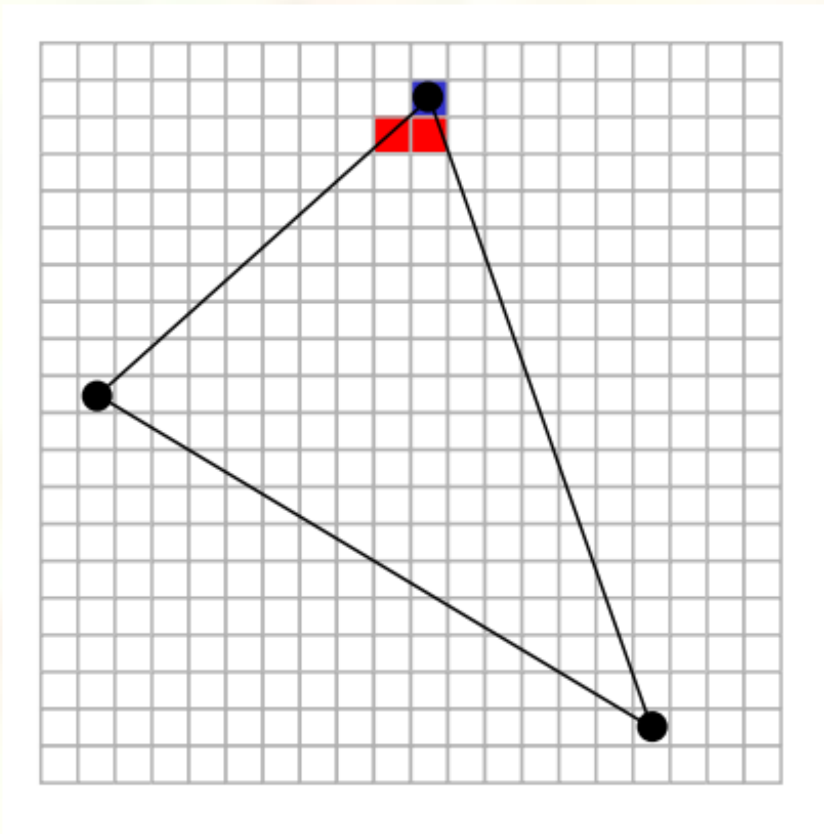
Scanline Rasterization

- Find a “top” to the triangle
  - Now walk down edges



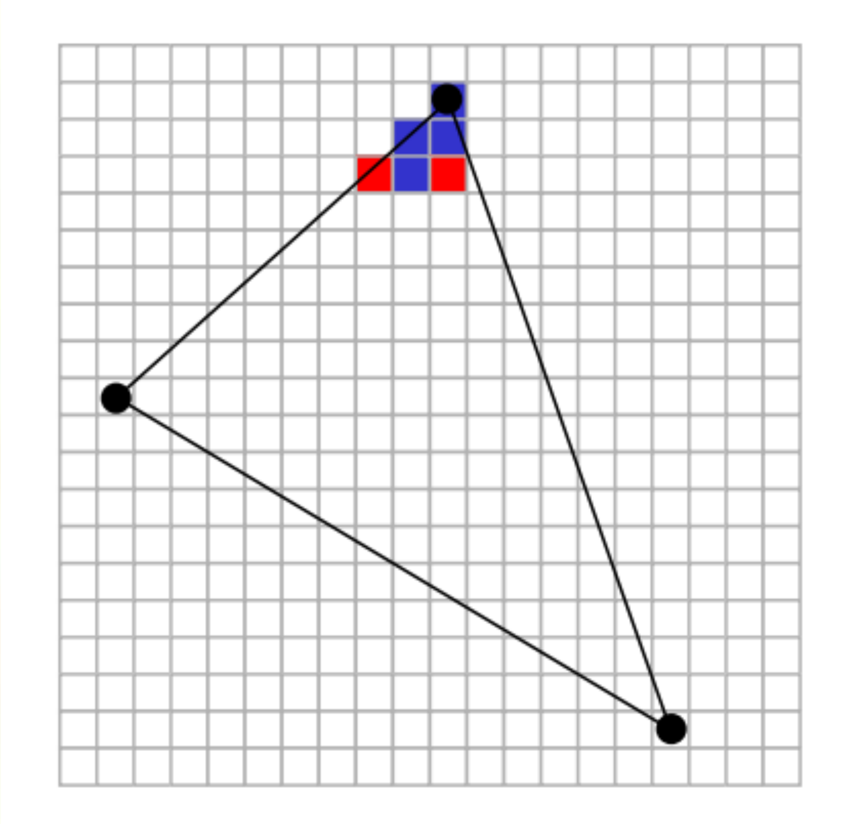
Scanline Rasterization

- Move down a scan-line, keeping track of the left and right ends of the triangle



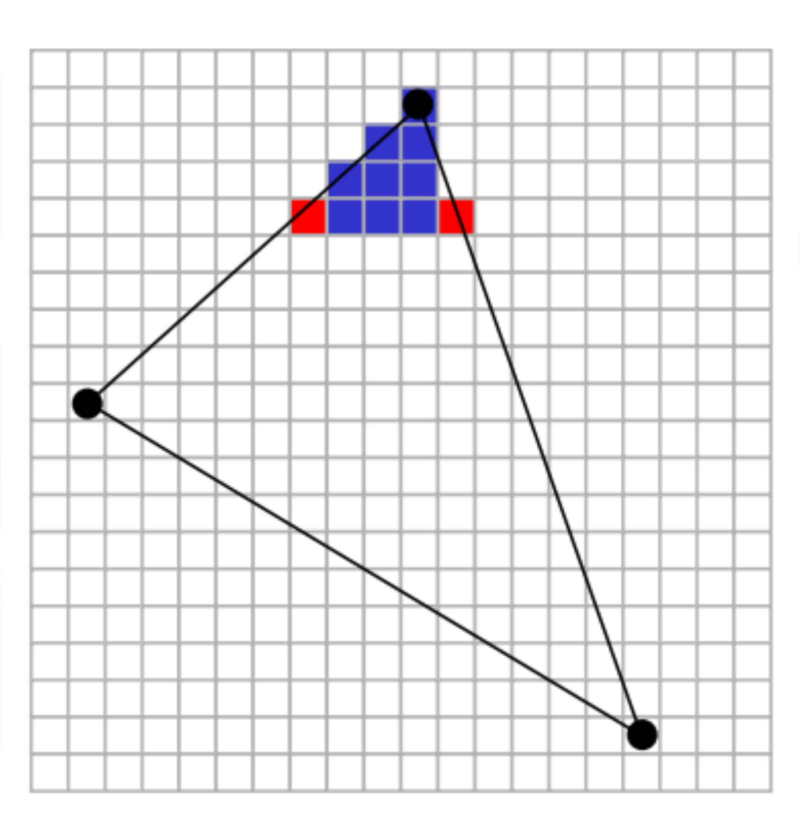
Scanline Rasterization

- Repeat, moving down a scanline
  - Cover the samples between the left and right ends of the triangle in the scan-line



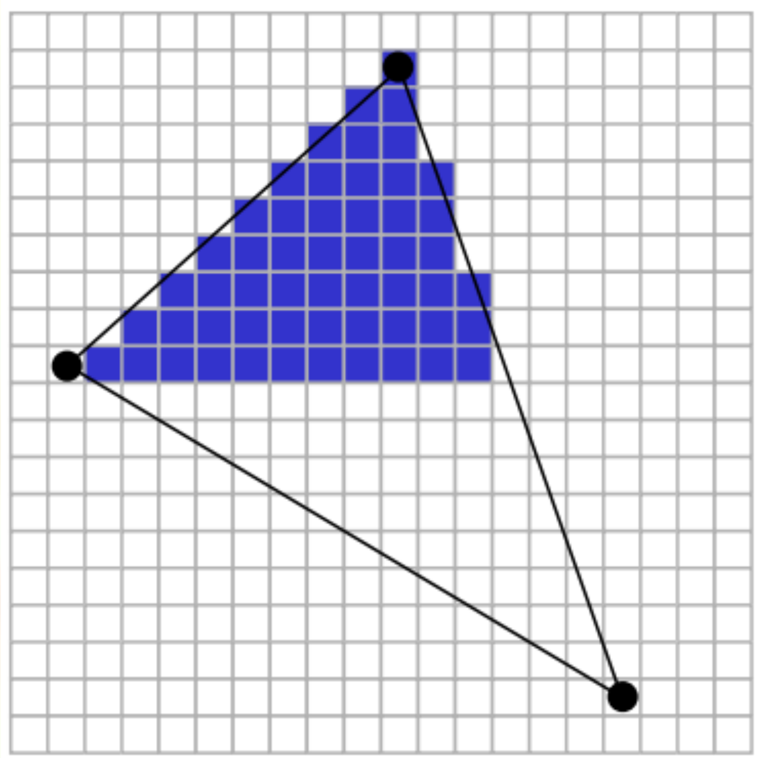
Scanline Rasterization

- Process repeats for each scanline
  - Easy to "step" down to the next scanline based on the slopes of two edges



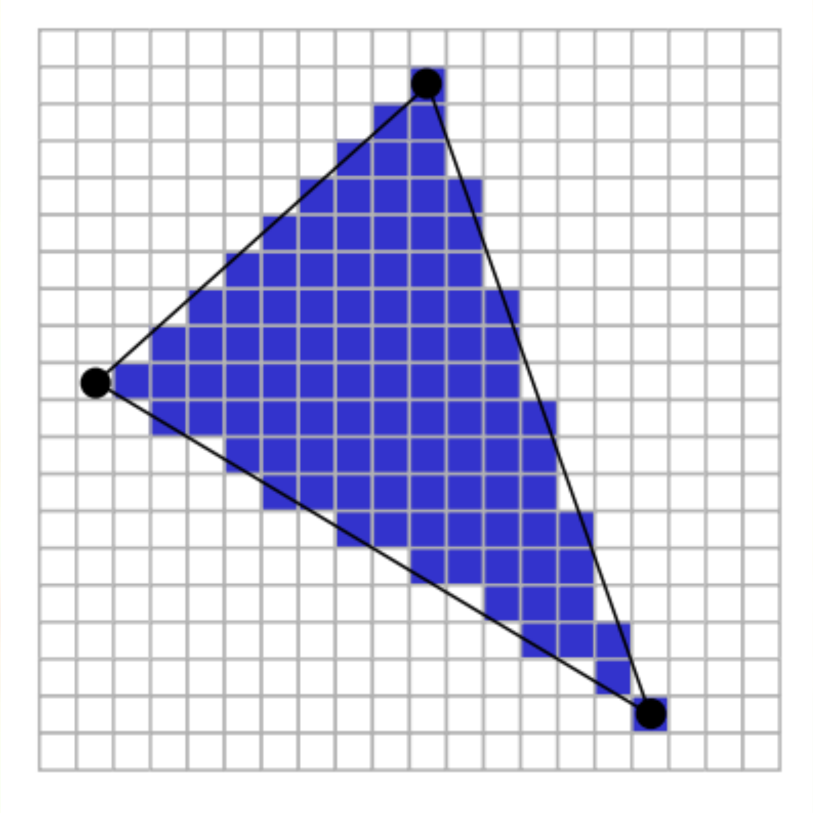
Scanline Rasterization

- Eventually reach a vertex
  - Transition to a different edge and continue filling the span within the triangle



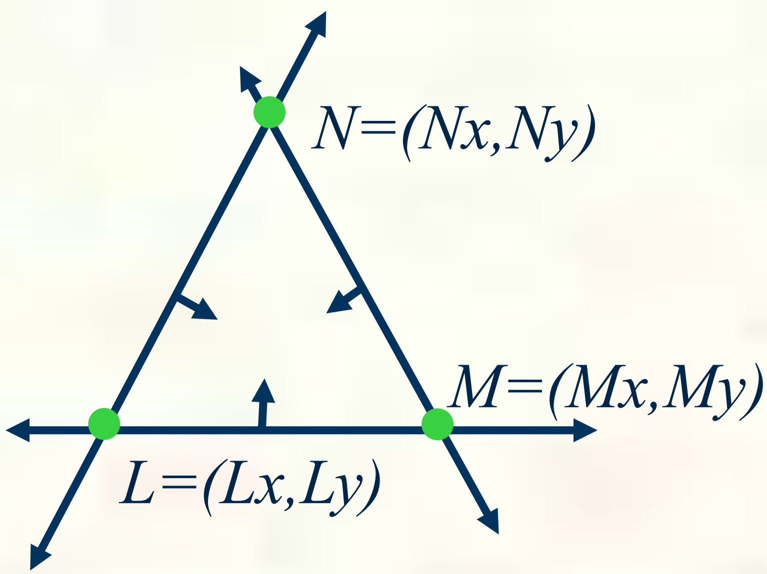
Scanline Rasterization

- Until you finish the triangle
  - Friendly for how CPU memory arranges an image as a 2D array with horizontal locality
  - Layout is good for raster scan-out too



Creating Edge Equations

- Triangle rasterization need edge equations
  - How do we make edge equations?
- An edge is a line so determined by two points
  - Each of the 3 triangle edges is determined by two of the 3 triangle vertexes (L, M, N)



How do we get

$$A*x + B*y + C \geq 0$$

for each edge from L, M, and N?

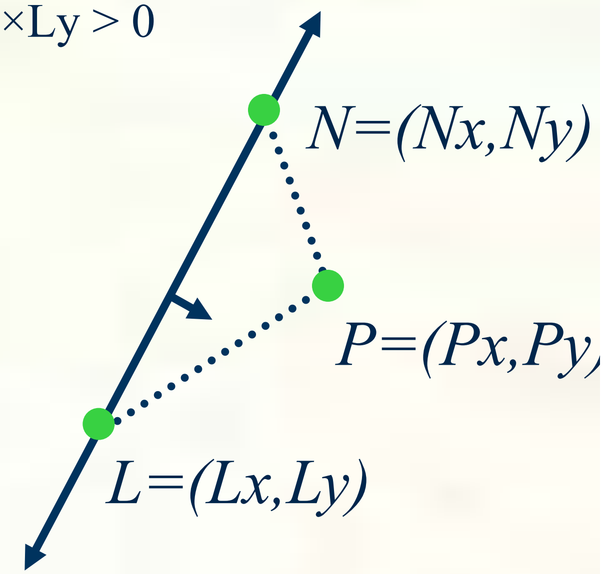
Edge Equation Setup

- How do you get the coefficients A, B, and C?
- Determinants help—consider the LN edge:

P is an arbitrary point

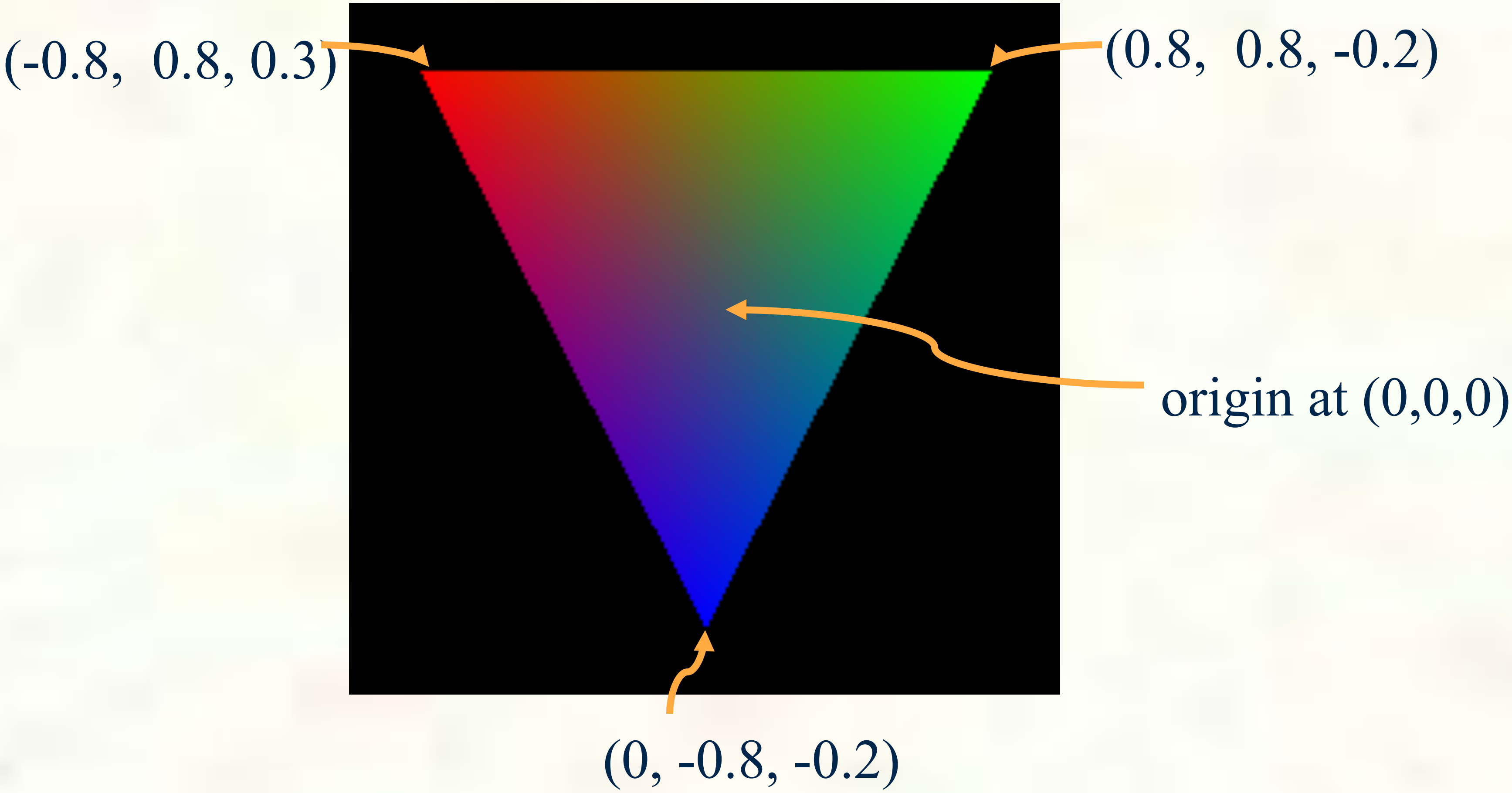
$$\begin{vmatrix} N_x - L_x & N_y - L_y \\ P_x - L_x & P_y - L_y \end{vmatrix} > 0 \quad \textit{or more succinctly} \quad \begin{vmatrix} N - L \\ P - L \end{vmatrix} > 0$$

- **Expansion:** $(Ly-Ny)\times Px + (Nx-Lx)\times Py + Ny\times Lx-Nx\times Ly > 0$
  - A_{LN} = Ly-Ny
  - B_{LN} = Nx-Lx
  - C_{LN} = Ny×Lx-Nx×Ly
- **Geometric interpretation:** twice signed area of the triangle LPN



Triangle Vertices in Screen Space

- Assume the window is 500x500 pixels
  - So glViewport(0,0,500,500) has been called

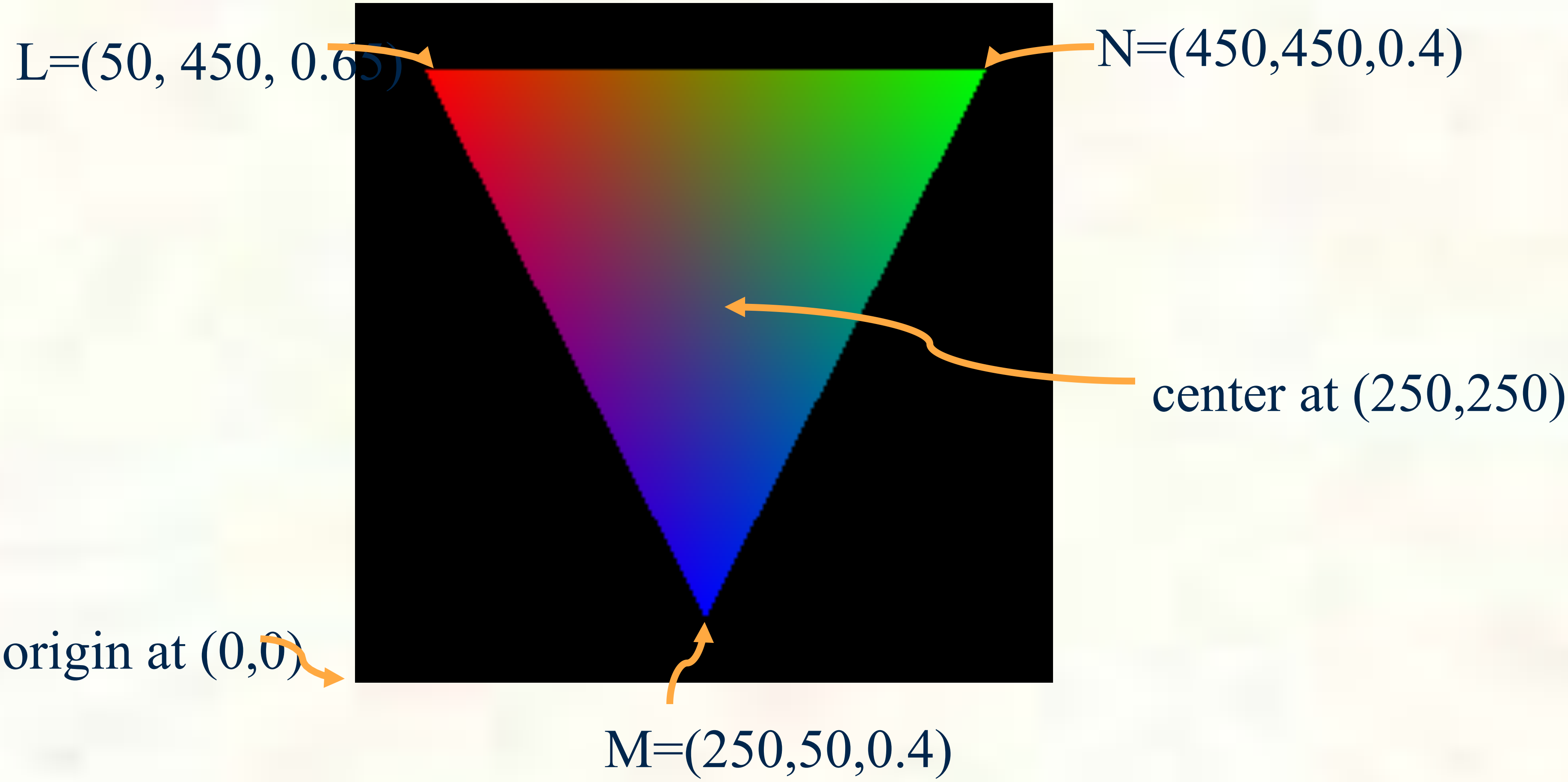


Apply the Transform

- First vertex :: (-0.8, 0.8, 0.3)
  - $w_x = (w/2)\times x + v_x + w/2 = 250\times(-0.8) + 250 = 50$
  - $w_y = (h/2)y + v_y + h/2 = 250\times(0.8) + 250 = 450$
  - $w_z = [(f-n)/2]\times z + (n+f)/2 = 0.65$
- Second vertex :: (0.8, 0.8, -0.2)
  - $w_x = (w/2)\times x + v_x + w/2 = 250\times(0.8) + 250 = 450$
  - $w_y = (h/2)y + v_y + h/2 = 250\times(0.8) + 250 = 450$
  - $w_z = [(f-n)/2]\times z + (n+f)/2 = 0.4$
- Third vertex :: (0, -0.8, -0.2)
  - $w_x = (w/2)\times x + v_x + w/2 = 250\times 0 + 250 = 250$
  - $w_y = (h/2)y + v_y + h/2 = 250\times(-0.8) + 250 = 50$
  - $w_z = [(f-n)/2]\times z + (n+f)/2 = 0.4$

Screen Space Coordinates of Triangle

- Assume the window is 500x500 pixels
  - So glViewport(0,0,500,500) has been called



Look at the LN edge

- **Expansion:**

  $(Ly\text{-}Ny)\times Px + (Nx\text{-}Lx)\times Py + Ny\times Lx\text{-}Nx\times Ly > 0$

  - $A_{LN} = Ly\text{-}Ny = 450\text{-}450 = 0$
  - $B_{LN} = Nx\text{-}Lx = 50\text{-}450 = -400$
  - $C_{LN} = Ny\times Lx\text{-}Nx\times Ly = 180{,}000$
- Is center at (250,250) in the triangle?
  - $A_{LN} \times 250 + B_{LN} \times 250 + C_{LN} = ???$
  - $0 \times 250 - 400 \times 250 + 180{,}000 = 80{,}000$
    - 80,000 > 0 so (250,250) is <u>in</u> the triangle

All Three Edge Equations

- All three triangle edge equations:

$$\begin{vmatrix} N-P \\ M-P \end{vmatrix} > 0 \qquad \begin{vmatrix} N-L \\ P-L \end{vmatrix} > 0 \qquad \begin{vmatrix} P-L \\ M-L \end{vmatrix} > 0$$

- Satisfy all 3 and P is in the triangle
  - And then rasterize at sample location P
- **Caveat:** if $\begin{vmatrix} N-L \\ M-L \end{vmatrix} < 0$ reverse the comparsion sense

Water Tight Rasterization

- Two triangles often share a common edge
  - Indeed in closed polygonal meshes, every triangle shares its edges with as many as three other triangles
    - Called adjacent or "shared edge" triangles
- Crucial rasterization property
  - No double sampling (hitting) along the shared edge
  - No sample gaps (pixel fall-out) along the shared edge
  - Samples along the shared edge must be belong to exactly one of the two triangles
    - Not both, not neight
- Water tight rasterization is crucial to many higher-level algorithms; otherwise, rendering artifacts
  - Possible artifact: if pixels hit twice on an edge, the pixel could be double blended
  - Example application: Stenciled Shadow Volumes (SSV)

Water Tight Rasterization Solution

- First "snap" vertex positions to a grid
  - Grid can (and should) be sub-pixel samples
  - Results in fixed-point vertex positions
- Fixed-point math allows exact edge computations
  - **Surprising? E**nsuring robustness requires discarding excess precision
- Problem
  - What happens when edge equation evaluates to exactly zero at a sample position?
  - Need a consistent tight breaker

Tie Breaker Rule

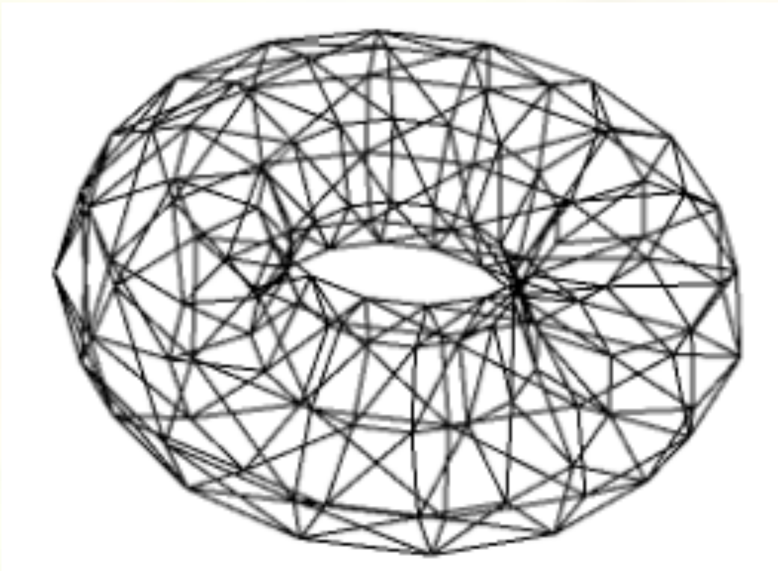
- Look at edge equation coefficients
- Tie-breaker rule when edge equation evaluates to zero
  - "Inside" edge when edge equation is zero <u>and</u> $A > 0$ when $A \neq 0$, or $B > 0$ when $A = 0$
- Complete coverage determination rule
  - if (E(x,y) > 0 || (E(x,y)==0 && (A != 0 ? A > 0 : B > 0)))

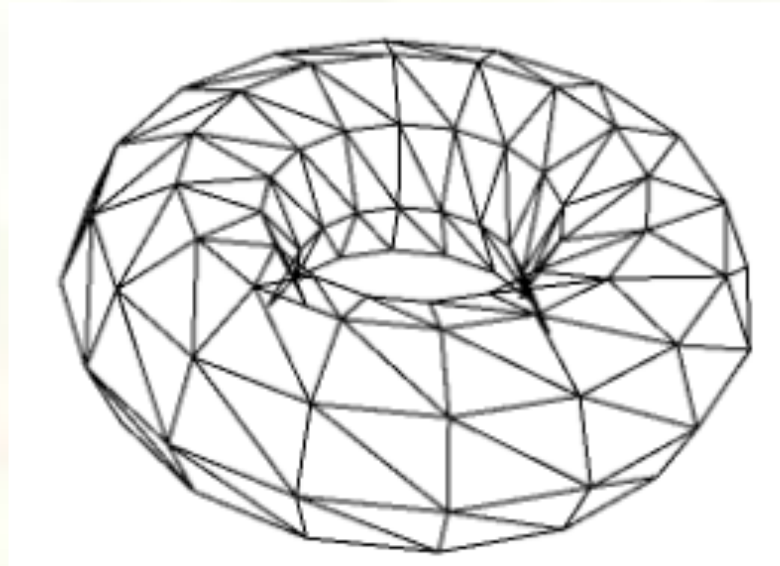
    sample at (x,y) is inside edge

Zero Area Triangles

- We reverse the edge equation comparison sense if the (signed) area of the triangle is negative
- What if the area is zero?
  - Linear algebra indicates a singular matrix
  - Need to cull the primitive
- Also useful to cull primitives when area is negative
  - OpenGL calls this face culling
    - Enabled with glEnable(GL_CULL_FACE)
  - When drawing closed meshes, back face culling can avoid drawing primitives assured to be occluded by front faces

Back Face Culling Example





Torus drawn in wire-frame <u>without</u> back face culling

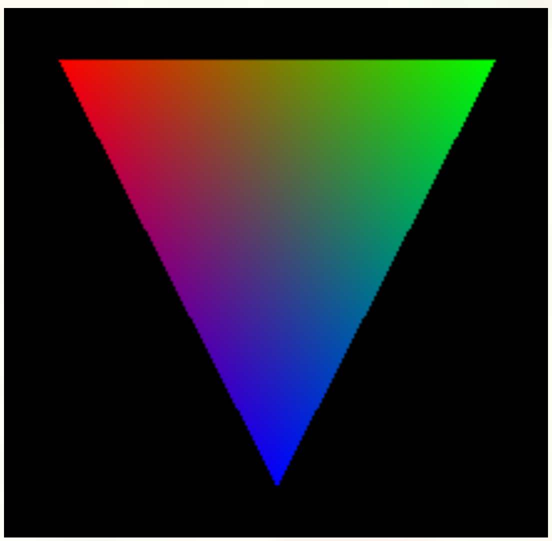
Notice considerable extraneous triangles that would normally be occluded

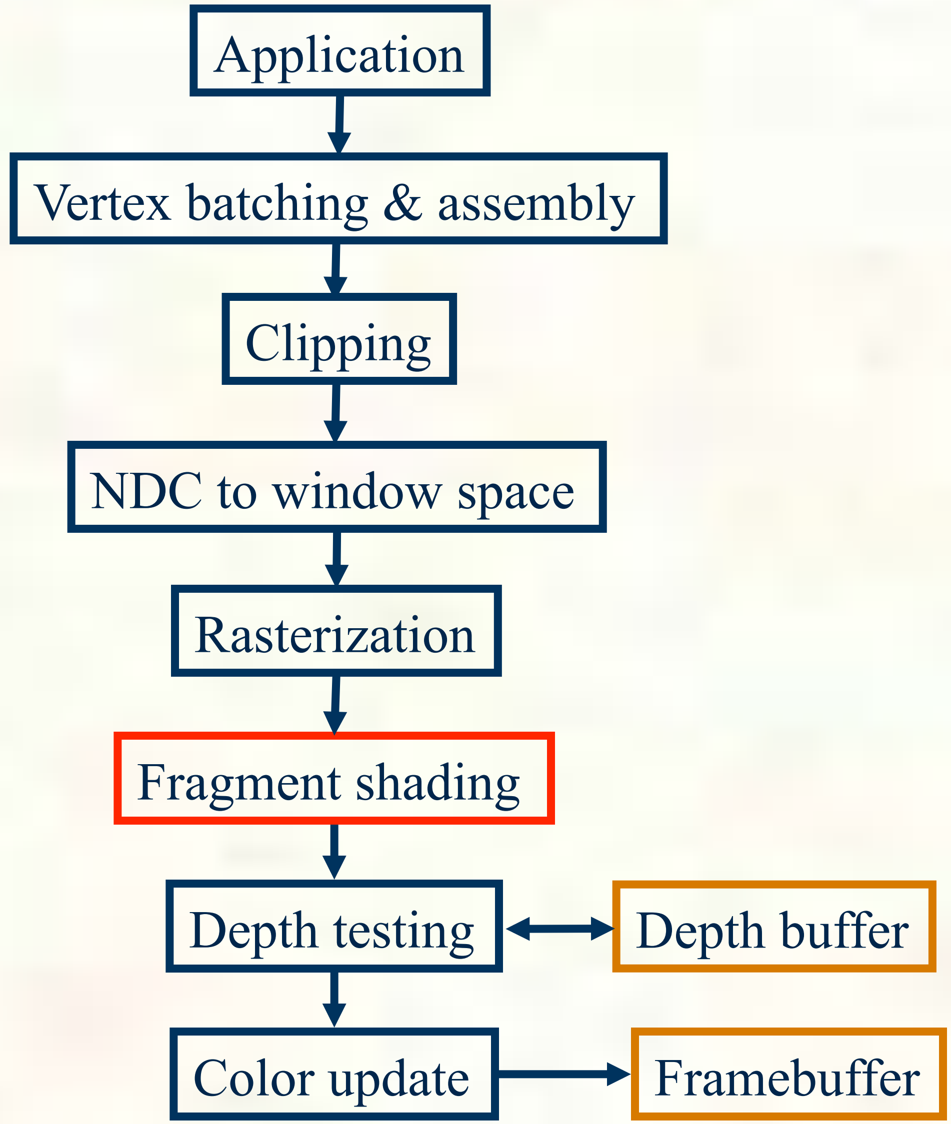
Torus drawn in wire-frame **with** back face culling

By culling back-facing (negative signed area) triangles, fewer triangles are rasterized

Simple Fragment Shading

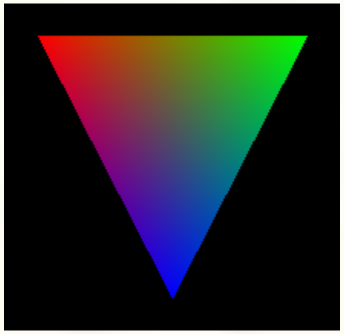
- For all samples (pixels) within the triangle, evaluate the interpolated color
  - Requires having math to determine color at the sample (x,y) location



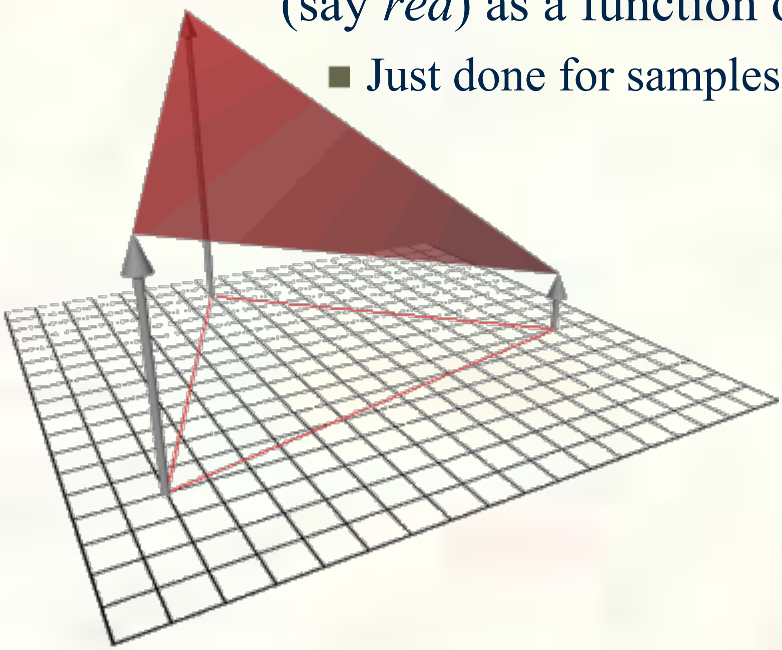


Color Interpolation

- Our simple triangle is drawn with smooth color interpolation
  - Recall: glShadeModel(GL_SMOOTH)



- How is color interpolated?
  - Think of a plane equation to computer each color component (say *red*) as a function of (x,y)
    - Just done for samples positions within the triangle



$$"redness" = A_{red}x + B_{red}y + C_{red}$$

Setup Plane Equation

- Setup plane equation to solve for "red" as a function of (x,y)

Setup system of equations

$$\begin{bmatrix} L_{red} \\ M_{red} \\ N_{red} \end{bmatrix} = \begin{bmatrix} L_x & L_y & 1 \\ M_x & M_y & 1 \\ N_x & N_y & 1 \end{bmatrix} \begin{bmatrix} A_{red} \\ B_{red} \\ C_{red} \end{bmatrix}$$

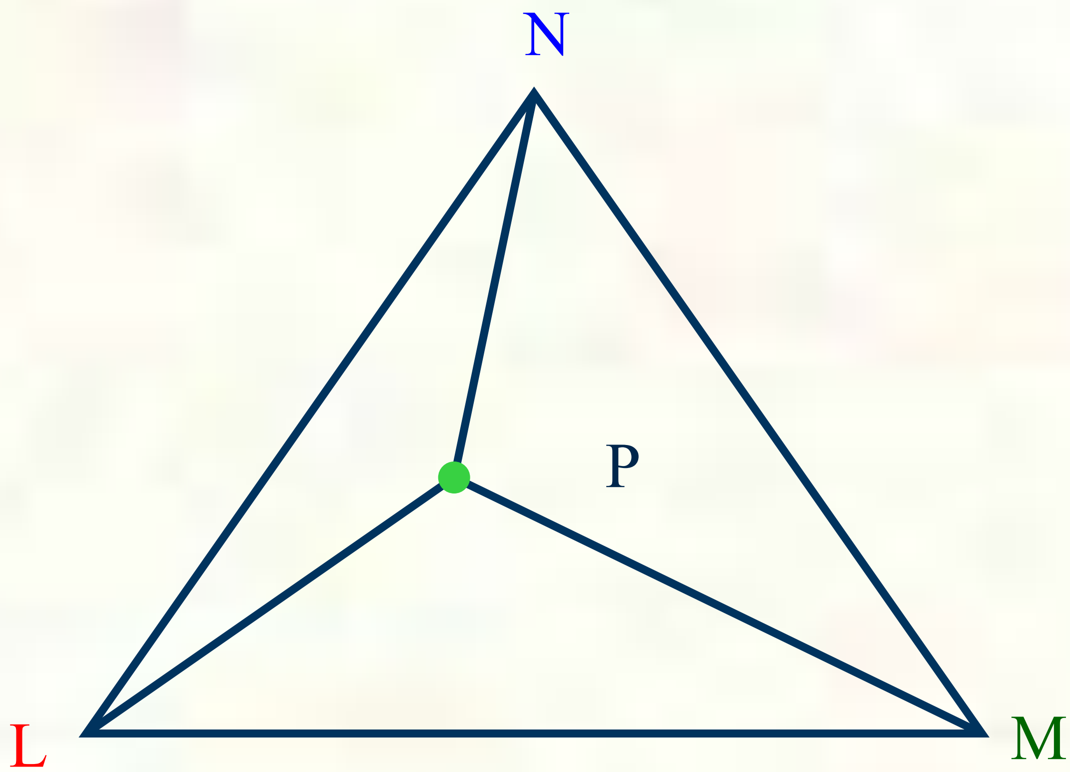
Solve for plane equation coefficients A, B, C

$$\begin{bmatrix} L_x & L_y & 1 \\ M_x & M_y & 1 \\ N_x & N_y & 1 \end{bmatrix}^{-1} \begin{bmatrix} L_{red} \\ M_{red} \\ N_{red} \end{bmatrix} = \begin{bmatrix} A_{red} \\ B_{red} \\ C_{red} \end{bmatrix}$$

Do the same for green, blue, and alpha (opacity)…

More Intuitive Way to Interpolate

- Barycentric coordinates



$$\frac{\text{Area}(PMN)}{\text{Area}(LMN)} = \alpha$$

$$\frac{\text{Area}(LPN)}{\text{Area}(LMN)} = \beta$$

$$\frac{\text{Area}(LMP)}{\text{Area}(LMN)} = \gamma$$

Note: $\alpha + \beta + \gamma = 0$ by construction

$$\text{attribute}(P) = \alpha \times \text{attribute}(L) + \beta \times \text{attribute}(M) + \gamma \times \text{attribute}(N)$$

Hardware Triangle Rendering Rates

- Top GPUs can setup over a billion triangles per second for rasterization
- Triangle setup & rasterization is just one of the (many, many) computation steps in GPU rendering

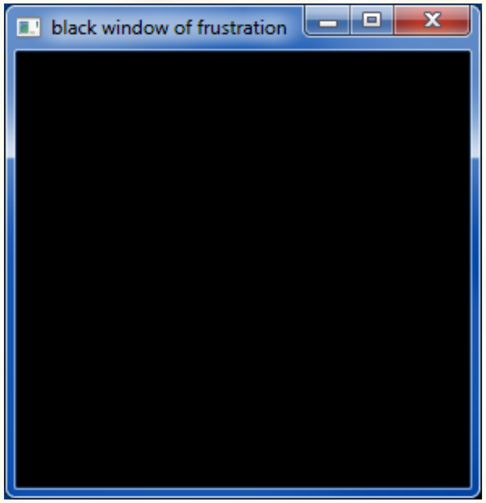
Remaining Steps

- Depth interpolation
- Color update
- Scan-out to the display

- *Next time…*

Programming tips

- 3D graphics, whether OpenGL or Direct3D or any other API, can be frustrating
  - You write a bunch of code and the result is



Nothing but black window; where did your rendering go??

Things to Try

- Set your clear color to something other than black!
  - It is easy to draw things black accidentally so don't make black the clear color
  - But black is the initial clear color
- Did you draw something for one frame, but the next frame draws nothing?
  - Are you using depth buffering? Did you forget to clear the depth buffer?
- Remember there are near and far clip planes so clipping in Z, not just X & Y
- Have you checked for glGetError?
  - Call glGetError once per frame while debugging so you can see errors that occur
  - For release code, take out the glGetError calls
- Not sure what state you are in?
  - Use glGetIntegerv or glGetFloatv or other query functions to make sure that OpenGL's state is what you think it is
- Use glutSwapBuffers to flush your rendering and show to the visible window
  - Likewise glFinish makes sure all pending commands have finished
- Try reading
  - http://www.slideshare.net/Mark_Kilgard/avoiding-19-common-opengl-pitfalls
  - This is well worth the time wasted debugging a problem that could be avoided

Next Lecture

- Finish OpenGL pipeline
- Transforms and Graphics Math
  - *Interpolation, vector math, and number representations for computer graphics*

Thanks

- Presentation approach and figures from
  - David Luebke [2003]
  - Brandon Lloyd [2007]
  - *Geometric Algebra for Computer Science* [Dorst, Fontijne, Mann]
  - via Mark Kilgard