Computer Graphics

Discussion 5

Garett Ridge garett@cs.ucla.edu,
Sam Amin samamin@ucla.edu,
Theresa Tong theresa.r.tong@gmail.com,
Quanjie Geng szmun.genggj@gmail.com

Recall midterm questions?

Part I: Assignment 2

Assignment 2: Animate anything

- Make any animation or game as long as it includes all the required techniques
- Graded on creativity, complexity, and attention to detail
- Will be played and voted on by the class -- the best receive extra credit towards the course grade
- Yours can join prior years' showcases online. Example:

  http://web.cs.ucla.edu/~dt/courses/CS174A/animations/assignment2-best-16s/

Assignment 2: Animate anything

Special requirements:

- Utilize lookAt() function
- Output frame rate
- Make a custom Shape subclass, featuring texture coordinates and visible seams
- Creativity, Complexity, Quality

How to call Ed Angel’s lookAt() (MV.js)

```
this.shapes.ball.draw( graphics_state, translation( 0, 0,  3), grey_material );
this.shapes.ball.draw( graphics_state, translation( 0, 0, -3), blue_material );

if( ( graphics_state.animation_time / 1000 ) % 2 < 1 )            // (alternating each second)
      graphics_state.camera_transform = lookAt( [0,0,0], [0,0,-1], [0,1,0] ); // Pass in eye position, at
else                                                                          // position, and up vector.
      graphics_state.camera_transform = lookAt( [0,0,0], [0,0, 1], [0,1,0] );
```

How to call Ed Angel's lookAt() (MV.js)

- Camera placement without lookAt():
  - Not as easy to point directly at things, but valid.
  - Generate it using mult()/rotate()/translate()/scale() instead of lookAt()
  - Remember inverse() concepts apply to cameras
    - Any incremental modifications you make will encounter properties of inverted products (reverse the order ***<u>and</u>*** invert each part)

Advanced Project Topics - Further reading

- Procedural textures (perlin noise, shadertoy.com code)
- Shader particle effects (sparkles, fire, smoke, splashing, etc.)
- Procedural shapes (plants) – look up L-systems
- Shadow mapping (Projection onto surfaces)
- Shape enhancement: Bump mapping, displacement maps
- Linear / rotational momentum
- Collision detection (shape intersection tests) and response

Advanced Project Topics - Further reading

- Spring/damper physics - algebra & neighborhood graphs
- Interpolated curved paths or shapes
- Movement interpolation - “tweening”
- Camera aim interpolation - Quaternions, geodesic lines
- Import OBJ files (new shapes from files)
- Marching cubes / drawing implicit fields & volumes
- Fractal generation & rendering – look up Mandelbulb videos
- Mouse picking (clicking to correctly select objects)

Advanced Project Topics - Further reading

- Scene Graphs – manipulating a scene tree & its transforms
  - Chapter 9 in your textbook
    - Each node stores a JavaScript array of references to child nodes… a whole tree data structure results.
    - Each node also stores its delta matrix (to be post-multiplied to model transform)
  - The tree's transforms can be altered procedurally

Project Topics

- Maybe make 3D objects first, then tell a story about the objects you manage to make
  - Good stories are drawn from real events / existing media
- Maybe choose an advanced topic first, then once it works make it interactive or a challenge

Part II: Scene Timing & Organizing Movement

Scene timing ideas

- Use “if” statements for switching scenes
- Keep multiple copies of the **animation_time** variable with different offsets subtracted from it.
- Name the copies like "**time since (scene #17) started**"
  - If it’s **negative**, the scene shouldn’t start yet, so skip those lines for this frame.
  - If it's **positive**, you know to proceed with the lines of code that animate that scene
  - If it’s **too positive**, it’s past the scene’s end time, so manually overwrite it to some negative number, making the scene go away again.

Scene timing ideas

- Concrete example:

...In display(), set up lots of variables like:

```
var     time_sinceFallingSceneStart = graphics_state.animation_time - 5000;   // Start the scene 5 seconds in.
    if( time_sinceFallingSceneStart >  7000                                    )        // This scene will last 7 seconds...
                time_sinceFallingSceneStart = -1;            // ...after which, it will go away again.
```

- Farther down:

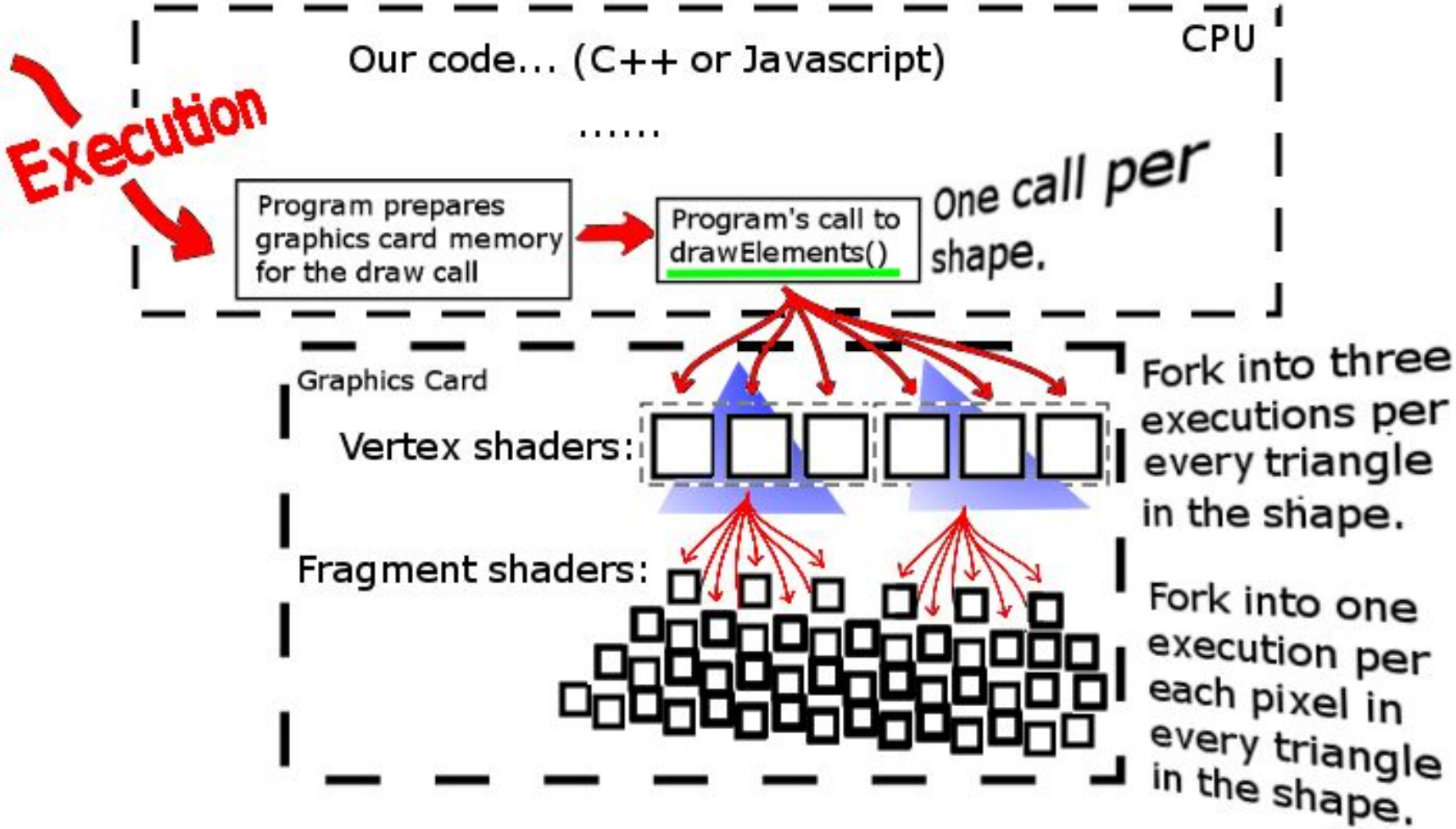
```
if( time_sinceFallingSceneStart > 0 )
        ... // Go on with displaying the falling scene
```

Scene timing ideas

If your program is interactive:

- May want less of a time-dependent scene system -- and more of a **finite state machine (FSM)**
- Conditionals in your display() function would consult the **state** before proceeding with any drawing code sections.
- Certain key inputs will change your state variable to a different enum value, causing those if's to act differently.
  - Multiple offset copies of your time variable will again be needed to track how long each animation has been going on for, so you can change state once it's done.

Part III: What’s a shader?


Execution
CPU
Our code... (C++ or Javascript)
.....
Program prepares graphics card memory for the draw call
Program's call to drawElements()
One call per shape.
Graphics Card
Vertex shaders:
Fragment shaders:
Fork into three executions per every triangle in the shape.
Fork into one execution per each pixel in every triangle in the shape.

A Vertex shader places all three points in their final place.

A Fragment shader samples color-relevant data <u>in between</u> the three vertices that have the data, using interpolation.

Interpolation

Special Cases

Linear combination

$\mathbf{w} = a_1\mathbf{v}_1 + \ldots a_m\mathbf{v}_m, \quad a_1,\ldots,a_m$ in R

Affine combination:

A linear combination for which $a_1 + \ldots + a_m = 1$

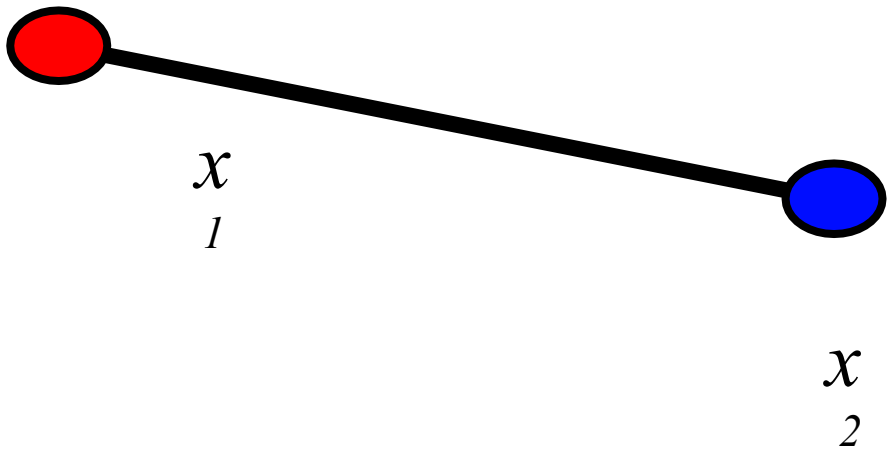
Convex combination

An affine combination for which $a_i \geq 0$ for $i = 1,\ldots,m$

Barycentric Interpolation

How do you interpolate values defined at vertices across the entire triangle?

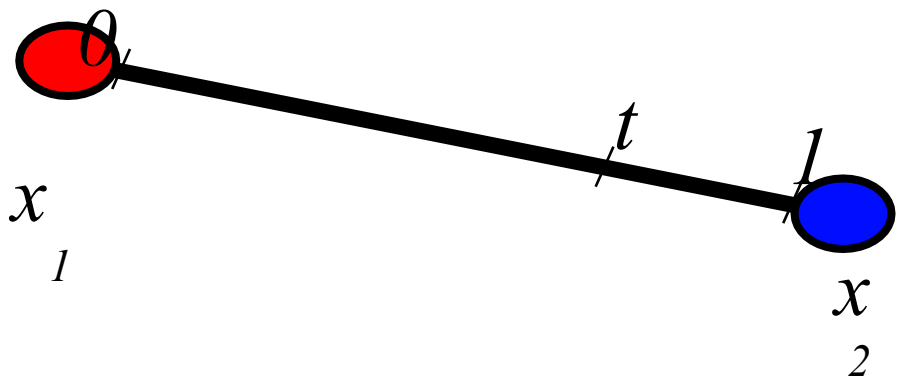
Solve a simpler 1D problem first (each vertex has a color):



Barycentric Interpolation

How do you interpolate values defined at vertices across the entire triangle?

Solve a simpler problem first:

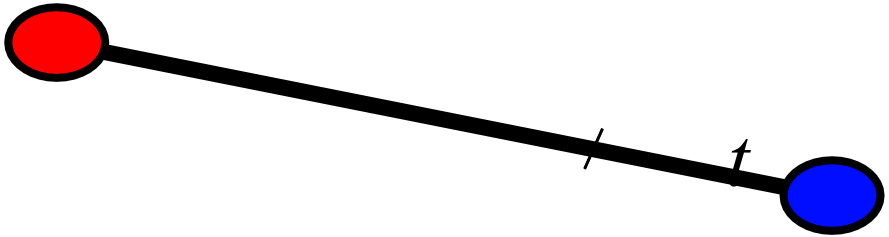


Want to define a value for every $t \in [0,1]$:

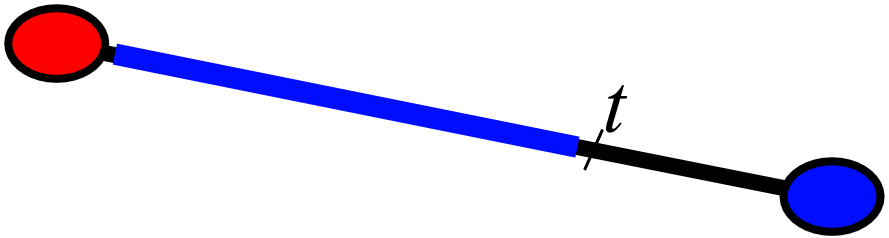
Barycentric Interpolation

How do we come up with this equation?

Look at the picture!

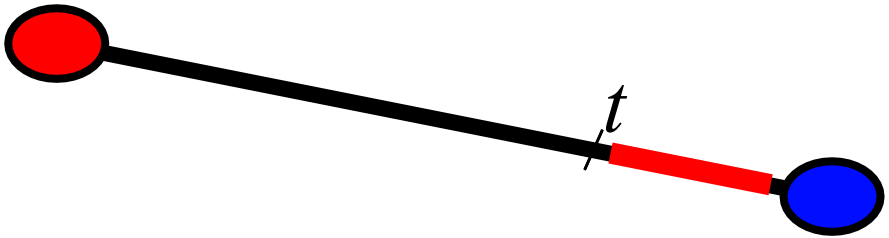


Barycentric Interpolation



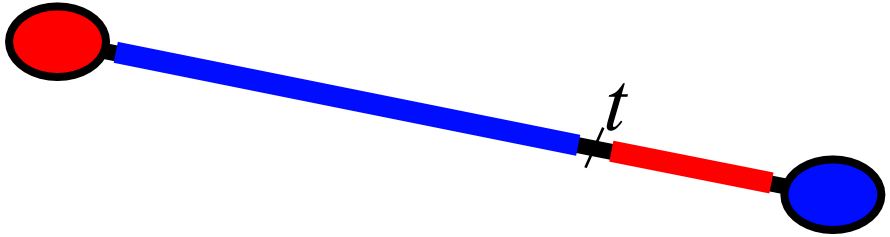
The further t is from the red point, the more blue we want.

Barycentric Interpolation



The further t is from the red point, the more blue we want. The further t is from the blue point, the more red we want.

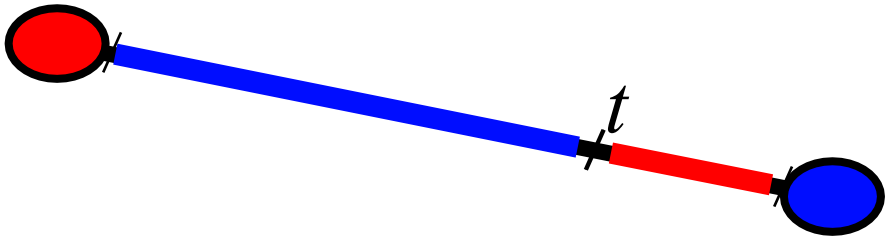
Barycentric Interpolation



The further t is from the red point, the more blue we want. The further t is from the blue point, the more red we want.

Percent blue = (length of blue segment)/(total length)

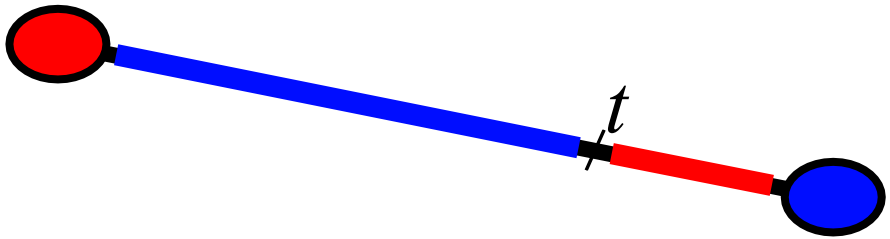
Barycentric Interpolation



The further t is from the red point, the more blue we want. The further t is from the blue point, the more red we want.

Percent blue = (length of blue segment)/(total length) Percent red = (length of red segment)/(total length)

Barycentric Interpolation

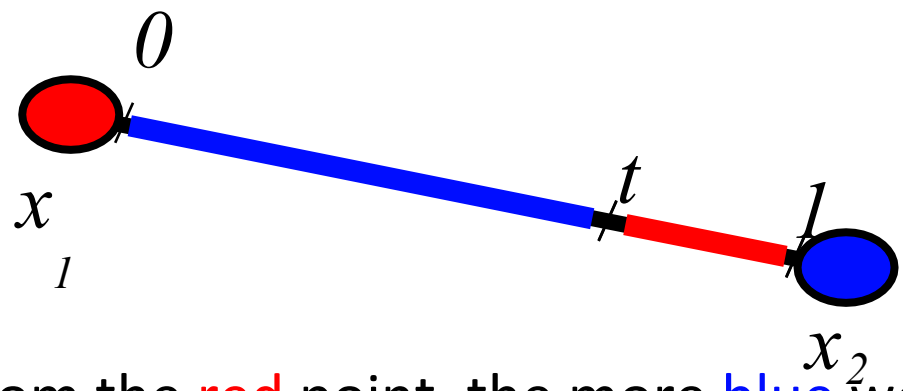


The further t is from the red point, the more blue we want. The further t is from the blue point, the more red we want.

Percent blue = (length of blue segment)/(total length) Percent red = (length of red segment)/(total length)

Value at t = (% blue)(value at blue) + (% red)(value at red)

Barycentric Interpolation



The further t is from the red point, the more blue we want. The further t is from the blue point, the more red we want.

Percent blue = t

Percent red = $1-t$

Value at t = $tx_1 + (1-t)x_2$

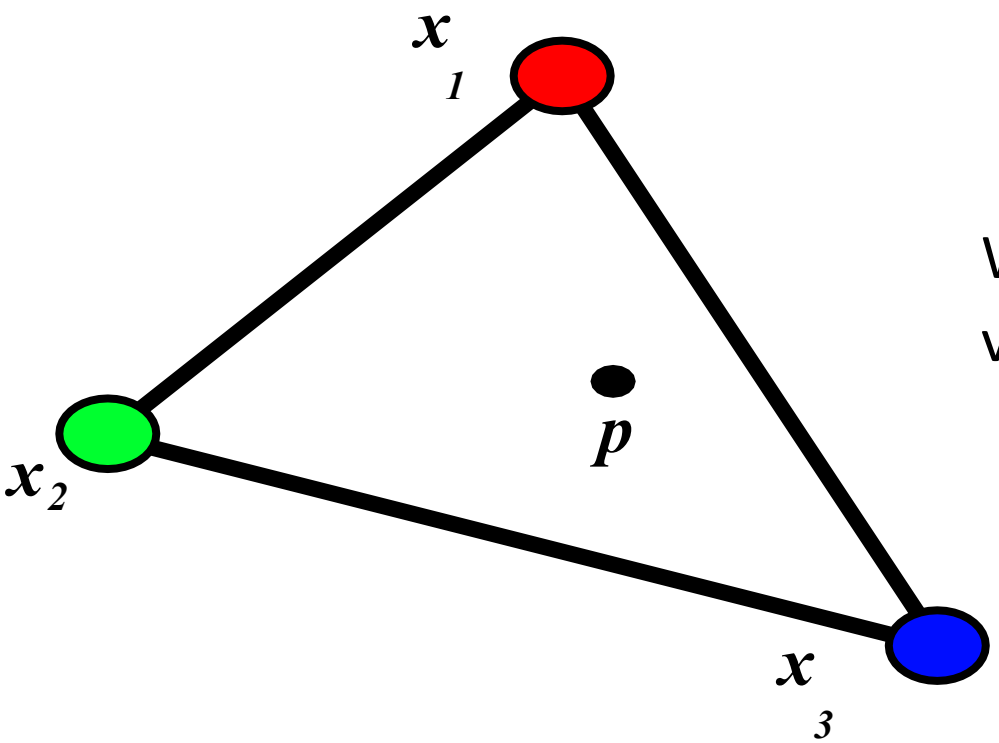
Barycentric Interpolation

Now what about triangles?

Barycentric Interpolation

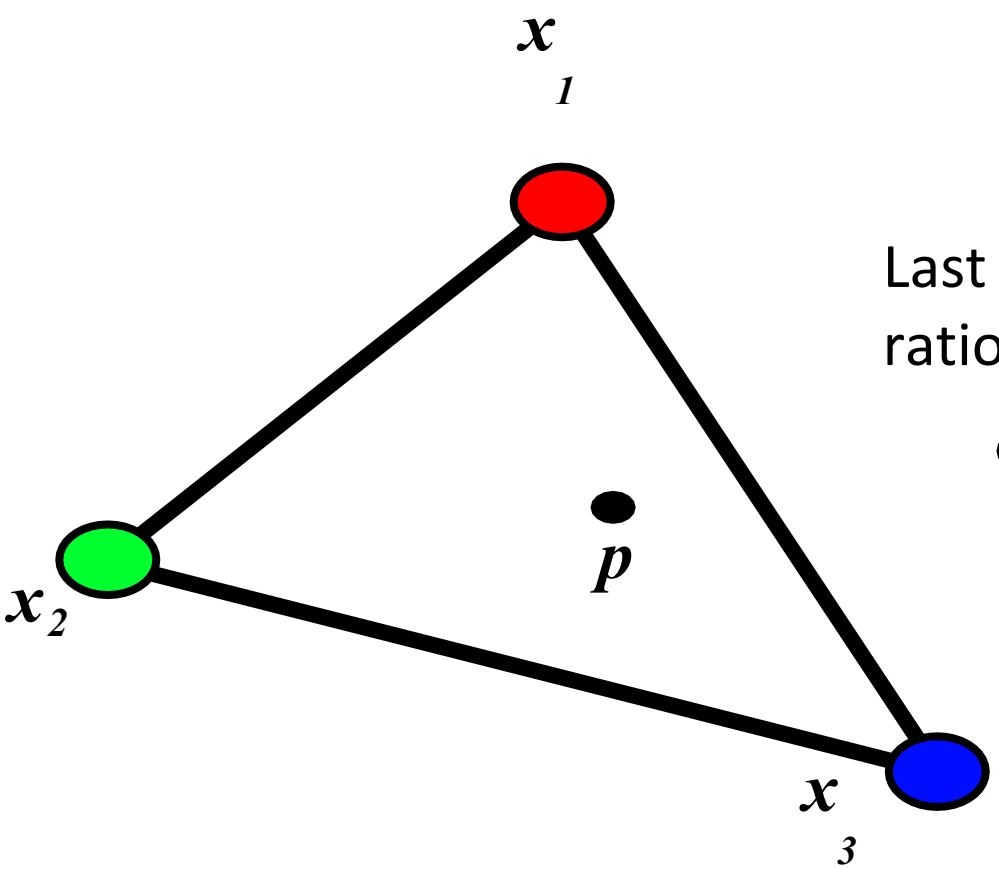
Now what about triangles?

Just consider the geometry:

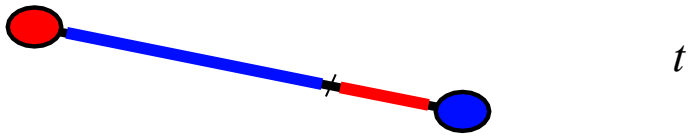


What's the interpolated value at the point $\boldsymbol{p}$?

Barycentric Interpolation

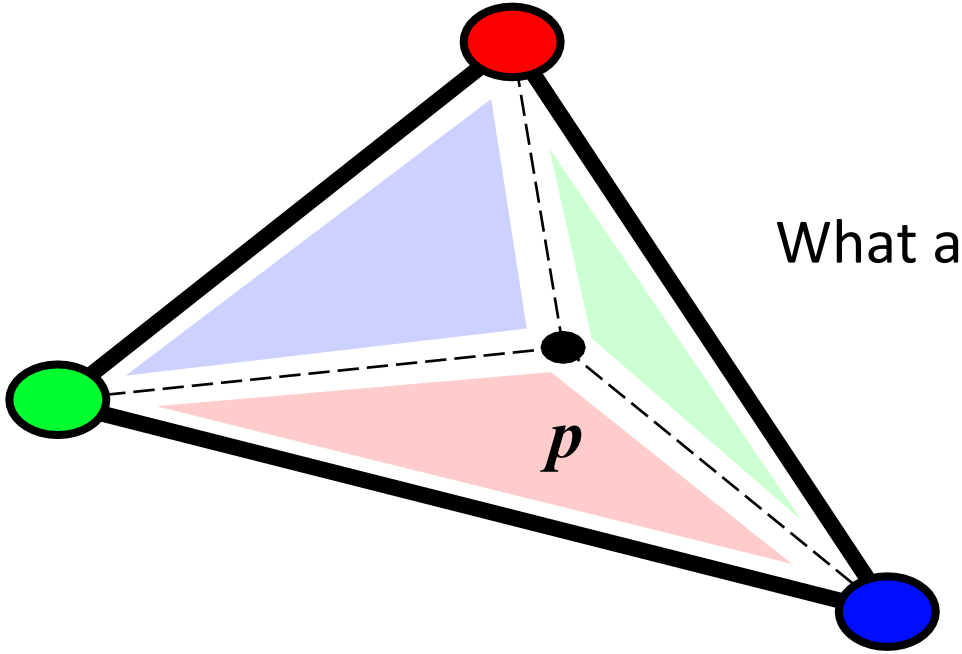


Last time (in 1D) we used ratios of lengths.



Barycentric Interpolation

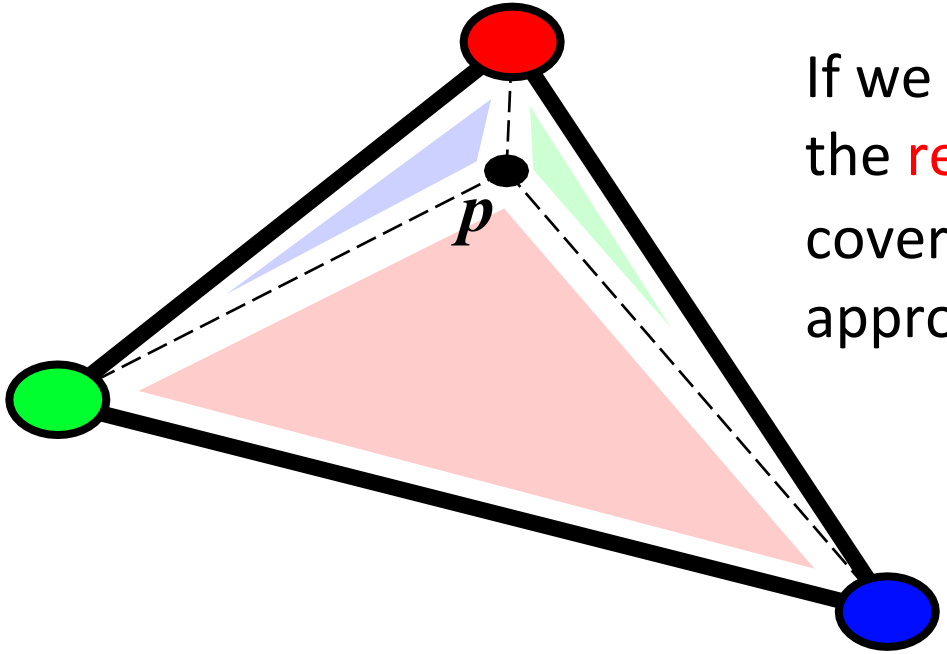
Now what about triangles? Just consider the geometry:



What about ratios of areas (2D)?

Barycentric Interpolation

Now what about triangles? Just consider the geometry:



If we color the areas carefully, the red area (for example) covers more of the triangle as $\boldsymbol{p}$ approaches the red point.

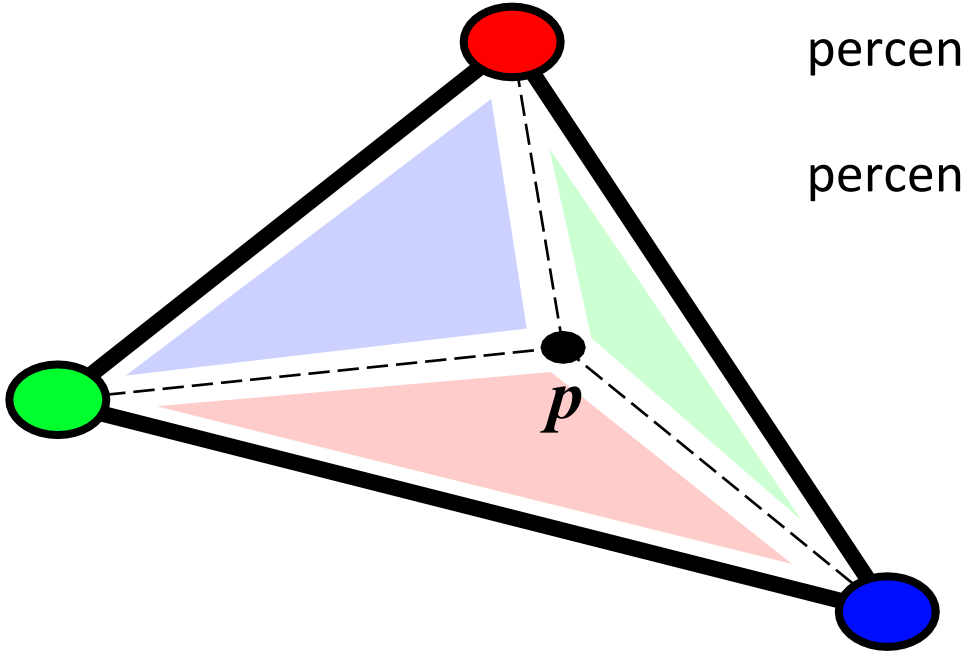
Barycentric Interpolation

Just like before:

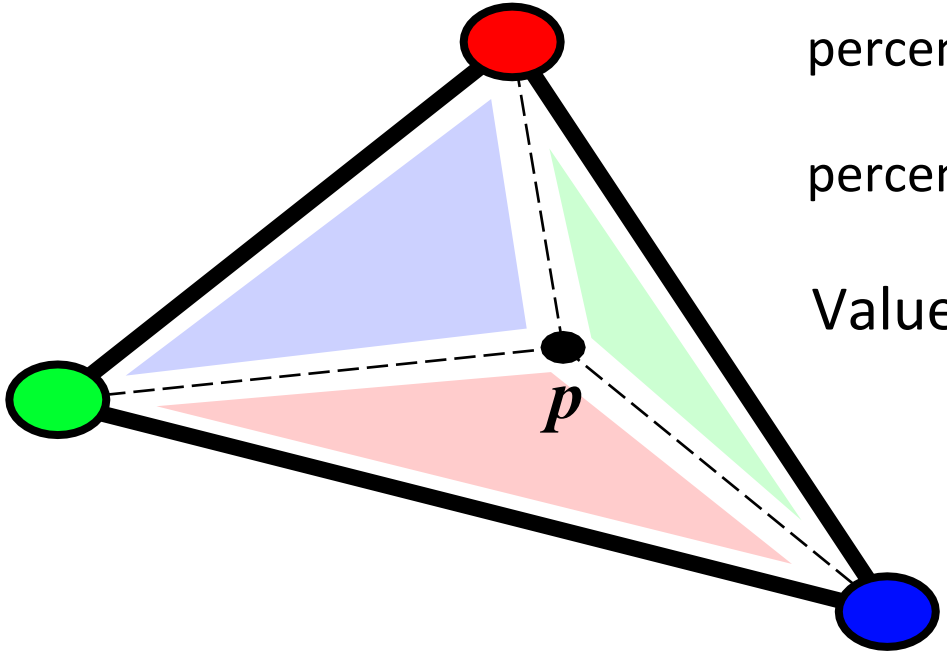
$$\text{percent red} = \frac{\text{area of red triangle}}{\text{total area}}$$

$$\text{percent green} = \frac{\text{area of green triangle}}{\text{total area}}$$

$$\text{percent blue} = \frac{\text{area of blue triangle}}{\text{total area}}$$



Barycentric Interpolation



Just like before:

$$\text{percent red} = \frac{\text{area of red triangle}}{\text{total area}}$$

$$\text{percent green} = \frac{\text{area of green triangle}}{\text{total area}}$$

$$\text{percent blue} = \frac{\text{area of blue triangle}}{\text{total area}}$$

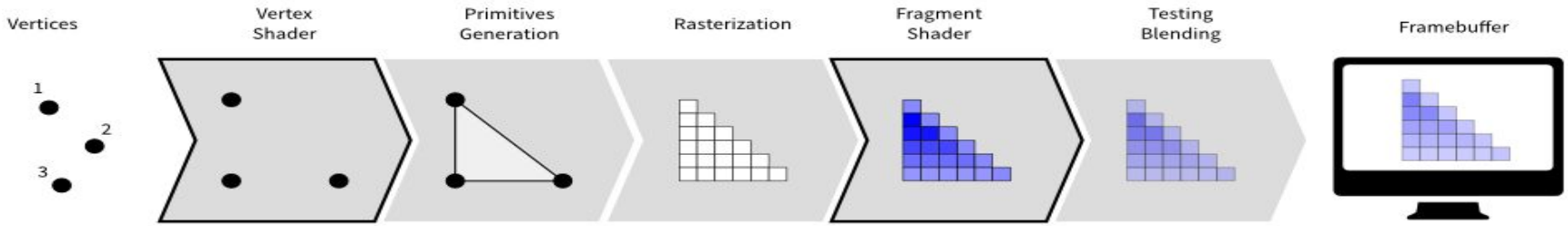
Value at p:

(% red)(value at red) + (% green)(value at green) + (% blue)(value at blue)

A Fragment shader samples color-relevant data <u>in between</u> the three vertices that have the data, using interpolation.

- Every variable in the GLSL programs that has the qualifier "**varying**" is received by the fragment shader as already interpolated from the three extreme vertex points' values for that variable, weighted according to the fragment's position in the triangle (barycentric coordinates)

The Process


Vertices
1
2
3
Vertex Shader
Primitives Generation
Rasterization
Fragment Shader
Testing Blending
Framebuffer

Part IV: Flat vs. smooth shading

The decision of whether or not triangles should share vertices

Normals

- Mathematics: A vector that is perpendicular to the surface at a given point -- particularly, pointing outward
- Graphics: We're only interested in vertex normals -- discrete "approximations" of the normal sampled at points on the imaginary surface
  - True realism would require calculation of normal/derivative at every point along a continuous shape - not feasible

What are they used for?

- Lighting! The direction of the normal determines how the light will bounce off each surface when modeling the rays

Our shapes so far have easy normal vectors.

- “Z axis” vector is perpendicular to Triangle and Square
- For a cube, normals would also just be axis-aligned
- For a sphere, we know analytically that the vector away from the center (perpendicular to the formula’s surface) will be the normal. Equals position coord.

What do you do when the normals aren’t known?

- Hint: Think per-triangle.

What do you do when the normals aren’t known?

- Using the indices, collect the positions of the three points of the triangle.
- Create two vectors out of the triangle.
- Cross product
- What to do if the normal points inside the shape instead of out?
- How to detect that? Assume shape is convex.

Flat vs Smooth shading

- That algorithm produces a normal per whole triangle
  - Distribute it to all three vertices identically
  - Result is flat lighting across the triangle
- When vertices are shared, we logistically cannot do that
  - A vertex can’t have two normals at once
  - Other triangles will fight over assigning normals to a vertex

Flat vs Smooth Shading

- As opposed to “Smooth shading”
  - Deriving normals from some source besides the triangle, like the analytic formula of the shape (like Sphere)
  - Lighting formulas can react to the normal being different across a surface
  - Appearance: “morph normals” demo in http://threejs.org/examples/

Flat vs Smooth Shading

Review:

- Flat shading:
  - All three normals are the same.
- Smooth Shading:
  - All three normals are not the same.

Flat vs Smooth Shading

- Smooth Shading:
  - All three normals are not the same.
    - Allows interpolation of infinity different normals as you move from one extreme point of the triangle to the other
    - This creates a different lighting formula estimate and different color at each pixel
    - Interpolation done in fragment shader (per pixel in triangle), and fed into the lighting formula

Flat vs Smooth Shading

- Flat shading
  - Nothing to interpolate. All three extreme points hold the same value. That value gets spread across whole triangle, giving the same color result everywhere.

But What If…?

- What if we already <u>did</u> share vertices when we built the shape, perhaps for simplicity?
  - Loop through all the index triples and create a new, unique, vertex for every index -- based on what used to be at that index.
  - Overwrite our old lists with this.

Part V: Specifying shapes from scratch with polygons

“example-shapes.js”

…But wait -

- You can begin Project 2 before you understand making your required custom polygons
  - Just use **primitives**
  - They serve as placeholders before you have custom shapes for every object
  - Overlap enough of them and you can already have a complex shape:

Star example from example-scenes

- A complex design made only from primitives
- The secret: dependence on loop indices

```
for( var j = 0; j < 20; j++ )
for( var i = 0; i < 20; i++ )
  { var model_transform =                        rotation   ( j * 18 *                  t/60  , 0, 0, 1   );
        model_transform = mult( model_transform, rotation   ( i * 18 * Math.sin(        t/21 ), 0, 1, 0 ) );
        model_transform = mult( model_transform, translation( 2 * i  * Math.sin(        t/31 ), 0, 0    ) );
        model_transform = mult( model_transform, scale      ( 1,  1  + Math.sin( i*18 * t/41 ), 1       ) );

        … Do the primitive draw call here
  }
```

Custom Polygons

Now let’s look at what goes into eventually making a unique shape.

OpenGL Implementations

- How does a javascript program display WebGL?
- Some standard setup functions got called upon creation of our objects:
  - Step 1: getContext() function of HTML canvases
    - (in our “Canvas_Manager” constructor)
  - Step 2: Tell the card to receive and compile your shaders
    - (in our “Shader” constructor)
  - Step 3: Specification of vertices (their positions, etc.), and sending over this vertex data (this is our coding job for shapes)

Vertex Specification

- We make arrays, then pass to the card
- Before that, our class Shape() holds these arrays
- Shape() superclass already knows Step 3 (“pass to the card”)
  - Let’s make a subclass:

Vertex Specification

- First, the simplest possible Shape – one triangle.
- 3 vertices, each having their own
  - 3D position
  - 3D normal vector
  - 2D texture-space coordinate
    - Vertices exist in two places at once in a way -- one in the 3D world and one in the image

```
// *********** TRIANGLE ***********
Declare_Any_Class( "Triangle",    // First, the simplest possible Shape – one triangle.  It has 3 vertices, each
 { 'populate'()                   // having their own 3D position, 3D normal vector, and 2D texture-space coordinate.
   { this.positions      = [ vec3(0,0,0), vec3(1,0,0), vec3(0,1,0) ];   // Specify the 3 vertices -- the point cloud that our Triangle needs.
     this.normals        = [ vec3(0,0,1), vec3(0,0,1), vec3(0,0,1) ];   // ...
     this.texture_coords = [ vec2(0,0),   vec2(1,0),   vec2(0,1)   ];  // ...
     this.indices        = [ 0, 1, 2 ];                   // Index into our vertices to connect them into a whole Triangle.
   }            // End of our class Triangle's list of methods.
 }, Shape )     // Notice we make Triangle a subclass of Shape.
```

How to use your Shape

- Put in your scene's constructor:

  this.submit_shapes(context, { 'funnyName' : new Triangle());

- Put in your scene's display():

  this.shapes.funnyName.draw(this.graphics_state, model_transform, …);

- Replace "…" with a result of calling a material() function
  - Just paste from the examples

```
// *********** SQUARE ***********
Declare_Any_Class( "Square",    // A square, demonstrating shared vertices.  On any planar surface, the interior edges don't make any important seams.
 { 'populate'()                  // In these cases there's no reason not to re-use data of the common vertices between triangles.  This makes all the
   {                             // vertex arrays (position, normals, etc) smaller and more cache friendly.
     this.positions        .push( vec3(-1,-1,0), vec3(1,-1,0), vec3(-1,1,0), vec3(1,1,0) ); // Specify the 4 vertices -- the point cloud that our Square needs.
     this.normals          .push( vec3(0,0,1), vec3(0,0,1), vec3(0,0,1), vec3(0,0,1) );     // …
     this.texture_coords.push( vec2(0,0),   vec2(1,0),   vec2(0,1),   vec2(1,1)   );         // ...
     this.indices      .push( 0, 1, 2,     1, 3, 2 );                    // Two triangles this time, indexing into four distinct vertices.
   }
 }, Shape )
```

What happens when you instantiate a shape?

Magical WebGL calls, wrapped inside Shape class's method 'copy_onto_graphics_card()'.

- Uses **gl.genBuffers()** to request buffers from the graphics card (and store the memory address of each in our Shape object – just ints)
- Uses **gl.BindBuffer()** to select a buffer as the current one being talked about
- Turns vectors into flat arrays (flatten()) and then calls **gl.bufferData()**
- Two buffers: **gl.ARRAY_BUFFER** and **gl.ELEMENT_ARRAY_BUFFER**

Part VI: Making a flat shaded shape - Tetrahedron

```
// *********** 1. SMOOTH TETRAHEDRON ***********
Declare_Any_Class( "Tetrahedron",          // A demo of flat vs smooth shading.  Also our first 3D, non-planar shape.
 { 'populate'()
    {
        var a = 1/Math.sqrt(3);




        // Method 1:  A tetrahedron with shared vertices.  Compact, performs better,
        // but can't produce flat shading or discontinuous seams in textures.

        this.positions          .push( [ 0, 0, 0], [1,0,0], [0,1,0], [0,0,1] );
        this.normals            .push( [-a,-a,-a], [1,0,0], [0,1,0], [0,0,1] );
        this.texture_coords.push( [ 0, 0     ], [1,0    ], [0,1,  ], [1,1   ] );
        this.indices            .push( 0, 1, 2,   0, 1, 3,   0, 2, 3,    1, 2, 3 );  // Vertices are shared multiple times with this method.

    }
 }, Shape )
```

```
// *********** 2. FLAT TETRAHEDRON ***********          Substitute into the previous code

this.positions.push( [0,0,0], [1,0,0], [0,1,0],        // Method 2:  A tetrahedron with
                     [0,0,0), [1,0,0], [0,0,1],        // four independent triangles.
                     [0,0,0], [0,1,0], [0,0,1],
                     [0,0,1], [1,0,0], [0,1,0] );

    this.normals.push( [0,0,-1], [0,0,-1], [0,0,-1],       // This here makes Method 2 flat shaded, since
                       [0,-1,0], [0,-1,0], [0,-1,0],       // normal values can be constant per-triangle.
                       [-1,0,0], [-1,0,0], [-1,0,0],       // Repeat them for all three vertices.
                       [ a,a,a], [ a,a,a], [ a,a,a] );

    this.texture_coords.push( [0,0], [1,0], [1,0],    // Each face in Method 2 also gets its own set of texture coords
                              [0,0], [1,0], [1,0],    //(half the image is mapped onto each face).  We couldn't do this
                              [0,0], [1,0], [1,0],    // with shared vertices since this features abrupt transitions
                              [0,0], [1,0], [1,0] );  // when approaching the same point from different directions.

    this.indices.push( 0, 1, 2,    3, 4, 5,    6, 7, 8,    9, 10, 11 );      // Notice all vertices are unique this time.
```

Part VII: Better Shapes

Beyond Tetrahedrons

A complicated shape: Windmill

Make non-trivial structures using:

- Transformation matrices on points
- Loops
  - Dependence on loop indices

```
// *********** WINDMILL ***********
Declare_Any_Class( "Windmill",    // As our shapes get more complicated, we begin using matrices and flow control
 { 'populate'( num_blades )       // control (including loops) to generate non-trivial point clouds and connect them.
   {
     for( var i = 0; i < num_blades; i++ )    // A loop to automatically generate the triangles.
     {
       var spin = rotation( i * 360/num_blades, [ 0, 1, 0 ] );      // Rotate around a few degrees in XZ plane to place each new point.
       var newPoint  = mult_vec( spin, [ 1, 0, 0, 1 ] );            // Apply that XZ rotation matrix to point (1,0,0) of the base triangle.
       this.positions.push( [ newPoint[0], 0, newPoint[2] ],        // Store this XZ position.                    This is point 1.
                            [ newPoint[0], 1, newPoint[2] ],        // Store it again but with higher y coord:  This is point 2.
                            [ 0, 0, 0                          ] ); // All triangles touch this location.         This is point 3.

       var newNormal = mult_vec( spin, [ 0, 0, 1, 0 ] ).slice(0,3);          // Rotate our base triangle's normal (0,0,1) to get the new one.  Careful!
                                                                             // Normal vectors are not points; their perpendicularity constraint gives them
                                                                             // a mathematical quirk that when applying matrices you have to apply the
       this.normals          .push( newNormal, newNormal, newNormal );       // transposed inverse of that matrix instead.  But right now we've got a pure
       this.texture_coords.push( [ 0, 0 ], [ 0, 1 ],   [ 1, 0 ] );          // rotation matrix, where the inverse and transpose operations cancel out.
       this.indices          .push( 3*i,      3*i + 1,  3*i + 2 );  // Procedurally connect the three
     }                                                              // new vertices into triangles.
   }
 }, Shape )
```

Closed Shapes

- That shape is pretty but a Windmill's geometry is not closed.
- Challenge: Make a closed, solid shape using those advanced practices.

- Surface_Of_Revolution() in your code generates common closed shapes
  - Produces a curved "sheet" of triangles with rows and columns.
    - Begin with an input array of points, defining a 1D path curving through 3D space
    - Imagine each point is a row.
    - Sweep that whole curve around the Z axis in equal steps, stopping and storing new points along the way; imagine each step is a column.
    - Now we have a flexible "generalized cylinder" spanning an area

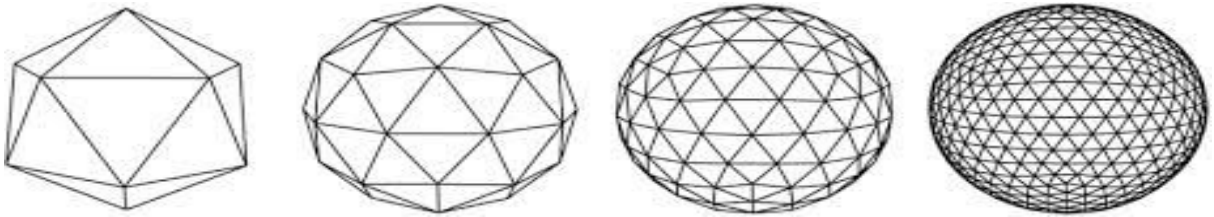
Part VIII: How to Sphere?

https://en.wikipedia.org/wiki/Subdivision_surface

Subdivision Surfaces

- Building our sphere shape
  - We know we want a lot of connected triangles around the origin. Norm = 1 for each point.
  - Simplest case: a tetrahedron

  - Next simplest: Split each tetrahedron face into 4 triangles, by connecting edge midpoints
    - Finally, force all new points to have norm=1, pushing points outward to the shape of a sphere

Subdivision Surfaces

- Result: