Specifying shapes from scratch with polygons

Custom Polygons

Let’s look at what goes into making a unique shape.

OpenGL Implementations

- First, review how a javascript program displays WebGL.
- Some standard setup functions got called upon starting our program:
  - 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 has a method that does the final step (“pass arrays 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 ***********
class Triangle extends Shape    // First, the simplest possible Shape – one triangle.  It has 3 vertices, each
{ constructor()                 // having their own 3D position, normal vector, and texture-space coordinate.
    { super();
      this.positions      = [ Vec.of(0,0,0), Vec.of(1,0,0), Vec.of(0,1,0) ];   // Specify the 3 vertices -- the point cloud that our Triangle needs.
      this.normals        = [ Vec.of(0,0,1), Vec.of(0,0,1), Vec.of(0,0,1) ];   // ...
      this.texture_coords = [ Vec.of(0,0),   Vec.of(1,0),   Vec.of(0,1)   ];   // ...
      this.indices        = [ 0, 1, 2 ];                                       // Index into our vertices to connect them into a whole Triangle.
    }
}
```

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 example materials in Tutorial_Animation

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

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 II: Making a flat shaded shape - Tetrahedron

```
// *********** 1. SMOOTH TETRAHEDRON ***********
class Tetrahedron extends Shape          // A demo of flat vs smooth shading.  Also our first 3D, non-planar shape.
  { constructor( using_flat_shading )
      { super();
        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( ...Vec.cast( [ 0, 0, 0], [1,0,0], [0,1,0], [0,0,1] ) );
          this.normals            .push( ...Vec.cast( [-a,-a,-a], [1,0,0], [0,1,0], [0,0,1] ) );
          this.texture_coords.push( ...Vec.cast( [ 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( ...Vec.cast( [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( ...Vec.cast( [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( ...Vec.cast( [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 III: Better Shapes

Beyond Tetrahedrons

A complicated shape: Windmill

Make non-trivial structures using:

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

```
// *********** WINDMILL ***********
class Windmill extends Shape    // As our shapes get more complicated, we begin using matrices and flow control
{ constructor( num_blades )     // control (including loops) to generate non-trivial point clouds and connect them.
   {  super();
      for( var i = 0; i < num_blades; i++ )     // A loop to automatically generate the triangles.
      {
        var spin = Mat4.rotation( i * 2*Math.PI/num_blades, Vec.of( 0,1,0 ) );          // Rotate around a few degrees in XZ plane to place each new point.
        var newPoint  = spin.times( Vec.of( 1,0,0,1 ) ).to3();   // Apply that XZ rotation matrix to point (1,0,0) of the base triangle.
        this.positions.push( newPoint,                    // Store this XZ position.                This is point 1.
                             newPoint.plus( [ 0,1,0 ] ),       // Store it again but with higher y coord:  This is point 2.
                                       Vec.of( 0,0,0 )    );      // All triangles touch this location.       This is point 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 transposed inverse of that matrix instead.  But right
        // now we've got a pure rotation matrix, where the inverse and transpose operations cancel out.
        var newNormal = spin.times( Vec.of( 0,0,1 ).to4(0) ).to3();
        this.normals      .push( newNormal, newNormal, newNormal );
        this.texture_coords.push( ...Vec.cast( [ 0,0 ], [ 0,1 ], [ 1,0 ] ) );
        this.indices      .push( 3*i, 3*i + 1, 3*i + 2                  );        // Procedurally connect the three new vertices into triangles.
      }
   }
}
```

Part IV: Closed Shapes

Arbitrary sheets of points, curved until they are geometrically closed

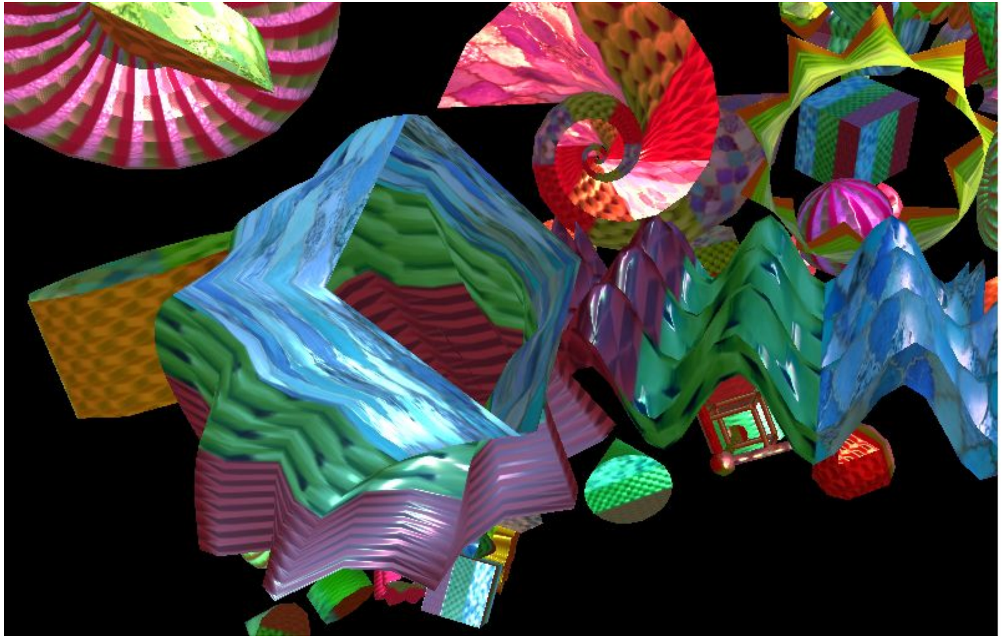
Closed Shapes

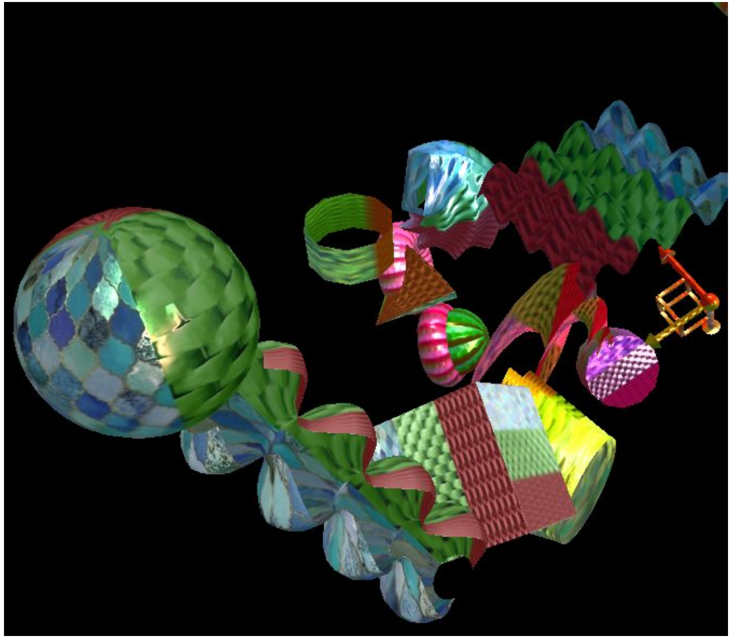
- Windmill is pretty but its geometry is not closed.
- Challenge: Make a closed, solid shape using those advanced practices.

- 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

Grid_Patch

- This class drives the most complex shapes in your “shapes upgrade” demo called Surfaces_Demo.





Grid_Patch

```
class Cylindrical_Tube extends Surface_Of_Revolution    // An open tube shape with equally sized sections, pointing down Z locally.
  { constructor( rows, columns, texture_range ) { super( rows, columns, [ ...Vec.cast( [1, 0, .5], [1, 0, -.5] ) ], texture_range ); } }

class Cone_Tip extends Surface_Of_Revolution          // Note:  Curves that touch the Z axis degenerate from squares into triangles as they sweep around
  { constructor( rows, columns, texture_range ) { super( rows, columns, [ ...Vec.cast( [0, 0, 1],  [1, 0, -1]  ) ], texture_range ); } }
```

- Most of these shapes are made using tiny code due to the help of two classes: Grid_Patch and Surface_Of_Revolution (a special case of Grid_Patch). Grid_Patch works by generating a tesselation of triangles arranged in rows and columns, and produes a deformed grid by doing user-defined steps to reach the next row or column.

Grid_Patch

- A cone and cylinder are among the simplest and most useful new shapes.

- Also available is a set of axis arrows that can be drawn anytime that you want to check where and how long your current coordinate axes are. Draw it in a neutral color with the "rgb.jpg" texture image and the axes will become identifiable by color - XYZ maps to red, green, blue.

Grid_Patch

- All of these shapes are generated as a single vertex array each. Building them that way, even with shapes like the axis arrows that are compounded together out of many shapes, speeds up your graphics program considerably.

Custom Shapes

- Grid_Patch is your most flexible class for making shapes. For Project 2, you will get more mileage out of it than anything else if you use it creatively.
  - All you need to provide is a functions for reaching the next row and column from the current one.
  - Your functions will receive from Grid_Patch arguments of (i, p, j) where:
    - i is the progress through the rows (from 0 to 1),
    - j is the progress through the columns, and
    - p is the previous row or column's point.

- Surface_Of_Revolution can also yield a complex shape satisfying Project 2's custom polygon requirement rather easily.

Custom Shapes

- Example: A custom bullet shape can be made by simply storing some points in an array that follow the outline of a bullet, and using Surface_Of_Revolution to sweep that curve around the Z axis.

  – To make the outline of a bullet, keep pushing points onto an array. To generate the round parts, either use:
    - Loops and trig, or better yet:
    - Keep a “temp” point that you keep incrementally applying matrices to, stepping along the curve you want and adjusting the transform depending on how far along the bullet you are.

Part V: How to make a good 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: