Lecture 6

SIGGRAPH trailers from 2013

Going backwards,

https://www.youtube.com/watch?v=FUGVF_eMeo4

And

https://www.youtube.com/watch?v=JAFhkdGtHck

Annoucements

- A1 grades up for real now
- A3 CODE is released on Piazza

- A3 directions release: Probably in the morning
- Let’s take a look

More Coding with Transforms

- Live demo: Stone arch

Making a flat shaded shape - Tetrahedron

```
class Tetrahedron extends Shape
{                                 // **Tetrahedron** demonstrates flat vs smooth shading (a boolean argument selects
                                  // which one).  It is also our first 3D, non-planar shape.  Four triangles share
                                  // corners with each other.  Unless we store duplicate points at each corner
                                  // (storing the same position at each, but different normal vectors), the lighting
                                  // will look "off".  To get crisp seams at the edges we need the repeats.
  constructor( using_flat_shading )
    { super( "position", "normal", "texture_coord" );
      var a = 1/Math.sqrt(3);
      if( !using_flat_shading )
      {                                       // Method 1:  A tetrahedron with shared vertices.  Compact, performs better,
                                              // but can't produce flat shading or discontinuous seams in textures.
          this.arrays.position      = Vec.cast( [ 0, 0, 0], [1,0,0], [0,1,0], [0,0,1] );
          this.arrays.normal        = Vec.cast( [-a,-a,-a], [1,0,0], [0,1,0], [0,0,1] );
          this.arrays.texture_coord = Vec.cast( [ 0, 0   ], [1,0  ], [0,1, ], [1,1  ] );
                                              // Notice the repeats in the index list.  Vertices are shared
                                              // and appear in multiple triangles with this method.
          this.indices.push( 0, 1, 2,   0, 1, 3,   0, 2, 3,   1, 2, 3 );
      }
      else
```

continued....

```
    else
    {                                                // Method 2:  A tetrahedron with four independent triangles.
      this.arrays.position = Vec.cast( [0,0,0], [1,0,0], [0,1,0],
                                       [0,0,0], [1,0,0], [0,0,1],
                                       [0,0,0], [0,1,0], [0,0,1],
                                       [0,0,1], [1,0,0], [0,1,0] );

                                        // The essence of flat shading:  This time, values of normal vectors can
                                        // be constant per whole triangle.  Repeat them for all three vertices.
      this.arrays.normal   = Vec.cast( [0,0,-1], [0,0,-1], [0,0,-1],
                                       [0,-1,0], [0,-1,0], [0,-1,0],
                                       [-1,0,0], [-1,0,0], [-1,0,0],
                                       [ a,a,a], [ a,a,a], [ a,a,a] );

                                        // Each face in Method 2 also gets its own set of texture coords (half the
                                        // image is mapped onto each face).  We couldn't do this with shared
                                        // vertices since this features abrupt transitions when approaching the
                                        // same point from different directions.
      this.arrays.texture_coord = Vec.cast( [0,0], [1,0], [1,1],
                                            [0,0], [1,0], [1,1],
                                            [0,0], [1,0], [1,1],
                                            [0,0], [1,0], [1,1] );
                                        // Notice all vertices are unique this time.
      this.indices.push( 0, 1, 2,    3, 4, 5,    6, 7, 8,    9, 10, 11 );
    }
  }
}
```

Complicated Shapes

Procedural Generation

A complicated shape: Windmill

Make non-trivial structures using:

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

```
class Windmill extends Shape
{                                 // **Windmill**  As our shapes get more complicated, we begin using matrices and flow
                                  // control (including loops) to generate non-trivial point clouds and connect them.
  constructor( num_blades )
    { super( "position", "normal", "texture_coord" );
                                                        // A for loop to automatically generate the triangles:
      for( let i = 0; i < num_blades; i++ )
        {                                          // Rotate around a few degrees in the XZ plane to place each new point:
          const spin = Mat4.rotation( i * 2*Math.PI/num_blades, Vec.of( 0,1,0 ) );
                                                   // Apply that XZ rotation matrix to point (1,0,0) of the base triangle.
          const newPoint  = spin.times( Vec.of( 1,0,0,1 ) ).to3();
          const triangle = [ newPoint,                      // Store that XZ position as point 1.
                             newPoint.plus( [ 0,1,0 ] ),    // Store it again but with higher y coord as point 2.
                             Vec.of( 0,0,0 )    ];          // All triangles touch this location -- point 3.

          this.arrays.position.push( ...triangle );
                        // 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, so it's ok.
          var newNormal = spin.times( Vec.of( 0,0,1 ).to4(0) ).to3();
                                                                          // Propagate the same normal to all three vertices:
          this.arrays.normal.push( newNormal, newNormal, newNormal );
          this.arrays.texture_coord.push( ...Vec.cast( [ 0,0 ], [ 0,1 ], [ 1,0 ] ) );
                                                                   // Procedurally connect the 3 new vertices into triangles:
          this.indices.push( 3*i, 3*i + 1, 3*i + 2 );
        }
    }
}
```

How to make a good Sphere?

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

Spheres

- We could define a sphere by specifying a grid of points in spherical coordinates of radius 1.
  – Rows and columns
- Problem: Spheres made that way have singularities at the North Pole and South Pole.
  – All the columns converge to one point
  – Triangles near the top are squeezed into needles
  – Inconsistent for shading, texturing, and simulation

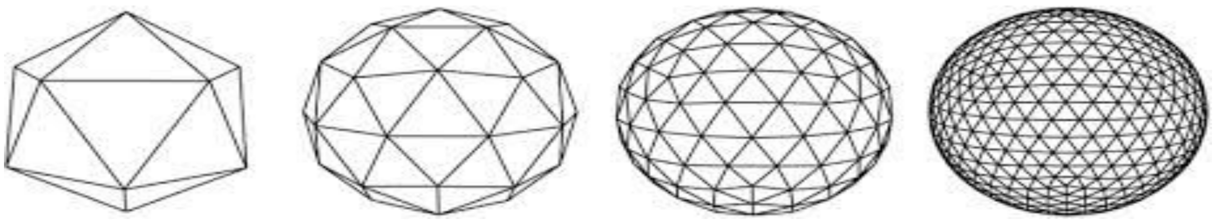
Subdivision Surfaces

- 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:



The Special Matrices

Camera, Projection, and Viewport

Main topics of today

- Positioning Cameras
- View Volumes
- Projections
- Perspective

Matrix Review

- All the objects you draw on screen are drawn one vertex at a time, by starting with the vertex's xyz coordinate and then multiplying by a matrix to get the final xy coordinate on the screen.

$$\underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}}_{M} * \underbrace{\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}}_{p}$$

Transforms

$$\underset{M}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}} * \underset{p}{\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}}$$

- Before that matrix, the xyz coordinate is always some trivial value like (.5, .5, .5)
  - In the reference system of the shape itself
  - For example, a cube's own coordinates for its corners
- After that matrix, it's some different xy pixel coordinate denoting where that vertex will show up on the screen.
  - And z for depth, and a fourth number for translations / perspective effects
- That mapping is all that the transform does.

Transform Process

- The transform is always just one 4x4 matrix.
- But calculating what it should be involves multiplying out a big chain of intermediate special matrices. That chain is always:

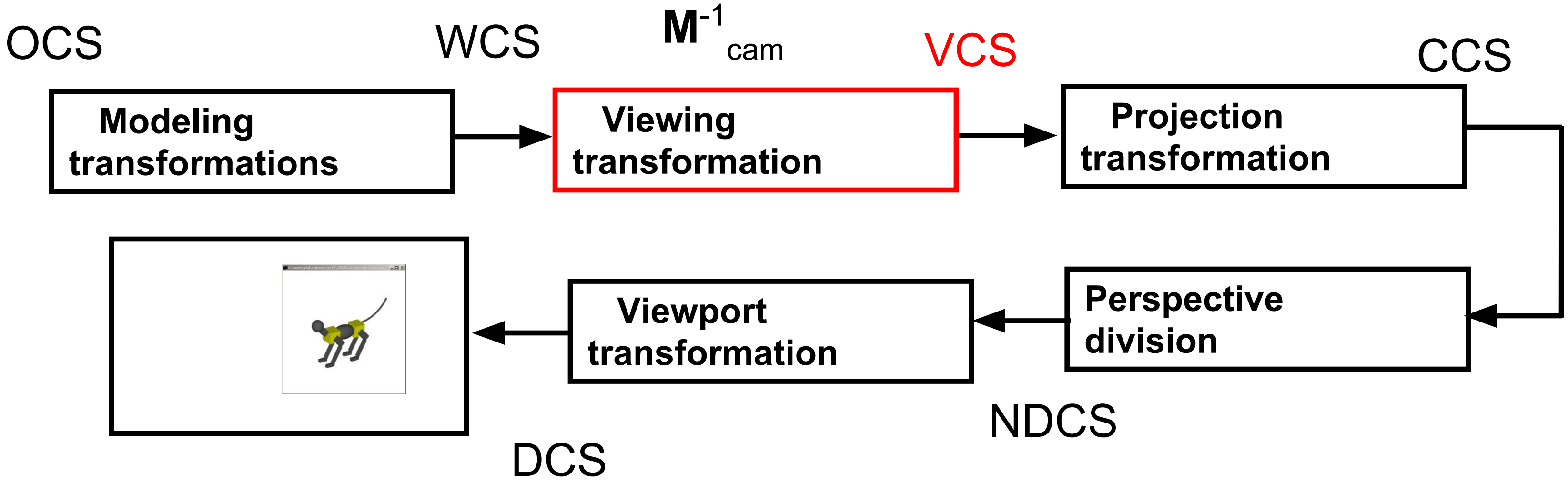
$$\underset{Viewport}{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Projection}{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}} * \underset{Camera}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Model}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Transform Process

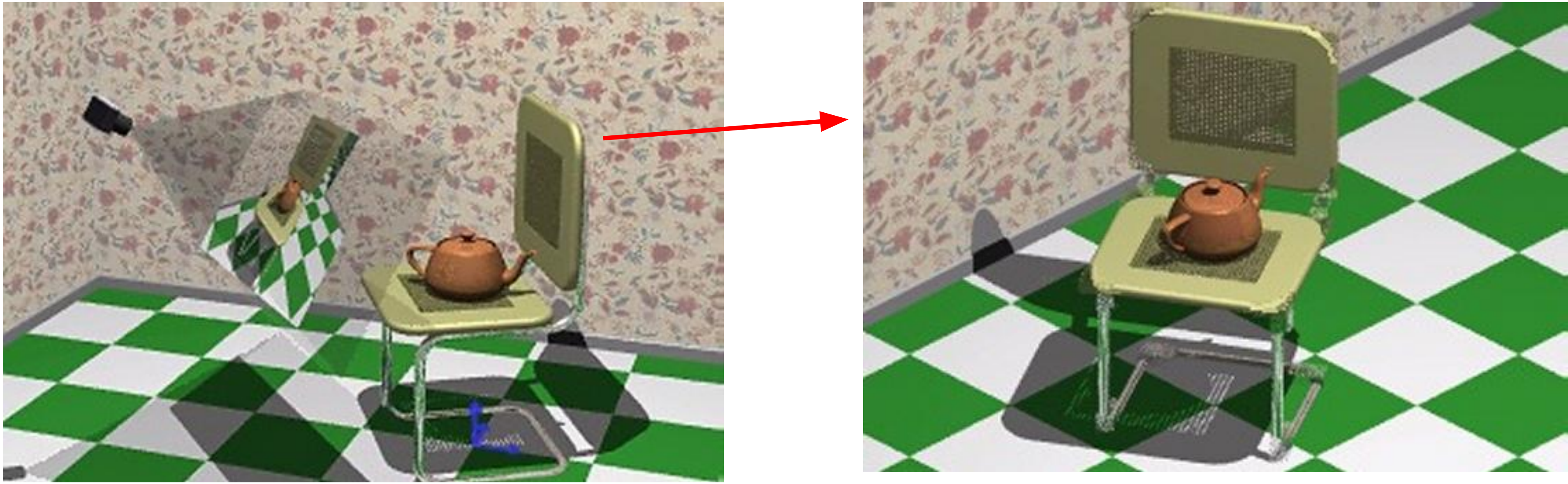
$$\underset{Viewport}{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Projection}{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}} * \underset{Camera}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Model}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Note: We never actually see the viewport matrix.
  - The viewport matrix is automatically applied for you at the end of the vertex shader.
  - Early during initialization, javascript set it up, calling gl.viewport(x,y,width,height).
- All the other special matrices you do manage.

Graphics Pipeline

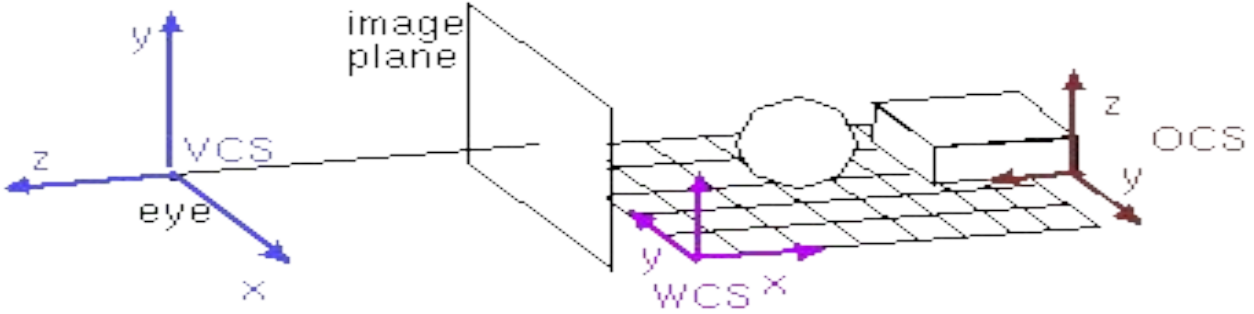


Rendering a 3D Scene From the Point of View of a Virtual Camera



Camera Transformation

Transforms objects to camera coordinates



$$\left.\begin{array}{l} P_{\mathrm{wcs}} = \mathrm{M}_{\mathrm{cam}} P_{\mathrm{vcs}} \rightarrow P_{\mathrm{vcs}} = \mathrm{M}_{\mathrm{cam}}^{-1} P_{\mathrm{wcs}} \\ P_{\mathrm{wcs}} = \mathrm{M}_{\mathrm{mod}} P_{\mathrm{ocs}} \end{array}\right\} \rightarrow$$

$$P_{\mathrm{vcs}} = \underbrace{\mathrm{M}_{\mathrm{cam}}^{-1} \mathrm{M}_{\mathrm{mod}} P_{\mathrm{ocs}}}_{\text{Modelview Transformation}}$$

Transform Process

$$\underset{Viewport}{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Projection}{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}} * \underset{Camera}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Model}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- The camera matrix is very much like the model transform matrix for placing shapes. But:
  - The shape being placed is the scene's observer
  - You actually use the **inverse matrix** of what you would have done to a 3D model of an actual camera