UCLA

CS174A : Introduction to Computer Graphics

Kinsey 1240
MW 4-6pm

Scott Friedman, Ph.D
UCLA Institute for Digital Research and Education

Term Project

- Details will be introduced next Monday.
- Things to start considering now are:
  - Teams will be a minimum of ***three*** people.
    - Use the class forum to find partners if you need to.
  - Teams can have up to ***five*** people in your group.
  - Your team will submit a one page project proposal.
    - "we are going to write a game", will not do for a proposal.
  - The proposals are due 2/10/2017 by midnight.
  - We will review proposals and make comments back to your group.

Depth Buffer

- When the depth buffer is enabled
  - A z value is written into buffer for every pixel.
  - If an incoming pixel has a z value less than the value already in the buffer
    - The pixel is written into the frame (color) buffer.
    - The z value is updated in the depth buffer.
  - Else
    - The pixel is rejected and not written to the frame buffer.
    - The depth buffer is not modified.

Depth Buffer

- What happens without it?
  - Last write to the frame buffer “wins”.
  - The order that objects are drawn then *matters*.
  - Objects have to be rendered back to front.
    - Why?
  - Potentially problematic cases
    - For inter-penetrating objects
    - Moving objects

Depth Buffer

- There is very little to do in order to use it
  - Enable the depth buffer.

```
function init()
{
    ...
    gl.enable( gl.DEPTH_TEST );
    ...
}
```

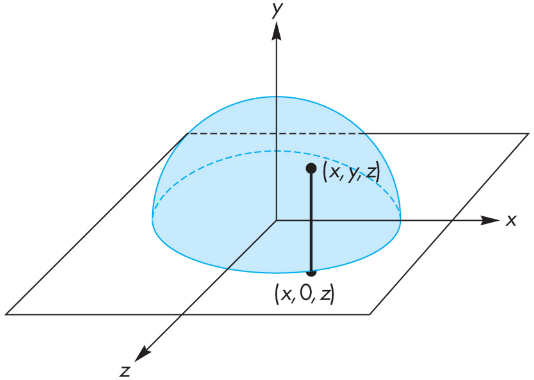
Depth Buffer

- Why do we need to enable?
  - OpenGL is a state machine, remember.
  - Also have to *clear (or reset)* the depth buffer when starting a new image.

```
function display()
{
    gl.viewport( ... );
    gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT );
    ...
}
```

Trackball

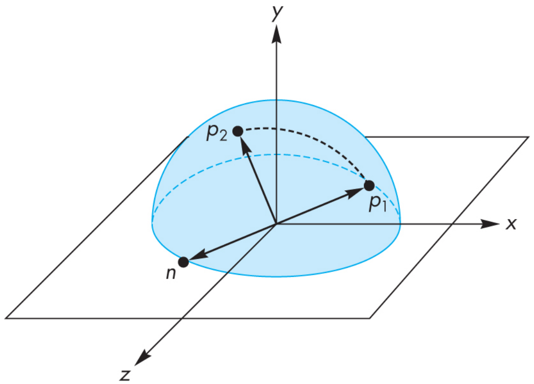
- Last time we talked about rotation…
  - A useful/intuitive interaction technique.
  - Plant a unit hemisphere into the x-z plane.
  - Using mouse positions we can reconstruct y
    - Because, $x^2 + y^2 + z^2 = 1$, $y = \sqrt{1 - x^2 - z^2}$



Trackball

- That's nice, we know *y*…
  - Now we can track, as the mouse moves, a position p_1 to p_2 on the surface of the hemisphere.
  - What we really want to do is rotate in the direction of the arc swept out from p_1 to p_2.
  - That axis of rotation can be found via the cross product of p_1 to p_2, the *normal*.

$$n = p_1 \times p_2$$



Trackball

- That's nice, we know n…
  - Conveniently, n can tell us the angle between p_1 to p_2 as well because we are using a *unit* hemisphere.

$$|\sin\theta| = |n|$$

  - Now we know an angle and a vector around which the rotation is supposed to occur.
  - The book mentions a nice trick when animating the rotation in small increments – and that is to recognize that for small
    - and you can avoid the inverse sine operation.

$$\theta \approx \sin\theta$$

Trackball

- A side note…
  - When animating a rotation in small increments
    - A lot of trigonometry is involved = slow
  - Helpful to recognize that for small $\theta \approx \sin\theta$
    - and then you can avoid the inverse sine operation.
  - This implies that, *for* ***small*** *angles* we can use
    - Another graphics "trick"

$$R = R_z(\psi)R_y(\phi)R_x(\theta) \approx \begin{bmatrix} 1 & -\psi & \phi & 0 \\ \psi & 1 & -\theta & 0 \\ -\phi & \theta & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Trackball

- That's nice, we know the *angle*…
  - But, we just know how to rotate around *x*, *y* and *z*!
  - Yes, but if we align the rotation vector with, say, the *z*-axis we perform our rotation. *Simple!* ☺

$$R = R_x(-\theta_x)R_y(-\theta_y)R_z(\theta_z)R_y(\theta_y)R_x(\theta_x)$$

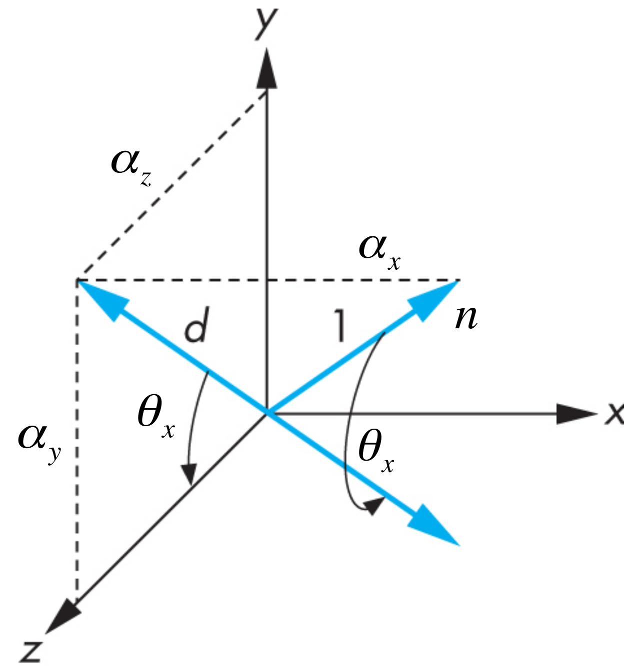
  - Ugh, we don't know $R_x(\theta_x)$ or $R_y(\theta_y)$ even if we decided that $R_z(\theta_z)$ was the rotation we wanted.
  - Yes, but let's understand what is going on first.

Trackball

- How can we find the *x* and *y* rotations?
  - First we need to rotate around the *x*-axis onto the *x*-*z* plane.
  - Recall that $\cos\theta_x = \alpha_x$
  - Then,

$$d = \sqrt{\alpha_y^2 + \alpha_z^2}$$

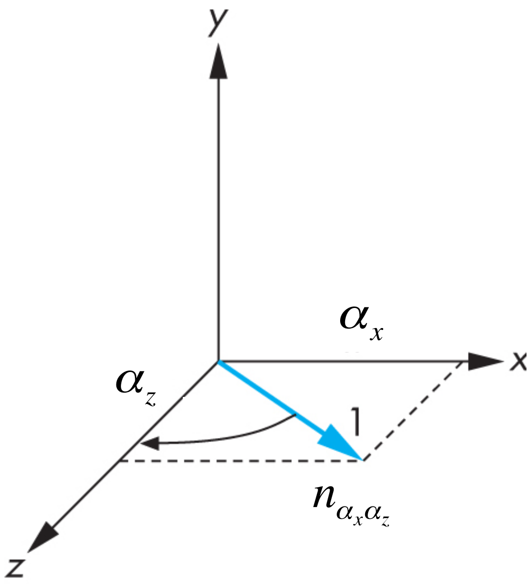
$$R_x(\theta_x) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \alpha_z / d & -\alpha_y / d & 0 \\ 0 & \alpha_y / d & \alpha_z / d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Trackball

- Now we need the *y* rotation?
  - We can follow the same process
  - Again, recall that $\cos\theta_y = \alpha_y$
  - Then,

$$R_y(\theta_y) = \begin{bmatrix} \alpha_z & 0 & -\alpha_x & 0 \\ 0 & 1 & 0 & 0 \\ \alpha_x & 0 & \alpha_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Trackball

- Actually, we can now rotate about *any* vector v fixed at a point p.
  - Concatenating into M

$$M = T(p)R_x(-\theta_x)R_y(-\theta_y)R_z(\theta_z)R_y(\theta_y)R_x(\theta_x)T(-p)$$

  - Now we can rotate our trackball vector!
  - Our point p is the **origin** and our rotation vector is n.

  - This is a lot of work – is there a better way?

Quaternions

- Same result with less computation.
  - A quaternion has the form $q=w+x\boldsymbol{i}+y\boldsymbol{j}+z\boldsymbol{k}$.
  - The terms $\boldsymbol{i}$, $\boldsymbol{j}$ and $\boldsymbol{k}$ are imaginary.
    - Fortunately, we can ignore this fact in this class.
    - But, they are what ultimately make quaternions work.
  - Lets consider them this way $q(w,x,y,z)$
    - Lets make w the rotation angle
    - Lets make x, y and z be the rotation vector.

Quaternions

- Same result with less computation.
  - It is ***critical*** that q be ***normalized***, *i.e.* $q=|q^2|$.
    - Or this *does. not. work.*
  - The resulting rotation matrix is

$$Q = \begin{bmatrix} 1-2(y^2+z^2) & 2(xy-wz) & 2(xz+wy) & 0 \\ 2(xy+wz) & 1-2(x^2+z^2) & 2(yz-wx) & 0 \\ 2(xz-wy) & 2(yz+wx) & 1-2(x^2+y^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Quaternions

- Same result with less computation.
  - A very nice feature of quaternions is that they allow for straightforward smooth interpolation.
    - You do this with a current rotation $\boldsymbol{R}$ and an increment $\boldsymbol{R_I}$
    - $\boldsymbol{R}$ starts with some initial/or no rotation and rotation vector
    - $\boldsymbol{R_I}$ has a small rotation and the same rotation vector.
    - Need to *renormalize* $\boldsymbol{R}$ occasionally to keep computation stable.

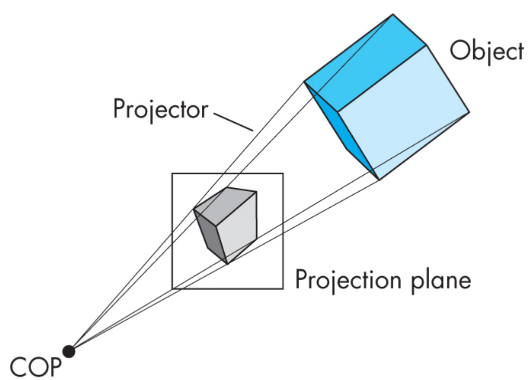
$$\boldsymbol{R=RR_I}$$

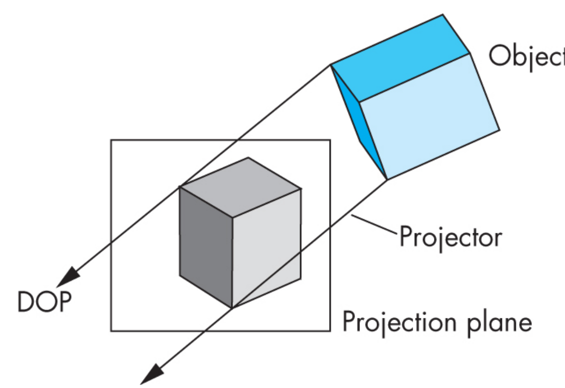
Viewing

- We are going to concern ourselves with two types of *viewing*.
  - Perspective viewing
  - Parallel viewing

Viewing

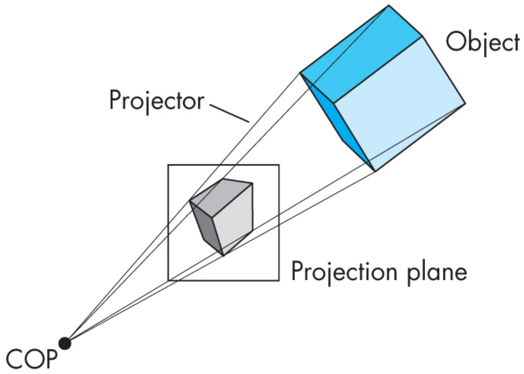
- In both cases we have
  - Objects,
  - Projection lines
  - Projection plane
  - Eye (COP: center of projection or DOP: direction of projection)

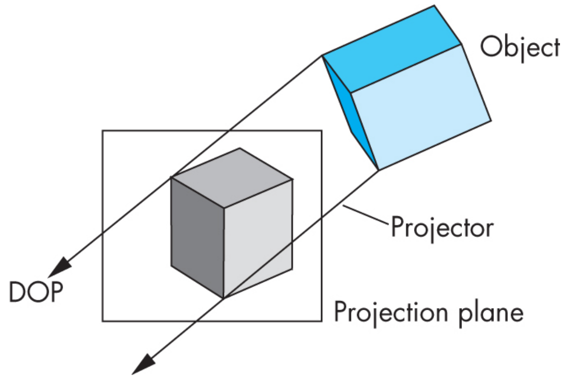




Viewing

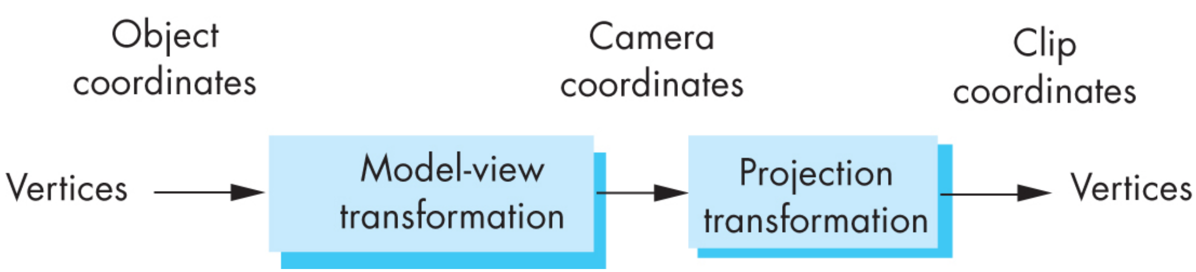
- Our goal is to ultimately
  - Use transformations to project the vertices of objects onto the projection plane.
  - Specifically we will create transformations to go from object space to world space to camera space to clip space.





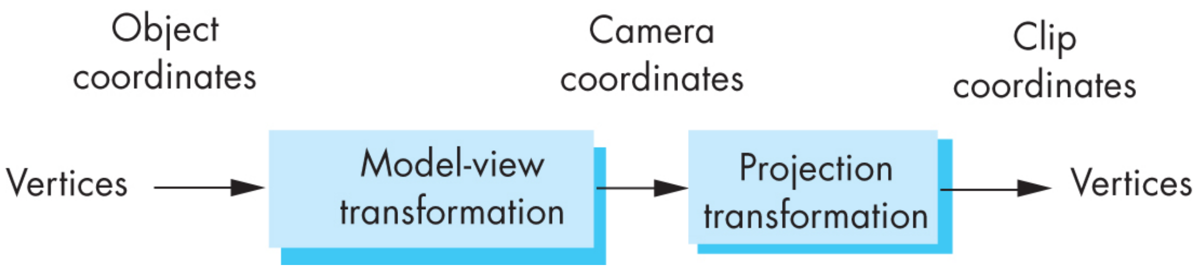
Viewing

- Previously
  - We used the default canonical view volume.
    - Which exists in clip space, i.e. (-1,1),(-1,1),(-1,1).
    - The ‘projections’ we implicit
  - Last time we saw how transformations can be combined to bring objects into camera (world) coordinates
    - Collectively, the ***model-view transformation.***
    - space == coordinates, terminology is equalivalent



Viewing

- Model-view transformation
    - Does not take us all the way to clip coordinates.
    - we need a ***projection transformation*** for that.
    - Model-view gets objects in front of the camera.
    - A Projection defines which and how and where those objects (verticies) will appear on the screen.



Instancing

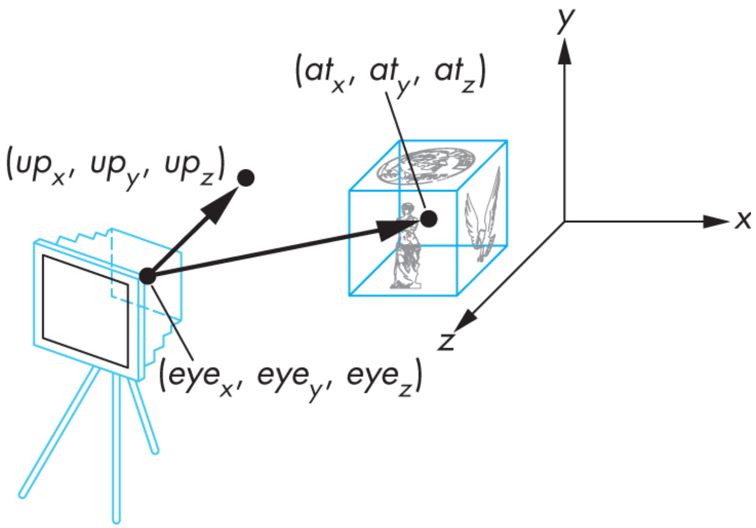
- Useful for Assignments and Project:
  - Take a single geometric object.
    - "Clone" it with only transformations (and possibly state).
  - Define the geometry of a cube (once).
    - Instance the cube by setting a transformation and setting some state (e.g. color) and drawing it.
    - Instance another cube by setting another transformation and setting state(e.g. color and scale) and drawing the *same geometry*.

Viewing

- Positioning the (getting things in from of) "camera"
  - Recall that the default is "looking" down the –*z* axis at the origin (0,0,0).
    - This is equivalent to model-view set to the identity matrix.
  - Remember, transformations are specified in *reverse*
    - in a post-multiplication world.
    - That means we specify the position of the camera *first*.
  - We are going to look at two methods
      - Look-at
      - Yaw, pitch and roll (euler angles)

Viewing

- Look-at
  - We define three terms
    - A point describing the location of the ***eye***.
    - A point the eye is looking ***at*** in the scene.
    - An ***up*** vector (direction) for the camera.



Viewing

- Look-at
  - The ***at*** and ***eye*** points give us
    - the *view-plane-normal* or ***vpn***
    - a vector perpendicular to the view plane
  - the ***up*** vector is usually (0, 1, 0)
    - Or, (0, 1, 0, 0) in homogeneous coordinates!
  - We then calculate the following

$$vpn = at - eye \qquad u = \frac{up \times n}{|up \times n|}$$

$$n = \frac{vpn}{|vpn|} \qquad v = \frac{n \times u}{|n \times u|}$$

Viewing

- Look-at
  - Once ***u*, *v*** and ***n*** are *normalized*
  - The following will position our camera

$$V = RT = \begin{bmatrix} u_x & u_y & u_z & -eye_x u_x - eye_y u_y - eye_z u_z \\ v_x & v_y & v_z & -eye_x v_x - eye_y v_y - eye_z v_z \\ n_x & n_y & n_z & -eye_x n_x - eye_y n_y - eye_z n_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

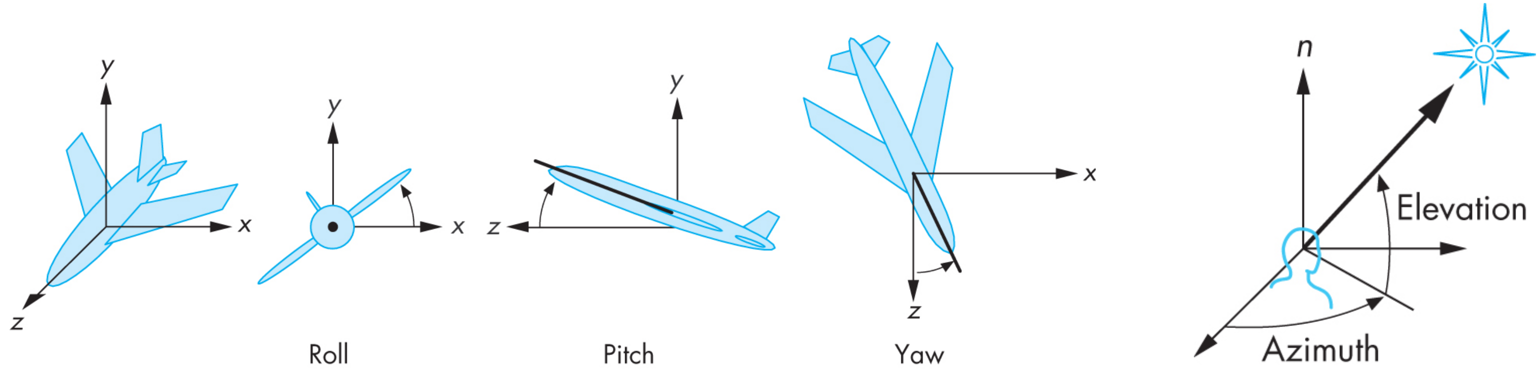
Viewing

- Look-at
  - Works reasonably well for positioning.
  - But not so well for moving the camera smoothly.
  - MV.js has a function for setting this up

$$V = RT = \begin{bmatrix} u_x & u_y & u_z & -eye_x u_x - eye_y u_y - eye_z u_z \\ v_x & v_y & v_z & -eye_x v_x - eye_y v_y - eye_z v_z \\ n_x & n_y & n_z & -eye_x n_x - eye_y n_y - eye_z n_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Viewing

- Yaw, pitch and roll (like an airplane)
  - Here we, essentially, use polar coordinates.
  - A simplified version uses just *azimuth* and *elevation.*
    - Rotate camera in the direction we desire.
    - Translate camera to ***eye*** point.



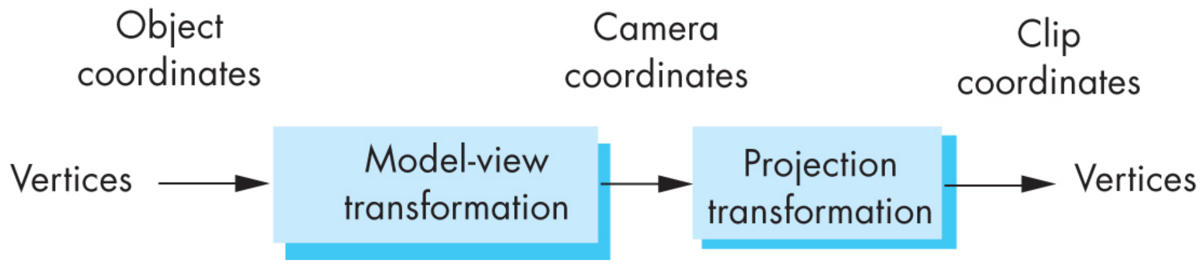
Viewing

- Yaw, pitch and roll
  - In reality we perform the *inverse* of what we want.
  - We are transforming world coordinates (all objects) into camera coordinates.
    - Move the world to the camera.
      - » That is, if I want to appear to rotate left 20 degrees.
      - » The transformation about the y-axis would be -20.
      - » Similarly, if I want to appear to move forward 5 units.
        - » I would transform everything by -5.
- So far we have only gotten things in front of the camera!
  - We now have both model and view transformations

Viewing

- Projections – Parallel (orthographic)
  - Once in camera coordinates we need a projection transformation to get us to clip coordinates.
  - The transformation matrix that gives us an orthographic projection is:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

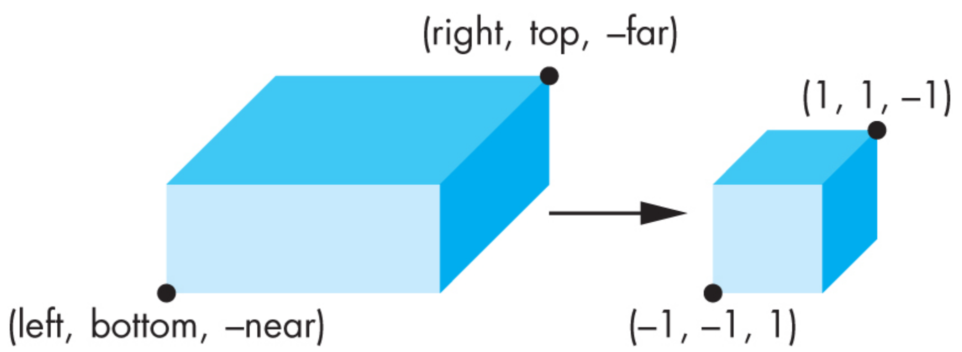


Viewing

- Projections – Parallel (orthographic)
  - However, ***M*** is applied by the hardware *after* the vertex shader.
    - Which gets our geometry into clip coordinates
  - How do we "include" or "see" more of our scene?

Viewing

- Projections – Parallel (orthographic)
  - We *scale* what we want to "include" to fit within the canonical view volume. i.e. (-1,1),(-1,1),(-1,1)
  - OpenGL provides a function for this called
    - `ortho(`*left, right, bottom, top, near, far*`) (ex: MV.js)`



Viewing

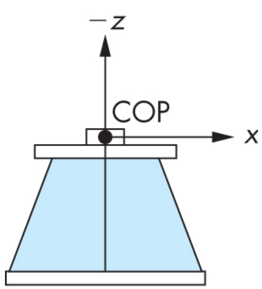
- Projections – Parallel (orthographic)
  - If you think about it `ortho` contains a scale and translation.
  - Here is what the transformation matrix looks like.
  - The translate moves everything relative to the origin (and the canonical view volume)
  - The scale (symmetrically) resizes everything to squeeze or stretch into the canonical view volume.

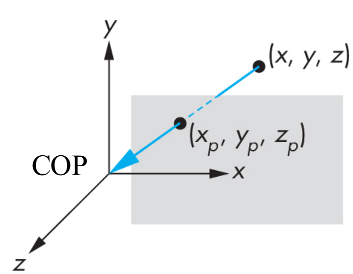
$$N = ST = \begin{bmatrix} \frac{2}{right - left} & 0 & 0 & -\frac{left + right}{right - left} \\ 0 & \frac{2}{top - bottom} & 0 & -\frac{top + bottom}{top - bottom} \\ 0 & 0 & -\frac{2}{far - near} & -\frac{far + near}{far - near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

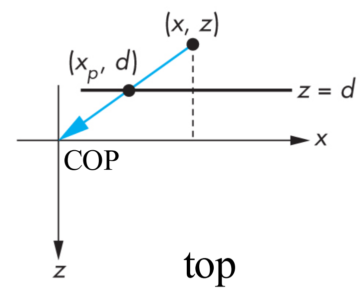
Viewing

- Projections – Perspective
  - Basic symmetrical perspective projection
  - The point (x, y, z) is projected through the projection plane to the eye point (or center of projection COP)
  - We can compute the point of intersection with

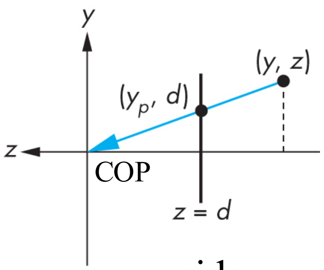
$$x_p = \frac{x}{z/d}, \quad y_p = \frac{y}{z/d}$$







top



side

Viewing

- Projections – Perspective
  - The simple perspective projection matrix is

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

  - The important thing to notice here is the position of the *-1* term.
  - This means our homogeneous coordinate, w, can be modified (will no longer be 1) when a vertex is multiplied by ***M***.

Viewing

- Projections – Perspective
  - Uh oh, the homogeneous coordinate is no longer 1?

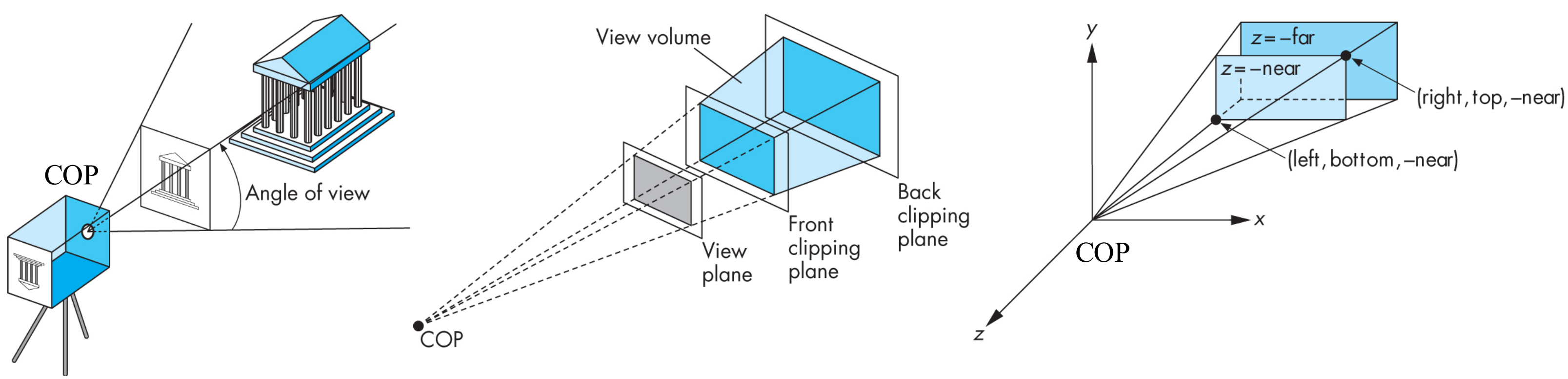
$$q = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

  - Not the end of the world, remember
    - We have to divide by the homogeneous coordinate to get back to 3D space.

$$q' = \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix}$$

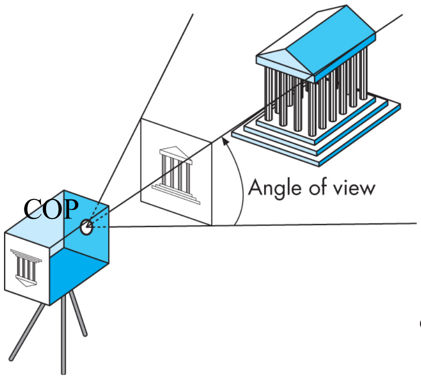
Viewing

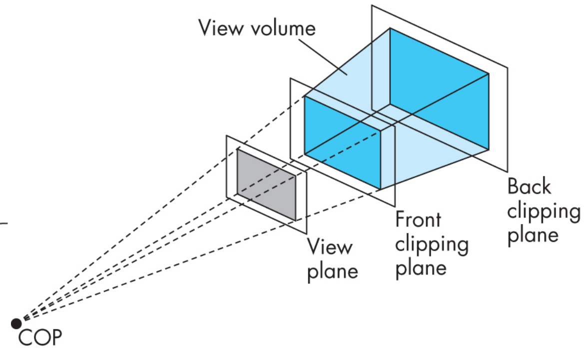
- Projections – Perspective
  - That's nice, but only gets our points onto the projection plane.
  - We want a transformation into clip coordinates!
  - That requires not only the specification of a perspective projection, but of a view volume as well.
    - Similar to the box we just defined for orthographic projection

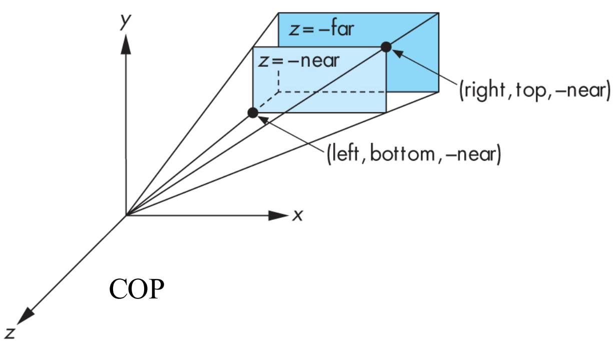


Viewing

- Projections – Perspective
    - View volume is a pyramid with apex its at the COP.
    - Top and bottom are the near and far clip planes, respectively.
    - Notice that the view and clip planes do not *necessarily* need to be the same/coincident.
        - In fact, don't worry about the view plane now

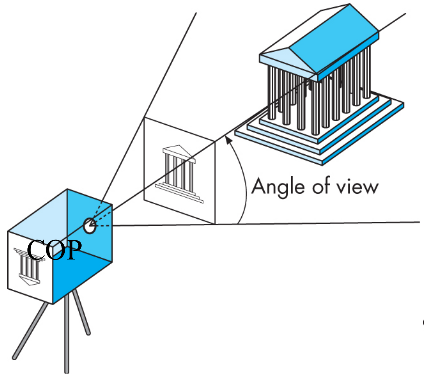


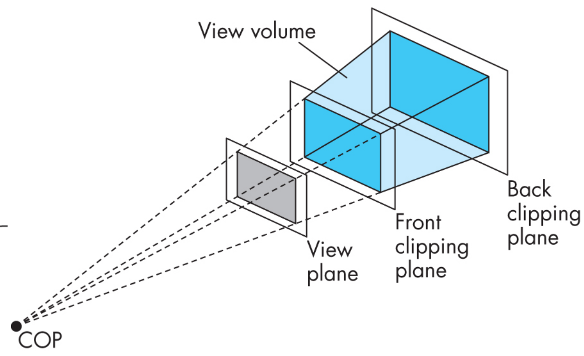


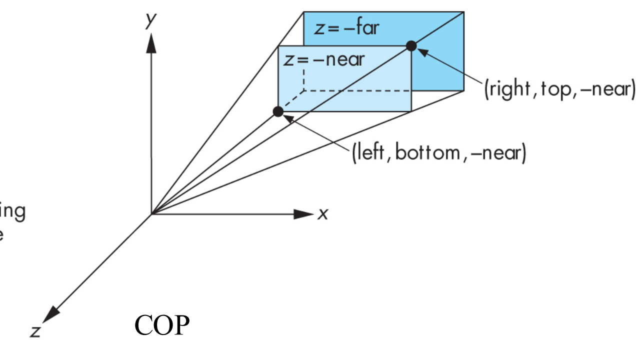


Viewing

- Projections – Perspective
  - The edges specify the near clip plane.
  - The edges implicitly define the *angle of view* of the projection.

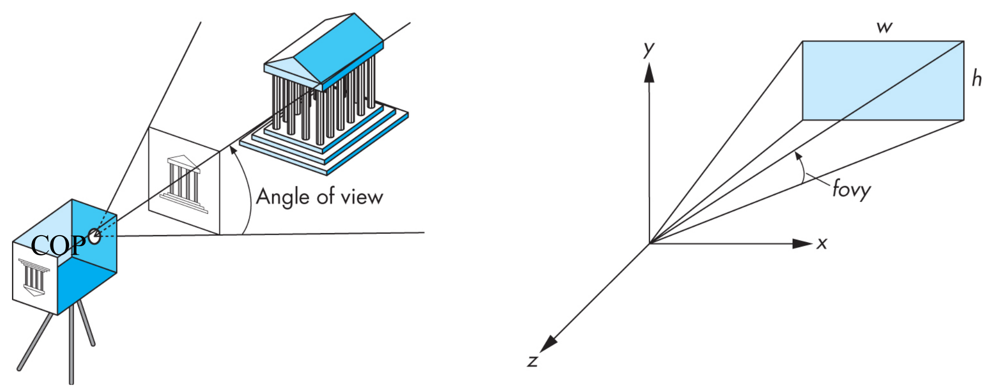






Viewing

- Projections – perspective
  - MV.js provides a utility function
    - `perspective(`*fovy, aspect, near, far*`)`
  - This form is sometimes more convenient to specify.
  - *Aspect* is the aspect ratio of the view volume.
    - *i.e. width / height*



Viewing

- Projections – perspective
  - Once again, what we had earlier is a projection not the full transformation we need to get to clip coordinates.
  - The full matrix we need is defined by:

$$P = NSH = \begin{bmatrix} \frac{right}{near} & 0 & 0 & 0 \\ 0 & \frac{near}{top} & 0 & 0 \\ 0 & 0 & \frac{-far+near}{far-near} & \frac{-2\,far \cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Viewing

- Projections – perspective
  - Matrices are passed from Javascript to the vertex shader just like we have seen earlier.
  - Matrices are **uniform** variables where all the data parallel processors on the GPU will see the same (uniform) value.

```
attribute  vec4 vPosition;
uniform mat4 modelView;
uniform mat4 projection;

void main( )
{
    //
    // The perspective division actually happens to gl_Position immediately
    // after the vertex shader completes. i.e. divided by gl_Position.w
    //
    gl_Position = projection * modelView * vPosition;
}
```

Getting your head around this

- Often there is an initial struggle understanding what is happening

- These next slides will hopefully illuminate what is happening to a vertex, where and when.

Getting your head around this

- First, remember that we are only working with arrays of vertices.
- It is up to us to keep track of what those vertices represent in our code (Javascript).
  - Point, line, triangle, car, dinosaur, kitten, hamburger, etc.

Getting your head around this

- WebGL knows points, lines and triangles
  - That is it
  - They are all made up of vertices
- When we draw an array of vertices we will tell WebGL what to interpret the array of vertices as: points, lines or triangles

Where do the vertices come from

- They could come from a modeling program
  - Overkill for the assignments
  - You have to ‘load’ into your code somehow
- You could define them directly
  - By hand
    - OK for simple geometry, like a cube
  - Algorithmically
    - Using code, can make a sphere this way

Where do the vertices come from

- In any case your vertices (geometry) will end up in a Javascript array
- Let's use a regular tetrahedron as an example (all edges the same length)
- We could compute the four vertices very simply using $\left(\pm 1, 0, \frac{1}{\sqrt{2}}\right)$ and $\left(0, \pm 1, \frac{1}{\sqrt{2}}\right)$

Where do the vertices come from

- (1, 0, 0.707)
- (-1, 0, 0.707)
- (0, 1, 0.707)
- (0, -1, 0.707)

- This is a start but not enough to get a tetrahedron on the screen

Where do the vertices come from

- A tetrahedron is four triangles
  - Something WebGL knows about
- We could define an array of four triangles, three vertices each
- We have enough information to that
- But what order?
  - Does it matter?

Where do the vertices come from

- The order of the individual triangles does not in this simple example
- The order of the vertices that comprise each triangle *does matter*!
- WebGL uses a 'right handed' system
  - Determines the 'direction' of the triangle's normal (which is the side that will be rasterized/painted)
  - Also known as the 'winding' of the vertices
  - Take your right hand and make a fist with your thumb sticking out.
  - The direction of your fingers will tell you the order of vertices to get the normal facing in the direction of your thumb
  - This is important because the side of the triangle the normal is facing away from is the one that will be rasterized (painted/filled in)
  - The back is not drawn at all!

Where do the vertices come from

- For our teraheadron we have the following
  - (1, 0, 0.707), (0, 1, 0.707), (-1, 0, 0.707)
  - (1, 0, 0.707), (0, -1, 0.707), (0, 1, 0.707)
  - (1, 0, 0.707), (-1, 0, 0.707), (0, -1, 0.707)
  - (0, 1, 0.707), (0, -1, 0.707), (-1, 0, 0.707)
- We now have four triangles
- Triangle vertices are in correct order

Where do the vertices come from

- We could put each triangle's data into it's own array and draw each one individually
- We could also put all four into a single array
- You can also just store each vertex once in an array and use another array of indexes into the the first.
- We'll use the second approach

Where do the vertices come from

- Here is one array with all the data
  - (1, 0, 0.707), (0, 1, 0.707), (-1, 0, 0.707), (1, 0, 0.707), (0, -1, 0.707), (0, 1, 0.707), (1, 0, 0.707), (-1, 0, 0.707), (0, -1, 0.707), (0, 1, 0.707), (0, -1, 0.707), (-1, 0, 0.707)
- This would still be an array in CPU memory
- This data is also centered around the origin
  - In ‘model space’
- we can now define a WebGL buffer and assign this data to it
  - We will receive a reference back to a copy of the data stored in GPU memory

Where do the vertices come from

- If we wanted the object to be somewhere else we need to transform it into ‘world’ space
- How do we do that?
- We set up a transformation that is part of the **model-view** transformation

Where do the vertices come from

- Let's assume we want to draw the tetrahedron twice
  - Once, centered at (1,0,0)
  - And again, centered at (-1,0,0)
- The implies two transformations
  - We will apply them one at a time while we draw the geometry

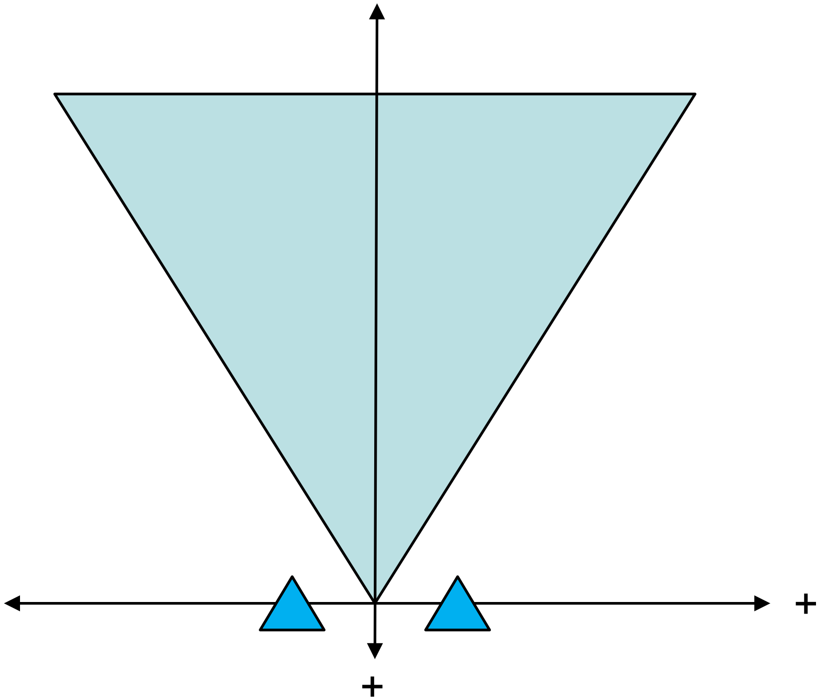
Where do the view come from

- Before we draw anything let's set up the view
- The default perspective view has a COP at the origin
- If we do not reposition the view we will not see anything, even after applying our model transformations.
  - WHY?

UCLA

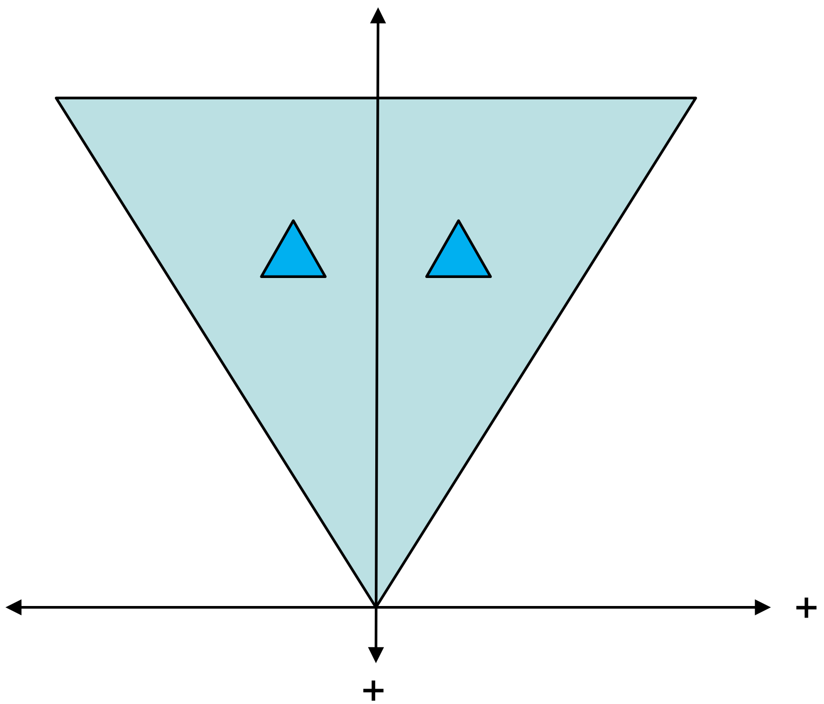
Where do the view come from

- Because our default view is looking away from where the geometry is in **world** space



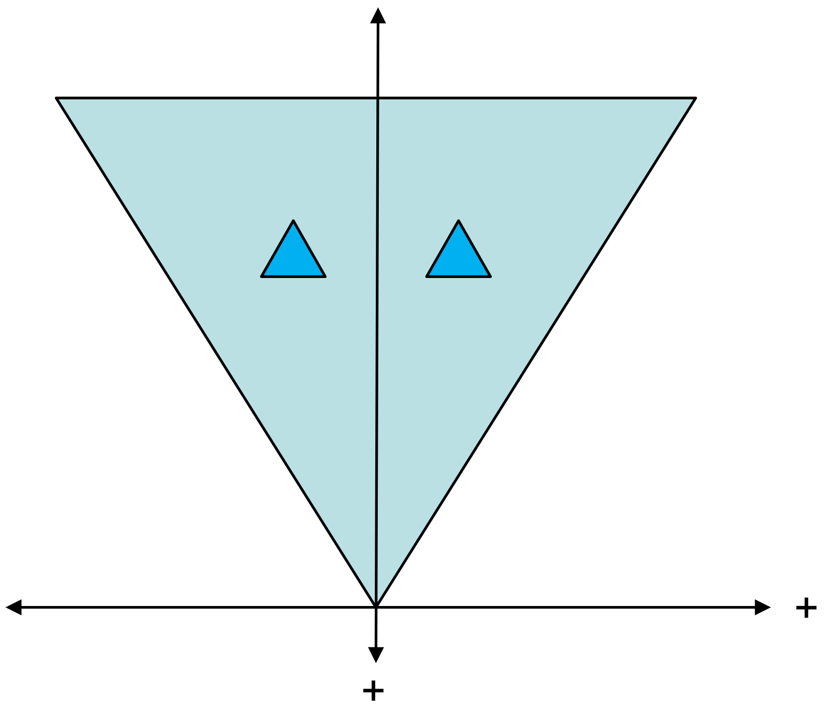
Where do the view come from

- Our view transformation needs to position the objects in the world in such as way to match what we want to see



Where do the view come from

- Let’s transform the objects by (0,0,-6)



Where do the view come from

- We now have
  - Two model transformations
  - One view transformation
- We also need (have) a projection transformation
  - This would be the light blue area on the previous slide

Almost ready to draw

- Four transformation matrices
- Two objects
- What do we ‘send’ to WebGL to draw?
- Some of what we need we have already we just need to get the WebGL state correct

Almost ready to draw

- Need to be sure vertex buffer is ‘bound’ to the WebGL state
- Need to be sure our shaders active (useProgram)

Let's draw

- We first compute our model-view matrix for object #1 – in Javascript

- MV = V * M1

Let’s draw

- We then set vertex shader uniform variables for our **model-view** and **projection** matrices

- This is so the shader has access to these transformation matrices when operating on the vertices

Let's draw

- We then tell WebGL to draw our vertex data

- We do this by passing the reference to the vertex array buffer and telling WebGL to interpret this data as a series of n triangles.
  - n would be 4 in this case

Let’s draw

- We then tell WebGL to draw our vertex data
- We do this by passing the reference to the vertex array buffer and telling WebGL to interpret this data as a series of n triangles.
  - n would be 4 in this case

Let’s draw

```
…
gl.viewport( 0, 0, canvas.width, canvas.height );
gl.clearColor( 1.0, 1.0, 1.0, 1.0 );
var program = initShaders( gl, "vertex-shader", "fragment-shader" );
gl.useProgram( program ); // setting state of which shaders to use
// Load the data into the GPU
var bufferId = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
gl.bufferData( gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW );
var vPosition = gl.getAttribLocation( program, "vPosition" );
gl.vertexAttribPointer( vPosition, 3, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vPosition );;);
var modelView= gl.getUniformLocation(program, "modelView");
var projection = gl.getUniformLocation(program, ”projection");
// call when we want to update the value passed to the shader
gl.uniformMatrix4fv(projection, false, new flatten(P));
gl.clear( gl.COLOR_BUFFER_BIT );
gl.uniformMatrix4fv(modelView, false, (V * M1));
gl.drawArrays( gl.TRIANGLES, 0, vertices.length );
gl.uniformMatrix4fv(modelView, false, (V * M2));
gl.drawArrays( gl.TRIANGLES, 0, vertices.length );
…
```

UCLA

Let’s draw

- When gl.drawArrays is called the GPU comes into the action
- This is when the vertex shader gets executed by the GPU
- Remember, the GPU is highly parallel
  - It first operates on vertices in parallel
  - ‘attributes’ are different for each execution
  - ‘uniforms’ are the same for each execution

Let’s draw

- Then, **modelView** will be the same everywhere and so will **projection** in the vertex shader
- vPosition will be each of the vertices in our buffer.
- We let the GPU do the heavy lifting of multiplying the modelView and projection matrices to each and every vertex

Let's draw

- Recall we only have one copy of the data for a tetrahedron centered at the origin loaded into the GPU
- Notice we reset the modelView matrix every time we want to draw a new object
- We do not include the projection matrix here because it stays the same for all objects – we'll let the GPU multiply it
- There are other ways to accomplish this – we could have sent the view matrix once and just kept adjusting the model matrix.

```
    …
    gl.uniformMatrix4fv(modelView, false, (V * M1));
    gl.drawArrays( gl.TRIANGLES, 0, vertices.length );
    gl.uniformMatrix4fv(modelView, false, (V * M2));
    gl.drawArrays( gl.TRIANGLES, 0, vertices.length );
…
```

What about color?

- Color, in this example, can be handled the same was as the gasket example
- Pass a uniform variable to the fragment shader
- Once the WebGL pipeline gets to the fragment shader its parallel operations are working on…fragments, which are assigned color values.
- We'll talk more about fragment shaders when we discuss lighting

```
   …
   gl.uniformMatrix4fv(modelView, false, (V * M1));
   gl.drawArrays( gl.TRIANGLES, 0, vertices.length );
   gl.uniformMatrix4fv(modelView, false, (V * M2));
   gl.drawArrays( gl.TRIANGLES, 0, vertices.length );
…
```