UCLA

CS174A : Introduction to Computer Graphics

Kinsey 1240
MW 4-6pm

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

Assignment #1

- Has been posted to Piazza
  - Lecture slides posted to resource section
- GitHub
  - The form is simply to tie your UID to your GitHub username
  - Repository is setup automatically when you click on the link in the assignments

A little math background

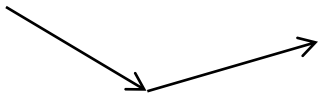
- Representing geometric objects
  - We want an *efficient* general representation.
    - Flexible coordinate systems.
      - » Understood to mean not imposing particular dimensional or unit constraints.
    - Using *homogenous* coordinates.
    - All good for *fast* hardware implementation.
  - What is that and how do I get it?

Spaces

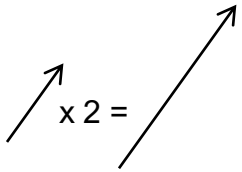
- We are going to concern ourselves with three kinds of "spaces".
  - Vector space
  - Affine space
  - Euclidian space
  - Each builds on the last to give us the tools we need.
  - Geometric objects will exist within these spaces.

Spaces

- **Vector Space**
  - Vectors have ***only*** direction and magnitude.
  - Addition and multiplication are permitted.

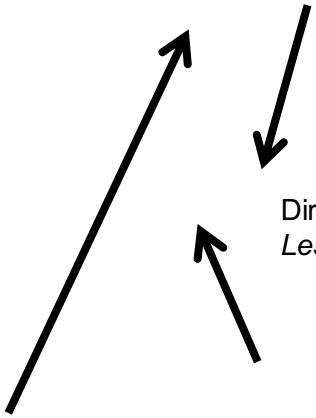


  - Addition
    - Head-to-tail axiom (e.g. connect tail to head of last)
  - Multiplication
    - By a scalar.
    - Changes only the magnitude.



Spaces

- Vector Space
  - There are two ways to understand a vector
    - Directed line segments
    - *N*-tuples of real numbers (what WebGL uses)



Directed line segments
Less useful (looks nice though)

$$v = (v_1, v_2, \ldots, v_n)$$

N-tuple of reals (floats)
useful

Spaces

- Vector Space
  - The space where operations on these vectors is defined is termed $\mathbf{R}^n$
  - This $\mathbf{R}^n$ is also where we can manipulate vectors using matrix algebra – which will be useful later.
  - The notion of ***linear independence*** defines what we understand as a ***dimension***.
  - The *D* in *3D*

Spaces

- Vector Space (linear independence)
  - Take a linear combination of vectors u.

$$u = \alpha_1 u_1 + \alpha_2 u_2 + \ldots + \alpha_n u_n$$

  - Iff the ***only*** set of scalars such that

$$\alpha_1 u_1 + \alpha_2 u_2 + \ldots + \alpha_n u_n = 0$$

  - is

$$\alpha_1 = \alpha_2 = \ldots = \alpha_n = 0$$

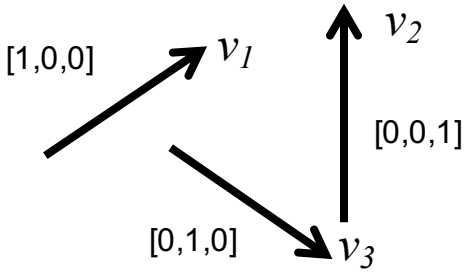
  - The vectors are said to be ***linearly independent***.
  - For n=3, the set u (1,0,1),(1,1,0),(0,1,1) satisfies this
    - Any vector is not a linear combination of any others
  - The ***largest*** number of linearly independent vectors that we can find in a ***space*** gives the ***dimension.***

Spaces

- Vector Space
  - Given a vector space of dimension ***n***, then ***<u>any</u>*** set of ***n linearly independent vectors*** forms a ***basis.***

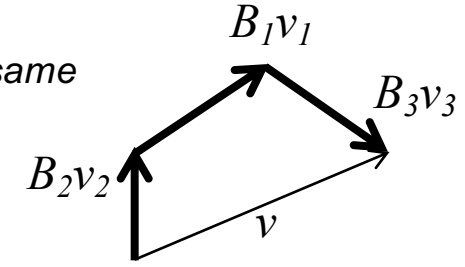
$$v = \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n$$

  - The scalars $\{\beta_i\}$ give a ***representation*** of v with respect to the basis, $v_1, v_2, \ldots, v_n$.



linearly independent vectors

Notice the *vectors* are the *same*



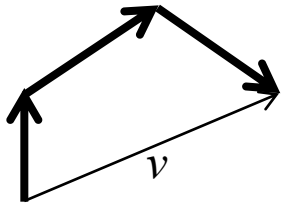
a "representation"

Spaces

- Vector Space
  - The key in the last slide was the word ***any*.**
  - The ***basis*** is, basically, every (all) ***linearly independent vector(s)*** that exists within a ***dimension***.
  - Also called a Span

Spaces

- Vector Space
  - …and what is a ***representation*** of a vector?
    - It is just a **specific** ***linearly independent vector*** within a particular ***dimension.***
  - Now we have ***linearly independent vectors, dimension, basis*** and ***representations.***



a specific "representation" of v

Spaces

- Vector Space
  - So what?
  - The ***basis*** gives us a ***representation***.
    - We can use matrix multiplication to change one ***representation*** into another representation using…
    - Transformations – (translation, scaling, rotation, etc.)
  - Remember, all in *abstract* vector space so far.

Spaces

- Vector Space
  - However, once we decide on a ***basis*** we have committed to using a ***dimension*** to describe our set of ***linearly independent vectors.***
  - If we restrict the scalars of the ***basis*** to real numbers
    - We can use *n*-tuples of reals and use matrix algebra.
    - Better than trying to do this all in abstract vector space.
  - We are inching towards something real
    - …promise.

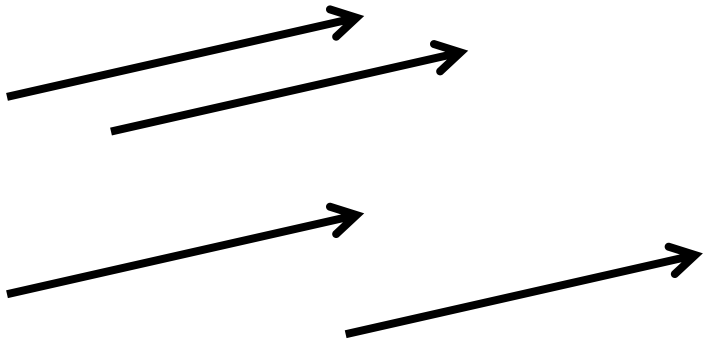
Spaces

- Vector Space
  - Matrices are useful for changing ***representations.***
    - *i.e. changing the representation of a vector*
  - We will use this later to be able to convert from one ***frame*** to another (we'll get to ***frames*** in a minute)

$$\begin{bmatrix} \beta'_1 \\ \beta'_2 \\ \vdots \\ \beta'_N \end{bmatrix} = M \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_N \end{bmatrix}$$

Spaces

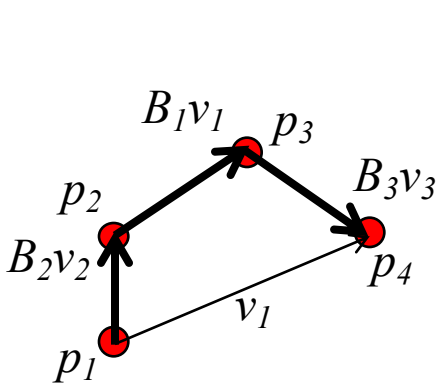
- **Affine Space**
  - Vector space has a downside
    - It does not have any concept of ***location***.
  - Vector space has vectors floating…nowhere.
    - Just ***direction*** and ***magnitude***, remember?
  - But, we can plant them in a ***dimensional*** space.
    - Remember $\mathbf{R}^n$?
    - How about $\mathbf{R}^3$?



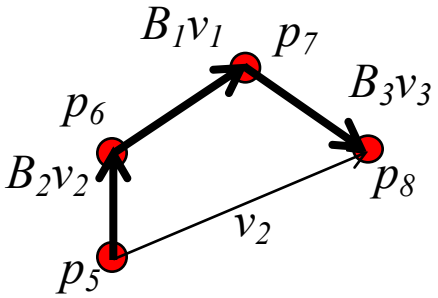
These are all the same
i.e. have the same ***representation***

Spaces

- Affine Space
  - Affine space introduces the ***point*** to vector space.
  - A ***point*** gives us ***location***
  - *Both representations below are distinct because* $\mathbf{p_n}$ *has planted them in* $\mathbf{R^3}$



a "representation"



another "representation"

Spaces

- Affine Space
  - Along with that ***point*** we add an operation
    - **Point subtraction**, given points P and Q.

$$v = Q - P$$

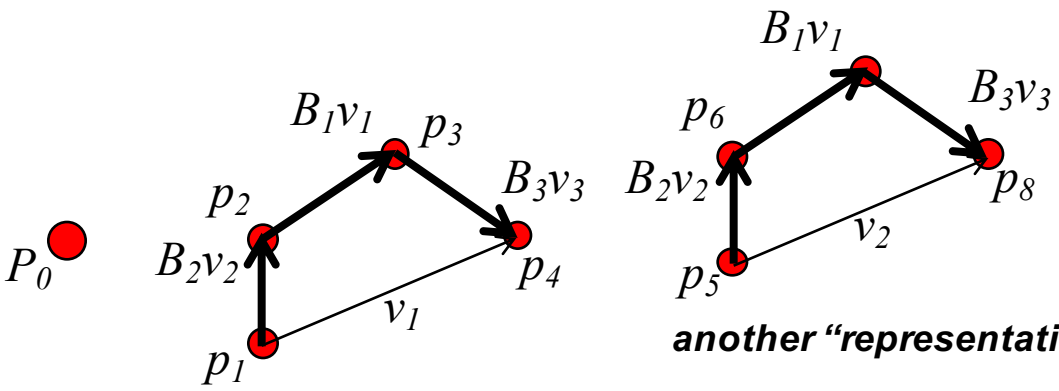
    - Gives us a ***vector***. Which, with a little rearranging, gives us vector/point addition.

$$Q = v + P$$

    - Adding two points does not make sense, right?

Spaces

- Affine Space
  - We can define *affine space* in terms of a ***frame***
    - Think of a ***frame*** as all the ***coordinate systems*** that exist within a particular ***basis***, $\mathbf{R^n}$.
    - Similar type of relationship as ***representation*** is to ***basis.***
  - A ***frame*** consists of a point P_0 (the origin) and ***basis***.



a "representation" within the frame

another "representation" within the frame

Spaces

- Affine Space
  - If P_0 is the ***origin*** of our ***frame.***
  - Any point within the ***frame*** can be defined by

$$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n$$

  - We now have the ability to define geometry with ***points*** in a coordinate system.

  - No concept of ***length*** or ***distance*** yet!

Spaces

- **Euclidian Space**
  - Consists of only vectors and scalars. (reals only)
  - We define a new operation, the ***dot product.***
    - The operation combines two vectors to form a scalar real.
    - Which also satisfies the following properties.

$$u \bullet v = v \bullet u,$$

$$(\alpha u + \beta v) \bullet w = \alpha u \bullet w + \beta v \bullet w,$$

$$v \bullet v > 0 \text{ if } v \neq 0$$

Spaces

- Euclidian Space
  - It means that if $u \bullet v = 0$, then u and v are ***orthogonal*** to each other. (or independent of each other)
  - The actual operation on two vectors looks like this:

$$u \bullet v = u_1 v_1 + u_2 v_2 + \ldots + u_n v_n$$

  - Interestingly this leads to the observation that

$$u \bullet u = |u|^2$$

  - Which is the square of the vector's ***length.***
    - ***Very Useful!***

Spaces

- Euclidian Space
  - Furthermore
$$u \bullet v = |u||v|\cos\theta$$
  - Gives the ***angle*** between the two vectors, u and v.
$$\theta = \cos^{-1}\frac{u \bullet v}{|u||v|}$$
  - All very interesting
    - Because when we add in the key concept from *Affine Space*…

Spaces

- Euclidian Space
  - Adding the concept of ***points*** from *affine space* we can now get the distance between those ***points***.
    - Recall that for two ***points*** P and Q, P-Q is a ***vector***, so

$$|P-Q| = \sqrt{(P-Q)\bullet(P-Q)}$$

  - …is the ***distance*** between P and Q.

  - We now have the foundation describing the geometric world we will be using.

Spaces

- **Planes**
  - So far we have vectors and points (and lines)
  - A ***plane*** exists in *affine* space and its defined by three points; P, Q and R.
    - All the points that make up a line between P and Q can be found via interpolation:

$$S(\alpha) = \alpha P + (1-\alpha)Q,\ 0 \le \alpha \le 1$$

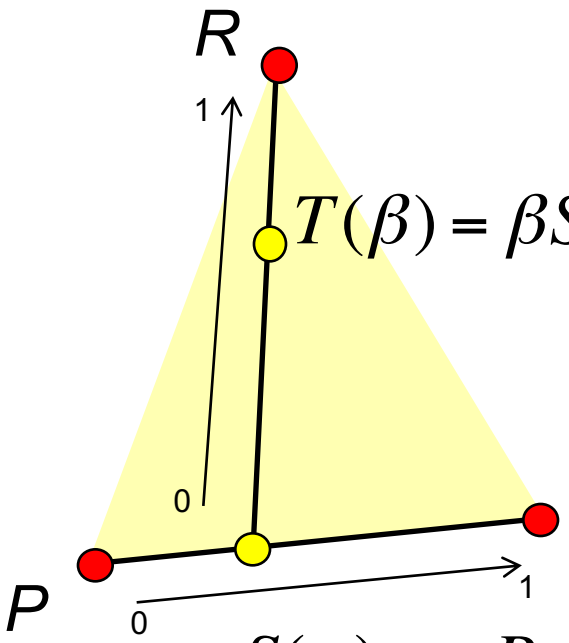
    - Picking an arbitrary point along this line and connecting it to a point R we get a second line:

$$T(\beta) = \beta S + (1-\beta)R,\ 0 \le \beta \le 1$$

    - All the points defined by α and β form the plane defined by points P, Q and R.

Spaces

- Planes (a picture helps)



$$T(\beta) = \beta S + (1-\beta)R,\ 0 \le \beta \le 1$$

$$S(\alpha) = \alpha P + (1-\alpha)Q,\ 0 \le \alpha \le 1$$

Spaces

- Planes
  - If we combine and rearrange this we get:

$$T(\alpha,\beta) = \beta\left[\alpha P + (1-\alpha)Q\right] + (1-\beta)R$$

    - …and then

$$T(\alpha,\beta) = P + \beta(1-\alpha)(Q-P) + (1-\beta)(R-P)$$

  - Which are all the points interior to the plane defined by the points P, Q and R
    a.k.a. a ***triangle***.
    - Remember, (Q-P) and (R-P) are arbitrary vectors.
    - So a plane can be defined by two vectors, as long as the two vectors are not parallel.

Spaces

- Planes
  - Simplifying a bit more we can get to:
    $$T(\alpha,\beta) = \beta\alpha P + \beta(1-\alpha)Q + (1-\beta)R$$
  - Which is also known as a point's ***barycentric coordinate*** representation.
  - Not super-critical to this class but you will sometimes see term mentioned in a derivation.

Spaces

- Planes
  - Interestingly, we can find a vector n that is ***orthogonal*** to our ***plane***, defined by the vectors u and v, by using an operator called the ***cross product.***

$$n = u \times v$$

  - This resulting vector is known as the ***normal*** to the plane defined by u and v.
    - We will find **many uses** for the normal in this course.

$$u = \alpha_1 + \alpha_2 + \alpha_3$$
$$v = \beta_1 + \beta_2 + \beta_3$$

$$n = \begin{bmatrix} \alpha_2\beta_3 - \alpha_3\beta_2 \\ \alpha_3\beta_1 - \alpha_1\beta_3 \\ \alpha_1\beta_2 - \alpha_2\beta_1 \end{bmatrix}$$

Homogeneous Coordinates

- Recall, a ***frame*** for an ***affine*** space is given by
  - a ***basis*** and an origin point P_0
  - Then, any point p within the ***frame*** can be written as

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n + 0 P_0$$

  - The $\mathbf{R^3}$ coordinate vectors of the vector v and point p can be written as $p = \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n + 1 P_0$
  - Notice the 0 and 1
  - 0 is nowhere
  - 1 plants it in space

$$v = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ 0 \end{bmatrix} \quad p = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ 1 \end{bmatrix}$$

Homogeneous Coordinates

- This is ***four*** dimensional?
  - Yes!
  - It has its benefits.
  - Now ***vectors*** and ***points*** have a distinct form where before they were both *n*-tuples.
  - But there is *more! You will see!*

$$v = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ 0 \end{bmatrix} \quad p = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ 1 \end{bmatrix}$$

Transformations

- ***Scale***, Rotation and Translation
  - I am going to use 2D transformations, it's simpler, but it trivially extends to 3D.
  - Let's define a ***scale*** transformation matrix.

$$S(\alpha,\beta)=\begin{bmatrix}\alpha & 0\\ 0 & \beta\end{bmatrix}$$

  - Applying the transformation gives
  - No problem.

$$\begin{bmatrix}\alpha & 0\\ 0 & \beta\end{bmatrix}\begin{bmatrix}x\\ y\end{bmatrix}=\begin{bmatrix}\alpha x\\ \beta y\end{bmatrix}$$

Transformations

- Scale, ***Rotation*** and Translation
  - Let's define a ***rotation*** transformation matrix.

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

  - Applying the transformation gives

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos(\theta) - y\sin(\theta) \\ x\sin(\theta) + y\cos(\theta) \end{bmatrix}$$

  - Again, no problem.
  - QUESTION FOR LATER – what is this rotating around?

Transformations

- Scale, Rotation and ***Translation***
  - Unfortunately, we cannot perform a ***translation*** with a matrix-vector multiplication.
  - This is a problem – we need to do translations!
  - Fortunately, there is a solution…

Transformations

- Scale, Rotation and Translation
  - Homogeneous coordinates - triumphantly return!
  - If we represent the ***points*** we wish to ***translate*** instead as such

$$\begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

  - Things start to work out very nicely.

Transformations

- Scale, Rotation and ***Translation***
  - Let's now define a ***translation*** transformation matrix like this
  - Oh my!
  - How nice!

$$\begin{bmatrix} 1 & 0 & j \\ 0 & 1 & k \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x+j \\ y+k \\ 1 \end{bmatrix}$$

  - But what happens to our other transformations?
    - Won't it break?
  - Lets see…

Transformations

- ***Scale***, Rotation and Translation
  - Let's see how the ***scale*** transformation works using ***homogeneous coordinate*** form.
  - Let's re-define a ***scale*** transformation matrix and apply it.

$$S(\alpha,\beta)=\begin{bmatrix}\alpha & 0 & 0\\ 0 & \beta & 0\\ 0 & 0 & 1\end{bmatrix},\quad \begin{bmatrix}\alpha & 0 & 0\\ 0 & \beta & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\ y\\ 1\end{bmatrix}=\begin{bmatrix}\alpha x\\ \beta y\\ 1\end{bmatrix}$$

  - Rotation follows in the same way.
  - The same holds true in $\mathbf{R}^3$.

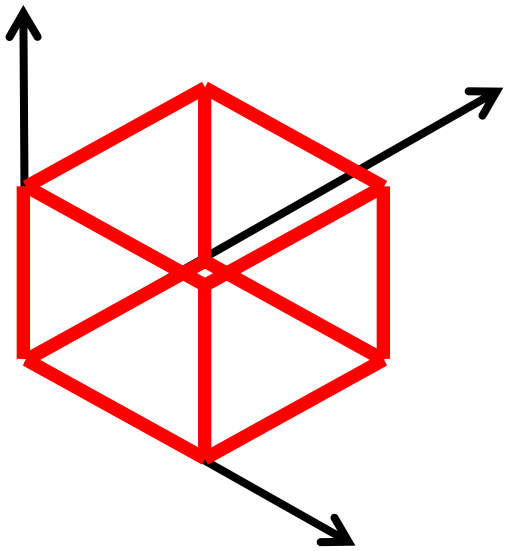
WebGL Frames

- There are, traditionally, six coordinate frames used in the WebGL pipeline.

1. Model (object/local) coordinates
2. World coordinates
3. View (camera/eye) coordinates
4. Clip coordinates
5. Normalized device coordinates
6. Window (screen) coordinates

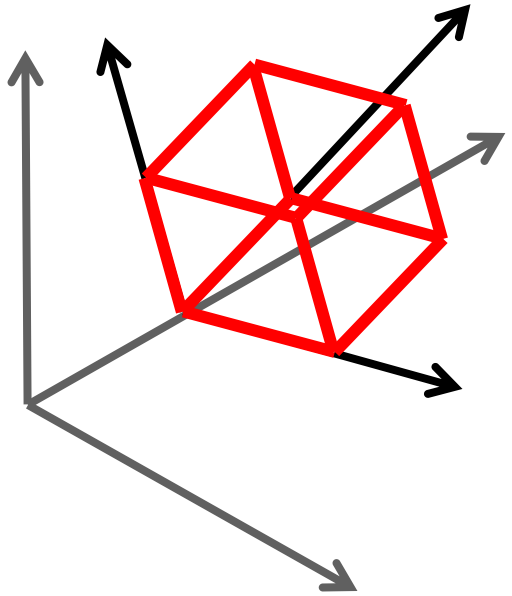
WebGL Frames

- geometry is represented in Model coordinates. (sometimes called ***object*** or ***local*** coordinates)
  - This is the local ***frame*** of the object that is convenient to model the geometry.



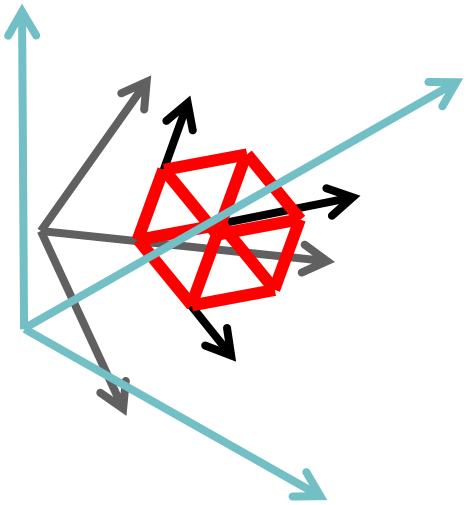
WebGL Frames

- ***Models*** are placed in ***world*** coordinates.
  - This is a global ***frame*** typically used to position ***objects*** relative to each other by ***scaling, translating and rotating***.



WebGL Frames

- View (camera or eye) coordinates
  - While ***object*** and ***world*** coordinates are convenient for modeling we need to decide where our ***view*** is in order to determine what we "see" (draw)
  - We use 4x4 matrices to transform from **model** to **world** to **view** coordinates.
  - We concatenate matrices into
    - the ***model-view*** transformation



WebGL Frames

- ***Clip***, Normalized device & Window coordinates
  - We'll get to the specifics later on…but, briefly
  - Clip coordinates - used to reject primitives outside of the canonical *view volume* after the projection transformation.
  - It is easiest to perform clipping when we transform the *view volume* into a cube centered around the origin.
    - Recall the "Canonical View Volume"
    - The hardware can perform clipping *very quickly* here.

WebGL Frames

- Clip, ***Normalized device*** & Window coordinates
  - After the projection transformation
    - ***Clip coordinates*** are still ***homogeneous coordinates. (4D)***
    - Dividing out the *w* component, perspective division results in a 3D point in ***normalized device coordinates.***

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \rightarrow \begin{matrix} perspective \\ division \end{matrix} \rightarrow \begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix}$$

WebGL Frames

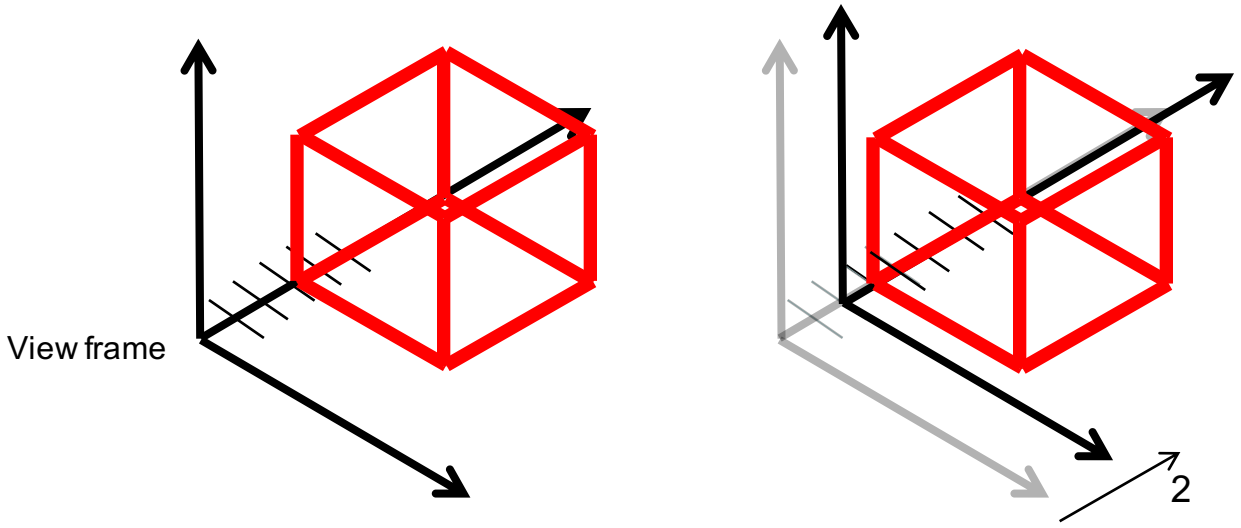
- Clip, Normalized device & ***Window*** coordinates
  - Finally, ***normalized device*** coordinates are transformed into ***window*** coordinates. (typically a 2D scale)
  - The results are 3D points in ***window*** coordinates.
  - Since ***window*** (screen) coordinates are 2D the pipeline just drops the *z* (depth) value. (actually those z values are useful)
  - Those ***window*** (screen) coordinates are actual pixel locations defined by the ***viewport.***

WebGL Camera

- There is *no camera,* per se, in WebGL
  - We control what we "see" by manipulating the ***model-view*** transformation.
  - There are two ways to think of this.
    1. Transform the ***world*** into the ***view*** coordinate frame.
    2. Transform the ***view*** to the ***world*** coordinate frame.
  - These are really the same thing.
    - Just different ways of thinking about the problem.
    - How do we get what we want to "see" in front of our ***eye***?

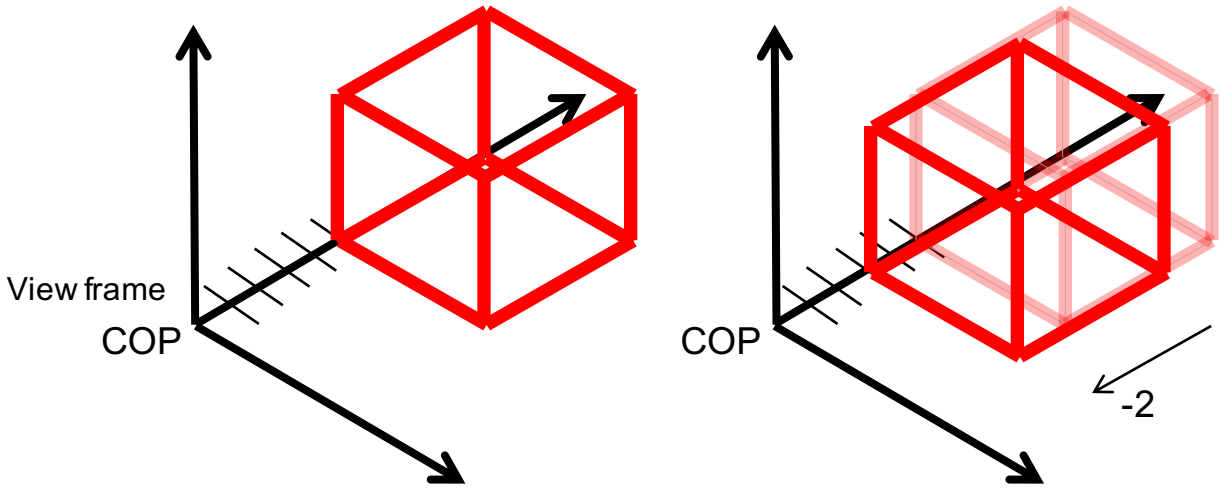
WebGL Camera

- World fixed at the origin
  - We transform the ***view*** into the ***world.***
  - If we wanted to move our view forward by 2.
    - We would transform the ***view*** frame by +2.



WebGL Camera

- Camera fixed at the origin
  - We transform the ***world*** in front of our ***view.***
  - If we wanted to move our view forward by 2.
    - We would transform the ***world*** frame by -2.
    - Why might this be a better approach?



WebGL Transformations

- All transformation matrices are 4x4
  - Performed using ***homogeneous coordinates***.
  - Vertices do not need to be represented using homogeneous coordinates in your code.
    - Since forth-dimension is always the same value, 1.

WebGL Transformations

- Homogeneous ***translation*** matrix.

$$T = T(\Delta x, \Delta y, \Delta z) = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  - Important to note that WebGL represents matrices as arrays interpreted as **column vectors**.

WebGL Transformations

- Homogeneous ***scale*** matrix.

$$S = S(\beta_x, \beta_y, \beta_z) = \begin{bmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  - Important to note that WebGL represents matrices as arrays interpreted as **column vectors**.

WebGL Transformations

- Homogeneous ***rotation*** matrix.
  - Rotations are performed about a single axis.
  - Rotations around the origin leads to definitions for rotation around the x, y and z axes.
  - For instance, rotation about the z-axis is defined as

$$R_z = R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

WebGL Transformations

- Homogeneous ***rotation*** matrix.
  - Rotations about the x-axis.

$$R_x = R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  - Rotations about the y-axis.

$$R_y = R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

WebGL Transformations

- Matrix multiplication *is* associative.

$$q = CBAp \rightarrow q = (C(B(Ap)))$$

  - This means we can concatenate matrices together.
  - Lets take transformation matrices A, B and C

$$M = CBA$$

  - Once concatenated into a single matrix we get the full transformation in a single step.

$$q = Mp$$

  - This typically results in a significant performance gain when transforming lots and lots of geometry.

WebGL Transformations

- Matrix multiplication *is not* commutative.
  - When performing multiple rotation transformations. (think about it)
  - Transformation of any single type, the order does not matter.
  - The **order** of transformation ***types*** does matter!
    - » i.e. ***q=TRSTp*** is not the same as ***q=TSRTp***

WebGL Transformations

- What order to get what I want?
  - They are applied in the opposite order that you expect them to be applied – last to first.
  - This is because they are post-multiplied.
    - An WebGL convention

$$q = TR_z Tp$$

$$C \leftarrow I$$

$$C \leftarrow CT(1.0, 2.0, 3.0)$$

$$C \leftarrow CR_z(45.0)$$

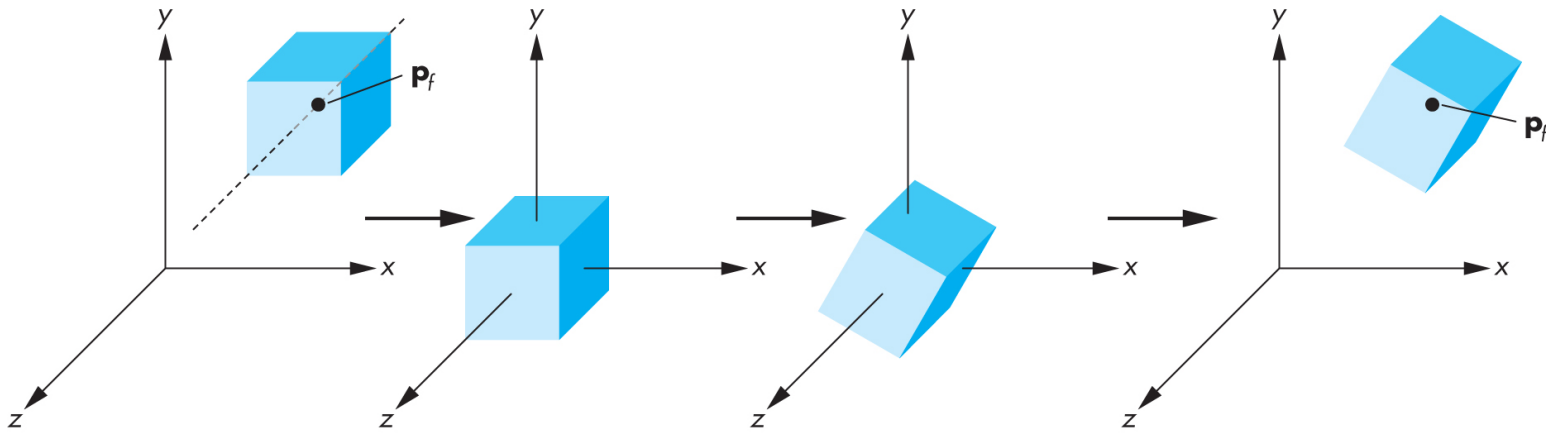
$$C \leftarrow CT(-1.0, -2.0, -3.0)$$

$$q = Cp$$

Appears to be
Applied in this order

WebGL Transformations

- What order to get what I want?
  - As you can see the transformations are applied last to first.
  - This is simply because of post-multiplication.



WebGL Transformations

- How do we apply transformations to geometry?
  - We could apply them to our geometry in our application.
    - Update buffer by repeatedly calling `glBufferData()`
      - » Although GL_STATIC_DRAW is now a pretty bad hint.
  - This would work but…
    - CPU is doing work better suited to the GPU
    - Geometry data would have to be re-sent to GPU each time we apply and or change the transform.

WebGL Transformations

- How do we apply transformations to geometry?
  - A better way would be to send the data once.
  - All we want to do is update the transformation.
  - We can pass the transformation to the GPU by specifying a `uniform` variable in our shader definition.
    - the setup is very similar to passing the vertex data.

WebGL Transformations

- How do we apply transformations to geometry?
  - After compiling and linking the vertex shader program.
  - We find the location of the variable we are after.

```
someFunc( )
{
    var rotationMatrix = rotate( 30.0, [ 0.0, 0.0, 1.0 ] );
    ...
    var program = initShaders( gl, "vertex-shader", "fragment-shader" );
    gl.useProgram( program );
    ...
    var mTransformation = gl.getUniformLocation(program, "mTransformation");
    ...
    // call when we want to update the value passed to the shader
    gl.uniformMatrix4fv(mTransformation, false, new flatten(rotationMatrix));
    ...
    gl.drawArrays( ... );
...
}
```

WebGL Transformations

- How do we apply transformations to geometry?
  - When we are ready to set the value in the shader we indicate the location in the shader and the data.

```
someFunc( )
{
    var rotationMatrix = rotate( 30.0, [ 0.0, 0.0, 1.0 ] );
    ...
    var program = initShaders( gl, "vertex-shader", "fragment-shader" );
    gl.useProgram( program );
    ...
    var mTransformation = gl.getUniformLocation(program, "mTransformation");
    ...
    // call when we want to update the value passed to the shader
    gl.uniformMatrix4fv(mTransformation, false, new flatten(rotationMatrix));
    ...
    gl.drawArrays( ... );
    ...
}
```

WebGL Transformations

- How do we apply transformations to geometry?
  - In the shader we specify the variable as `uniform`.
  - The calculation should be obvious.
  - You might try this out on gasket demo by rotating the fractal around the origin.

  - Use keyboard to dynamically rotate fractal.
    - onkeypress (onkeydown, onkeyup)

```
attribute  vec4  vPosition;
uniform  mat4  mTransformation;

void  main(  )
{
    gl_Position  =  mTransformation  *  vPosition;
}
```

Next Time

- We will focus on
  - Projection.
  - View transformations.