UCLA

CS174A : Introduction to Computer Graphics

Royce 190
TT 4-6pm

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

Assignment #1

- Push back deadline to Friday 10/17 11:55pm
- You should have started by now!
- As I have said, the expectation is that you can pick up what you need to know about HTML and Javascript
- 7th Edition is the required text for this class
  - If you can make it work without it that is up to you

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.

UCLA

Spaces

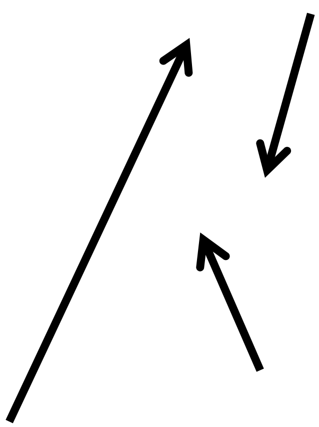
- Vector Space
  - Vectors have ***only*** direction and magnitude.
  - Addition and multiplication are permitted.

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

UCLA

Spaces

- Vector Space
  - There are two ways to understand a vector
    - Directed line segments – good for understanding, not so good as a practical matter.
    - n-tuples of real numbers – what OpenGL uses.



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

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 – 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*.

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

  - If the *only* set of scalars such that

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

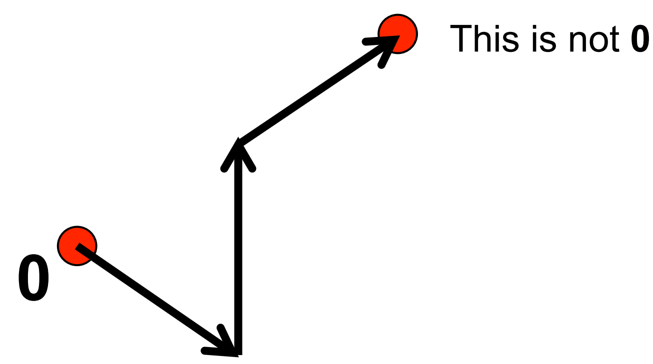
  - is

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

  - The vectors are said to be *linearly independent*.
  - For n=3, there is only one set u (1,0,1),(1,1,0),(0,1,1)
  - The *largest* number of linearly independent vectors that we can find in a *space* gives the *dimension*.

Spaces

- Vector Space
  - The set of ***linearly independent vectors*** we care about are these – you may be familiar with them.
  - You can see, very easily, that the ***only*** way to sum them and get **0** as a result is if you multiply them all by zero.

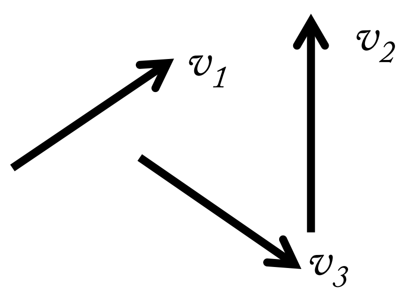


Spaces

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

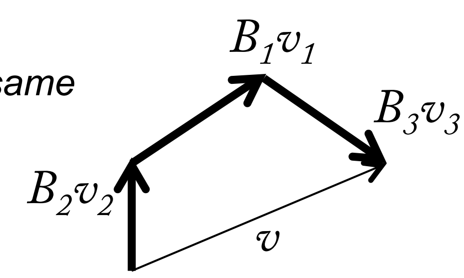
$$v = \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n$$

  - The scalars $\{\beta_i\}$ give the *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”

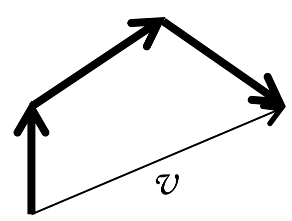
Spaces

- 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*.

UCLA

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"

Spaces

- Vector Space
  - So what?
  - The *basis* gives us a *representation* as we have seen.
    - We can use matrix multiplication to change one *representation* into another using.
    - Translation, scaling, rotation, etc.

  - Remember this is all in abstract space 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.

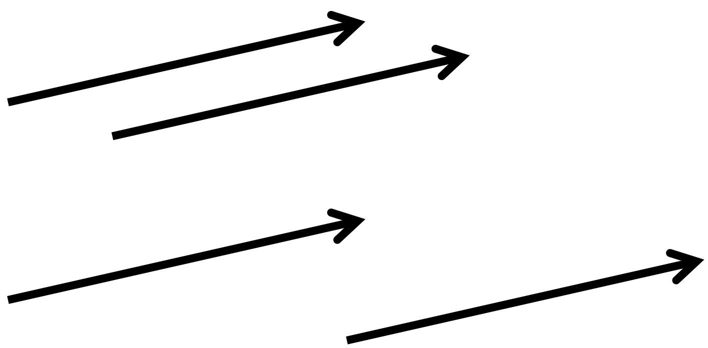
Spaces

- Vector Space
  - Matrices are useful for changing ***representations***.
  - We will use this later to be able to convert from one ***frame*** to another (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

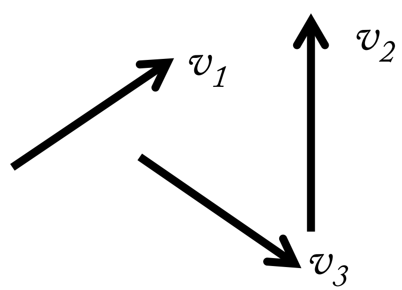
- Affine Space
  - Vector space does not have any concept of ***location***.
  - Vector space has vectors floating nowhere.
    - Just *direction* and *magnitude*, remember?
  - But, we can plant them in a ***dimensional*** space.
    - Remember $\mathbf{R}^n$?
    - How about $\mathbf{R}^3$?



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

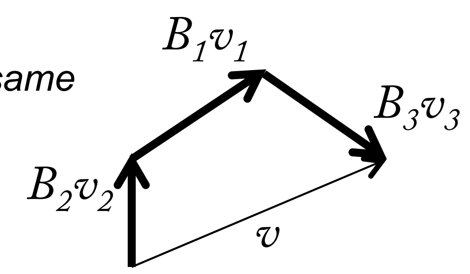
Spaces

- Affine Space
  - Even using a *representation* of a vector
  - Vectors can *appear* to be emerging from anywhere.
  - We still do not have *location*.



linearly independent vectors

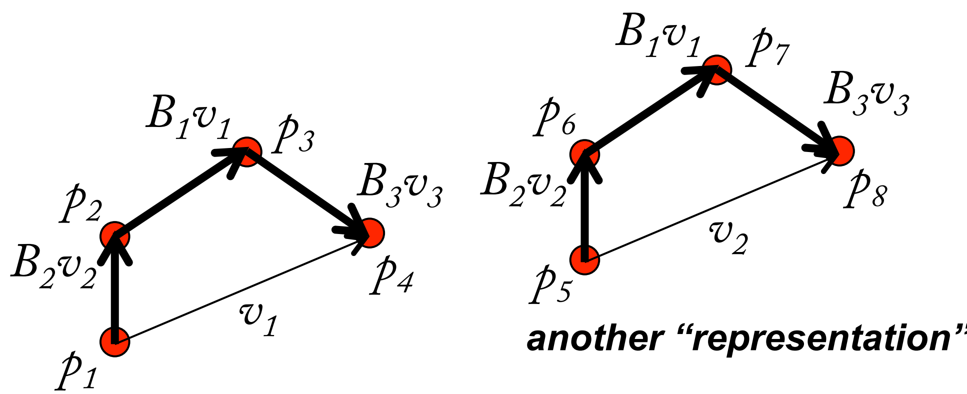
Notice the *vectors* are the *same*



a "representation"

Spaces

- Affine Space
  - Affine space introduces the concept of a ***point*** to vector space.
  - A ***point*** gives us ***location***



a "representation" ***another "representation"***

Spaces

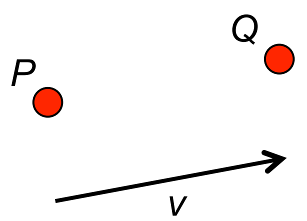
- Affine Space
  - Along with that ***point*** we add an operation
    - **Point subtraction,** given points P and Q.

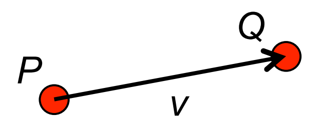
$$v = P - Q$$

    - Gives us a ***vector***. Which leads us to vector/point addition.

$$Q = v + P$$

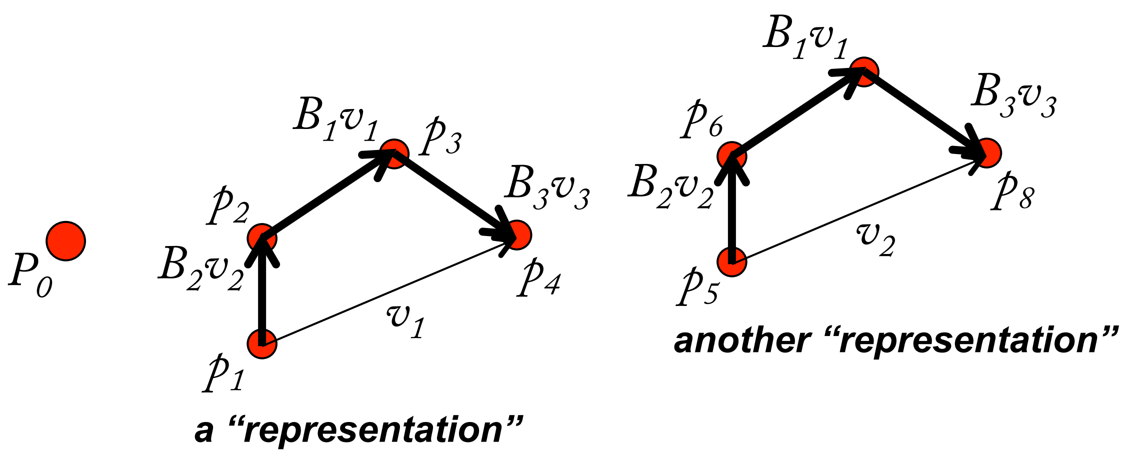
    - Adding two points does not make sense, why?





Spaces

- Affine Space
  - We can define *affine space* in terms of ***frames***.
    - Think of a ***frame*** as all ***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***.



UCLA

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.
  - We do not have the concept of length or distance yet, however.

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 real.
    - It 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.
  - 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 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*…

UCLA

Spaces

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

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

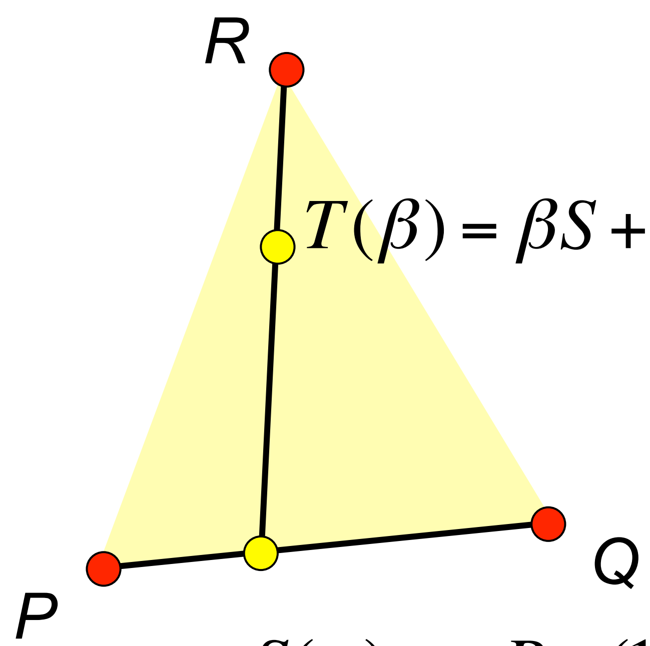
  - …is the ***distance*** between P and Q.
  - We now have the foundation describing the geometric world we will be using.

Spaces

- Planes
  - So far we have vectors, points (and lines)
  - A *plane* exists in *affine* space and it defined by three points; P, Q and R.
    - All the points of a line can be found via: (you know this)
      $$S(\alpha) = \alpha P + (1 - \alpha)Q,\ 0 \le \alpha \le 1$$
    - Picking an arbitrary point on this line and connecting it to a point R we get the second line:
      $$T(\beta) = \beta S + (1 - \beta)R,\ 0 \le \beta \le 1$$
    - All these points defined by α and β form the plane defined by points P, Q and R.

Spaces

- Planes (a picture helps)



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

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

UCLA

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*.
    - Point is, (Q-P) and (R-P) are arbitrary vectors.
    - So a plane can be defined by two vectors and a point, 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 the ***cross product***.

$$n = u \times v$$

  - This new vector is known as the ***normal*** to the plane.
    - We will find **many uses** for the normal in this course.

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

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

UCLA

Homogeneous Coordinates

- Recall, a *frame* for an *affine* space is given by
  - a ***basis*** and an origin point P_0

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

  - Then, any point p within the *frame* can be written as

$$p = \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n + 1 P_0$$

  - The $\mathbf{R}^3$ coordinate vectors of the vector v and point p can be written as
  - Notice the 0 and 1
  - 0 is nowhere
  - 1 plants it in space

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

UCLA

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.
  - Let's define a ***scale*** transformation matrix.

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

  - Applying the transformation gives

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

  - No problem.

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.

Transformations

- Scale, Rotation and *Translation*
    - Unfortunately, we cannot perform a ***translation*** with a matrix-vector multiplication.
    - This is a problem – we want 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* as such

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

  - Things start to work 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?
  - 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.

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

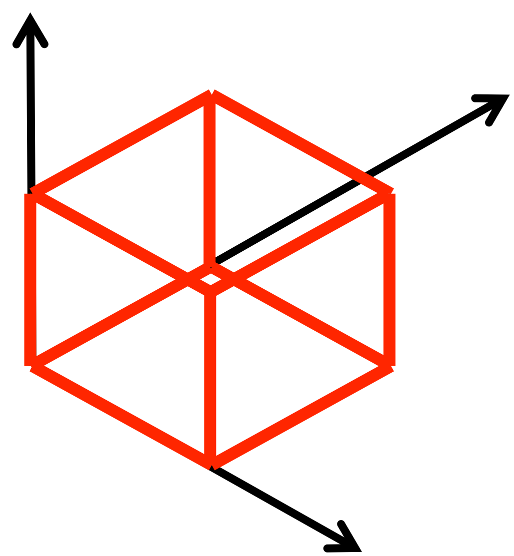
  - Rotation follows in the same way.
  - The same hold true for 3D.

OpenGL Frames

- There are, traditionally, six coordinate frames used in the OpenGL pipeline.
    1. Object (model) coordinates
    2. World coordinates
    3. Eye (camera) coordinates
    4. Clip coordinates
    5. Normalized device coordinates
    6. Window (screen) coordinates

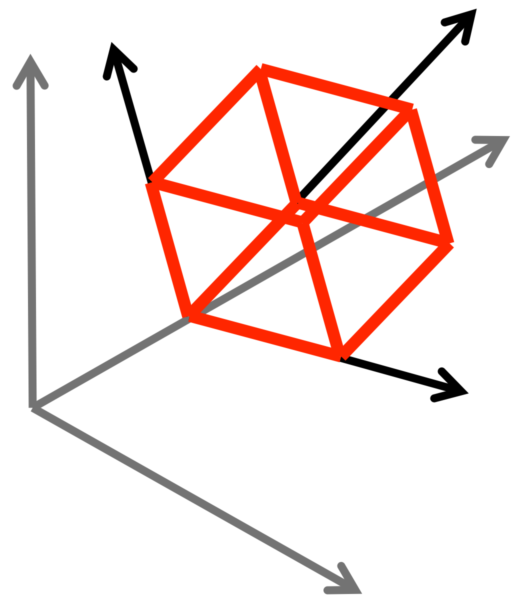
OpenGL Frames

- An *object* is represented in Object (model) coordinates.
  - This is a local *frame* of the object that is convenient to model the geometry.



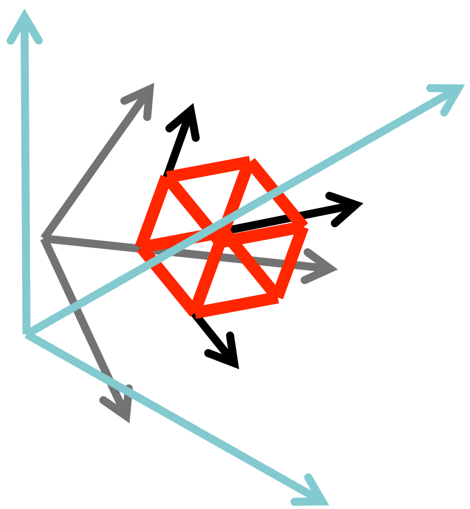
OpenGL Frames

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



OpenGL Frames

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



OpenGL 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 *view volume* after the projection transformation.
    - It is easiest to do this when we transform the *view volume* into a cube centered around the origin.
        - Recall the "Normalized View Volume"
        - The hardware can perform clipping *very quickly* here.

UCLA

OpenGL Frames

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

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

OpenGL Frames

- Clip, Normalized device & *Window* coordinates
  - Finally, *normalized device* coordinates are transformed into *window* coordinates.
  - The result are 3D points in *window* coordinates.
  - Since *window* (screen) coordinates are 2D the pipeline just drops the z (depth) value.
  - Those *window* (screen) coordinates are pixel locations defined by the *viewport*.

OpenGL Camera

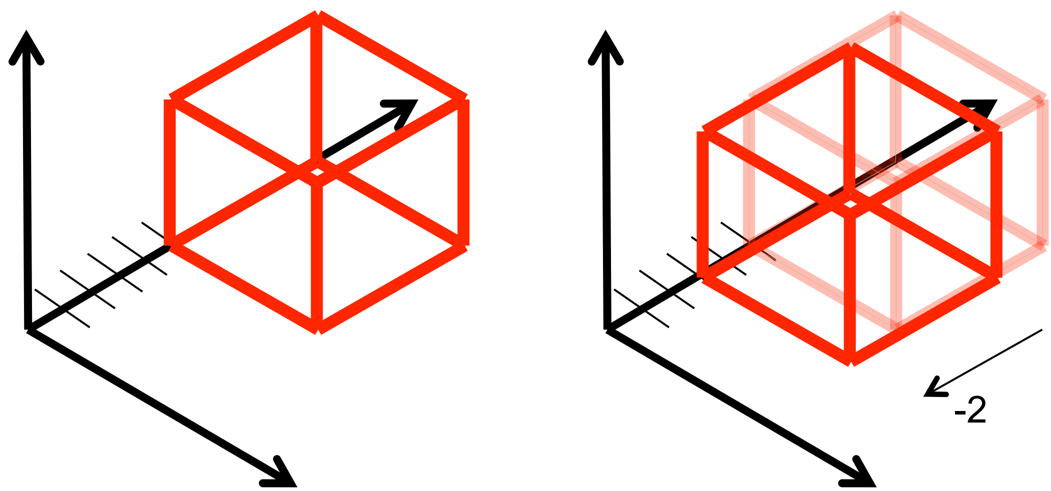
- There is *no camera* per se in OpenGL
  - We control what we "see" by manipulating the ***model-view*** transformation.

  - There are two ways to think of this.
    1. Move the ***world*** to the ***eye*** coordinate frame.
    2. Move the ***eye*** 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 the ***eye***.

OpenGL Camera

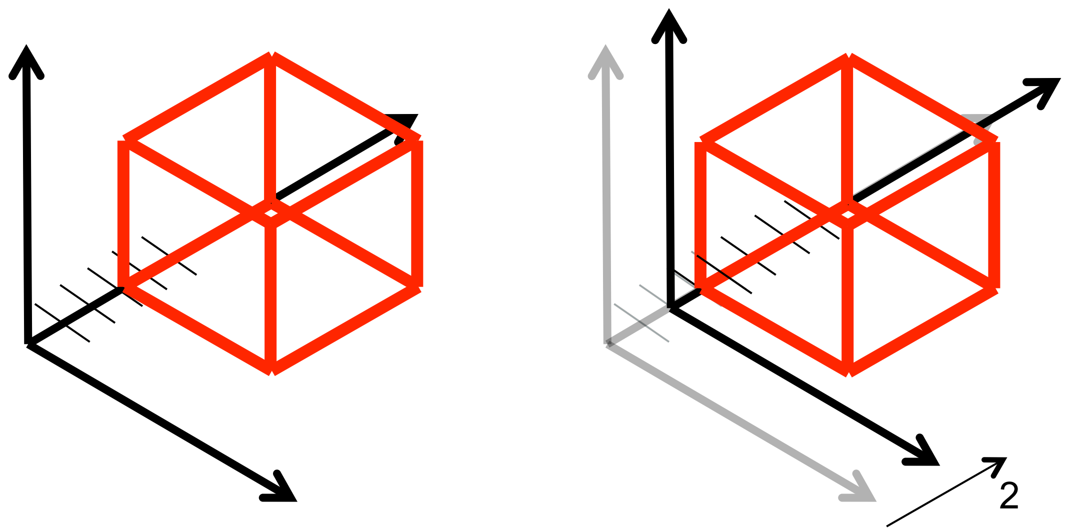
- Camera fixed at the origin
  - We transform the *world* in front of our *eye*.
  - If we wanted to move our view forward by 2.
    - We would transform the *world* frame by -2.



UCLA

OpenGL Camera

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



OpenGL Transformations

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

OpenGL 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 OpenGL represents matrices as **column vectors**.
  - So, for example, $T[3][1] = \Delta y$

OpenGL 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 OpenGL represents matrices as **column vectors.**
  - So, for example, $T[2][2] = \beta_z$

OpenGL 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

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

UCLA

OpenGL 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.

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

OpenGL 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.

OpenGL Transformations

- Matrix multiplication *is not* commutative.
  - When performing multiple rotation transformations.
    - However, when performing a series of transformation of a particular type the order does not matter. Try it.
      - » i.e. series of rotations
    - The **order** of transformation ***types*** matters.
      - » i.e. $q=TRSTp$ is not the same as $q=TSRTp$

OpenGL 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 with the current transformation matrix.

$$q = TR_zTp$$

$$C \leftarrow I$$

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

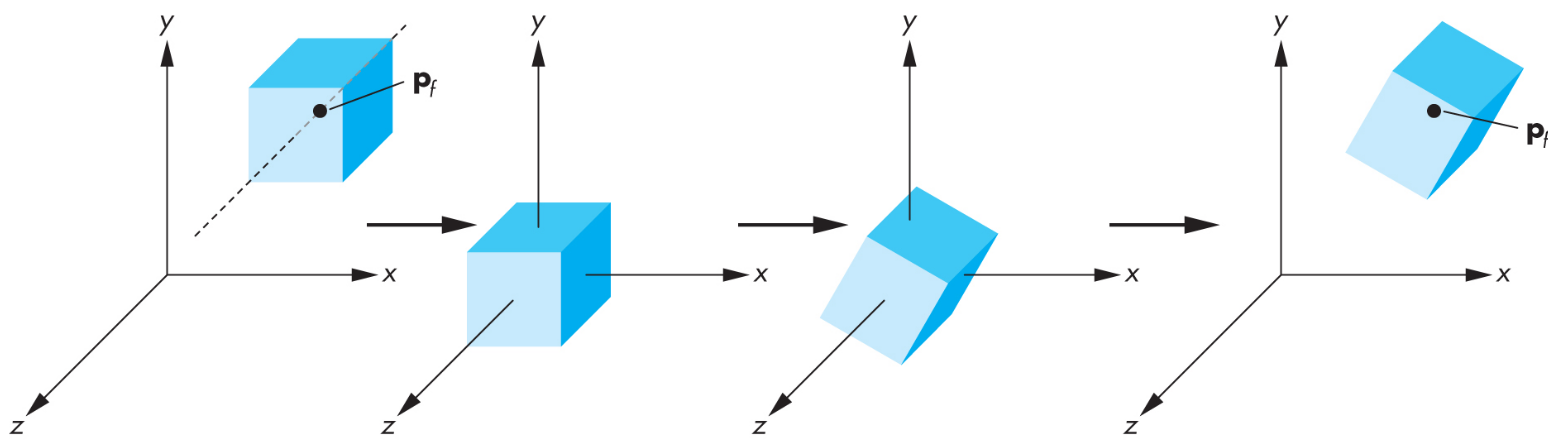
$$C \leftarrow CR_z(45.0)$$

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

$$q = Cp$$

OpenGL 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.



UCLA

OpenGL Transformations

- How do we apply transformations to geometry?
  - We could apply it to our geometry in our application.
    - Update buffer by call `glBufferData()`, although specifying 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.

UCLA

OpenGL 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.
        - the setup is very similar to passing the vertex data.

OpenGL 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( )
{
          // compile and link our shader program
          ...
          Glint matrixLoc;
          matrixLoc = glGetUniformLocation( program, "mTransformation" );
          ...
          // call when we want to update the value passed to the shader
          glUniformMatrix4fv( matrixLoc, 1, GL_TRUE, myMatrix );
          ...
}
```

OpenGL 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.
    - 3rd param indicates whether row major or not.

```
someFunc( )
{
          // compile and link our shader program
          ...
          Glint matrixLoc;
          matrixLoc = glGetUniformLocation( program, "mTransformation" );
          ...
          // call when we want to update the value passed to the shader
          glUniformMatrix4fv( matrixLoc, 1, GL_TRUE, myMatrix );
          ...
}
```

UCLA

OpenGL 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 assignment #1 by rotating the fractal around the origin.

  - Use keyboard to dynamically rotate fractal.

```
In vec4 vPosition;
Uniform mat4 mTransformation;

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

Next Time

- We will focus on
    - Perspective projection.
    - View transformations.
- Get started on your assignment!