Graphics Programming I

- Agenda:
  - Linear algebra primer
  - Transformations in OpenGL
  - Timing for animation
  - Begin first programming assignment

Linear algebra primer

- Three important data types:
  - Scalar values
  - Row / column vectors
    - 1x4 and 4x1 are the sizes we'll most often encounter
  - Square matrices
    - 4x4 is the size we'll most often encounter

Scalars

- These are the numbers you know!
  - Example: 3.14, 5.0, 99.9, $\sqrt{2}$, etc.

Row vectors

- These are special matrices that have multiple columns but only one row.
  - Example: $\begin{bmatrix} 5.0 & 3.14 & 37 \end{bmatrix}$
- Add and subtract the way you would expect.
  - Example: $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 9 & 10 & 11 \end{bmatrix} = \begin{bmatrix} 10 & 12 & 14 \end{bmatrix}$
  - Both vectors *must* be the same size.
- Operate with scalars the way you would expect.
  - Example: $3.2 \times \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 3.2 & 6.4 & 9.6 \end{bmatrix}$
- Notice that vector multiplication is missing...

Column vectors

- These are special matrices that have multiple rows but only one column.
  - Example: $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$
- Work just like row vectors.

Vector Operations

- There are only a few operations specific to vectors that are *really* important for us.

Dot Product

- Noted as a "dot" between two vectors (e.g., $A \cdot B$)
- Also known as "inner product."
- Multiply matching elements, sum all the results.
  - Example:

$$\begin{bmatrix} 2.3 & 1.2 \end{bmatrix} \cdot \begin{bmatrix} 1.7 & 6.5 \end{bmatrix} = (2.3 * 1.7) + (1.2 * 6.5) = 11.71$$

Vector Magnitude

- Noted by vertical bars, like absolute value.
- Take the square root of the dot product of the vector with itself...like absolute value.
- Result is the magnitude (a.k.a. length) of the vector.
  - Example: $\left\|\begin{bmatrix}\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2}\end{bmatrix}\right\| = \sqrt{\begin{bmatrix}\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2}\end{bmatrix} \cdot \begin{bmatrix}\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2}\end{bmatrix}} =$

$$\sqrt{\left(\frac{\sqrt{2}}{2}\right)^2 + \left(\frac{\sqrt{2}}{2}\right)^2} = \sqrt{\frac{2}{4} + \frac{2}{4}} = 1$$

Normalize

- Noted by dividing a vector by its magnitude.
  - Example: $\dfrac{A}{|A|}$
- Results in a vector with the same direction, but a magnitude of 1.0.
- Works the same as with scalars.

Why is the dot product so interesting?

- In 3-space, the dot of two unit vectors is the cosine of the angle between the two vectors.
  - If the vectors are not already normalized (unit length), we can divide the dot product by the magnitudes.
  - Example:

$$\frac{a \cdot b}{|a||b|} = \cos\theta$$

Cross Product

- Noted as an X between two vectors (e.g., $a \times b$)
- Derivation of the cross product is not important. The math is:

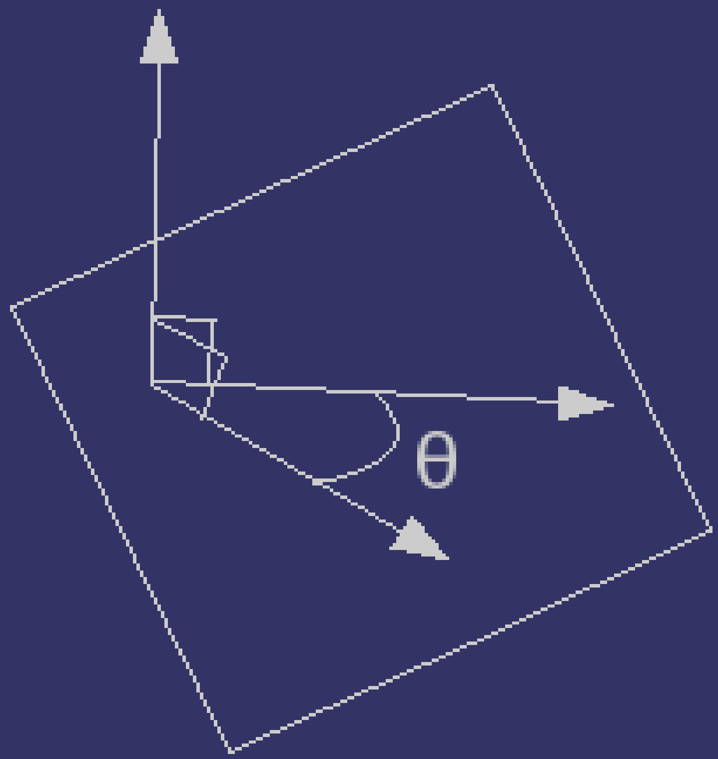
$$a \times b = \begin{bmatrix} a_y b_z - a_z b_y & a_z b_x - a_x b_z & a_x b_y - a_y b_x \end{bmatrix}$$

- *Only* valid in 3-dimensions.

Why is the cross product so interesting?

- Two really useful properties.
  - The result of the cross product between two vectors is a new vector that is perpendicular (also called normal) to both vectors.
  - If the source vectors are normalized:

$$|a \times b| = \sin\theta$$



Matrices

- Like vectors, but have multiple rows and columns.
  - Example: $\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$
- Add and subtract like you would expect.
  - Like vectors, both matrices must be the same size...in both dimensions.

Matrix / vector multiplication

- Special rules apply that make it different from scalar multiplication.
  - **Not** commutative! e.g., $M \times N \neq N \times M$
  - Is associative. e.g., $(NM)P = N(MP)$
  - Column count of first matrix must match row count of second matrix.
    - If M is a 4-by-3 matrix and N is a 3-by-1 matrix, we can do $M \times N$, but *not* $N \times M$.
  - If the source matrices are n-by-m and m-by-p, the resulting matrix will be n-by-p.

Matrix / vector multiplication (cont.)

- To calculate an element of the matrix, C, resulting from AB:

$$C_{ij} = \Sigma_{r=1}^{n} a_{ir} b_{rj}$$

- What does this look like?

Matrix / vector multiplication (cont.)

- To calculate an element of the matrix, C, resulting from AB:

$$C_{ij} = \Sigma_{r=1}^{n} a_{ir} b_{rj}$$

- What does this look like?
  - The dot product of a row of A with a column of B.
  - This is why the column count of A must match the row count of B...otherwise the dot product wouldn't work.

Multiplicative Identity

- There is an identity for matrix multiplication.

$$I = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

- Just like any other multiplicative identity, $AI = A$.
  - If you pretend that a scalar is a 1x1 matrix, this should make sense.

Transpose

- Noted by a "T" in the exponent position (e.g., M^T).
- The rows become the columns, and the columns become the rows.
  - Example:

$$\begin{bmatrix} 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{bmatrix}^T = \begin{bmatrix} 2 & 4 & 6 \\ 3 & 5 & 7 \end{bmatrix}$$

References

http://en.wikipedia.org/wiki/Matrix_multiplication

http://en.wikipedia.org/wiki/Dot_product

http://en.wikipedia.org/wiki/Cross_product

Rotation using matrices

- Rotation around the Z-axis.
  - If ϴ is 0, this is the identity matrix.
- Rotations around the Y-axis.

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Question!

- Looking at the previous equations, we can do a rotation using 4 multiplies and 2 adds, but the matrix multiply requires 16 multiplies and 12 adds.
  - $x' = x \cos \Theta + y \sin \Theta$
  - $y' = -x \sin \Theta + y \cos \Theta$
  - $z' = z$
- Why use the matrix method?

Matrices are more expressive!

- If we want to do a series of rotations with matrices we could do:

$$v' = M_1 v$$
$$v'' = M_2 v'$$
$$v''' = M_3 v''$$

- Which is the same as:

$$M_3 (M_2 (M_1 v))$$

- Look familiar?

Matrices are more expressive!

- If we want to do a series of rotations with matrices we could do:

$$v' = M_1 v$$
$$v'' = M_2 v'$$
$$v''' = M_3 v''$$

- Which is the same as:

$$M_3(M_2(M_1 v))$$

- Look familiar?
  - Matrix multiplication is associative!

$$(M_3 M_2 M_1) v$$

Translations with Matrices

- Points are stored as $p = [\, x\ y\ z\ 1 \,]$
- Remember the definition of matrix multiplication:

$$\begin{aligned}
p_x' &= p_x M_{11} + p_y M_{12} + p_z M_{13} + p_w M_{14} \\
p_y' &= p_x M_{21} + p_y M_{22} + p_z M_{23} + p_w M_{24} \\
p_z' &= p_x M_{31} + p_y M_{32} + p_z M_{33} + p_w M_{34} \\
p_w' &= p_x M_{41} + p_y M_{42} + p_z M_{43} + p_w M_{44}
\end{aligned}$$

- Since p_w is always 1, the 4th column of the matrix acts as a translation.

Scaling with Matrices

- To scale, we just want to multiply each component by a scale factor. Piece of cake!

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

References

http://en.wikipedia.org/wiki/Rotation_matrix

Break

OpenGL Transformation Matrices

- OpenGL has several matrix “stacks.”
  - Each stack has a specific purpose.
  - Active stack selected with `glMatrixMode()`.
    - OpenGL is *modal*, so ones a matrix stack is selected, all matrix commands will affact that stack.
- For object transformations, use the `GL_MODELVIEW` matrix.

Loading Matrices

- Initialize to the identity matrix with `glLoadIdentity()`.
- Load the matrix from your program with `glLoadMatrix[fd]()`.
  - GL matrices are *column major*, but C stores 2-dimensional arrays *row major.* **Watch out!**
  - We'll use `glLoadTransposeMatrix[fd]()` later.
- Can read matrix back with `glGetFloatv()`.

Matrix Operations

- Multiply a matrix with the top of the stack using glMultMatrix[fd]().
  - This is a post-multiply. $M_{top} = M_{top} * M$
- glPushMatrix() and glPopMatrix() save and restore the current top of stack.
  - This means you can push, make changes, then pop to get back the previous matrix.
  - The stack has a limited depth, so don't go crazy!

Built-in Transformation Operators

- Several routines to create and concatenate common transformations:
  - `glRotate[fd]()` - Create a rotation around an abritrary vector.
  - `glTranslate[fd]()`
  - `glScale[fd]()`

Orthonormal Basis

- Yes, it's a mouthful...but what does it mean?
- A vector space where all of the components are *orthogonal* to each other, and each is *normal.*
  - Normal meaning unit length.
  - Orthogonal meaning at right angles to each other.
- All pure rotation matrices (i.e., no scaling) are orthonormal bases.
  - As is the identity matrix!
- But how is this **useful**?

Question!

- Q: Given a world position for a camera, a world position to point the camera at, and an “up” direction, how can we construct a transformation using just rotations and translations?

Question!

- Q: Given a world position for a camera, a world position to point the camera at, and an “up” direction, how can we construct a transformation using just rotations and translations?
- A: We can't. We can construct an orthonormal basis from those 3 vectors.

Camera "Look At" Matrix

- E is the eye position
- V is the point being viewed
- U is the "up" direction
- The function `gluLookAt` does this.

$$F = V - E$$

$$f = \frac{F}{|F|}, u = \frac{U}{|U|}$$

$$s = u \times f$$

$$t = s \times f$$

$$M = \begin{bmatrix} s_0 & s_1 & s_2 & -E_0 \\ t_0 & t_1 & t_2 & -E_1 \\ -f_0 & -f_1 & -f_2 & -E_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Example

$$U=\begin{bmatrix}0 & 1 & 0\end{bmatrix}, E=\begin{bmatrix}0 & 0 & 0\end{bmatrix}, V=\begin{bmatrix}\sin\frac{\pi}{2} & 0 & -\cos\frac{\pi}{2}\end{bmatrix}$$

$$f=V$$

$$s=f\times U=\begin{bmatrix}f_y U_z - f_z U_y & f_z U_x - f_x U_z & f_x U_y - f_y U_x\end{bmatrix}=\begin{bmatrix}\cos\frac{\pi}{2} & 0 & \sin\frac{\pi}{2}\end{bmatrix}$$

$$t=s\times f=\begin{bmatrix}s_y f_z - s_z f_y & s_z f_x - s_x f_z & s_x f_y - s_y f_x\end{bmatrix}=\begin{bmatrix}0 & -\left(\sin\frac{\pi}{2}\right)^2+\left(\cos\frac{\pi}{2}\right)^2 & 0\end{bmatrix}$$

$$M=\begin{bmatrix}\cos\frac{\pi}{2} & 0 & \sin\frac{\pi}{2} & 0\\ 0 & 1 & 0 & 0\\ -\sin\frac{\pi}{2} & 0 & \cos\frac{\pi}{2} & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Look familiar?

$$M = \begin{bmatrix} \cos\frac{\pi}{2} & 0 & \sin\frac{\pi}{2} & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\frac{\pi}{2} & 0 & \cos\frac{\pi}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

References

http://www.wikipedia.org/Orthonormal_basis

Break

Timing for Animation

- Animations should run at a consistent speed regardless of the rendering speed
  - If someone starts ripping a CD or compiling a C program in the background, the animation should not slow down.
- What do we need in order to accomplish this?

Timing for Animation

- Animations should run at a consistent speed regardless of the rendering speed
  - If someone starts ripping a CD or compiling a C program in the background, the animation should not slow down.
- What do we need in order to accomplish this?
  - We need to know how much time has elapsed since the last frame.
    - `SDL_GetTicks()` gives us this information.
  - We also need to think of animation in terms of how long it takes to do something.

Example

```
static Uint32 last_t = ~0;

// Calculate time elapsed since last frame.
// SDL_GetTicks returns the time in milliseconds.
const Uint32 t = SDL_GetTicks();

if (last_t != (Uint32) ~0) {
    float dt = (float)(t - last_t) / 1000.0;

    // One complete revolution every 3 seconds.
    rotation_angle += dt * ((2.0 * M_PI) / 3.0);
}

// Update last_t.
last_t = t;
```

Next week...

- Lighting and materials
- Second programming will be assigned.
- First programming assignment is due.

Legal Statement

- This work represents the view of the authors and does not necessarily represent the view of IBM or the Art Institute of Portland.
- OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.
- Khronos and OpenGL ES are trademarks of the Khronos Group.
- Other company, product, and service names may be trademarks or service marks of others.