Review for Graphics Midterm on Tuesday October 11th
also generally useful knowledge for future OpenGL programming

Shaders:

OpenGL is a Graphics Library. GLSL is a Shader Language.

We have 2 shaders, which run on the GPU:
- The vertex shader operates on vertices (points that define the primitive)
- The fragment shader operates on fragments (potential pixels)

We load the shaders into our application by passing their file name to an OpenGL function that attaches it to a program object.

In our OpenGL application we produce primitives, which get sent to the vertex shader. The vertex shader does its processing on a per-vertex basis and produces output vertices, which are then sent to the rasterizer. The rasterizer breaks up the primitive into fragments. The fragments are then sent to the fragment shader, which processes the fragments and produces pixels that get stored in the frame buffer and then sent to the screen.

There are 3 different shader variable qualifiers:
1. Attribute-qualified variables: can change at most once per vertex
    - user defined in application program
    - use "in" qualifier to get them in the shader
2. Uniform-qualified variables: constant for an entire primitive
    - can be changed in application and sent to shaders
    - cannot be changed in shader
3. Varying-qualified variables: passed from vertex shader to fragment shader
    - automatically interpolated by the rasterizer
    - use "out" in vertex shader and "in" in fragment shader
    - (old style used the "varying" qualifier)

GLUT:

GLUT is a windowing system that sits outside of OpenGL and allows us to easily write OpenGL programs. It also gives us ways of capturing events, e.g. mouse and keyboard events.

When we hit the end of our main method in an application program using GLUT, we enter the glutMainLoop. This is an infinite loop that listens for events and calls the callback functions that we've registered for each event.

Whenever we reach the end of this loop without calling any function, then the Idle callback function is called (assuming an Idle callback function was registered).

When we register a callback function, we are passing the name of the function, which is actually a pointer to the function.

The one callback function that we will definitely always register is the display callback, which is what draws the image. This is usually called at the end of the glutMainLoop.

You should never explicitly call your display function in the glutMainLoop. (Like, because you changed something and now you need to re-draw the screen). Instead, call glutPostRedisplay(), which marks the current window as needing to be redisplayed. Then, glut will call the display function for you, once it reaches the end of the next iteration of glutMainLoop.

(This saves you from redrawing the display a million times inside the loop, like once for every individual change that makes it need to be redrawn. Instead, you just mark it "needs to be redrawn" a million times, and it still only gets redrawn once at the end of the loop, which is all you need.)

Primitives:

Primitives we can draw in OpenGL:
- Point
- Line
- Line strip
- Line loop
- Triangle
- Triangle strip
- Triangle fan

Qualities we want a primitive to have:
- Simple: Edges don't cross each other
- Flat: All the vertices are in the same plane
- Convex: All the points on a line segment between any two points inside the polygon, are also inside the polygon

Triangles are guaranteed to have those properties, which is why OpenGL only draws triangle primitives.

Vector Spaces, Representation, etc:

Vector spaces consist of vectors with several operations defined:
- Inverse of a vector
- Multiplication of a vector by a scalar
- Addition of two vectors

There is also a zero vector (zero magnitude, undefined orientation)

Vectors lack position. In order to do anything useful with them, we need points. An "affine space" is a vector space that allows point operations. The following operations are defined in affine spaces:
- Vector/vector addition (head-to-tail rule)
- Scalar/vector multiplication (same direction, different magnitude)
- Point/vector addition (yields a point)
- Scalar/scalar operations
- Scalar/point operations: For any point P, define: 1*P = P, and 0*P = 0 (the zero vector)

Linear Independence and Frames:

- A "linear combination" of vectors is just the addition of some number of vectors, where each vector is multiplied by some scalar.
- If you set all the scalars to zero, then adding the vectors together would produce the zero vector.
- Linear Independence: If you have 3 or more vectors, and the ONLY WAY to form a linear combination of them that equals the zero vector is to set all the scalars to zero, then they are linearly INdependent.
- In other words, it is impossible to represent any one of the vectors in terms of the others.

Bonus footage:
- In a vector space, the maximum number of linearly independent vectors is fixed, and is called the "dimension", i.e. an n-dimensional space.
- In an n-dimensional space, any set of n linearly independent vectors form a "basis" for the space.

We form a "frame" by adding a single point, the origin, to the basis vectors in an affine space.

Typically we're dealing with 3-dimensional frames: a set of 3 linearly independent vectors, together with a point (representing the origin).

Vectors in an n-dimensional coordinate system are represented as a linear combination of the n vectors that make up the "basis" of that coordinate system. So, in 3 dimensions, given the basis v1, v2, v3:

A vector v is written as the linear combination: v = a1v1 + a2v2 + a3v3

That list of scalars, {a1, a2, a3}, is the "representation" of v with respect to the given basis. We can write the representation as a row or column array of scalars, e.g. a = [a1 a2 a3].

A 3-dimensional Frame is determined by (P0, v1, v2, v3). Within this frame, every vector can be written as:

v = a1v1 + a2v2 + a3v3,

and every point can be written as:

P = P0 + b1v1 + b2v2 + b3v3 (in other words, a vector pointing to that point from the origin).

Using that ^ example, the representation of v would look like [a1 a2 a3], and the representation of P would look like [b1 b2 b3]. But those look exactly alike! So we need to add something to distinguish them.

Homogenous Coordinate Systems:

Recall that we defined 1*P (a point) = P, and 0*P = 0 (the zero vector).
Therefore, any vector (v = a1v1 + a2v2 + a3v3) can be written as:

[a1 a2 a3 0] * [v1 v2 v3 P0].

And any point (P = P0 + b1v1 + b2v2 + b3v3) can be written as:

[b1 b2 b3 1] * [v1 v2 v3 P0].

These are called the "homogenous coordinate" representations, the size of which will always be 1+ however many dimensions we're using. The last coordinate will always be 0 if it's representing a vector, and 1 if it's representing a point*. That coordinate is called "w" (because that way we can call the other ones x, y, and z).

* If the last coordinate is not 1 or 0, that just means that w is being used as a scaling factor. It's still a 1 really, it's just that all the coordinates are being multiplied by some value. So you gotta divide everything by w if you want the regular representation of that point.

Transformations:

To transform a primitive, we derive a homogenous-coordinate transformation matrix representing the desired transformation, then multiply our primitive by that matrix. Complex transformation matrices can be built by combining simpler ones.

The standard geometrical transformations are:

Translation:

- Move a point to a new location
- Displacement determined by a vector d
- To move a whole object, translate all the points by the same vector
- Translation matrix:

$$T = \begin{array}{ccccl} 1 & 0 & 0 & dx & \\ 0 & 1 & 0 & dy & \\ 0 & 0 & 1 & dz & \\ 0 & 0 & 0 & 1 & \text{<-- because the new point is a point} \end{array}$$

- Translation matrices are additive: You can add two translation matrices together by adding the d components inside the matrix

Rotation:

- In 2D: Rotate a point about the origin by theta degrees.
- This is equivalent to rotating about the z-axis in 3D.
- In 3D: To rotate about a given axis, use the identity matrix for that axis's row and column, but for the other two axes, use:

$$\begin{array}{cc} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{array}$$

- For example, Rotation matrix for rotating about the x-axis:

$$R = \begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{array}$$

- Rotation matrices are additive. I think this means: the dot product of two rotation matrices (for angles theta1 and theta2) is the rotation matrix for the angle (theta1 + theta2).

Scaling:

- Expand or contract along each axis, with a fixed point of origin
- Determined by a vector s
- Scaling matrix:

$$S = \begin{matrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

- If the s-components are all equal, it's called "uniform scaling"
- Scaling matrices are multiplicative: You can multiply them by multiplying the individual s-components.
- **Reflection** is achieved with negative scale factors. For example, to reflect on the y-axis (in 2D), set sy = 1 and sx = -1.

Shear:

- I think we can mostly ignore this for the midterm. But we might need to know this: You can tell that a matrix is a shearing matrix if it has non-zero values in the non-diagonal part of the inner matrix. Anyway, about shearing:
- What is it:
  - Angel: "Pulling faces in opposite directions"
  - Wolfram: One plane remains fixed, and all other points are displaced in some direction parallel to that plane, by a distance that is proportional to their distance from that plane.
  - Me: So it looks like the object is leaning to the side, but the base doesn't move.
- Shearing a polygon doesn't change its area.
- Shear matrix (to shear along the x-axis):

$$H = \begin{matrix} 1 & \cot(\theta) & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

- But I don't understand that, or know how to apply it to other axes, and the matrices I found online for shearing along the x-axis are different from this. So whatever about shearing.

Midterm question clarification:

- He wants us to be able to identify a "pure rotation matrix", which (I think!) just means a matrix that is *purely* a rotation, i.e. it's not a composite matrix that includes any translation or scaling. In other words, it has the sines and cosines in the upper left 3x3 sub-matrix (minus the row/column for the axis it's rotating around), and nothing else.
- Similarly, to identify a *pure* translation matrix, it's the identity matrix with *only* the extra translation vector in the 4th column.
- Pure scaling matrix would be the identity matrix but with non-1 values in the diagonal.

FYI: Inverses of each transformation matrix:
- Translation: inverse of T(dx, dy, dz) = T(-dx, -dy, -dz)

- Rotation: inverse of R(theta) = R(-theta)
- Scaling: inverse of S(sx, sy, sz) = S(1/sx, 1/sy, 1/sz)

Rigid body transformations:

Transformations that do not change parallel lines, angles, or line lengths, are called rigid body transformations. The property of a matrix that preserves these properties is called “special orthogonal”.

Special Orthogonal:

Given this rotation matrix (in 2D), look at the 2x2 sub-matrix in the top left, and consider each row as a vector:

$$\begin{matrix} \text{cos(theta)} & \text{-sin(theta)} & 0 \\ \text{sin(theta)} & \text{cos(theta)} & 0 \\ 0 & 0 & 1 \end{matrix}$$

They are both unit vectors, and each is perpendicular to the other. Now consider each column as a vector. If the same is true, then this is a "special orthogonal" matrix, which means it preserves angles and lengths, aka it represents a "rigid body transformation”.

If you multiply two special orthogonal matrices together, the result will also be special orthogonal.

Affine Transformations:

Any arbitrary sequence of *rotation* and *translation* is going to be special orthogonal, but once we add *scaling*, we are no longer special orthogonal, we are "affine". Affine transformations preserve parallel lines, but angles and lengths may change.

Concatenation:

We can build composite transformation matrices by concatenating these standard transformation matrices together (by multiplying them).

Order matters:
- To rotate about a fixed point other than the origin, you first have to translate the object (move the fixed point to the origin), then rotate it, then move it back (using inverse of previous translation).
- Before scaling, also translate to the origin first, because scaling affects position.
- We often start with an object centered at the origin, oriented with the axis, and at a standard size. Then we apply an "instance transformation" which scales it, orients it, and moves it, all at once.

BUT! Remember to put the transformations in reverse order. If you want to translate to the origin first and then scale, you have to multiply your identity matrix by the *scaling* matrix first, *then* the translation matrix. Because math.

Changing coordinate systems:

Here's some background information that doesn't seem like it'll be on the test…

Consider two different bases:
a = [a1 a2 a3] = a1v1 + a2v2 + a3v3
b = [b1 b2 b3] = b1u1 + b2u2 + b3u3

Then each of the vectors that make up one of the bases can be written in terms of the other basis:
u1 = c11v1 + c12v2 + c13v3
u2 = c21v1 + c22v2 + c23v3
u3 = c31v1 + c32v2 + c33v3
which, if you just take the coefficients, produces a matrix:

$$M = \begin{matrix} c11 & c12 & c13 \\ c21 & c22 & c23 \\ c31 & c32 & c33 \end{matrix}$$

(the slides use gamma here, I'm using c because who cares)

Now, we can relate the two bases, a and b, like this:
a = M^T*b (M-transpose times b)

In homogenous coordinates, we can do the same thing, but with an additional coordinate. Consider two different frames:

a = (P0, v2, v2, v3)
b = (Q0, u1, u2, u3)

We can still represent every element in b in terms of a:

u1 = c11v1 + c12v2 + c13v3
u2 = c21v1 + c22v2 + c23v3
u3 = c31v1 + c32v2 + c33v3
Q0 = c41v1 + c42v2 + c43v3 + c44P0 <— c44 will be 1 since Q0 is a point

which defines a 4x4 matrix:

$$M = \begin{matrix} c11 & c12 & c13 & 0 \\ c21 & c22 & c23 & 0 \\ c31 & c32 & c33 & 0 \\ c41 & c42 & c43 & 1 \end{matrix}$$

Thus, within the two frames, any point or vector has a representation of the same form:

a = [a1 a2 a3 a4] in the first frame
b = [b1 b2 b3 b4] in the second frame
where a4 and b4 = 1 for points or = 0 for vectors, and
a = M-transpose * b

The matrix M is 4x4 and specifies an affine transformation in homogenous coordinates.

Here's the useful information…

How to determine the values in M:
- This is what he was drawing up on the board.
- Say you have two bases (coordinate systems), A and B. If you want to represent the points in A within B's coordinate system, then you need M to represent the transformation from A to B. This will probably include translating A's origin to B's origin, then scaling A to match B's scale (and possibly rotating A to match B, but not for this midterm)
- Translating: Using B as your coordinate system, where is A's origin? If it's at (2,3), then the translation vector is [2 3]. You could think of it like: "The point (0,0) in A is (2,3) in B, so translate all points in A by [+2 +3]".
- Scaling: Think of it like "One tick in A's scale is __ times as big as a tick in B's scale". If it's half as big, the scaling factor is 1/2. If it's twice as big, the scaling factor is 2, etc.

Every linear transformation is equivalent to a change in frames!

Why the hell would we do it that way? (this isn't on the midterm):

Because it simplifies our frequent transitions between the World and Camera frames.

In OpenGL we work with n-tuples. Changes in frame are then defined by 4x4 matrices. We start with the World frame, and eventually represent those entities in the Camera frame.

If the two frames were the same, then the transformation matrix between them would be the Identity matrix (M=I). So we can "move the camera", for example, backwards on the z-axis by distance d, by changing the transformation matrix, like this:

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