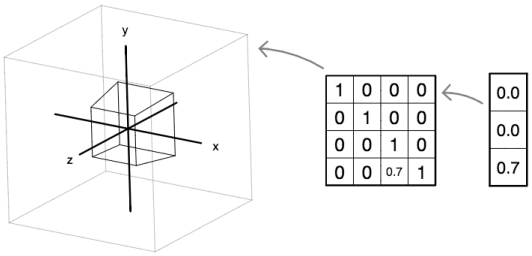
Notes for March 11 class -- More matrices

Perspective, part 1

Perspective is a linear transformation, which uses the bottom row of the 4×4 transformation matrix.

Which makes sense, because in order for a point (x,y,z,w) to "go out to infinity" or to "come in from infinity", the matrix needs to change the homogeneous coordinate of that point either from or to zero.



Perspective, part 2

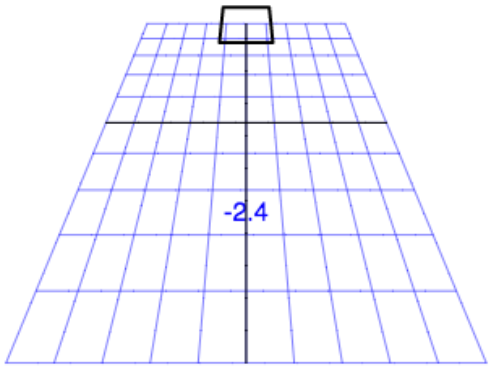
The figure to the right is a top-down view of what happens when we apply perspective transformation to z. In this case the "focal length" is -2.4. That is, the "point at infinity" (0,0,-1,0), also known, as the negative z direction, has been moved to (0,0,-2.4,1).

Practically speaking, if your camera is placed at the origin (0,0,0,1), and your camera's focal length is is f (where f is a negative value in z), then you can just do the following to create a persective linear transformation equivalent to what you were doing in ray tracing:

$$(x,y,z,1) \rightarrow (x , y , 1 , z/f)$$

When we divide by the homogeneous coordinate, this turns into the point:

$$(fx/z , fy/z , f/z)$$



Parametric cylinder

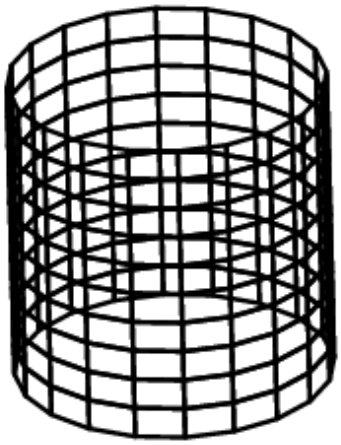
You can describe many surfaces parametrically, using the two parameters $0 \le u \le 1$ and $0 \le v \le 1$ to define values of x, y and z over the surface.

For example, the open cylindrical section to the right is described by:

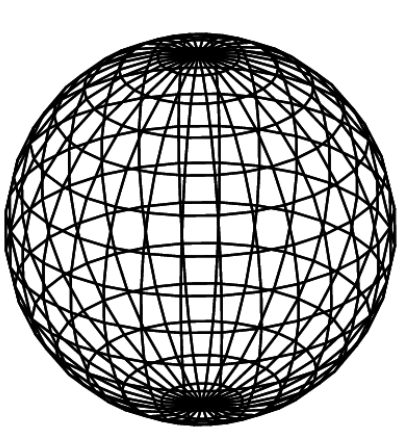
$$x = \sin(\theta)$$
$$y = 2 * v - 1$$
$$z = \cos(\theta)$$

where:

$$\theta = 2 \pi u$$



Parametric sphere



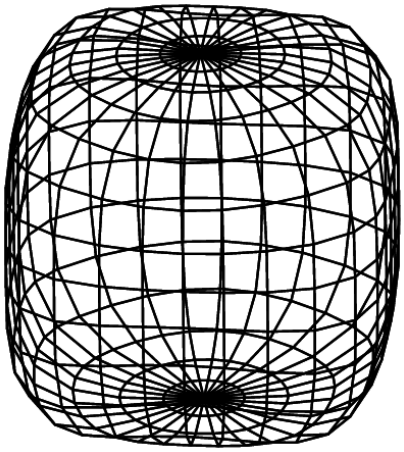
Similarly, the longitude / latitude parameterization of a sphere to the right is described by:

$$x = \cos(\phi) * \sin(\theta)$$
$$y = \sin(\phi)$$
$$z = \cos(\phi) * \cos(\theta)$$

where:

$$\theta = 2 \pi u$$
$$\phi = \pi v - \pi / 2$$

Parametric superquadric



We can also modify parameterizations procedurally. For example, the image to the right shows a *superquadric* surface, a surface consisting of points for which $x^p + y^p + z^p = 1$, for some positive value of p.

This shape was formed by modifying the previous example.

In this case, rather than following the constraint $x^2 + y^2 + z^2 = 1$, points on the surface follow the constraint $x^4 + y^4 + z^4 = 1$, points on the surface follow the constraint.

This shape was made by scaling each vertex (x,y,z) of the sphere by:

$$\frac{1}{(x^4 + y^4 + z^4)^{1/4}}$$

Chaining matrix transformations

Generally speaking, computer graphics scenes contain many vertices, and each of those vertices may go through multiple transformations. In practice we only need to transform any vertex once, because linear transformations are *associative*. In other words, the result of $(A \times B) \times C$ is the same as $A \times (B \times C)$, if A, B and C are transformation matrices.

This also means that the two following operations are equivalent:

(1) return $A \times (B \times (C \times p))$ *Apply three successive linear transformations to a point*

(2) return $M \times p$

where *Form a single linear transformation, and apply that transformation to the point.*

$$M = A \times B \times C$$

Matrix multiply

In order to form a composite matrix such as the matrix M in the above discussion, we need to be able to multiply matrices.

Matrix multiplication $A \times B$ between two matrices A and B proceeds by multiplying each of the four row vectors that constitute A by each of the four column vectors that constitute B:

$$\begin{matrix} A_{0,0} & A_{1,0} & A_{2,0} & A_{3,0} \end{matrix} \qquad \begin{matrix} B_{0,0} \\ B_{1,0} \\ B_{2,0} \\ B_{3,0} \end{matrix}$$

$$C = \begin{matrix} A_{0,1} & A_{1,1} & A_{2,1} & A_{3,1} \\ A_{0,2} & A_{1,2} & A_{2,2} & A_{3,2} \\ A_{0,3} & A_{1,3} & A_{2,3} & A_{3,3} \end{matrix} \times \begin{matrix} B_{0,1} & B_{1,1} & B_{2,1} & B_{3,1} \\ B_{0,2} & B_{1,2} & B_{2,2} & B_{3,2} \\ B_{0,3} & B_{1,3} & B_{2,3} & B_{3,3} \end{matrix}$$

So the value of any entry in the result matrix $C_{col,row}$ is:

$$C_{col,row} = A_{(0,row)} B_{(col,0)} + A_{(1,row)} B_{(col,1)} + A_{(2,row)} B_{(col,2)} + A_{(3,row)} B_{(col,3)}$$

Building a matrix by applying successive primitives

In order to animate objects, as in the swinging arm example we showed in class, you will want to keep an *composite* matrix M. At every animation frame, set the value of M to identity. Then apply successive linear transformations to M, at each step multiplying it on the right (locally) by one of your primitives (translate, rotate or scale):

```
M ← M × T    // translate
M ← M × R    // rotate
M ← M × S    // scale
```

Then you can use matrix M to transform each of the vertices of a shape.

After you have transformed any vertex, you will want to apply a perspective transform to that transformed vertex, followed by the viewPort transform. Then you can draw your edges onto the canvas.

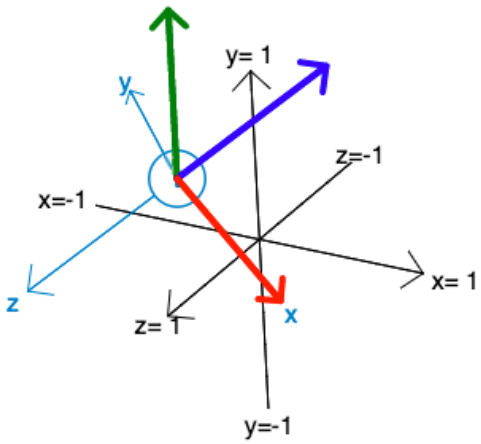
Defining a camera matrix

In order to fly a camera around a scene, you need to do two things:

1. Define a camera matrix C
2. Replace M by $C^{-1}M$

In other words, you need to multiply on the left (globally) by the inverse of the C matrix.

To compute C, you need to know the location of the camera (its *origin*) and the direction it is aiming (its *aim vector*), as in the figure to the right.



- The origin will be the translation (the rightmost column of C) of your C matrix.
- C_z, the z axis of C (the third column of C), will be the negative of your aim vector.
- To compute the C_x, the x axis of C (the leftmost column of C), take the normalized cross product of the global y (upward) direction [0,1,0] and C_z:

  C_x = normalize([0,1,0] × C_z)

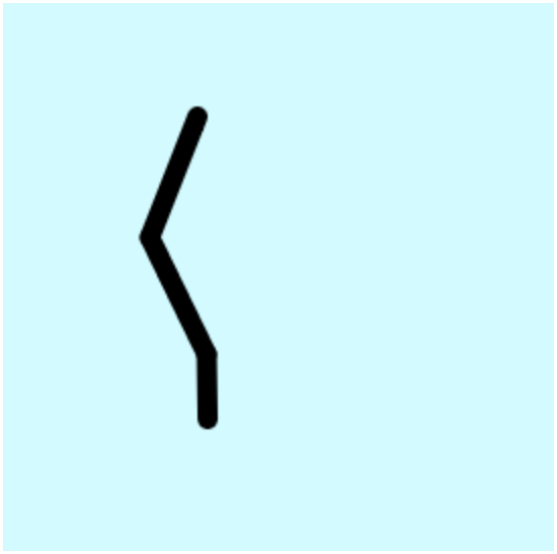
- Finally, get C_y (the second column of C) by: $C_z \times C_x$.

Here is the javascript code for a very simple matrix inversion routine that you can use to invert your camera matrix, which takes arrays of length 16 as input and output.

This implementation of matrix inversion doesn't handle every case, but it will handle the special case where the x,y,z axes of the source matrix are of the same magnitude and are perpendicular to each other (which is the case here).

Swinging arm example

In class we showed the example of a swinging arm. The code we wrote in class is below.



If you already have a matrix object implemented, and you implement a reasonable drawLine() method, as well as methods that apply translate, rotate and scale cumulatively to the matrix value, then the below code will result in a swinging arm like the one we showed in class (right).

```
var origin     = new Vector3(0,0,0);
var shoulder   = new Vector3();
var elbow      = new Vector3();
var wrist      = new Vector3();
var fingertips = new Vector3();

var m = new Matrix();
canvas.update = function(g) {
   this.g = g; // so the drawLine method will know where to draw to.

   var theta = 3 * time;

   g.lineCap = "round";
   g.lineJoin = "round";

   g.fillStyle = 'rgb(220,250,255)';
   g.beginPath();
   g.moveTo(0,0);
   g.lineTo(this.width,0);
   g.lineTo(this.width,this.height);
   g.lineTo(0,this.height);
   g.fill();

   g.lineWidth = 10;

   m.identity();

   m.translate(.2,.3,0);

   m.rotateZ(Math.cos(theta) * .5);
   m.transform(origin, shoulder);

   m.translate(0,-.2,0);

   m.transform(origin, elbow);

   m.rotateZ(Math.sin(theta) * .5 + .5);
   m.translate(0,-.2,0);

   m.transform(origin, wrist);

   m.rotateZ(Math.cos(theta) * .25 - .25);
   m.translate(0,-.1,0);

   m.transform(origin, fingertips);

   this.drawLine(shoulder, elbow);
   this.drawLine(elbow, wrist);
   this.drawLine(wrist, fingertips);
}
```

Homework, due by start of class on Wednesday March 25

- Add simple perspective to your scenes (that is, replace (x,y,z) by (fx/z,fy/z,f/z)).
- Create some interesting parametric surfaces. Try to make them time-varying. Use interesting time-varying functions. For example, you can try to animate shapes using this Javascript implementation of the Noise function.
- Change the translate, rotateX, rotateY, rotateZ and scale methods from your previous assignment so that rather than just setting your matrix to be a translation, rotation or scale matrix, they instead modify an existing matrix value.

  For example, your new `M.translate(x,y,z)` method should modify the value of matrix M as follows:

  1. Given translation values (x,y,z), compute the value of a translation matrix T (as in your previous assignment);
  2. Replace the value of your matrix M by the matrix product M×T.
- Build an animated scene with multiple moving parts, by alternating successive changes to a matrix M at each animation frame with commands to draw primitive objects (such as boxes, cylinders and spheres). You can draw objects as edge-connected vertices, as in the previous assignment.

  Don't forget to reset the value of M by calling `M.identity()` at the start of every animation frame.

As always, you get extra points for making something that is fun, exciting, beautiful or original.
