3. Geometric Modeling

Reading

Recommended:

- Foley *et al.*, Chapter 5
- David F. Rogers and J. Alan Adams, *Mathematical Elements for Computer Graphics*, Second edition, McGraw-Hill, New York, 1990, Chapters 2–3.
- Edward Angel, *Interactive Computer Graphics*, Addison-Wesley, 1997, Sections 8.1–8.6.

Outline

1. 2-D transformations
2. 3-D transformations
3. Perspective projections
4. Hierarchical modeling
5. Keyframe animation

Representation

We can represent a *point* $p = (x, y)$ in the plane

- as a column vector $\begin{bmatrix} x \\ y \end{bmatrix}$
- as a row vector $\begin{bmatrix} x & y \end{bmatrix}$

Representation, cont.

We can represent a *2-D transformation* M by a matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

If p is a column vector, M goes on the left:

$$p' = Mp$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

If p is a row vector, M^{T} goes on the right:

$$p' = pT$$

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} a & c \\ b & d \end{bmatrix}$$

We will use *column vectors*.

Two-dimensional transformations

Here's all you get with a 2×2 transformation matrix M:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

So

$$x' = ax + by$$

$$y' = cx + dy$$

We will develop some intimacy with the elements $a, b, c, d \ldots$.

Identity

Suppose we choose $a = d = 1, b = c = 0$:

- Gives the "identity" matrix

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

- Doesn't move the points at all

Scaling

Suppose we set $b = c = 0$, but let a and d take on any positive value:

- Gives a "scaling" matrix

$$\begin{bmatrix} a & 0 \\ 0 & d \end{bmatrix}$$

- Provides differential scaling in x and y:

$$x' = ax$$

$$y' = dy$$

Suppose we keep $b = c = 0$, but let a and d go negative.

Examples:

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$$

Now let's leave $a = d = 1$ and experiment with c. . . .

The matrix

$$\begin{bmatrix} 1 & 0 \\ c & 1 \end{bmatrix}$$

gives:

$$\begin{aligned} x' &= x \\ y' &= cx + y \end{aligned}$$

Effect is called a "_____."

Effect on unit square

Let's see how a general 2×2 transformation M affects the unit square:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & a & a+b & b \\ 0 & c & c+d & d \end{bmatrix}$$

Effect on unit square, cont.

Observe:

- Origin invariant under M
- M can be determined just by knowing how the corners $(1, 0)$ and $(0, 1)$ are mapped
- a and d give x- and y-scaling
- b and c give x- and y-shearing

Rotation

From our observations of the effect on the unit square, it should be easy to write down a matrix for "rotation about the origin":

- $\begin{bmatrix} 1 \\ 0 \end{bmatrix} \rightarrow$
- $\begin{bmatrix} 0 \\ 1 \end{bmatrix} \rightarrow$

Can also be viewed as a simultaneous "squash and shear."

Limitations of the 2×2 matrix

A 2×2 matrix allows

- Scaling
- Rotation
- Reflection
- Shearing

Q: What important operation does that leave out?

Homogeneous coordinates

Idea is to loft the problem up into 3-space, adding a third component to every point:

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

And then transform with a 3×3 matrix:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & m \\ 0 & 1 & n \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

. . . Gives *translation*!

Rotation about arbitrary points

Until now, we have only considered rotation about the origin.

With homogeneous coordinates, you can specify rotations about any point q with a matrix:

1. Translate to q
2. Rotate
3. Translate back

Note: Transformation order is important!

Mathematical properties of affine transformations

All of the transformations we've looked at so far are examples of "affine transformations."

Here are some useful properties of affine transformations:

- Lines map to lines
- Parallel lines remain parallel
- Midpoints map to midpoints
  (in fact, ratios are always preserved)

Basic 3-D transformations

The basic 3-D transformations are just like the 2-D ones:

Translation

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Scaling

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Basic 3-D transformations, cont.

Rotation

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

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

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

Projections

"Projections" transform points in n-space to m-space, where $m < n$.

In 3-D, we map points from 3-space to the "projection plane" (PP) along "projectors" emanating from the "center of projection" (COP):

There are two basic types of projections:

- "Perspective" — distance from COP to PP finite
- "Parallel" — distance from COP to PP infinite

Perspective vs. parallel projection

Perspective projection pros and cons:

+ Size varies inversely with distance — looks realistic
− Distance and angles are not (in general) preserved
− Parallel lines do not (in general) remain parallel

Parallel projection pros and cons:

− Less realistic looking
+ Good for exact measurements
+ Parallel lines remain parallel
− Angles not (in general) preserved

Vanishing points

Under perspective projections, any set of parallel lines that are not parallel to the PP will converge to a "vanishing point."

Vanishing points of lines parallel to a principal axis x, y, or z are called "principal vanishing points."

There are at most _____ of these.

Types of perspective drawing

Perspective drawings are often classified by the number of principal vanishing points.

- One-point perspective — simplest to draw
- Two-point perspective — gives better impression of depth
- Three-point perspective — most difficult to draw

All three types are equally simple with computer graphics.

Parallel projections

For parallel projections, we specify a "direction of projection" (DOP) instead of a COP.

There are two types of parallel projections:

- "Orthographic projection" — DOP perpendicular to PP
- "Oblique projection" — DOP not perpendicular to PP

There are two kinds of oblique projections:

- "Cavalier projection"
  - DOP makes 45° angle with PP
  - Does not foreshorten lines perpendicular to PP
- "Cabinet projection"
  - DOP makes 63.4° angle with PP
  - Foreshortens lines perpendicular to PP by one-half

Mathematical properties of projective transformations

The perspective projection is an example of a "projective transformation."

Here are some properties of projective transformations:

- Lines map to lines
- Parallel lines do not necessarily remain parallel
- Ratios are not preserved (but "cross-ratios" are):

A non-ratio-preserving projection (Farin, fig 14.1)

Coordinate systems for computer graphics

The real computer graphics guru uses lots of different coordinate systems:

- "*Model space*" — for describing the objects (a.k.a. "object space," a.k.a. "local space")
- "*World space*" — for assembling collections of objects (a.k.a. "object space," a.k.a. "problem space," a.k.a. "application space")
- "*Eye space*" — a canonical space for viewing (a.k.a. "camera space")
- "*Screen space*" — result of perspective transformation (a.k.a. "normalized device coordinate space," a.k.a. "normalized projection space," a.k.a. "3D screen space")
- "*Image space*" — a 2-D space that uses device coordinates (a.k.a. "window space," a.k.a "screen space," a.k.a. "normalized device coordinate space," a.k.a. "raster space," a.k.a. "pixel space")

A typical eye space

- "Eye"
  - Acts as the COP
  - Placed at the origin
  - Looks down the z-axis (gives left-handed coordinate system)
- "Screen"
  - Lies in the PP
  - Perpendicular to z-axis
  - At distance D from the eye
  - Centered on z-axis, with radius S

Q: Which objects are visible?

Projecting eye space into screen space

Q: How do we perform the perspective projection from eye space into screen space?

Using similar triangles gives:

Projecting eye space into screen space, cont.

We can write this transformation in matrix form:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/D & 0 \end{bmatrix}$$

General perspective projection

Now, at last, we can see what the "last row" does.

In general, the matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ p & q & r & s \end{bmatrix}$$

performs a perspective projection into the plane $px + qy + rz + s = 1$.

Q: Suppose we have a cube C whose edges are aligned with the principal axes. Which matrices give drawings of C with

- one-point perspective
- two-point perspective
- three-point perspective?

Perspective depth

Q: What did our perspective projection do to z?

Often, it's useful to have a z around — e.g., for hidden surface calculations.

Hither and yon planes

In order to preserve depth, we set up two planes:

- The "hither" plane $z_e = N$
- The "yon" plane $z_e = F$

We'll map:

$$z_e = N \;\Rightarrow\; z_s = 0$$

$$z_e = F \;\Rightarrow\; z_s = 1$$

Exercise: Derive the matrix to do this projection.

Symbols and instances

Most graphics API's support a few geometric "primitives" — e.g.:

- spheres
- cubes
- cylinders

These "symbols" (or "masters") are "instanced" using an "instance transformation":

Instance transformation (Angel, fig 8.2)

Q: What is the matrix for the instance transformation above?

Instancing in OpenGL

In OpenGL, instancing is created by modifying the "model-view matrix":

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef( ... );
glRotatef( ... );
glScalef( ... );
cylinder();
```


Hierarchical modeling

Hierarchical models can be composed of instances using trees or DAGS:

Two representations of a simple car (Angel, figs 8.6 and 8.7)

- Edges contain geometric transformations
- Nodes contain geometry (and possibly drawing attributes)

Example: A robot arm

Consider this robot arm with 3 degrees of freedom:

1. Base rotates about its vertical axis by θ
2. Lower arm rotates in its xy-plane by ϕ
3. Upper arm rotates in its xy-plane by ψ

A robot arm (Angel, fig. 8.8)

Q: What is the tree structure of the robot?

Q: What matrix do we use to transform the base?

Q: What matrix for the lower arm?

Q: What matrix for the upper arm?

Robot arm implementation

The robot arm can be displayed by altering the model-view matrix incrementally:

```
display()
{
   glRotatef(theta, 0., 1., 0.);
   base();
   glTranslatef(0., h1, 0.);
   glRotatef(phi, 0., 0., 1.);
   lower_arm();
   glTranslatef(0., h2, 0.);
   glRotatef(psi, 0., 0., 1.);
   upper_arm();
}
```


A more complex example: Human figure

Human figure (Angel, figs. 8.12 and 8.14)

Q: What's the most sensible way to traverse this tree?

Human figure implementation

The traversal can be implemented by pushing the model-view matrix onto a stack:

```
figure()
{
   torso();
   glPushMatrix();
      glTranslate
      glRotate3
      head();
   glPopMatrix();
   glPushMatrix();
      glTranslate
      glRotate3
      left_upper_leg();
      glTranslate
      glRotate3
      left_lower_leg();
   glPopMatrix();
   glPushMatrix();
      .
      .
}
```


Animation

The above examples are called "articulated models":

- rigid parts
- connected by joints

They can be animated by specifying the joint angles (or other display parameters) as functions of time.

(Note that each one can be changed independently.)

Kinematics and dynamics

Definitions:

- "Kinematics": How the positions of the parts vary as a function of joint angles.
- "Dynamics": How the positions of the parts vary as a function of applied forces.

Questions:

Q: What do the terms "inverse kinematics" and "inverse dynamics" mean?

Q: Why are these problems more difficult?

Key-frame animation

One way to get around these problems is to use "key-frame animation":

- Each joint specified at various "key frames" (not necessarily the same as other joints)
- System does the interpolation or "inbetweening"

Doing this well requires:

- A way of smoothly interpolating key frames: "splines"
- A good interactive system
- A lot of skill on the part of the animator!

Scene graphs

The idea of hierarchical modeling can be extended to an entire scene, encompassing:

- many different objects
- lights
- camera position

Called a "scene tree" or a "scene graph."

Summary

What to take away from this lecture:

1. How points and transformations are represented.
2. What all the elements of a 2×2 transformation matrix do.
3. What homogeneous coordinates are and how they work.
4. How to concatenate transformations.
5. The mathematical properties of affine vs. projective transformations.
6. The classification of different types of projections.
7. The concepts of vanishing points and one-, two-, and three-point perspective.
8. An appreciation for the various coordinate systems used in computer graphics.
9. How the persepctive transformation works.
10. How primitives can be instanced and composed to create hierarchical models using geometric transformations.

11. How this notion can be extended to entire scenes.
12. How keyframe animation works.