Computer Graphics

P. Healy

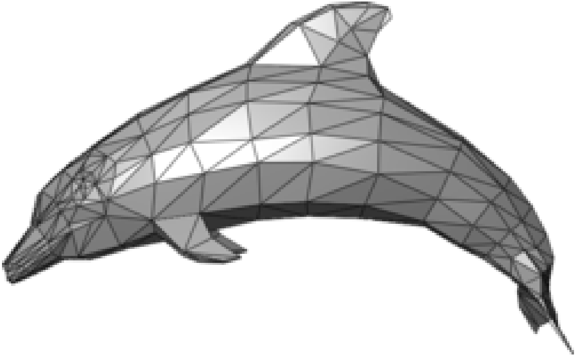
CS1-08
Computer Science Bldg.
tel: 202727
patrick.healy@ul.ie

Spring 2021–2022

Outline

1. Wire-Frame Meshes
2. Affine Transformations (contd.)
3. Composition of Affine Transformations
4. OpenGL Point and Line Functions – §4.{3,4}

Approximate Representations



Affine Transformations

- are preserved by an AT: the image of a straight line is another straight line but maybe not parallel to original!
  (guarantees polygons get mapped into polygons; but maybe not same "shape")
- remain parallel
- Line lengths do not get preserved; but between two lines do get preserved

Affine Transformations

- are preserved by an AT: the image of a straight line is another straight line but maybe not parallel to original!
- remain parallel (guarantees parallelograms get mapped into parallelograms); but maybe not same “shape”
- Line lengths do not get preserved; but between two lines do get preserved

Affine Transformations

- are preserved by an AT: the image of a straight line is another straight line but maybe not parallel to original!
- remain parallel
- Line lengths do not get preserved; but between two lines do get preserved
  (guarantees line mid-point gets mapped to line mid-point)

Affine Transformations

The five elementary affine transformations in 2-D are:

translation scaling

$$\begin{pmatrix} 1 & 0 & x_b \\ 0 & 1 & y_b \\ 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

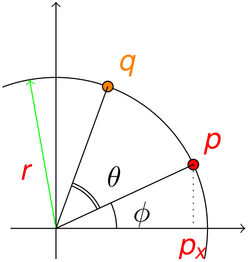
and

rotation reflection shearing

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} \pm 1 & \pm 1 & 0 \\ \pm 1 & \pm 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Affine Transformations: rotations

What is transformation matrix that **rotates** p below to q?

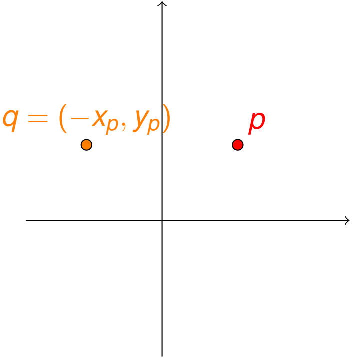


- We wish to rotate point p (originally at an angle of ϕ) through θ radians with respect to the origin – always the origin.

$$
\begin{aligned}
p &= (p_x, p_y) = (r\cos\phi, r\sin\phi) \\
q &= (r\cos(\theta+\phi), r\sin(\theta+\phi)) \\
&= (r(\cos\theta\cos\phi - \sin\theta\sin\phi), \\
&\quad\; r(\sin\theta\cos\phi + \cos\theta\sin\phi)) \\
&= \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} r\cos\phi \\ r\sin\phi \end{pmatrix} \\
&= \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} p
\end{aligned}
$$

Affine Transformations: reflections

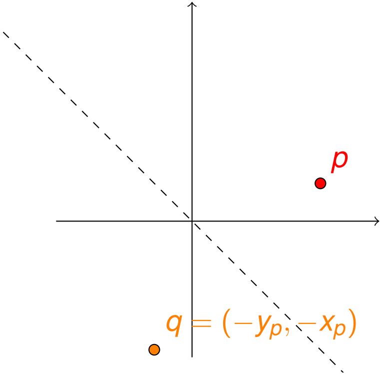
What is the transformation matrix that **reflects** $p = (x_p, y_p)$ below to q?



$$
\begin{aligned}
p &= (x_p, y_p) \\
q &= (-x_p, y_p) \\
&= \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_p \\ y_p \end{pmatrix} \\
&= \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} p
\end{aligned}
$$

Affine Transformations: reflections

What is the transformation matrix that **reflects** $p = (x_p, y_p)$ below to q?



$$p = (x_p, y_p)$$
$$q = (-y_p, -x_p)$$
$$= \begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} x_p \\ y_p \end{pmatrix}$$
$$= \begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix} p$$

Affine Transformations: shear

- An object is **sheared** by “slanting” it sideways
- This is achieved by adding to the x component of each point, a fraction of its y component
- Thus, points further north get slanted more
- Think of a deck of cards pushed sideways: the cards higher up (larger y) get pushed further sideways (larger x)

Affine Transformations: rotations and reflections

- If
$$R_\theta p = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} p$$
rotates p through an angle θ then applying a subsequent rotation of $-\theta$ via $R_{-\theta}$ should return us to original (undo first rotation). That is
$$p = R_{-\theta}(R_\theta p) = (R_{-\theta}R_\theta)p = Ip$$
A matrix is a rotation matrix if and only if $R_{-\theta}R_\theta = I$ holds
- Since a second reflection undoes the effect of an initial reflection it must be that
$$F^2 p = F(Fp) = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} p = Ip = p$$

AT Compositions

- Usually we want to perform more than one transformation on an object of interest e.g. performing a stretch, rotation and translation on an object
- It would be nice if we didn't have to perform each transformation on every point that defines the object before proceeding to next transformation
- By **composing** the transformations we can derive a single transformation (matrix) that is the net effect of all transformations

Composition of Affine Transformations

- Composing (combining) two affine transformations:

$$\vec{u'} = \begin{bmatrix} A & b \\ 0 & 1 \end{bmatrix} \vec{u}$$

and

$$\begin{aligned} \vec{u''} &= \begin{bmatrix} A' & b' \\ 0 & 1 \end{bmatrix} \vec{u'} \\ &= \begin{bmatrix} A' & b' \\ 0 & 1 \end{bmatrix} \begin{bmatrix} A & b \\ 0 & 1 \end{bmatrix} \vec{u} \\ &= \begin{bmatrix} A'A & A'b + b' \\ 0 & 1 \end{bmatrix} \vec{u} \end{aligned}$$

- Note that if $\vec{b} = 0$ the net effect is A' applied to A with a translation of $\vec{b'}$

Composition of Affine Transformations

- Composing (combining) two affine transformations:

$$\vec{u'} = \begin{bmatrix} A & b \\ 0 & 1 \end{bmatrix} \vec{u}$$

and

$$\begin{aligned} \vec{u''} &= \begin{bmatrix} A' & b' \\ 0 & 1 \end{bmatrix} \vec{u'} \\ &= \begin{bmatrix} A' & b' \\ 0 & 1 \end{bmatrix} \begin{bmatrix} A & b \\ 0 & 1 \end{bmatrix} \vec{u} \\ &= \begin{bmatrix} A'A & A'b + b' \\ 0 & 1 \end{bmatrix} \vec{u} \end{aligned}$$

- Note that if $\vec{b} = 0$ the net effect is A' applied to A with a translation of $\vec{b'}$

Specifying entities in OpenGL

- We need to “register” all points, lines, polygons, etc. in OpenGL
- We can register **primitives** in OpenGL as follows

```
glBegin (GL_XXXXX); // register a GL_XXXXX
 createPrimitivePart(data1);
 createPrimitivePart(data2);
    :
 createPrimitivePart(datan);
glEnd();
```

- OpenGL provides functions along the lines of `createPrimitivePart()` (*not its real name*) that enable it to keep track of the primitive
- The arguments to “`createPrimitivePart()`” specify the object

Specifying entities in OpenGL

- We need to “register” all points, lines, polygons, etc. in OpenGL
- We can register **primitives** in OpenGL as follows

```
glBegin (GL_XXXXX); // register a GL_XXXXX
 createPrimitivePart(data1);
 createPrimitivePart(data2);
    :
 createPrimitivePart(datan);
glEnd();
```

- OpenGL provides functions along the lines of `createPrimitivePart()` (*not its real name*) that enable it to keep track of the primitive
- The arguments to “`createPrimitivePart()`” specify the object

Specifying entities in OpenGL

- We need to “register” all points, lines, polygons, etc. in OpenGL
- We can register **primitives** in OpenGL as follows

```
glBegin (GL_XXXXX); // register a GL_XXXXX
 createPrimitivePart(data1);
 createPrimitivePart(data2);
   :
 createPrimitivePart(datan);
glEnd();
```

- OpenGL provides functions along the lines of `createPrimitivePart()` (*not its real name*) that enable it to keep track of the primitive
- The arguments to “`createPrimitivePart()`” specify the object

Specifying entities in OpenGL

- We need to “register” all points, lines, polygons, etc. in OpenGL
- We can register **primitives** in OpenGL as follows

```
glBegin (GL_XXXXX); // register a GL_XXXXX
 createPrimitivePart(data1);
 createPrimitivePart(data2);
    :
 createPrimitivePart(datan);
glEnd();
```

- OpenGL provides functions along the lines of `createPrimitivePart()` (*not its real name*) that enable it to keep track of the primitive
- The arguments to “`createPrimitivePart()`” specify the object

Specifying points in OpenGL

- To make OpenGL aware of some floating point vertices

```
glBegin (GL_POINTS); // register 'GL_POINTS'
 glVertex2f(3.14, -20.5);
   :
 glVertex3f(-5.6, 3.0, 0.6);
glEnd();          // silly mix of 2D & 3D points
```

- Can have our (2D, float) vertices stored in a `vector` (a.k.a. array) and use `glVertex2fv(vect);`
- For **lines** there are three styles

```
glBegin(GL_LINES);
glBegin(GL_LINES_STRIP);
glBegin(GL_LINES_LOOP);
```

Specifying points in OpenGL

- To make OpenGL aware of some floating point vertices

```
glBegin (GL_POINTS); // register 'GL_POINTS'
 glVertex2f(3.14, -20.5);
   :
 glVertex3f(-5.6, 3.0, 0.6);
glEnd();         // silly mix of 2D & 3D points
```

- Can have our (2D, float) vertices stored in a `vector` (a.k.a. array) and use `glVertex2fv(vect);`
- For **lines** there are three styles

```
glBegin(GL_LINES);
glBegin(GL_LINES_STRIP);
glBegin(GL_LINES_LOOP);
```

Specifying points in OpenGL

- To make OpenGL aware of some floating point vertices

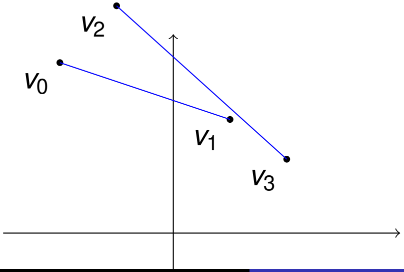
```
glBegin (GL_POINTS); // register 'GL_POINTS'
 glVertex2f(3.14, -20.5);
    :
 glVertex3f(-5.6, 3.0, 0.6);
glEnd();         // silly mix of 2D & 3D points
```

- Can have our (2D, float) vertices stored in a `vector` (a.k.a. array) and use `glVertex2fv(vect);`
- For **lines** there are three styles

```
glBegin(GL_LINES);
glBegin(GL_LINES_STRIP);
glBegin(GL_LINES_LOOP);
```

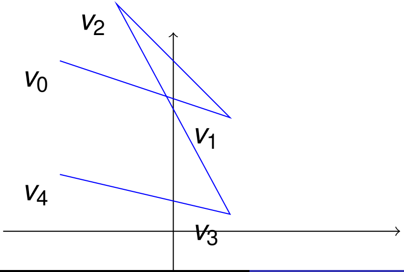
Specifying lines in OpenGL

```
glBegin(GL_LINES);
  glVertex2i(-2, 3); // v0
  glVertex2iv(p1); // v1; p1 is a 2-el. vector {1,2
  glVertex2iv(p2); // v2; likewise p2 {-1,4}
  glVertex2f(2.0, 1.3); // v3
  glVertex2i(-2, 1);  // orphan ==> will be ignored
glEnd();
```



Specifying polylines in OpenGL

```
glBegin(GL_LINES_STRIP);
  glVertex2i(-2, 3); // v0
  glVertex2iv(p1);   // v1 (1,2)
  glVertex2iv(p2);   // v2 (-1,4)
  glVertex2f(1.0, 0.3); // v3
  glVertex2i(-2, 1);  // v4, no longer ignored
glEnd();
```



Specifying closed polylines in OpenGL

```
glBegin(GL_LINES_LOOP);
  glVertex2i(-2, 3); // v0
  glVertex2iv(p1);   // v1 (1,2)
  glVertex2iv(p2);   // v2 (-1,4)
  glVertex2f(1.0, 0.3); // v3
  glVertex2i(-2, 1);  // v4
glEnd();
```