

Computer Science 110 The Art & Science of Computer Graphics Mount Holyoke College Spring 2010

Topic Notes: Animation

Our next goal is to take our programming techniques and expand them, adding some new Mead constructs, to create simple animations. Before we can get to animations, we need to study Bezier curves.

Bezier Curves

We'll begin with an example, then think about exactly what's happening.

See Example:

/home/jteresco/shared/cs110/examples/SimpleBezier

The Prism and Lathe classes take an additional message: (bezier), that instructs the class to treat the profile polygon as a set of control points for one or more *Bezier curve*.

Looking at the images generated by SimpleBezier, we can see that this results in objects with rounded instead of pointed surfaces.

In the example, we use a couple outline polygons to create objects with the Lathe and Prism constructs.

To understand what is going on here and how we can use this to create the curved surfaces we want, we will look in detail at Bezier curves.

We will consider *cubic* Bezier curves, at least to start.

These are defined by 4 *control points*; we'll call them A, B, C, and D. Intuitively, the curve defined by these points is the one traced out by an object "launched" from A in the direction of B and "arriving" at D from the direction of C.

All points on the Bezier curve always lies within the figure ABCD.

More formally and precisely, the points on the Bezier curve specified by points A, B, C, and D are the determined by the formula:

$$A(1-t)^{3} + 3B(1-t)^{2}t + 3C(1-t)t^{2} + Dt^{3}$$

where t ranges from 0 to 1.

If you think a bit about the formula, you'll see that at t = 0, it evaluates to A (as all other terms become 0) and at t = 1, it evaluates to D.

We can use a geometric construction to compute specific points on the Bezier curve. We will build an "A-frame scaffolding" to do this. To compute the point on the curve at, for example at $t = \frac{1}{3}$:

- 1. Draw straight lines from A to B, B to C, and C to D.
- 2. Find the points $\frac{1}{3}$ of the way from A to B (call it E), B to C (call it F), and C to D (call it G).
- 3. Draw straight lines E to F and F to G.
- 4. Find the points $\frac{1}{3}$ of the way from E to F (call it H), F to G (call it I).
- 5. Draw a straight line from H to I.
- 6. Find the point $\frac{1}{3}$ of the way from *H* to *I* (call it *J*). *J* is the point on the Bezier curve for $t = \frac{1}{3}$.

The Wikipedia article for Bezier curves has some excellent diagrams and animations that demonstrate this construction.

There are a number of web sites that you might find useful to specify Bezier curves. Links are on the lecture page.

Looking back at the SimpleBezier example, the polygon outline, when thought of as a Bezier curve, is really two sets of points, defining two cubic Bezier curves:

```
(define outline
(2to3d '((0 0) (25 100) (50 25) (75 0)
(75 0) (50 -25) (25 -100) (0 0))))
```

When a Prism or a Lathe is sent the (bezier) message, it will interpret the profile as sets of 4 Bezier control points.

We can define very complex and interesting curves with this method.

Some tips about stringing together Bezier curves:

- If you want a straight segment from A to D, place B at A and C at D.
- If you want two adjacent Bezier curves (defined by control points A, B, C, D and A', B', C', D') to join together smoothly, you need to make sure that D and A' are the same point, and that C, D/A', and B' are colinear. Otherwise, there will be a sharp corner at D/A'.

See Example:

/home/jteresco/shared/cs110/examples/Bell

The bell uses a straight segment, then two curved Bezier segments, but since we want it to be smooth, the last two points of the second segment and the first two points of the third are colinear.

blend and morph

Mead provides functions that we will use in conjunction with Bezier curves and, soon, to help guide our animations.

First, blend.

In its simplest form, blend does *linear interpolation*. If we provide three parameters:

(blend t from to)

t is expected to be a value from 0-1. If t is 0, the function returns from. If t is 1, the function returns to. If t is some number in between, the function returns a value that is a "blend" of from and to, using t of from and 1-t of to.

The parameters from and to may be numbers or lists of numbers (or lists of lists of numbers, etc.) so long as they are the same "shape" and corresponding elements in the lists can be "blended".

Examples:

```
> (blend .23 0 100)
23.0
> (blend .75 '(0 0) '(-1 -1))
(-0.75 -0.75)
> (blend .5 red blue) ; RGB colors
(0.5 0 0.5)
> (blend .3 '(0 1 1) '(240 1 1)) ; HSV colors
(72.0 1.0 1.0)
```

The morph function does a series of blends. In its simplest form, it takes three parameters as well:

(morph n from to)

In this case, morph returns a list of n+1 blended values that go from from to to.

The first entry of the list is the result of (blend 0 from to) (or simply, from), the next is (blend (/ 1.0 n) from to), the next is (blend (/ 2.0 n) from to), on up to the last, which is (blend 1 from to), which is simply to.

Examples:

```
> (morph 4 0 100)
(0 25.0 50.0 75.0 100)
> (morph 10 '(0 0) '(2 2))
((0 0)
  (0.2 0.2)
  (0.4 0.4)
```

```
(0.8 \ 0.8)
(1.0 1.0)
 (1.20000000000002 \ 1.20000000000000000)
 (1.6 \ 1.6)
(1.8 \ 1.8)
(2 2))
> (morph 5 red blue)
((1 \ 0 \ 0))
(0.8 \ 0 \ 0.2)
(0.6 \ 0 \ 0.4)
(0.399999999999999 0 0.60000000000000)
(0.1999999999999996 \ 0.8)
(0 \ 0 \ 1))
>
```

Remember that passing a value of n results in n+1 elements in our resulting list.

We will make extensive use of these, but to start, let's consider the potential for a "time-lapse" image.

See Example:

/home/jteresco/shared/cs110/examples/MorphReplication

So far, this is just a different way to do something we could have easily done with something like multiAdd.

We can complicate matters a bit and do some things that might be trickier with multiAdd style functions.

The bulk of the work in this example is done in two places:

1. Building our list of positions for our objects with the morph function. In this case:

(morph 19 '(-200 0 0) '(200 0 0))

This gives us 20 positions, the first at $(-200\ 0\ 0)$, the last at $(200\ 0\ 0)$, evenly spaced along the x-axis.

2. Then, we have a function addAtPositions, that works much like our multiAdd and similar functions, but which takes a list of positions where to add our objects.

This takes the place of 3 of the parameters in multiAdd – n is determined by the number of positions in the list, and initialXform and deltaXform are not needed to compute positions since we already have a list of positions.

Most of addAtPositions uses familiar ideas and constructs. A couple of things are new and/or notable:

• we have a new conditional expression – we don't have a number n to compare to 0 to decide when we stop. Instead, we want to stop when we've run out of positions in the list of positions

To see when we've run out of positions, we can have scheme ask if the list is "null" (a computer word for "empty") using the null? function.

null? returns true if the given list has no entries, false otherwise.

- If we're done, we just return the word 'done. We could put just about anything here remember that in our previous examples of this type, we've returned group.
- When adding an object, group our "if false" part inside a let*, where we also define (for convenience), a name pos for the first element in our position list.

We then tell the group to add a copy of the object, translated by an appropriate amount in the x, y, and z directions.

We then add "the rest" by making a recursive call using cdr to get all but the first element (which we've finished with) from our list of positions.

See Example:

/home/jteresco/shared/cs110/examples/MoreMorphReplication

The blend and morph functions can also compute cubic Bezier curve positions. If we provide 5 arguments instead of 3, the values will be interpreted as A, B, C, and D from the Bezier curves.

See Example:

/home/jteresco/shared/cs110/examples/BezierMorphReplication

Finally, these functions can take a set of 3 points, which will produce a quadratic Bezier curve – a parabolic shape.

See Example:

/home/jteresco/shared/cs110/examples/QuadraticMorphReplication

Animation

Armed with Bezier curves and the ideas of the blend and morph functions, it's time to make our first animation.

See Example:

/home/jteresco/shared/cs110/examples/BallToss

There's plenty to digest here. Some key things to notice:

• The new image properties we have not seen before:

- CS 110
 - (fileName "BallToss") instead of generating image and movie files based on your login ID, use this word
 - (frameNumber 0) tell the animation code to start numbering the frames we'll generate at 0
 - (viewResult #f) don't display all images as they're generated (as we usually do), instead display only the generated movie at the end
 - We add the ball to the scene but make sure we have a name for it we'll need this later to adjust its position between frames
 - The function moveBall moves the ball to a given position
 - this will be called when we generate the frames of our movie.
 - note the need for absoluteXform it makes our task easier here to forget all the ways in which the ball was previously transformed, and start over from its default size, position, and orientation - until now, we have always been using a relativeXform (without really being aware of it).
 - The magic happens in the film message to our camera.
 - film generates a sequence of frames
 - here, we generate 51 frames, numbered 0-50
 - for each frame, the function moveBall will be called this is our adjustment function
 - for each call made to moveBall, its parameter will be determined by a morph of the next parameter to film - in this case, a cubic Bezier curve determined by the four points given
 - in general, one parameter will be generated and passed to the adjustment function for each additional parameter to film (we'll see much more interesting examples)
 - With the frames generated by film, we send the message buildMovie to paste these together into an animation!

Our first extension to the example:

See Example:

/home/jteresco/shared/cs110/examples/CrazyBallToss

Instead of just animating the ball, two more objects and several more change parameters have been added.

What happens and how?

• The sun is setting and getting dimmer. The setting is controlled by doing a relative transformation on the sun object every frame (it moves down by 5 every frame) and by setting the Light's intensity according to the second parameter to moveThings. It varies linearly from 1 down to .5.

- There is a ball that moves along the ground and changes color. The motion is in a straight line and the colors are fully saturated ("bright") HSV colors varying from red, through green and blue, and back to red. Notice the function that takes an RGB color and returns a new "plaster" material with that color.
- There is a "crazy" cube that spins, changes size, and bounces. Here, we add three parameters to moveThings, one to control the spin, one to control the scaling, and one to control the bouncing. It spins around three times, so the rotation goes from 0 to 1080. It starts at 50x50x50, expands up to around 100x100x100, back down to about 25x25x25 and back to its original 50x50x50. This one is a cubic Bezier.

To get a little crazier, consider adding camera motion to all of this:

See Example:

/home/jteresco/shared/cs110/examples/CrazyBallTossMoveCamera

We can string together multiple film messages, which will all generate frames that will be put into the same movie by buildMovie.

See Example:

/home/jteresco/shared/cs110/examples/BallTossAround

We next build a wind turbine and start it spinning:

See Example:

/home/jteresco/shared/cs110/examples/WindTurbine

This is a much simpler model – just three cones and a cylinder. Our interest is in the grouping and animation.

- We define a Group called threeBlades of (unsurprisingly) three turbine blades, centered at the origin.
- The threeBlades is added to a Group called turbine along with a support cylinder. The threeBlades are located above the origin, now.
- The animation is controlled by the adjustment function rotateBlade. It does a *relative* transformation on the threeBlades object, rotating about the z-axis some number of degrees.

The questions: Why is the zRot doing the right thing? Shouldn't a zRot of an object that is not located along the z-axis result in the whole collection of blades around the z-axis?

The answer: not in this case! The zRot is being applied to the threeBlades Group *before* it is added to the turbine Group.

The explanation: There are multiple transformations applied, and they are applied successively as objects are built and grouped.

In this case, we build:

- 1. a blade, which is a Cone that has been scaled and translated Mead associates a *transformation* with the definition of blade that represents the composition of the scaling and translation.
- 2. a threeBlades, which is a Group of three blades. Each of these blades, when added to the group, is given a new transformation, specific to that instance, which is composed with the blade transformation from above. In this case, the first added blade has no additional transformation applied (though a "do nothing" transformation the *identity transformation* is stored by Mead), and the second and third each have a rotation about z stored for their transformations. The threeBlades also gets a transformation, which is in this case, the identity transformation since we have not specified and transformations.
- 3. a turbine, which is also a Group, this time consisting of a threeBlades and a cylinder. Each instance added to the group is also given a new transformation – for threeBlades it contains the translation and for the cylinder, it is the composition of the two scalings and the translation.
- 4. Finally, we add the turbine to our scene and a new transformation (the identity, again, since we have not specified any additional transformations) that applies to this instance of the turbine.

So when, in the rotateBlade function, we apply a new transformation to threeBlades, it is to the Group called threeBlades, which in turn applies to the specific instance we added to our turbine (which was then translated).

For this reason, the zRot is applied *before* the translate that was added during the Group construction.

We can see that this particular transformation would apply in this same way to all instances of turbine that we might add to the scene.

See Example:

```
/home/jteresco/shared/cs110/examples/WindTurbines
```

To get a better understanding of these multiple levels of transformations, consider:

See Example:

/home/jteresco/shared/cs110/examples/MovingCubes

Note that an absoluteXform replaces *only* the transformation on the specific instance of the object to which it is applied! This can be hard to keep track of, but can be very helpful when managing animated scenes.

To see that we can use mathematical functions of parameters to our adjustment function as well as the values themselves, consider this simple example:

See Example:

```
/home/jteresco/shared/cs110/examples/SineBall
```