Computer Science 112
Art & Science of Computer Graphics
The College of Saint Rose
Fall 2015

Topic Notes: Animation

At long last, we are just about ready to create our first *animations*.

Our first animation is very simplistic, but we will begin there to see the overall structure of an Ambrosia animation.

On the Wiki: Ballroll

When we run it, if all is set up correctly, a movie file is generated that shows a ball rolling across the image. Ambrosia constructs an animation by generating a series of images, called *frames*. The movie file is a series of these frames, displayed on the screen in succession quickly enough that they appear to the human eye and brain as moving smoothly. This is exactly how movies, television, and computer screens work. As long as each frame is replaced by the next about 24 times per second, it will look to us like smooth motion rather than a series of still images.

There are several items here that are new to us.

- We define a function, in this case, `rollBall`, called our *adjustment function*, that modifies the scene in some way between successive frames of the movie. We will see in a bit how this function gets called repeatedly to generate a series of frames.
- All of these frames need to go somewhere, and we can define a folder (within the folder where we're working) to contain these images:

  ```
  environment.projectFolder('movieImages')
  ```

  In this case, you would need to have a folder `movieImages` inside the folder where your model file exists.
- You probably noticed along the way that the image files Ambrosia generates have names that include your username and some seemingly random number before the `.png` extension. With animations, we usually want to give these frame files another name. This is done with the statement:

  ```
  image.fileName('rollingball')
  ```

- We have developed models that generated multiple images before. You likely recall that this results in each image opening up in a new "Preview" window for each image. When generating an animation, we don't normally want to see each frame as it's generated. We can prevent that from happening with this statement:

```
image.viewResult(False)
```

- A movie is generated from a series of numbered frames, and later we might see a situation where we want to manipulate the frame numbers to generate a sequence of "camera shots" that would later be pasted together into a single movie. For now, we want them all in the same movie, and that means starting our frame numbering at 0:

  ```
  image.frameNumber(0)
  ```

- Rather than a single "`camera.shoot()`", which would generate only a single frame, we use the `film` message to our `camera` object that generates an entire sequence of frames:

  ```
  camera.film(60, rollBall)
  ```

  In this case, we generate 61 frames (the "60" refers to the number of times we adjust the scene between frames, so we always end up with one more frame than the number we give), and the function `rollBall` is called between each pair of frames.

  If we look at the `rollBall` adjustment function, we see that it simply tells the sphere we've named `ball` to move to the right by 10 units.

  Adjustment functions don't get any simpler than this. We will see much more complex adjustment functions that modify multiple objects in our scene by changing their positions, orientations, scale, material properties. We can even change the lighting or camera properties in our adjustment functions.

- Once we have used one or more `film` messages to generate frames of our movie, we need to tell the camera that it should paste them together into a single movie:

  ```
  camera.buildMovie()
  ```

`blend` **and** `morph`

Before we begin work on more interesting animations, we will take a look at a couple of functions provided by Ambrosia that we will use in conjunction first with Bézier curves and then 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, tuples, 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 or tuples can be "blended".

Examples:

```
>>> red
(1, 0, 0)
>>> blend(.23, 0, 100)
23.0
>>> blend(.75, (0,0), (1,1))
(0.75, 0.75)
>>> blend(.5, red, blue)
(0.5, 0.0, 0.5)
>>> blend(.3, (0, 1, 1), (240, 1, 1)) # think: HSV
(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/n, from, to)`, the next is `blend(2/n, from, to)`, on up to the last, which is `blend(1, from, to)`, which is simply `to`.

Examples:

```
>>> morph(4, 0, 100)
[0.0, 25.0, 50.0, 75.0, 100.0]
>>> morph(10, (0,0), (2,2))
[(0.0, 0.0), (0.2, 0.2), (0.4, 0.4), (0.6, 0.6),
 (0.8, 0.8), (1.0, 1.0), (1.2, 1.2), (1.4, 1.4),
 (1.6, 1.6), (1.8, 1.8), (2.0, 2.0)]
>>> morph(5, red, blue)
[(1.0, 0.0, 0.0), (0.8, 0.0, 0.2), (0.6, 0.0, 0.4),
 (0.4, 0.0, 0.6), (0.19999999999999996, 0.0, 0.8), (0.0, 0.0, 1.0)]
```

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.

On the Wiki: Morphreplication

So far, this is just a different way to do something we could have easily done with a `range` of values in our `for` loop.

We can complicate matters a bit and do some things that would be trickier with a loop over a simple range.

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. The `for` loop retrieves the translation amounts in x, y, z from each tuple in our list, and then adds a sphere and translates it by that amount.

Our next example uses `morph` on positions, scaling factors, and colors, then uses values from all of those lists to add some spheres.

On the Wiki: Moremorphreplication

Here, we create three lists from the `morph` function. Then a series of 20 spheres, each of which gets its properties from a given position in each of the three lists generated by `morph`.

We do have one new Python construct: the statements that access the value at a given position in our lists:

```
s = scalingFactors[i]
```

The `blend` and `morph` functions can also compute cubic Bézier curve positions. If we provide 5 arguments instead of 3, the values will be interpreted as control points (our "A", "B", "C", and "D" points from the scaffolding construction) of a cubic Bézier curve.

On the Wiki: Beziermorphreplication

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

On the Wiki: Quadraticmorphreplication

Applying `morph` to Animation

Now that we've seen `morph` and the basics of animation, let's bring the two together.

On the Wiki: Balltoss

There are a few things here that we have not yet seen.

- Our adjustment function `moveBall` again applies to a ball we've added to the scene. In this case, however, we have passed an extra parameter to `camera.film` that specifies a list of 4 points that determine the values that will be passed to the `moveBall` function. Since the list contains 4 points, the points will be those along the cubic Bézier curve defined by those points. Those values (the same as those that would be returned by `morph` with 50 transitions) are passed as the parameter `pos` to each call to the adjustment function.
- Unlike our first animation example, where we always moved the ball by a small amount from its previous position, in this case, we are given the new location where the ball should be at each frame. The usual transformations we apply are *cumulative* (by composition) and thereby *relative* to previously-applied transformations. Ambrosia provides a function that we use here, `absoluteXform`, that causes an *absolute transformation*, where all previous transformations are forgotten, and only the ones given are to be applied. Therefore, the previous position of the ball is forgotten, so we can then move it to the absolution position as given by `pos`.
- When retrieving the x, y, and z positions from the `pos` parameter, we again use the `[]` notation to get the values from our tuple by position.

Bringing together multiple `films`

Our next example shows how to put together multiple "phases" or "segments" of animation into a single movie. Here, we don't just toss the ball from one place to another, but have it then be tossed to a third location, and finally back to the first.

On the Wiki: Balltossaround

Note here the multiple `film` statements, each of which generates 51 frames of the movie. The single `buildMovie` combines the frames from all three segments into a single movie file.

The program has also been improved with a few names defining the parameters of the scene and animation: the three positions and the height of the control point between them. It also uses a quadratic Bézier path (3 control points instead of 4 for cubic) to obtain a truly parabolic path.

Animating multiple things simultaneously

Our adjustment functions and `film` messages can take many parameters that can be used to adjust pretty much anything we wish in our scene from frame to frame of the animation. We have a pretty extreme example of this:

On the Wiki: Crazyballtoss

First, notice that after the name of the adjustment function in the `camera.film`, we have 7 lists. Each of these is going to result in the values of a "`morph` 50" using that list as a linear, quadratic, or cubic Bézier path.

Next, notice that the adjustment function also expects 7 parameters. The first parameter will get the values from the `morph` of the first list in `camera.film`, the second parameter will get the values from the `morph` of the second list, *etc.*.

Specifically, the first call to the adjustment function `moveThings` will receive the parameters from the `morph` at $t = 0$:

```
moveThings((-200,50,0), 1, (-100,25,-100), (0,1,1), 0, .5, 0)
```

The next would be the `morph` at $t = .02$

```
moveThings((-192,59.8,0), 0.99, (-96,25,-96), (7.2,1,1), 21.6, .585, 7.84)
```

The third would be the `morph` at $t = .04$

```
moveThings((-184,69.2,0), 0.98, (-92,25,-92), (14.4,1,1), 43.2, .661, 15.36
```

and so on until the 51st call would result in the `morph` at $t = 1$:

```
moveThings((200,50,0), 0, (100,25,100), (360,1,1), 1080, .5, 0)
```

Those values are assigned to the formal parameters of `moveThings` for us to use inside the adjustment function to move or otherwise modify all of the items.

See the comments, especially in the adjustment function `moveThings` to see how these are applied to change the positions, orientations, materials properties, sizes, and intensities of objects in our scene.

Moving the Camera

While the above example makes many changes on each adjustment between frames, the camera remains in a fixed position. But that's also movable.

On the Wiki: Cameraviewaround

In this case, we put lots of objects in the scene, but never move them. Instead, we move the camera to positions that trace out a circle 300 units above the xz-plane, centered around the y-axis. Don't be too scared by the sine and cosine in there, it's just a convenient way to compute coordinates of the points along that circle.

We could also use interpolation directly to compute camera positions. Here's the same scene, but with a "fly-through" instead.

On the Wiki: Cameraviewfly

For *Star Wars* fans, here's a scene inspired by the attack on the Death Star.

On the Wiki: Canyonfly

Advanced Animation

The following examples show a "sliding" ball *vs.* a rolling ball:

On the Wiki: Uglyballroll

On the Wiki: Niceballroll