

# Lecture on Animation

## 1 Animation Basics

Animation of simple motion is fairly straightforward. Instead of producing one frame, we produce several. The way this is usually done is via an *idle callback*. This is similar to other callbacks: we register it and the window system calls it. This callback, however, is called when the graphics system has nothing better to do — it's *idle*. Essentially, this means that when it's done rendering your scene, it calls this callback.

If the idle callback does the following:

- adjusts some global variables, and
- `glutPostRedisplay()`

The system will continually adjust the variables and redisplay your scene.

What this means is that you need to code your `display` function so that the computation depends on some global variables that you will adjust in your idle callback.

Examples:

- `~cs307/public_html/demos/animation/MovingMobile.cc` This program has an array of part angles and an array of speeds that is used to increment the part angles. The `display` function uses the array of part angles when drawing the mobile.
- spinning around an axis. This is what TW programs do when you hit “x” or “y” or “z”: spin the entire scene around the given axis. We'll look at the TW code to see how this is done. This is in `~cs307/public_html/tw/tw-camera.cc` Look at the functions `twSpin`, `startSpinning`, and `twSpinCommand`.

But you didn't think it was that easy, did you? It's not. The hard part is always in modeling the physics of the situation. We'll look at some hard cases later.

## 2 Double Buffering

To animate smoothly, we have to use double-buffering. All of our example programs have used double-buffering, but now we'll learn about why they do, and the effects of not using double-buffering.

To understand the concept of double-buffering, let's first look at the problem it solves.

Demo: `~cs307/public_html/demos/animation/SingleDouble.cc`

This program opens two windows, both of which have an idle callback that rotates a square. One looks like a fine animation, the other has a terrible flicker. What causes the flickering?

The graphics system is constantly erasing and redrawing the scene. The monitor is constantly refreshing the screen. (Most modern monitors refresh between 50–100 times per second, so every 10 to 20 milliseconds.) If the screen is refreshed when the new image is only partly drawn (this includes filling areas in the framebuffer), you'll see, briefly, that partial image. That's what causes the flicker.

The solution is to somehow “synchronize” the two so that the monitor never draws an incomplete image. The way this is done is:

- The monitor reads out from the “front” buffer, while
- the graphics system draws into the “back” buffer, and when it's done,
- they swap

The names “front” and “back” buffer are conventional: the front buffer is the one that is “on stage” and the back buffer is the one that is being prepared for the next scene.

All throughout this course, our programs have done:

- use `glInit(GL_DOUBLE...)`

- use `glutSwapBuffers()`

Now you know why. The first tells OpenGL that you want to use double-buffering, so it sets up two buffers and it automatically draws in the “back” buffer. The second says that the program is done drawing in the back buffer and to swap it with the front buffer. The combination means that when we do animations or even just move the viewpoint with the mouse, we don’t get any flicker.

**Note:** if you are using double buffering and you forget to do `glutSwapBuffers`, your screen will be blank! Why? Because you are drawing in the back buffer and there is nothing in the front buffer.

In my view, the *only* reason to ever use single-buffering is when you know you’re only drawing a static scene and you’re short on memory on the graphics card. Pretty rare nowadays.

Note that this double-buffering idea is a general notion that is also used in database I/O and lots of other areas of CS.

### 3 MPEGs

Once we have an animation, we’d like to capture it for posterity. We can do that using MPEG files.

The file types discussed before are for single images (except for GIF, which can also do animations). Another file format is called MPEG, devised by the Motion Picture Experts Group. It is a multi-frame format with fancy cross-frame compression (since most frames are pretty similar). It actually comes in versions. The MP3 format that we all know and love is the audio track of MPEG version 3.

There is a program called `ppmtompeg` that will convert a collection of PPM files to an MPEG movie. It requires a parameter file, but I found a useful shell script that can create that parameter file and run the program for us. The shell script is called `make-mpeg`. It lives in `~cs307/public_html/demos/animation/`.

The script assumes that the frames are in files named in a pattern like “baseNNN.ppm”: that means that there is some base name, three frame number digits, starting with 001, and the “.ppm” extension. So, we will make sure that happens.

Returning to the `MovingMobile` demo, try the following callbacks:

- ‘3’ This does just a single frame of the animation. Being able to go one frame at a time is a tremendous help to debugging, so I recommend it.
- ‘2’ Start spinning. This starts and stops the animation, but setting up the idle callback.
- ‘1’ toggle saving of frames. This writes each frame of the animation to a file in the current directory, as long as the animation proceeds. Each time to start saving frames, it starts over again, overwriting previous animations. When you use this, make sure you are running the `MovingMobile` demo in a directory you own (or `/tmp`)! For example:

```
cd /tmp
~cs307/public_html/demos/animation/MovingMobile
```

If you turn on saving frames and turn on the animation, you’ll get a collection of PPM files in the current directory, all named `MovingMobileXYZ.ppm`, where `XYZ` is the frame number. Once you have this collection of saved frames, you can turn them into an MPEG animation by

Then:

```
% ~cs307/pub/demos/animation/make-mpeg MovingMobile
```

Check out the result on our web site!

Note that PPM files are pretty big. You should do your work in `/tmp`, so that the temporary files won’t count against your quota, then delete the PPM files and move the MPEG file back to your regular account.

It takes a little while to make an MPEG. It took 6 minutes to make the 360 frame MPEG of the moving mobile.

## 4 Animation Techniques

We can break animation techniques down into two broad categories, roughly:

- derivative: how does the scene change?
- positional: where are things supposed to be?

(For you calculus types, you can see that the first technique gets its name because it is the derivative of the position function.)

### 4.1 Derivative Techniques

All the techniques we've seen so far use the derivative technique. We simply update the positions of objects using information about their motion. For example, we update the current angle of a bar of the mobile by incrementing an array element, and then draw the scene based on all the current angles. Essentially, all our idle callback computations are based on something like:

```
void idleCallback() {
    position += velocity;
    glutPostRedisplay();
}
```

We called the variable on the right `velocity` because by definition velocity is the change in position. If the velocity is large, there will be a big change in position; if the velocity is small, there will be a small change. Note also that velocity can be either positive or negative, so position can increase or decrease.

You'll notice that time does not appear in the equation. Essentially, each frame of our animation is one time step. In some sense, we're doing the following computation:

```
void idleCallback() {
    float DELTA_T = 1;
    position += velocity*DELTA_T;
    position += velocity;
    glutPostRedisplay();
}
```

where `DELTA_T` is our time step, and it has the value 1 by the way we are building the animation. If your model is based on real-world objects with real-world positions and speeds (say, meters and meters per second), you have to understand that each frame of the animation is one time unit (say, one second).

The time unit and speeds are also crucial for determining the *smoothness* of your animation. If your object jumps by a whole bunch from one frame to the next, the animation may look jerky. To fix that, you'd need to reduce your `deltaT`, say from one second to half a second, or even a tenth of a second. Thus, your computations become:

```
void idleCallback() {
    float DELTA_T = 0.1;
    position += velocity*DELTA_T;
    glutPostRedisplay();
}
```

You can see the effects of this in the `MovingMobile` demo. In fact, you can specify the size of the time step on the command line. Try it! Try some low values (should be very slow and smooth) and some high values (should be quick, but jerky).

For a slightly more complex motion, still based on the derivative approach, consider the `MassSpring` demo. This is a classic example from Physics, where the mass moves due to a force exerted on it by the spring. The spring exerts a force that depends on the amount that the spring is stretched and hence on the position of the mass. The force yields an acceleration that depends on the mass (more mass, less acceleration). Acceleration, of course, results in a changing velocity, and velocity results in changing position. Thus, at each time step, the idle callback computes:

```

void idleCallback() {
    massA = - springK / mass * massX;
    massV += massA * DT;
    massX += massV * DT;
    glutPostRedisplay();
}

```

The nature of this particular model is such that the velocity is sometimes negative and sometimes positive, so the mass moves back and forth, oscillating endlessly.

There are variant mass-spring models with damping (friction) that slows the mass down based on its velocity. A google search for “mass-spring” yields many results, so feel free to investigate if you like.

## 4.2 Positional Techniques

A limitation of the derivative approach is that, because time is absent from the computation, you can’t have things start and stop, or change direction. The derivative approach works very well for continuous, unchanging models like the mobile and the mass-spring, but not so well for, say, cars that start, speed up, turn, slow down, and stop. To do that, we need to explicitly introduce time as a variable.

In general, what we’d love to have is a position function that tells us where the object is at a particular time. If so, our idle callback could be as simple as:

```

void idleCallback() {
    Time += DeltaT;
    glutPostRedisplay();
}

```

Our display callback would then use the Time variable as an argument to the position function to compute where everything is supposed to be right now:

```

void display() {
    ...
    x = position(Time);
    glTranslatef(x,0,0);
    drawObject();
    ...
}

```

For example, suppose we had an object that we wanted to move smoothly from point A to point B. Using the ideas of parametric equations, and using the Time variable as the parameter, we would come up with something like this:

```

void display() {
    ...
    twTriple A = { ... };
    twTriple B = { ... };
    twTriple dir; // direction of motion
    twTriple currPos; // current position
    twVector(dir,B,A); // computes dir = B-A
    twPointOnLine(currPos,A,dir,Time); // computes currPos = A + dir*Time
    glTranslatef(currPos[0],currPos[1],currPos[2]);
    drawObject();
    ...
}

```

This idea is captured in the Laser demo, in which a UFO drifts across the scene and fires laser bolts (like photon torpedoes) downwards. The laser bolts are drawn with up to five frames, unless the laser bolt hits something, in which case successively larger spheres are drawn, to represent the explosion. Try it! Look at the code.

What if we want to have the object, such as a car, be motionless for a while, then start moving from A to B, then stop, then do something else, and so on? For that, we need to start thinking about particular values of the Time variable. If Time starts at 0 and increments with each frame, that might mean we want to have the car start at time 15, move from A to B during time units 15 to 25, then stop. Our code would look something like:

```
void display() {
    ...
    if( Time >= 15 && Time <= 25 ) {
        param = (Time - 15)/(25-15);
        twTriple A = { ... };
        twTriple B = { ... };
        twTriple dir;           // direction of motion
        twTriple currPos;      // current position
        twVector(dir,B,A);     // computes dir = B-A
        twPointOnLine(currPos,A,dir,param); // computes currPos = A + dir*param
        glTranslatef(currPos[0],currPos[1],currPos[2]);
        drawObject();
        ...
    }
}
```

Notice the computation of `param`. Remember that as the parameter for our line goes from 0 to 1, the object moves from A to B. So, we have to map the Time units 15 to 25 onto the time interval [0,1]. This is simply another example of translation and scaling.

You'll notice that in the example above, the object isn't drawn except when the time is between 15 and 25. To take care of that problem just requires a bit more coding.

### 4.3 Solid Objects

One problem is that objects can pass right through each other: we've always been able to draw overlapping objects in OpenGL. In order to handle this, your program has to detect collisions (when two objects intersect) and decide what to do (does the moving one stop, bounce off, and if so where?). Computing intersections isn't easy. Imagine computing whether two teapots intersect!

One approximation that can be helpful is to use bounding boxes and bounding spheres. Consider bounding spheres first. If you imagine that each of your objects exists inside a bubble of a particular radius, you can compute the distance between each pair of bubbles, using just the pythagorean theorem. If the distance between the bubbles is greater than the combined radii of the bubbles, the two objects *can't* intersect. You can then go on to consider another pair of objects. If the distance isn't greater than the combined radii, the objects *may* intersect, and you can, if you want, try to do additional geometric tests to determine if they do. In some cases, it might be sufficient to simply use the bounding bubble. Using bounding boxes is similar, although the geometry isn't quite as easy. For example, if the minimum  $x$  of one object is greater than the maximum  $x$  of the other, they cannot intersect. Considering the other two dimensions gives you a rough idea of whether they can intersect. Thus, bounding boxes gives you a quick-and-dirty way to eliminate certain pairs of objects from more exacting geometry tests.