Dragonfly

Program a Game Engine from Scratch

Mark Claypool

Development Checkpoint #9

Sprite Animation

This document is part of the book "Dragonfly – Program a Game Engine from Scratch",
(Version 9.0). Information online at: `http://dragonfly.wpi.edu/book/`

4.12.5 Using Sprites and the Animation Class

At this point, the game programmer can load sprites into the ResourceManager in a few simple steps. The first step is to create a sprite file, such as the one in Listing 3.3 on page 18. The second is to load the sprite into the ResourceManager so the game can make use of it. Example code to load the saucer sprite for Saucer Shoot (Section 3.3) is shown in Listing 3.2 on page 17.

To actually use Sprites, say to draw them in an animated fashion on the window, Dragonfly needs to be extended in a couple of ways. A Sprite holds the "static" properties of an animation in that they are fixed for all Objects that use the sprite. To actually animate the Sprite, an Animation class is created to provide control of the Sprite animation for each associated Object.

Animation is shown in Listing 4.141. The class needs `Sprite.h` as well as `<string>`. The attribute `m_p_sprite` indicates what Sprite is associated with the Animation and `m_name` the corresponding name. The attribute `m_index` keeps track of which frame is currently being drawn. The attribute `m_slowdown_count` is a counter used in conjunction with the Sprite slowdown rate (see Section 4.12.2 on page 154) to provide animation through cycling the frames. Methods to get and set each attribute are also provided. The `setSprite()` methods also sets the bounding box for the Object (described in the upcoming Section 4.13.2).

Listing 4.141: Animation.h

```cpp
// System includes.
#include <string>

// Engine includes.
#include "Sprite.h"

class Animation {

 private:
  Sprite *m_p_sprite;      // Sprite associated with Animation.
  std::string m_name;      // Sprite name in ResourceManager.
  int m_index;             // Current index frame for Sprite.
  int m_slowdown_count;    // Slowdown counter.

 public:
  // Animation constructor
  Animation();

  // Set associated Sprite to new one.
  // Note, Sprite is managed by ResourceManager.
  // Set Sprite index to 0 (first frame).
  void setSprite(Sprite *p_new_sprite);

  // Return pointer to associated Sprite.
  Sprite *getSprite() const;

  // Set Sprite name (in ResourceManager).
  void setName(std::string new_name);

  // Get Sprite name (in ResourceManager).
  std::string getName() const;
```

```
31
32    // Set index of current Sprite frame to be displayed.
33    void setIndex(int new_index);
34
35    // Get index of current Sprite frame to be displayed.
36    int getIndex() const;
37
38    // Set animation slowdown count (-1 means stop animation).
39    void setSlowdownCount(int new_slowdown_count);
40
41    // Set animation slowdown count (-1 means stop animation).
42    int getSlowdownCount() const;
43
44    // Draw single frame centered at position (x,y).
45    // Drawing accounts for slowdown, and advances Sprite frame.
46    // Return 0 if ok, else -1.
47    int draw(Vector position);
48 };
```

The Animation `draw()` method, shown in Listing 4.142, basically makes a call to Sprite `draw()` then advances the sprite index to the next frame. Line 12 asks the Sprite to draw the current frame at the indicated position. The block of code at line 15 checks if the sprite slowdown count is set to -1 – if so, this indicates the animation is frozen, not to be advanced, so the method is done. Otherwise, the slowdown counter is advanced ,and on line 24 checked against the slowdown value to see if it is time to advance the sprite frame. Advancing increments the index, with the code starting at line 31 taking care of looping from the end of the animation sequence to the beginning. The last two actions at the end of the method set the slowdown counter and the sprite indices to their values for the next call to `draw()`.

Listing 4.142: Animation draw()

```
0  // Draw single frame centered at position (x,y).
1  // Drawing accounts for slowdown, and advances Sprite frame.
2  // Return 0 if ok, else -1.
3  int Animation::draw()
4
5    // If sprite not defined, don't continue further.
6    if m_p_sprite is NULL then
7      return
8    end if
9
10   // Ask Sprite to draw current frame.
11   index = getIndex()
12   Sprite draw(index, pos)
13
14   // If slowdown count is -1, then animation is frozen.
15   if getSlowdownCount() is -1 then
16     return
17   end if
18
19   // Increment counter.
20   count = getSlowdownCount()
21   increment count
```

```
22
23    // Advance sprite index, if appropriate.
24    if count >= getSlowdown() then
25
26      count = 0   // Reset counter.
27
28      increment index   // Advance frame.
29
30      // If at last frame, loop to beginning.
31      if index >= p_sprite -> getFrameCount() then
32        index = 0
33      end if
34
35      // Set index for next draw().
36      setIndex(index)
37
38    end if
39
40    // Set counter for next draw().
41    setSlowdownCount(count)
```

With Frame, Sprite and Animation defined, Object can be extended to support sprite animations. The Object is provided with an Animation object (m_animation) with corresponding setAnimation() and getAnimation() methods, and a method to set the associated sprite (setSprite()). Up until now, game objects needed to define their own draw() methods to display something on the window. But with a Sprite now associated with an Object, the draw() method can now be defined to draw the animated sprite.

Listing 4.143: Object class extensions to support Sprites

```
0  private:
1    Animation m_animation;   // Animation associated with Object.
2
3  public:
4
5    // Set Sprite for this Object to animate.
6    // Return 0 if ok, else −1.
7    int setSprite(std::string sprite_label);
8
9    // Set Animation for this Object to new one.
10   // Set bounding box to size of associated Sprite.
11   void setAnimation(Animation new_animation);
12
13   // Get Animation for this Object.
14   Animation getAnimation() const;
15
16   // Draw Object Animation.
17   // Return 0 if ok, else −1.
18   virtual int draw();
```

The revised Object draw() method shown in Listing 4.144 method simply calls the Animation draw() method, passing in the Object position.

Listing 4.144: Object draw()

```
0  // Draw Object Animation.
1  // Return 0 if ok, else −1.
2  int Object::draw()
3    pos = getPosition()
4    return  m_animation.draw(pos)
```

Note, `draw()` is still defined as `virtual`. This allows a derived class (a game object) to define its own `draw()` method, should it so choose. In such a case, the game object's `draw()` would get called. The game programmer could write code for object-specific functionality (say, displaying a health bar above an avatar), and still call the built-in Object `draw()` explicitly, via `Object::draw()`.

The `setSprite()` method is shown in Listing 4.145. The first block of code retrieves the Sprite by name (`sprite_label`) from the Resource Manager, checking that the Sprite can be found. Then, the Sprite is associated with the `m_animation` object.

Listing 4.145: Object setSprite()

```
0  // Set Sprite for this Object to animate.
1  // Return 0 if ok, else −1.
2  int Object::setSprite(std::string sprite_label)
3
4    p_sprite = RM.getSprite(sprite_label)
5    if p_sprite == NULL then
6      return error
7    end if
8
9    m_animation.setSprite(p_sprite)
10
11   // All is well.
12   return ok
```

4.12.6   Development Checkpoint #9!

Continue Dragonfly development, getting the engine to support Sprites. Steps:

1. Create an Animation class, following Listing 4.141 and stubbing out the methods. Make sure that it compiles, first. Then, implement the methods to get and set the simple attributes

2. Next, implement Animation `draw()` as per Listing 4.142 testing it carefully.

3. Extend the Object class to support Sprites, as per Listing 4.143.

4. Write the code for the revised Object `draw()` in Listing 4.144 that uses Animation to draw. Write code for a game object (inherited from Object) that associates with a Sprite. Integrate this game object into a game and test the functionality of the Object `draw()`. Debugging can be visual (what is seen on the screen), but use logfile messages to help determine when/where there are problems.

5. Test a variety of game objects with a variety of Sprites (from the Saucer Shoot tutorial or created by hand). Verify the Sprites can be advanced, slowed down and stopped and are drawn without visual glitches. Test and debug thoroughly before proceeding.