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 11.0). 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 4.143. 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 156) 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), as well as resets/initializes the `m_slowdown_count` and `m_index`.

Listing 4.143: Animation.h

```
0  // System includes.
1  #include <string>
2
3  // Engine includes.
4  #include "Sprite.h"
5
6  class Animation {
7
8   private:
9    Sprite *m_p_sprite;     // Sprite associated with Animation.
10   std::string m_name;     // Sprite name in ResourceManager.
11   int m_index;            // Current index frame for Sprite.
12   int m_slowdown_count;   // Slowdown counter.
13
14  public:
15   // Animation constructor
16   Animation();
17
18   // Set associated Sprite to new one.
19   // Note, Sprite is managed by ResourceManager.
20   // Set Sprite index to 0 (first frame).
21   void setSprite(Sprite *p_new_sprite);
22
23   // Return pointer to associated Sprite.
24   Sprite *getSprite() const;
25
26   // Set Sprite name (in ResourceManager).
27   void setName(std::string new_name);
28
```

```
29    // Get Sprite name (in ResourceManager).
30    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 4.144, 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 4.144: 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(Vector position)
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)
```

4.12.5.1 Transforming Sprites (optional)

Sometimes, a game programmer may want to change the drawing orientation of the original sprite, flipping it horizontally or vertically (or both) along the center axis. For example, consider a Hero sprite that is drawn facing right as it moves through the world left to right. If the player turns the Hero around, moving right to left, the sprite should now face left. The game developer could make two sprites – one facing left and one facing right – but the engine could also be extended with a *transform* property to allow the same sprite to be flipped.

In Dragonfly, this transform is not a property of the Sprite (which holds the base frames), but is rather a property of each game object so as not to change the sprite orientation for all game objects at once (i.e., one instance of a game object may want its sprite to be flipped vertically, while another may not). To support transformation, the Animation class is extended with an additional transform property – one of none, horizontal, vertical or both. Then, when the Animation tells the Sprite to draw itself, it passes in the transform. The Sprite, in turn passes in the transform to the Frame when telling the Frame to draw itself. Finally, the Frame re-arranges the characters to represent the transform (if any) when drawing.

First, the transform property is added to `Frame.h`, shown in Listing 4.145 line 1, and the Frame `draw()` method is also extended on line 17 to take in the transform as a parameter (with a default of `NONE` if not provided).

Listing 4.145: Frame class extensions to support transformation

```
0 // Options to transform frame before drawing.
1 enum Transform {
2   NONE,         // No frame transform (default).
3   VERTICAL,     // Frame flipped vertically.
```

```
4   HORIZONTAL,   // Frame flipped horizontally.
5   BOTH,         // Frame flipped both vertically and horizontally.
6 };
7
8 class Frame {
9
10   ...
11
12   // Draw self, centered at position (x,y) with color.
13   // Don't draw transparent characters (0 means none).
14   // Return 0 if ok, else -1.
15   // Note: top-left coordinate is (0,0).
16   int draw(Vector position, Color color, char transparency,
17     Transform transform = NONE) const;
18
19 };
```

To do the actual transform when drawing, the Frame `draw()` code is extended as in Listing 4.146. The `switch` statement starting on line 8 sets up starting horizontal and vertical values. The drawing loop, lines 26-35, figures out what character to draw on line 26 and lastly moves `h` and `v` along, based on the transform values set in the `switch`.

Listing 4.146: Frame draw() extensions to support transformation

```
0 // Draw self centered at position (x,y) with color.
1 // Don't draw transparent characters (0 means none).
2 // Return 0 if ok, else -1.
3 // Note: top-left coordinate is (0,0).
4 int Frame::draw(Vector position, Color color, char transparency, Transform
    transform) const {
5
6   ...
7   // Get xy start, end and incr based on transform.
8   switch (transform)
9   case NONE
10    h_start = 0, h_incr = 1
11    v_start = 0, v_incr = 1
12  case HORIZONTAL:
13    h_start = m_width-1, h_incr = -1
14    v_start = 0, v_incr = 1
15  case VERTICAL:
16    h_start = 0, h_incr = 1
17    v_start = m_height-1, v_incr = -1
18  case BOTH:
19    h_start = m_width-1, h_incr = -1
20    v_start = m_height-1, v_incr = -1
21  end switch
22
23  // Draw character by character.
24  v = v_start
25  h = h_start
26  for y = 0 to m_height-1
27    for x = 0 to m_width-1
28      ...
29      ch = m_frame_str[v*m_width + h]
30      DM.drawCh(temp_pos, ch, color)
```

```
      ...
      h += h_inc
    end for // x
    v += v_inc
  end for // y
```

To use this transformation feature, the Sprite `draw()` method takes the transform as a parameter and passes that to the Frame `draw()`. The Animation class is extended with another attribute, `m_transform` (shown below), which is passed into the Sprite `draw()` method when called.

```
...
class Animation {

 private:
  ...
  Transform m_transform; // Transform sprite before drawing.
```

The Animation constructor should initialize `m_transform` to `NONE`, and provide `getTransform()` and `setTransform()` methods.

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 4.147: Object class extensions to support Sprites

```
private:
  Animation m_animation;  // Animation associated with Object.

public:

  // Set Sprite for this Object to animate.
  // Return 0 if ok, else -1.
  int setSprite(std::string sprite_label);

  // Set Animation for this Object to new one.
  // Set bounding box to size of associated Sprite.
  void setAnimation(Animation new_animation);

  // Get Animation for this Object.
  Animation getAnimation() const;

  // Draw Object Animation.
  // Return 0 if ok, else -1.
  virtual int draw();
```

The revised Object `draw()` method shown in Listing 4.148 method simply calls the Animation `draw()` method, passing in the Object position.

Listing 4.148: Object draw()

```
// Draw Object Animation.
// Return 0 if ok, else -1.
int Object::draw()
  pos = getPosition()
  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 4.149. 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 4.149: Object setSprite()

```
// Set Sprite for this Object to animate.
// Return 0 if ok, else -1.
int Object::setSprite(std::string sprite_label)

  p_sprite = RM.getSprite(sprite_label)
  if p_sprite == NULL then
    return error
  end if

  m_animation.setSprite(p_sprite)

  // All is well.
  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 4.143 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 4.144 testing it carefully.
3. Extend the Object class to support Sprites, as per Listing 4.147.
4. Write the code for the revised Object `draw()` in Listing 4.148 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.