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 5.0). Information online at: `http://dragonfly.wpi.edu/book/`

4.12.5 Using Sprites

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 17. 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 16.

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. Up to now, the DisplayManager can only draw a single character on the window. While characters are the basic drawing unit for Dragonfly, many Objects have a frame of animation as a basic drawing unit. So, for convenience, the DisplayManager is extended to be able to draw a frame. The pseudo code for `drawFrame()` is shown in Listing 4.125 and a public method prototype is added to `DisplayManager.h`.

Listing 4.125: DisplayManager drawFrame()

```
0  // Draw single sprite frame at window location (x,y) with color.
1  // If centered true then center frame at (x,y).
2  // Return 0 if ok, else -1.
3  int DisplayManager::drawFrame(Vector world_pos, Frame frame,
4                                bool centered, Color color) const;
5
6    // Error check empty string.
7    if frame is empty then
8      return error
9    end if
10
11   // Centered? Then offset (x,y) by 1/2 (width,height).
12   if centered then
13     x_offset = frame.getWidth() / 2
14     y_offset = frame.getHeight() / 2
15   else
16     x_offset = 0
17     y_offset = 0
18   end if
19
20   // Frame data stored in string.
21   std::string str = frame.getString()
22
23   // Draw row by row, character by character.
24   for y = 0 to frame.getHeight()-1
25     for x = 0 to frame.getWidth()-1
26       Vector temp_pos(world_pos.getX() - x_offset + x,
27                       world_pos.getY() - y_offset + y)
28       drawCh(temp_pos, str[y * frame.getWidth() + x], color)
29     end for // x
30   end for // y
```

The first block of code starting on line 6 checks if the Frame is empty to avoid subsequent parsing errors. This can be checked with the `empty()` method call on the Frame string and, if true, an error (-1) is returned.

Next, a Sprite needs to be associated with a game Object. This means extending the

Object class to have additional private attributes to handle Sprites, shown in Listing 4.126. The attribute `p_sprite` indicates what Sprite is associated with the Object. Most Sprites are drawn centered on the Object position, but this is just an option specified by `sprite_center`. In the normal course of animation, drawing proceeds sequentially through all the frames in a sprite until the end, then loops. The attribute `sprite_index` keeps track of which frame is currently being drawn. By default, a sprite frame is advanced sequentially each game loop. However, for many animations, this will be too fast. In order to slow down the animation, the attribute `sprite_slowdown` provides a slowdown rate. For example, a slowdown of 5 would mean the animation is only advanced by 1 frame for every 5 steps of the game loop. A slowdown of 1 means no slowdown, and a slowdown of 0 has a special meaning, to stop the animation altogether. The attribute `sprite_slowdown_count` is a counter used in conjunction with the slowdown rate 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 4.126: Object class extensions to support Sprites

```
0  private:
1    Sprite *p_sprite;     // Sprite associated with object.
2    bool sprite_center;   // True if sprite centered on object.
3    int sprite_index;     // Current index frame for sprite.
4    int sprite_slowdown; // Slowdown rate (1 = no slowdown, 0 = stop).
5    int sprite_slowdown_count; // Slowdown counter.
6
7  public:
8    // Set Object Sprite to new one.
9    // If set_box is true, set bounding box to size of Sprite.
10   // Set sprite index to 0 (first frame).
11   void setSprite(Sprite *p_new_sprite, bool set_box=true);
12
13   // Return pointer to Sprite associated with this Object.
14   Sprite *getSprite() const;
15
16   // Set Sprite to be centered at Object position.
17   void setCentered(bool centered=true);
18
19   // Indicates if sprite is centered at Object position.
20   bool isCentered() const;
21
22   // Set index of current Sprite frame to be displayed.
23   void setSpriteIndex(int new_sprite_index);
24
25   // Return index of current Sprite frame to be displayed.
26   int getSpriteIndex() const;
27
28   // Slows down sprite animations.
29   // Sprite slowdown is in multiples of GameManager frame time.
30   void setSpriteSlowdown(int new_sprite_slowdown);
31   int getSpriteSlowdown() const;
32   void setSpriteSlowdownCount(int new_sprite_slowdown_count);
33   int getSpriteSlowdownCount() const;
34
35   // Draw single sprite frame.
```

```
36   // Drawing accounts for center & slowdown, and advances Sprite frame.
37   virtual void draw();
```

In addition, while the `draw()` method has previously been defined in Listing 4.75 on page 112, it is currently empty. 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. Basically, this involves a call to DisplayManager `drawFrame()`, then advancing the sprite index to the next frame. If done strictly this way, the rate of change on the animation will be one frame every game loop (so, 30 updates per second). Many 2d animations are not designed to progress that fast – basically, they want a lower frame rate. So, the `draw()` method allows the game programmer to control the rate the sprite animation progresses through its frames.

Object `draw()` is shown in Listing 4.127. Line 12 asks the DisplayManager to draw the current frame at the indicate position and in the indicated sprite color.

The block of code at line 16 checks if the sprite slowdown value is set to 0 – 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 25 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 32 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 4.127: Object draw()

```
0  // Draw single sprite frame.
1  // Drawing accounts for: centering, slowdown, advancing Sprite Frame.
2  virtual void Object::draw()
3
4    // If sprite not defined, don't continue further.
5    if p_sprite is NULL then
6      return
7    end if
8
9    index = getSpriteIndex()
10
11   // Ask graphics manager to draw current frame.
12   DisplayManager drawFrame(pos, p_sprite->getframe(index),
13                            sprite_center, p_sprite->getColor())
14
15   // If slowdown is 0, then animation is frozen.
16   if getSpriteSlowdown() is 0 then
17     return
18   end if
19
20   // Increment counter.
21   count = getSpriteSlowdownCount()
22   increment count
23
24   // Advance sprite index, if appropriate.
25   if count >= getSpriteSlowdown() then
```

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

Note, `draw()` is still defined as `virtual`. This allows a derived class (a game object) to still define its own `draw()` method. 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()`).

4.12.5.1 Transparency (optional)

In some cases, the characters making up a sprite do not occupy the full extent of their box. For example, a stick figure will have a bounding box around the whole figure, but there will be empty regions around the head, under the arms, etc. By default, Dragonfly will draw such blank spaces, occluding whatever characters may have been drawn below it (e.g., the background), when it may look better to not draw the blanks. For images, providing this functionality is typically done by declaring one color to be "transparent" where that color, wherever found in the image, is not rendered on the window, allowing any underlying image to be seen instead. For Dragonfly, transparency is done in a similar fashion, with the option of a character being specified as the transparency character – whenever this character is part of the sprite frames it is not rendered, thus not occluding any underlying characters.

In order to support drawing with transparency, the DisplayManager `drawFrame()` method must be refactored as shown in Listing 4.128 (refer to Listing 4.125 on page 156 for the original method). The refactored `drawFrame()` method takes an additional parameter indicating the transparent character, with a default value of 0 (*not* the character '`0`') meaning no transparency. Inside the loops iterating over the frame, before a character is drawn (via `drawCh()`), it is verified that either the transparency is not 0 or the character is not the transparent character.

Listing 4.128: DisplayManager extension to support transparency

```
0 // Draw single sprite frame at window location (x,y) with color.
1 // If centered true, then center frame at (x,y).
2 // Don't draw transparent characters (0 means none).
3 // Return 0 if ok, else -1.
4 int DisplayManager::drawFrame(Vector world_pos, Frame frame,
```

```
                                bool centered, Color color,
                                char transparent) const;

  // Draw character by character.
  for y = 0 to frame.getHeight()-1
    for x = 0 to frame.getWidth()-1
      if (transparent not defined) or
         (str[y*frame.getWidth() + x] != transparent) then
        // drawCh normally
        ...
      end if
      ...
```

The transparency character itself is an attribute of an Object. Extensions needed to the Object class are shown in Listing 4.129. Transparency is stored in a `char` attribute, `sprite_transparency` (set to 0 in the Object constructor), with methods to get and set it.

Listing 4.129: Object class extensions to support transparency

```
private:
 char sprite_transparency;   // Sprite transparent character (0 if none).

public:
 // Set Sprite transparency character (0 means none).
 void setTransparency(char transparent=' ');

 // Get Sprite transparency character (0 means none).
 char getTransparency() const;
```

Lastly, the Object `draw()` method needs to be changed to correspond to the refactored DisplayManager `drawFrame()`, providing `getTransparency()` as a parameter.

4.12.6 Development Checkpoint #9!

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

1. Extend the DisplayManager by coding `drawFrame()`, referring to Listing 4.125. Test outside of an actual Sprite by writing a game object (inherited from Object) `draw()` method that requests drawing of a variety of sizes and contents for frames.

2. Extend the Object class to support Sprites, adding attributes from Listing 4.126 and stubbing out the methods. Make sure that it compiles, first, then implement the methods to get and set the simple attributes.

3. Write the code for the revised Object `draw()` in Listing 4.127 that uses Sprites 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.

4. 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.