Program a Game Engine from Scratch

Mark Claypool

Development Checkpoint #8

Sprite & Resource Manager

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 Resource Management

Games have a wide variety and often large number of resources, also known as *assets* or *media*. Examples include meshes, models, textures, animations, audio clips, level layouts, dialog snippets and more. Offline, most game studios use tools to help create, store and archive assets during game development. When the game is running, however, the game engine needs to manage the assets itself in an efficient manner, loading, unloading and manipulating the assets as needed.

Since many assets are large, for efficiency, a game engine uses the *flyweight* design pattern, sharing as much data as possible with similar objects. For the game engine, this means keeping only one copy of each asset in memory and having all game objects that need the asset to refer to this one copy. This helps manage memory resources, which can be scarce for high-end games, or on resource constrained devices, such as mobile hand-helds. Similarly, a game engine often manages the lifetime of the asset, bringing it in from memory on demand ("streaming"), and removing it from memory when it is no longer needed. Some game engines, particularly 3d game engines, handle composite resources, such as keeping a mesh, skeleton and animations all grouped with a 3d model. Assets that have been loaded into memory sometimes need additional processing before rendering. Ideally, support for all of the above is provided in a single, unified interface for both the game programmer and for other components in the engine.

In Dragonfly, one of the assets managed are sprites, stored as text files. A game studio using Dragonfly could have offline tools that help create, edit and store sprites as part of the development environment. Such tools could help artists correctly animate and color text-based sprites, and provide revision control and archival functions for sharing and developing the sprites.

However, the Dragonfly engine itself needs to only be able to understand the format of a sprite file so that it can load it into the game when requested. To do this, data structures (classes) are required for *Frames* (see Section 4.12.1 on page 154) that provide the dimensions of the sprite and hold the data, *Sprites* (Section 4.12.2 on page 156) that provide identifiers for the asset and hold the frames, and *Animations* (Section 4.12.5 page 175) for providing support for per-Object sprite animation. The *ResourceManager* (Section 4.12.3 on page 161) provides methods for the game programmer to use the sprite assets.

4.12.1 The Frame Class

Frames in Dragonfly are simply text rectangles of any dimension that know how to draw themselves on the screen. The frames themselves do not have any color, nor do individual characters – color is an attribute associated with a Sprite. Some frame examples are shown in Listing 4.115. Frames are not animated. In order to achieve animation, sequences of frames are shown in rapid succession so it looks like animation (see Figure 3.2 on page 14).

Listing 4.115: Frame examples

```
0
1      ____          \            _____
2     /__o_\         ~==-        / ___/____ ___  __________  _____
3                    /           \__ \/ __ `/ / / / ___/ _ \/ ___/
4                               ___/ / /_/ / /_/ / /__/  __/ /
5                              /____/\__,_/\__,_/\___/\___/_/
```

The individual cells in a frame are characters. These could be stored in a two dimensional array. However, in order to use the speed and efficiency of the C++ string library class, Dragonfly stores the entire frame as a single string.

The definition for the Frame class is provided in Listing 4.116. While the attribute `m_frame_str` holds the frame data in a one dimensional list of characters, the attributes `m_width` and `m_height` determine the shape of the frame rectangle. There are two constructors: the default constructor creates an empty frame (`m_height` and `m_width` both zero with an empty `m_frame_str`)), while the method on line 14 allows construction of a frame with an initial string of characters and a given width and height. Frames know how to draw themselves at a given position with a given color, via `draw()`. Most of the rest of the Frame methods allow getting and setting the attributes.

Listing 4.116: Frame.h

```
0  #include <string>
1
2  class Frame {
3
4   private:
5    int m_width;              // Width of frame.
6    int m_height;             // Height of frame.
7    std::string m_frame_str; // All frame characters stored as string.
8
9   public:
10   // Create empty frame.
11   Frame();
12
13   // Create frame of indicated width and height with string.
14   Frame(int new_width, int new_height, std::string frame_str);
15
16   // Set width of frame.
17   void setWidth(int new_width);
18
19   // Get width of frame.
20   int getWidth() const;
21
22   // Set height of frame.
23   void setHeight(int new_height);
24
25   // Get height of frame.
26   int getHeight() const;
27
28   // Set frame characters (stored as string).
29   void setString(std::string new_frame_str);
30
31   // Get frame characters (stored as string).
32   std::string getString() const;
33
34   // Draw self, centered at position (x,y) with color.
35   // Return 0 if ok, else -1.
36   // Note: top-left coordinate is (0,0).
37   int draw(Vector position, Color color) const;
```

```
38 };
```

Most of the Frame methods are straightforward getters/setters, except for `draw()`, shown as pseudo code in Listing 4.117.

Listing 4.117: Frame draw()

```
0  // Draw self, centered at position (x,y) with color.
1  // Return 0 if ok, else -1.
2  // Note: top-left coordinate is (0,0).
3  int Frame::draw(Vector position, Color color) const;
4
5    // Error check empty string.
6    if frame is empty then
7      return error
8    end if
9
10   // Determine offset since centered at position.
11   x_offset = frame.getWidth() / 2
12   y_offset = frame.getHeight() / 2
13
14   // Draw character by character.
15   for y = 0 to m_height-1
16     for x = 0 to m_width-1
17       Vector temp_pos(position.getX() + x - x_offset,
18                       position.getY() + y - y_offset);
19       DM.drawCh(temp_pos, m_frame_str[y*m_width + x], color)
20     end for // x
21   end for // y
```

The first block of code starting on line 5 checks if the Frame is empty to avoid subsequent parsing errors. This can be checked with the `empty()` method call on the Frame string (`m_frame_str`) and, if true, an error (-1) is returned.

Subsequently, the method computes the x and y offsets since the frame is always drawn centered at the position.

The bulk of the method iterates through the frame characters one by one, drawing them on the screen by calling DisplayManager `drawCh()` with the appropriate (x,y) position, character and color.

4.12.2 The Sprite Class

Sprites are sequences of frames, typically rendered such that if a sequence is drawn fast enough, it looks animated to the eye. The Sprite class in Dragonfly is primarily a repository for the Frame data, and that knows how to draw one frame. Sprites do not know what the display rate should be for the frames for animation nor do they keep track of the last frame that was drawn – that functionality is tracked by the Animation class. Sprites record the dimension of the Sprite (typically, the same dimension of the Frames), provide the ability to add and retrieve individual Frames, and give a method to draw a Frame. A Sprite sequence may look like the example in Listing 4.118.

Listing 4.118: Sprite sequence example

```
 ____       ____       ____       ____       ____
/____\     /___o\     /__o_\     /_o__\     /o___\
```

The Sprite class header file is shown in Listing 4.119. The class needs `Frame.h` as well as `<string>`. The attributes `m_width` and `m_height` typically mirror the sizes of the frames composing the sprite. The frames themselves are stored in an array, `m_frame[]`, which is dynamically allocated when the Sprite object is created. In fact, the default constructor, `Sprite()`, is `private` since it *cannot* be called – instead, Sprites must be instantiated with the maximum number of frames they can hold as an argument (e.g., Sprite(5)). This maximum is stored in `m_max_frame_count`, while the actual number of frames is stored in `m_frame_count`. The color, which is the color of all frames in the sprite, is stored in `m_color`. Each sprite can be identified by a text label, `m_label` – for example, "ship" for the Hero's sprite in Saucer Shoot (Section 3.3).

In the normal course of animation, drawing proceeds sequentially through all the frames in a sprite until the end, then loops. By default, a sprite frame is advanced sequentially each game loop (so, 30 frames per second). However, for many animations, this will be too fast. In order to slow down the animation, the attribute `m_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.

Most of the methods are to get and set the attributes, with `addFrame()` putting a new frame at the end of the Sprite's `m_frame` array.

Listing 4.119: Sprite.h

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

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

class Sprite {

 private:
  int m_width;           // Sprite width.
  int m_height;          // Sprite height.
  int m_max_frame_count; // Max number frames sprite can have.
  int m_frame_count;     // Actual number frames sprite has.
  Color m_color;         // Optional color for entire sprite.
  int m_slowdown;        // Animation slowdown (1=no slowdown, 0=stop).
  Frame *m_frame;        // Array of frames.
  std::string m_label;   // Text label to identify sprite.
  Sprite();              // Sprite always has one arg, the frame count.

 public:
  // Destroy sprite, deleting any allocated frames.
  ~Sprite();

  // Create sprite with indicated maximum number of frames.
  Sprite(int max_frames);

  // Set width of sprite.
```

```
27    void setWidth(int new_width);
28
29    // Get width of sprite.
30    int getWidth() const;
31
32    // Set height of sprite.
33    void setHeight(int new_height);
34
35    // Get height of sprite.
36    int getHeight() const;
37
38    // Set sprite color.
39    void setColor(Color new_color);
40
41    // Get sprite color.
42    Color getColor() const;
43
44    // Get total count of frames in sprite.
45    int getFrameCount() const;
46
47    // Add frame to sprite.
48    // Return -1 if frame array full, else 0.
49    int addFrame(Frame new_frame);
50
51    // Get next sprite frame indicated by number.
52    // Return empty frame if out of range [0, m_frame_count-1].
53    Frame getFrame(int frame_number) const;
54
55    // Set label associated with sprite.
56    void setLabel(std::string new_label);
57
58    // Get label associated with sprite.
59    std::string getLabel() const;
60
61    // Set animation slowdown value.
62    // Value in multiples of GameManager frame time.
63    void setSlowdown(int new_sprite_slowdown);
64
65    // Get animation slowdown value.
66    // Value in multiples of GameManager frame time.
67    int getSlowdown() const;
68
69    // Draw indicated frame centered at position (x,y).
70    // Return 0 if ok, else -1.
71    // Note: top-left coordinate is (0,0).
72    int draw(int frame_number, Vector position) const;
73 };
```

The Sprite constructor is shown in Listing 4.120. The width, height and frame count are initialized to zero. The `m_frame` array is allocated by `new` to the indicated size. Like all memory allocation, this should be checked for success – if the needed memory cannot be allocated (not shown), an error message is written to the logfile and the maximum frame count is set to 0. The Sprite should initially have the default color, `COLOR_DEFAULT`, as defined in `Color.h` (Listing 4.74 on page 115).

Listing 4.120: Sprite Sprite()

```
// Create sprite with indicated maximum number of frames.
Sprite::Sprite(int max_frames)
  set m_frame_count to 0
  set m_width to 0
  set m_height to 0
  m_frame = new Frame [max_frames]
  set max_frame_count to max_frames
  set m_color to COLOR_DEFAULT
```

The Sprite destructor is shown in Listing 4.121. The only logic the destructor has is to check if frames are actually allocated (`frame` is not `NULL`) and, if so, `delete` the `frame` array.

Listing 4.121: Sprite ~Sprite()

```
// Destroy sprite, deleting any allocated frames.
Sprite::~Sprite()
  if m_frame is not NULL then
    delete [] m_frame
  end if
```

Once a Sprite is created, frames are typically added to it one at a time until the entire animation sequence has all been added. Pseudo code for Sprite `addFrame()`, which adds one Frame, is shown in Listing 4.122. The method first checks if the frame array (`m_frame`) is filled – if so, an error is returned. Otherwise, the new frame is added and the frame count is incremented. Remember, as in all C++ arrays, the index of the first item is 0, not 1.

Listing 4.122: Sprite addFrame()

```
// Add a frame to the sprite.
// Return -1 if frame array full, else 0.
int Sprite::addFrame(Frame new_frame)
  if m_frame_count is m_max_frame_count then // Is Sprite full?
    return error
  else
    m_frame[m_frame_count] = new_frame
    increment m_frame_count
  end if
```

Sprite `getFrame()` is shown in Listing 4.123. The first block of code checks if the frame number is outside of the range [0, `m_frame_count`-1] – if so, am empty frame is returned. Otherwise, the frame at the index indicated by `frame_number` is returned.

Listing 4.123: Sprite getFrame()

```
// Get next sprite frame indicated by number.
// Return empty frame if out of range [0, m_frame_count-1].
Frame Sprite::getFrame(int frame_number) const

  if ((frame_number < 0) or (frame_number >= frame_count)) then
    Frame empty  // Make empty frame.
    return empty // Return it.
  end if

```

```
  return frame[frame_number]
```

The Sprite `draw()` method makes a straightforward call to Frame `draw()` for the indicated frame and position, using the Sprite `m_color`. For error checking, make sure `frame_number` is within the Sprite bounds before accessing the frame.

4.12.2.1 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 Frame `draw()` method must be refactored as shown in Listing 4.124 (refer to Listing 4.117 on page 156 for the original method). The refactored `draw()` 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 4.124: Frame extension to support transparency

```
// Draw self centered at position (x,y) with color.
// Don't draw transparent characters (0 means none).
// Return 0 if ok, else -1.
// Note: top-left coordinate is (0,0).
int Frame::draw(Vector position, Color color, char transparent) const;

  ...

  // Draw character by character.
  for y = 0 to m_height-1
    for x = 0 to m_width-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 Sprite. Extensions needed to the Sprite class are shown in Listing 4.125. Transparency is stored in a `char` attribute, `m_transparency` (set to 0 in the constructor), with methods to get and set it.

Listing 4.125: Sprite class extensions to support transparency

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

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

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

Lastly, the Sprite `draw()` method needs to be modified pass in the transparency character to Frame `draw()`.

The actual transparency character is typically defined in the sprite file (e.g., transparency #). See the ResourceManager code (Section 4.12.3) for parsing sprite files, adding in the ability to handle an optional transparency character.

4.12.3 The ResourceManager

With data structures for frames and sprites in place, a manager to handle resources is needed – the ResourceManager. The ResourceManager is a singleton derived from Manager, with private constructors and a `getInstance()` method to return the one and only instance (see Section 4.2.1 on page 55). The header file, including class definition, is provided in Listing 4.126.

The ResourceManager constructor should set the type of the Manager to "ResourceManager" (i.e., `setType("ResourceManager")` and initialize all attributes.

Listing 4.126: ResourceManager.h

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

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

// Maximum number of unique assets in game.
const int MAX_SPRITES = 500;

class ResourceManager : public Manager {

 private:
  ResourceManager();                        // Private (a singleton).
  ResourceManager(ResourceManager const&); // Don't allow copy.
  void operator=(ResourceManager const&);  // Don't allow assignment.
  Sprite *m_p_sprite[MAX_SPRITES];          // Array of sprites.
  int m_sprite_count;              // Count of number of loaded sprites.

 public:
  // Get the one and only instance of the ResourceManager.
  static ResourceManager &getInstance();

  // Get ResourceManager ready to manager for resources.
```

```
24    int startUp();
25
26    // Shut down ResourceManager, freeing up any allocated Sprites.
27    void shutDown();
28
29    // Load Sprite from file.
30    // Assign indicated label to sprite.
31    // Return 0 if ok, else -1.
32    int loadSprite(std::string filename, string label);
33
34    // Unload Sprite with indicated label.
35    // Return 0 if ok, else -1.
36    int unloadSprite(std::string label);
37
38    // Find Sprite with indicated label.
39    // Return pointer to it if found, else NULL.
40    Sprite *getSprite(std::string label) const;
41 };
```

The ResourceManager uses strings for labels so needs `#include <string>`. In addition, it inherits from `Manager.h` and has methods and attributes to handle Sprites, so also `#includes` `Sprite.h`.

The ResourceManager stores Sprites in an array of pointers (the attribute `m_p_sprite[]`), bounded by `MAX_SPRITES`.

Media files, such as sprite files but also `jpeg`, `wmv` and `mp3` files, typically have three parts: 1) a header that contains key parameters for the media file, such as number of frames and playback rate, 2) a body that had the media data, often with delimiter tags, and 3) a footer with any final wrap-up information.

Listing 4.127: Saucer sprite file

```
0   5
1   6
2   2
3   4
4   green
5    ____
6   /____\
7    ____
8   /___o\
9    ____
10  /__o_\
11   ____
12  /_o__\
13   ____
14  /o___\
```

Listing 4.127 shows an example of a sprite file for the Saucer in the Dragonfly tutorial. The first four lines (with numbers) and the line right after the numbers (saying "green") are the header, containing information needed to animate the sprite. The meaning of the header lines are as follows:

1. Frames - the number of frames in the sprite.

2. Width - the width of the sprite frames.
3. Height - the height of the sprite frames.
4. Slowdown - the number of frames to "pause" between frame animations.
5. Color - the Dragonfly color (a string).

The lines following the header are the body with contents for the sprite frames. The width of each of the body lines is the same as the number on line 2 in the header (the width parameter, 6 in Listing 4.127). The number of lines for each frame is specified by the number on line 3 in the header (the height parameter, 2 in Listing 4.127). The lines are combined to makeup the frame, with the total frames specified by the number on line 1 in the header (the frames parameter, 5 in Listing 4.127).

Note, sprite files using this format are provided for most of the sprites in the Dragonfly Saucer Shoot tutorial. They are available for download with versions of the game (e.g., `https://dragonfly.wpi.edu/tutorials/saucer-shoot/game-final.zip`) and from the book web page (`https://dragonfly.wpi.edu/book/sprites.zip`), available under the `sprites-simple/` directory.

Tip 18! File input in C++. There are many ways to read from a file in C++. One example of reading from a text file (such as a Sprite file) is in Listing 4.128. The sample code treats the file as an `ifstream`, using `getline()` to read the file a line at a time. Note, the function `getline()` automatically removes the newline ('`\n`') delimiter. After each line is read, it is added to a vector (`data`) via the method `push_back()` - this is not strictly needed for just reading from a file, but is useful when the file data is later parsed (e.g., as is a sprite file). The method `good()` is true if the file still has data (and there are no other file errors) – when the end of the file is reached, `good()` returns `false`.

Listing 4.128: Example of file input

```
0  // Example of reading text file.
1  // Read one line at a time, writing each line to stdout.
2  #include <iostream>
3  #include <fstream>
4  #include <string>
5
6  using std::cout;
7  using std::endl;
8
9  int main() {
10   std::string line;
11   std::ifstream myfile ("example.txt");
12   std::vector<std::string> data;
13
14   // Open file.
```

```
  if (myfile.is_open()) {

    // Repeat until end of file.
    while (myfile.good()) {

      getline(myfile, line);  // Read line from file.
      data.push_back(line);   // Add line to vector.
      cout << line << endl;   // Display line to screen.

    }

    // Close file when done.
    myfile.close();

  } else

    // If here, unable to open file.
    cout << "unable to open file" << endl;
}
```

4.12.3.1 Loading Sprites

The method `startUp()` gets the ResourceManager ready for use – basically, just setting `m_sprite_count` to 0 and calling `Manager::startUp()`.

The method `loadSprite()` reads in a sprite from a file indicated by `filename`, stores it in a Sprite and associates a label string with it. The method `unloadSprite()` unloads a Sprite with a given `label` from memory, freeing it up. The method `getSprite()` returns a pointer to a Sprite with the given label.

The ResourceManager `loadSprite()` is shown in Listing 4.129. The name of the sprite file is passed in as a string (`filename`), along with a string with the label name to associate with the sprite once it is successfully loaded (`label`).

Listing 4.129: ResourceManager loadSprite()

```
// Load Sprite from file.
// Assign indicated label to sprite.
// Return 0 if ok, else -1.
int ResourceManager::loadSprite(std::string filename, std::string label)

  // Check if room in array.
  if m_sprite_count is MAX_SPRITES then // Sprite array full?
    return error
  end if

  open file

  // Read sprite Header.
  Get first line from file
  frames = atoi() on line
  get second line from file
  width = atoi() on line
  get third line from file
  height = atoi() on line
```

```
19    get fourth line from file
20    slowdown = atoi() on line
21    get fifth line from file
22    if line is "black" then
23      color = BLACK
24    else if line is "red" then
25      color = RED
26    ...
27    end if
28
29    // Make new Sprite.
30    new Sprite (with frame count)
31    set sprite height
32    set sprite width
33    set sprite slowdown
34    set sprite color
35
36    // Read and add frames to Sprite.
37    for f = 1 to frame count
38      create empty string
39      for h = 1 to height
40        get line from file
41        append line to string
42      end for
43      set frame string to string
44      set frame height
45      set frame width
46      add frame to Sprite
47    end for
48
49    close file
50
51    // If no errors in any of above, add to resource manager.
52    add label to Sprite
53    m_p_sprite[m_sprite_count] = p_sprite
54    increment m_sprite_count
55
56    return ok
```

The method starts by opening the file indicated by `filename`. After that, all the lines in the header are read in and parsed one by one, lines 13 to 27. Once the number of frames is known from the header, on line 30 a new Sprite is created (e.g., if the sprite has 5 frames, then `new Sprite(5)`). Then, the method loops for each frame (starting on line 37), reading each line and adding it to the frame string (lines 39 to 42). When a complete frame is read in, the frame attributes are set (lines 43 to 45) and the frame is added to the Sprite (line 46). When the specified (in the header) number of frames are read in, the file is closed. Assuming everything above went well, the final steps from line 52 are to: 1) associate `label` with the Sprite, 2) add the Sprite to the `m_p_sprite` array, and 3) increase `m_sprite_count`.

Note, error checking should be done throughout, checking header information, (e.g., header lines corresponding to expected parameters), length of lines (e.g., body lines exactly as long as the frame width), number of lines (e.g., exactly enough for the specified number of frames, each the specified height), and general file read errors. If any errors are

encountered, the line number in the file where the error occurred should be reported along with a descriptive error in the logfile. Listing 4.130 shows an example of a possible error message. In this example, line 12 of the file has "`/o___\ `" which is 7 characters (there is an extra ' ' at the end, a common error), while the header had indicated the width was only 6. Making the error message as descriptive as possible is helpful to game programmers to help "debug" their sprite files. Upon encountering an error, all resources should be cleaned up (i.e., `delete` the Sprite and close the file), as appropriate.

Listing 4.130: Example error reported when parsing Sprite file

```
0  Loading 'explosion' from file 'sprites/explosion-spr.txt'.
1  Error line 12.  Line width 7, expected 6.
2  Line is: "/o___\ "
3  Sprite 'explosion' not loaded.
```

Dragonfly can run on Windows, Linux or Mac computers. Unfortunately, text files are treated slightly differently on Windows versus Linux and Mac. In Windows text files, the end of each line has a newline ('`\n`') character *and* a carriage return ('`\r`') character, while in Unix and Mac, the end of each line only has a newline character. In order to allow Dragonfly to work with text files created on any of the three operating systems, pseudo code for an optional utility, `discardCR()`, is shown in Listing 4.131. A `string`, typically just read in from a file, is passed in via reference (`&str`). The function examines the last character of this string and, if it is a carriage return, it removes it via `str.erase()`.

Listing 4.131: discardCR() (optional)

```
0  // Remove the carriage return ('\r') from line
1  // (if there - typical of Windows).
2  void discardCR(std::string &str)
3    if str.size() > 0 and str[str.size()-1] is '\r' then
4       str.erase(str.size() - 1)
5    end if
```

Once in place, `discardCR()` can be called each time after reading a line from a file.

The complement of `loadSprite()` is `unloadSprite()`, which is much simpler. `unLoadSprite()` is shown in Listing 4.132. The method loops through the Sprites in the ResourceManager. If the label being looked for (`label`) matches the label of one of the Sprites (`getLabel()`) then that is the Sprite to be unloaded. The Sprite's memory is deleted via `delete`. Then, in a loop starting on line 11, the rest of the Sprites in the array are moved down one. Since one Sprite was unloaded, the sprite count is decremented on line 11. If the loop terminates without a label match, the sprite to be unloaded is not in the ResourceManager and an error is returned.

Listing 4.132: ResourceManager unLoadSprite()

```
0  // Unload Sprite with indicated label.
1  // Return 0 if ok, else -1.
2  int ResourceManager::unloadSprite(std::string label)
3
4    for i = 0 to m_sprite_count-1
5
6      if label is m_p_sprite[i] -> getLabel() then
```

```
7
8       delete m_p_sprite[i]
9
10      // Scoot over remaining sprites.
11      for j = i to m_sprite_count-2
12        m_p_sprite[j] = m_p_sprite[j+1]
13      end for
14
15      decrement m_sprite_count
16
17      return ok
18
19    end if
20
21  end for
22
23  return error // Sprite not found.
```

The final method needed by the ResourceManager is `getSprite()`, with pseudo code show in Listing 4.133. The method loops through all the Sprites in the ResourceManager. The first Sprite that matches the label `label` is returned. If line 10 is reached, the label was not found and an error (`NULL`) is returned.

Listing 4.133: ResourceManager getSprite()

```
0 // Find Sprite with indicated label.
1 // Return pointer to it if found, else NULL.
2 Sprite *ResourceManager::getSprite(std::string label) const
3
4   for i = 0 to m_sprite_count-1
5     if label is m_p_sprite[i] -> getLabel() then
6       return m_p_sprite[i]
7     end if
8   end for
9
10  return NULL // Sprite not found.
```

Lastly, ResourceManager `shutDown()`, shown in Listing 4.134 frees up any allocated Sprites by iterating through the `m_p_sprite` array and calling `delete` on each. After that, it sets the array count to zero and calls `Manager::shutDown()`.

Listing 4.134: ResourceManager shutdown()

```
0 // Shut down manager, freeing up any allocated assets.
1 void ResourceManager::shutDown()
2   for i = 0 to m_sprite_count-1
3     if m_p_sprite[i] not NULL then
4       delete m_p_sprite[i]
5     end if
6   end for
7
8   set m_sprite_count to 0
9
10  call Manager::shutDown()
```

4.12.3.2 Sprites with Robust Formatting (optional)

The trouble with the sprite file format as it is currently specified is that it is not very forgiving of errors for the artists that created the sprites. The header numbers (e.g., width and height) have to be specified in *exactly* the right order (e.g., not height and then width) or the sprite will not parse properly, but there are no keywords or guides to help the artist "debug" their sprite files when they create them. In short, the file format is "brittle". While this makes the code to parse sprite files easier, and hence is used for the initial implementation, a more "robust" format for creating sprites can help facilitate the art-game production pipeline.

See the Dragonfly tutorial for an example of a sprite file with a "robust" format.

For robust Dragonfly sprite files, the delimiters are indicated with all caps (e.g., `HEADER`) in order to make creating and parsing sprite files easier. For parsing code, `HEADER`, `BODY`, and `FOOTER` are used to deliminate the header, body and footer of the sprite, respectively. In the header, the keywords `frames`, `height`, `width`, `slowdown`, and `color` define parameters for the sprite. The header parameters can be in any order. In the body, `end` is used to mark the end of each frame. In the footer, `version` is used to indicate sprite version information.

To provide support for sprites with robust formatting, add the `const`s in Listing 4.135 to the ResourceManager header file. These are delimiters used to parse the sprite files – the ResourceManager "understands" the file format and uses the delimiters as tokens during parsing.

Listing 4.135: ResourceManager.h extensions to support a robust sprite format

```
// Delimiters used to parse Sprite files -
// the ResourceManager 'knows' file format.
const std::string HEADER_TOKEN = "HEADER";
const std::string BODY_TOKEN = "BODY";
const std::string FOOTER_TOKEN = "FOOTER";
const std::string FRAMES_TOKEN = "frames";
const std::string HEIGHT_TOKEN = "height";
const std::string WIDTH_TOKEN = "width";
const std::string COLOR_TOKEN = "color";
const std::string SLOWDOWN_TOKEN = "slowdown";
const std::string END_FRAME_TOKEN = "end";
const std::string VERSION_TOKEN = "version";
```

Then, a re-factored version of the ResourceManager `loadSprite()` is shown in Listing 4.136.

Listing 4.136: ResourceManager loadSprite()

```
// Load Sprite from file.
// Assign indicated label to sprite.
// Return 0 if ok, else -1.
int ResourceManager::loadSprite(std::string filename, std::string label)

  open file filename

  read all header lines // header has sprite format data
  parse header

```

```
10    read all body lines // body has frame data
11    new Sprite (with frame count)
12    set sprite height
13    set sprite width
14    set sprite slowdown
15    set sprite color
16    for f = 1 to frame count
17      parse frame
18      add frame to Sprite
19    end for
20
21    read all footer lines // footer has sprite version
22    parse footer
23
24    close file
25
26    // If no errors in any of above, add to resource manager.
27    add label to Sprite
28    m_p_sprite[m_sprite_count] = p_sprite
29    increment m_sprite_count
```

The method proceeds by opening the file indicated by `filename`. After that, all the lines in the header are first read in and then parsed. Once the number of frames is known from the header, on line 11 a new Sprite is created (e.g., if the sprite has 5 frames, then `new Sprite(5)`). Then, the method reads in all the lines in the body, and parses them as frames, one frame at a time. Each frame is added to the Sprite as it is parsed. When the specified (in the header) number of frames are read in, all the lines in the footer are read in and then parsed. Assuming everything above went well, the final steps from line 27 are to: 1) associate `label` with the Sprite, 2) add the Sprite to the `m_p_sprite` array, and 3) increase `m_sprite_count`.

Note, as for the simple format, error checking should be done throughout, looking at file format, length of line, number of lines, frame count and general file read errors. If any errors are encountered, the line number in the file should be reported along with a descriptive error in the logfile.

Coding the `loadSprite()` method is much easier with a few "helper" functions, shown in Listing 4.137.

Function `getLine()` reads a single line from a file and does some error checking.

Function `readData()` reads a section (e.g., the body, recognized by the `delimiter`, such as `BODY`) from the sprite file and stores each line in a vector for later parsing. And error is indicated (an empty vector is returned) of the section beginning and end is not found.

Function `matchLineInt()` matches a specified token from the vector (a sprite file section), returning the associated integer value. For example, the line "frames 5" called with a tag of "frames" would return the integer "5". Lines that match are removed from the vector.

Function `matchLineStr()` does the same thing, but returning associated string found. For example, the line "color green" called with a tag of "color" would return the string "green").

Function `matchFrame()` reads a frame of a given width and height from a file, returning the Frame. The used frame lines are removed from the vector.

All functions should report any parsing errors in the logfile. None of these methods are part of the ResourceManager, but rather are stand alone utility functions. They are not general engine utility functions either (i.e., it is unlikely a game programmer would ever use them), so do not really belong in `utility.cpp`. Instead, they can be placed directly into `ResourceManager.cpp`.

Listing 4.137: ResourceManager helper functions for loading sprites

```
// Get next line from file, with error checking ("" means error).
std::string getLine(std::ifstream *p_file);

// Read in next section of data from file as vector of strings.
// Return vector (empty if error).
std::vector<std::string> readData(std::ifstream *p_file,
                                  std::string delimiter);

// Match token in vector of lines (e.g., "frames 5").
// Return corresponding value (e.g., 5) (-1 if not found).
// Remove any line that matches from vector.
int matchLineInt(std::vector<std::string> *p_data, const char *token);

// Match token in vector of lines (e.g., "color green").
// Return corresponding string (e.g., "green") ("" if not found).
// Remove any line that matches from vector.
std::string matchLineStr(std::vector<std::string> *p_data, const char *
    token);

// Match frame lines until "end", clearing all from vector.
// Return Frame.
Frame matchFrame(std::vector<std::string> *p_data, int width, int height);
```

Function `getLine()` is shown in Listing 4.138. The function reads one line from the file into the string `line`. The code also needs to `#include` the `fstream` header file. Note, error checking (`m_p_file -> good()` is done to catch any file input errors.

Listing 4.138: getLine()

```
// Get next line from file, with error checking ("" means error).
std::string getLine(std::ifstream *p_file)

  string line
  getline(*p_file, line)
  if not (p_file -> good()) then
    return error
  end if

  return line
```

Function `readData()` is shown in Listing 4.139. The function takes in a token that is used to delimit the section of the sprite file (i.e., HEADER, BODY, FOOTER) and the file to be read from. It then pulls all the lines from the file using the delimiter to mark the beginning and end, storing the "good" lines as data in an `std::vector`. That vector is returned. Errors are checked for missing the delimiter beginning or end and file errors.

Listing 4.139: readData()

```
// Read in next section of data from file as vector of strings.
// Return vector (empty if error).
std::vector<std::string> readData(std::ifstream *p_file,
                                  std::string delimiter)

  beginning = "<" + delim + ">"  // Section beginning
  ending = "</" + delim + ">"    // Section ending

  // Check for beginning.
  s = getLine()
  if s not equal beginning then
    return error
  end if

  // Read in data until ending (or not found).
  s = getLine()
  while (s not equal ending) and (not s.empty()) do
    push_back(s) onto data
    s = getLine()
  end while

  // If ending not found, then error.
  if s.empty() then
    return error
  end if

  return data
```

Function `matchLineInt()` is shown in Listing 4.140. The function examines the data vector one line at a time, looking for a match of the indicated `token`. The match on Line 9 can be made using `compare()`, – e.g.,

```
  i -> compare(0, strlen(token), token)
```

If the token is the one expected, the remainder of the string after the token is converted on Line 10 into an integer using `atoi()` – e.g.,

```
  atoi(line.substr(strlen(token)+1).c_str())
```

The number extracted with `atoi()` is returned.

Listing 4.140: matchLineInt()

```
// Match token in vector of lines (e.g., "frames 5").
// Return corresponding value (e.g., 5) (-1 if not found).
// Remove any line that matches from vector.
int matchLineInt(std::vector<std::string> *p_data,
                 const char *token)

  // Loop through all lines.
  auto i = p_data -> begin() // vector iterator
  while i not equals p_data -> end()
    if i equals token then
      number = atoi() on line.substr()
```

```
11        i = p_data -> erase(i) // clear from vector
12      else
13        increment i
14      end if
15    end while
16
17    return number
```

The same logic is used for `matchLineStr()` with the exception that the final string after the token is not converted to an integer, but is instead just returned (e.g., `line.substr( strlen(token) + 1 )`).

The method `matchFrame()` is shown in Listing 4.141. The function is provided the height of the frame via the `height` parameter. So, using a `for` loop, the function loops for a count of the height of the frame, handling a line at a time. Each line represents one row of the frame. If any line is not the right width (also passed in as a parameter, via `width`), an error is returned in the form of an "empty" Frame. If the frame is read in successfully, an additional line is handled, shown on line 20. Since the frame is over, this line should be `END_FRAME_TOKEN` ("end"), otherwise there is an error in the file format.

Errors of any kind should result in an error code (empty Frame) returned by the function. If line 26 is reached, the frame has been read and parsed successfully, so a Frame object containing the frame is created and returned. The line number should be used to report (in the logfile) where any errors occurred in the input file, and should be appropriately incremented as the parsing progresses.

Listing 4.141: matchFrame()

```
0  // Match frame lines until "end", clearing all from vector.
1  // Return Frame (empty if error).
2  Frame matchFrame(std::vector<std::string> *p_data, int width, int height)
3
4    string line, frame_str
5
6    for h = 1 to height
7
8      line = p_data -> front()
9
10     if line width != width then
11       return error
12     end if
13
14     p_data -> erase(p_data->begin())
15
16     frame_str += line
17
18   end for
19
20   line = p_data -> front()
21   if line is not  END_FRAME_TOKEN then
22     return error
23   end if
24   p_data -> erase(p_data->begin())
25
26   create frame (width, height, frame_str)
```

```
27
28  return frame
```

With robust sprite formatting in place, it can be convenient for Dragonfly to support both the original simple sprite file format as well as the robust sprite file format. A technique that can "auto-detect" whether a sprite file is in the simple format or the robust format is shown in Listing 4.142. Basically, the first line of the file is read. If this line is the string "¡HEADER¿", it is assumed the file has a robust sprite file format and loading proceeds assuming this format. Otherwise, it is assumed the file has a simple sprite file format. In this (simple) case, the first line should be an integer (the number of frames), so the line is converted to a number (using `atoi()`) and checked. If the number is not a positive, an error is indicated – the file may not be a sprite file at all.

Listing 4.142: Auto-detecting sprite file format.

```
0
1   get line from file
2
3   if line is "<HEADER>" then
4     loadRobustSprite()
5   else if atoi() on line > 0
6     loadSimpleSprite()
7   else
8     error // unknown format
9   end if
```

4.12.4 Development Checkpoint #8!

Continue Dragonfly development. Steps:

1. Make the Frame class, referring to Listing 4.116. Add `Frame.cpp` to the project and stub out each method so it compiles. Implement and test the Frame class outside of any game engine components, making sure it can be loaded with different strings and frame dimensions.

2. Implement the Frame `draw()`, referring to Listing 4.117. Test with a variety of Frames and positions outside of an actual Sprite or game Object.

3. Make the Sprite class, referring to Listing 4.119. Add `Sprite.cpp` to the project and stub out each method so it compiles. Code and test the simple attributes first (`int`s and `label`).

4. Implement the constructor `Sprite(int max_frames)` next, allocating the array. Implement `addFrame()` based on Listing 4.122 and `getFrame()` based on Listing 4.123. Test that you can create Sprites of different numbers of frames and add individual Frames to them. Be sure the upper limit on frame count is protected. Testing should be done outside of the other engine components, and Frames can be arbitrary strings at this point.

5. Implement the Sprite`draw()` and test with a variety of Sprites (use those from Chapter3 – Saucer Shoot), again still outside of a game Object.

6. Make the ResourceManager, as a singleton (described in Section 4.2.1 on page 55), referring to Listing 4.126. Add `ResourceManager.cpp` to the project and stub out each method so it compiles.

7. Implement `loadSprite()` referring to Listing 4.129. For testing, sprites with a simple format can be made from the Sprite files from Saucer Shoot by keeping the line orders the same, but removing all keywords (e.g., "width", "height" and "end"). Test thoroughly. Purposefully introduce errors – to the headers (e.g., count, number), body (frame data), and footer – and verify that all errors are caught and helpful error messages reported in the logfile on the right lines.

8. Implement `getSprite()` based on Listing 4.133 and `unloadSprite()` based on Listing 4.132. Test each method thoroughly. Write test code that uses all the ResourceManager methods, loading a Sprite, getting frames, and unloading a Sprite. Repeat for multiple sprites.