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 5.0). 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 have 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 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 144) that provide the dimensions of the sprite and hold the data, and *Sprites* (Section 4.12.2 on page 145) that provide identifiers for the asset and hold the frames. The *ResourceManager* (Section 4.12.3 on page 148) 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. 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 4.108. 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 13).

Listing 4.108: 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, the entire frame is stored as a single string.

The definition for the Frame class is provided in Listing 4.109. While the attribute `frame_str` holds the frame data in a one dimensional list of characters, the attributes `width` and `height` determine the shape of the frame rectangle. There are two constructors: the default constructor creates an empty frame (`height` and `width` both zero with an empty `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. The Frame is essentially a container, and most of the methods allow getting and setting the attributes.

Listing 4.109: 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 };
```

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 just a repository for

the Frame data. Sprites do not know how to draw themselves, nor even what the display rate should be for the frames – that functionality is provided by the Object class. Sprites record the dimension of the Sprite (typically, the same dimension of the Frames), along with providing the ability to add and retrieve individual Frames. An Sprite sequence may look like the example in Listing 4.110.

Listing 4.110: Sprite sequence example

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

The Sprite class header file is shown in Listing 4.111. The class needs `Frame.h` as well as `<string>`. The attributes `width` and `height` typically mirror the sizes of the frames composing the sprite. The frames themselves are stored in an array, `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 `max_frame_count`, while the actual number of frames is stored in `frame_count`. The color, which is the color of all frames in the sprite, is stored in `color`. Each sprite can be identified by a text label, `label` – for example, "ship" for the Hero's sprite in Saucer Shoot (Section 3.3). Most of the methods are to get and set the attributes, with `addFrame()` putting a new frame at the end of the Sprite's `frame` array.

Listing 4.111: Sprite.h

```
0  // System includes.
1  #include <string>
2
3  // Engine includes.
4  #include "Frame.h"
5
6  class Sprite {
7
8   private:
9    int m_width;              // Sprite width.
10   int m_height;             // Sprite height.
11   int m_max_frame_count;    // Max number frames sprite can have.
12   int m_frame_count;        // Actual number frames sprite has.
13   Color m_color;            // Optional color for entire sprite.
14   Frame *m_frame;            // Array of frames.
15   std::string m_label;      // Text label to identify sprite.
16   Sprite();               // Sprite always has one arg.
17
18  public:
19   // Destroy sprite, deleting any allocated frames.
20   ~Sprite();
21
22   // Create sprite with indicated maximum number of frames.
23   Sprite(int max_frames);
24
25   // Set width of sprite.
26   void setWidth(int new_width);
27
```

```
28    // Get width of sprite.
29    int getWidth() const;
30
31    // Set height of sprite.
32    void setHeight(int new_height);
33
34    // Get height of sprite.
35    int getHeight() const;
36
37    // Set sprite color.
38    void setColor(Color new_color);
39
40    // Get sprite color.
41    Color getColor() const;
42
43    // Get total count of frames in sprite.
44    int getFrameCount() const;
45
46    // Add frame to sprite.
47    // Return -1 if frame array full, else 0.
48    int addFrame(Frame new_frame);
49
50    // Get next sprite frame indicated by number.
51    // Return empty frame if out of range [0, frame_count].
52    Frame getFrame(int frame_number) const;
53
54    // Set label associated with sprite.
55    void setLabel(std::string new_label);
56
57    // Get label associated with sprite.
58    std::string getLabel() const;
59 };
```

The Sprite constructor is shown in Listing 4.112. The width, height and frame count are initialized to zero. The `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 4.69 on page 106).

Listing 4.112: Sprite Sprite()

```
0 // Create sprite with indicated maximum number of frames.
1 Sprite::Sprite(int max_frames)
2   set frame_count to 0
3   set width to 0
4   set height to 0
5   frame = new Frame [max_frames]
6   set max_frame_count to max_frames
7   set color to COLOR_DEFAULT
```

The Sprite destructor is shown in Listing 4.113. The only logic the destructor has is to check that frames are actually allocated (`frame` is not `NULL`) and, if so, `delete` the `frame` array.

Listing 4.113: Sprite ~Sprite()

```
// Destroy sprite, deleting any allocated frames.
Sprite::~Sprite()
  if frame is not NULL then
    delete [] 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 4.114. The method first checks if the array 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 4.114: Sprite addFrame()

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

Sprite `getFrame()` is shown in Listing 4.115. The first block of code checks if the frame number is outside of the range [0, `frame_count`-1] – if so, am empty frame is returned. Otherwise, the frame at the index indicated by `frame_count` is returned.

Listing 4.115: Sprite getFrame()

```
// Get next sprite frame indicated by number.
// Return empty frame if out of range [0, frame_count].
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]
```

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 53). The header file, including class definition, is provided in Listing 4.116.

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

Listing 4.116: 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 = 1000;

// Delimiters used to parse Sprite files -
// the ResourceManager 'knows' file format.
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 END_FRAME_TOKEN = "end";
const std::string END_SPRITE_TOKEN = "eof";

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.
  int startUp();

  // Shut down ResourceManager, freeing up any allocated Sprites.
  void shutDown();

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

  // Unload Sprite with indicated label.
  // Return 0 if ok, else -1.
  int unloadSprite(std::string label);

  // Find Sprite with indicated label.
  // Return pointer to it if found, else NULL.
  Sprite *getSprite(std::string label) const;
};
```

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 `p_sprite[]`), bounded by `MAX_SPRITES`. The other `const`s are delimiters used to parse the sprite files – the ResourceManager "understands" the file format and uses the delimiters as tokens during parsing. See Listing 3.3 on page 17 for an example of a sprite file. For parsing, `frames`, `height`, `width` and `color` define parameters for the sprite, while `end` and `eof` are used to mark the end of each frame and the end of the file, respectively.

The method `startUp()` gets the ResourceManager ready for use – basically, just setting `sprite_count` to 0 and calling `Manager::startUp()`. The complement, `shutDown()`, frees up any allocated Sprites (e.g., `delete p_sprite[1]`) and calls `Manager::shutDown()`.

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 4.117. 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 4.117: ResourceManager loadSprite()

```
0  // Load Sprite from file.
1  // Assign indicated label to sprite.
2  // Return 0 if ok, else -1.
3  int ResourceManager::loadSprite(std::string filename, std::string label)
4    open file filename
5    read header
6    new Sprite (with frame count)
7    while not eof do
8      read frame
9      add frame to Sprite
10   end while
11   close file
12   add label to Sprite
```

The method proceeds by opening the file indicated by `filename`. After that, the header is read and parsed. Once the number of frames is known from the header, on line 6 a new Sprite is created (e.g., if the sprite has 5 frames, then `new Sprite(5)`). Then, the method loops through the rest of the file, reading a frame at a time and adding it to the Sprite. When the `eof` token is found (actually, `END_SPRITE_TOKEN)`, the parsing stops and the loop exits. The final step on line 12 is to associate `label` with the Sprite.

Note, 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. Listing 4.118 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 file had indicated the width was only 6. Making the error message as descriptive as possible is helpful to game programmers that may have "bugs" in 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 4.118: Example error reported when parsing Sprite file

```
Error line 12.  Line width 7, expected 6.
Line is: "/o___\ "
```

Coding the `loadSprite()` method is much easier with a few "helper" functions, shown in Listing 4.119. Function `readLineInt()` reads a single line from a file and expects it to be in the form of "`tag number`" and returns the number (e.g., the line "frames 5" called with a tag of "frames" would return 5). Function `readLineStr()` does the same thing, but expects `number` to be a string, returning the string found. Function `readFrame()` reads a frame of a given width and height from a file, returning the Frame. All three methods take in the line number so that if they encounter an error, they can report (in the logfile) on which line number the error occurred. They use a pointer to the line number so they can increment it for each line read. 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 4.119: ResourceManager helper functions for loading sprites

```
// Read single line 'tag number', return number (e.g., "frames 5").
int readLineInt(ifstream *p_file, int *p_line_num,
                const char *tag);

// Read single line 'tag string', return string (e.g., "color green").
std::string readLineStr(ifstream *p_file, int *p_line_num,
                        const char *tag);

// Read frame until 'eof', return Frame.
Frame readFrame(ifstream *p_file, int *p_line_number,
                int width, int height);
```

Function `readLineInt()` is shown in Listing 4.120. The function reads one line from the file into the string `line`. Note, error checking (`p_file -> good()` needs to be done on this operation. The `if` conditional uses `line.compare()` to see if the `tag` is the same as the first part of the `line` (e.g., `line.compare(0, strlen(tag), tag)`). If the tag is not the one expected, an error is returned. If the tag is the one expected, the remainder of the string after the tag is converted into an integer using `atoi()` – e.g.,

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

The number extracted with `atoi()` is returned. The line number in the file should be appropriately incremented.

Listing 4.120: readLineInt()

```
// Read single line of form 'tag number', return number.
int readLineInt(ifstream *p_file, int *p_line_number,
                const char *tag)

  string line

  getline() into line
```

```
7   if not line.compare(line, tag) then
8     return error // Not right tag.
9   end if
10
11  number = atoi() on line.substr()
12
13  increment line number
14
15  return number
```

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

The method `readFrame()` is shown in Listing 4.121. The function is provided the height of the frame via `height` as a parameter. So, via a `for` loop, it loops for a count of the height of the frame, reading in 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 read from the file, shown on line 16. Since the frame is over, this line should be `END_FRAME_TOKEN` ("end"), otherwise there is an error in the file format. Note, error checking (`p_file -> good()`) needs to be done on the file read operations. Errors of any kind should result in an error code (empty Frame) returned by the function. If line 21 is reached, the frame has been read and parsed successfully, so a Frame 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 4.121: readFrame()

```
0  // Read frame until 'eof', return Frame.
1  Frame readFrame(ifstream *p_file, int *p_line_number, int width, int height
       )
2
3   string line, frame_str
4
5   for i = 1 to height
6
7     getline() into line // Error check.
8     if line width != width then
9       return error (empty Frame)
10    end if
11
12    frame_str += line
13
14  end for
15
16  getline() into line // Error check.
17  if line is not  END_FRAME_TOKEN then
18    return error (empty Frame)
19  end if
20
21  create frame (width, height, frame_str)
22
23  return frame
```

Dragonfly should run on both Windows, Linux or Mac. 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') *and* a carriage return ('\r'), while in Unix and Mac, the end of each line only has a newline. In order to allow Dragonfly to work with text files created on either operating system, pseudo code for an optional utility, `discardCR()`, is shown in Listing 4.122. 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 4.122: discardCR() (optional)

```
0 // Remove '\r' from line (if there - typical of Windows).
1 void discardCR(std::string &str)
2  if str.size() > 0 and str[str.size()-1] is '\r' then
3     str.erase(str.size() - 1)
4  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 4.123. 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 4.123: 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 sprite_count-1
5
6      if label is p_sprite[i] -> getLabel() then
7
8        delete p_sprite[i]
9
10       // Scoot over remaining sprites.
11       for j = i to sprite_count-2
12         p_sprite[j] = p_sprite[j+1]
13       end for
14
15       decrement sprite_count
16
17       return success
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 4.124. 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 4.124: 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 sprite_count-1
5      if label is p_sprite[i] -> getLabel() then
6        return p_sprite[i]
7      end if
8    end for
9
10   return NULL // Sprite not found.
```

4.12.4 Development Checkpoint #8!

Continue Dragonfly development. Steps:

1. Make the Frame class, referring to Listing 4.109. 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. Make the Sprite class, referring to Listing 4.111. 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`).

3. Implement the constructor `Sprite(int max_frames)` next, allocating the array. Implement `addFrame()` based on Listing 4.114 and `getFrame()` based on Listing 4.115. 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.

4. Make the ResourceManager, as a singleton (described in Section 4.2.1 on page 53), referring to Listing 4.116. Add `ResourceManager.cpp` to the project and stub out each method so it compiles.

5. Build helper functions from Listing 4.119, using logic from Listing 4.120 and Listing 4.121. Test each helper function thoroughly, outside of the ResourceManager or any other engine code. Use Sprite files from Saucer Shoot (e.g., the Saucers (Section 3.3.2 on page 16)) rather than custom sprites. Purposefully introduce errors – to the headers, frames, and frame count – and verify that the helper functions catch all errors and report helpful error messages in the logfile on the right lines.

6. Return to the ResourceManager and implement `loadSprite()` based on Listing 4.117, using the helper functions. Test thoroughly with a variety of Sprite files, verifying working code through logfile messages.

7. Implement `getSprite()` based on Listing 4.124 and `unloadSprite()` based on Listing 4.123. Test each method thoroughly. Write test code that uses all the ResourceManager methods, loading at a Sprite, getting frames, and unloading a Sprite. Repeat for multiple sprites.