Program a Game Engine from Scratch

Mark Claypool

Chapter 3 - Tutorial

This document is part of the book "Dragonfly – Program a Game Engine from Scratch", (Version 6.0). Information online at: `http://dragonfly.wpi.edu/book/`

Chapter 3

Tutorial

This chapter provides a tutorial to make a game from scratch using the Dragonfly game engine. There are several goals for this chapter:

- Setup a development environment that is the same as the one needed to build the Dragonfly engine.
- Build a game with Dragonfly to provide foundational knowledge, useful for understanding how the engine is build piece-by-piece in subsequent chapters.
- Become familiar with Dragonfly by making a game from the game programmer's perspective. The game is made from scratch, illustrating most of the functionality of the dragonfly engine. For some, this may be the first game developed using a game engine. As such, it also provides an example of where game code ends and game engine code begins.
- Provide appropriate scope for ideas for subsequent Dragonfly games of one's own invention.

3.1 Overview

The game created in the tutorial is called *Saucer Shoot*, a 2-d shoot 'em up. True to the nature of Dragonfly, Saucer Shoot uses the keyboard for input and provides text-based graphics for output. It may be illustrative to see exactly what text-based graphics means. Figure 3.1 depicts a screenshot of Saucer Shoot. The blue object on the left is the player-controlled ship, while the green objects towards the right are saucers. The player has fired some missiles, show blue slightly to the right of the ship, and the red explosion is the result of a missile hitting a saucer. For an action visual, a video of the game is available on YouTube at: `http://youtu.be/rVIxygRactI`.

Before beginning the tutorial, it may be helpful to have an overview of what game programmer code using Dragonfly (and most engines) looks like. From 10,000 feet, game programmer code looks like:

1. *Startup Game Engine.* This step initializes graphics, input devices and file logging.

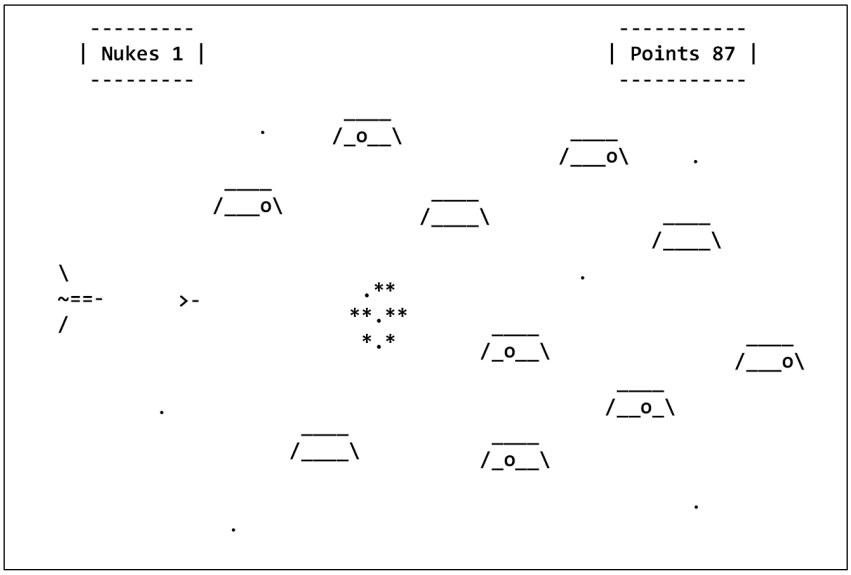

Figure 3.1: Saucer Shoot screenshot

2. *Populate World with Objects.* This step puts initial game objects into the world, such as a player object (e.g., the Hero) and bad-guy objects (e.g., the Saucers).

3. *Run Game.* Once initialized and the world populated, the game engine runs the game by moving objects, getting input, sending events to objects (e.g., collisions), and drawing on the screen

4. *Shutdown.* When the game is finished, the game engine cleanly shuts down services and returns the computer to the original state.

Explicit attributes of the Dragonfly engine encountered in the tutorial include:

- *Startup.* Invoking the game engine managers needed to start the game.
- *Resource loading.* Getting the game sprites, such as the sequence of frames that make a Saucer appear to rotate, into the game.
- *Objects.* Initial objects that are created once and persist, such as the Hero and Stars, and dynamic objects that spawn throughout the game, such as Bullets, Saucers and Explosions.
- *Events.* Built-in events, such as Keyboard and Collision events, as well as custom events, such as Nuke.
- Debugging. While not explicit in the tutorial, invariably bugs in development code will be encountered, and the game programmer exposed to debugging game code.

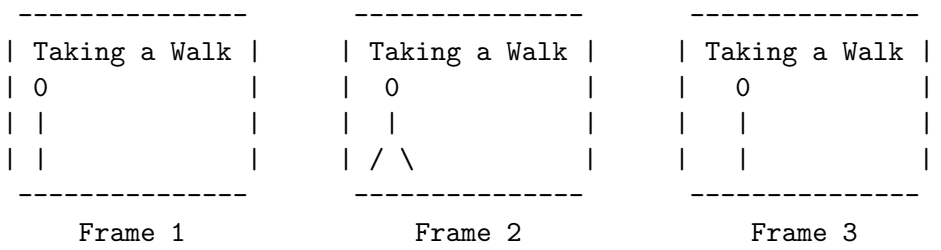

Figure 3.2: Text-based animation of walk-cycle

3.2 Text-based Graphics

In making this game, and then making Dragonfly in later chapters, the question may arise as to why only text-based graphics are supported. In a nutshell, creating and providing support for text-based graphics is easy, much easier than 2d and especially 3d graphics. The 2d, grid-based layout of text is conceptually easy to design and implement, allowing the programmer, both the game programmer and the game engine programmer, to focus on other aspects of development. Along these lines, text-based graphics has the added benefit of making the size and scope of the programming effort to create a game engine from scratch manageable, meaning it is more likely the aspiring programmer will complete the work.

Text graphics have basic (x,y) coordinates with a potential "canvas" as big as as the terminal window.[1] Text is supported by nearly all computers and there are even libraries solely used for text graphical output, such as Curses. With text as the lowest common denominator, Dragonfly can be developed in a mostly platform-independent fashion.

Lastly, during development, having art limited to just text characters is a blessing in disguise for many development projects. Limited to text-only graphics, there is no need, nor even the ability, to spend time making detailed graphics, much less realistic graphics, since the underlying characters do not support them. And for many games, the visuals do not make or break the game – a game that is boring with simple characters and test-based shapes will *not* suddenly be fun with fancy graphics. Conversely, a game with interesting gameplay and eye-popping graphics will *still* be fun with basic text-based graphics. Text-based games reduce the temptation for the programmer to spend time on art, and the development emphasis shifts to the game design and programming. In other words, "programmer art" is perfectly fine for developing games with the Dragonfly game engine!

An immediate question may be how does animation occur with text-based graphics. This can be answered with a simple example. Assume a basic walk-cycle animation of a stick-figure is needed. Figure 3.2 illustrates 3 frames that show how such an animation might be constructed with text graphics.

Frame 1 is displayed first, replaced by Frame 2 and then Frame 3. If done fast enough, the eye is tricked into seeing motion. This is the same as a flip-book animation seen in stores, or perhaps doodled on a notepad by bored kids in class. Only about 3 frames per second are needed to approximate motion for the human eye, but a higher frame rate can result in smoother-appearing motion. Typical game engines update animations from 30-60

[1] Traditionally 80x24, but can be much bigger if so desired.

times per second. Refresh rates higher than 60 frames per second can be done, but are not perceptible by human vision.

While text-based graphics may seem limiting, they are not as limiting as one might think. The interested reader might check out some movies that have been written from scratch with text-only, including Star Wars (Asciimation),[2] or videos that have been converted to text, such as the bullet-dodging scene from *The Matrix* movie.[3]

3.3 Saucer Shoot

The tutorial works through creating the *Saucer Shoot* game from scratch. The tutorial is provided "cook-book" style, where the steps that are needed are given explicitly. Development is done in an "agile" fashion, meaning working versions of the game are made throughout the tutorial that can be immediately be tried, debugged as necessary, and played before proceeding to the next version.

An online version of the tutorial can be found at:

`http://dragonfly.wpi.edu/tutorial/`

3.3.1 Startup

Let's get this party started!

Download and setup Dragonfly, appropriate for your development environment.

Create a C++ file called `game.cpp` that will hold your game code. It should contain:

Listing 3.1: Initial game.cpp

```
0  // Engine includes
1  #include "GameManager.h"
2  #include "LogManager.h"
3
4  int main(int argc, char *argv[]) {
5
6    // Start up Game Manager.
7    if (GM.startUp())  {
8      LM.writeLog("Error starting game manager!");
9      GM.shutDown();
10     return 0;
11   }
12
13   // Set flush of logfile during development (when done, make false).
14   LM.setFlush(true);
15
16   // Show splash screen.
17   df::splash();
18
19   // Shut everything down.
20   GM.shutDown();
21 }
```

[2] `http://www.asciimation.co.nz/`

[3] `http://youtu.be/wQhGSu1V9TI`

(It may be easier to download a zip file with the program template, with an accompanying `Makefile` (for Linux) or Visual Studio project file (for Windows), described below, at the game0.zip link below.)

Note, the `df::` tag in front of the Dragonfly-specific code elements. This is to access the Dragonfly namespace. Namespaces provide a mechanism to prevent potential name conflicts in large projects. Typically large, 3rd-party libraries (such as a game engine) will use a namespace to help developers avoid conflicts in names their own code may use with names the libraries use. In the case of Dragonfly, `df::` is needed to access any Dragonfly-specific code element.

The `#include`s at the top of the program are the game engine header files needed so far. Basically, each header file is needed for each game engine service that is used (although some are included by the engine services themselves).

The GameManager (and all game engine managers) are *singletons*, meaning there can be one and only one instance of each for each game. Managers cannot be created via a normal constructor (or assigned or copied), but must be accessed via a `getInstance()` call. After that, the Managers' methods can be accessed normally. For readability and convenience, the one (and only one) instance of each manager can be accessed by a two-letter acronym. Note, in the code above, "GM" is for the GameManager (manager of the game loop) and "LM" is for the LogManager (manager of the logfile).

Here, the game manager starts up (via `startUp()`), enables all other engine services. When the game is done, the game manager shuts down (via `shutDown()`), turning off all game engine services. All Dragonfly games do the same, with the `startUp()` first and the `shutDown()` last.

The LogManager is used by the engine to write to the log file, which is always named "dragonfly.log". During debugging and development, game programs should write messages to the logfile via the LogManager, too. The LogManager supports C's `printf()` notation, providing flexibility in printing out the values of variables. In our code so far, if there is an error in starting up the game manager, this is reported in the log file and the game engine is shut down.

The line `LM.setFlush(true)` makes it so the LogManager flushes all output to disk immediately after a `writeLog()` call. This is useful when developing as that way if a game crashes (e.g., a segfault), the logfile output is still written to disk instead of being lost in memory.

The `splash()` line invokes the Dragonfly splash screen, which includes zooming text and a dragonfly that turns into a 'y'. The splash screen is not strictly needed by any game, but including it here in the first game is useful to help ensure the development environment is setup and working properly.

Dragonfly games need to be compiled the path for includes to the Dragonfly header files and the SFML header files. The Dragonfly library and the SFML libraries (graphics, window, system and audio) need to be linked in. On Linux, the real-time library also needs to be linked for timing.

However, as Saucer Shoot, and nearly all games, will have several `.cpp` files and different compiler flags, it is much easier to `Makefile` (for Linux) or Visual Studio project file (for Windows).

Download a `Makefile` template (if developing in Linux), a Visual Studio Solution (if

developing in Windows), and the game template described above:

http://dragonfly.wpi.edu/tutorial/game0.zip

The Makefile/Solution (will will call this your "project" to be general) may need to be adjusted (e.g., changing the path to wherever SFML is installed on your system), depending upon your particular setup. As Saucer Shoot is developed, file names will be added to the project corresponding to new C++ classes, such as `Saucer.cpp` and `Bullet.cpp`.

Compile your game and try it out!

Once setup, games can be compiled via `make` (if in Linux or Mac) or `Build/F7` (if in Windows). Doing so will ensure all 3rd party libraries are installed (e.g., SFML and Dragonfly) and the Makefile/Project file is setup correctly properly. Once built, games can be run from the shell via `./game` (if in Linux or Mac) or via F5 from Visual Studio (if in Windows). When run, the current game, such as it is, should pop-up a window, play the Dragonfly splash screen, then exit. There should corresponding log messages in the file "dragonfly.log" in the directory where the game was run.

3.3.2 Saucers

Your first party guests

Start by adding an object to the game, in this case a saucer - the main opponent the player will shoot. The saucer has an animated sprite.

You can download the sprite pack for all the sprites used for this tutorial:

http://dragonfly.wpi.edu/tutorial/sprites.zip

Extract the sprites to a directory immediately under the location of the saucer shoot executable (e.g., extract to `game/`, making sure it is named `sprites/`.

For the game code, the ResourceManager is used to load the sprite. Add:

```
0  #include "ResourceManager.h"
```

to the top of the `game.cpp` code with the `#include`s. Since there will be several resources by the time the game is complete, create a function called `loadResources()` that takes in nothing (void) and returns nothing:

```
0  void loadResources(void) {
1  ...
2  }
```

Put a prototype for this function above `main()`. Most games have a lot of resources to load, but for now only the Saucer sprite is loaded. Inside the `loadResources()` function body, put:

Listing 3.2: Example of loading Sprite

```
0  // Load saucer sprite.
1  RM.loadSprite("sprites/saucer-spr.txt", "saucer");
```

The ResourceManager, is used to load and manage the sprites. Note, the ResourceManager was already started up by the GameManager in its `startUp()` call. The ResourceManager is used to load a saucer sprite from the text file named `sprites/saucer-spr.txt`, giving it the label "saucer". The sprite files are text and human-readable. Take a look at `saucer-spr.txt`, if you'd like (using any editor you are comfortable with). The contents are as below:

Listing 3.3: Saucer sprite file

```
<HEADER>
frames 5
width 6
height 2
color green
slowdown 4
</HEADER>
<BODY>
 ____
/____\
end
 ____
/___o\
end
 ____
/__o_\
end
 ____
/_o__\
end
 ____
/o___\
end
</BODY>
<FOOTER>
version 1
</FOOTER>
```

The sprite file is divided into sections: HEADER, BODY and FOOTER. The top five lines in the HEADER provide information on the number of frames, frame width, frame height, sprite color, and animation slowdown, respectively. The subsequent BODY lines provide the frames to be animated, each delimited by a single line with "end". The last FOOTER section has sprite version information. For this tutorial, you will not need to modify or add any sprite files by hand, but can do so upon tutorial completion, if you'd like, using any text editor.

Now, create a new file, `Saucer.h` which will contain the Saucer class definition:

```
#include "Object.h"

class Saucer : public df::Object {

 public:
  Saucer();
};
```

All game objects, including the Saucer, inherit from the Dragonfly Object class. Create a new file called `Saucer.cpp`. It needs to `#include` your `Saucer.h` as well as engine header files of `LogManager.h`, `WorldManager.h`, and `ResourceManager.h`. Define the constructor (`Saucer()`) first. The constructor will look like:

```
Saucer::Saucer() {
...
}
```

The first piece of code handles associating the Saucer Sprite with the Saucer object. The Sprite is associated with the Object via `setSprite()`. Note, by default this also sets the Object's bounding box (used for collisions) to the size of the Sprite and centers the Sprite on the object location. The rate of Sprite animation is specified by the slowdown in the sprite file (see above) - for the Saucer, it's 4, meaning the animation is only advanced once every 4 frame times. Dragonfly's default frame time is 33 milliseconds, which provides a frame rate (and update rate) of 30 frames per second, so a slowdown of 4 advances the animation about once every 130 milliseconds.

```
// Setup "saucer" sprite.
setSprite("saucer");
```

The last block of code sets some object properties. Indicating the type via `setType()` is needed for collisions (for example, to determine what type of object a Bullet hits). Object velocities are set by a Vector that indicates the magnitude of speed horizontally (x) and vertically (y). For the Saucer, setting the x-velocity to a negative value moves the Saucer to the left, while a positive value would move the Saucer to the right (similarly, an Object could set the y-velocity to move up or down, but Saucers do not move vertically). The units for velocity are the number of spaces moved each game step (again, with Dragonfly, one step defaults to 33 milliseconds). Setting the Saucer value to -0.25 means the Saucer moves one space to the left every 4 game steps.

```
// Set object type.
setType("Saucer");

// Set speed in horizontal direction.
setVelocity(df::Vector(-0.25,0)); // 1 space left every 4 frames
```

Lastly, set the starting location to be in the middle of the window. This can be obtained from the WorldManager via `getBoundary()`. This returns a Box, which has horizontal and vertical components (see

http://dragonfly.wpi.edu/include/classdf_1_1Box.html

for details). In general, the game world can be a different size (larger or smaller) than the terminal window, but for Saucer Shoot the game world and the terminal window are the same size.

Listing 3.4: Saucer starting in middle of window.

```
// Set starting location in the middle of window.
int world_horiz = (int) WM.getBoundary().getHorizontal();
int world_vert = (int) WM.getBoundary().getVertical();
```

```
3   df::Vector p(world_horiz/2, world_vert/2);
4   setPosition(p);
```

The "WM" acronym is for the WorldManager, which manages the objects in the game world.

Compile your program to make sure you have no errors. To do so, be sure to add `Saucer.cpp` to your project. Running your game will not show any new output, however, since although you have finished defining a basic Saucer, you have not added it to the game world.

Since you generally populate a game with a lot of objects, create a method in `game.cpp` called `populateWorld()` that returns `void` and takes in nothing along with a prototype.

```
0   void populateWorld(void) {
1   ...
2   }
```

Add a `#include` to `Saucer.h` since you will be adding your Saucer as an object. For the method body, for now simply create a Saucer via a `new` call. Note, you do not need to grab the value returned by `new`, which may look odd, because in the constructor for an Object (the parent class of a Saucer), it automatically adds itself to the game world via the WorldManager `insertObject()` call.

```
0   new Saucer;
```

In the body of `main()` after the GameManager has started up, first add a call to `loadResources()`, then to `populateWorld()`.

```
0   // Load game resources.
1   loadResources();
2
3   // Populate game world with some objects.
4   populateWorld();
```

Now, you have your resources loaded and your world populated, so you are ready to run the game from within `main()` in your game code. You do this by calling the `run()` method from the GameManager.

```
0   GM.run();
```

The `run()` call blocks until `setGameOver()` is called in the GameManager. Since your game does not yet do this, you will have to terminate it by hand.

Compile your game and try it out!

You should see a saucer start centered in the middle of the window and move off to the left. If you got stuck with functionality or compiler errors, you can obtain a zipped version of the source code:

http://dragonfly.wpi.edu/tutorial/game1.zip

Having a single saucer fly off and never be seen again is not what the player might expect. Instead, when it leaves the left side of the window, make it re-appear on the right side, as if it is a new Saucer. To do this, the Saucer will respond to the "outofbounds"

event that Dragonfly generates when an game object moves from inside the game world to outside (in this case, when the game object moves off the left edge of the window). To the Saucer class definition (in `Saucer.h`), add a prototype for the event handler.

```
int eventHandler(const df::Event *p_e);
```

Add `EventOut.h` to the list of includes in `Saucer.cpp`. Define the event handler next, named `eventHandler()`. The `eventHandler()` method is inherited from the parent Object class and gets invoked with every event the game world passes to the Object. The event type is returned by `getType()`, a method in the Event class. The Saucer at this point will only handle the "out" event. When the event is an `OUT_EVENT`, the Saucer `out()` method is called. If the event is something else, it is ignored by the Saucer.

```
int Saucer::eventHandler(const df::Event *p_e) {

  if (p_e->getType() == df::OUT_EVENT) {
    out();
    return 1;
  }

  return 0;
}
```

Then, define the `out()` method:

```
void Saucer::out() {
..
}
```

First, in the method body, if the Saucer is in this method not as a result of moving off the edge of the window (instead, say, by spawning out of bounds) it does not want to do anything. Check this by getting the position (a Vector) via `getPosition()` and then looking at the x value via `getX()`. If it is greater than 0, the Saucer did not go out of bounds on the left side of the window.

```
if (getPosition().getX() >= 0)
  return;
```

Otherwise, you want to move the Saucer back to the right edge of the game world. Move it just beyond the window edge and it will look like it is a new Saucer flying into view. Since this functionality will get invoked in more than one place (it gets called whenever a Saucer spawns, too) create a method in `Saucer.cpp` called `moveToStart()`.

```
void Saucer::moveToStart() {
...
}
```

Put the method prototype in the class definition in `Saucer.h`. In the method body, move the saucer by creating a Vector object, and setting the x and y values to be randomly placed past the right of the window. In order to invoke the `rand()` function calls, put an include to `<stdlib.h>` at the top of `Saucer.cpp`.

```
df::Vector temp_pos;
```

```
1
2   float world_horiz = WM.getBoundary().getHorizontal();
3   float world_vert = WM.getBoundary().getVertical();
4
5   // x is off right side of window
6   temp_pos.setX(world_horiz + rand() % (int) world_horiz + 3.0f);
7
8   // y is in vertical range
9   temp_pos.setY(rand() % (int) (world_vert-1) + 1.0f);
```

Now that `temp_pos` is a position to the right of the visible window, tell the WorldManager to actually move the Saucer there.

```
0   WM.moveObject(this, temp_pos);
```

Since this code starts the Saucer in a random location, replace the code in the Saucer constructor that places the Saucer in the middle of the window with a call to `moveToStart()`.

Compile your game and try it out!

You should see the saucer start past the right edge of the window (it may take a moment to appear, depending upon the location), moving on to the window as it moves right to left, whereupon after leaving the left edge of the window it re-appears a short time later back on the right.

If you get stuck adding your own code, you can download a zipped version of this game up to this point:

http://dragonfly.wpi.edu/tutorial/game2.zip

3.3.3 Hero

Here to save the day!

Games are all about interaction, so add a game object the player can control. Make a class definition called Hero inside of `Hero.h`. Like all game objects, Class Hero will inherit from the Object class. It is going to respond to "keyboard" events, which are generated by the InputManager, since the player will control the Hero via the keyboard. So, `Hero.h` will need to include `Object.h` and `EventKeyboard.h`.

```
0   class Hero : public df::Object {
1
2    private:
3     void kbd(const df::EventKeyboard *p_keyboard_event);
4     void move(int dy);
5
6    public:
7     Hero();
8     int eventHandler(const df::Event *p_e);
9   };
```

In `Hero.cpp`, add an include to `Hero.h` as well as includes for `LogManager.h`, `WorldManager.h`, and `ResourceManager.h` - these latter Managers will be used shortly. Define the Hero constructor:

```
Hero::Hero() {
..
}
```

First, write code to associate the "ship" sprite with the Hero. This should look familiar after the Saucer constructor code you wrote above.

```
// Link to "ship" sprite.
setSprite("ship");
```

Each Object needs to register for the events it is interested in via `registerInterest()`. At this point, the Hero is interested in "keyboard" events.

```
registerInterest(df::KEYBOARD_EVENT);
```

Set the Object type and the Hero location and the constructor is complete. For location, put the Hero on the left edge of the window, mid-way down vertically.

```
setType("Hero");
df::Vector p(7, WM.getBoundary().getVertical()/2);
setPosition(p);
```

The Hero `eventHandler()` method is going to respond to only the "keyboard" event at this point. To do so, it casts a generic Event object pointer as an EventKeyboard object pointer, then calls the Hero `kbd()` method.

```
int Hero::eventHandler(const df::Event *p_e) {
  if (p_e->getType() == df::KEYBOARD_EVENT) {
    const df::EventKeyboard *p_keyboard_event =
              dynamic_cast <const df::EventKeyboard *> (p_e);
    kbd(p_keyboard_event);
    return 1;
  }
  return 0;
}
```

Next, write the Hero `kbd()` method. It will inspect the key that is pressed, obtained via the `getKey()` method from the EventKeyboard, and act accordingly.

```
// Take appropriate action according to key pressed.
void Hero::kbd(const df::EventKeyboard *p_keyboard_event) {

  switch(p_keyboard_event->getKey()) {

    ...

  }
}
```

First off, during the game, and during development, it is useful for the player to hit 'Q' to exit. Do this by adding code to the Hero `kdb()` method to tell the game to via the `setGameOver()` method.

```
case df::Keyboard::Q:     // quit
```

```
    if (p_keyboard_event->getKeyboardAction() == df::KEY_PRESSED)
      GM.setGameOver();
    break;
```

If the key pressed is a 'W' or 'S' *and* the key is still being pressed down, the Hero will move up or down, as appropriate.

```
  case df::Keyboard::W:     // up
    if (p_keyboard_event->getKeyboardAction() == df::KEY_DOWN)
      move(-1);
    break;
  case df::Keyboard::S:     // down
    if (p_keyboard_event->getKeyboardAction() == df::KEY_DOWN)
      move(+1);
    break;
```

And other keys and other actions (such as a key release) are ignored at this point.

With the movement keys in place, you need to write the Hero `move()` method. It basically creates a new Vector object, setting the location either 1 up or 1 down from the Hero's current position. If the desired move keeps the Hero completely inside the window, a call to `moveObject()` in the WorldManager actually moves it.

```
// Move up or down.
void Hero::move(int dy) {

  // If stays on window, allow move.
  df::Vector new_pos(getPosition().getX(), getPosition().getY() + dy);
  if ((new_pos.getY() > 3) &&
      (new_pos.getY() < WM.getBoundary().getVertical()-1))
    WM.moveObject(this, new_pos);
}
```

The above code will work fine to move the Hero up and down. However, since it moves the Hero as long as either the 'W' or 'S' key is pressed, this means the Hero will move every game loop ("step"), so up to 30 spaces per second - too fast for this game. You can use a common "countdown" technique to limit how often a player can move the Hero. Declare two private integer variables in `Hero.h` called `move_slowdown` and `move_countdown` and initialize them in the Hero constructor:

```
  move_slowdown = 2;
  move_countdown = move_slowdown;
```

The countdown variables are used in `move()` to limit the rate of movement by not doing anything unless the Hero has counted down to 0 since the last move. To do this, at the top of `move()` add:

```
// See if time to move.
if (move_countdown > 0)
  return;
move_countdown = move_slowdown;
```

The `move_countdown` variable gets decreased every game step. Include `EventStep.h` in the `Hero.cpp` header section. Modify the Hero constructor to register for "step" events (`STEP_EVENT`) and update the `eventHandler()` to call a new Hero method, `step()`.

Listing 3.5: Hero step()

```
// Decrease rate restriction counters.
void Hero::step() {
  // Move countdown.
  move_countdown--;
  if (move_countdown < 0)
    move_countdown = 0;
}
```

Now, add code to your game.cpp to put a Hero into the world. Add Hero.h to the list of includes. In loadResources(), add ship-spr.txt with a label of "ship" to be loaded by the ResourceManager. And to populateWorld(), add new Hero to instantiate a Hero object. Make sure to add Hero.cpp to your project so it will be compiled.

Compile your game and try it out!

In addition to the Saucer functionality, you should see a Hero ship spawn on the window and should be able to move it up and down via the 'W' and 'S' keys. Pressing 'Q' should quit and end the game.

If you need it, this version of the game can be downloaded:

http://dragonfly.wpi.edu/tutorial/game3.zip

3.3.4 Bullets

Let the action begin!

To make the "shoot" part of "Saucer Shoot", you will add the ability for the player's Hero ship to shoot bullets. First, make a Bullet class, with the ability to respond to "outofbounds" events, for moving, and to "collision" events for hitting Saucers. The #includes need to be "Object" and "EventCollision" in the Bullet.h file.

```
class Bullet : public df::Object {

  private:
    void out();
    void hit(const df::EventCollision *p_collision_event);

  public:
    Bullet(df::Vector hero_pos);
    int eventHandler(const df::Event *p_e);
};
```

In Bullet.cpp, define the constructor similar to the Saucer and Hero constructors. The Bullet constructor should link to the sprite "bullet", and set the object type (setType()) to "bullet". A major difference in the Bullet constructor is it takes in the position of the Hero (a Vector) as an argument and uses it for the Bullet's initial location - this makes sense since the Hero is the one that fired the bullet. In the Bullet constructor, the Bullet sets its location just to the right of the Hero's location.

```
// Set starting location, based on hero's position passed in.
df::Vector p(hero_pos.getX()+3, hero_pos.getY());
```

```
  setPosition(p);
```

Also in the constructor, set the bullet speed. Bullets are relatively fast, moving 1 space every game loop (so, 30 spaces per second). Unlike for Saucers that always move to the left, the Hero sets each each Bullet's direction when it fires to move towards the mouse reticle.

```
  // Bullets move 1 space each game loop.
  // The direction is set when the Hero fires.
  setSpeed(1);
```

The Bullet `eventHandler()` should look like the Saucer's, handling an `OUT_EVENT`. In addition, the Bullet will want to react when it hits a Saucer by handling the "collision" event.

```
  if (p_e->getType() == df::COLLISION_EVENT) {
    const df::EventCollision *p_collision_event =
                dynamic_cast <const df::EventCollision *> (p_e);
    hit(p_collision_event);
    return 1;
  }
```

Like all Dragonfly `eventHandler()` methods, it should return 1 when the event is handled (i.e., "collision" and "outofbounds" events) and return 0 when not.

With the above code in place, you next need to write methods for `out()`, and `hit()`.

For `out()`, since this method is called when the Bullet leaves the window (moves off the right edge), you want to destroy the Bullet. In Dragonfly, this is done by calling `markForDelete()` in the WorldManager. All Objects that are scheduled for deletion in the course of one iteration of the game loop are deleted at the same time.

```
  // If Bullet moves outside world, mark self for deletion.
  void Bullet::out() {
    WM.markForDelete(this);
  }
```

For `hit()`, if the Bullet has hit a Saucer it marks itself and the Saucer for deletion. The two Objects involved in a collision can be obtained from the methods `getObject1()` and `getObject2()` in the EventCollision class. The Object that moved that initiated the collision is always object 1.

```
  // If Bullet hits Saucer, mark Saucer and Bullet for deletion.
  void Bullet::hit(const df::EventCollision *p_collision_event) {
    if ((p_collision_event -> getObject1() -> getType() == "Saucer") ||
      (p_collision_event -> getObject2() -> getType() == "Saucer")) {
      WM.markForDelete(p_collision_event->getObject1());
      WM.markForDelete(p_collision_event->getObject2());
    }
  }
```

Now that you have defined a fully-functional working Bullet object, you need to add capability for the Hero object to fire bullets. First, it needs to include the new `Bullet.h` class definition. For the Hero constructor, you will want to limit the fire rate of the player (so s/he cannot just spam bullets) using a technique similar to controlling movement speed.

Declare two private integer variables in `Hero.h` called `fire_slowdown` and `fire_countdown` and initialize them in the Hero constructor:

```
fire_slowdown = 15;
fire_countdown = fire_slowdown;
```

The player will be able to aim the bullet with the mouse, so in the Hero constructor, register for interest in mouse events (`MOUSE_EVENT`), similar to registering for interest in keyboard events.

Next, define a new method related to firing - `fire()` that creates a Bullet object. The player will be able to aim the bullet with the mouse, so the `fire()` method will take in a target position (a Vector) as input.

```
void Hero::fire(df::Vector target) {
  ...
}
```

The Hero uses the countdown and slowdown variables to limit the rate of fire by not creating a new Bullet unless the Hero has counted down to 0 since the last time a Bullet was fired.

```
if (fire_countdown > 0)
  return;
fire_countdown = fire_slowdown;
```

In the Hero `step()` method, decrease the `fire_countdown` variable every game step.

```
// NOTE - in step()
// Fire countdown.
fire_countdown--;
if (fire_countdown < 0)
  fire_countdown = 0;
```

Back in the `fire()` method, when actually firing, a Bullet is created via `new`, passing in the position of the Hero (a Vector). Then, the target position passed in to `fire()` is used to adjust the y-velocity to move the bullet vertically towards the target.

```
// Fire Bullet towards target.
// Compute normalized vector to position, then scale by speed (1).
df::Vector v = target - getPosition();
v.normalize();
v.scale(1);
Bullet *p = new Bullet(getPosition());
p->setVelocity(v);
```

One subtle code addition is to make the Bullets SOFT in the Bullet constructor to allow them to pass through the Hero if firing backwards.

```
// Note: in Bullet constructor
// Make the Bullets soft so can pass through Hero.
setSolidness(df::SOFT);
```

The `fire()` method is finished, but needs to be invoked when the player wants to fire. This is done by clicking the left mouse button with the mouse cursor where the player wants

to fire the bullet. To do this, a mouse event must be handled in the Hero's `eventHandler()` method.

```
if (p_e->getType() == df::MSE_EVENT) {
  const df::EventMouse *p_mouse_event =
            dynamic_cast <const df::EventMouse *> (p_e);
  mouse(p_mouse_event);
  return 1;
}
```

If there is a mouse event, this invokes the `mouse()` method you will define next.

```
// Take appropriate action according to mouse action.
void Hero::mouse(const df::EventMouse *p_mouse_event) {
  ...
}
```

The body of `mouse()` examines the mouse event first, to see if a button was clicked, and if so, second, to see if the button is the left mouse button. If so, the `fire()` is invoked, passing in the (x,y) position of the mouse.

```
// Pressed button?
if ((p_mouse_event->getMouseAction() == df::CLICKED) &&
    (p_mouse_event->getMouseButton() == df::Mouse::LEFT))
  fire(p_mouse_event->getMousePosition());
```

Lastly, in `game.cpp`, make sure to modify `loadResources()` to have the resource manager load the sprite `bullet-spr.txt` with a label of "bullet".

Make sure to add `Bullet.cpp` to your project so it will be compiled.

Compile your game and try it out!

You should be able to click the mouse and have your ship shoot bullets. (Note, you will not actually see the mouse cursor - we will fix that next). If the bullets strike the Saucer, it should destroy both of them and they will disappear.

The version of the game up to this point can be downloaded:

http://dragonfly.wpi.edu/tutorial/game4.zip

3.3.5 Reticle

Ready! Fire! Aim!

Playing the game as-is will find that shooting using the mouse is difficult when you cannot see the mouse cursor. We will fix this by creating a sight (a reticle) that shows the mouse in the Dragonfly window. Define a Reticle class in `Reticle.h`. It will need to include `Object.h`.

```
#define RETICLE_CHAR '+'

class Reticle : public df::Object {

 public:
  Reticle();
```

```
    int draw(void);
    int eventHandler(const df::Event *p_e);
};
```

In Reticle.cpp, add includes for EventMouse.h, DisplayManager.h and WorldManager.h and, of course, Reticle.h. The, create a constructor that will setup the initial Reticle attributes.

```
Reticle::Reticle() {
  ...
}
```

In the constructor, set the Object type to "Reticle".

Unlike a default Object, the Reticle should not collide with other Objects. You can make this happen via the solidness attribute of an Object. Objects can be HARD, SOFT or SPECTRAL. Hard objects will generate a collision and impede movement. Soft objects will generate a collision, but not impede movement. Spectral objects do not generate collisions nor impede movement. So, the Reticle should be SPECTRAL.

```
setSolidness(df::SPECTRAL);
```

The Reticle should always be drawn on top in the foreground so as not to be "hidden" behind Saucers or other Objects. This can be done with Dragonfly layering. While the world view is only 2D, Dragonfly supports 5 visual layers, where the bottom layers are drawn first followed by the top layers. The altitude variable controls the visual layering. Values can range from 0 to 4 (MAX_ALTITUDE) with the lowest altitudes drawn first. The default altitude for all Objects is 2. Note, for purposes of collisions, there is only 1 layer - the values discussed here are only for display.

To always be drawn in the foreground, the Reticle should have the maximum altitude.

```
setAltitude(df::MAX_ALTITUDE); // Make Reticle in foreground.
```

Like the Hero, the Reticle should also register for interest in the mouse event since it is going to move whenever the mouse moves.

Lastly, the Reticle should start centered in the middle of the window, as the Saucer did initially. (See Listing 3.4).

The Reticle eventHandler() responds to mouse events, with all other events being ignored. If there is a mouse event and the event is that the mouse has moved, the Reticle position is changed to the mouse's new location.

```
int Reticle::eventHandler(const df::Event *p_e) {

  if (p_e->getType() == df::MSE_EVENT) {
    const df::EventMouse *p_mouse_event =
              dynamic_cast <const df::EventMouse *> (p_e);
    if (p_mouse_event->getMouseAction() == df::MOVED) {
      // Change location to new mouse position.
      setPosition(p_mouse_event->getMousePosition());
      return 1;
    }
  }

```

```
    // If get here, have ignored this event.
    return 0;
  }
```

Unlike other Objects, the Reticle does not use a Sprite. Instead, it overrides the `draw()` method in the Object parent class and draw itself as a single character, defined as `RETICLE_CHAR`.

```
  // Draw reticle on window.
  void Reticle::draw() {
    return DM.drawCh(getPosition(), RETICLE_CHAR, df::RED);
  }
```

The "DM" acronym is for the DisplayManager, which manages display to the graphics device (e.g., the screen).

Lastly, since the Hero uses the Reticle to aim, it makes sense to have the Hero create the Reticle when the Hero first spawns. Add an attribute to the Hero class for holding the Reticle.

```
private:
  Reticle *p_reticle;
```

Then, in the Hero constructor, add code to create a new Reticle.

```
  // Create reticle for firing bullets.
  p_reticle = new Reticle();
  p_reticle->draw();
```

Make sure to add `Reticle.cpp` to your project so it will be compiled, and include the `Reticle.h` header file, where needed.

Compile your game and try it out!

The game functionality will be as before, but you should now see the mouse in the form of the Reticle (a red '+') that moves where the mouse moves.

The version of the game up to this point can be downloaded:

http://dragonfly.wpi.edu/tutorial/game5.zip

3.3.6 Explosions

Saucer, meet Bullet

You may notice the Saucer disappears without so much as a whimper. You can fix that by making an explosion object that gets created whenever a Saucer is destroyed. Define an Explosion class in `Explosion.h`. Unlike other game objects, this one will live for a finite amount of time then destroy itself. To do this, it will have a "time to live" counter that gets decremented each game step.

```
  class Explosion : public df::Object {

   private:
    int time_to_live;
```

```
    void step();

  public:
    Explosion();
    int eventHandler(const df::Event *p_e);
};
```

In Explosion.cpp, create a constructor that links to a "explosion" sprite. In addition, the time_to_live variable needs to be set. It should be set to provide a countdown long enough for the Explosion animation to play, which is equivalent to the number of frames in the sprite. This can be obtained from the Animation and Sprite objects via getFrameCount(). Note, the below code also illustrates error checking the setSprite() method, which returns -1 if the indicated sprite label is not found.

```
// Link to "explosion" sprite.
if (setSprite("explosion") == 0)
  // Set live time as long as sprite length.
  time_to_live = getAnimation().getSprite()->getFrameCount();
else
  time_to_live = 0;
```

Like the Reticle, Explosions should be SPECTRAL. Also, have the Explosion register for a "step" event (df::STEP_EVENT) in the constructor.

The Explosion eventHandler() should call the step() method when it gets a "step" event. In step(), the Explosion decrements the time_to_live variable. When this reaches 0, it will mark itself for deletion with the WorldManager.

```
void Explosion::step() {
  time_to_live--;
  if (time_to_live <= 0)
    WM.markForDelete(this);
}
```

An Explosion is created when a Saucer is destroyed. In the Saucer's event handler, add the same code that responds to a collision event as you have in the Bullet class, calling a hit() method in the Saucer. Put a prototype of the hit() method in Saucer.h along with a #include to the EventCollision.h header file. Next, define the Saucer's hit method:

```
void Saucer::hit(const df::EventCollision *p_c) {
...
}
```

In the body, there are a couple of scenarios to consider. First, if a Saucer runs into another Saucer, that should be ignored.

```
// If Saucer on Saucer, ignore.
if ((p_c -> getObject1() -> getType() == "Saucer") &&
    (p_c -> getObject2() -> getType() == "Saucer"))
  return;
```

If a Saucer runs into a Bullet, however, there should be fireworks! Namely, you want to create an Explosion by using new and then setting the Explosion position to the Saucer's position. Remember, the Bullet collision handler will already mark the Saucer for deletion.

In order to continually give the player Saucer's to shoot at, have the Saucer spawn a new Saucer when it is hit by the Bullet.

```
// If Bullet...
if ((p_c -> getObject1() -> getType() == "Bullet") ||
    (p_c -> getObject2() -> getType() == "Bullet")) {

  // Create an explosion.
  Explosion *p_explosion = new Explosion;
  p_explosion -> setPosition(this -> getPosition());

  // Saucers appear stay around perpetually.
  new Saucer;
}
```

Make sure you have the ResourceManager load `sprites/explosion-spr.txt` with label "explosion" in `game.cpp`.

Compile your game and try it out!

You should be able to shoot Saucers, whereupon hitting one you get a nice, if brief, explosion animation. The Saucer should respawn, too, giving you infinite target practice. If you are stuck, the version of the game up to this point can be downloaded:

http://dragonfly.wpi.edu/tutorial/game6.zip

3.3.7 End of Game

All good things must come to an end

You have most of the functionality needed for gameplay at this point, but the game aspects could use a bit of work.

You'll want to spawn more Saucers at the beginning of the game, say 16. In `game.cpp`, in `populateWorld()`, add a loop creating Saucers.

```
// Spawn some saucers to shoot.
for (int i=0; i<16; i++)
  new Saucer;
```

Once there are a lot of Saucers spawning in random locations, they may end up landing on top of one another. In such a case, Dragonfly generates a "collision" event, thereby causing the Saucers to move back their original location (all Objects spawn at (0,0), by default). You need to add some code to the Saucer `moveToStart()` method to prevent this. The idea is to check if there is a collision at the Saucer's randomly chosen location. If so, the Saucer is inched over to the right and the location checked again, repeating until a free location is obtained.

```
// If collision, move right slightly until empty space.
df::ObjectList collision_list = WM.getCollisions(this, temp_pos);
while (!collision_list.isEmpty()) {
  temp_pos.setX(temp_pos.getX()+1);
  collision_list = WM.getCollisions(this, temp_pos);
}
```

You will also want a way to make the game get more difficult for the player as time progresses. While there are many ways to do this, adding a "new Saucer" call to the end of the Saucer `out()` method works pretty well for just increasing the number of enemies as the player lets them get by. You can do this by creating a new one each time a Saucer moves past the Hero on the left edge of the window. At the very end of the Saucer `out()` method, add code to make a new Saucer.

```
0  // Spawn new Saucer to make the game get harder.
1  new Saucer;
```

You will want to add an end-of-game condition next. This should happen when the Hero collides with a Saucer or vice versa. On such a collision, a "collision" event is sent to both objects. Handle it in the Saucer in the `hit()` method. The Saucer will check if either Object involved in the collision is the Hero. If so, it schedules both Objects for deletion.

```
0  // If Hero, mark both objects for destruction.
1  if (((p_collision_event -> getObject1() -> getType()) == "Hero") ||
2      ((p_collision_event -> getObject2() -> getType()) == "Hero")) {
3    WM.markForDelete(p_collision_event -> getObject1());
4    WM.markForDelete(p_collision_event -> getObject2());
5  }
```

In order to make the collision with the Hero end the game, the GameManager needs to be told to end the game loop via the `setGameOver()` method. This code should be put in the Hero destructor (which is not defined yet, so it will need a method definition in the Hero class in `Hero.h`, first).

```
0  GM.setGameOver();
```

Compile your game and try it out!

You should now have lots of Saucers to shoot, and they should keep coming in increasing numbers as they slip by. If the Hero is hit by a Saucer the game should end.

The version of the game up to this point can be downloaded:

http://dragonfly.wpi.edu/tutorial/game7.zip

3.3.8 Nukes

Desperate times call for desperate measures

As you may find, the game can get pretty hard once a few Saucers slip by. To illustrate a user-defined event (and to give the player a fighting chance) you are going to add the capability for a single "nuke" event usable by the player. Launching the nuke will destroy all the Saucers currently in the game (of course, we'll make more since each will spawn another Saucer when destroyed). Still, the player will have a momentary respite from the onslaught.

Dragonfly provides standard events, such as "step," "collision" and "outofbounds". However, it also provides the mechanism to let game programmers define their own, game-specific events. You will use this for creating a "nuke" event.

To do so, create a class called EventNuke (in EventNuke.h) that inherits from the Event class with a string constant NUKE_EVENT string to "nuke".

```
const std::string NUKE_EVENT = "nuke";

class EventNuke : public df::Event {

 public:
  EventNuke();
};
```

It will need to include Event.h.

In EventNuke.cpp, only the constructor needs to be defined, setting the type of the Event to NUKE_EVENT.

```
EventNuke::EventNuke() {
  setType(NUKE_EVENT);
};
```

Add EvenNuke.cpp to your project so it will be compiled. Although simple, compile it and make sure it builds without error.

The Nuke event is triggered when the player presses the spacebar. Since the Hero already handles keyboard events, extend the Hero class to support nukes. First off, add EventNuke.h to the includes for Hero.cpp. Then, modify the Hero kdb() method to detect the spacebar and invoke the nuke() method.

```
void Hero::nuke() {
...
}
```

In the body of nuke(), first check if the player has any nukes left. Do this by keeping a nuke_count variable for the Hero object and see if it is greater than zero. If not, there is nothing to be done, so return. If so, decrement nuke_count and proceed. Note, nuke_count should be declared as a private integer in Hero.h and initialize it to 1 in the Hero constructor.

```
// Check if nukes left.
if (!nuke_count)
  return;
nuke_count--;
```

If the nuke is allowed, the Hero creates a "nuke" event and sends it to every Object that has registered for interest in it in the WorldManager. This is done by calling the WorldManager onEvent() method, passing in the address of the EventNuke.

```
// Create "nuke" event and send to interested Objects.
EventNuke nuke;
WM.onEvent(&nuke);
```

The Saucer objects are the ones affected by the Nuke. Add EventNuke.h to the Saucer.cpp's included headers. The constructor for the Saucer needs to register for interest in a NUKE_EVENT. Then, in the Saucer eventHandler(), add a check for event type NUKE_EVENT. If found, the Saucer creates an Explosion (as it does when it hits a Bullet), marks itself for deletion with the WorldManager, and spawns another Saucer.

```
if (p_e->getType() == NUKE_EVENT) {
...
}
```

Compile your game and try it out!

The game should play as before, but the player should be allowed to hit the spacebar and destroy all the Saucer's in the game. They will all respawn, of course, and the player can only do this "nuke" action one time, but it should temporarily clear the window of all baddies.

As usual, the version of the game up to this point can be downloaded:

http://dragonfly.wpi.edu/tutorial/game8.zip

3.3.9 Game On!

It doesn't matter if you win or lose ... until you lose

Your game should have some nice interaction going, but it isn't really a game. Why not? Because there is no obvious score, no real incentive to live longer or shoot more Saucers. So, you will next add some points to your game and display them for the player.

Dragonfly has a ViewObject that is a special type of Object used for displaying values, such as "score" or "lives". You will use this for the game score. Create a new object of type Points (in `Points.h`) that inherits from ViewObject.

```
#define POINTS_STRING "Points"

class Points : public df::ViewObject {

 public:
  Points();
};
```

It will need to `#include ViewObject.h` and `Event.h`.

In `Points.cpp`, create a constructor.

```
Points::Points() {
...
}
```

Set some attributes of the base ViewObject to display the score in the top right of the window, with the color yellow.

```
setLocation(df::TOP_RIGHT);
setViewString(POINTS_STRING);
setColor(df::YELLOW);
```

The game will reward the player with 1 point for every second survived, keeping track of this by checking the step count (number of game loop iterations). To do this, register to receive the "step" event from the GameManager in the constructor.

```
// Need to update score each second, so count "step" events.
registerInterest(df::STEP_EVENT);
```

Next, add a public method in Points.h for handling events.

```
int eventHandler(const df::Event *p_e);
```

In Points.cpp, define the event handler method.

```
int Points::eventHandler(Event *p_e) {
  ...
  // If get here, have ignored this event.
  return 0;
}
```

In the method body, at the top, add a call to the parent class (the ViewObject) eventHandler() when there is a score update.

```
// Parent handles event if score update.
if (df::ViewObject::eventHandler(p_e)) {
  return 1;
}
```

If this is not a score update, then if a "step" event arrives, Points checks if it is evenly divisible by 30 - if so, one second has elapsed and the player has earned a point! The points are stored in the base ViewObject, accessed by getValue() and setValue().

```
// If step, increment score every second (30 steps).
if (p_e->getType() == df::STEP_EVENT) {
  if (dynamic_cast <const df::EventStep *> (p_e)
                    -> getStepCount() % 30 == 0)
    setValue(getValue() + 1);
  return 1;
}
```

Next, the game should reward the player for shooting Saucer's, too. To do this, make use of Dragonfly's EventView objects, which are Events that ViewObjects are automatically registered to receive. Create a Saucer destructor, defined in the .h file and the body in the .cpp file. Add code to create an EventView and send it to all interested objects.

```
// Send "view" event with points to interested ViewObjects.
// Add 10 points.
df::EventView ev(POINTS_STRING, 10, true);
WM.onEvent(&ev);
```

The parameters "10" and "true" tell the Points object that it should add 10 to its value when it handles the events. To compile, you need to add EventView.h to the list of includes in Saucer.cpp. In order to recognize POINTS_STRING, you also need to include Points.h.

Lastly, you want to create the Points object at the beginning of the game. In game.cpp, after the code for spawning a Hero, add code to spawn a Points object.

```
// Setup heads-up display.
new Points;   // points display
```

You need `#include Points.h` at the top of `game.cpp`. You can compile your game and try it out. You should see the player Points displayed at the top center of window, with rewards given for time alive (1 point per second) and destroying enemies (10 points per saucer).

You can use the same ViewObject mechanism to keep track of the number of nukes the player has, too. Since this display does not need to define any new behaviors, unlike the Points object, you can use a standard ViewObject and define some attributes without creating a separate class. In `game.cpp`, right after creating the Points object, add code to create another ViewObject for the Nukes display.

```
df::ViewObject *p_vo = new df::ViewObject; // Count of nukes.
```

After creation, set the attributes to display it in the top left of the window, with the display string of "Nukes", an initial value of 1, and drawn in yellow.

```
p_vo->setLocation(df::TOP_LEFT);
p_vo->setViewString("Nukes");
p_vo->setValue(1);
p_vo->setColor(df::YELLOW);
```

You need to include `Color.h` in order to recognize `df::YELLOW`. In the Hero, at the end of the `nuke()` method, add code to send an EventView to the display, decrementing the nuke value.

```
// Send "view" event with nukes to interested ViewObjects.
EventView ev("Nukes", -1, true);
WM.onEvent(&ev);
```

Compile your game and try it out!

You should see a display for points and nukes, with all Objects traveling behind them. The version of the game up to this point can be downloaded:

http://dragonfly.wpi.edu/tutorial/game9.zip

3.3.10 Stars

Billions and billions of 'em

Your game is nearly there, but a few tweaks will make it look more refined. One visual problem is that the window background is rather plain. Saucer Shoot is supposed to be in space, but there are no stars!

To make some stars, you utilize the visual layers that placed the Reticle in front, but this time making the stars behind the scenes as a backdrop. Create a Star class in `Star.h` that inherits from Object. It will have event handling the "outofbounds" events, so a corresponding `out()` method.

Like the Reticle, the star overrides the `draw()` method and draws itself as a single character, defined as `STAR_CHAR`.

```
#define STAR_CHAR '.'

```

```
class Star : public df::Object {

 private:
  void out();

 public:
  Star();
  int draw(void);
  int eventHandler(const df::Event *p_e);
};
```

Star.cpp needs to include Star.h, and <stdlib.h> for random placement. It also needs WorldManager.h and EventOut.h.

Create the Star constructor.

```
Star::Star() {
...
}
```

The Star constructor should set the type to "Star" and set itself to be SPECTRAL.

```
setType("Star");
setSolidness(df::SPECTRAL);
```

In movies, as a spaceship travels through space stars closer to the camera appear to move faster than stars further away. To achieve this motion parallax affect, set the velocity to a random value, one of 10 different speeds.

```
setVelocity((float) df::Vector((float) -1.0 /(rand()%10 + 1), 0));
```

Since we want Stars in the background, give them an altitude of 0.

```
setAltitude(0);        // Make Stars in background.
```

Lastly, set the position of the Star to be randomly chosen over the whole window.

```
df::Vector p((float) (rand()%(int)WM.getBoundary().getHorizontal()),
             (float) (rand()%(int)WM.getBoundary().getVertical()));
setPosition(p);
```

When the draw() method for Stars are called (once each game loop done by the game manager), the Star will invoke the drawCh() method of the DisplayManager, giving it the position of the Star and the character to be drawn.

Listing 3.6: Star draw()

```
void Star::draw() {
  return DM.drawCh(getPosition(), STAR_CHAR, df::WHITE);
}
```

The Star eventHandler() should be done as for Saucers, only handling the "out" event via the Star out() method. The Star out() method should move the Star back to a random vertical location on the right of the window and also randomize the velocity, as it already does in the constructor.

```
// If Star moved off window, move back to far right.
void Star::out() {
  df::Vector p((float) (WM.getBoundary().getHorizontal() + rand()%20),
               (float) (rand()%(int)WM.getBoundary().getVertical()));
  setPosition(p);
  setVelocity((float) df::Vector(-1.0 /(rand()%10 + 1), 0));
}
```

All the Stars should be created right at the beginning of the game. In `game.cpp`, add `Star.h` to the list of includes. In `populateWorld()`, add a loop creating 16 Stars.

```
// Create some Stars.
for (int i=0; i<16; i++)
  new Star;
```

Compile your game and try it out!

You should see many stars moving at different speeds in the backdrop of your Saucer Shoot game.

The version of the game up to this point can be downloaded:

http://dragonfly.wpi.edu/tutorial/game10.zip

3.3.11 Polish

Along with spit, this will make the game shine

You are nearly done – two more objects will make the whole game a little more game-like. First, the end of game event is rather abrupt. Rather than have the end come so quickly, it would be better to display a message and let the player collect his/her breath before terminating. Likewise, it would be better if there was a pause before the start of the game, where the game name and instructions could be shown.

To achieve these effects, you will create a new object called a GameOver. It will behave much like an Explosion in that stays around only until its Sprite animation is complete. However, rather than being part of the game itself, GameOver will be a ViewObject like "Points" and "Nukes".

Declare `GameOver.h` and define the GameOver class to be like the Explosion class, except make it derived from a ViewObject instead of a Object.

```
class GameOver : public df::ViewObject {
  ...
};
```

In `GameOver.cpp`, the GameOver constructor needs to set its type to "GameOver" and link the "gameover" Sprite to the object. As for other Sprites, the Sprite in `sprites/-gameover-spr.txt` must be loaded via the ResourceManager in `loadResources()` in `game.cpp`.

```
RM.loadSprite("sprites/gameover-spr.txt", "gameover");
```

Note, the GameOver sprite uses , with the sprite file HEADER indicating all `'#'` characters should not be drawn.

```
transparency #
```

Like the Explosion above, the GameOver object has a `time_to_live` equal to the frame count, multiplied by the sprite slowdown (set to 15 in the sprite file). This can be done by querying the Sprite object `getFrameCount()` and `getSlowdown()` methods:

```
// Link to "message" sprite.
if (setSprite("gameover") == 0)
  time_to_live = getAnimation().getSprite()->getFrameCount() *
      getAnimation().getSprite()->getSlowdown();
else
  time_to_live = 0;
```

GameOver should register interest in the "step" event (`STEP_EVENT`) and center itself in the window:

```
// Put in center of window.
setLocation(df::CENTER_CENTER);

// Register for step event.
registerInterest(df::STEP_EVENT);
```

The GameOver `eventHandler()` should recognize the "step" event, whereupon it calls `step()` to decrease the `time_to_live`. When `time_to_live` reaches zero, it marks itself for deletion with the WorldManager.

```
// Count down to end of message.
void GameOver::step() {
 time_to_live--;
  if (time_to_live <= 0)
    WM.markForDelete(this);
}
```

In the GameOver destructor, indicate to the GameManager that the game is over via `setGameOver()` (remember to include `GameManager.h` at the top of the file).

```
// When object exits, indicate game over.
GameOver::~GameOver() {
  GM.setGameOver();
}
```

Since GameOver does not want to display any values associated with the default ViewObject , override the `draw()` method in `GameOver.h`.

```
int draw();
```

In `GameOver.cpp`, define the overridden `draw()` to just call the parent Object `draw()` method to display the Sprite.

```
// Override default draw so as not to display "value".
void GameOver::draw() {
  return df::Object::draw();
}
```

To hook in the GameOver object, the Hero object no longer ends the game in its destructor. Instead, it creates a GameOver object.

```
// Create GameOver object.
new GameOver;
```

The Hero destructor should also ask the WorldManager to delete the Reticle.

```
// Mark Reticle for deletion.
WM.markForDelete(p_reticle);
```

You may want to add code to create a really big explosion too, for a nice effect. Rather than a fixed animation like the Explosion, we'll use particle effects for a contrast.

```
// Make a big explosion with particles.
df::addParticles(df::SPARKS, getPosition(), 2, df::BLUE);
df::addParticles(df::SPARKS, getPosition(), 2, df::YELLOW);
df::addParticles(df::SPARKS, getPosition(), 3, df::RED);
df::addParticles(df::SPARKS, getPosition(), 3, df::RED);
```

With the end of the game finished, it is time to turn attention to the start of the game. Create a GameStart class for when the game begins that looks like GameOver except it replaces the private method `step()` with the private method `start()`. And GameStart does not have a `time_to_live` attribute nor a destructor.

```
class GameStart : public df::ViewObject {

 private:
  void start();

 public:
  GameStart();
  int eventHandler(const df::Event *p_e);
  int draw();
};
```

The GameStart constructor should set the Object type to "GameStart", place itself in the center of the window, and link to the "gamestart" sprite. Make sure to have the ResourceManager load the appropriate Sprite in `loadResources()` in `game.cpp` GameStart also registers for keyboard events since it has the user press keys to start or quit the game.

The GameStart `eventHandler()` only needs to handle keyboard events. It should check for a 'P', then call `start()` which starts the game or 'Q' which quits by setting the game to be over in the GameManager.

```
int GameStart::eventHandler(const df::Event *p_e) {

  if (p_e->getType() == df::KEYBOARD_EVENT) {
    df::EventKeyboard *p_keyboard_event = (df::EventKeyboard *) p_e;
    switch (p_keyboard_event->getKey()) {
    case df::Keyboard::P:                    // play
      start();
      break;
    case df::Keyboard::Q:                    // quit
      GM.setGameOver();
```

```
        break;
      default: // Key is not handled.
        break;
      }
      return 1;
    }

    // If get here, have ignored this event.
    return 0;
  }
```

Now, when the user hits 'Q' during the game, the game should return to the main menu. Do this by changing the code in the Hero `kdb()` method to destroy the Hero.

```
  case df::Keyboard::Q:                  // quit
    if (p_keyboard_event->getKeyboardAction() == df::KEY_PRESSED)
      WM.markForDelete(this);
    break;
```

With GameStart, instead of having all the game objects spawn in `populateWorld()` in `game.cpp`, have the Saucers and the Hero spawn in the GameStart `start()` method. The `populateWorld()` function still creates the Stars and also spawns the GameStart object.

```
  // Populate world with some objects.
  void populateWorld() {

    // Spawn some Stars.
    for (int i=0; i<16; i++)
      new Star;

    // Spawn GameStart object.
    new GameStart();
  }
```

In the GameStart `start()` method, move the code currently in `populateWorld()` to create the Hero, Saucers and display objects.

```
  // Create hero.
  new Hero;

  // Spawn some saucers to shoot.
  for (int i=0; i<16; i++)
    new Saucer;

  // Setup heads-up display.
  new Points;                                  // Points display.
  df::ViewObject *p_vo = new df::ViewObject; // Count of nukes.
  p_vo->setLocation(df::TOP_LEFT);
  p_vo->setViewString("Nukes");
  p_vo->setValue(1);
  p_vo->setColor(df::YELLOW);
```

At the end of the `start()` method, when the game is ready to start, the GameStart object becomes inactive.

```
// When game starts, become inactive.
setActive(false);
```

Add the needed #includes (e.g., GameStart.h to game.cpp) to all files.

The last bit of work to do is in the GameOver destructor, where it needs to re-activate the GameStart object. Before doing this, however, the GameOver object has some housekeeping to do. It needs to remove all the Saucers and the ViewObjects (i.e., "Nukes" and "Points"). It can do this by getting all the objects from the WorldManager and iterating through them, deleting the ones it does not want. When it sees the GameStart object, it sets it to be active.

```
GameOver::~GameOver() {

  // Remove Saucers and ViewObjects, re-activate GameStart.
  df::ObjectList object_list = WM.getAllObjects(true);
  df::ObjectListIterator i(&object_list);
  for (i.first(); !i.isDone(); i.next()) {
    df::Object *p_o = i.currentObject();
    if (p_o -> getType() == "Saucer" || p_o -> getType() == "ViewObject")
      WM.markForDelete(p_o);
    if (p_o -> getType() == "GameStart")
      p_o -> setActive(true);
  }
}
```

Compile your game and try it out!

You should see a blinking banner that gets displayed until the player hits 'P' or 'Q'. When the player hits 'P', the core gameplay commences, letting the player shoot Saucers until the Hero dies, then the game return to the start menu. This continues until the player hits 'Q' from the start menu.

The version of the game up to this point can be downloaded:

http://dragonfly.wpi.edu/tutorial/game11.zip

3.3.12 Audio

Sound is one-third the experience

In space, no one can hear you scream. While perhaps true, computer games set in virtual space, such as Saucer Shoot, are certainly more enjoyable with some sound effects. In fact, as game audio professor Keith Zizza[4] often says, "sound is one-third of the experience" (the other two-thirds being graphics and gameplay).

There are two different kinds of audio supported by Dragonfly - sound effects and music. First, you will add some sound effects for bullets and explosions. Like sprites, sounds are loaded and managed by the ResourceManager.

Download the sound pack used for this tutorial:

http://dragonfly.wpi.edu/tutorial/sounds.zip

[4] https://en.wikipedia.org/wiki/Keith_Zizza

Extract the sounds to a directory immediately under the location of the Saucer Shoot executable (e.g., `game/`), making sure it is named `sounds/`. Add code to `populateWorld()` in `game.cpp` that loads in the sound files.

Listing 3.7: Example of loading sound

```
RM.loadSound("sounds/fire.wav", "fire");
RM.loadSound("sounds/explode.wav", "explode");
RM.loadSound("sounds/nuke.wav", "nuke");
RM.loadSound("sounds/game-over.wav", "game over");
```

The "fire" sound is played when a bullet is fired. Do this by adding a call to the Sound `play()` method at the end of the Hero `fire()` method.

Listing 3.8: Example of playing sound

```
// Play "fire" sound.
df::Sound *p_sound = RM.getSound("fire");
p_sound->play();
```

The "explode" sound is triggered when a Saucer is destroyed. At the end of the Saucer `hit()` method, add code to play the explosion sound.

```
// Play "explode" sound.
df::Sound *p_sound = RM.getSound("explode");
p_sound->play();
```

The Hero should play the "nuke" sound at the end of the Hero `nuke()` method.

```
// Play "nuke" sound.
df::Sound *p_sound = RM.getSound("nuke");
p_sound->play();
```

When the Hero is destroyed, the "gameover" sound is played. There are several places this code could be added, but putting it in the constructor of the GameOver object works well.

```
// Play "game over" sound.
df::Sound *p_sound = RM.getSound("game over");
p_sound->play();
```

You might compile your game and try it out so see if the sound effects are working - firing sounds, explosion sounds, big nuclear explosion around and a big explosion when the game is over.

In addition to the game sound effects, Saucer Shoot could benefit from some music - specifically, music that plays in the starting menu. Given the continuous nature of music, it is treated somewhat differently by Dragonfly than the typically shorter sound effects. Because of this, the ResourceManager has different methods for sounds versus music. For example, loading the music for Saucer Shoot in `populuateWorld()` looks similar, but with a different method name.

Listing 3.9: Example of loading music

```
RM.loadMusic("sounds/start-music.wav", "start music");
```

For Saucer Shoot, the music plays when the GameStart object is active and displaying the Saucer Shoot banner, but does not play when the GameStart object is not active (i.e., when the game is in progress). To enable, this, create a new `public` method for GameStart called `playMusic()` and a new attribute called `p_music` for handling the music.

```
private:
 df::Music *p_music;
...
public:
 void playMusic();
```

`GameStart.h` needs to include `Music.h` for this.

In `GameStart.cpp`, in the GameStart constructor, add code to get the music from the ResourceManager and play it.

Listing 3.10: Example of playing music

```
// Play start music.
p_music = RM.getMusic("start music");
playMusic();
```

In the GameStart `start()` method, when the GameStart object becomes inactive, the music is paused.

```
// Pause start music.
p_music->pause();
```

Then, in the GameOver destructor, when GameStart is re-actived, the music is resumed.

```
if (p_o -> getType() == "GameStart") {
  p_o -> setActive(true);
  dynamic_cast <GameStart *> (p_o) -> playMusic(); // Resume start music.
}
```

Compile your game and try it out!

Saucer Shoot should start with the main menu banner and some music. When the game is in progress, the music should stop and there should be action sound effects. When the game ends, the music should resume.

The version of the game up to this point can be downloaded:

http://dragonfly.wpi.edu/tutorial/game12.zip

3.3.13 Final Touches

Look, something shiny!

For one of the final adjustments, it would be nice if the Hero and the Saucers both stay below the display objects (Points and Nukes). To do this, adjust the `move()` method of the Hero to be 3.

Listing 3.11: Hero move()

```
// If stays on window, allow move.
```

```
if ((new_pos.getY() > 3) &&
    (new_pos.getY() < WM.getBoundary().getVertical()))
  WM.moveObject(this, new_pos);
```

Then, adjust the Saucer `moveToStart()` method by 3 also.

```
// y is in vertical range.
temp_pos.setY(rand()%(int)(world_vert-4) + 4.0f);
```

Also, for the player, it can be useful to pause the game to attend to something in real life (e.g., answering the phone). In Dragonfly, this is done by instantiating a special Pause object in `game.cpp` right after the call to `populateWorld()`.

```
// Enable player to pause game.
new df::Pause(df::Keyboard::F10);
```

A `#include` to `Pause.h` is needed at the top of the file for this. Once enabled, the player can press 'F10' to pause the game.

The final version of the game can be downloaded:

http://dragonfly.wpi.edu/tutorial/game-final.zip

3.3.14 Where to From Here?

The sky is the limit!

Hopefully, you have been able to follow the tutorial all the way through, both to understand how to use Dragonfly and to provide a foundation for developing additional game enhancements. You can probably already think of a bunch of ways of extending the Saucer Shoot game. In case you need more ideas, here are a few ...

You could add additional weapon types. This might best be done by making a parent Projectile class that Bullet and other weapons inherit from. Lazers might be fast and destroy many Saucers in a line and Missiles might explode in a damage radius. You may need an additional event to handle the Missile explosion, in that case.

The final explosion of the Hero is made up of a bunch of little explosions. This works, but could look better with a bit of work. You could make a new Explosion sprite with a single, really big explosion animation. You could then generalize the Explosion class to take the sprite name of the explosion Sprite to be use as a parameter, probably in the constructor. When adding new sprites, the list of colors Dragonfly supports can be found in the `Color.h` header file.

The Hero could have multiple lives and/or with health. Health would be decreased upon being hit, with loss of life at zero health. The health could be displayed with a ViewObject, similar to "Points" and "Nukes".

While Stars have a parallax effect in their movement, they are all the same size. Stars closer to the camera should move faster *and* should be larger. This can be done using the Dragonfly `Shape` class. The code below shows how the `Shape` class can be used to make Stars use a white circle (besides `CIRCLE`, other supported shapes are `TRIANGLE` and `SQUARE`).

Listing 3.12: Star draw() modification to use Shapes.

```
// Draw Stars with circle.
// Closer Stars are bigger and move faster.
df::Shape s;
s.setColor(df::WHITE);
s.setType(df::CIRCLE);
s.setSize(5 * getVelocity().getMagnitude());
setShape(s);
```

You would put the code above in the Star constructor and also at the end of the Star `out()` method. You would lastly need to remove the Star `draw()` method, since drawing the shapes would be handled automatically by Dragonfly.

Having the screen "shake" during major events, such as a Nuke or Hero destruction, is another neat visual effect. To the Hero destructor, try adding:

```
// Shake screen (severity 20 pixels x&y, duration 10 frames).
DM.shake(20, 20, 10);
```

If you like it, you can also add a `DM.shake()` with about 3/4 the severity and 1/2 the duration to the Hero `nuke()` method.

You could extend scores to a High Score table that was stored (say, on disk) past a single game.

You could have a way for the player to launch the game from an initial menu. A game-difficulty setting, such as number of initial Saucers or Hero fire rate could be set. You would create objects for these interactions that were spawned first, similarly to how the GameStart object is, leading into the main game when done.

These are just some of what are many ideas you could use to extend Saucer Shoot. Of course, even more ambitious work would create a different game, perhaps a Pac-Man-, Tetris-, or Chess-type game or even something totally new.

Have fun!