The Game Loop and Frame Rate Management

Brandon Foltz

www.brandonfoltz.com

Revision 1.0

Contents

Introduction

This lesson is intended for someone who is interested in learning the basics of game programming. I've assumed that you know almost nothing about game programming, so if you're a beginner then you're in the right place. You don't have to be an expert programmer, but you should be able to understand common concepts and programming fundamentals. The examples in this lesson are written in C, though they should be easy to understand if you've used a language with syntax that is similar to C such as C++, Java, or PHP.

1.0 The Game Loop

1.1 What is the Game Loop?

Most games consist of a series of steps that are repeated until a certain condition is met, or an important event takes place, at which point the game logic changes accordingly. For example, in PONG there are several things that happen repeatedly until someone wins the game. The game has to update the position of the ball on the screen based on it's velocity and direction, take input from the player(s) and update the paddles position on the screen based on this input, detect when the ball impacts a players paddle or top/bottom of the screen and take the appropriate action (such as bouncing the ball back), and detect if the ball has reached a players goal and increment the score accordingly (and then reset the ball, of course).

In a real-time game like PONG, each of these steps are repeated many times per second. More specifically, they happen every time the frame is updated. This is dependent on the games frame rate, but we'll get to that in a bit.

The series of steps I've described above take place in the **Game Loop**. This is the loop that contains all of the logic that needs to be repeated every frame. Even turn-based non real-time games have a game loop, although it may not cycle as often as in a real-time game like PONG, or Galaga, or Counter-Strike. However, in this lesson I'll be focusing on the game loop as it's used in a real-time game like those mentioned previously. If you're interested in making turn-based non real-time games (like Chess or Tic-Tac-Toe), the section on Frame Rate Management may not be relevant (depending on how you implement your game logic). However, when it comes to game development, knowing more certainly won't put you at a disadvantage. Even turn based games could be built on a constantly

updating game loop like one used in Counter-Strike, and your game logic would have to be written to handle this (it would probably spend a lot of time in an idle-loop, waiting for a player to provide input for their turn).

1.2 Implementing a Game Loop

While the job of the **Game Loop** is an important one, it is not a particularly complex thing. At it's base, it is a loop just like any other. The game loop has to start at some point, and will continue to loop until some condition is met. An example game loop is displayed below, written in C.

```
int bRunning = 1;
while (bRunning)
{
     //game logic goes here
}
```

The above loop will cycle infinitely until something in the game logic sets `bRunning` to zero. This is a very primitive game loop, but it's the base upon which we will build a more advanced game loop. If you're fresh off the “don't-know-a-thing-about-game-programming” train, we can add some abstracted functions into the example above, to give you a better idea of what might go on in the game loop.

```
int bRunning = 1;
while (bRunning)
{
     //some functions for a PONG game
     GetPlayerInput();
     MoveBall();
     MovePaddles();
     DetectCollisions();
     UpdateScreen();
}
```

2.0 Frame Rate Management

2.1 Fixed Frame Rate

The example game loop presented in the last section is a good start. Though without some more code it will run uncontrollably, and at different speeds on different computers! This is certainly undesirable, so this brings us to **Frame Rate Management.**

Assuming your game is going to be real-time and have graphics, the graphics displayed on the screen are going to need to be updated repeatedly. The rate at which they are updated is called the **Frame Rate.** If you're a movie aficionado, you probably know that the images displayed in movies are updated at a specific frame rate as well. The difference between the frames in games and movies is that in a movie, each frame is already created and ready to be displayed from a file or film. In a game, each frame has to be created on the spot, very quickly by the computer.

If we choose to make our game run at a **Fixed Frame Rate** we will have a static amount of time between each frame being displayed, during which the computer can generate the next frame. This has some advantages, as well as disadvantages.

One advantage is that some of the game code (such as physics calculations) can be slightly simplified, since the game will run at the same speed on every computer. Additionally if you're writing for a static platform (like a console), you can be sure that the available hardware will run your game without issues. Many old consoles and arcade games used this method.

There is a pretty significant disadvantage to this method though. If the computer running your game isn't fast enough to generate each frame in the alloted period of time, your game will run visibly slower as there is no mechanism in place to compensate for this.

2.1.1 Implementing a Fixed Frame Rate Game Loop

First, we need to decide what frame rate we want our game to operate at. A higher frame rate will be visibly smoother, but it will also take a faster computer to generate the higher number of frames each second. Using a lower frame rate may make the game appear choppy (if the frame rate is less than about 20), but it will also be less resource demanding for the computer it's running on. Usually, 30 frames per second is a safe bet. It's not so low as to be visibly choppy, and not so high that common

computers won't be able to run the game.

So what do we need to do to make our game loop cycle at a specified rate? Well, first off we need a function that will allow us to keep track of time very precisely (at least to 1/1000 of a second). We also need a function that can delay the cycling of our loop for a precisely specified period of time. There are a number of libraries we can use to do this. Some of them are system specific, but the one we're going to be using is cross-platform and available for Windows, Linux, Mac OSX, and a number of other systems.

2.1.2 Introducing SDL

SDL (Simple Directmedia Layer) is a cross-platform multimedia library that provides low level access to audio, keyboard, mouse, joystick, 2D video framebuffer, and 3D hardware (via OpenGL) . You'll need to get the SDL development and runtime libraries for your system, they are available at www.libsdl.org. It is a relatively easy to use library, with many functions useful for writing games. We're only interested in two of them at the moment.

`SDL_GetTicks()` returns the number of system ticks that have taken place since the start of the program. There are 1000 ticks per second, so we can effectively use this to track the passage of time down to the millisecond (one one-thousandth of a second).

`SDL_Delay(Uint32 ms)` is used to delay the program from processing further instructions for a specified number of milliseconds. We specify how long to delay by passing an unsigned 32-bit integer as the functions only parameter, `ms`.

2.1.3 Using SDL to implement frame rate control

So now that we've decided what frame rate we want our game to run at (in this case, 30 frames per second), we need to calculate how long each frame has to be generated (and consequently, how long each frame will be displayed on the screen). So take 1000 (the number of milliseconds in a second, and also the unit of measurement provided by the function `SDL_GetTicks()`) and divide it by our chosen frame rate. In this case, we get about 33 milliseconds (1000ms / 30fps = 33.33ms per frame).

Depending on how complex the game is, a given computer may not take the entire alloted period of time to generate each frame. We need to time how long it takes to generate each frame, and

then subtract that from the alloted period of time to establish how long to idle until the frame should be displayed on screen. Let's say that the computer takes 9ms to complete all the game logic and render the frame on screen. We still have 24ms before the next frame should be rendered, so we wait. Below is a code example of how this type of loop could be written in C:

```
#include <SDL.h> //need to include the SDL header file
int main(int argc, char *argv[])
{
     int bRunning = 1;
     int frameRate = 30;
     frameMs = 1000 / frameRate; //calculate the length of each frame
     while (bRunning)
     {
          startMs = SDL_GetTicks(); //when the frame starts
          DoGameLogic(); //game code
          RenderFrame(); //display the frame
          endMs = SDL_GetTicks(); //when the frame calculations end
          delayMs = frameMs - (endMs - startMs); //how long to delay
          SDL_Delay(delayMs); //delay processing
          //once delay is finished, loop cycles again
     }
     return 0;
}
```

2.2 Variable Frame Rate

In the last section you learned how to use a fixed frame rate game loop. A fixed frame rate loop can be useful in some situations, but in most cases it's disadvantages outweigh it's benefits. This is especially apparent if the game is running on a slow computer, or if some action going on in the game takes a lot of processing time and slows the game down.

Enter, **Variable Frame Rate**. As the name implies, a variable frame rate will change constantly. This method can take into account slow computers and intense calculations, all the while making sure

events in the game happen at the right speed. The game loop and some of your game logic will be a bit more complex, but the added complexity will more than pay off with it's flexibility. This type of loop can make sure that everything moves at the right speed because all of your calculations will be based on the passage of time, as opposed to the passage of each frame.

2.2.1 How it works

Before we can get into the code and see this type of loop in action, there are a few more concepts to discuss. The **Variable Frame Rate Game Loop** is noticeably different from it's fixed counterpart, and requires a bit more code to be used effectively. Since our frame rate can fluctuate, there is no limit on how much time the computer can take to render a frame. Simply put, the frame is displayed on the screen when it is finished.

Almost certainly, each frame is going to take a different amount of time to be rendered due to different things going on in the game. Some things such as intense physics calculations can drastically fluctuate in the time they take to complete. At one moment, a computer might be running a game at 60 frames per second, and the next moment something happens which brings it to a crawling 20 frames per second. If you've ever gone out and purchased a brand new game, only to bring it home and discover that it runs poorly on your several-year-old machine, you know exactly what I'm talking about. Even though the computer cannot display the frames as rapidly as one might hope, the game physics and events still take place at their proper rates. For example, in a racing game your vehicle might move at 50 units per second regardless of whether the game is running at 15 frames per second, or 200!

So how can you apply this very useful feature to your game? First you need to understand **Delta Time**. In mathematics, the term **Delta** is used to describe a change in something. In this case, we're describing the change in time between frames. Since each frame will take a different amount of time to be processed, we need to account for **Delta Time** in our games calculations. Most obviously, we will have to use it in physics calculations that update the positions of game objects.

In order to do this, we have to time how long each frame takes to process and pass that value to each calculation. You might be thinking “How do I know how long a frame is going to take to take to process, before it gets processed?”. That's a valid question, and the answer is: you don't. Since we can't see into the future, we take the **Delta Time** of the *previous* frame, and pass it to the calculations of the *next* frame. This creates a slight delay between player input, and the effects of that input being updated

on the screen. However as long as the frame rate is kept relatively high (20+), this delay is not perceptible. Here's an example of a common game physics calculation that takes **Delta Time** into account.

```
object.x = object.x + (object.speed * deltaTime);
```

Your game objects velocity should be measured with the same unit as `deltaTime`. In this case, if `object.speed` were set to 300 units per second, and `deltaTime` were set to 1, then the objects position would be adjusted by 300 units according to the one second that has passed.

2.2.2 Implementing a Variable Frame Rate Game Loop

If you haven't read section 2.1.2 “Introducing SDL”, quickly go back and read it. We're going to continue to use some functions from the SDL library. Below you will see an example of a program with a **Variable Frame Rate Game Loop** written in C.

```
#include <stdio.h>
#include <SDL.h> //need to include the SDL header file

void Update(int deltaTimeMs);
int bRunning;
int main(int argc, char *argv[])
{
     int frameStart, frameEnd, deltaTime;
     frameStart = 0;
     frameEnd = 0;
     deltaTime = 0;
     bRunning = 1;
     while (bRunning)
     {
```

```
            frameStart = SDL_GetTicks();
            Update(deltaTime);
            frameEnd = SDL_GetTicks();
            deltaTime = frameEnd - frameStart;
      }
      return 0;
}
void Update(int deltaTimeMs)
{
      float deltaTimeS;
      deltaTimeS = (float)(deltaTimeMs) / 1000;

      //update code
      //example physics calculation using delta time:
      //object.x = object.x + (object.speed * deltaTimeS);

      printf("Elapsed time: %fS.\n", deltaTimeS);
}
```

If we examine the above code, it's relatively easy to see how this type of loop works. You will also see that our game calculations are moved to the `Update(int deltaTimeMs)` function. This is not absolutely necessary, but it improves code readability. Before game calculations take place in `Update(int deltaTimeMs)`, we set `frameStart` to the current tick count. Once calculations are finished, we set `frameEnd` to the current tick count. We then find the difference between these two points in time, which is equal to how much time (in milliseconds) has passed. This is the **Delta Time**. When the loop cycles around again, the **Delta Time** of the last frame gets fed into `Update(int deltaTimeMs)` to be used in the game calculations for the next frame.

If you're thinking ahead, you might notice a slight problem with the code above that only occurs under certain circumstances. What happens if the amount of time that passes between each frame is less than one millisecond? Our timer is only able to track time down to the millisecond, anything less than that will return a **Delta Time** of zero. So even though some time has passed (albeit very little time), our

calculations get fed a **Delta Time** value of zero. Of course we all know that anything multiplied by zero is zero, so `Update(int deltaTimeMs)` will essentially do *nothing,* even though time *has* passed. This probably sounds like an issue that's unlikely to occur very often, and in most cases you'd be right. With modern computers however, even complex calculations can be completed extremely quickly. Sometimes, in less than one millisecond (especially if your game is very simple!).

So how can we rectify this issue? The solution is rather simple, we just have to make sure that at least one millisecond passes between each frame. We can add a bit of code into the above loop that will accomplish this. Place the following code immediately after the start of your game loop, and immediately before `frameStart = SDL_GetTicks();`.

```
if (deltaTime < 1)
{
      frameStart = SDL_GetTicks();
      SDL_Delay(1);
      frameEnd = SDL_GetTicks();
      deltaTime = frameEnd - frameStart;
}
```

We could simplify the above condition to include only `SDL_Delay(1);`, but that presents us with yet another problem. `SDL_Delay(Uint32 ms)` is not a perfect function. It will occasionally delay the program for up to 10 milliseconds longer than the time we specify! If we don't time how long it actually delays the program for, this will cause our calculations that use **Delta Time** to become inaccurate. Though as long as we time how long the program is actually delayed for (and as long as it's longer than 1ms, which we can be sure of), the game calculations will be accurate.

Conclusion

In this lesson we have learned what the game loop is, how to use it, as well as fixed and variable frame rate management techniques. Additionally, we touched on using the SDL library to provide some basic functionality needed to write programs featuring the concepts discussed. I hope you can take this knowledge and apply it to your exciting game projects in the future!