Introduction

This tutorial follows on from **Space Invaders 101**. The tutorial will cover high resolution timers and the issues that raises in Java. In addition to timing the tutorial will adapt the previously developed source to add some basic animation.
The result of this tutorial can be seen **here**. The complete source for the tutorial can be found **here**. Its intended that you read through this tutorial with the source code at your side. The tutorial isn't going to cover every line of code but should give you enough to fully understand how it works.

Context highlighted source is also available here:
Game.java
SpriteStore.java
Sprite.java
Entity.java
ShipEntity.java
ShotEntity.java
AlienEntity.java
SystemTimer.java

Disclaimer: This tutorial is provided as is. I don't guarantee that the provided source is perfect or that that it provides best practices.

Timing

What exactly do we mean by timing? Timing is used in many places in games. For many games timing is used to limit the rate at which the screen is updated, the frames per second. Most games use some sort of movement and animation which is normally based on timing.
Currently, Java has some issues with timing. In Java 1.4.2 the simplest way to get hold of the time is to use System.currentTimeMillis(). Infact it was used to time the movement in the first tutorial. However, on some Windows based machines the resolution of the timer (the smallest value you can time) isn't very useful. On most systems (Linux, SunOS, etc) the resolution is at least 1 millisecond. However, on some Windows sytems the resolution is bad enough to mean that timing can be relied on to give a consistant result, which in turn can lead to stuttering in updates. Lets look at some possible solutions.

Averaging the Time

One way to deal with the inconsistency of timing on Windows is to average the change in time between frames. Historically this was a very common way to deal with the problem. Using this method the change in time between frames is recorded across the last 5 (or more) frames. Instead of using the actual change in time, the average of the last few frames is taken and used instead.

So the algorithm looks something like this:

- Each game loop record the render time between frames.
- Average the last 5 frame times
- Use the average to move and update game elements

While this doesn't give perfect results, it does give "good" results. Its not required on anything other than Windows. So if you do choose to implement this method its nice to add a check on the system property "os.name".

Wait for Java 1.5

The timing issue with Java has been well known for a while. Finally, in Java 1.5 its going to be fixed. However, Java 1.5 is currently in beta (in testing) and its not available for commerical use right now. Sun rarely give fixed dates for releases of Java so no one is quite sure when 1.5 will be available. Most Java releases have also taken a while to really be ready for use. In addition, its fair to say that games players are only going to upgrade their Java when absolutely required, so most won't have 1.5 for quite a while. That being said, using 1.5 is by far the most simple way to get a high resolution timer.
Using the Java 1.5 timer couldn't be simpler:

```
long currentTime = System.nanoTime()
```

There you have it! As a side note, there is a high resolution timer "hidden" in Java 1.4.2, however this tutorial doesn't cover it. Why? There is no guarantee the hidden timer will be available in any given JVM and hence there is no point using it except in very specific circumstances.

EDIT: Obviously this tutorial was written before Java 1.5, 1.6 or any future release of Java. Java 1.5 did indeed bring System.nanoTime(), unfortunately as of now (1.6 being the current release) nanoTime() is still bugged and responds in strange ways on windows and any system with multiple processors. The advice given here to consider a native library still holds true - GAGETimer in particular will adapt based on the current version of Java.

Native Timing Library

Java's problem with timing on Window's isn't actually a result of the operating system not supporting high resolution timers. Windows itself does support high resolution timing, its just not possible to access it directly in Java (well not until 1.5). What we need to do is access the timer from the operating system at a native level.. but how!?

Why do we need native libraries?

Java can be viewed as 3 parts. First, the Java Language. The lanaguge is what makes most developers come to love Java, its pushes you towards writing structured and elegant code. Second, the JDK, the set of Java software libraries

Disclaimer
Note that the views on this page are not intended to offend. If they do, you might be taking the content too seriously.

Sponsors
Stock 3D Game Models

Twitter Updates
follow me on Twitter

More Games

2D OpenGL Based Game Library

2D Game Physics Engine in Java

Game Developers
How about a **list of the developers** doing interesting things in java gaming.

Game Dev Resources
Looking for Game Development Resources? Check out **the List**!

Projects

- Completed Games
- Libraries and Components
- Tools and Utilities
- Tutorials
  - Space Invaders - 2D Rendering in Java
    - Space Invaders 101 - An

that make it easy to put applications together quickly in Java. Finally, we have the Java Virtual Machine (JVM), the interface from Java to the machine and the operating system its running on. The virtual machine, while being a key factor in allowing Java to be platform indepedant is also a key factor in limiting what is possible with Java. If something is possible in C++ it first has to be added to the JVM before its possible in pure Java.
Well, that sounds terrible doesn't it? We have to wait for Sun to update the JVM before we can access the latest and greatest platform dependant feature. Actually, no, this is exactly what the Java Native Interface (JNI) is designed to allow. The interface lets us use native libraries from Java to access features of the platform not yet exposed to the JVM. However, the downside is that unless you implement a native library for every platform your software now only works on a particular operating system. Its a trade off that should be taken pretty seriously when you're looking at writing Java software.

Native Timing

Where high resolution timing is concerned we get very very lucky. The only platform that the JVM has an issue with getting good timing is Windows. So if we implement a native library to get us a high resolution timer on Windows we're clear for multi-platform development!
Whats even better, is that there are at least a few free implementations of this timing library already available, so we don't even need to touch the C++. A good implementation that a large number of people use is the **GAGE Timer**. Its freely available and has a good track record.

If you download the GAGE Timer package, you'll have a dynamic link library (DLL) for Windows and a Jar file containing the interface to the timer. To continue with this tutorial the DLL must be in the directory you are running from (only if you're on Windows of course), and the Jar file must be referenced in your classpath.

Wrapping up the Timing

As you can see timing can be implemented in a series of different ways in a set of difference circumstances. With this in mind lets wrap up our use of the timer in a single class. This way if we choose to try out a different timing method then we only have to make our changes in one place.

```
package org.newdawn.spaceinvaders;

import com.dnsalias.java.timer.AdvancedTimer;

/**
 * A wrapper class that provides timing methods. This class
 * provides us with a central location where we can add
 * our current timing implementation. Initially, we're going to
 * rely on the GAGE timer. (@see http://java.dnsalias.com)
 *
 * @author Kevin Glass
 */
public class SystemTimer {
        /** Our link into the GAGE timer library */
        private static AdvancedTimer timer = new AdvancedTimer();
        /** The number of "timer ticks" per second */
        private static long timerTicksPerSecond;

        /** A little initialisation at startup, we're just going to get the GAGE timer going */
        static {
                timer.start();
                timerTicksPerSecond = AdvancedTimer.getTicksPerSecond();
        }

        /**
         * Get the high resolution time in milliseconds
         *
         * @return The high resolution time in milliseconds
         */
        public static long getTime() {
                // we get the "timer ticks" from the high resolution timer
                // multiply by 1000 so our end result is in milliseconds
                // then divide by the number of ticks in a second giving
                // us a nice clear time in milliseconds
                return (timer.getClockTicks() * 1000) / timerTicksPerSecond;
        }

        /**
         * Sleep for a fixed number of milliseconds.
         *
         * @param duration The amount of time in milliseconds to sleep for
         */
        public static void sleep(long duration) {
                timer.sleep((duration * timerTicksPerSecond) / 1000);
        }
}
```

This timer class is based on the use of the GAGE timer. We need to support two main operations, getting the time and sleeping for a set period of time. The gage timer supports both of these operations, however it requires you specify the time in "timer ticks" not in milliseconds. The main job of this class is to map between the timer ticks provided from the native timer to milliseconds and back.

Simply put, we create an "AdvancedTimer", the timer given to us by GAGE. We find out its resolution and start it off running. The final step is to provide our methods based on using the timer.

Updating the Game Loop

Next we need to modify our existing game loop to make use of the high resolution timer. First, lets change how we calculate our delta (change in time) value. Instead of calling System.currentTimeMillis() we'll use our new SystemTimer class. That'll look something like this:

```
// work out how long its been since the last update, this
// will be used to calculate how far the entities should
// move this loop
long delta = SystemTimer.getTime() - lastLoopTime;
lastLoopTime = SystemTimer.getTime();
```

Now, in the last tutorial since the timer wasn't designed to be perfect we didn't worry to much about a few lost milliseconds. This time we can rely on our timer to be millisecond accurate so we're going to try and strictly limit our frame time so we get exactly 100 frames per second (FPS).

To do this we're going to want each cycle round the game loop to take exactly 10 milliseconds. We know at what time the cycle started (lastLoopTime) and we know what time it is now, so with a small amount of maths we can sleep for the right amount of time like this:

```
// we want each frame to take 10 milliseconds, to do this
// we've recorded when we started the frame. We add 10 milliseconds
// to this and then factor in the current time to give
// us our final value to wait for
SystemTimer.sleep(lastLoopTime+10-SystemTimer.getTime());
```

Note: GAGE Timer actually supports a "sleepUntil()" method that could be used here. However, since the SystemTimer is trying to allow us to change between timing mechanisms we should try to rely on simply sleeping for the right amount of time.

Animating them Aliens

One of the things directly dependant on accurate timing is any form of animation. Since we've now got acurate timing lets do some animation! The aliens coming down towards our player are little static for traditional space invaders. Lets make them dance around on the way down.
Since we designed our source nicely last time our changes are limited to one class, **AlienEntity**. Instead of the entity maintaining just a single sprite we'll add a few sprites and flip between them over time, i.e. Animation. Our first step is to add some addition variables to our AlienEntity:

```
/** The animation frames */
private Sprite[] frames = new Sprite[4];
/** The time since the last frame change took place */
private long lastFrameChange;
/** The frame duration in milliseconds, i.e. how long any given frame of animation lasts */
private long frameDuration = 250;
/** The current frame of animation being displayed */
private int frameNumber;
```

The *frames* array is going to hold our frames of animation. *lastFrameChange* is going to be a record of the last time we changed animation frame. *frameDuration* will be the length of time that each frame will be displayed on the screen. Making this small will make the aliens dance more quickly. Finally, *frameNumber* will be the index of the frame we are currently showing in a frames array. This will be incremented to cycle us through the animation.

Next we're going to need to load up our sprites. We're going to modify the constructor to grab the frames. However, our standard Entity class will already load one sprite for us (the one it used to display). So we need to load two additional ones, like so:

```
public AlienEntity(Game game,int x,int y) {
        super("sprites/alien.gif",x,y);

        // setup the animatin frames
        frames[0] = sprite;
        frames[1] = SpriteStore.get().getSprite("sprites/alien2.gif");
        frames[2] = sprite;
        frames[3] = SpriteStore.get().getSprite("sprites/alien3.gif");

        this.game = game;
        dx = -moveSpeed;
}
```

We've asked the Entity class to load "sprites/alien.gif" for us. Then we need to go off and load up a couple of additional sprites. We put the frame loaded by Entity and our two additional frames in the array in the right place to play the animation. Note that we've modified the constructor slightly to remove the name of the sprite, so
the **Game** class wll need some minor modifications.

The final step in getting our animation to play is to update the current sprite as time progresses. We already have a handy place in which we can perform this action. Our "move()" method already gets told when time passes, so we can update the animation there.

```
public void move(long delta) {
        // since the move tells us how much time has passed
        // by we can use it to drive the animation, however
        // its the not the prettiest solution
        lastFrameChange += delta;

        // if we need to change the frame, update the frame number
        // and flip over the sprite in use
        if (lastFrameChange > frameDuration) {
                // reset our frame change time counter
                lastFrameChange = 0;

                // update the frame
                frameNumber++;
                if (frameNumber >= frames.length) {
```

```
                        frameNumber = 0;
                }

                sprite = frames[frameNumber];
        }

        ...
}
```

So, as time passes our lastFrameChange counter will get updated. Once its passed our frameDuration limit we reset it. In addition we move to the next frame number. Then we reset the current sprite by setting the "sprite" member in the Entity super class to the current frame of animation. Next time the entity is rendered a different sprite is drawn and the animation takes place!

Finishing Off

Hopefully this tutorial has been of some help. While I recommend using the GAGE timer the other methods and other native libraries have been used by a great number of people and are probably just as good. Its worth looking around and choosing the method that suits your need and methods best.
If you have any comments or corrections feel free to mail me **here**

Exercises for the Reader

The following exercises might help the reader in understanding timing and animation more fully in this context.

Add animation for the shots

This should be a matter of adding some functionality similar to that added to AlienEntity to ShotEntity.

Implement timing using a different timer

There are a few native timer implementations. It should be relatively simple to try a different one. This would help understanding how Java uses native libraries to provide extra functionality.

Implement time averaging.

One of the other methods for providing useful timing on all platforms is to average the change in time across frames. The algorithm for doing is discussed above.

Add an Animated Sprite

Another way to implement the animation would have been to add a "AnimatedSprite". The flip over of images could have been handled within the class. In this way any entity could have been animated by simply using an animated sprite.

Credits

Tutorial and Source written by **Kevin Glass**
Game sprites provided by **Ari Feldman**
A large number of people over at the **Java Gaming Forums**

---

previous **up** **next**

Space Invaders 101 - An Accelerated Java 2D Tutorial Space Invaders 103 - Refactoring and OpenGL

printer-friendly version | **add new comment**

Thanks for good article

Thanks for good article about timing and animation in java.

By rbytes (not verified) at Wed, 2007-07-11 08:09 | **reply**