## Introduction

Java has always been puzzling. From what intuition let me expect, and to conclusions i could draw from full featured tests (macrobenchmarks), it always amazed me. Most often in good, as i have to admit i have rarely been disappointed by performances, stability and quality of the general Java platform. Some do not agree with that. I sometimes take time to try to see if they are right or not by creating macrobenchmarks so i can draw my own conclusions.
This time, the first that i make public, i wanted to see whose of the contradicting general advices i have been reading here and there from Sun and users of various discussions boards (JGO, javadesktop, Sun forums,..) were right.
As i had something like a week instead of the hours or day i normally take to create that kind of test, i decided to go for a small game-like application so it would be closer to reality than a loop displaying sprites moving on the screen. Moreover, it would be a bit funnier.
Actually, the 'analysis' is not deep but will be deeper as time goes by and as i get clues about what is going on.

## The goal

Test those various advices i got, or intuitions i had and their contraries when existing:

| | |
|---|---|
| Pool objects, don't allocate in game tight loop or thou shalt fear GC wrath. | Generational Garbage Collector lets you reuse objects at almost no cost. Allocating in small tight loop is okay. |
| Don't blit transluscent images, they are way too slow for anything. | Transluscent image blitting is accelerated, you can use it. |
| Don't transform images (scale, rotate) realtime. Precompute sprites or it will be too slow.. | Affine transformations are accelerated. Go ! |
| Java2D is too slow for gaming. Go for a GL library. 1.5 GL java2D is not suitable. | More and More things are accelerated. J2D can be suitable for gaming. 1.5's GL pipeline will give you full power. |
| Whenever you can, give GC some time so it can work. Use sleep() after rendering. | |

## The means

I played the devil's advocate. From the beginning i decided not to take care and do exactly what is said to be bad and forbiden as i felt that it was the best way to bring down the system. Nevertheless, i took care doing correct and clean OO code, or at least as correct and clean as my skills permit.
The game would allocate everything when needed, i would not take any time to reduce allocation and concentrate on coding the logic. I would act as a newbie game programmer (while not rookie general case programmer), using only what is provided to me in the platform. One of the goal is to program a game that would run reasonnably fast, be visually appealing, featured, has a reduced time-to-market and low entry game programming skill.

## Simplifications and things taken for granted

Platform timers are bad. GageTimer will be used for precise timing and accurate benchmarking on various platforms. This is the only exception to the rule stated above.

JRE 1.3 is excluded from tests. So are 1.1 and 1.2. The benchmark is meant to run 1.4 and 5.0. It is targeted to desktop users.
Area to Area collision had no intersect() method, so i created my own little buggy one. It is not done through boolean operation as i felt it was not appropriate.

## How it was done, aka « Call me heretic !!».

The game uses a simple architecture. It has a list of *Actor* that gets iterate three times per frame. First actors are added in the *init()* method of the main class, extending *Game*. Then, actors are responsible for the game. Actors can be anything, simple sprite, decoration, logic, ... Some might simply draw things (*BackgroungActor*, Foreground*Actor*, ..), some might act like logic parts only (*EnnemyCreatorActor*) or both (*PlayerActor, MineActor, SeaActor...*).
Those actors are treated in the order they are listed, but as they might not have to be drawn in that order, they hold a stack position that will be used to sort them at rendering step.
One of those actors is a *ViewportActor*. That actor is responsible for viewport movements and is treated seperatly: it is inited first so that viewport is set correctly for all actors rendered. Nevertheless ViewportActor is optional.
A game context object is allocated each frame -because non mutable-, containing data necessary for the actors, and acting as a bridge to *Game*. Amongst other things, it contains frame size, invocation time for last frame and game 3D viewport position.Context also contains input information as mouse position and pressed buttons. Key pressed informations are not present in current implementation.

First step is updating. Actors will have to set their internal state accordingly to given context. They might create other Actor objects (bubbles behind submarine, for example, or those over shoots), die or keep on living.
Second step is collision. All sprites are tested againsts all others. Collisions are done through the use of classes extending *java.awt.Shape. Area* collisions with other *Area* shapes are treated seperatly because there are no methods to detect overlapping of them two (except doing a boolean operation, which is something i avoided). Actor might die at this step and/or create new other actors.
Last step is drawing. All sprites are put in a freshly allocated list. That list is sorted bottom to top based on each *Actor*'s deepness. List is then iterated and sprites are drawn. Each sprite is free to redefine draw mode, compositing, and anything it needs.

Almost every game actors are performance demanding.

*SeaActor*, who is responsible for sea, sky and 'ground' rendering uses three gradient sprites and a filled rectangle to fill the gap between them. Gradients are 32*128 24 bits sprites, blitted 75 times each frame. While ground and dark blue fill are done normally, sea and sky rendering methods are special.Sea level acts as a limit for both parts of he two zones. It would have been bad to have a flat sea. As i wanted it to be a little active and react to bubbles, it was now difficult to paint the two gradients without seeing them overlapping. Sea rendering was a perfect candidate for an arbitrary shape clipped blit test. What i did was preparing an array of height and velocities in *update()* and use it to create two Area objects in *draw()* pass. Problem with Area objects is that they are non mutable. Once created you can not modify them -thus can't morph and had to be created each frame-. That worried me a bit, but that was the only way. Moreover, i was afraid of the time needed for painting images in non rectangular zones, like the one i was using.

Mine actors consist of a translucent sprite, and an *Area* for clipping. Both use the same *AffineTransform* for drawing and collision. *Area* gets transformed using an inverted *AffineTransform* computed each frame in order not to see it concatenate all transforms.

The mine is scaled by a size factor and rotated by a time based sinusoidal formula. When MineActor collides with a *PlayerActor*, it dies and creates some amount of *WobblingBubble*. That amount depends on the mine size. Mine can also have a scaled bubble drawn over it, in case it accumulated air thrown by the submarine.

*PlayerActor* is using an Area for collision and a big translucent sprite (256*128) having about 20% translucent surface.*PlayerActor* suffers from the very same problems as *MineActor*. It is realtime rotated based on mouse movements, and has a time based rotation stabilisation behavior. When firing, it creates a *PlayerShootActor* in the same direction the submarine points. Each shoot empties the air tank a bit. The only way to refill the tank is to go to the surface. Shooting while colliding with a SeaActor is not possible. PlayerActor also reacts to collision with MineActor. When they collide, PlayerActor takes damage, and MineActor dies. When PlayerActor has no more life, game ends and stats are printed.

*PlayerShootActor* is a collidable bubble that dies when colliding a mine and expurges a *WobblingBubble* from time to time. It is done with the same sprite as all bubbles, but its size is dependent on its age, and its height is divided by two. Bubble sprite is made of over 50% translucent pixels and less than 10% opaque ones. (numbers not verified)

*WobblingBubble* is a bubble that simply goes up and reacts to collision with a *SeaActor* instance. It is realtime transformed using an *AffineTransform*. Width and height are changed by a time based sinusoidal formula. Once the bubble hits a *SeaActor* instance, it dies and creates a *SurfaceBubbleActor* actor if big enough. That new actor will be a portion of the original bubble. When dying, the actor also sends an impulse to the *SeaActor* so that its surface moves.

*SurfaceBubbleActor* acts like a small bubble following the surface of the *SeaActor*. It dies shortly, but life length depends on its size.
Each and every bubble type has a global transparency based on its size. They are blended using
`g.setComposite( java.awt.AlphaComposite.getInstance( java.awt.AlphaComposite.SRC_OVER, size ) );`

*BackgroundActor* and *ForegroundActor* are the two actors responsible for drawing rocks in front and back of the submarine, bubbles and mines. *BackgroundActor* generates a line of 10 transparent rocks scrolling at low speed. Transparency helps in integrating them in the background, giving an impression of depth. They are scaled down around two times, and positionned differently by a small amount. *ForegroundActor* shows two lines of rocks. One consists of 6 non scaled rocks, perfectly aligned and sliding faster, while the other consists of two 200% scaled ones sliding even faster. The last line actually fills a pretty big part of the screen.

## How did major parts perform?

First, the game got tested here using two machines:
machine1: intel P4 2.8Ghz, radeon 9700, winXP
machine2: Amd duron 1.4Ghz, geForce1, win98

Note: Actor count is not the sprite/blit count. Sea actor blits 75 sprites and fills an area, Background and ForegroundActor blit 18 rocks and mines often show two sprites. You should have that in mind when reading numbers generated.

### *Java2D*

Jre 1.4.2_04
As is, the game ran at an average 55FPS on machine1 and 20FPS on machine2, both with around an average of 50 actors, which, when including background blits, makes

around 150 sprites. No doubt that i was pretty impressed. When i started that test, i thought it would be around 2 to 5 fps to handle all that stuff in the conditions i setup. The game suffered from no visual artifacts or incorrectness.
Under XP, using the various ddscale, transaccel, ddoffscreen flags did not help.

Jre 1.5.0_b2
normal pipeline:
Almost same speed, a bit slower in fact, same remarks. Nothing special to note on the rendering side of things.
OpenGL pipeline: (sun.java2d.opengl=True)
No surprises, it was way faster. 100FPS on machine1 due to cap on rendering. Unthrottled version was running an average 177FPS( 333 max ). More surprising, there were no glitches, no rendering errors, and quality was exactly the same. Apart from framerate, nothing could tell if i was using GL or regular Java2D. That was an excellent news, as from what i have read, i should have expected problems.
Machine2 died when i was about to make the GL test. (installed latest nvidia drivers and machine never went to win98 anymore. I will have to completely reinstall it and post the results later. Note to self: never do that again. )
The display of the configuration frame was buggy under GL pipeline. Looked like red and blue channels were swapped and content did not respond correctly to window movements (stayed on same place relatively to screen corner, not window one.) All sprites of the swing interface had channels swapped. Don't know if that was due to implementation or drivers, but i would tend on thinking implementation is the culprit.

Trying to mesure the impact of different things, i removed some parts and did other tests: (results are for machine1, jdk1.4.2_04)
- SeaActor: removing sky, water and ground *drawImage()* calls gained 6fps (11%)
- removing clipping and creation of zones while still drawing did not accelerate enough to be clearly perceptible. Maybe the fact that only a small part of the sprites get clipped plays a role.
- Changing the *WobblingBubble* rendering not to have a global alpha did not change anything. Test was done shooting at max for 15 seconds, and no perceptible change occured (41 FPS each version)

Bad news came when i made it fullscreen. Framerate dropped to 20fps on machine1! The reasons are still unknown as nothing changes in the rendering methods on my side, but i hope that some people at java2D team will look at that and find why.

### *Garbage colllection*

Garbage collection was a huge surprise to me. Despite my efforts to give it hard times, i could only see a small pause in the very first seconds, then nothing. Puzzled, i watched the memory and saw it extremely slowly growing around 22MBytes. Something like 8KB per second. That was of course an unrelated amount compared to what i was supposed to use. I then started the gamechmark using Sun's unsupported flag Xloggc (which is equivalent to -verbose:gc, but does not need piping to be saved. Why do i feel ashamed? :) ). Except the big surprise of seeing over 140 collections a second (!!) that were less than a millisecond each, ( generally .5ms to .2ms) the amount of data output was not really self explanatory. I decided to take nedit and macroed a bit to get a result.
First test was done in a session where i did not shoot. I took the log at 10s from the beginning, on a 1s period. I had 147 GC during that single second, totalizing 56581KB of collected ram. Yes, that is 56.6MB. Total GC time for that second was of 46ms, which makes -of course- 4.6% of the rendering time. Due to complexity of such a work and due to the amount of objects to initialise, i found this to be pretty nice. On an other hand, i

don't know what real object amount i'm creating, what amount is created by standard libs, and what is the size of those objects.
Second test was done in a session where i kept firing all the time. I took same period of the log. 60041KB (60MB) were collected and collection last... 43ms for 127 GC.
Looked like eden got automagically increased a bit due to usage, which seems a good thing.
Later, using VMSTAT confirmed all that.

A note on GC performance: The dell inspiron 9100 used for tests has an Intel 865PE chipset. It uses DDR400 and Intel claims 6.4GB/s ram transfer (apply marketing ratio here). 60MB in 43ms is a 1.4GB/s data manipulation as all objects are initialized when reused. A small C test filling 500 times a 4MB buffer revealed a rough performance of 1.2 seconds, which makes 1.6GB/s throughput. (mingw 3.2.3, -O3 -mcpu=pentium -march=pentium ). If the test is considered somewhat valid, GC reuse performance can be thought as efficient, at least.

Playing longer, i saw that there was a 30ms to 40ms pause about every 40 to 50 seconds. That pause was not that noticeable, and i would not have really percieved it without looking at GC logs. I don't know why all objects i create are not retrieved within eden, as i don't think i create any long term objects. No one that did test the gamechmark reported visible GC pauses.
Using a sleep() throttling method returned weird results. During some days, it paused a bit every ten seconds, but now does not anymore. Unfortunatly, that was before i started benchmarking and thought it was normal, so i will not know what it was. Anyway, now, it makes absolutly no differences, so if you have a reliable sleep() method, better use it. First, it gives time to the system to do what it needs and let him a chance to do it without interrupting you, and it does not turn on the fans of laptops during whole game. (which is, believe me, irritating)

***Startup time***

There are two behaviours that i remarked when using configuration frame.

Starting windowed is immediate and runs as expected. It is less than half a second to see the window appear and first frame is rendered.

Starting fullscreen acts a bit differently. It takes a bit longer, i guess it is because of the resolution change. Sometimes i get submarine drawn over a grey background for some frames, then it becomes normal. Reasons for that are unknown to me at the moment. Having the game started is below 2s.

When starting directly (calling *GoSub.main()*), the game is running in less than one second, which is a pretty nice startup time.

Due to the very small startup time and the impossibility to be certain of what i would measure, i stayed with that subjective data. Anyway, a one second long startup time was low enough as even 3 or 4 seconds would have pleased me.

Of course, i only load a very few sprites, which is a reason for short startup, but jvm startup has always been bashed for its duration. While i agree that it is long for doing command line, it is fast enough for gaming in some order of magnitude.

## Conclusion

... if any.

Well, i have been surprised positively. It was way faster than i -and other- thought, easy to code and globally efficient. The biggest thing and surprise in that test was Garbage

Collection. Coding object pools require time, enhance complexity and does not completely remove the risk of having a GC pause or system pause at any time. Not caring about them is a great news and makes me feel Sun should be proud of it.

New GL pipeline of 5.0 holds a lot of promises. When it will be completely successful (that is, it will also work flawlessly in swing and work with any GL capable configuration ) that will be a huge enhancement globally, but it is actually a big win for gaming. We will soon be able to make high quality games -even for low profile machines- and generally 2D games in a blink of an eye without this slight panic we get when thinking about those slow machines.

Actual pipeline does not seem to correctly handle non common graphic adapter (intel, at least) acceleration for unknown reasons. Its error output when not being able to display is extremely sparse, which is more than annoying in a beta feature that still needs being debugged.

A big flaw -in my humble opinion- in Java2D at the moment is that there are no additive/substractive/multiplicative blending modes. Those are an important graphic quality enhancer for actual gaming and will miss more and more as time goes. My biggest hope is that J2D team will rapidly add such features so each and every title can be written.

## TODOs:

Refactor in order to also use a BufferedImage as a backbuffer and add an option.

Finish to debug some little things in the game related to behaviours of actors related to time.

Improve sea.

Many thanks to Jareson Banes (jBanes), Abuse, Kthuul, RoughDuck, Malohkan and Skip 'Skippy' Balk for their tests and help.