**Metanact - Approaches to Testing**

*Seb Holzapfel*

**Overview**

Metanact has a codebase > 6000 lines of code; hence, there’s a huge probability of some bugs being present! In this document, I’ll detail various tests I performed on the program throughout its development in order to meet 3 central criteria: stability *within the program (intra)*, stability across *different computer systems (Inter)*, and *sufficient performance*.

**Intra-programmatic Stability**

When I say Intra-programmatic, I’m talking about instantaneous bugs. Features I’ve just implemented and aren’t working as I’d like. To address these sorts of bugs, we use debugging tools: such as stepwise debuggers, in-program assertions – even manual means, such as desk-checks. Throughout the development of Metanact, I used these tools in various combinations to rid of unwanted bugs. As an example, I’ll go through an *intra-programmatic* bug that tortured me during the early half of the project:

**What was broken:** The game’s camera keeps a pointer to the area in memory where the entity is that it should follow. It uses the entities’ pointer to fetch x & y coordinates to go to. I found this worked fine, BUT, only for about ~10 seconds (which varied) after which the game crashed. What the heck was going on?!

**Looking for a solution:** I stepped through the game’s relevant code by hand (effectively an automated desk-check) and found that the entities’ pointer was becoming invalid after a little while. Why was this? Well, after some scrolling through the entities’ pointer-return code, my brain ticked.

**The fix:** Turns out I was continuously making a local copy of the entity on the CPU’s STACK, rather than HEAP. This is important, as stack memory is freed at the return of every function call, whereas HEAP memory stays allocated until it’s freed by hand. My code should have crashed instantly – it only survived for a short while until some other data in the program overwrote the free memory I’d mistakenly thought had been permanently allocated for an entity.

**Inter-systematic Stability**

Inter-systematic stability is talking about inconsistencies between computer systems in terms of functionality. For example, where one feature of the program may work on one computer; it may not work on the other & crash the program. Usually, this is a driver or OS issue – less often, it can be due to hardware. To solve these sorts of problems, it usually involves looking at manufacturer’s bug reports on Google as well as a code investigation. Luckily for me, I managed to come across an example of hardware inconsistency in my project:

**What was broken:** The game worked fine on most computers – but on some, the program would crash after about 5 seconds logging the error ‘Syntax error while compiling shader: equality operator not supported between Vec2’.

**Looking for a solution:** I did some research and soon discovered that shaders are always compiled at runtime by the graphics card’s drivers. Looking for common ground between the computers that were crashing, they all were using Intel integrated cards. I then had a poke around Google, and turns out there’s a subtle manufacturer inconsistency between AMD, NVIDIA & Intel’s shader language compiler.

**The fix:** After some more research I managed to implement a snippet of someone else’s code that was cross-compatible across graphics cards – shaders then worked on all 4 of my test systems.

**Performance Targets**

For the application to be playable, it inevitably must meet some performance targets. This involves the diagnosis of program bottlenecks and optimizing code accordingly to remove these bottlenecks. I use a program called Very Sleepy, which is a performance profiler – and shows a list of the functions in your program in descending order of percentage runtime used. (i.e, GravityCalc() = 25%, DrawEntity = 40% etc.). After observing what used the most time, I’m able to optimize the code that will speed up the program the most. As an example, let’s look at a problem I had with draw-culling:

**What was broken:** The game was a lot slower than it should have been! I was getting ~40 FPS with a few hundred objects when I expected performance in the reion of ~400 FPS.

**Looking for a solution:** After profiling, it turned out 90% of the program’s time was spent in the DrawEntity() function. Most graphical 2D libraries do their own sprite-culling – that is, when a bitmap is to be drawn off-sceen, the graphics card isn’t addressed. The library I was using though; for some reason didn’t do this.

**The fix:** I stuck a conditional in the entities’ draw code so that when it was off-screen, it wouldn’t call SFML’s internal draw function. Performance went from 40 FPS to 600 FPS on larger levels.