

Integration Test Servers for Videogames

by Philip

A modern videogame is a complex beast, requiring special strategies to tame. High Moon Studios' Darkwatch project applied a breadth-first strategy, to collate test results from all game levels, transmit summaries, and trigger test runs when source changed.

The closest analog to a modern, high-end videogame project is a major motion picture. Both focus the talent of hundreds of skilled professionals onto a script. Both wager millions of dollars on success. And both require treading a careful path through a minefield of complex technical risks.

Advanced, 3-dimensional, animated videogames, unlike movies, have only a decade of experience. Like the rest of the software engineering field, the videogame industry is still on the steep part of its learning curve.

This paper describes an automated test strategy in terms of growing a test server from simple and accessible game components.

The goals of this test server were:

- **integration** testing—verify the game each time source upgrades
- **scenario** testing—scripting the game hero to spot-check a game effect
- **soak** testing—abuse the game at much higher intensity than normal gameplay
- **display** of test cases, results, and metrics in a rapid, flexible web site.

The resulting framework extends easily into new test categories.

Game Architecture

A videogame project is like a spider and her orb web. At the center, the spider is the game engine. Her labor pulls together and integrates the activities on each web spoke. They represent a game's visual & sound assets, special effects, design scripts, actors & props, user interface, physics, database, rendering & animation systems, and input channels. A spiral of threads cross-link each spoke, representing the interdependencies between all the game elements. The engine weaves many different kinds of threads together.

Growing a videogame requires repeatedly disturbing the spokes in that web. Small problems in one spoke might grow into large ones in another spoke. For example, an artist might change a level, and not notice a new hole in its geometry. The navigation systems then permit the game's hero to fall through the hole, and the physics system now obligingly drops our hero into the netherworld, unable to finish that level. Problems like these might ripple through the system before they get attention. Symptoms appear far from their sources. The spider in our parable must work very hard to maintain her web.

Modern software engineering practices revolve around automating tests for each requirement, feature, and capacity in a program. For example, developers who support the game asset pipeline can configure the art tools to detect holes relevant to this game's

physics engine. Tests isolate the source of ripples in one thread from affecting other threads, or disturbing the spider.

Functional Tests

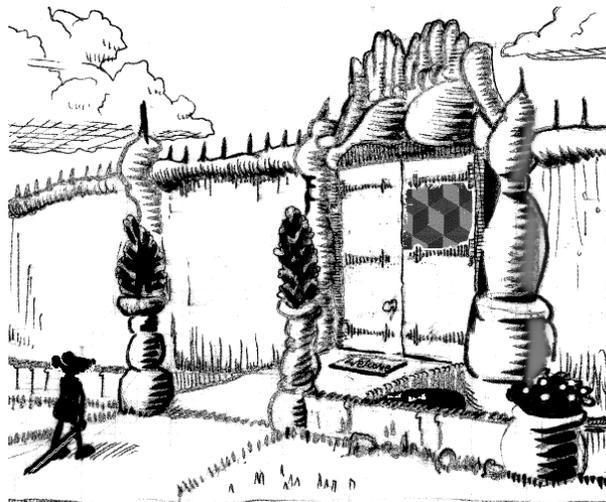
To test the complete web and spider—to test the assembled game in its releasable mode—configure a game to accept command line parameters that activate various features. Every module in the game should report any suspicious condition to an error log. For example, the character controller module should write an error message if the character’s altitude becomes excessively negative. This will indicate the character has fallen out of the world. Each test case will work by submitting its command line parameters to the game, and running it briefly. Then the test runner will read the log and escalate any error messages it finds.

(For a “legacy” game engine, written without unit tests, that technique has the added benefit that new error messages require no restructuring or decoupling of their host modules. And the logging system should conditionally compile away, so the “Release” and “Final” versions of the game can run with a high frame rate.)

So let’s take a given game, call it `Game.exe`, and force it to reveal some error messages.

```
Game.exe -levelname PalaceGate
```

That starts a game, and leaps our hero over the game’s plotline and directly to the Palace Gate level. Let’s suppose that the data file lacks gold filigree textures. That bug creates a big blank spot on the gate:



Tests must run unattended, and must test a situation for a fixed time period, permitting reliable metrics. That means `Game.exe` requires another departure from normal gameplay. The game must turn itself off, after playing for a given number of frames:

```
Game.exe -levelname PalaceGate -runforframes 100
```

Running that from a command line, targeting the Debug version of `Game.exe`, will generate a log containing an error statement, complaining about the missing texture. Jumping the game's hero all the way to the PalaceGate level, placing the actual gate within her or his line of sight, is enough to at least force the game to discover a texture is missing.

We are, of course, after bigger fish. Tests on the game's asset pipeline can easily detect the simpler issues. In our spider web parable, many complex issues arise from mismatches in the spiral of interdependencies linking all the spokes. Playing the complete game is our last, best chance to force relevant errors in all of those interdependencies to surface.

The game engine's unit of currency is the frame. The engine's job is to bond the various modules into an efficient whole that can swiftly render frames of animation. So tests exercise the engine in terms of its capacity for frames. Our game, under the above test, will iris-open at the Palace Gates. Our hero will gaze at them, in motionless awe, for a couple minutes. Then the game will shut down. Any incidental effects, such as background animations, sounds, or onrushing palace guards, will operate during the test.

`Game.exe` writes a log file; call it "`Log.txt`". When the test ends we read that file, extracting each error message. This requires a light scripting language. (We *don't* need one of the heavy compiled languages that game developers know and love!)

Our new program, `Test.rb`, builds that command line and passes it into a shell function, such as `system()`. The game runs, and turns itself off. `Test.rb` then scans the resulting `Log`, detects error statements, and collates them.

Scripting languages, such as Ruby or Python, can read text files using regular expressions. Suppose our game pushes this error message into the log:

```
Error - 09:46:02 - Art - missing texture "gold_filigree"
```

The log also contains warning messages, event messages, and generic information. Our `Test.rb` script would read each line into a variable called `line`, and detect errors like this:

```
if line =~ /^Error - (.*)/ then
  recordError($1)
end
```

Like many regular expression systems, the `$1` marker represents the part of `line` that `(.*)` matched. The systems inside `Game.exe` that write error messages may grow arbitrarily complex, but regular expressions can keep up with them, so long as they always follow the same format. The caret `^` mark ensures that no spaces appear before the "Error" tag, so non-error strings containing "Error - " in their middles cannot trigger a false reading.

The function `recordError()` writes the error to an internal record. Later functions format a complete report on levels, command lines and error messages:

```

case title: PalaceGate_100
  -levelname PalaceGate -runforframes 100
  09:46:02 - Art - missing texture "gold_filigre"
  09:46:02 - Art - missing texture "pewter_mosaic"
  09:46:02 - Art - missing texture "copper_handle"
case title: PalaceCourtyard_100
  -levelname PalaceCourtyard -runforframes 100
  09:46:02 - Art - missing texture "pewter_mosaic"

```

Many individual Log files would be hard to scan. That list is easier to scan, by only reporting the errors. To make the error list even easier to scan, write another summary, atop this one, containing only the unique messages from the errors:

```

Art
- missing texture "gold_filigre"
- missing texture "pewter_mosaic" (2x)
- missing texture "copper_handle"

```

Test.rb now e-mails that report to a mailing list.

Suites and Cases

For our next technical challenge, Test.rb needs a list of levels and command line parameters, so it can test more than just the PalaceGate. While we could store this table in any number of formats, including a Python or Ruby array declaration, XML has some benefits that will become valuable shortly. Here is "suite.xml":

```

<suite>
  <case title = "PalaceGate_100"
    levelname = "PalaceGate"
    command_line = "-runforframes 100" />

  <case title = "PalaceCourtyard_100"
    levelname = "PalaceCourtyard"
    command_line = "-runforframes 100" />

  <case title = "PalaceDungeon_100"
    levelname = "PalaceDungeon"
    command_line = "-runforframes 100" />

  <case title = "PalaceDungeon_500"
    levelname = "PalaceDungeon"
    command_line = "-runforframes 500 -playercount 2" />

</suite>

```

More than one case will test each game level and test level in our database. Some cases re-test a level with different options.

An XPath query expression of “/suite/case” selects a batch of those nodes. Test.rb takes an XML filename and an XPath string on its command line, like this:

```
Test.rb suite.xml "/suite/case"
```

Test.rb will assemble the levelname and command_line attributes to form complete command lines, pass them into system(), interpret the resulting Log files, repeat for each case, and e-mail the collated results.

To test only the cases that use the PalaceDungeon level, use the XPath like the “where” clause in a simple SQL query:

```
Test.rb suite.xml "/suite/case[@levelname='PalaceDungeon']"
```

Some test runs will select only one interesting case. Some will select batches of cases with some interesting common detail. The integration test runs will select all cases.

Extending the Tests

The effort of building a test server provides the scaffolding. Now we must make the tests cover more significant features than a motionless hero. And we must ensure that test failures are reproducible, so developers can fix the problems they uncover.

Script Tests

Most advanced applications use more than one language. For example, a data-driven application may use an Object Oriented language for its logical engine, and use a Declarative language, SQL, to manage and query a database. Similarly, many games use C++ for the hard engine layer, and Lua for a scripting layer. The distinction lies (roughly) between code that must be fast, and code that must change easily.

Our tests exploit this scripting layer, to trigger situations, and verify their outcomes. Suppose a test level contains a spawn point, a barrel full of dynamite, and our hero. A monster pops out of the spawn point, runs towards our hero, and passes close to the barrel.

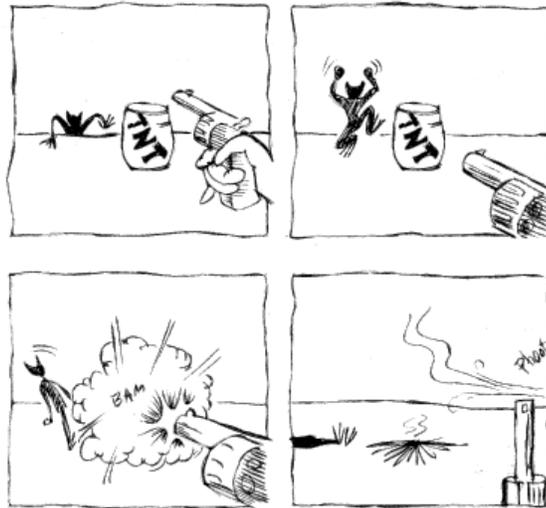
We write functions in Lua to manipulate our hero at a high level:

```
aimAt("Powder_Keg")

while 3 > distanceToKeg("Enemy_1") do
  delay(0.3)
end

Shoot()
takeScreenshot()
Assert(isDead("Enemy_1"), "enemy should die")
Assert(isDead("Powder_Keg"), "keg should be destroyed")
```

We aim at the keg, wait until the enemy passes it, and shoot the keg:



For a final flourish, `takeScreenshot()` records an image of the keg exploding and killing the enemy.

The `Assert()` statement pushes an error line into the log if its condition fails. More advanced assertion systems could reflect their source into their error logs:

```
Error - 09:46:02 - Script - false == isDead("Powder_Keg") -  
                                keg should be destroyed
```

Forcing Lua to reflect the source statements into its assertion output (`isDead("Powder_Keg")`) is left as an exercise for the reader.

That test may fail for any number of reasons. If a designer changes the enemy's spawning behaviors, it might not run close to the barrel on its way to our hero. If the physics system lost the ability to explode barrels, or if the barrel's damage radius changes, it might not affect the enemy. These scripts operate at the level of gameplay, matching users expectations. If a player depended on such a "barrel strategy" to defeat an enemy, and if the player executed the operations correctly but the trap failed, the player would feel disappointed.

To run such a test, insert the Lua script into the node contents of the XML:

```
<case title = "barrel_strategy"  
  levelname = "PalaceGate"  
  command_line = "-runforframes 100" >  
  
    aimAt("Powder_Keg")  
  ...  
    Assert(isDead("Powder_Keg"), "keg should be destroyed")  
  
</case>
```

`Game.exe` now must take command line arguments that specify our XML file and our target node's location (by XPath or `title` attribute). This permits developers to reproduce error situations by running `Game.exe` in their IDE, with the same command line arguments as `Test.rb` used. (To close this loop, `Game.exe` should print its build number and command line arguments into its Log.)

Script tests make excellent platforms to capture design bugs with tests. One can reproduce bug scenarios with tests that fail until developers fix the bugs. Such tests then prevent those bugs from regressing. Designers and programmers collaborate to ensure their Lua layer can query many game variables, such as the current level name, that the Lua design scripts won't need.

Script tests will constrain design, and prevent churn (where fixing each issue causes more issues) as the design grows and tunes. However, games entertain by providing elaborate emergent behaviors. One can't automate a test for every single situation that hundreds of thousands of potential users will encounter.

Random Tests

To soak each level in testage, add a new command line parameter:

```
Game.exe -randominput
```

This replaces the game's controller with a Mock Object controller that randomly operates a subset of its buttons. Whereas our hero in the test entitled "`PalaceGate_100`" merely stared in awe at the Palace Gates, a random hero could charge the gates, jump into the moat, run away, shoot at the sky, shoot himself, and generally have a good time.

These tests should run for a longer amount of time, to allow the hero to leap, fight, shoot, and squirm into situations that manual testers might never consider. That makes failures hard to reproduce. These tests must write to the Log sufficient trace statements to reconstruct the activities of the hero and the non-player characters. The tests must also write error statements if the random hero finds a way into a situation, such as monsters spawned out of order, or a *cul-de-sac* in the map, that a real player should not get into.

To characterize issues, read the Log and trace the hero's steps on the game map. At High Moon Studios I wrote a small script, called `Travel.rb`, to convert a Log into a sequence of Maya MEL commands. These drew cones at each point the hero visited. The trace of cones thru a map illustrated the random activities.

Triggers

Our tools, so far, include a game, command line arguments to run that game in little snips, a data file to select which snips to run, and a test runner to evaluate the file and test the game. The next step launches tests automatically, after each code build.

Large games require large build farms, to distribute the load of compiling all their source code and art assets. The last script or batch file in that process should send a signal to our integration server, triggering it to download the latest version of the game engine and assets, to test them.

Games typically require real display hardware to render their scenes. Many remote trigger systems, such as `telnet` or `rexec`, are closed to games when they evaluate commands as services, without access to a real Desktop and its display hardware.

The simplest link between two servers is the file system, so we use this to trigger test runs. Write a script called `Trigger.rb` that takes two command line arguments:

```
Trigger.rb runTests.bat touch_me.txt
```

The file `touch_me.txt` lives on another server, and each finished build changes its time stamp. The file contents are irrelevant. `Trigger.rb` waits for this change, then executes `runTests.bat`. That batch file contains “`Test.rb suite.xml /suite/case`”.

When developers or artists commit a code change, the build servers construct a new `Game.exe`. Then its last script calls `touch` on the test server’s `touch_me.txt` file. This spawns a test run, and e-mails the results to developers.

Lava Lamps

This test strategy—random tests for each level, and script tests for a swatch of gameplay scenarios, all running on a server—will catch many issues, and will inspire rapid response. However, the ultimate goal of testing is to prevent issues, not clean up after them.

Put `Test.rb` and a short suite of tests into a batch file, and configure this to work on any development workstation. The configuration should contain the standard path to the `Game.exe` executables that developers’ IDEs create, and the standard path to the asset files that artists change.

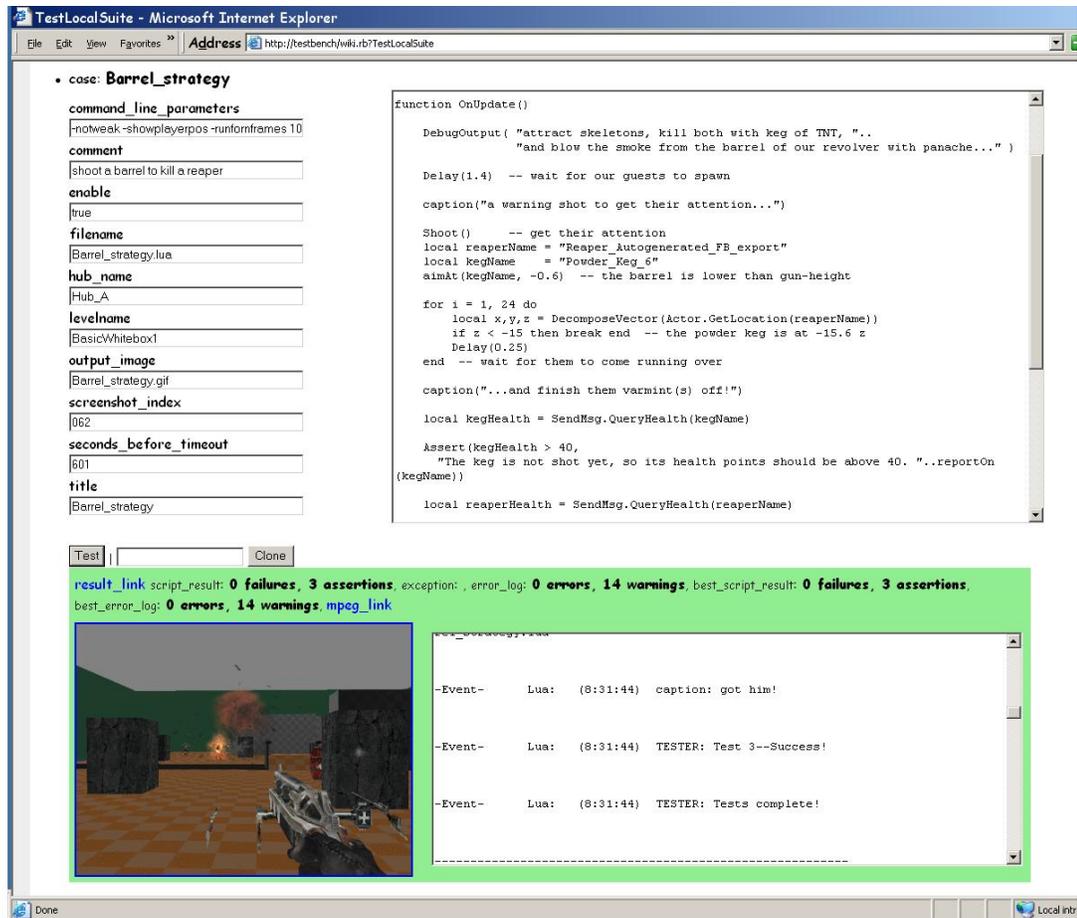
Developers and artists now run this batch file each time source changes. The batch file launches a web page with the test results in it. Any new errors must be due to a local change, and you can fix it before submitting the new code.

To broadcast the test status, without adding to developers’ crowded e-mail Inboxes, use an x10 controller and red & green lava lamps, raised on a platform so all developers can see them. Failing tests trigger a red lava lamp, motivating developers to track down faults and activate the green lamp.

Broadband Feedback

This strategy generates a huge volume of information—test cases, formatted test results, raw Log files, screenshots, etc. To store, format, access, navigate, and broadcast this much information, install a web site on your test server, and upload results to it as linked pages.

This is an experimental web site, `MiniRubyWiki`, designed to host test cases:



That web page requires some explaining.



1. That Wiki is why we wrote test cases in XML, not in Lua tables or in a proprietary file format. MRW is a Wiki that can transform XML with XSLT to produce an HTML form with a data entry field for each XML attribute. That web page contains a long list of cases. Each one is displayed as formatted, editable test information, and a summary of test results. That page edits the XML attributes, such as `command_line_parameters`, `enable`, `levelname`, `title`, etc, to power the test cases.
2. That is the text contents of the XML `<case>` node. Our `Test.rb` delivers this as a Lua script to `Game.exe`, to run this case's scenario.

3. Clicking that Test button will save the changes to this case, and launch a test run for that case. The button passes enough data to the server for it to execute a command line of `Test.rb localSuite.xml /suite/case[title='Barrel_strategy']`. The server refreshes the test results into the green output area.
4. Entering a unique title into that little field, and clicking the Clone button, will clone this test case. That permits rapid entry of many test cases, each with small variations.
5. This area's background color represents the case's passing status. Cases that produced error messages are pink, disabled cases are grey, and passing cases are green. The "result_link" leads to a read-only web page containing the complete test output for this case. The "mpeg_link" leads to an MPEG file recording a test run in progress.
6. The `takeScreenshot()` method created a full-sized image. We used ImageMagick to convert that to PNG format, and store it in a web page under the "result_link". ImageMagick can also scale an image, so we display a small thumbnail of it here. The screenshot reveals a cloud of smoke from the exploded powder keg, and the "Exploded" effect of the defeated enemy.
7. That is the Log file, poured into a read-only `<textarea>` for easy review.

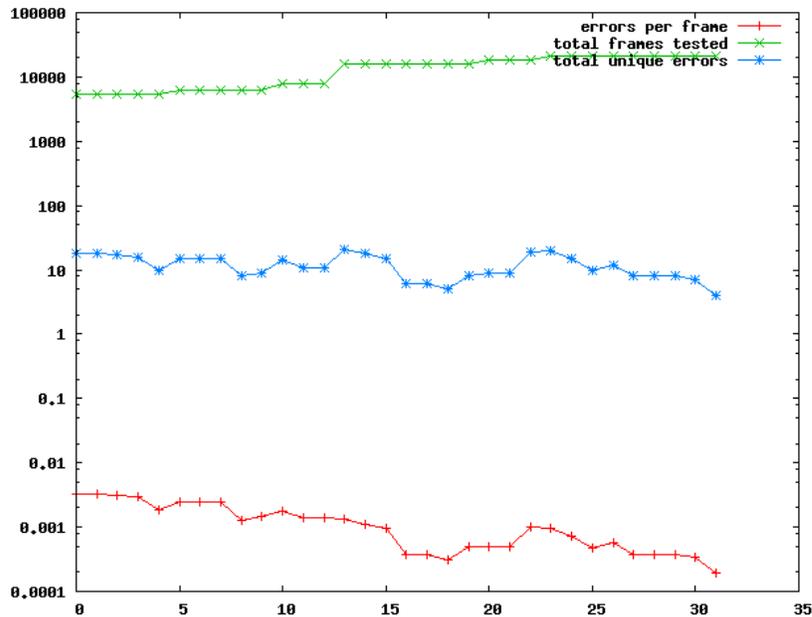
This dynamic Wiki, full of test cases, provides the backbone for all the other static HTML pages, the output of the test runs, that highlight and detail all aspects of our game's progress.

Metrics

A game engine's unit of currency is the frame. To track progress towards reducing the error count (and, indirectly, to reduce the overall issue count), add the total of all the `-runforframes` values in a test run. This total frame count represents the total volume of your videogame under test.

Next, collate all error messages in a test run, trim out their time stamps, and determine how many are unique. An error message that reoccurs periodically, across several cases, only needs one fix, so count it once. The total unique error count, for a entire test run, is an indirect metric of a game's health. However, if you increase the volume of testage, then the total frame count will go up, and the error count may go up. This is not poor health; it is progress, so we need another metric to directly rate health.

Divide the unique error message count by the total frames tested, to detect the odds that a frame produces a new, unique error:



Further tweaks convert the X axis into a time scale. The Y scale is logarithmic, because only large improvements, over orders of magnitude, matter in terms of progress. Disturbances in our metrics are like the Richter scale for earthquakes. As we add new test cases, and increase the duration of existing cases, the green line goes up. As developers stay up late, and work over the weekend, the blue line churns upwards. As developers constrain their code, and fix errors, the blue line goes down.

The red line—representing the odds that our game has issues—must go down over time. Our game is ready to ship after that red line reaches zero.

About the Author: Philip is an undiscoverable genius, and a self-proclaimed expert on many nebulous topics, including software engineering, GUIs, games, comics, and tests.