УДК 004.432.2

Krivulyak V. V., instructor

APPLCATION OF TDD TECHNOLOGY IN GAME DEVELOPMENT

Zaporizhzhya state engineering academy, department SoAS

Let us start by looking at the traditional programming process. Once programmer decides to implement a specific piece of functionality, he breaks the problem down mentally into smaller problems, and starts implementing each of them. He writes code, and compiles it to know whether things are going well. After a while, there might have been written enough source code to build and run the game and try to see if the feature works. Maybe the programmer might step in the debugger to make sure the new code is being called and doing what it's supposed to do. Once he is happy with it, he checks it into source control and moves on to another task.

Test-driven development (TDD) turns the programming process around: As before, programmer breaks down a problem mentally into smaller problems, but now he writes a unit test for that small feature, sees it fail (since you still haven't implemented anything), and then writes the code to make that test pass. Then he repeats the process with another, very small piece of functionality that will get him closer to the complete feature he is trying to implement. The cycles are very short, perhaps only a minute or two.

Let's have a more detailed look at the TDD cycle:

- Write a single unit test for a very small piece of functionality.
- Run it and see it fail.
- Write simple and short amount of code that will make the test compile and pass.
- Refactor the code and/or tests. Run tests and see them pass.

A unit test is a test that verifies a single, small behavior of the system (also called a developer test). Specifically, in this context, each unit test deals with something almost trivially small, which can probably be done in less than a minute [1].

There are “laws of TDD” that summarize the description of Test Driven Development into three simple rules:

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

You must begin by writing a unit test for the functionality that you intend to write. But by rule 2, you can't write very much of that unit test. As soon as the unit test code fails to compile, or fails an assertion, you must stop and write production code. However, by rule 3 you can only write the production code that makes the test compile or pass, and no more [2].

These are the benefits, listed roughly in order of importance:

- Better code design

What is the first thing you do when writing some code with TDD? You are forced to use that code in a test. That simple fact makes it so all code is created with the user of the code in mind, not the implementation details. The resulting code is much easier to work with as a result, and it directly solves the problems it was intended to fix. Also, because the code was first created by testing it in isolation from the rest of the code, you will end up with a much more modular, simpler design.

- Safety net

With TDD, just about every bit of code has some associated tests with it. That means you can be merciless about refactoring your code, and you can still be confident that it will work if all the tests continue to run. You can confidently apply obscure performance optimizations to squeeze that last bit of performance out of the hardware and know that nothing is broken. Similarly, chang-

ing functionality or adding new feature towards the end of the development cycle suddenly becomes a lot less risky and scary.

- Instant feedback

The unit tests provide you with instant feedback up to several times per minute. If at any point you thought tests should be passing and they aren't, you know something has gone wrong: not an hour ago, not ten minutes ago, but sometime in the last minute. In the worst case, you can just revert your changes and start over.

- Documentation

The best thing about unit tests as documentation: they can never get out of date. The tests are like little design documents, little coding examples, that describe how the system works and how to use it [1, 2].

Implementing TDD

Let's start with simple test that requires the least amount of effort: if we have a player and a power-up, and the player isn't anywhere near the power-up, his amount of health doesn't change. Full test looks like this:

```
TEST (PlayersHealtDoesNotIncreaseWhileFarFromHealthPowerup)
{
	World world;
	const initialHealth = 100;
	Player player(initialHealth);
	world.AddObject(&player, Transform(AxisY, 0, Vector3(10,0,10));
	HealthPowerup powerup;
	world. AddObject(&powerup, Transform(AxisY, 0, Vector3(-10,0,20);
	world.Update(0.1f);
	CHECK_EQUAL(initialHealth, player.GetHealth());
}
```

The macros `TEST` and `CHECK_EQUAL` are part of a UnitTest++ framework. The framework is intended to make the task for writing and running unit tests as simple as possible. It is a lightweight C++ unit-test framework that supports multiple platforms, is easy to adapt and port, and was created after using TDD in games for several years [3].

When writing unit tests, there are three main ways of testing your code:

1. Return value

Make a function call, and check the return value. This is the most direct way of testing, and it works great for functions that do computations and return the computed value. Because this is the easiest and most straightforward way of testing, we should use this approach whenever possible.

For example, a function called `GetNearestEnemy()` would be a perfect candidate to be tested this way.

2. Object state

Make a function call, then check that the state of the object or some part of the system has changed correctly. This tests that state changes directly, so it's also a very straightforward form of testing. For example, we could send an event of nearby noise to an AI in “idle” state and check that its state changes to “alert” in response. It might lead to larger interfaces than absolutely necessary, but it also sometimes shows that a class really should be split into two, and one of the classes should contain the other one.

3. Object interaction

Here we make a function call, and we want to test that the object under test did a sequence of actions with other objects. We don't really care about state, just that a certain number of function calls happened. The most common testing pattern for this situation is a *mock object*. A mock object is an object that implements the interface of another object, but its only purpose is to help with the test. For example, a mock object could record what functions were called and what values were passed, or could be set up to return a set of fixed values when one of its functions is called [1].

Using mock objects frequently could be an indication that the code relies too much on heavy objects with complex interactions instead of many, loosely-coupled, simpler objects.

TDD and game development

Applying TDD to game development has its own unique challenges.

Graphics, middleware, and other APIs. Probably the biggest barrier that people see to doing unit testing and TDD with games is how to deal with graphics. The first thing to realize is that graphics are just part of a game. It is common sense and a good software engineering practice to keep all the graphics-related code in one library or module, and make the rest of the game independent of the platform graphics API and hardware.

There are three different approaches, from most involved to least involved:

Third-party game engines. If dealing with APIs was not straightforward, working with a full third-party game engine that was not developed with TDD is even more challenging. After all, an API is just a list of classes or functions, and you have a lot of control over how and when they get called. Working with a full game engine, you might end up writing a small module that is fully surrounded by the engine code. If the engine was not developed with TDD, it can be very difficult to use your code in isolation or figure out how to break things up so they can be tested.

Randomness and games. Most games involve a fair amount of randomness: the next footstep sound you play can be any one of a set of sounds, the next particle emitted has a random speed between a minimum and a maximum, etc. As a general rule, you want to remove the randomness from your tests. A better approach is to separate the random decision from the code that uses it. For example, instead of having a `PlayFootstep()` rather split into `int ComputeNextFootstep()` which is just a random function call, and a `PlayFootstep(int index)`, which can now be tested easily. Another approach is to take control over the random number generator at the beginning of the test and rig the output so we know what sequence of numbers is going to come up (use mock object).

High-level game scripts. As a general rule, we find that if any other part of the game is going to depend on the code we are writing, then it's probably worth doing it with TDD and having a full set of unit tests for it. Otherwise, if it's a one-shot deal with the highest-level code, then it's probably fine without TDD. Also, code at that level is often written by designers in a game-specific scripting language, so TDD might not be a viable option [1].

Conclusions:

1. TDD is not just writing unit tests rather it is a development methodology, not a testing one. TDD ensures that your code does whatever you wanted it to do, not that it does it correctly.
2. The fact that it's hard to do pure TDD for UI does not mean you can't do TDD for almost everything else. The trick here is to follow the Single Responsibility Principle (SRP) and separate those kinds of code that you have to fiddle with, from those kinds that are deterministic. Do not put easy-to-test algorithms in with your UI. Do not mix speculative code with non-speculative code.
3. The problem with writing tests "after the fact" is that often the code is so coupled that it is hard to write the kinds of surgical tests that are most helpful. If you are writing the kind of code that is hard to test first, you should take care to follow The Dependency Inversion Principle (DIP), and The Open Closed Principle (OCP) in order to keep the code decoupled enough to test after the fact [1, 4].

References

1. N. Llopis, S. Houghton. Backwards Is Forward: Making Better Games with Test Driven Development / N. Llopis, S. Houghton, High Moon Studios – Game Developers Conference, 2006.
2. UncleBob's blog: The three rules of TDD. [Electronic resource]. Article, Oct. 2005 – access http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd / open access
3. UnitTest++ in brief [Electronic resource]: Unit testing framework for C++ designed for simplicity and portability. – 2011 – access http://unittest-cpp.sourceforge.net / open access
4. Stack Overflow: Topic Is test-driven development a normal approach in game development? [Electronic resource] – March 2009, access http://stackoverflow.com/questions/619634/is-test-driven-development-a-normal-approach-in-game-development / open access