Automated AI Testing: Simple tests will save you time

Malte Skarupke

1 Introduction

Game programmers have been slow to adopt automated testing. There are many reasons for that, one of which is that games fall under the category of code that is "hard to test." But inroads are being made: I have seen good test coverage for lower level libraries, and graphics programmers have figured out how to test their code well in recent years. (automated screenshot diffing with good visualization showing how two screenshots differ) I hope to include AI code in the group of code that is "easy to test," and I will explain how to do that in this chapter.

The main insights are that you can reproduce a lot of AI bugs with no more than two characters, that fibers are a great match for writing sequential code that takes many seconds to run, (such as AI tests) and that most complicated failures are a result of simple underlying causes that can be tested in isolation. I will illustrate how to write a simple testing framework that not only helps us write tests, it also makes debugging of problems easier.

2 Getting Value from Automated Testing

These days it's uncontroversial to say that automated test are an accepted best practice for programmers. An informal survey among my coworkers found that most of them want to write more tests, they're just finding it hard to do so. It's understood that all code lies on a spectrum from "easy to test" to "hard to test" with things like `std::vector` and `std::sort` being on the "easy" side and video games being on the "hard" side.

But AI code is actually fairly easy to test manually: Often all we have to do is set up a test level with a few AI characters and watch what happens. We will try to convert that ease of manual testing to an ease of automatic testing. In fact we will start with the simplest, easiest automated test.

If you're new to testing, you have to start testing the easy things. There is no shortcut to getting better at testing, and if you start with difficult tests, you will just get frustrated and give up. So if you are writing a small utility function or a container, (for example your own spatial data structure, which C++ has a shortage of) start with writing tests for those. Then as you get more experienced you should not move on to the things that are "hard to test," but instead you will find that more things now seem "easy to test" than did at first. Maybe that big, hairy class actually has an ad-hoc implementation of a container inside of it. If you pull that out, you can test the code. Making the class less hairy, and giving you a safety net to make further changes. The more tests you write, the better you'll get at this.

So the first lesson for getting value from automated tests is this: If something is hard to test, don't write a test for it. It's just not worth the time invested in writing and maintaining the test. Instead try to find a way to make it easier to test. But don't fret if you

can't. You don't need to write tests for everything.

To really get value out of tests though, we have to see that tests have a second dimension. Not only are some tests "easy to write" and others "hard to write", there is also a dimension of specificity: Some tests tell you exactly what is wrong, others are only able to tell you that "something broke." The first kind of test can significantly cut down on your debugging time. The second kind of test doesn't shorten your debugging time, it merely points out that there is a problem. An example for a test that's "easy to write but not specific" is a smoke test. A smoke test is usually a very simple test like "start the game, shut down the game, tell me if there were any errors." It's a good test to run automatically after every submit to version control. It finds problems surprisingly often. However all it can tell you is that there is a problem. It doesn't help you narrow down where the problem came from: If a smoke test tells you that you get an error on shutdown, that will take exactly as long to debug as if another person tells you that they got an error on shutdown. (that's why you have to run smoke tests on every submit, because that narrows down potential culprits)

With that, the best kind of tests are tests that are easy to write and tell you exactly what is wrong. The peak of that are unit tests: If a unit test for `std::sort` breaks, you know exactly where to start looking for the problem.

Since unit tests can be so valuable, I will talk a little bit about unit testing, but most of AI testing doesn't fit into unit tests, so after a brief detour into unit tests, we will talk about how to write AI tests specifically.

3 Unit Testing

Unit tests are the best kinds of tests because they are easy to write and point at a specific area of code for the source of the problem. I will keep this section short because I have found it hard to write unit tests for most of my AI code, but I still felt it necessary to include this section because unit tests can be so valuable.

In AI code I have found that unit tests are appropriate in small utility functions like matrix math, or utility classes like spatial data structures. As an example you can see a simple unit test in code listing 1. The test is written in the Google Test library, which is the unit testing library I use most often. [Googletest]

I've included summaries of the `VisionCone` and `VisionConeCollection` classes so that you can understand what's going on in the test, but we won't be too concerned with those specific classes. The idea is to have multiple vision cones, each of which has a differnet "vision score." But what's important about this unit test is that

1. It was fast to write. My rule of thumb is that it should be faster to write a test and to run it than it takes to test the same code in the game. If it takes me a minute to launch the game and to spawn AI characters on which I can test my vision code, then it should take less than a minute to write and run the test. If a test is the fastest way to run your code, you will write more tests. This is mostly a pipeline issue and you simply have to set up your dev environment to make it fast to run tests.
2. It runs fast. Google Test always shows how long a test takes, and this test always runs in "0ms." If a test of mine takes longer than 10 milliseconds, I either rewrite it or delete it. If you have too many slow tests around, you will not use tests as often.

Listing 1 Testing the `VisionConeCollection` class with a single `VisionCone`

```
struct VisionCone
{
    float cos_angle; // result of dot product
    float radius;
    float vision_score;
};
struct VisionConeCollection
{
    VisionConeCollection(std::vector<VisionCone> cones);
    float ScoreAt(Matrix4x4 head_matrix, Vec3 position)
const;
    // ...
};

TEST(vision_cones, single_cone)
{
    VisionConeCollection c({ { 0.7071f, 10.0f, 0.5f } });
    Matrix4x4 id;
    // in cone
    ASSERT_EQ(0.5f, c.ScoreAt(id, { 0.0f, 0.0f, 5.0f }));
    // too far away
    ASSERT_EQ(0.0f, c.ScoreAt(id, { 0.0f, 0.0f, 15.0f }));
    // behind head matrix
    ASSERT_EQ(0.0f, c.ScoreAt(id, { 0.0f, 0.0f, -5.0f }));
}
```

This simple example will be all I use to illustrate how to use unit tests. The earlier that you start to write tests for a piece of code, the better the interfaces of that piece of code will be for testing. From here it's easy to extend the tests by adding more cases. Or let's say you implement different behavior for the y-axis than for the xz-plane. You should add a test for it to get faster iteration times. Or if you want to add hysteresis so that visibility doesn't flicker on and off when the player is standing right on a threshold, add a test for it. You can also add edge cases (like what if a character asks if it can see itself) and be sure that they never break. As the feature grows, the tests grow with it.

4 Confidence for More Complex Tests

I can sense your skepticism about the previous example: AI vision rarely breaks because you introduced a bug in the vision cone code. Instead AI vision probably breaks because AI characters are now wearing helmets, and the raycast is hitting the helmet. Or because an artist introduced a new glass model and forgot to set the see-through flag on it. And even if AI vision breaks, that's an easy bug to fix. The hard bugs are results of several characters interacting in unexpected ways. How are you possibly supposed to test all that?

To start with remember the first lesson: We're not going to write tests for things that

are hard to test. But if we're only going to write easy tests, how much value are we going to get? Research from other fields suggests that we'll get a lot of value.

In a study on concurrency bugs, Lu et al. found that 96% of concurrency bugs can be reproduced by enforcing a certain execution order of just two threads (Lu 2008). Similarly 97% of deadlock bugs are the result of two threads waiting for two resources. In a study on distributed computing, Yuan et al. found that 84% of bugs are reproducible with just two computers. 98% are reproducible with three computers (Yuan 2014). I claim that something similar is true for AI code: Most AI bugs are reproducible with two characters. I don't have the percentage numbers on how many bugs exactly can be caught with simple tests, but the numbers from the other fields should give us some confidence that it's a good amount.

I want to clarify that I'm not talking about simple bugs here. In a talk about Ding Yuan's study on distributed computing, he emphasizes that a lot of the investigated bugs were really complicated. But when you trace the bug all the way back, you often find a root problem that would have been easy to detect. Similarly in AI we often see confusing behavior, and we have to look at several frames of history to find out that the AI is acting weird because of a simple root cause. Maybe it failed to play a certain animation, or the animation failed to move the character. Or maybe as soon as it got into a vehicle it stopped seeing its target because the raycasts collide with the vehicle.

When that happens wouldn't you rather have a test that verifies that the AI can move to where it's supposed to be able to move to? Or that an AI can still see when it's sitting in a vehicle? If one of those tests breaks, the problem is a lot easier to fix than if you're just seeing unexplained weird behavior in the game. Even if the problem ended up being caused by something else, it's still useful to quickly rule out whole categories of problems: "it's probably not an AI vision problem because the test for that is currently green."

5 Testing in a Game Engine

You probably already have a bunch of manual tests for your AI. Test levels with gray boxes where you can spawn a few characters and observe them in simple scenarios. You probably also have a "free-fly mode" (or "ghost mode") in which no player character spawns and you can just observe the AI. All we have to do is run those manual tests automatically and detect whether they behave the way we want or not.

How to jump into a test level and set the game into a mode where you can observe your AI will depend on your engine. You should set it up to be able to launch tests from a command line argument, or using an in-game command. The command line argument is for launching the test on an autobuilder. The in-game command is useful to quickly launch tests on other people's computers to verify if something works or to show bugs to other programmers. But then what does writing a test actually look like?

After a few misguided attempts, I realized that fibers are a perfect match for writing the kind of logic that a test needs. The main benefit is the ability to suspend a fiber at any point in the function. That makes it possible to write similar checks as in Google Test, but to give the engine time to fulfill the criteria.

As an example here is a very simple test: I spawn two characters of opposing factions. One of them has a weapon, the other is brain-dead. The test simply asserts that the

character with a weapon will defeat the brain-dead character.

Listing 2 A test that waits for 30 seconds for a character to die

```
jt::TestResult RunTest(jt::TestRunner& test_runner)
{
    Character* bd = GetObjectByAlias<Character>("braindead");
    JT_ASSERT(bd != nullptr);
    return test_runner.WaitUntil([&]
    {
        return !bd->IsAlive();
    },
    "Waiting for the character to be killed", 30.0f);
}
```

In this test most of the setup is done in content. The two characters are configured in the test level. I don't have to refer to two characters in code because I only care about what happens to one of them. I get that character through it's "alias" which is an engine-specific feature to refer to objects from code or script. Your engine probably has a similar feature. The `TestRunner` is a small wrapper around the fiber that's used internally. It provides one important function: `WaitForOneFrame()`. All other test functionality can be built on that function. The `WaitUntil` function I use in the example just calls the condition-lambda and if it's false, calls `WaitForOneFrame`. I pass a message into the function to display on the screen while the test runs. That message will also be displayed on the autobuilder if the test fails in this step. The final argument is a time-out in seconds. If after 30 seconds the lambda still returns false, the test fails. The other possible place where this test can fail is in the `JT_ASSERT` macro, which simply returns `TEST_FAILED` if the given condition is false.

Before we get into how this is implemented, I want to point out a few features that make this easy to implement. First, the test doesn't have to establish preconditions. The `RunTest` function is actually a virtual function in a class, and there is a second virtual function called `GetPreconditions`, which the test framework calls before calling this function. Since many tests require a similar setup, I wrote the code for that once. In `GetPreconditions` you can indicate a test level that you want to have loaded, the position of the camera, whether you want to have a player character or be in ghost mode, and you have the ability to turn off some global engine features. (such as spawning of traffic on roads) The test framework then ensures that all your preconditions are true before it runs the test, so that the `RunTest` function really only contains what's necessary for the test. The precondition code is engine-specific, so I won't go into it further.

The second important feature is the ability to specify a time-out for a condition. In a normal testing framework like Google Test you only want to assert that something is true after a given sequence of calls, but when testing AI you often can't be that precise. Depending on the length of animations and on details of your behavior tree, things can take a few frames more or less. So instead I found it useful to check whether something becomes true in less than X seconds.

The third important feature is that this is written in normal C++ and that I have

access to all of the normal functions of the engine. I believe that it is very important that tests are easy to write, and that is only the case if I can access a function from a test without having to do any extra steps. (such as exposing the function to a scripting language)

To start implementing a test framework, you just need the ability to implement the magic `WaitForOneFrame` function. If you have that, you can implement all other utility functions that you may need on top of that. As an example, I sometimes need the ability to get an object by its alias (as in the test above) but the object may be not have spawned yet, and I just want the test to wait until the object is spawned. That is easy to implement as a utility function: Try to get the character and call `WaitForOneFrame` if it doesn't exist yet. If the character doesn't exist after X seconds, I fail the test.

So how do we implement the `WaitForOneFrame` function? I implemented it using a fiber, but you could also implement it using a thread. When using a fiber it's a simple wrapper around the yield function provided by your fiber library. (for example it's simply called `yield` in boost::coroutine)

When using threads you can implement this using two semaphores: The test runs in its own thread, the rest of the game gets controlled by a main thread. At a good point in the frame, when it's safe for the test thread to access and mutate global state, the main thread signals on the first semaphore that the test thread should run. It then waits on the second semaphore. The test thread was waiting on the first semaphore and runs now. In `WaitForOneFrame` it signals the second semaphore and waits on the first semaphore again. With that the two threads take turns. All other threads are sleeping while this is happening. They get woken up by the main thread when it is done waiting.

Making the main thread and all other threads sleep while the test thread is running slows down the engine a little bit, but I haven't found that to be a problem. It's a price I'm willing to pay to make the tests reliable. I simply don't have to worry about which state I can access or mutate because I know that nothing else is running.

6 Pausing, Canceling, Restarting and Repeating a Test

Running tests with timeouts comes with a few problems, all related to time. The first is that the timeout can make things hard to debug. In my example above of giving one character thirty seconds to kill a different character, what happens if something goes wrong and I want to debug the problem? I open our developer menu, turn on some debug drawing look at the internal state of the AI, and before I know it the thirty seconds are over, the test fails and everything de-spawns. Oops. So the first additional feature you'll want is to be able to pause the test. This simply means not calling into the test fiber. The game simulation continues to run. So in my example if I pause the test, it merely pauses the time-out. (but the character continues fighting) But if the test had multiple steps, the test wouldn't advance to the second or third step of the test as long as it's paused. The pause feature is toggled using a global variable that's easy for me to change from our debug menu.

The second problem is that if you accidentally launch the wrong test, you have to wait 30 seconds or a minute for the test to finish. The solution for that is the ability to cancel the test. For that I changed the `TestResult` enum to have a `TEST_INTERRUPTED` value. The function `WaitForOneFrame` will now return that value when I issue the

command to cancel the test from our debug menu. There are two subtleties with this:

The first sublety with interruptions is that you have to make sure that the return value from waiting calls is always handled the same way, so that the test returns should it be interrupted. It might be possible to implement that using exceptions, but we (like many game developers) compile without exceptions. The second best approach I have found is to implement a macro called `JT_CHECK` that returns on interrupts or failed test. Any call to a waiting function has to be wrapped in `JT_CHECK`. Here is what the macro looks like:

Listing 3 Implementation of the `JT_CHECK` macro

```
#define JT_CHECK(...) do {\
::jt::TestResult wait_result = __VA_ARGS__;\
if (wait_result == ::jt::TEST_INTERRUPTED\
          || wait_result == ::jt::TEST_FAILED)\
    return wait_result;\
} while(false)
```

As usual in C++ macros, this isn't pretty. (sorry) To use this in the test above, instead of returning the result of the wait function manually, I would wrap that call in a `JT_CHECK` macro. It doesn't make a difference for a one-step test like that, but for a test consisting of multiple steps, every line that can wait has to be wrapped in this macro.

The second problem comes directly from this macro and from the `JT_ASSERT` macro: The test can return at any point. What this means is that all the state that the test creates has to be wrapped in RAII structs so that the state gets cleaned up at the end of the test. For example if the test spawns characters, they have to despawn when the test gets canceled, so there needs to be a RAII wrapper that despawns the character in its destructor. I avoided that in the test above by doing most of my setup in content, not in code, so the test framework handles this for me by unloading the test level.

With that out of the way, we are able to interrupt tests. Which immediately allows us to implement the next feature: Restarting of tests. It's an option in our debug menu that cancels the current test and immediately starts it again. This feature makes working with tests a pleasure because I can very quickly test a certain situation over and over again. If a problem happens one out of ten times, I create a test with the initial conditions of the problem, and restart the test until the problem occurs.

The final feature is the ability to run a test on repeat so that when it finishes, it automatically restarts itself. This is also now trivial to implement and is also done to reproduce rare issues.

With these tools the testing framework is not only a useful tool to find issues, it also saves us time by giving us additional debugging features such as the ability to run a scenario on repeat until a problem occurs.

7 Simple Tests

We finally have everything in place to start writing tests. So what kind of tests should we write? Earlier in the chapter I said we should test things that are "easy to test" and we should

write tests that tell us specifically what is wrong. An easy test might be to set up a fight scene between ten characters of one faction on one side and five characters of an opposing faction on the other side. Then we assert that the side with ten characters wins. But even though that's an easy test to write, it's not very specific. A test like that can fail in a million ways. Also who says that five characters can't win against ten characters? What if one of them gets lucky with a well-placed grenade?

So we want tests that tell us much more directly what is wrong. The test of "one character with a weapon against one brain-dead character" that I used as an example above is an improvement, but we can be more specific still.

A good test to start is to test the perception of characters. Place two characters in an open field and assert that they can see each other. Place a wall between them and assert that they can not see each other. Then do the same thing for a pane of glass. Then put one of the characters in a vehicle. Then put one of the characters behind a mounted gun on a raised platform. Add more edge cases as you encounter them.

We can also make that test dynamic: Put two characters in the open and assert that they can see each other. Then make one of them walk behind a wall and assert that they can no longer see each other. This test can fail even if all the previous tests succeeded, for example if you have a bug in the logic for when to update vision information. (because you don't want to do a raycast every frame) Next, if characters have an "investigate" mode you can add a third step to the test by testing that after one character has disappeared behind the wall, the other character "investigates" and catches up with the first character. That test can be seen in listing 4. I will use that test to illustrate a couple patterns that I often use.

The first pattern is that I have one character with a custom behavior tree. That tree is written specifically for this test. All it does is it waits for the global event "start_walking" and then walks to a predetermined spot behind a wall. The other character is a completely normal enemy as it would appear in the game. Since I only care about the behavior of one of the two characters in this test, I like to have complete control over the other one. It also makes the test code easier because all I have to do is send a global event.

With that we can look at how the test works: Before the test starts, we enter a test level with two characters and a wall. I assert that the two characters exist, then I set the faction of the custom character to the `PLAYER_FACTION`. This is a second pattern: Our AI has different behavior in AI-vs-AI fights than when fighting the player. They never enter the "investigate" mode when fighting other AI. They only do that when fighting the player. So to test the investigate behavior, we simply set the faction of the other character to the player faction. In all of our AI logic we only use the faction to determine whether an enemy is the player or another AI, so if I set the faction, I get player behavior.

Finally we see that the logic is actually quite simple: I wait one second for the characters to see each other. Then I give the signal to start walking. Then I wait ten seconds until the characters can no longer see each other. At this point the normal characters should enter investigate mode. I now wait 30 seconds until they can see each other again. Since each of these wait is an upper bound, the test actually runs faster.

Even though this test is testing a complicated sequence, it's actually very simple in the implementation. For example if there is a bug in the "investigation mode" then the last step would time out after 30 seconds and the test would fail.

Listing 4 A test to check whether a character correctly investigates

```
jt::TestResult RunTest(jt::TestRunner& test_runner)
{
    Character* c = GetObjectByAlias<Character>("custom");
    Character* n = GetObjectByAlias<Character>("normal");

    JT_ASSERT(c != nullptr);
    JT_ASSERT(n != nullptr);
    c->SetFaction(PLAYER_FACTION);
    auto can_see = [&] { return n->CanSee(c); };
    JT_CHECK(test_runner.WaitUntil(can_see,
    "Waiting for normal to see custom", 1.0f));

    SendGlobalEvent("start_walking");

    JT_CHECK(test_runner.WaitWhile(can_see,
    "Waiting for custom to walk behind the wall", 10.0f));

    JT_CHECK(test_runner.WaitUntil(can_see,
    "Waiting for normal to investigate", 30.0f));
    return jt::TEST_SUCCEDED;
}
```

Your tests shouldn't get more complicated than this. I will list a few more tests to get you started, then I'll explain how to select your own tests to write. First, test movement. If your characters can climb ladders, test that they can climb ladders. If your characters can enter vehicles, test that they can enter and exit vehicles, and that they can drive vehicles where they're supposed to be able to drive to. You can also repeat my first example test for all kinds of situations: Make sure that a shooting helicopter can kill a brain-dead opponent. Make sure that a character at a mounted gun can kill a brain-dead opponent. Then test that in a combat scenario, all characters enter cover within X seconds. Test that all characters have shot at least once within Y seconds.

Overall I don't recommend writing too many tests though. I recommend writing tests for one of two reasons only: 1. To more quickly iterate on a new feature. 2. To reproduce a bug. Those two reasons ensure that I have a small set of tests that runs somewhat quickly. Sometimes people try to write really slow tests like "spawn every vehicle and check for errors." Those kinds of tests are just an invitation for lots of maintenance work. You will certainly run into situations where one vehicle doesn't work, and when you tell the responsible person they answer "oh yeah we know. It's only used in one mission, and that mission has bigger problems right now. We'll fix it before alpha." (where alpha is a year away) And now you have a broken test in your system that just always gets in the way. So don't be too aggressive about your tests, and try to write specific tests.

The other open question is what to do about more complicated situations. Like what do we do if we have squad behavior for up to four characters? I would say that if something seems complicated to test, leave it alone for now. Tests don't solve everything. We don't lose anything by not writing a test for this. But we may lose something if we add an overly

complicated test that requires lots of maintenance. So leave it alone, and just debug it the old-fashioned way. Maybe you will come up with nicely targeted tests later that can reproduce certain problems. Until then you still have a bedrock of simple tests that you can rely on. Don't test things that are hard to test. And with experience you will be able to make more things easy to test.

8 Conclusion

I hope to have moved AI code from the things that are "hard to test" to the things that are "easy to test." The biggest thing I didn't show was the code for ensuring preconditions, but that code is mostly just code for loading test-levels. (or teleporting to test islands and waiting for streaming in our open world engine) Otherwise the tricks of using fibers to be able to write the logic in sequence, and the trick of using time-outs instead of asserts were most of what was required to make AI testable. That, and the realization that most AI bugs have simple causes that can be reproduced using two characters.

The testing framework that we ended up with not only helps us reproduce bugs, (and catch them early should they come back) it also saves us time while debugging issues by providing the pause, restart and repeat features. Also since all the tests are written in C++, we can call normal functions and we can step through the code using normal debuggers. While I am not yet able to write good tests for the most complicated AI bugs, I have caught many bugs with relatively simple tests. And the longer I'm doing this, the better I get at coming up with simple tests that catch sources of complex problems.

9 References

[GoogleTest] https://github.com/google/googletest

[Lu 2008] Shan Lu, So yeon Park, Eunsoo Seo and Yuan yuan Zhou. 2008. Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics. In ASPLOS '08

[Yuan 2014] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation