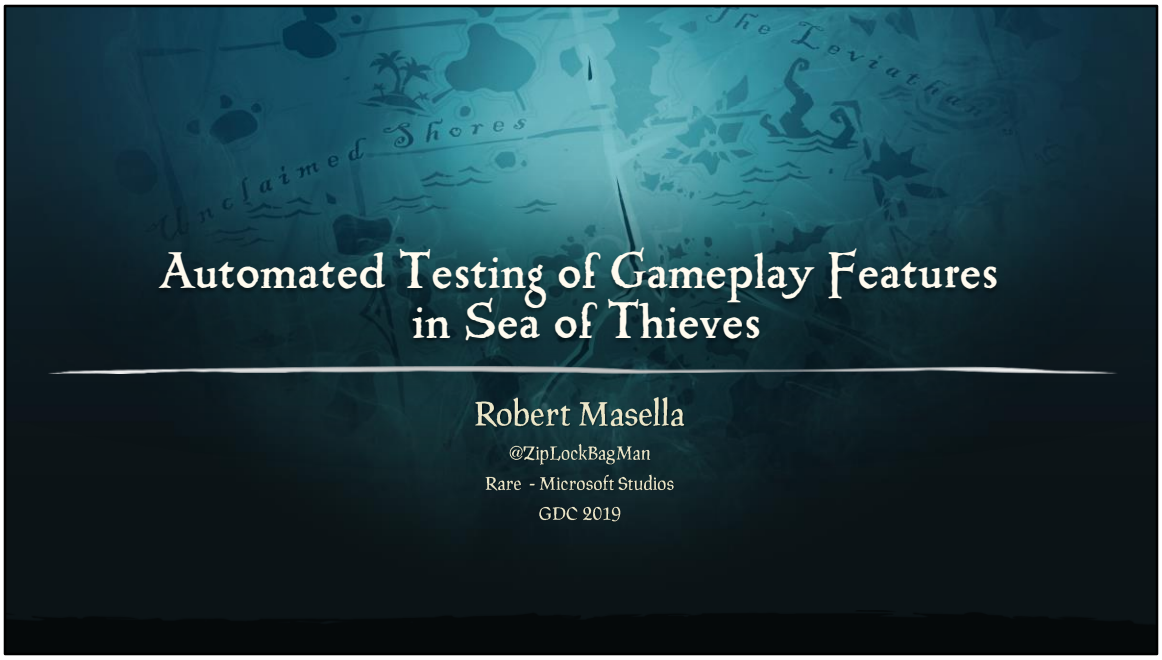
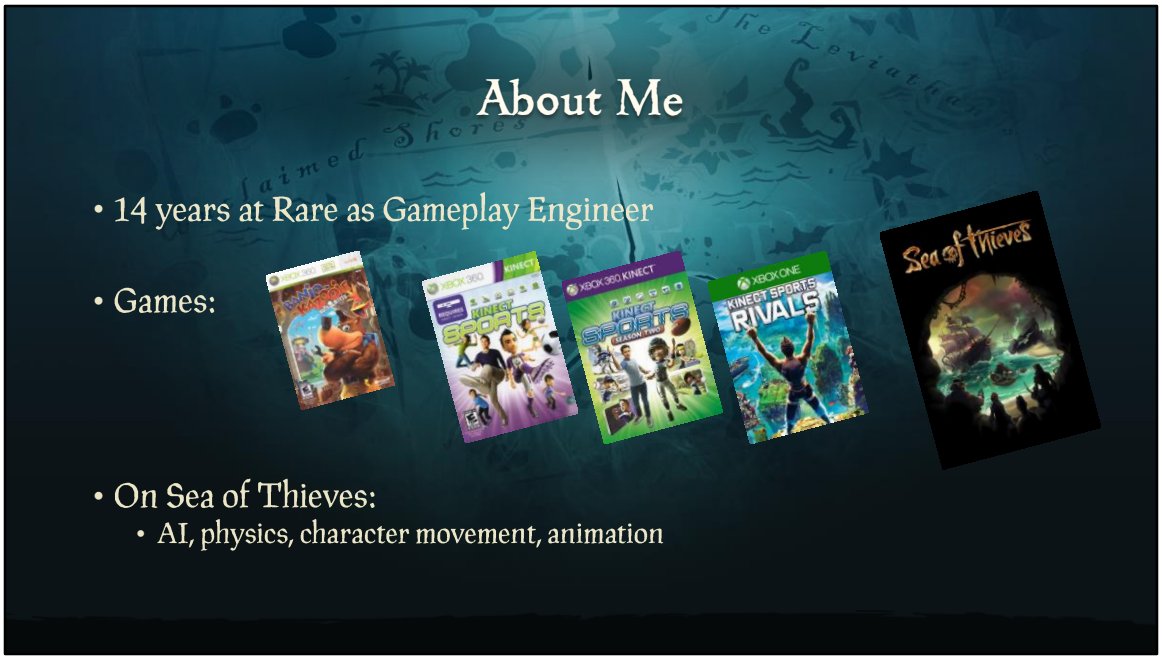


Good morning everyone

Thanks for coming. Can I remind everyone to silence their cell phones and fill out their surveys after the talk.

I'm Robert Masella and I'm a software engineer at Rare.

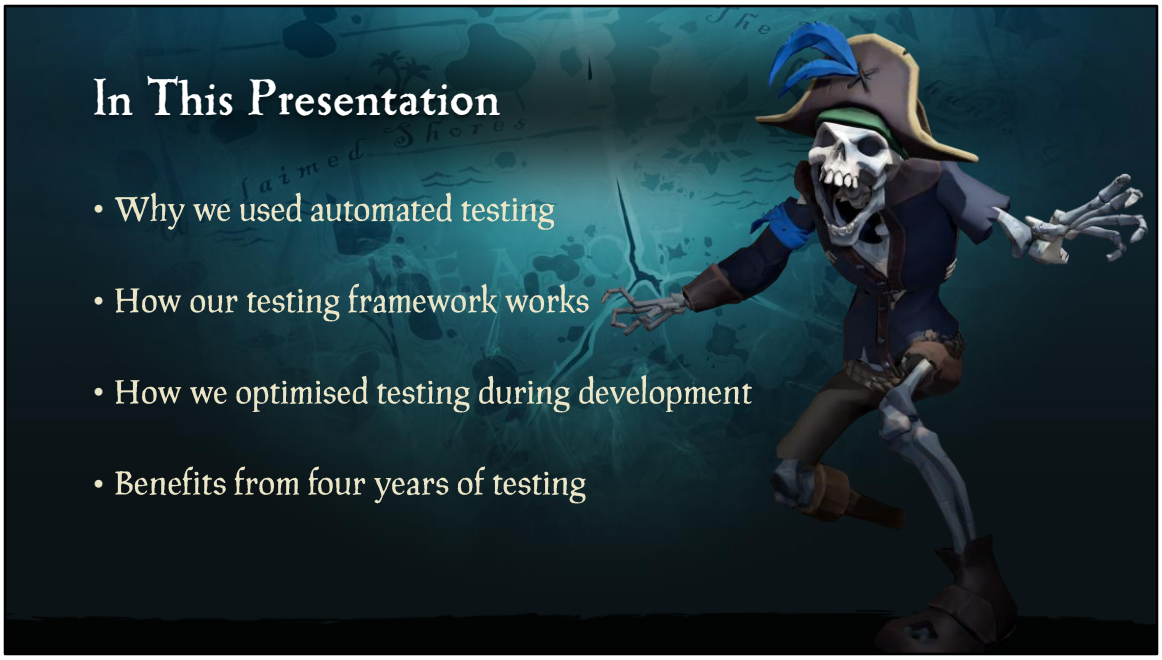
Because we'd had such bad experiences with testing on our previous projects, when we started work on our latest project Sea of Thieves, we decided to completely change our approach. Instead of relying on manual testing, we used automated testing to check every part of the codebase, including gameplay features, which are often notoriously tricky to test. In this talk, I'm going to walk you through our approach, our learnings and how we benefited from it.



-- First a few quick details about me
- I've been a gameplay engineer at Rare for almost 14 years. For those who don't know Rare, we're a Microsoft first party studio with a long history of game development. We're based in central England and currently have around 200 people on staff.
- The notable projects I've worked on at Rare are Banjo Kazooie Nuts and Bolts, all three of our Kinect Sports games and now for the last four years Sea of Thieves
- At Rare, gameplay engineers do a bit of everything and tend not to specialise too much. This meant that on Sea of Thieves I worked on a variety of gameplay areas, including AI, physics, character movement and animation.

-- For those who aren’t aware of sea of thieves and what kind of game it is, it’s a multiplayer open world game for Xbox and Windows, where players can take the role of pirates. They get to cooperatively sail their own pirate ships around the world and do all the things you expect pirates to be able to do, like follow maps to find treasure, steal treasure from other players and fight AI enemies like skeletons. Or they can just hang around drinking grog and playing instruments, its completely up to them.

-- Since we released the game a year ago, in fact a year ago yesterday, we're added multiple new updates that expand the game even more.
Some of those new features include skeleton AI ships for the player to fight, a new set of volcanic islands to explore, underwater statues to find and a megalodon shark that can randomly appear and attack player ships.

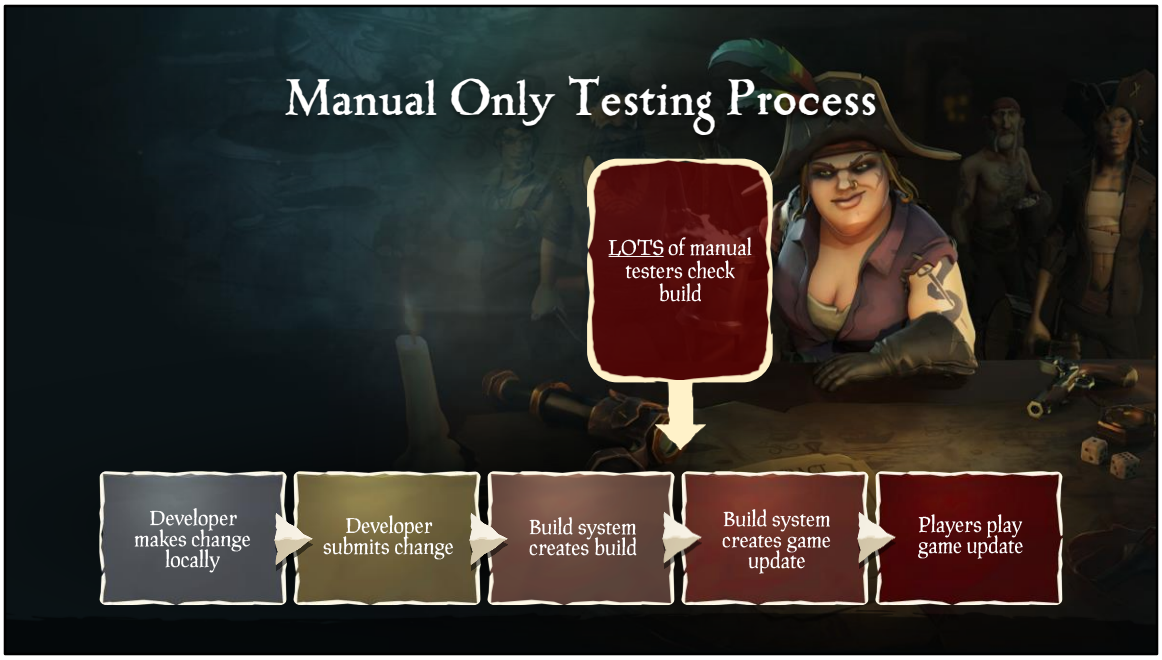


-- So in this presentation, I'll be talking about:
- Why we thought automated tests would be a good fit for Sea of Thieves.
- How we created our automated tests within our own test framework, and how we integrated tests into our build process.
- How our automated testing was optimised during the production of the game.
- Finally I'll talk about the benefits we got from automated testing over the four years the game has been in production.

-- so before I get into specifics about our tests, why did automated testing make sense for Sea of Thieves.

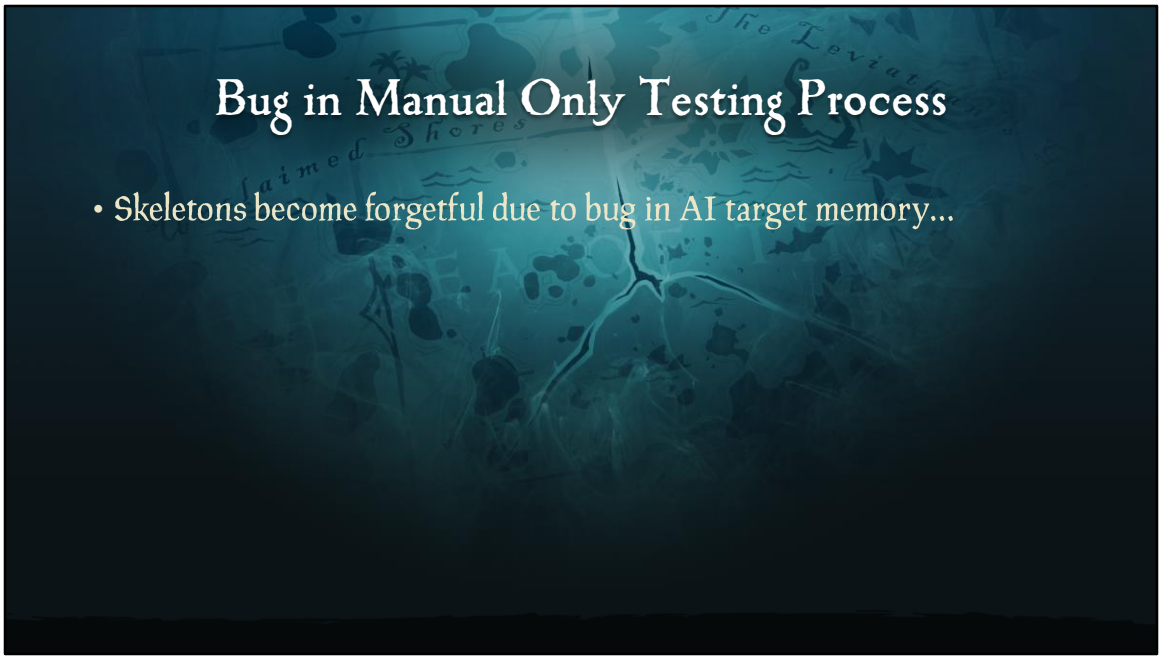
-- Sea of Thieves was going to be a very different game to any Rare had made before, and some of those differences meant testing the game would be a significant challenge.
- The first big difference was the type of game Sea of Thieves was. Sea of Thieves was an open world game, with very open gameplay that allowed players have complete freedom on how they played the game.
- This openness meant that the scope for bugs in the game was very high. Testing everything would be a big challenge, particularly when there were so many interactions between gameplay features that would need to be checked.
- Next, Sea of Thieves was the first game Rare have worked on that really fits the game as a service model. We were planning to continually evolve the game, responding to player feedback and adding new ways to play. In fact, the team working on Sea of Thieves now is bigger, than it was at the time we released the game last year.
- The risk with a game that is constantly changing is that new updates are likely to break previously implemented features. Inevitably this constant churn will raise the chance of more bugs appearing in the game, affecting player's enjoyment.
- The aim for Sea of Thieves was to respond to player feedback quickly, with an update able to be released with a week's notice if necessary.

- On our last game Kinect Sports Rivals, turning around a game update required about two weeks for it to be verified. The same process would've been even longer for a much more complex game like Sea of Thieves. If we carried on using our old process but at a faster rate, we wouldn't have enough confidence that a game update wouldn't have serious problems in it when it went to players.



-- To show you what our testing process looked like previous to Sea of Thieves, here is a timeline of the development stages that a change goes through before it reaches players,

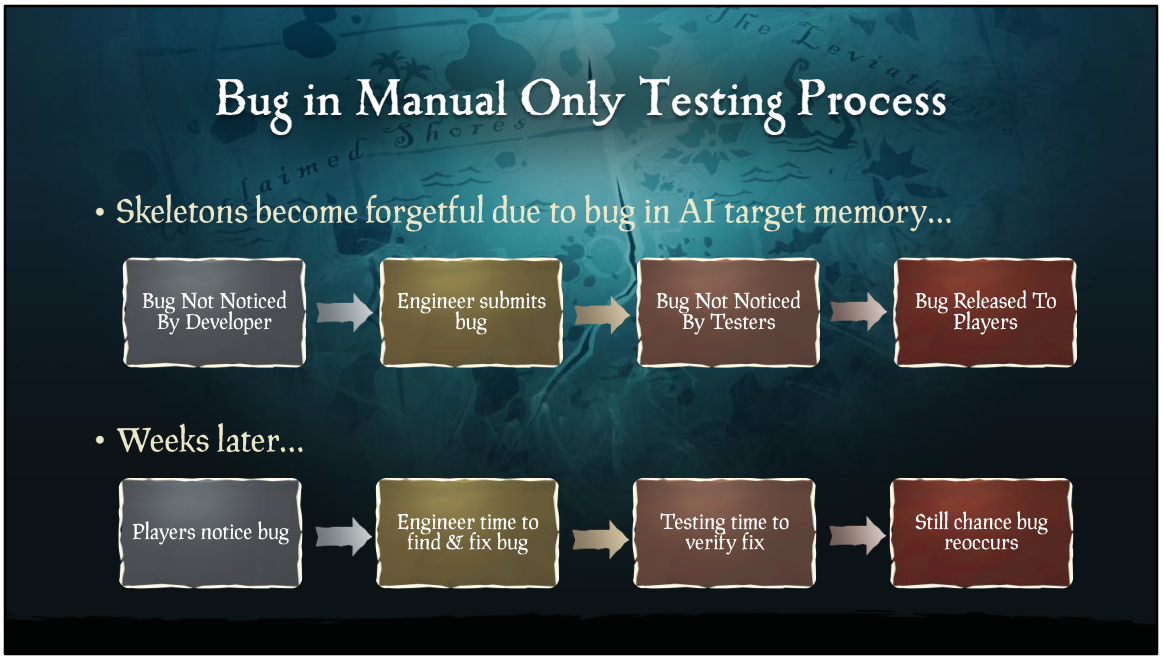
- And on this timeline we only had one real point of testing, where we relied on a large manual testing team checking a build with the most recent changes, to find all the bugs. These bugs are then hopefully all squashed before the build is released to players. This process however, is a slow and unreliable way to find issues with the game.



-- To show the failures of this process, here's an example of a bug being added to Sea of Thieves and then eventually fixed, when no automated tests were in place to catch it.
- In this bug, an engineer has made a change to the game that meant the skeleton AI's target memory was not working correctly. That engineer was me, and this was actually a bug I added to Sea of Thieves during the beta period.

-- Here’s a video showing what the bug looks like in game:
(During) Now when a player goes around a corner and breaks LOS with an AI, the AI instantly forgets the player, stops attacking and wanders off instead. This is obviously not what players would expect.

-- Whereas we would of course expect something like this to happen instead:
(During) Now the AI will remember the player for a reasonable amount of time and follow the player around the corner to carry on attacking.



-- So lets look at the process that this bug went through.
- First of all the developer (i.e. me) didn't spot this bug before it went into the build. How did this happen? Well, I didn't test the game enough. I observed the skeletons in action before I submitted, but that was on an open test level without obstacles where the problem didn't occur.
- So I submitted the code with the bug, then the build is verified by our manual testers.
- There may be a small chance that testers notice the AI being less responsive, but in this case the bug was subtle enough that the testers don't notice anything different about the AI.
- So the bug ends up in a build that's released to players.
- Now lets look at how the bug was found and then fixed, a few weeks later.
- First our community team, who are collecting our beta feedback, start seeing forum posts saying that the skeletons seem dumber and less aggressive than before.
- Eventually this gets flagged as a bug for the engineers to look into, which takes them away from the feature work they've currently in the middle of. Finding the issue is difficult as thousands of changes were made as part of that game update, so engineers can't just narrow the bug down to a few suspect changes.
- After the issue is discovered and a fix is submitted, our manual test team then have

to play the game and verify that the fix works, before a new build is released to players.
- And after all that, there's *still* a risk that the issue could re-occur if another engineer accidentally breaks the game in a similar way, which will cause the whole process to be repeated again.

-- To avoid this long scattershot process of locating bugs, fixing them and checking they don't reoccur, we need to regularly check that this issue isn't in the current version of the game.

- For this AI example, we could get a tester to run a scenario like the one in the video everyday, getting a skeleton to chase them around a corner and checking that it remembers to follow them. That wouldn't take too long, but if we wanted checks for a whole game of the size of Sea of Thieves we'd have thousands of these scenarios that needed testing. Now these checks would take so much time that they would become a manual tester's entire job, and a pretty mind-numbingly boring one at that.

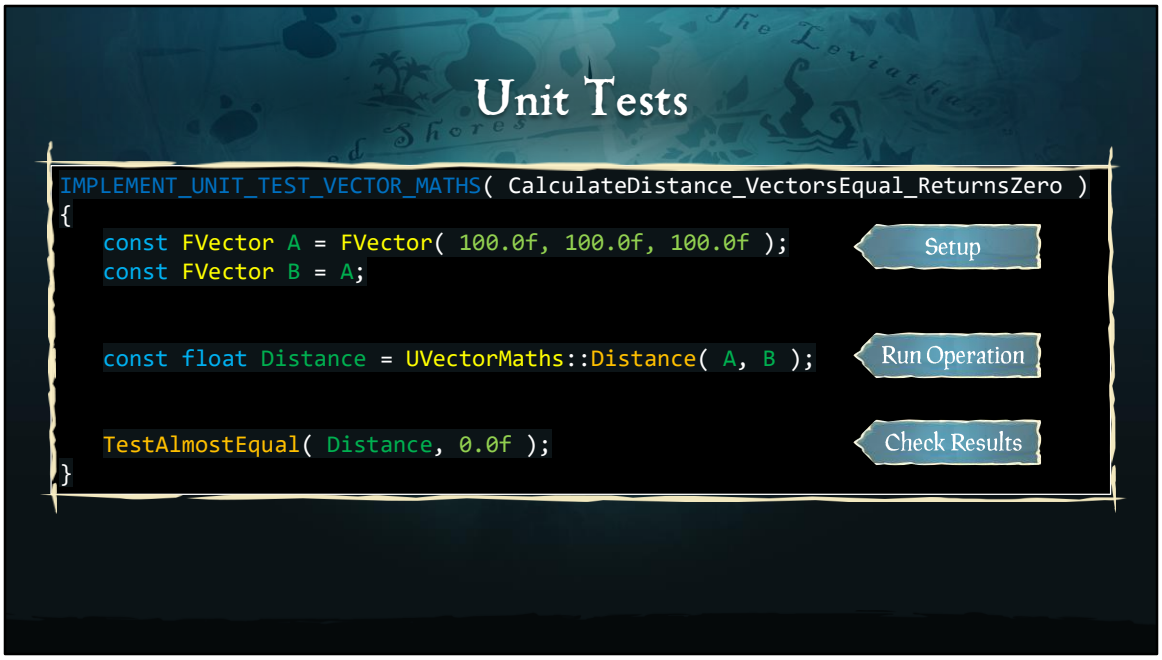
- If instead of wasting a human's time with these checks, we let the game do it itself via an automated test, we not only avoid wasting a tester's time but we get many other advantages.
- First the game can run the checks much faster than a human could, which means they can be run more often so the dev team will know even sooner that there's a problem. Next, the game can be more precise when running tests as its able to read and check its own state, so more accurate checks can be made than a human could detect. Finally, automated tests can also interact with the game at different levels, testing individual code functions if necessary, while a human can only play the game in its complete form.

- Automation is not appropriate for everything though. Humans will generally be much better at recognising defects associated with the visual and audio parts of the game. Humans are also much better at doing exploratory testing and finding new, unexpected problems. Finally, checking game experience and assessing how the game feels to play really can't be done by an AI.
In future, maybe we'll see advanced AIs that could test games like humans do, but for now, a combination of the two testing methods is the best approach. Manual testers can let automation do all the repetitive checks, leaving them free to do something more productive with their time.

-- Seeing the benefits, we made the decision to use automated testing in our testing process. Next I'll explain how we created tests within the testing framework that we created.

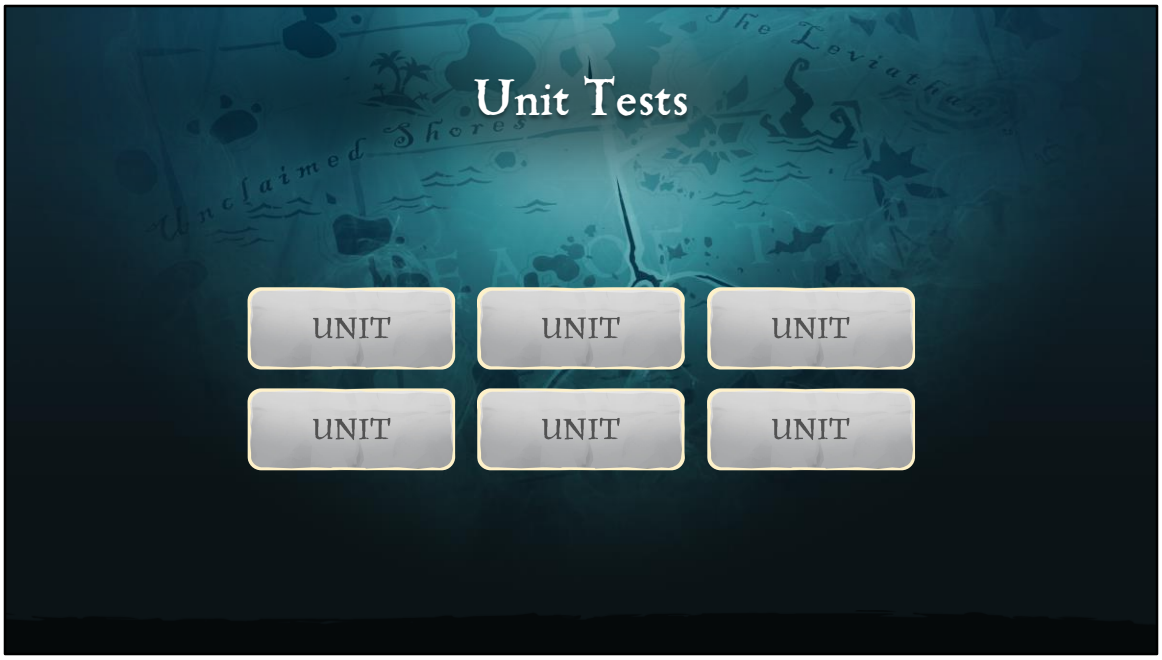
Unreal Engine Automation System

-- Sea of Thieves was built from the start as an Unreal engine game, although the version we built the game on top of is now a few years old so what I show you in this presentation may not exactly correlate with the latest version of Unreal.
We implemented our test framework by heavily modifying the automation system that already existed in the engine. [If you're more familiar with Unity instead, Unreal's automation system is the equivalent of the Unity Test Runner.]
- Running tests in the editor was as simple as going to the automation tab, selecting the tests and executing them like this. In addition, the tests could be run on built executables or remotely by our build system. And to help developer workflows, we also created a standalone tool that allowed running of tests from outside the editor. This meant that developers often didn't need to run the game at all to see if their latest code iteration had broken anything, as the tests gave them fast feedback on their changes.

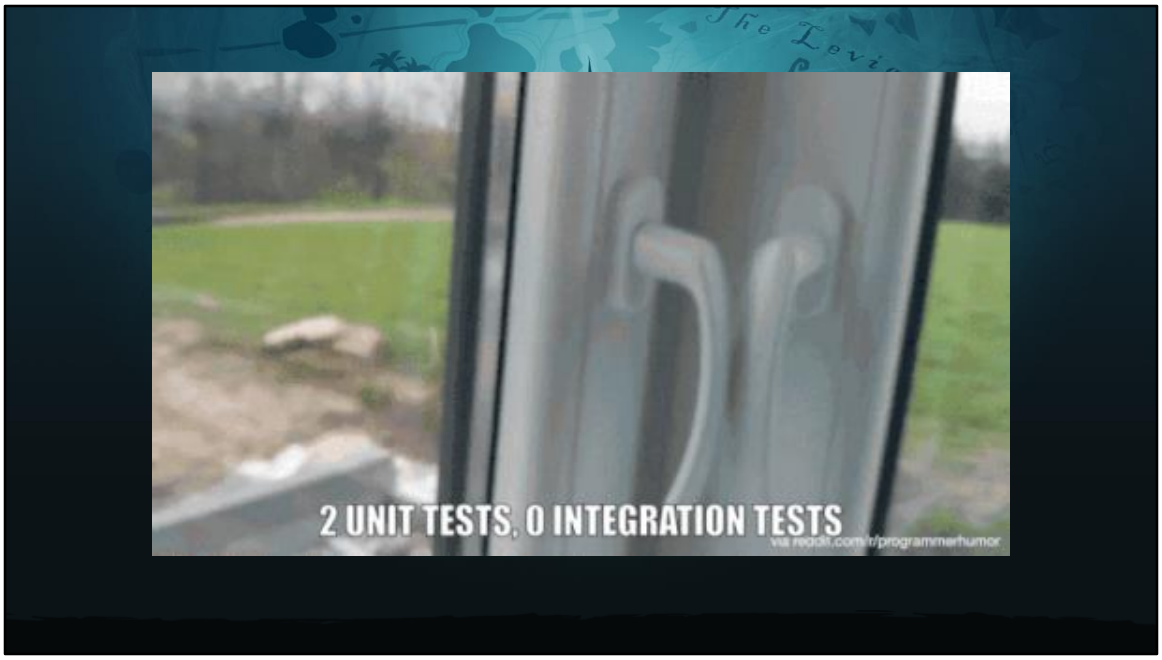


-- The simplest kind of test we could create with the test framework were unit tests. If you're familiar with well known test frameworks like NUnit, these will look very similar. It's essentially just a bit of code that registers with the automation framework so it can be run as a test. By definition, unit tests check a specific operation on the smallest testable part of the code, which generally means testing at the code function level. For example, here is a test that checks a calculate distance function gives the correct result of zero for identical vectors. In this example, you can see the three steps that make up each test we create:

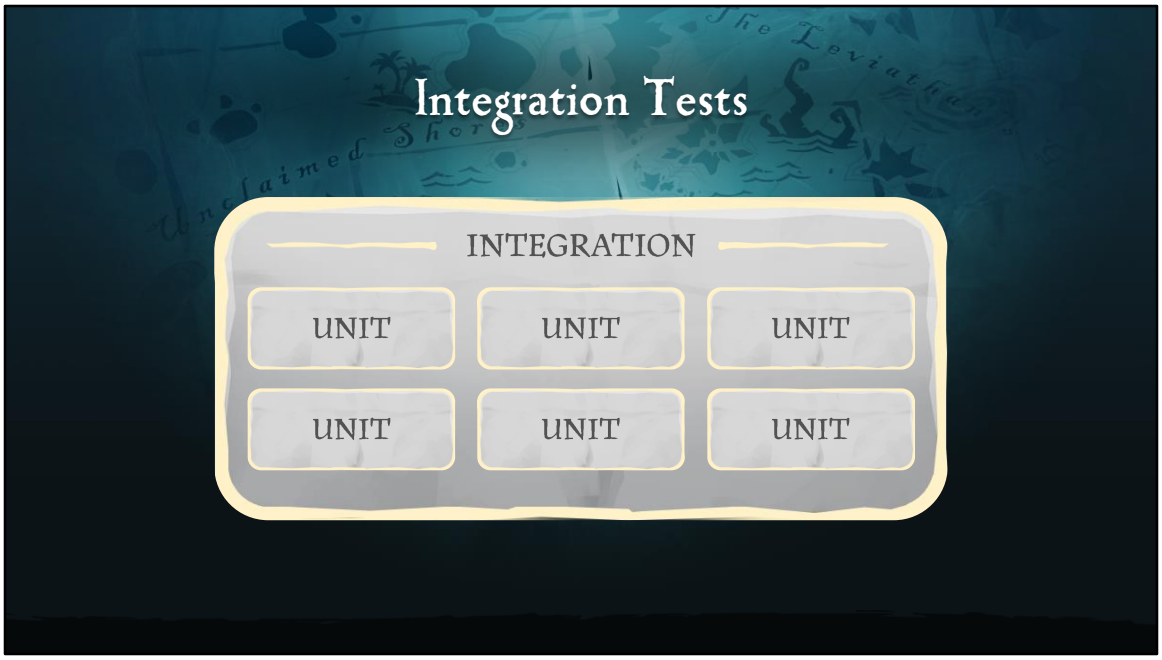
- First we set up the data or objects that we want to test,
- Next we run the operation that we're testing,
- then finally we assert that the results are what we expect. If the assertion fails, an error will be logged which will be picked up the automation system and mark the test as failed.



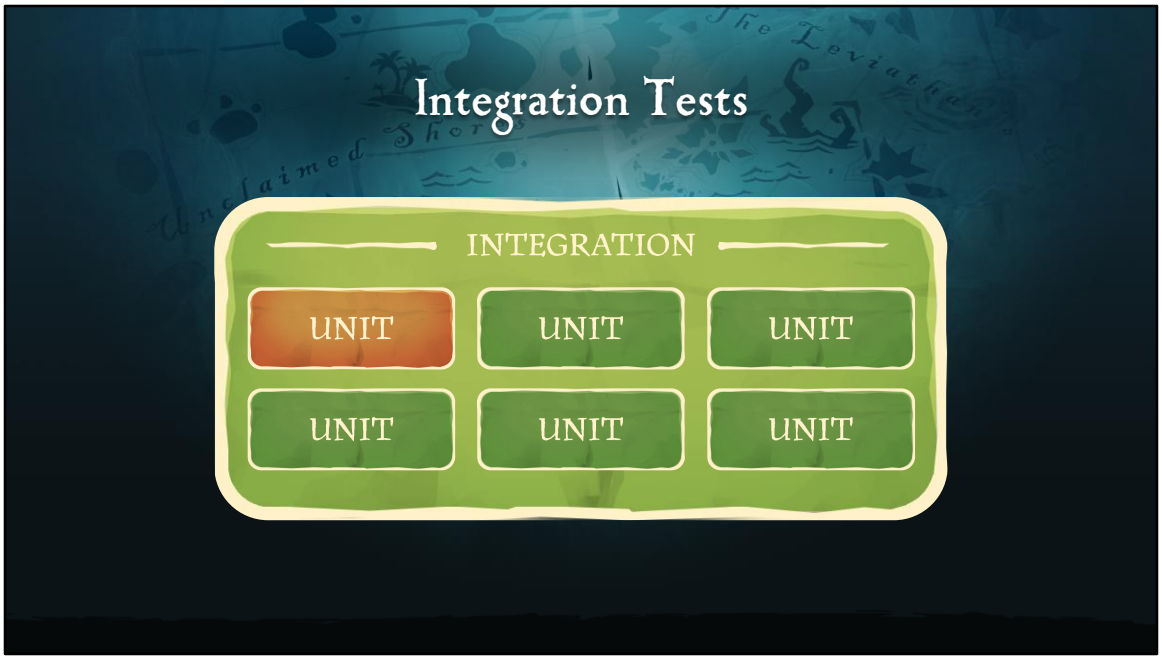
-- If we had unit tests that cover every individual function of the game, we could in theory build up testing that covers all of our game.



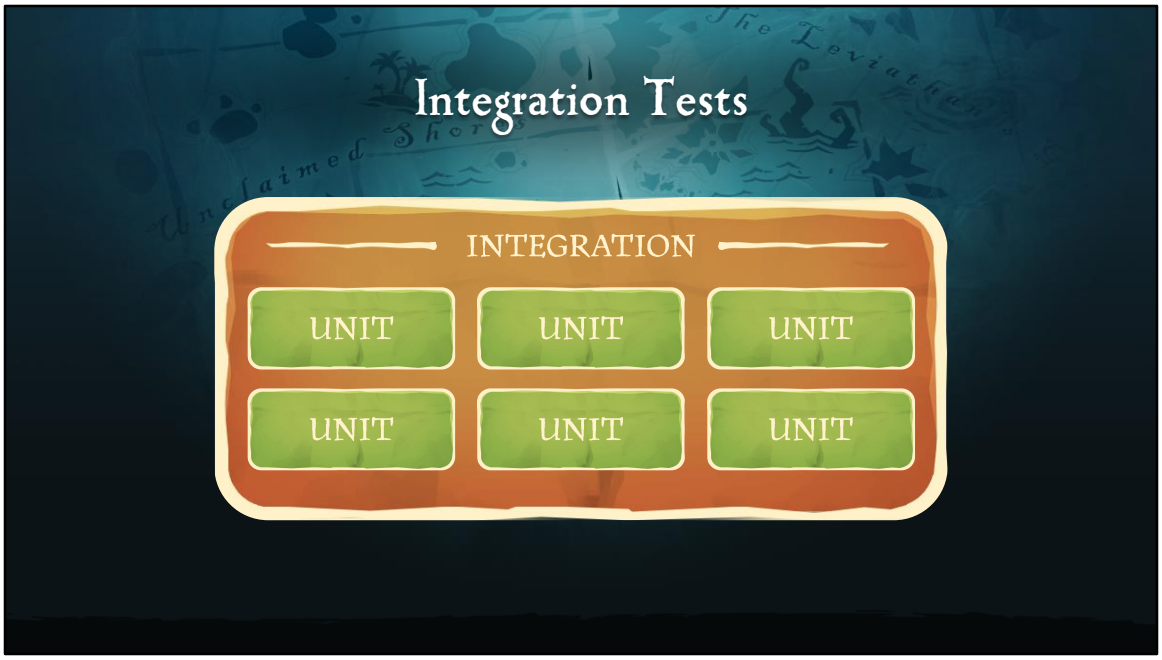
-- However sometimes the way units interact can itself contain bugs…



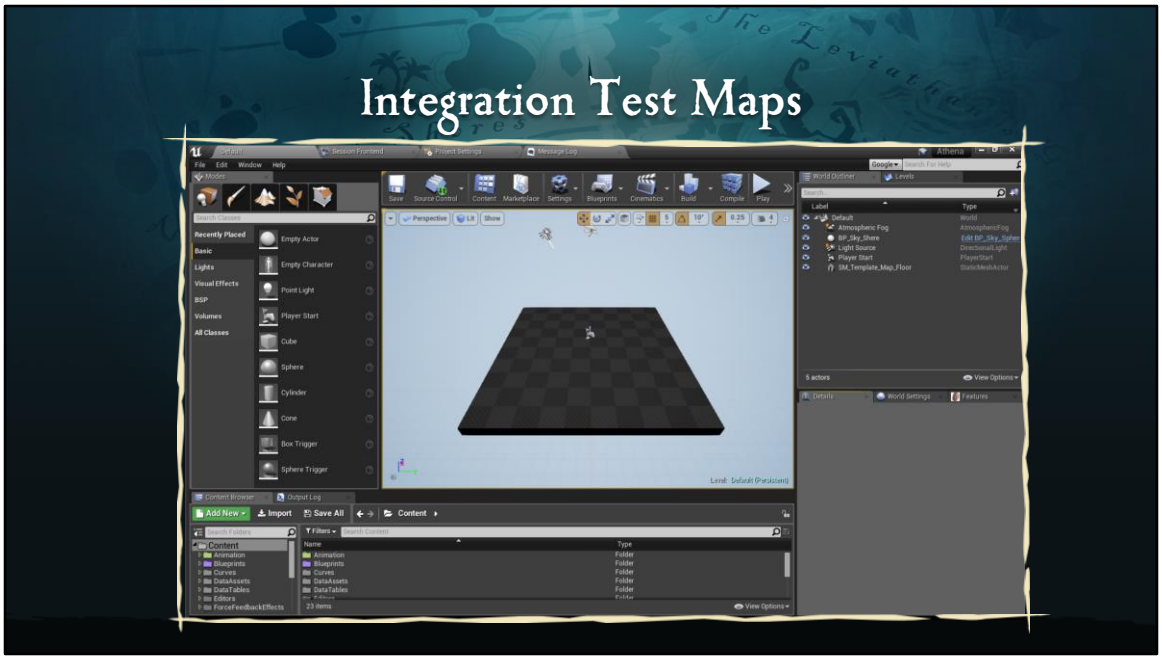
-- So alongside the unit tests we created, we also created integration tests that generally cover a whole feature or action in the game. This meant they provided coverage of multiple units and the communication between them.



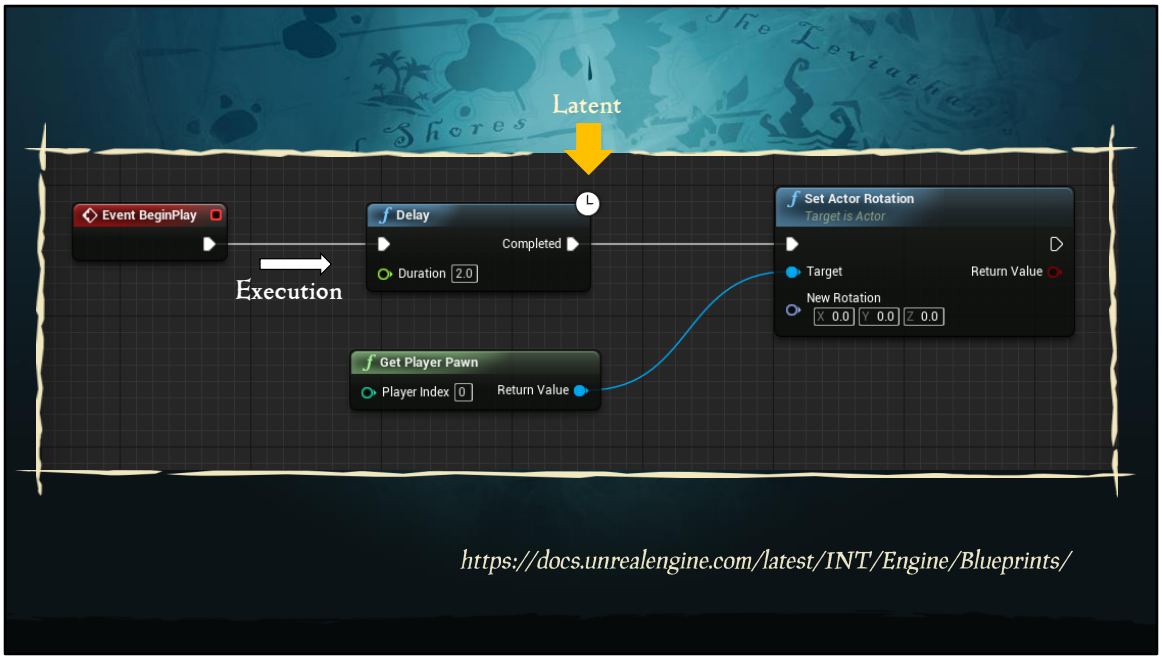
So now if a unit test fails for example, we know straight away that that specific unit of code is at fault.



However if all our unit tests pass and an integration test fails, we can generally discount the units themselves and look for other causes, perhaps to do with the communication between units.
An integration test failure, due to the larger scope it covers, will take longer to investigate than a unit test that tests something very specific, but they're still a useful high level signal that something about a feature has broken.



-- Integration tests in sea of thieves were created as maps in the Unreal editor.
Each map would test a specific game behaviour in a fixed scenario and report back its results as a pass or fail, based on whether the behaviour happened as expected or it didn't.



-- To create the logic of what happens in these integration tests, we made use of the Unreal Blueprint system. If you're not aware of blueprint, its is a node based-scripting system within the Unreal engine.
- To follow the flow of blueprint, look for the white line which represents the thread of execution. In this example, we're setting the actor rotation of the player to all zeroes two seconds after receiving the begin play event.
- Blueprint was very convenient for running integration tests as nodes can be latent and pause execution until a certain condition has occurred. For example, here the delay node will pause execution until two seconds have passed. We could have written integration test logic in code, but the latent node support and the speed of iteration of blueprint made it very convenient.

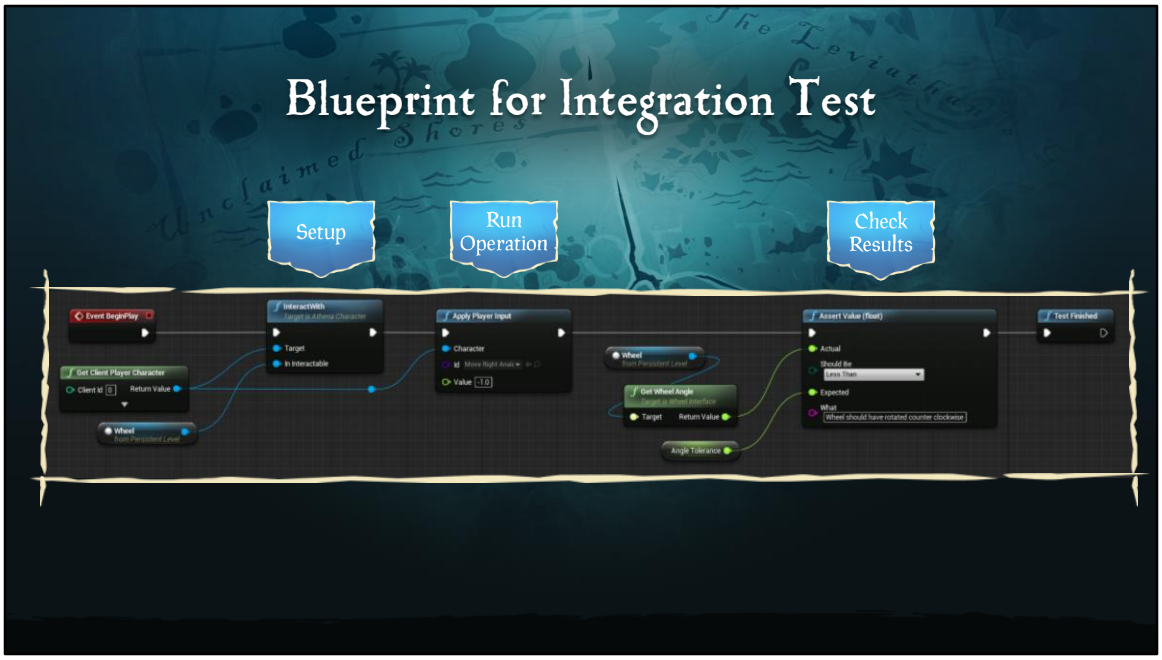
Testing Player Turning Ship's Wheel

-- As an example of an integration test for Sea of Thieves, let's say that we want to create a test for one of the most basic activities of the game, checking that when the player interacts with and turns the wheel, that the angle of the wheel changes.

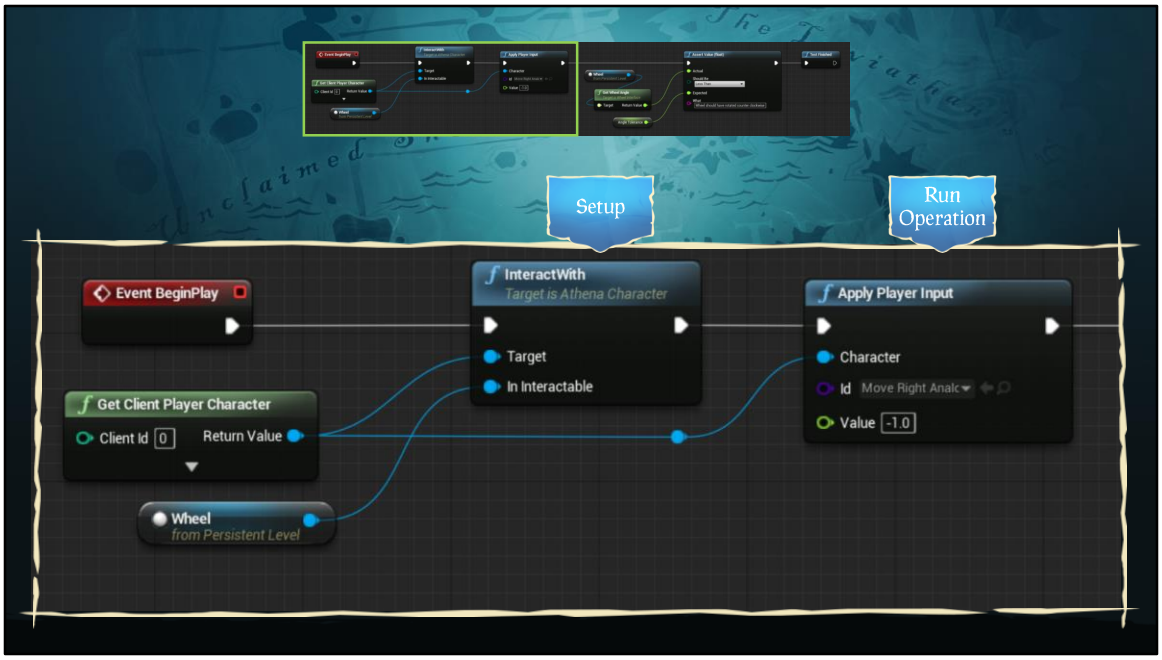
-- That's what the activity looks like in game, but now we want to create an integration test that will provide coverage on this feature.
To do this, we could automate loading a full client of Sea of Thieves, connect to a server, then put the player on a ship and let them turn a wheel. But that would be very slow to load and would mean the test is affected by a lot of other systems. Instead we want to narrow things down to test only the actual interaction between the player and the wheel.

- So this is the scene I created in the editor to test the wheel functionality. We have a player, a platform for the player to stand on, a ship's wheel and that's it.

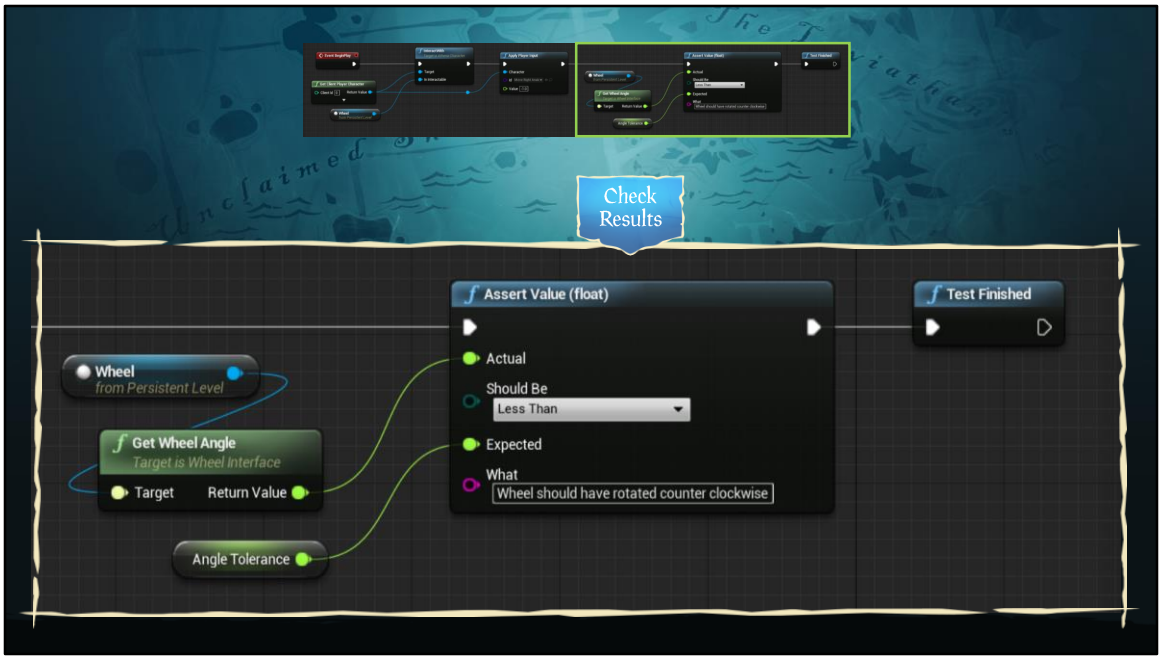
[An argument could be made that we should include a ship here too, as by far the most common use of the wheel is to control the rudder of the ship. But even though this is an integration test, we still want to test the wheel feature in isolation where possible. This will make it more reliable, as well as quicker to run thanks to not loading an expensive object like the ship. For these reasons, I decided to test just the wheel rotation in this test, and test the ship rudder separately.]



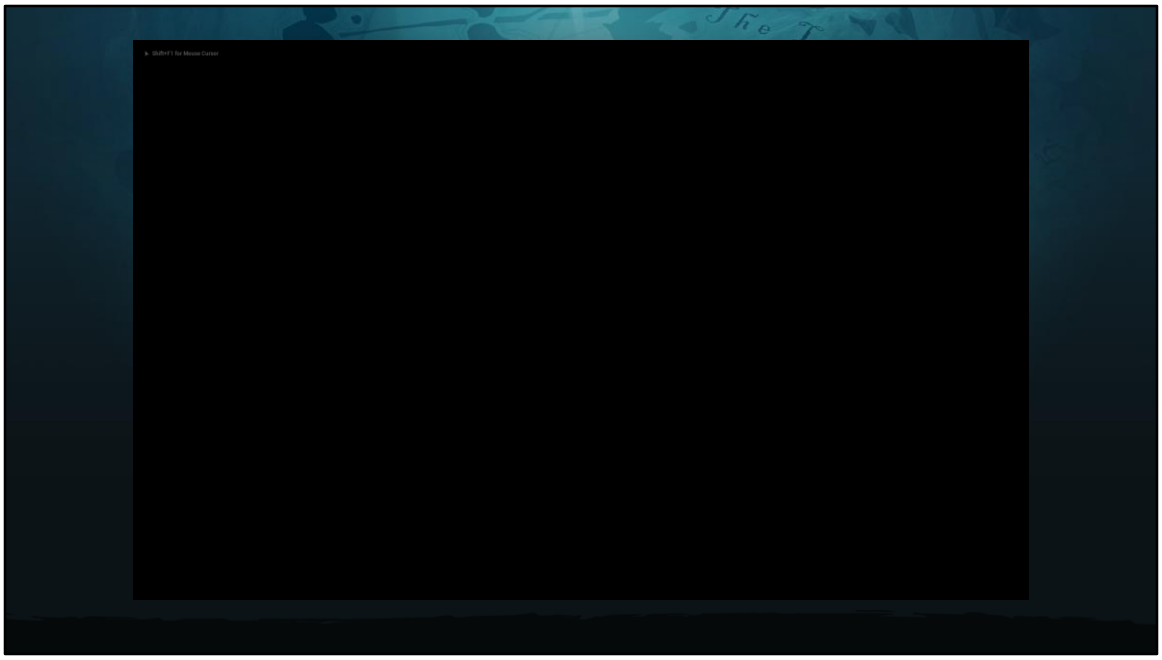
-- Here's the blueprint that does the logic for the test. As you can see it follows the three test stages in a similar way to the unit test: first do the setup, then run the operation we want to test and finally check the results are as expected.



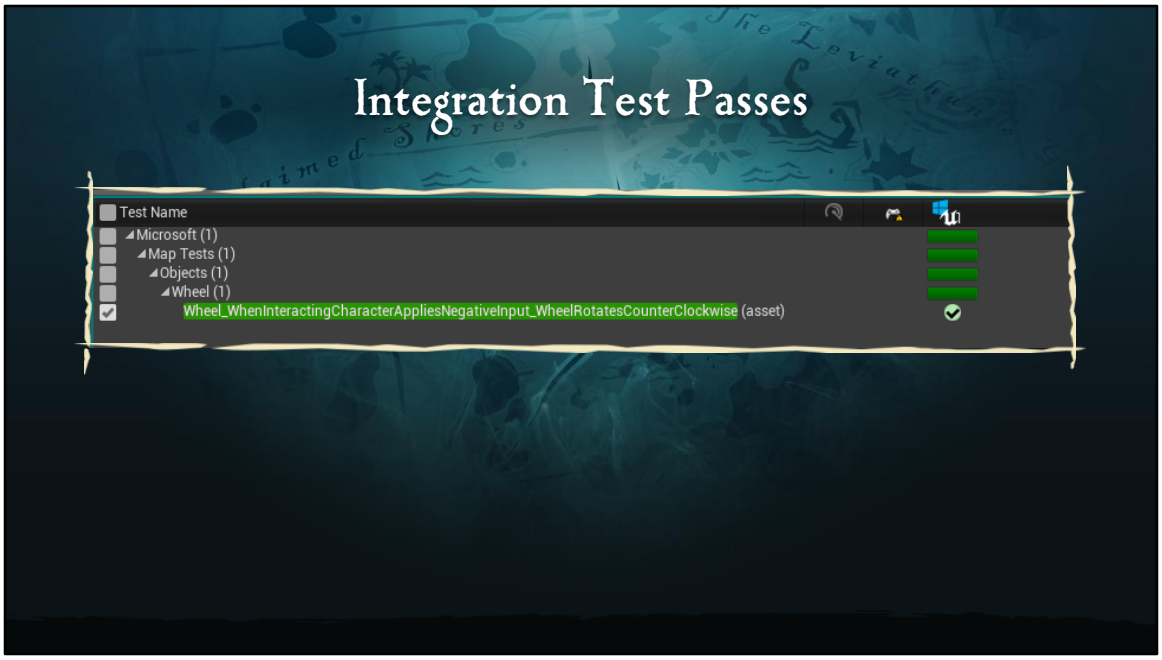
-- Now I'll zoom in and look at the blueprint in more detail. First the test does the set up phase. In this case, most of that is taken care of by adding the wheel and the player to the level. We do however need to have the player interact with the wheel before he can turn it.
Next we run the behaviour we want to test, which in this case is done by faking the player input we need to turn the wheel.



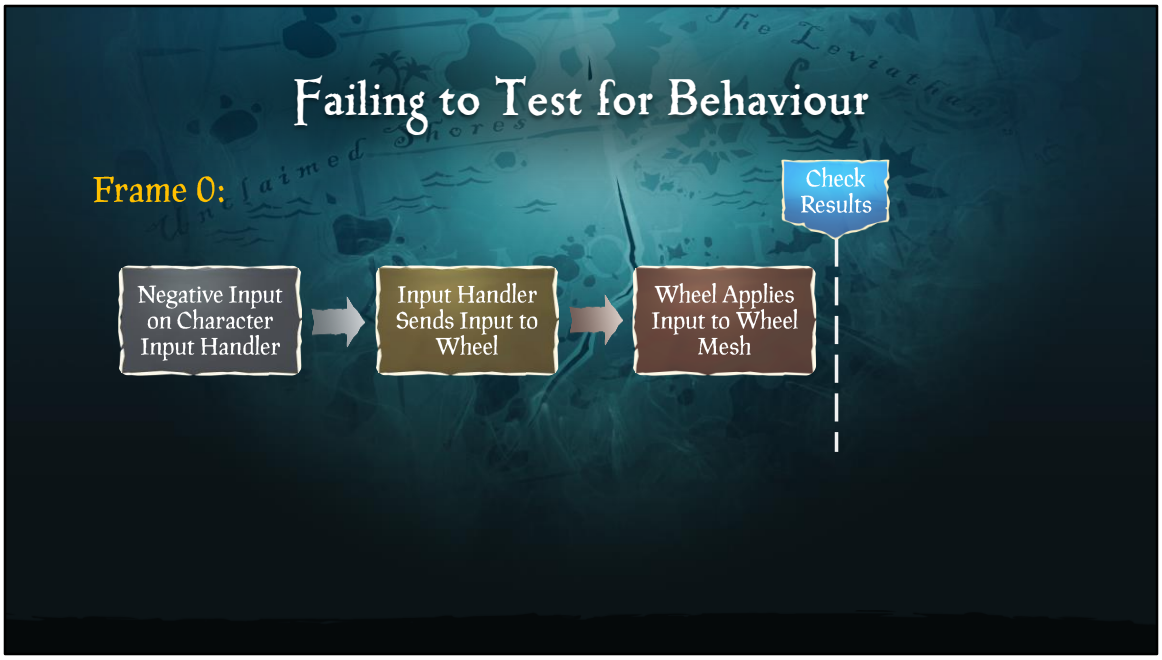
-- Then we check with a ‘Assert’ node that the wheel was successfully turned at a large enough angle. If no turn was detected this node will print an error to the log. The test runner will then pick up this error and mark the test as failed.



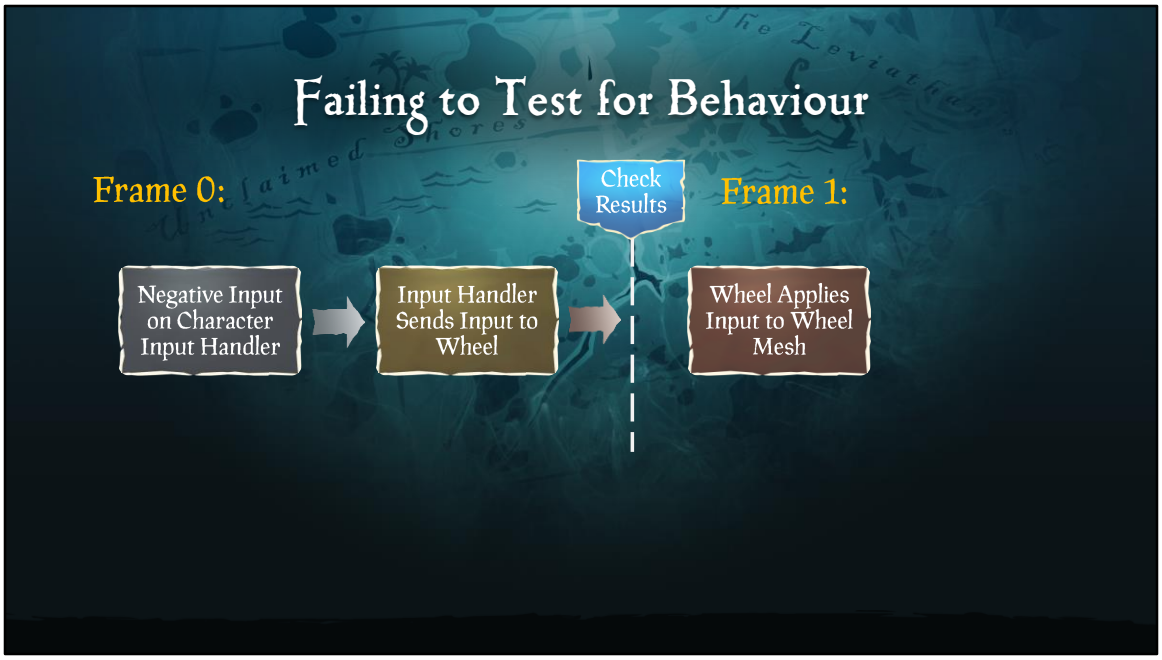
-- Here’s a quick video of the test running.
I’ve slowed it down a bit and made the angle larger, just to show visually what’s happening. Assuming everything is already loaded in the editor, the test runs in about a second.



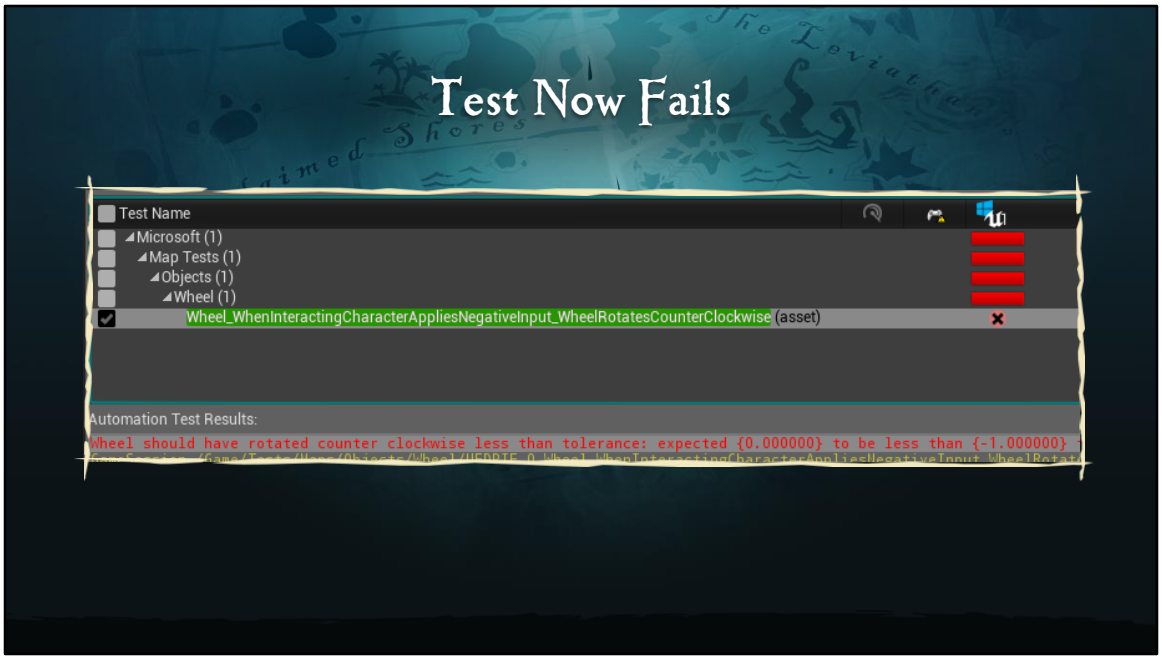
-- Going back to the Unreal automation window, we can see the test passes.
So now we have a test that is checking the whole operation of a player turning the wheel, not just at a code unit level, but also that it works with the in-game assets. If this test is run regularly we will very quickly see if some code or asset change breaks this feature.



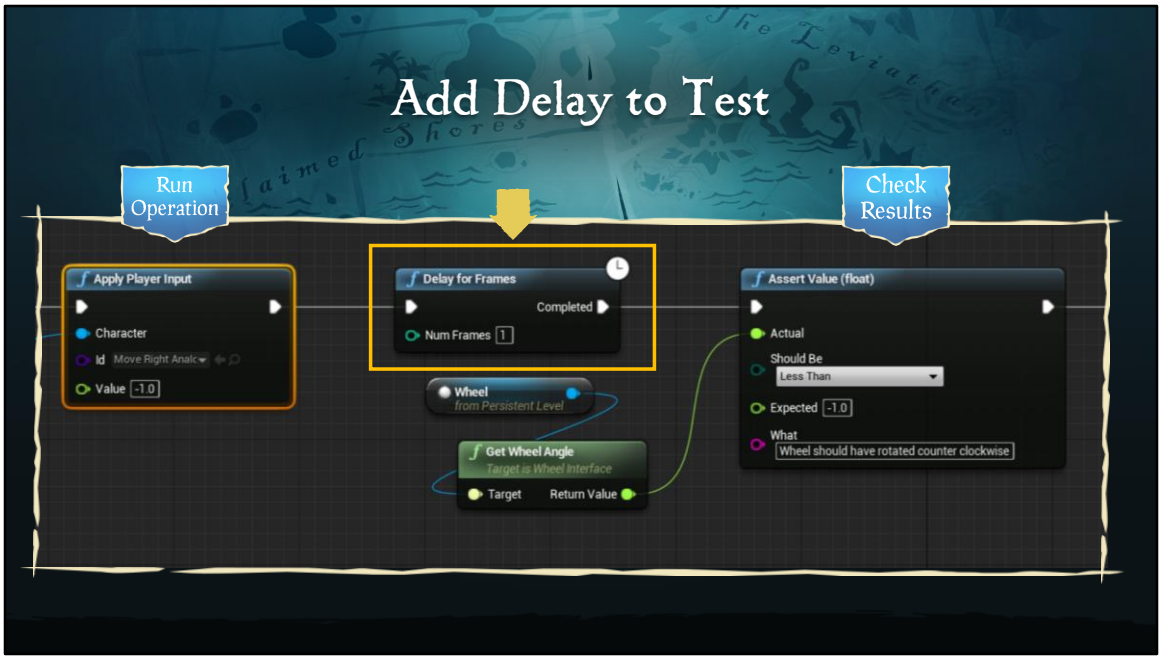
-- Its good practice that a test, particularly an integration test, is made to be robust to all changes that still confirm to the behaviour we want to test, which in this case is that a player can interact with and change the angle of the ship's wheel. If the test relies on more specific game details than that, its in danger of relying too much on the implementation of a feature instead, and you may end up having to continuously revise the test due to changes in the implementation.
- As an example of how a change in implementation could break our example test, here broadly are the three steps that the wheel turning code goes through that leads to a successful test result. First input is sent to the player character's input handler, then the input handler sends input to the wheel, then finally the wheel input is applied to the wheel mesh. At the end of the frame after all this is done, the test checks the wheel has turned. This is fine at the moment, but the test is relying on the wheel mesh angle change happening on the same frame as the input was initiated.



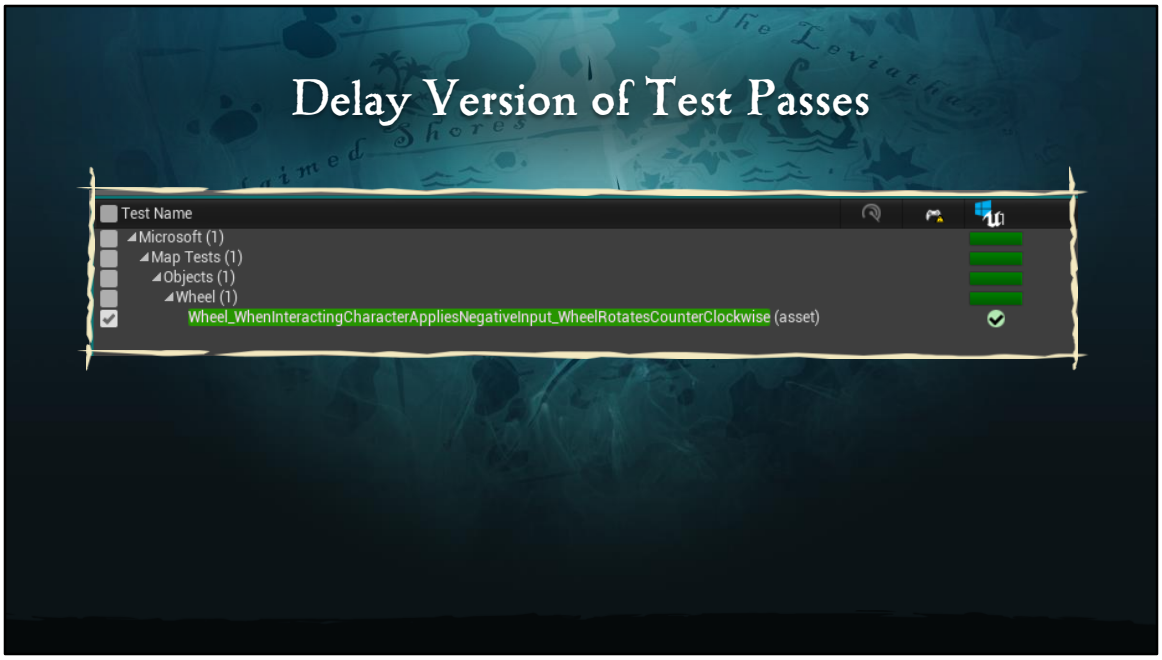
-- Lets imagine a code refactor takes place to defer the operation of applying the wheel's angle to the next frame. Now when the wheel is checked the wheel angle hasn't changed yet, so the test fails even though we know the wheel itself is still working. A case like this, when a test reports a failure when the behaviour is actually correct, is known as a false negative.



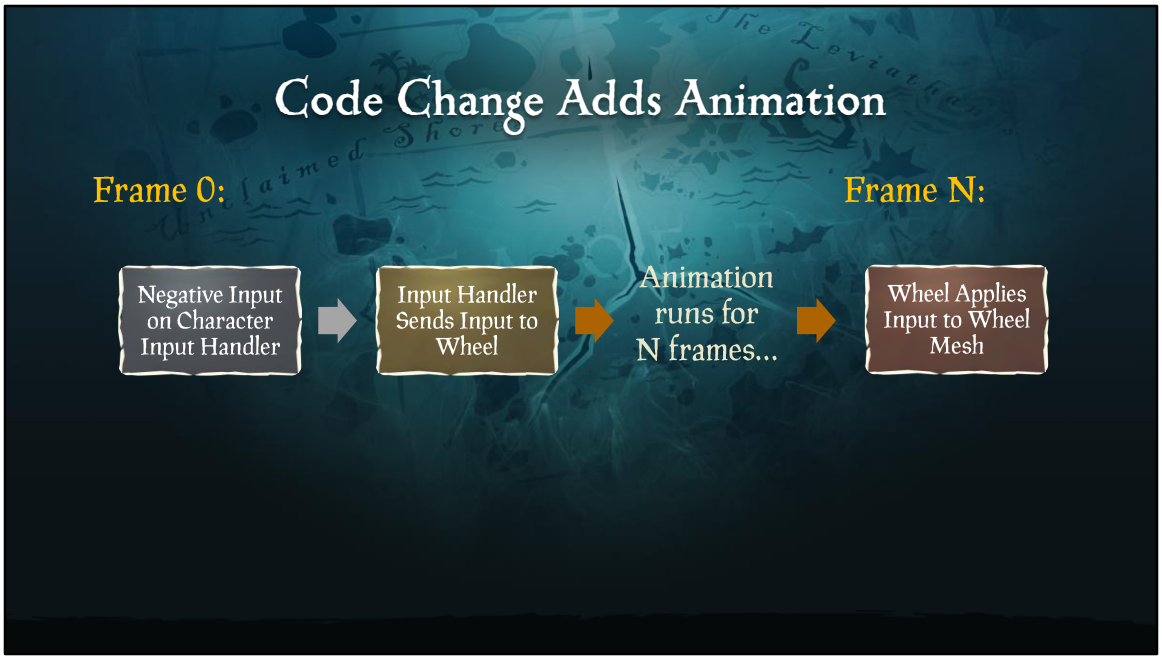
-- And if we go back to the automation window, we can confirm the test does now fail, with an error saying the wheel's angle is not correct.



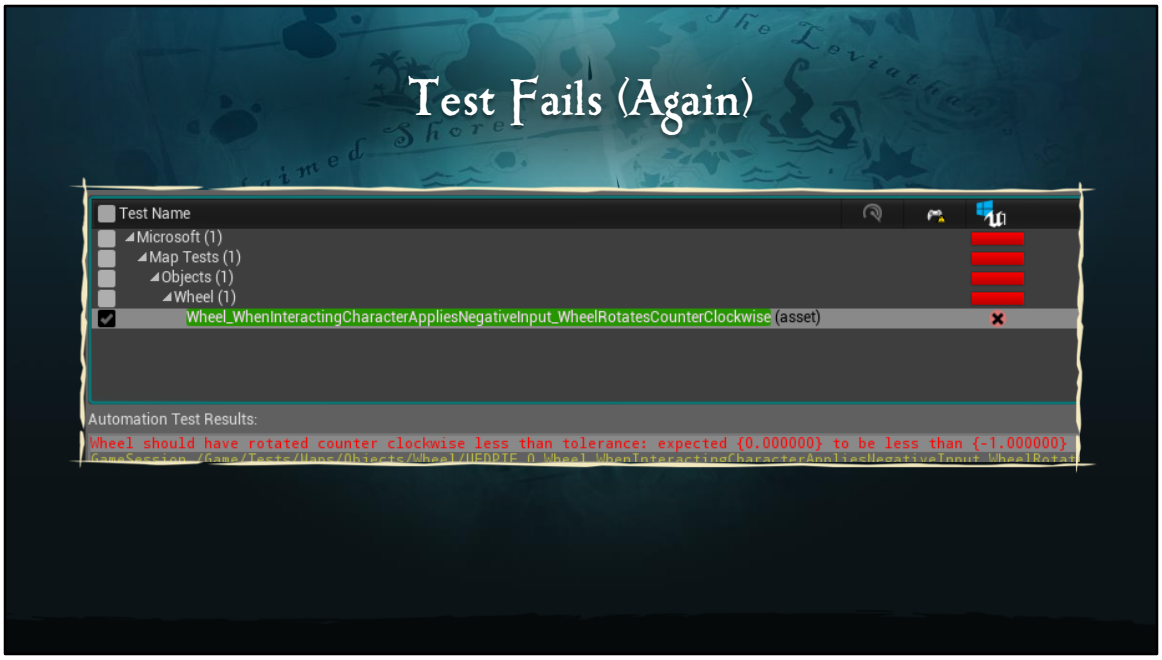
-- The most straightforward way to fix the test is to delay one frame before we check the state of the wheel. To do this, we add a ‘delay for frames’ node with a one frame delay, between the applying of the player input and the checking of the result.



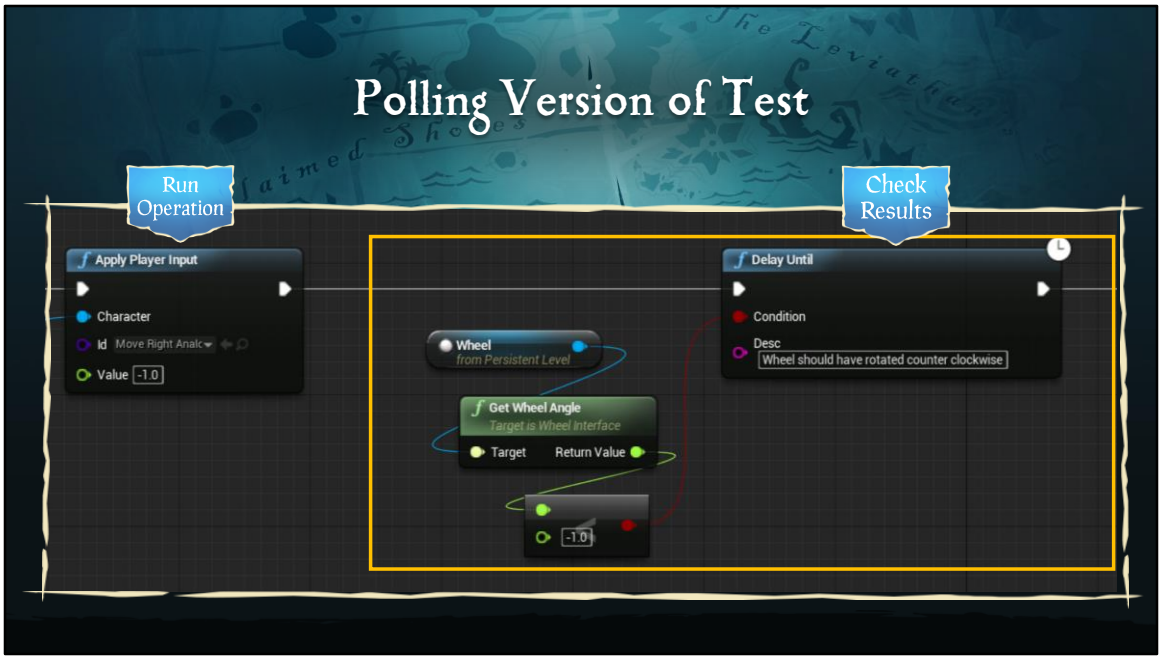
-- And now when we run the test, we can see it passes again.



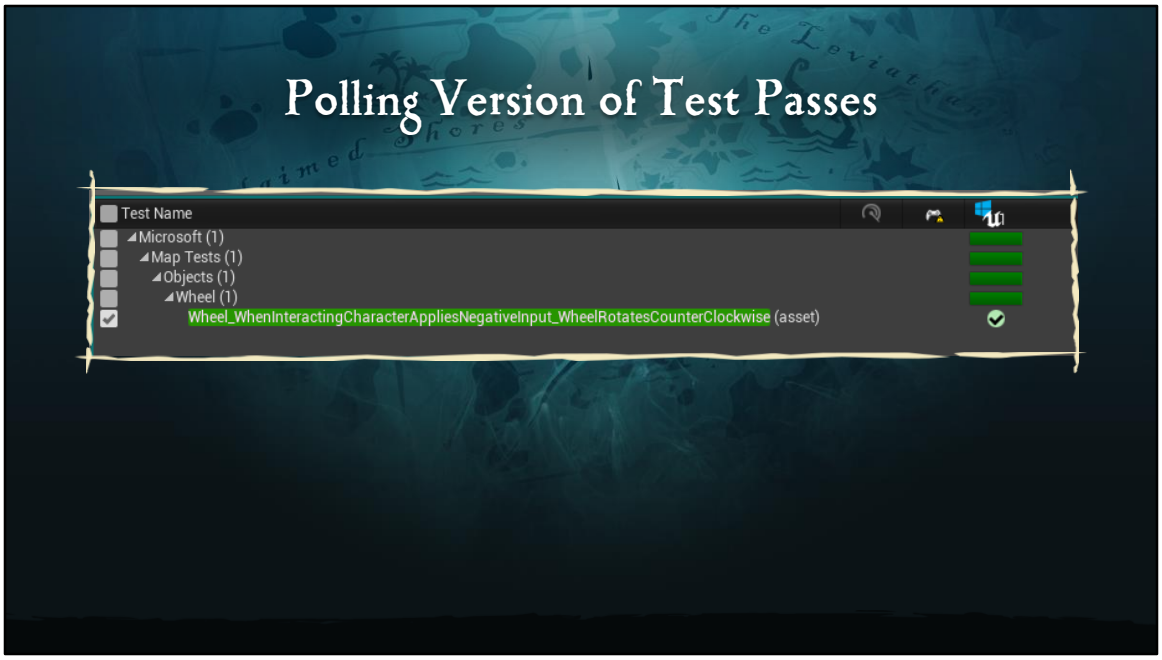
-- That's good but what if another code change is made. This time the wheel won't be turned until an animation of an unknown length has finished playing.



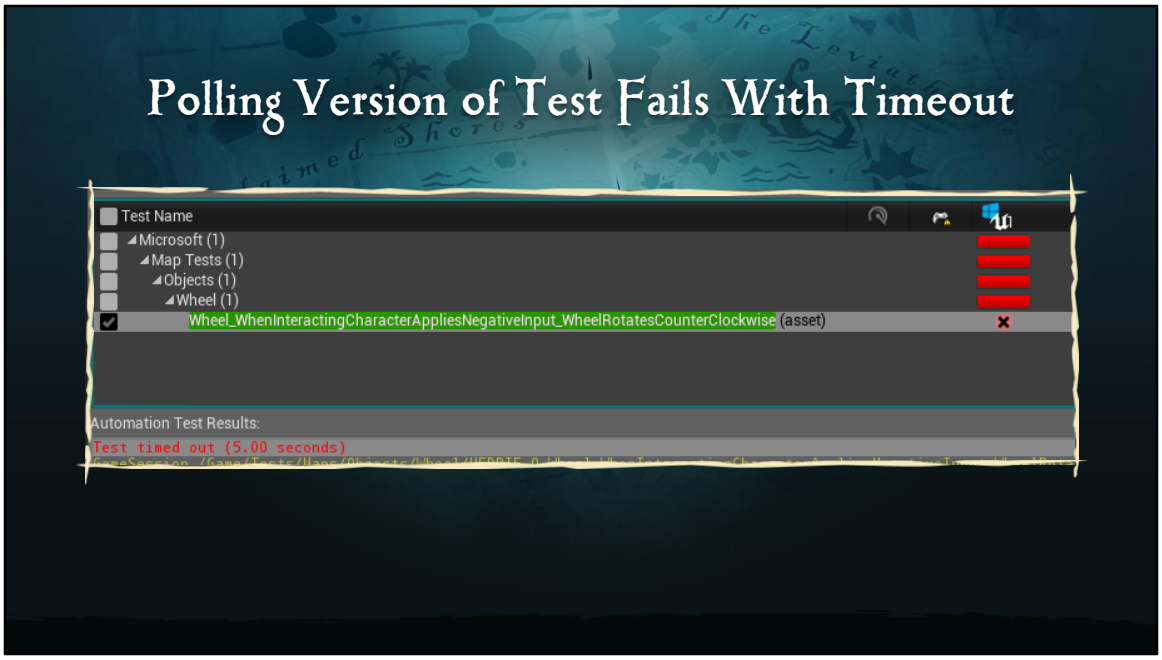
-- We can see the test now fails again. There are several ways we could fix this. We could have the test inspect the animation and wait till its over? That would work but that's making the test rely on an implementation detail again.
Do we continue to add more frames of delay? Perhaps we could just add a large delay, say 100 frames, that will always be enough to make sure the animation has finished playing. This would work, but we've now added 100 frames to the test time, when we expect the wheel to have been turned much earlier than that. 100 frames isn't a big problem when we run the test once, but when we do so many times a day on the build system, all this wasted time will add up.



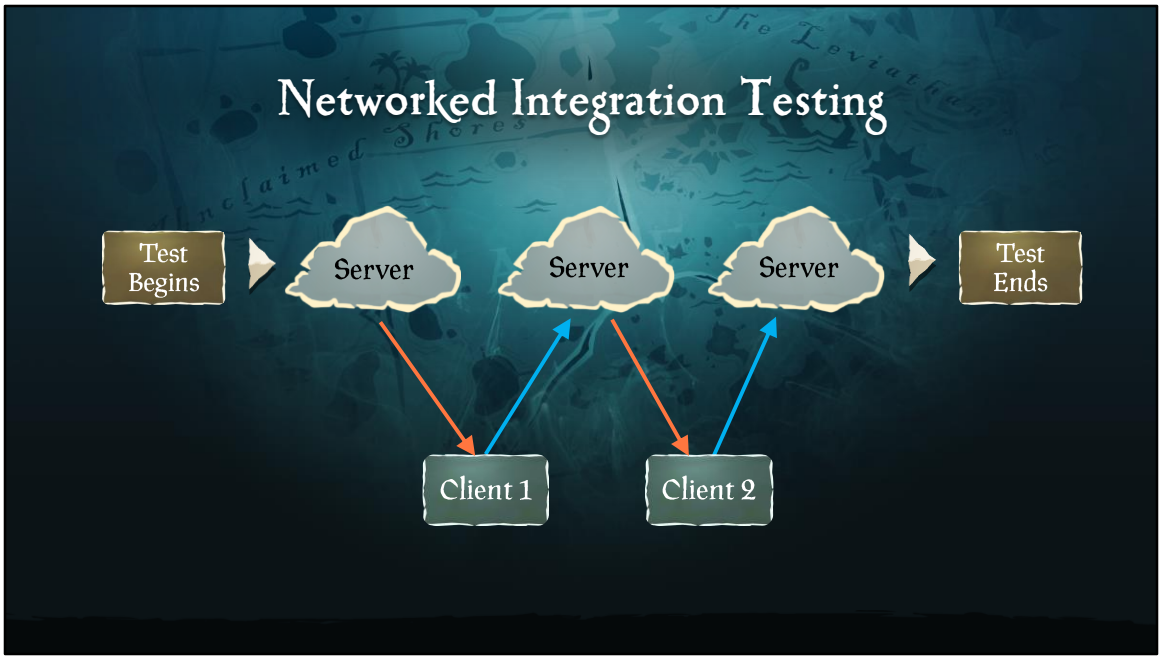
-- A better solution that is not implementation specific, is if we remove the fixed delay and use polling each frame instead to check for when the state of the wheel is what we expect.



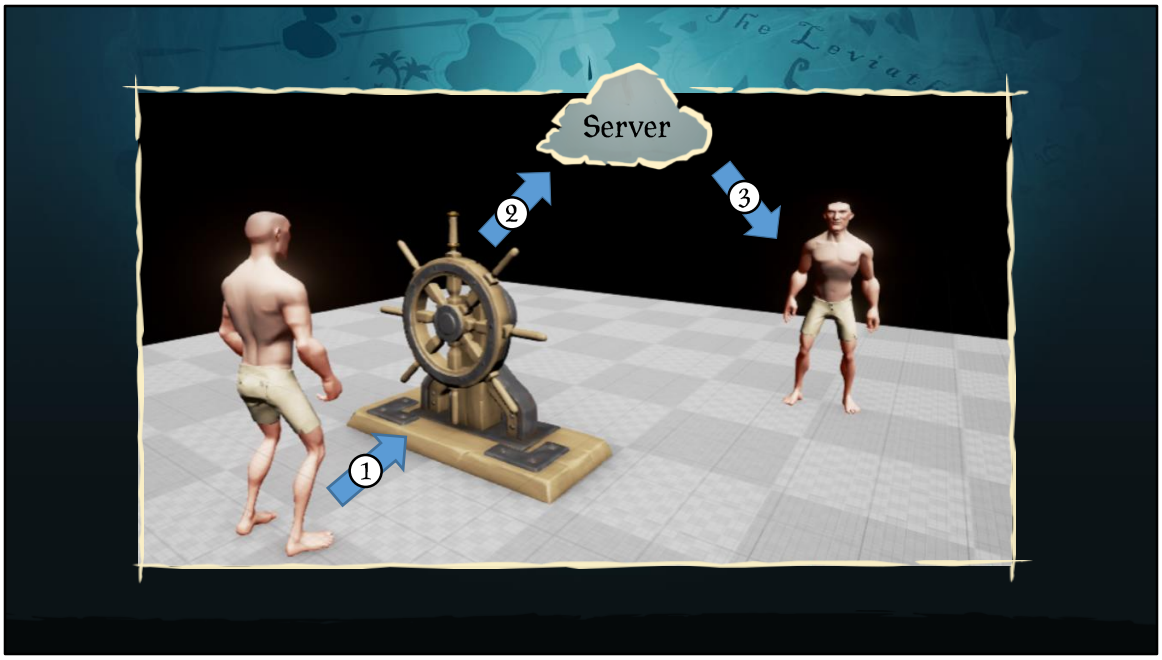
-- With this new polling version of the test, it now passes in the editor again, even with the added animation.



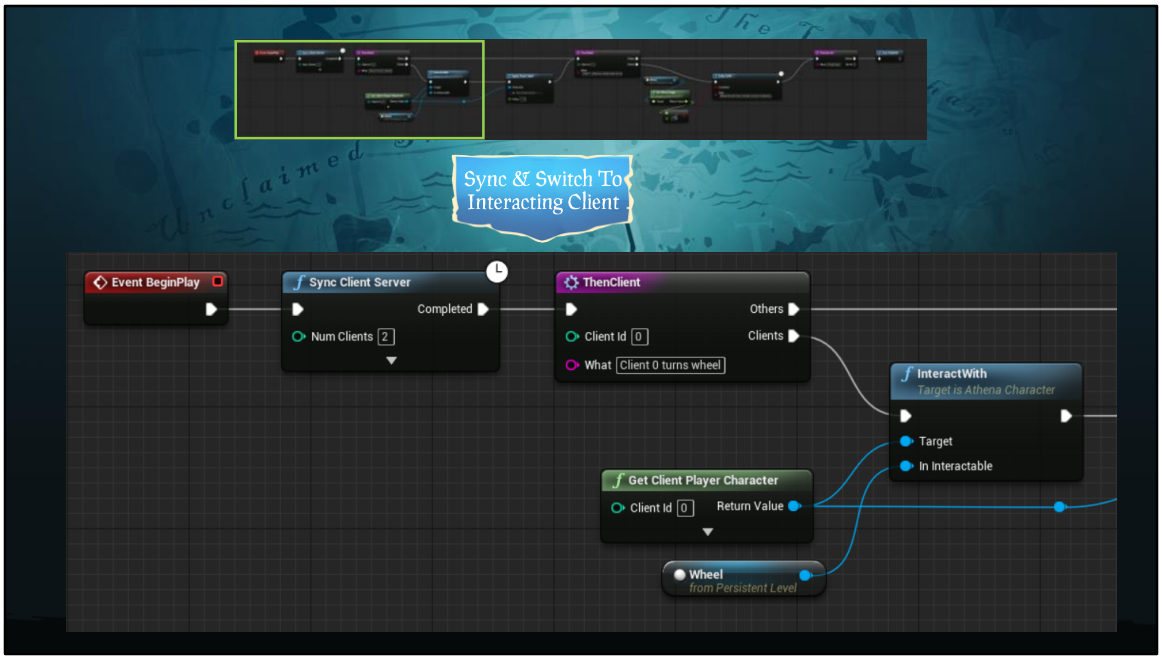
-- However if the wheel fails to turn, the test will eventually time out and fail as it never sees the correct state on the wheel. We do have to be careful about the timeout time is. It can't be too short as there is a chance it could give a false negative again, but too long and it would waste a lot of time waiting if the test has failed. We tended to make our timeouts quite long, as we assume that the failure case should be very rare, so the test would only need to be run to that time very infrequently.



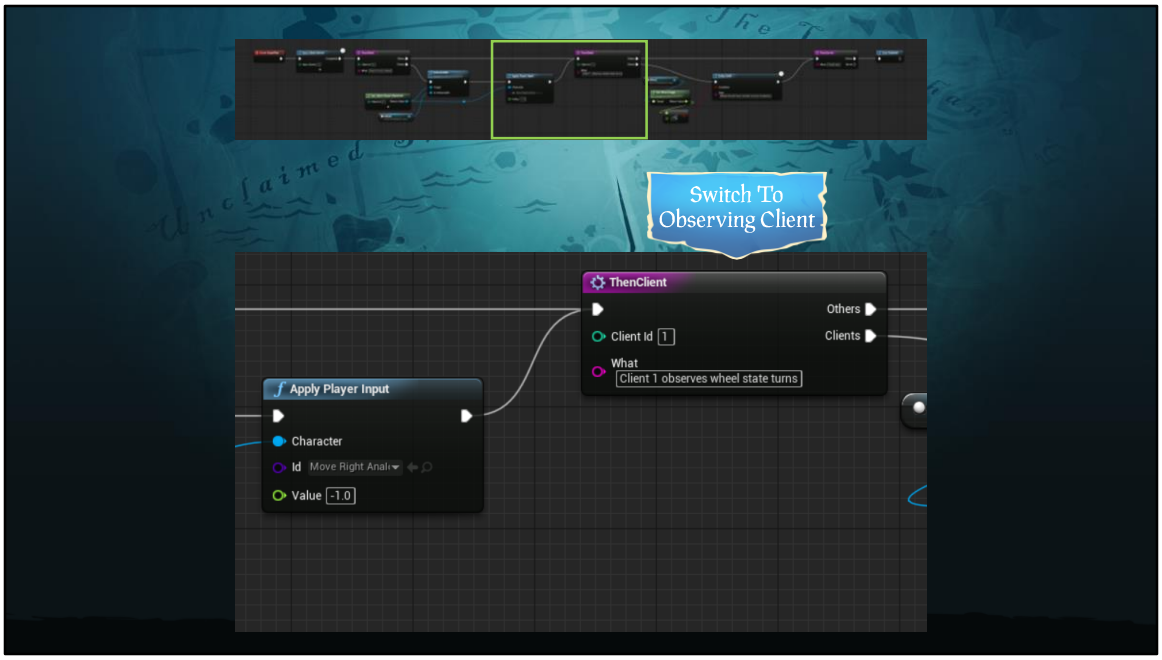
-- Sea of Thieves is a multiplayer game, it uses a networked server / client architecture. To provide coverage of that aspect of the game, our test framework supported networking too.
Integration tests could execute on both the server and one or more clients, from within the same test. For speed of iteration during development, a networked test could be also run in the editor with virtual server and client processes.
This is what would happen in a typical networked integration test:
- First the test starts on the server where typically we'll run the operation we want to test,
- After that we'll switch execution over to the client, where we'll check that the results of that operation has been communicated correctly over the network.
- Once that's done, we shift execution back to the server.
- At that point, we could switch from the server to a different client if we're running a test that requires checking communication between the server and multiple clients.
- We can then keep switching execution further, but to end the test we finish back at the server.



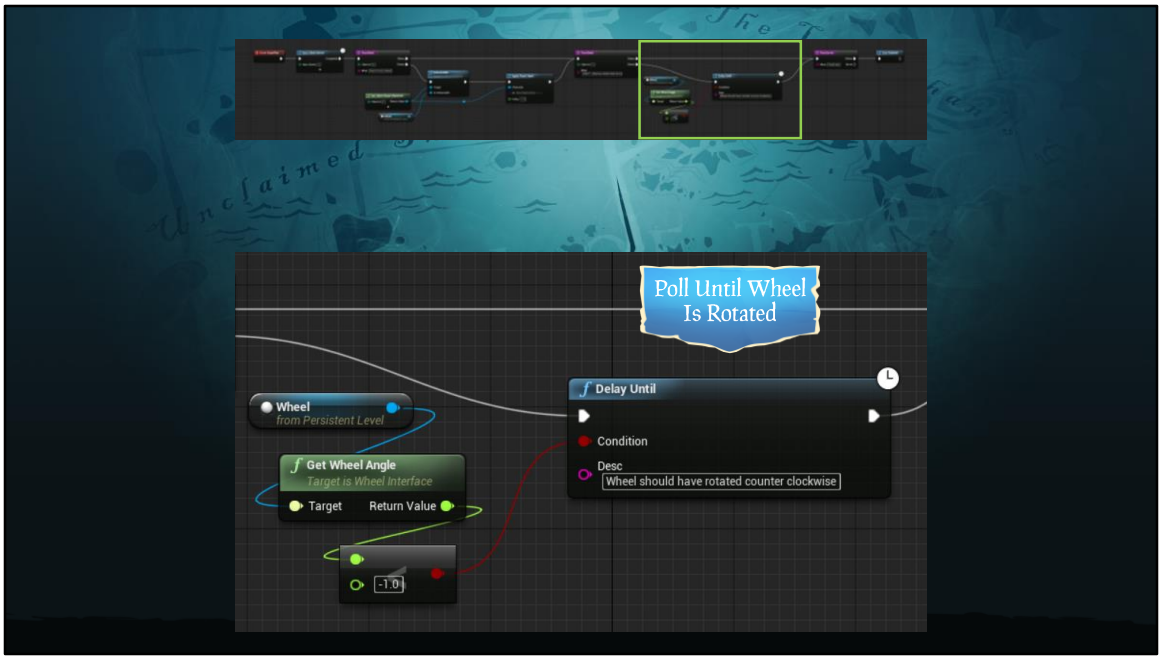
-- To see what a multiplayer integration test looks like in practice, we'll once again create a test that checks the player turns the wheel as before, but this time we'll check that the wheel state is also correctly communicated over the network. The test set up is the same as the previous test, except we'll also have another client present in the scene. We want to make sure that this second client sees the wheel state update correctly as the first client turns it.
So the sequence of network communication we expect to see is:
- first, a player interacts with the ship's wheel as before and turns it.
- Then the wheel state is updated and is sent to the server via a network message.
- Then finally, the server will communicate, or replicate, the new wheel state to the second client and the test will check its angle is correct on that client.



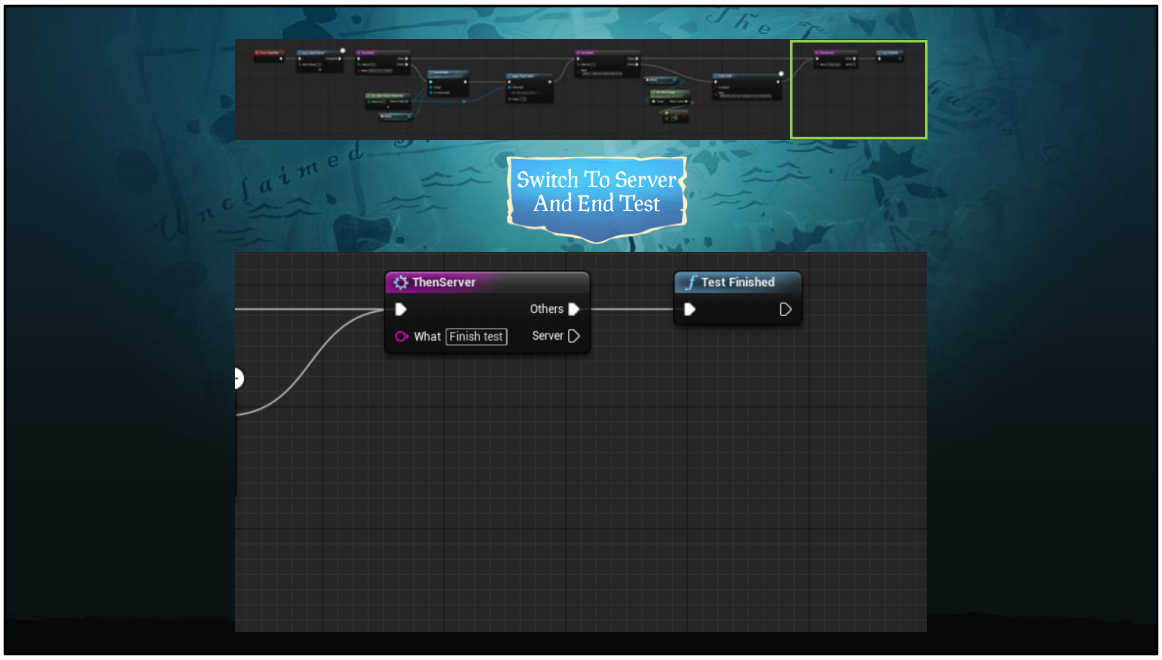
-- So lets step through the blueprint of a test that checks this.
- At the start of the test, we use a setup node to begin a networked test with two clients. This node does the initial handshaking required and stops the test from executing until server and client are fully loaded and in sync. We then shift execution of the test to the first client. That client's player is directed to interact with the wheel…



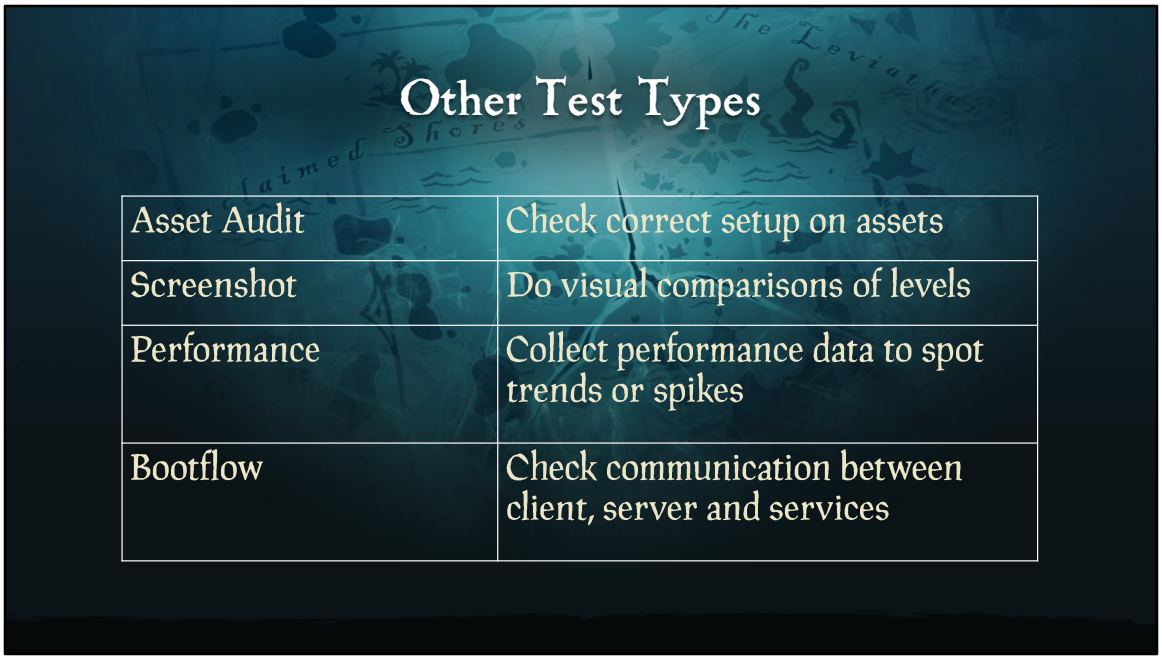
-- and turn it as before. After that, we shift execution to the second, observing client, going there via the server.



-- On this client, we do polling of the wheel angle to delay until its correct on this client, to confirm that it was communicated successfully over the network.



-- When the wheel angle is verified, we shift back to the server, so that the test can be marked as successful.



-- So far I've mentioned unit tests and integration tests, but there were a few other test types that were supported by our framework, which I'll briefly mention too.
First we have asset audit tests that run checks to warn about data setups unsupported by the game.
Next are screenshot tests, which are similar to integration tests, but output screenshots for use in a visual comparison against last good images, to check for unforeseen changes in the game visuals.
Then we have performance tests, which collect data from the game to spot trends in load times, memory usage or framerate.
Finally there are bootflow tests, which are the closest tests we have to actually running the real game. They're there to check the communications between client, server and game services are working for common scenarios, such as a client requesting to join a server.

[Screenshot tests - if you're asked about it, we do %age of pixels different and a manual test against last pinned for the non-deterministic visual tests. Often only every 1-2 weeks between manual checks.]

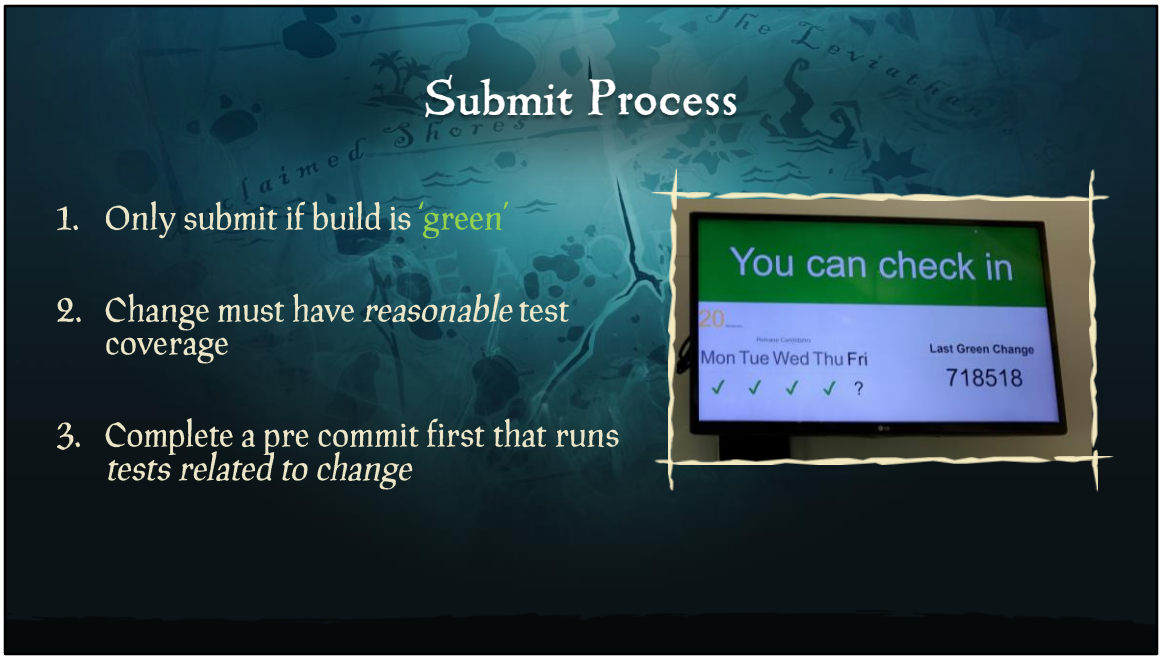
Testing Infrastructure

- Tests run as part of build system
- Each test run once every 20 minutes
- Test failure causes 'red' build

Jafar Soltani – 'Adopting Continuous Delivery', GDC 2018

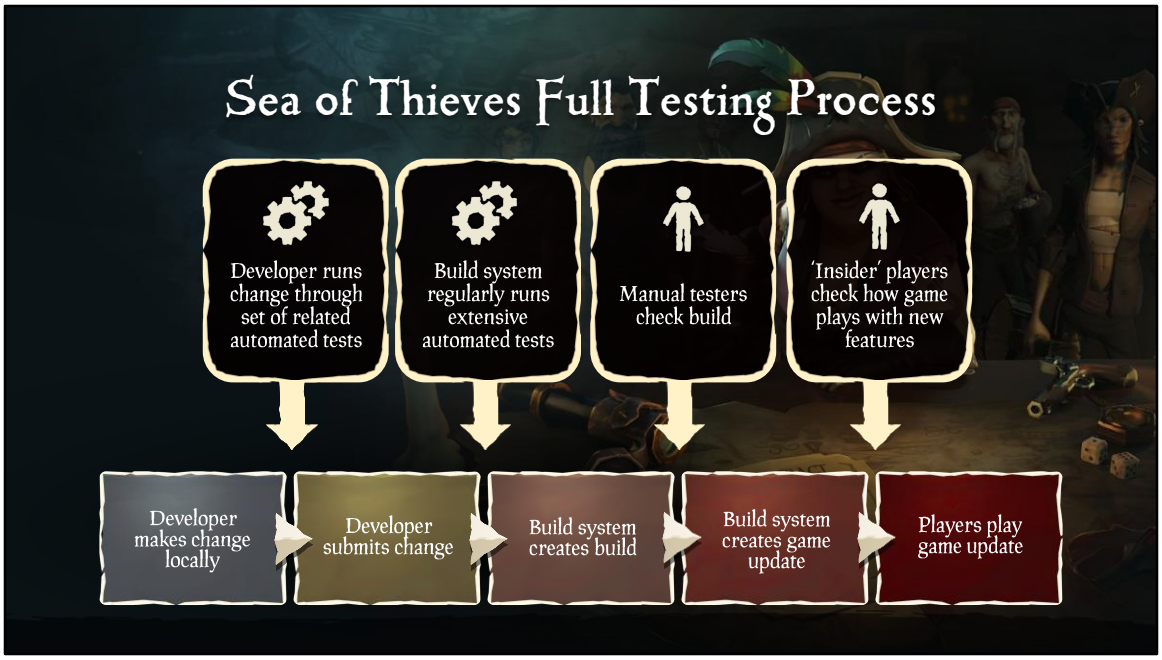
-- I'd like to talk briefly about the infrastructure we use to run our automated tests. I don't have time to go into too much detail, but if you do want to hear more, I would recommend watching my colleague Jafar's talk '*Adopting Continuous Delivery'* from last year's GDC.

- Our build system uses the Teamcity continuous integration software, which allocates PCs from a build farm to do various jobs, such as build the game and run automated tests.
- How often we run each individual test varies based on how long the test takes, but in general all our tests are run at least once every 20 minutes.
- If the build system runs a test and gets a failure, the build status will go 'red' to indicate there is a currently a problem with the game that needs to be fixed. To help with visibility of the build status, we have screens all around the studio that let the team know the status of the build, and if its broken which specific jobs have failed. We also put the name on the screen who last made a change that affected that test job, though in this case I've hidden the name to avoid embarrassing anyone back at Rare in front of everyone here at GDC ☺.



-- To make sure we avoid a broken build and the team disruption that comes along with it, all team members need to follow the same process before submitting:
- First of all, no submits can happen unless the build is 'green' and all tests are currently passing. We do this because we don't want more changes to be submitted if things are broken that could make the problem worse. That's why if the build is broken, we require that the change that broke the build is removed as soon as possible so the rest of the dev team is able to submit changes again.
- Next, every change must be covered by some kind of automated testing if it makes sense to do so. Most code changes from engineers will usually require some new test coverage, but an asset change from a designer or artist probably wouldn't need a new test as it would get picked up by our asset audit testing. If a new test is required, it's the team member who made the change who is required to create the test, and the test should be submitted with the change itself. At Rare, we don't have test engineers who are responsible for creating all the of test coverage for the rest of the development team. Instead we felt that the creator of a change is the person with the best understanding of it, and so should take responsibility for deciding what test coverage is required.
- Developers have some responsibility to check their local changes haven't broken the game, but inevitably they will miss things or forget to test certain situations.

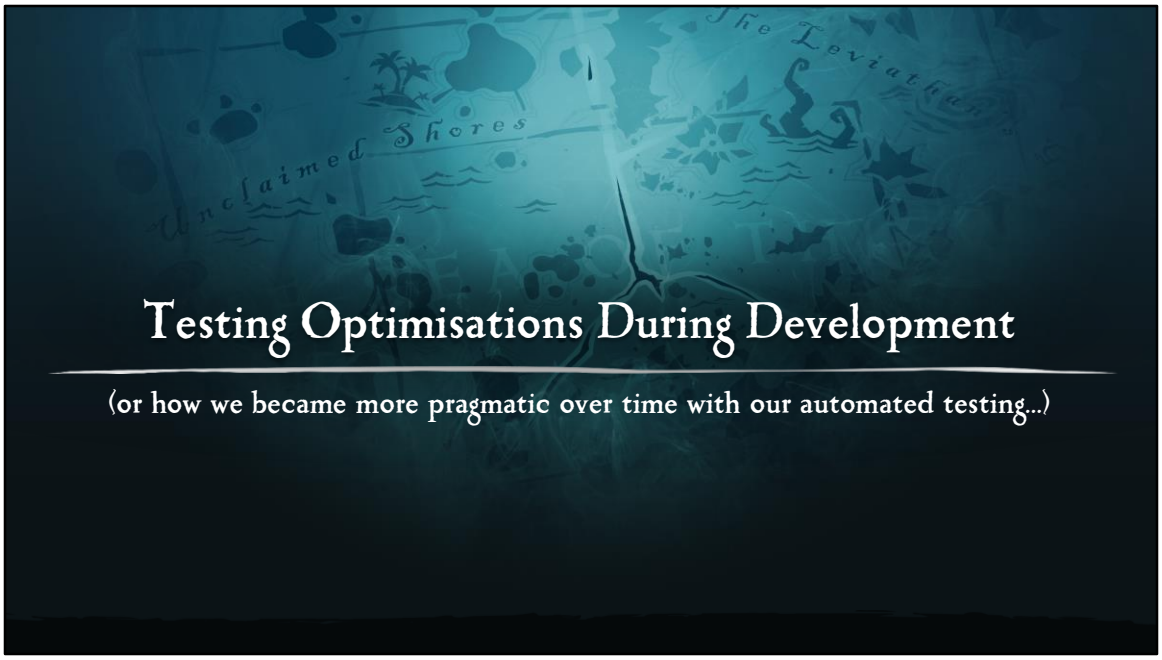
They could run automated tests locally, but that takes time to do and if a developer only runs the tests they *think* should be affected by the change, they could easily get caught out by a test failure that the developer did not realise was connected to their change. So instead, every submit also has to have completed a successful ‘pre commit’ process on our build system, to show that it is unlikely to turn the build ‘red’ and impact the rest of the team. The ‘pre commit’ builds the game from that local change and runs a subset of tests that are related to it. Running all the tests for every ‘pre commit’ was not possible, as that would take too long to verify and slow down the team too much.



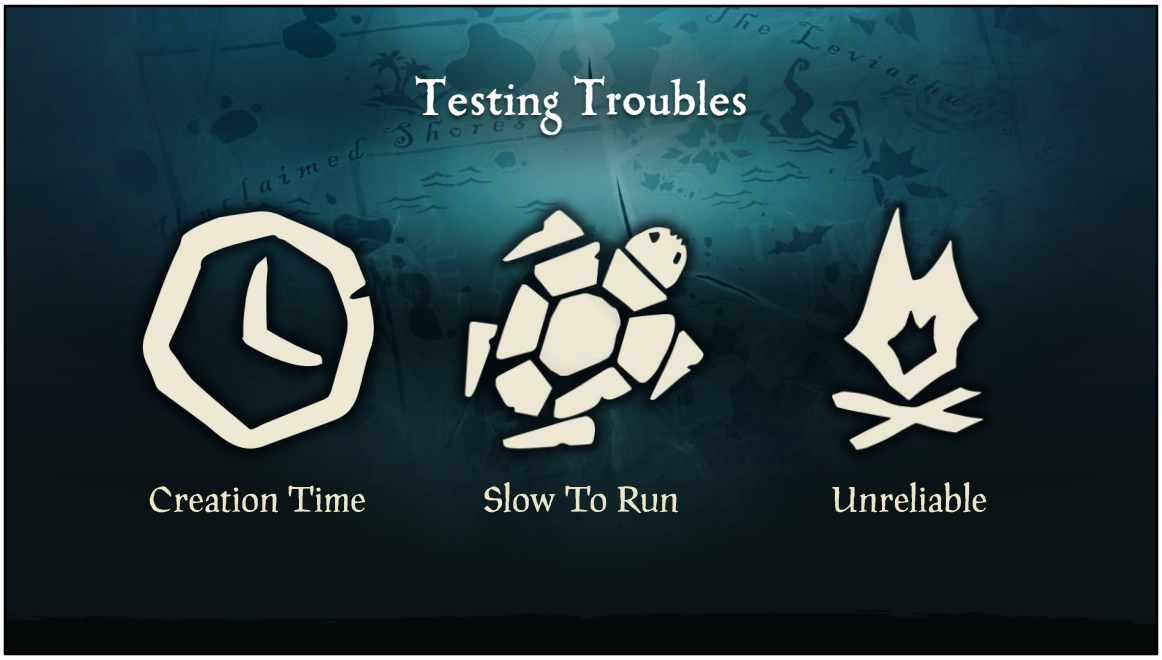
-- So in summary, here is the whole sea of thieves testing process that replaced the older, manual only process of previous games.
- First we have the previously mentioned 'pre commit' process that runs automated tests which are related to each change. When a pre commit comes back as successful, the developer is able to submit with reasonable confidence that their change won't impact the rest of the team.
- To be warned of problems as early as possible, we run a more extensive set of tests regularly to let the team know about intermittent issues or be aware of undesirable trends like worse loading times or performance. If one of these tests fail we can be alerted that the 'build is not usable' and the breaking change can be removed. This means the team can be confident that when it comes time for a public facing build to be created, all features will be valid.
- We are now reasonably confident that the game is functional and has no known problems in it. Of course, that doesn't mean that no problems exist, so just like the on old testing process, we have a team of manual testers who will check the build for problems. The difference with the old process though, is that when manual testers play the build, we are almost certain that the build won't have show stopping bugs that mean no productive testing can be done.
- Often when we're working on new features, we want some feedback from real

players before we release an update to everyone. We like many other game developers do this via beta testing. However, we don't however rely on these 'insider' testers to find bugs. Our previous stages of the process should have hopefully found most of those, which means our insiders can concentrate on giving us feedback about the game itself.

So far what I've shown you is what our testing looked like when we started full production on Sea of Thieves, next I'll show you how we optimised our testing during development.

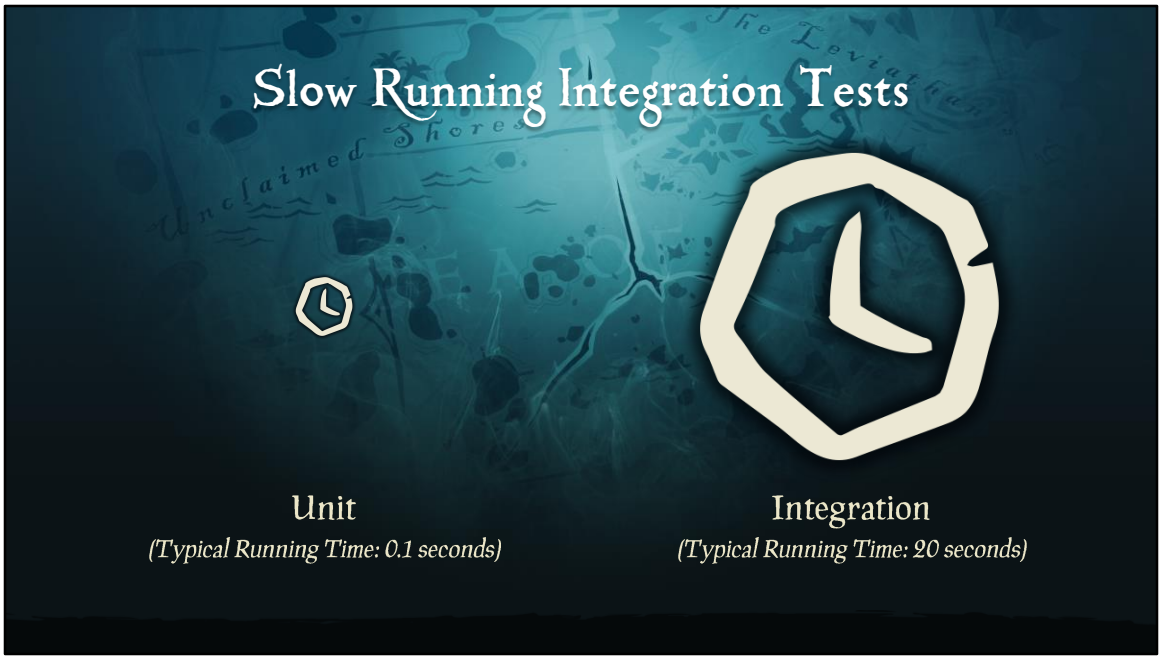


Or to give it another title, ‘



-- As we entered full production on Sea of Thieves, we now had a whole team contributing and running automated tests. We were seeing the benefits of the extra build stability, but we were also finding the tests to be quite a burden in several ways.
- First of all there was the time required in test creation. Obviously having engineers spending time working on tests, rather than contributing directly to the game, meant that features were being completed slower, but the time spent was greater than we'd have liked.
- A combination of the time individual tests were taking, as well as the large quantity of tests we had, meant that running them was causing a strain on our build system, and slowing down development. We had aimed for our 'pre commit' process to last less than 1 hour, but as the time crept beyond that due to slow running tests, developers were having to wait in a queue for half a day or more until their change was verified and they could submit.
- No set of automated testing will be perfectly reliable, especially not testing at the scale we were creating, but engineers were still spending more time than was bearable fixing flaky tests, with the worst culprits often being tests which were very long running, or complex.
Despite these issues, we still thought extensive automated testing coverage was a worthy goal. We just had to find ways to make the tests we created more efficient to

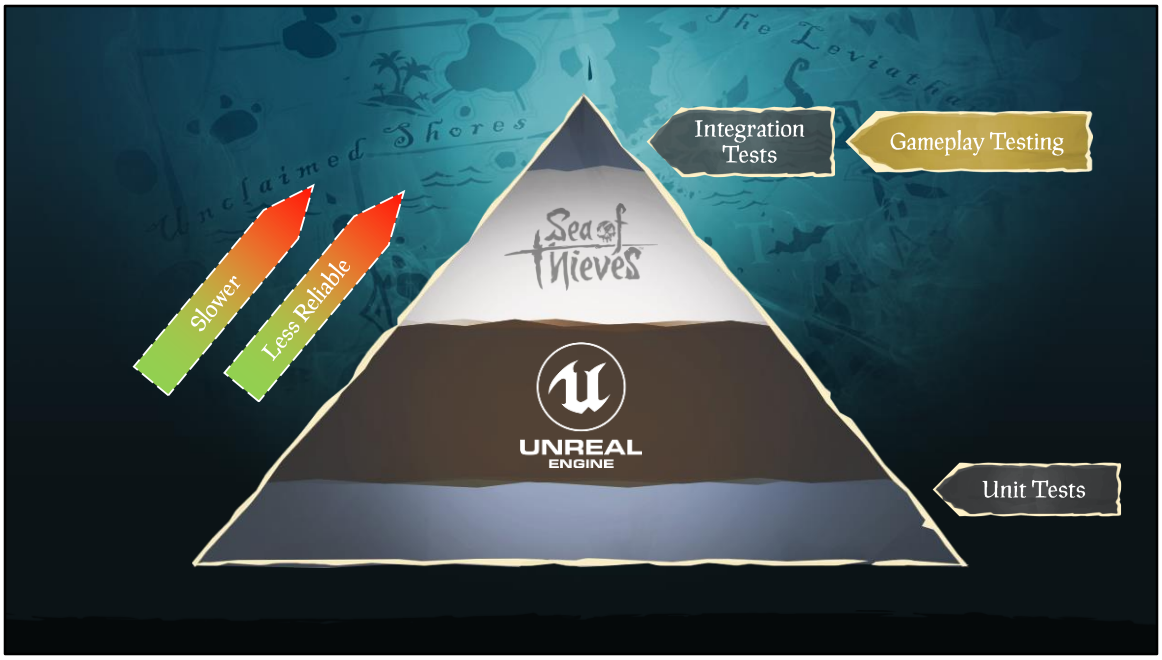
create and run.



-- For the three big issues we had with tests: creation time, running time and unreliability, almost all of them were much, much worse for our map-based integration tests. And the biggest issue of all was the time it took to run those tests.

- As a comparison, a unit test typically took up to a tenth of a second to run.
- Whereas, the time it took to run an integration test was around 20 seconds.

This gigantic difference was due to several factors, including the time needed to load all the assets required, and initialising a network connection in the case of a networked test. We still wanted to use integration tests as they would give us useful test coverage that we couldn't get elsewhere, but there was a need to improve them, and if possible use a lot less.

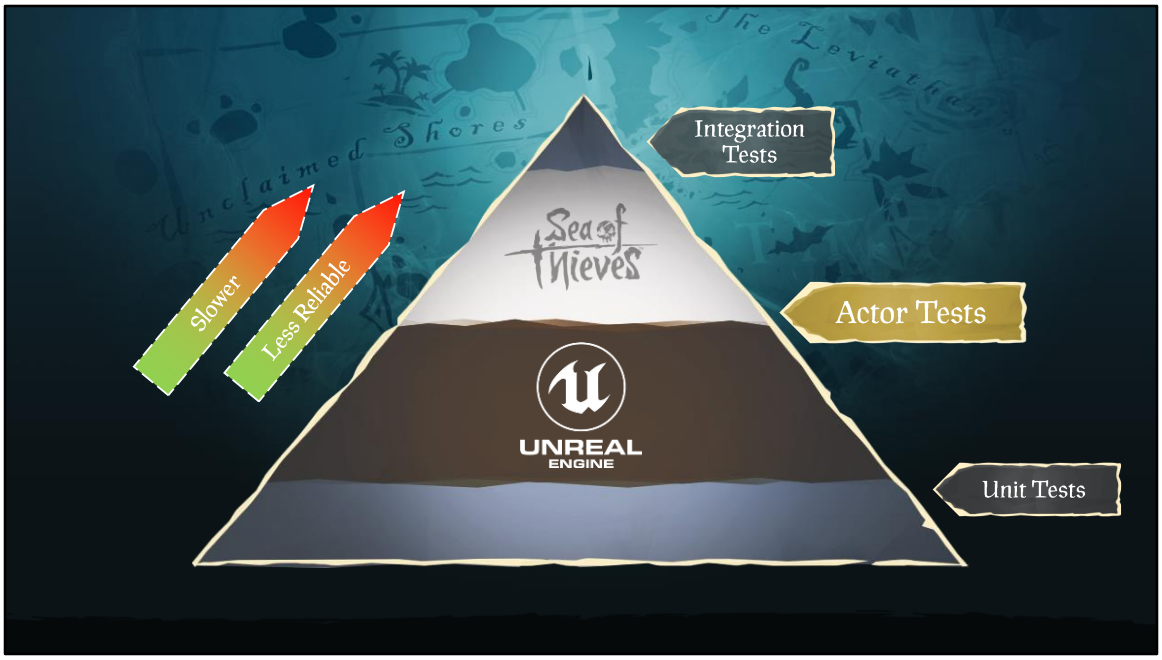


-- We were adding a lot of integration tests for gameplay features in particular. Why was that? If we imagine a pyramid showing the hierarchy of code the game is built on, we have the Sea of Thieves code sitting on top of the Unreal engine code.

- Unit tests, such as the vector distance test I showed you earlier, don't require unreal code so they sit here below the unreal codebase.
- Integration tests are at the highest level as they require a built version of the game or editor, which in turn relies on the whole Unreal engine.

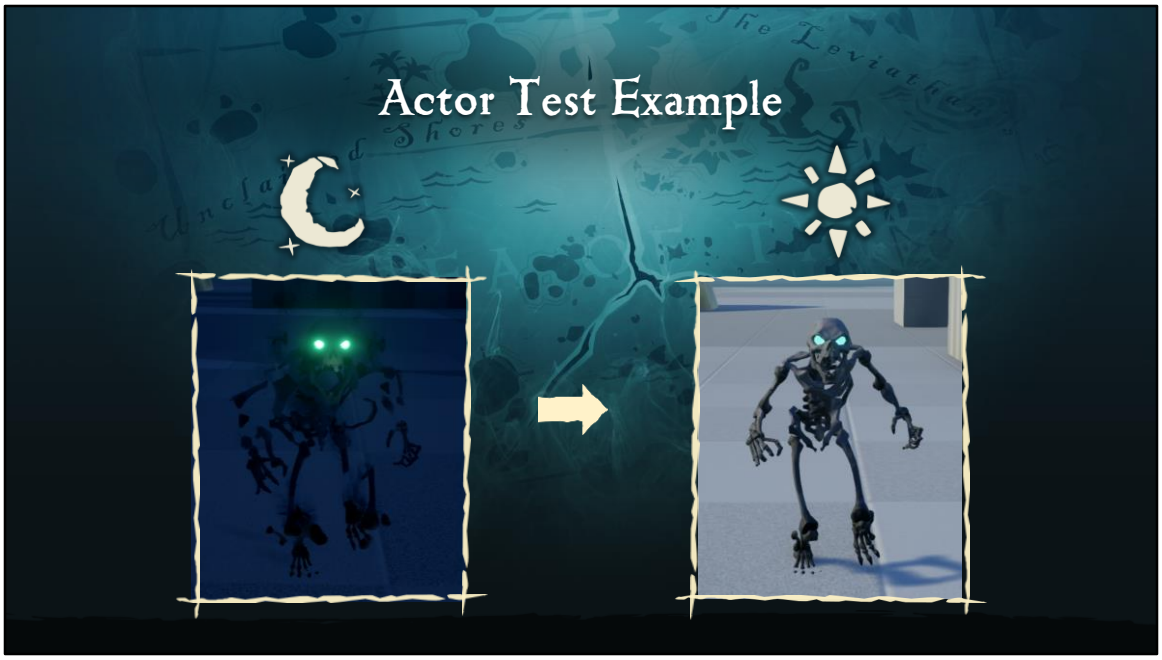
- The higher the level of the test, or the further up the pyramid in terms of this diagram, the more dependencies it has. This inevitably means that test will be slower and less reliable, so we wanted to only use integration tests sparingly and would prefer to use unit tests more.

- This was a particular problem for our gameplay code, which was built using common Unreal engine classes, like actors and components. To get test coverage which matched the game, we felt we needed to use these features in their usual environment, within an Unreal map, which meant that we tended to use integration tests heavily. When this started to become unsustainable though, we began to rethink our assumptions and thought about whether we could reposition our gameplay tests at a lower level instead that just checked the logic of a feature without all these dependencies.

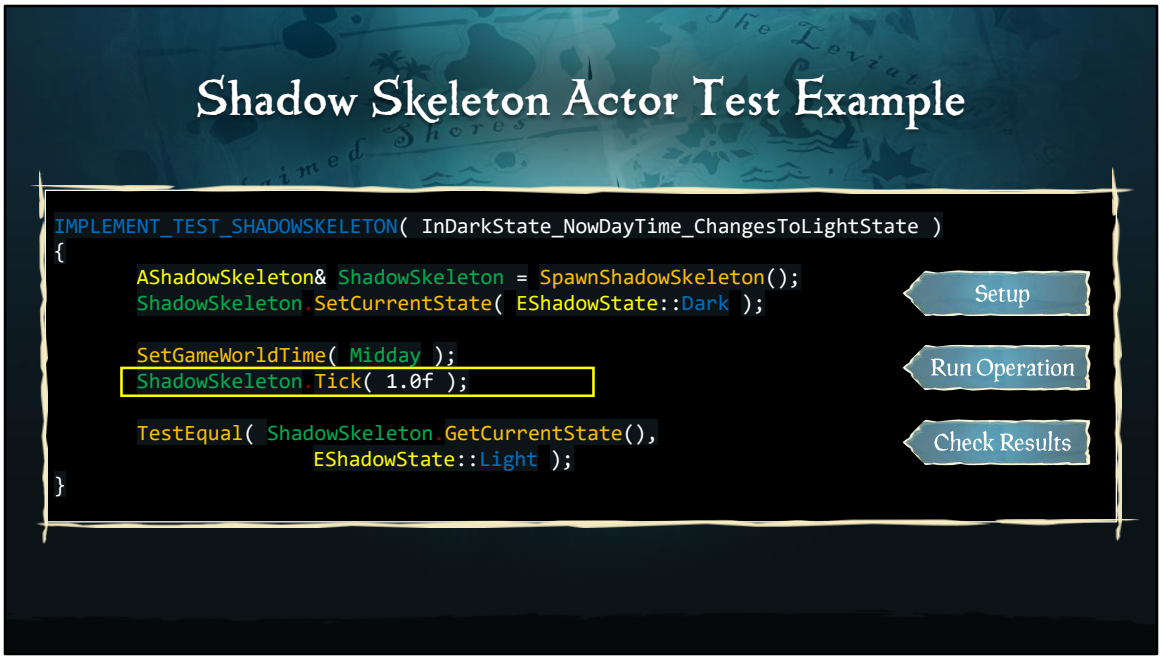


-- For this reason, we created a new type of test called ‘actor tests’, so called because they used the Actor Unreal game object class.

-- The actor test type gave us a useful middle ground between integration tests and unit tests. They were in effect a unit test for game code, but one that treated Unreal engine concepts like actors and components as first class dependencies. So not a unit test in the strictest sense, but engineers could treat them in a similar way.



-- As an example of an actor test, here's a typical game scenario we had during development.
- In Sea of Thieves, we have a variety of different skeleton types, one of which is the shadow skeleton. The shadow skeleton has two states, dark and light, with different visuals. Its current state is based on whether its currently exposed to light or not. When in darkness, its virtually invulnerable,
- whereas in light, it has the same vulnerabilities as a normal skeleton. We want to create a test that will check that a shadow skeleton shifts from its dark state to its light state when the time of day changes. We could've done this with an integration test, but doing it as an actor test is much more efficient.

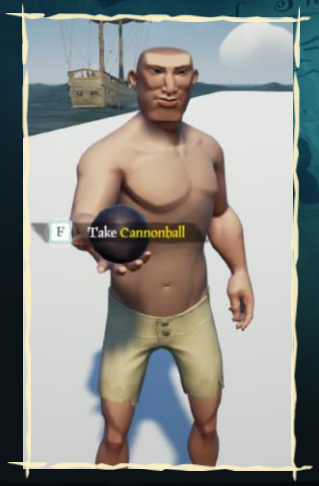


-- This is the actor test that checks that feature, and you can see it looks a lot like a unit test. Behind the scenes there is some more setup to create a minimal world for the unreal actors to use, but for the engineer creating a test, they can effectively treat it like a standard unit test.

- Like the other test examples, we have three phases. First we create a shadow skeleton and set its current state to dark. Then we run the behaviour we want to test, which in this case is setting the world time to midday. Finally in the check phase, we assert that the actor is now in the light state like we expect.

- Note this line here where we tick the shadow skeleton explicitly. We need to do this so the skeleton can detect the new world time and change its state. In the real game, this ticking would be done inside the engine's update loop, but to test it like that we would require a full integration test with all its associated costs. Having to explicitly call a tick like this is the downside of actor tests, as we're testing the object outside of its normal environment. The major benefit we get though is that this test runs incredibly quickly compared to running an integration test.

'Golden Path' Integration Testing



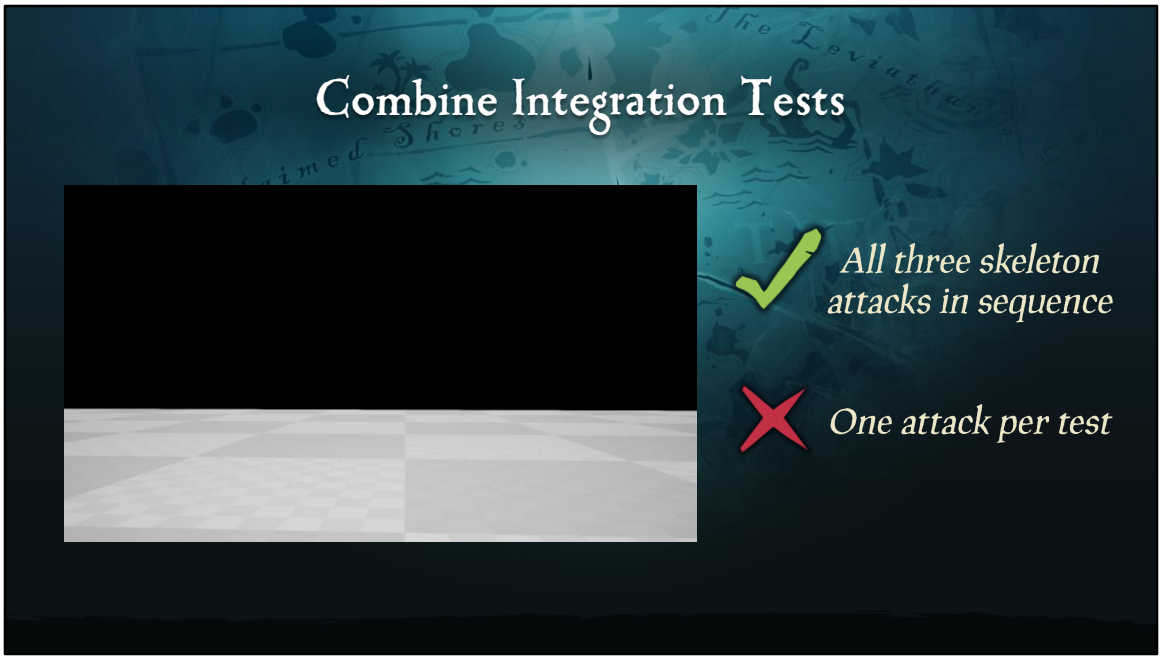
Actor	• *Player can't give item due to having no item* • *Player can't give that type of item* • *Player not close enough to give item*
Integration	• *Player successfully gives item to another player.*

Ratio of actor tests to integration tests is around 12 : 1.

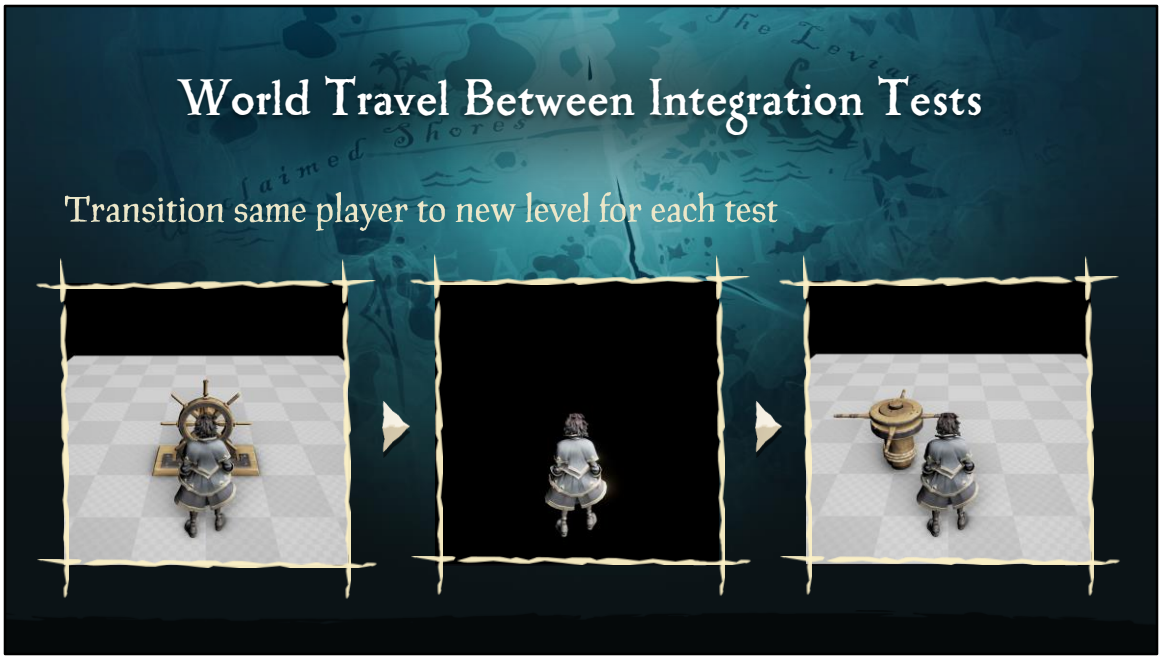
-- Even after adding actor tests, we didn't get rid of integration tests completely, as they gave us useful test coverage that other tests didn't, including checking asset setup and the interaction with engine code. But also due to their cost, we didn't want to create too many.

We settled on using integration tests for the successful or 'golden path' of a gameplay scenario, and using actor tests for the failure or edge cases. This gave us a mix that balanced fast execution time with reasonable test coverage.

- As an example, lets look at the feature in the game that allows players to give items to other players.
- This table shows how some of the tests that cover this feature were implemented. The failure cases where a giving action can't be done are checked with actor tests, whereas the successful giving case is checked with an integration test.
- In numeric terms, our ratio of actor tests to integration tests is about 12 to 1.

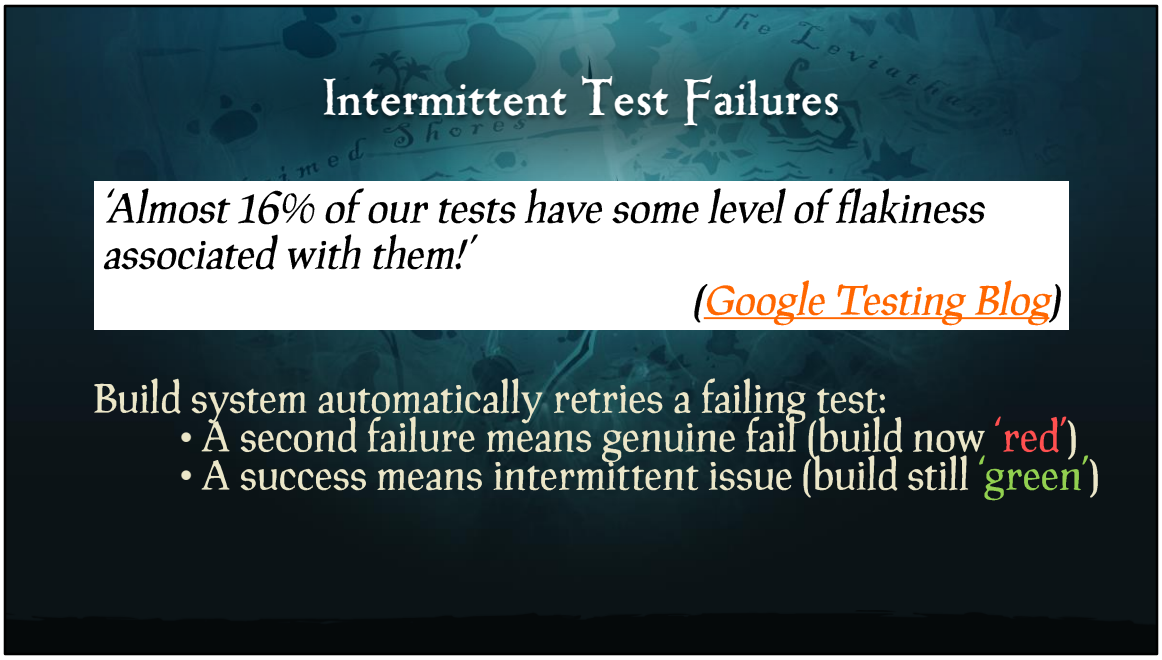


-- By using integration tests for only ‘golden path’ testing we reduced their number by a large amount, but we still wanted to improve the speed of those we had. One way we found to do this was to check multiple, related features in one test. This broke a general rule of testing to only test one bit of behaviour per test. But for integration tests, we decided that reducing the run time was more important.
- As an example, in this test we’re checking the three types of skeleton AI attacks in one test, running each one in a sequence, as opposed to running a separate test for each attack. By using just one test, we only pay the initialisation costs of creating a world, establishing network connections and loading all the skeleton game assets once, rather than incurring them multiple times.



-- Combining tests only works if the features they are testing are related. Even if they weren't, almost all our tests had the player character as a common element, which was one of the most expensive objects we had in the game. In many cases, the time spent loading and initializing the player with all its associated assets took longer than the test itself.

- To speed all our tests up, we made use of the world transition feature of the Unreal engine, to retain the player when we unloaded one test map and loaded the next one. So in this example, we take the player from a test map involving a ships wheel to another involving a ship's capstan. This stops the automated tests from incurring the cost of the player multiple times. Some care had to taken to not let state leak from one test to another, but as before, the speed up gain was more important.



-- Another issue we encountered were the intermittent test failures we found on our build system.
- Some level of test flakiness is inevitable when you're running a large amount of tests continuously, even for companies like Google, as you can see from this quote.
When we looked into the reasons for our intermittent test failures, they included a mix of: infrastructure causes, network issues and leaking state from previous tests.
We would investigate and fix the causes of these failures, but we found that stopping the team from checking in changes whenever we had one of these rare failures was too disruptive.
- Instead we decided that when a test failure is encountered on the build system, the failing test is automatically retried straight away. Only if the same test fails again on the second run do we turn the build 'red'. This stopped intermittent tests from blocking the whole team from committing changes.

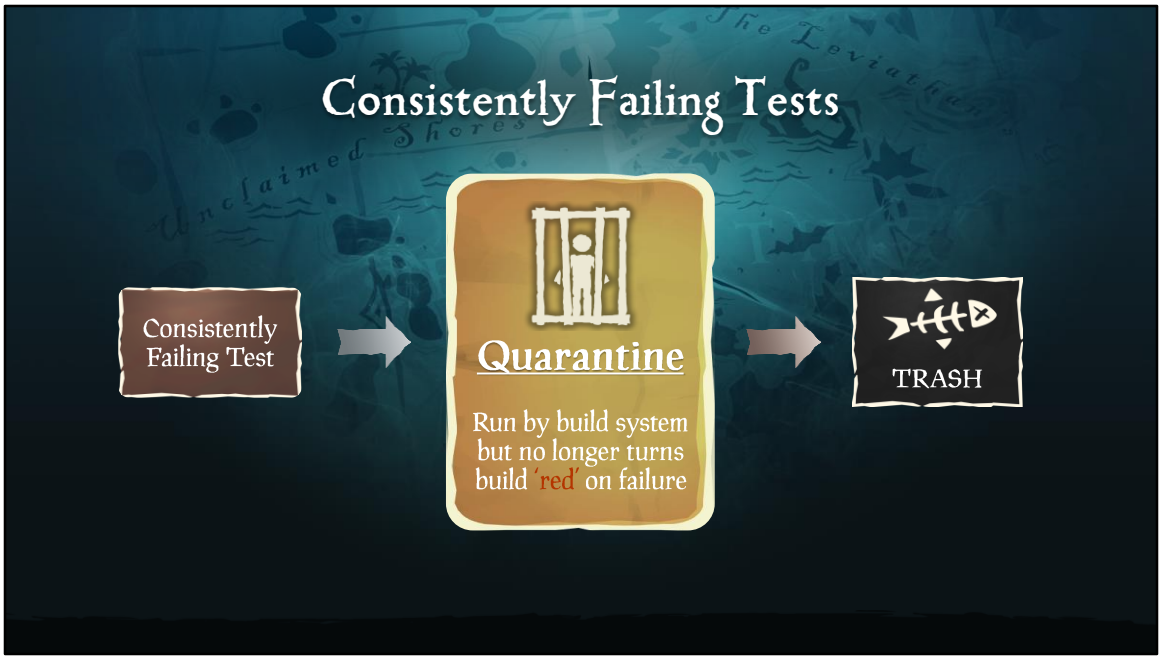
Top Intermittent Test Failures

Most failed Intermittent tests

Count of Intermittent Test Failures

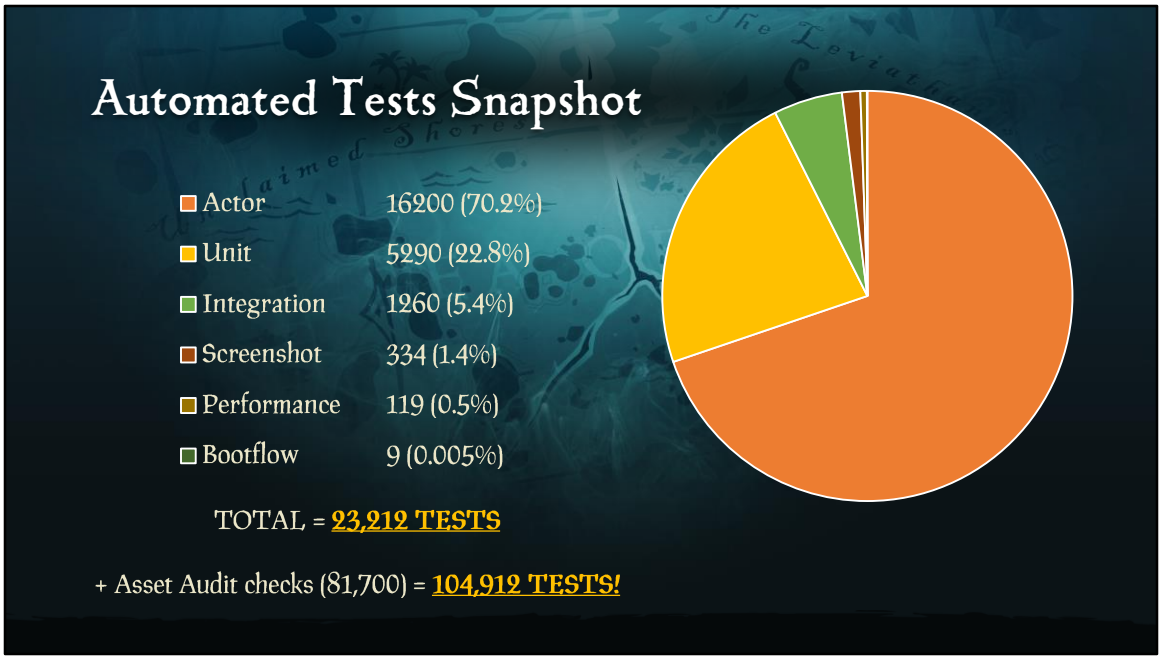
Name	TotalBuilds
Microsoft.Map Tests.MonkeyTest.Multiplayer.PopulateShips_WithMultiplePlayersAndRoles_DistributesPlayersAmongstAllShips_MP2	50
Microsoft.Map Tests.Game.AthenaGameMode.AthenaGameMode_WhenPlayerJoinsAfterShipStartsSinking_PlayerSpawnsAroundShip_MP0	44
Microsoft.Map Tests.MonkeyTest.Multiplayer.AllocateRoleToClient_WithShipPlayerAndSailAngleRole_ForcesInteractionWithSailAngle_MP1	17
Microsoft.Map Tests.MonkeyTest.Multiplayer.PopulateShips_WithFewerRolesThanPlayers_DoesNotDockExcessPlayers_MP2	14
Microsoft.Map Tests.UI.HUD.AthenaHUD_Initialization_AllCoherentViewsShouldBecomeReady_MP1	13
Microsoft.Map Tests.Athena.UI.RunAllJavaScriptTestsInV2Project	10
Microsoft.Map Tests.UI.Frontend.FrontendHUD_WhenCrewChanges_SendsCrewUpdateEvent	8

-- We didn't just ignore the intermittent failures completely though, we just wanted to be smarter as to where we concentrated our efforts. Those intermittent failures could be due to genuine flaws in either the test or the feature being tested.
- To help keep on top of this, we kept track of each test failure and drew up a list each week that showed us which tests were failing the most. The tests at the top of the list were the ones that were more likely to be genuine issues that were worth assigning an engineer to investigate.



-- The final change we made to our test infrastructure was the handling of tests that were consistently failing, generally due to them being badly written.
- Tests like this can't be trusted and are often worse than having no test at all, due to the time they waste on the build system and the false information they give to the team.
- If a test begins failing regularly over a certain amount of time, the build system moves that test into quarantine. This meant that the test was still run, but if it failed it would no longer turn the build red and stop the team committing changes. The engineer who was responsible for the test is then notified that this quarantining has happened and is encouraged to fix the test as soon as possible. If the test is not fixed by a certain amount of time, the test is automatically deleted.
- This sounds harsh, but the thinking was if engineers are unable to prioritise fixing a test, then what it's testing is probably not important enough to warrant the drag its having on the build system. So in the bin it goes.

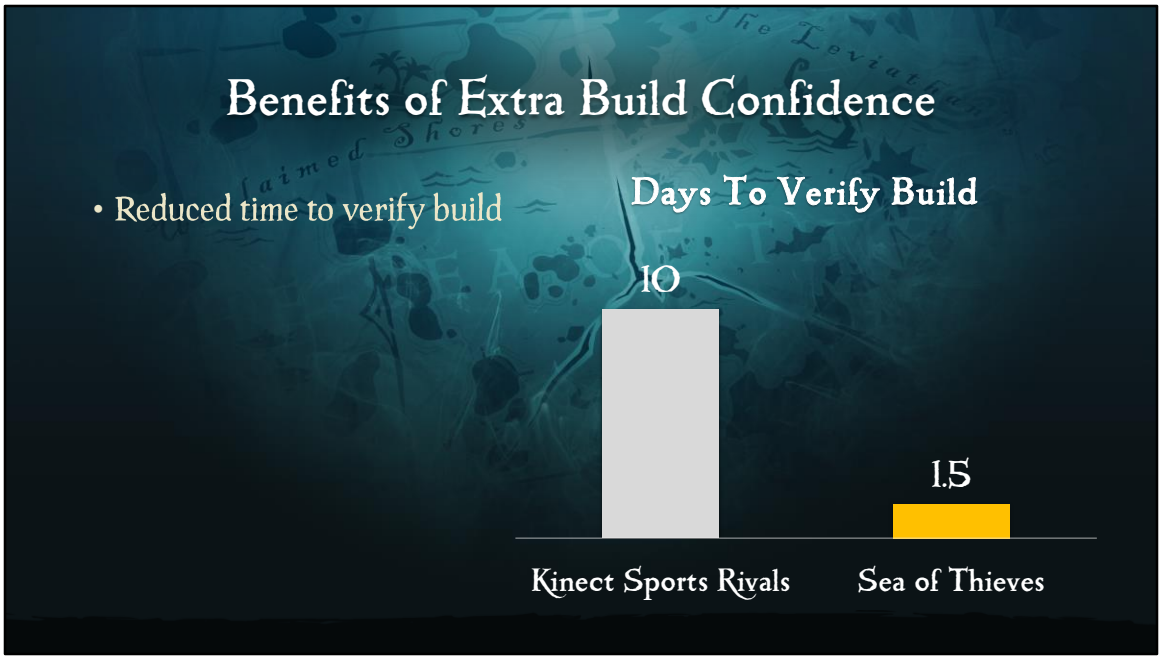
-- For the final part of this presentation, I'd like to talk about the benefits automated testing gave us on the project.



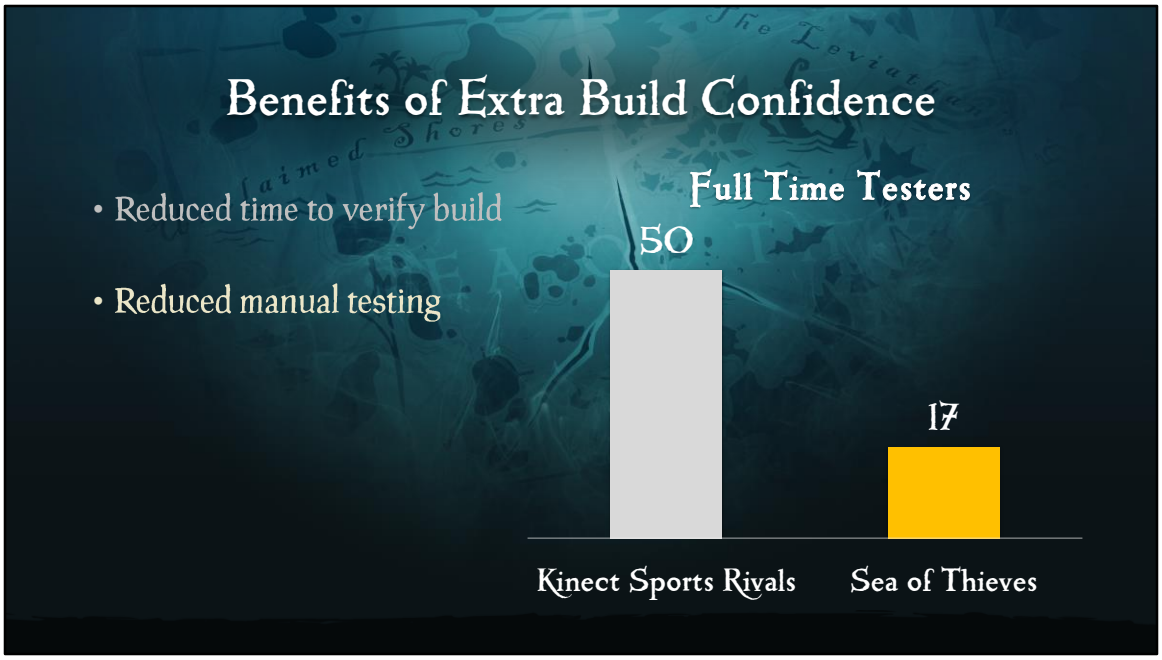
-- Here is a breakdown of the tests we currently have in our codebase.
You can see that 70 percent of our tests are actor tests, which makes them the most common type by far. This was because it was the most convenient test type to provide coverage for our gameplay features.
Only 5% of our tests were integration tests, but they provided vital high level coverage of game features. Roughly half of those were networked and checked communication between the client and server too.
We only created a relatively small amount of performance, screenshot and bootflow tests. These were by far the slowest, so we used these as sparingly as possible.

- Adding all this up we end up with just over 23 thousand tests, which is quite a lot. But this doesn't include all the asset audit checks that we do. I didn't include these in the graph as I didn't want to skew it too much.
- If those are included too, the total number of tests becomes over 100 thousand! This amount of tests would've overwhelmed our build system and slowed development down to a crawl if we hadn't put a lot of effort into making the tests efficient to run.

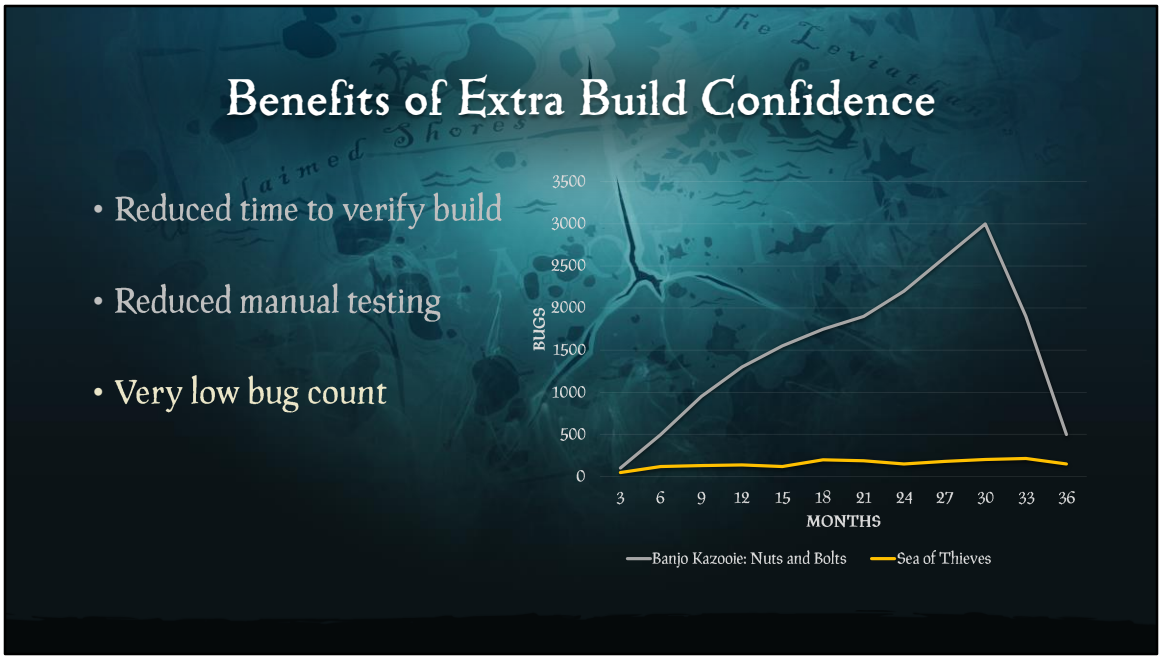
[And as for the amount of test coverage we had over the codebase, we ended up with 60% decision and 70% function coverage]



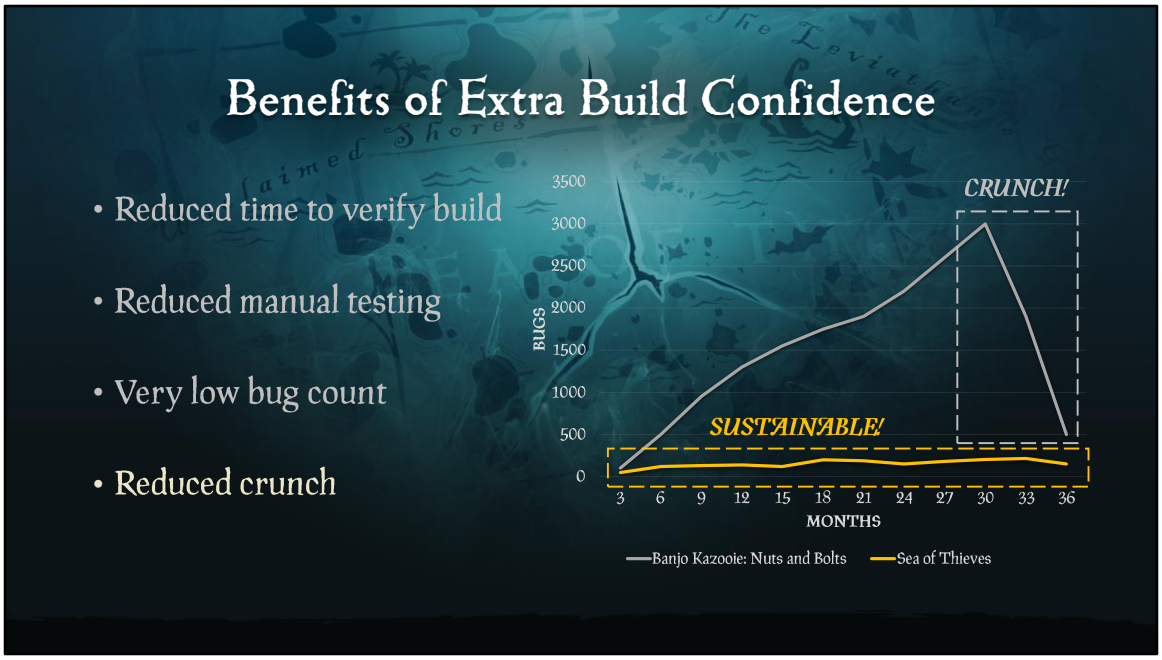
-- So with all those automated tests we'd added, we now had much greater confidence in our build at all times, and this had a number of great benefits.
- First of all, we were now able to create and verify builds in a day and a half. For an evolving game like Sea of Thieves, this means we can respond rapidly to player feedback if we needed to, or add a hotfix. In contrast during development on the last game I worked on Kinect Sport Rivals, it would take a full two weeks to verify a build.



-- The desire on Sea of Thieves was to have less testers than on our previous projects, and we managed that thanks to the amount of automated testing we used. In terms of numbers, on Kinect Sports we were using 50 full time testers at the time of release, whereas on Sea of Thieves we had 17. Considering how much more complicated Sea of Thieves is, this is quite a significant difference. Having less testers saves money obviously, but what the number doesn't show is how we were also able to make much more effective use of this smaller test team, getting them to work more closely with the rest of the development team to assess the game from a player's perspective, all thanks to repetitive checks now being done by automated testing.



-- We also managed to maintain a very low bug count throughout development. The max bug count we achieved on sea of thieves was 214, whereas on Banjo Kazooie Nuts and Bolts it was nearly 3000. Looking at the graph of bug count over production time, you can see that on our older project the number of bugs kept rising gradually till they were cleared near the end of the project, whereas the number of bugs on Sea of Thieves remained fairly steady. Because automated testing was always being run, we knew straight away when a known problem had entered the build. So many issues were fixed at the point they were introduced and never made it as far as a bug. For those new bugs we hadn't anticipated, we used a development policy that bugs should be prioritised before feature work, which prevented the number from mounting up too much.

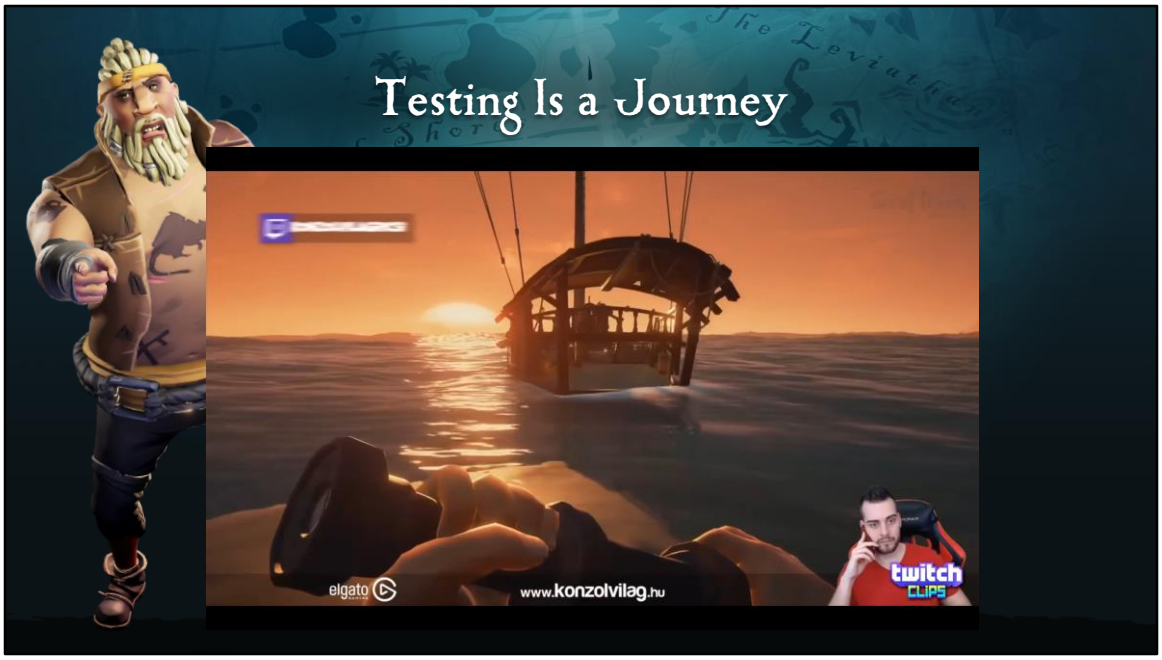


-- Reducing crunch is really important to us at Rare, and we hoped to do better in this regard on Sea of Thieves than on previous projects. I don't have concrete stats for this unfortunately, but anecdotally developers at the studio found that they worked much less overtime on this project and we believe that our automated testing was a big reason for this.

On previous games, we'd left long running problems in the build, until a time came when the game needed to be released or shown publicly, at which point all these bugs needed to be fixed before a deadline, which often meant the team had to crunch. Instead on Sea of Thieves, with automated testing highlighting issues as soon as they appear, there's a more consistent focus on build quality throughout development, which meant less need for these intense periods of bug fixing and optimisation, and developers could maintain more regular working hours.

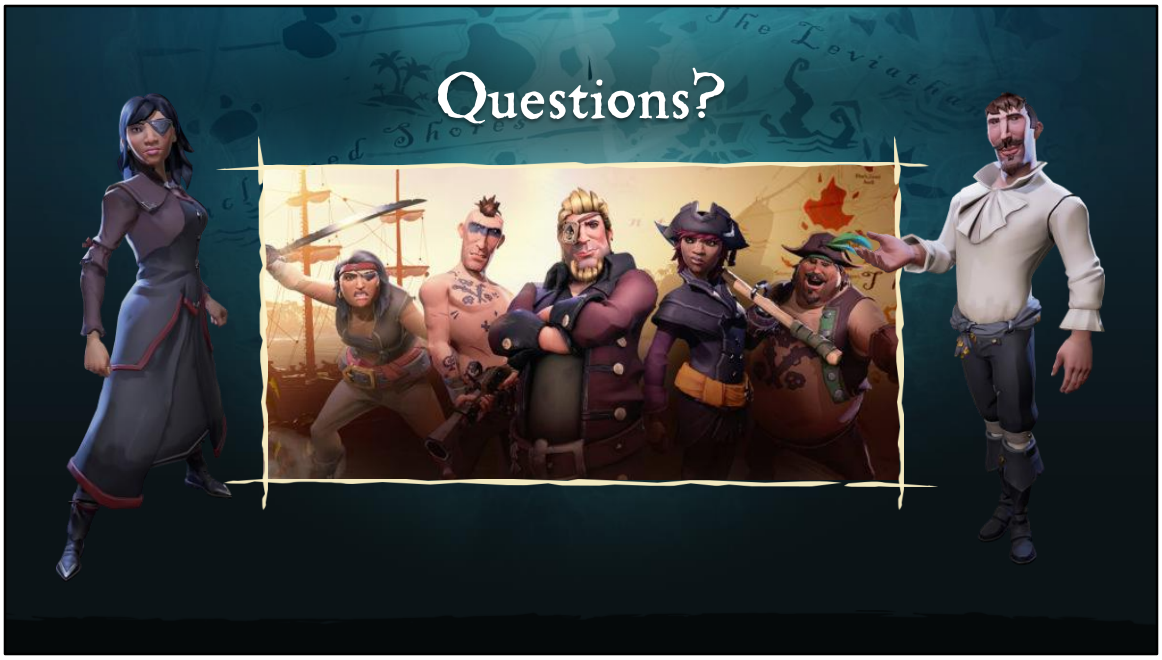
-- Finally, here are some bonus lessons we learnt on our automated testing adventure that I'd like to pass on.
- The most common reason not to use automated testing are concerns about the time it takes, time which could instead be spent working on the actual game. My counter argument would be that over the long term, speed of development will be about the same, as the team will be spending less time fixing reoccurring bugs. We were lucky that everyone on the project, including producers and project managers, agreed about the benefits that automated testing would give us. Without that, it would've been very difficult to achieve what we did.
- Don't underestimate the time it will require to get your team up to speed on how to write tests well. Also the infrastructure for running tests will need time to make it stable and efficient, as you'll be heavily reliant on it. Starting up automated testing on this project took a while, mostly because we were adding testing to our whole codebase. To get started quicker, adding testing to just one part of your game project to begin with, may be easier.
- Automated testing is more of a hindrance than a help if you're still iterating on your game to find if its fun. On Sea of Thieves, we maintained a separate prototype branch without automated testing, where the team could try out ideas. Even when working on production code, we rarely created tests first in the test driven development style.

The workflow for a developer was almost always to make a change, see if it works and then create tests to pin down its behaviour for the future.
- As a team it took us a while to settle on the best practices for how to use automated testing, and we're still evolving our processes even now. Many times, we had to abandon the text book way to do testing and be more pragmatic when that way wasn't working for us. Also tests have a cost, so focus on creating tests for areas that are likely to have bugs and avoid creating tests that will need a lot of work to maintain or that are overly complex. Perfect testing coverage is a worthy goal, but unachievable in practice.



-- And finally, just to prove our testing isn't perfect, here's one of my favourite videos of a bug that made it through to our live game.
[after vid finishes] What can I say, testing is a journey and we're always striving to improve.

-- That's the end of my talk. What I've talked about here represents a combination of work from lots of people, so I'd like to thank everyone at Rare for helping to contribute to our testing process on Sea of Thieves.
And Rare are hiring, including in many software engineer positions, so please look up job opportunities on our website or come and talk to me at the conference.



-- And that’s the end. Does anyone have any questions?

There are some great resources online about using automated testing specifically for games, that I've added here if you want to find out more.
If you want to go further and look for advice about testing outside game development you'll be spoilt for choice, here are just a few examples.