Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Mikael Korpinen

Yksikkö- ja integraatiotestaus Unity-pelimoottorilla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto ja viestintä -tekniikka

Insinöörityö

2.2.2020

Tekijä Otsikko Sivumäärä Aika	Mikael Korpinen Yksikkö- ja integraatiotestaus Unity-pelimoottorilla 24 sivua 2.2.2020
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotekniikka
Ohjaajat	Heini Puuska Miika Mäkiuuro

Työssä käydään läpi yksikkö- ja integraatio testausta ja näiden tekniikoiden antamia hyötyjä niin koodauksen, kuin liiketoiminnan kannalta. Pohditaan esimerkiksi sitä, mitä etuja ja käyttökohteita voitaisiin testiautomaatiolla saavuttaa ja milloin testiautomaatio kannattaa ottaa käyttöön. Pohdintaa ja kokeilua siitä, miten testejä kannattaa luoda, käyttää ja mitkä ovat testiautomaation käyttöönoton riskit. Voiko ilman testejä tulla toimeen ja ohjelmoida testejä ilman ongelmia ajan kanssa? Työssä sukelletaan myös testien tekniseen puoleen Unity-ympäristössä.

Työssä tehtiin yksinkertainen projekti, jossa testattiin Unity-pelimoottorin työkalua yksikkö- ja integraatiotestien ajamiseen. Testiprojektin tuloksena oli testiautomaatio, jolla pystytään testaamaan eri aseiden vuorovaikutusta ympäristön kanssa. Työ rajattiin yhteen pelimoottoriin, vaikka kokemusta löytyy myös WebGL-ympäristöstä.

Tämän lisäksi syntyi hyvä ajattelumalli testien automatisointiin ja sen hyötyihin ohjelmoinnin tukena sekä hyvä automaatio aseiden testaamiseksi virtuaalimaailmassa.

Avainsanat	Testiautomaatio, liiketoiminta, yksikkö ja integraatiotestaus, pelimoottori

Author Title Number of Pages Date	Mikael Korpinen Unit and Integration testing with Unity game engine 24 pages 2 February 2020
Degree	Bachelor of Engineering
Degree Programme	Information technology
Professional Major	Software Engineering
Instructors	Heini Puuska, Title (for example: Project Manager) Miika Mäkiuuro, Title (for example: Principal Lecturer)

The purpose of this thesis is to go through unit and integration -testing and the benefits of these technologies. For example. What are the advantages and use cases of test automation? Thinking about the best way on how to create these tests, how to use them, and what are the risks of implementing automated testing. Can you work without tests and how to program tests without problems with time?

Result of the test project was a system that can automatically test a specified number of weapons against environment. Thus, saving time on the testing when the project starts to grow.

In addition to this the thesis resulted in a logical way of thinking about test automation and its usages.

Keywords	Test automation, business, unit and integration -testing, game-engine

Sisällys

Lyhenteet

WSA Windows käyttöjärjestelmän kaupan sovellus.

WebGL Khronos-järjestön tekemä nettiselaimella toimiva grafiikkakirjasto.

Nunit .NET-kielisten alustojen suosittu yksikkötestikehys.

3d Kolmiulotteinen.

Tekninen velka Sovelluskehityksessä käytetään termiä tekninen velka tilanteisiin, joissa aikataulu ei salli koodin laadun parantamista. Toteutus tehdään kiireellä eikä koodin laadusta ehditä välittää. Testejä ei kirjoiteta, koska toiminnallisuus tai korjaus on saatava nopeasti valmiiksi.

1 Johdanto

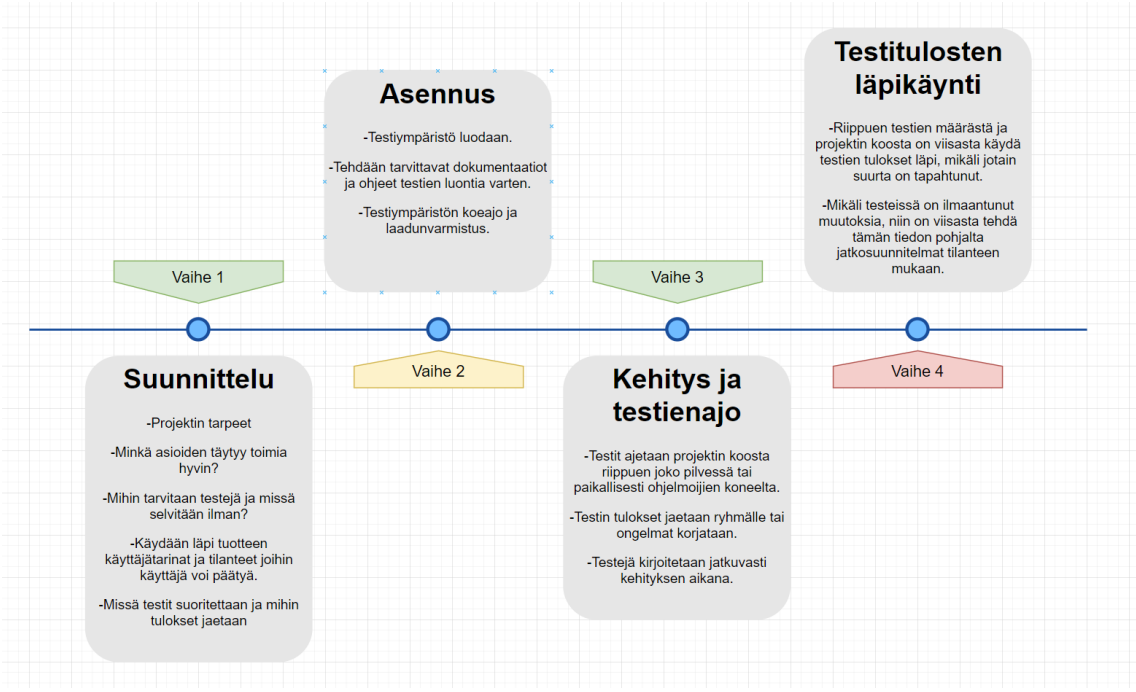
Työn aihe on tullut tarpeesta tehdä pelejä ja sovelluksia tehokkaasti ja luotettavasti. Lisäksi haluan oppia, tutustua ja löytää helposti lähestyttävän ja yleishyödyllisen tavan ajatella sekä toteuttaa testiautomaation perustaa liiketoiminnalle. Tästä syystä en lähtenyt alan kirjallisuutta opintomateriaaleja enempää tutkimaan, vaan enemmänkin työkokemuksen, koulutuksen, tutoriaalien, pohjamateriaalin, nettisivujen ja reflektoinnin kautta hakenut omaa hyvää tapaa toteuttaa perusta yleishyödylliselle testiautomaatiolle. Työn tarkoituksena ei ole ollut tutkia ja kehittää kaikista nykyaikaisinta perustaa testiautomaatiolle eikä välttämättä tutkia, mikä on se viimeisin trendi. Sen voi tehdä jatkona. Työssä käydään läpi yksikkö- ja integraatiotestauksen työkaluja ja menetelmiä. Tämän jälkeen käydään läpi, kuinka pitkälle ja mitä kaikkea pystytään koneellisesti testaamaan. Testit ajetaan Unity-pelimoottorin testinajajalla. Yksikkö- ja integraatiotestikehyksenä toimii NUnit.

Työn aihetta valittaessa mietin paljon, mikä olisi sellainen alue, jota en osaa ja johon ei ole löytynyt aikaa, mutta joka kuitenkin voisi olla iso valttikortti, kun sen kultaisen langan tajuaa ja sen toteutuksen osaa. Kun aloitin työelämän, testiautomaatio oli tullut esille niin työelämässä ja koulussa muutamaan otteeseen. Aihe on mielestäni mielenkiintoinen, sillä on hyvin vaikeaa ymmärtää, minkä vuoksi kalliin ajan vaihtaminen automaatioon, joka testaa, että koodi toimii, kun sen voi nähdä itsekin kokoamalla sovelluksen. Tuote täytyy käsin testata joka tapauksessa. Aihetta tutkiessa tuli vastaan väitteitä kuten testejä koodattaessa oppii kirjoittamaan parempaa koodia ja että se on myös hyvä tapa tutustua sovelluksen toimintaan. Ohjelmoitaessa olen huomannut, että useasti nopea kokeilu osoittautuukin isommaksi työnaiheeksi, kun tämä päätetään toteuttaa itse sovellukseen tai peliin. Yksinkertainen toiminnallisuus onkin loppujen lopuksi iso kokonaisuus, testattavien ominaisuuksien määrä kasvaa pelin tai sovelluksen kehityksen edetessä. Kun ihmisiä on enemmän kuin yksi, voi jatkuvien muutoksien seuraaminen olla haastavaa. Tuotteita ja pelejä voi olla useita. Joitain asioita on mahdollisesti testailtu paljon ja ollaan jo varmoja niiden toimivuudesta. Ongelmat tulevat esille jossain vaiheessa, kun jotain halutaan muuttaa tai päivittää. Osat eivät enää toimi keskenään, mutta missä on vika, kun liikkuvia osia on kymmenistä satoihin ja koodia useita tuhansia rivejä. Osa on uusien koodaajien tekemää ja osa hyvin vaikeasti luettavaa ja uusimpia ominaisuuksia käyttäviä koodinpätkiä. Jotkin koodin osa-alueet voivat olla jopa hätäisesti koodattuja.

Kuinka tällainen tilanne voidaan älykkäästi hoitaa? Miten saadaan sovellus pysymään kasassa niin, että sitä ei tarvitse 10 vuoden päästä heittää roskakoriin?

2 Testiautomaatio kehityksen tukena

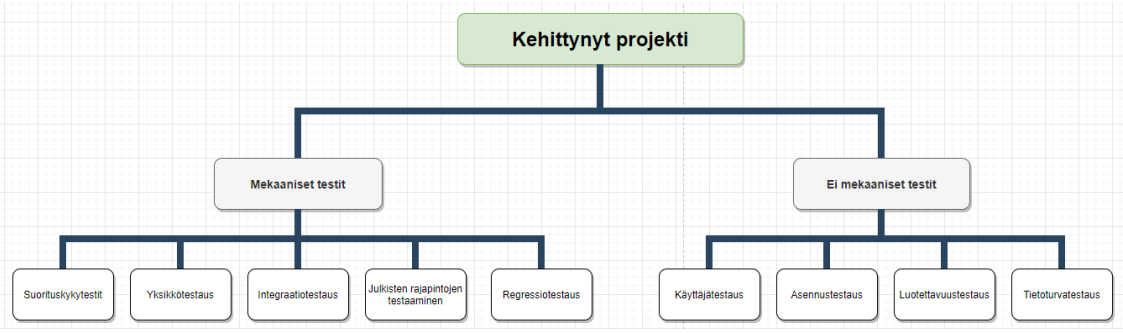
Yksikkö- ja integraatiotestaus on osa mekaanista testausta. Yksikkö- ja integraatiotestiautomaation avulla voidaan varmistaa, että ohjelmiston osat matalalla ja korkealla tasolla toimivat edelleen odotetusti. Pelaajien ja käyttäjien suosion kannalta on tärkeää, että peli tai sovellus on ehjä ja toimii vähintään riittävällä tasolla, sillä kuka haluasi pelata peliä tai käyttää sovellusta, joka ei vain toimi. Testiautomaatiota voidaan käyttää työkaluna laadunvarmistuksen, hyväksymistestauksen ja kaiken muun testauksen sivulla. Esimerkiksi, pelin sääntöjen tai sen mekaniikkaan vaikuttavien parametrien tulosten todentaminen.



Kuva 1. Mahdollinen kehittyneen ympäristön testien automatisoinnin kulku. Kuvassa käydään lyhyesti läpi, miltä ohjelmointiryhmän testipuoli voi näyttää. Inspiraatiota kuvan tekemiseen Sayantini Debin blogista (Edureka) ja tosielämän kokemuksista. Vastaavanlaisia kuvia voi löytyä myös muualta internetistä.

Testiautomaation avulla voidaan pelin tai sovelluksen kehitysprojektin koon alkaessa kasvamaan tarkistaa, täyttyvätkö pelin tai sovelluksen vaatimukset peli- tai sovellusmekaniikkaan tehtyjen parametrien muutosten jälkeen. Esimerkiksi, voiko käyttäjä edelleen maksaa laskunsa tai onko ohjelmistoteknisen työkalun lopputulos edelleen sama? Pääseekö pelaaja tarinassa eteenpäin?

Onko jonkin esineen tai pelimaailman toiminnon testaamisen automatisointi ajan, resurssien, testaajan, koodaajan ja turhautumisen säästämistä pidemmällä aikavälillä projekteissa, joissa tarvitaan luotettavuutta? Usein sovellus tai sen osa on rikki riippuen siitä, mitä ja miten sitä kehitettään. Olisiko järkevää vain automatisoida kaikki mihin pystyy? Testaaminen vie aikaa, resursseja ja se voi myös helposti unohtua kiireen ja kuormituksen keskellä, kun on kiire jo saada seuraava tehtävä valmiiksi. Riskinä on arvokkaan ohjelmointiajan kuluttaminen automaatiotesteihin. Tästä syystä on hyvä, että testien kirjoittaminen sujuu rutiinilla, sillä testien kirjoittamiseen liittyy kompastuskiviä.



Kuva 2. Esimerkki kuvitteellisen ja kehittyneen ympäristön testiperheestä. Ei mekaanisia testejä, kuten nappien painalluksia voi nykyään suorittaa pilvessä koneoppimisen tai robotiikan avulla. Periaatteessa kaiken pystyy automatisoimaan. On kysymyksen arvoista, kuten mikä on tietyllä hetkellä järkevää automatisoida. Jälleen inspiraatiota kuvan tekemiseen haettiin netistä Sayantini Debin blogista (Edureka) sekä tosielämän kokemuksista. Kuvassa eroaa Sayantinin versioon se, että sen sijaan että jaoteltaisiin funktionaalisiin ja ei funktionaalisiin, niin toisella puolella on usein koneella tehdyt mekaaniset testit ja jälkimmäisenä ei mekaaniset testit, joissa usein käytetään ihmistä kokonaan tai automaation apuna. Tuloksena kuva, joka on mielestäni todellisuutta lähempänä. Huomion arvoista on myös, että ei mekaanisesti suoritetut testit ovat usein vaikeasti automatisoitavia ja joskus myös vaikeasti luotettavia tietokoneen haltuun.

Mielestäni on suuri etu, jos pystytään sanomaan, kuinka iso pelisovelluksen kriittisistä osista ja järjestelmistä on toiminnassa päivityksen jälkeen. Voidaan jopa väittää, että nykypäivänä on välttämätöntä, kun sovellus tai peli kasvaa liiketoimintana huipputasolle.

Kuinka muuten käyttäjät saavat tiedon tilanteessa, jossa uudessa epävirallisessa kokeiluversiossa ei kaikki yhteensopivuudet toimi? Nykyään yhä useammat pelimoottorit tukevat useita eri käyttöjärjestelmiä ja alustoja. Tästä syystä on hyvin kätevää, jos palvelin voi tehdä projektin käännöt ja hoitaa pakolliset testit. Tämä on hyödyllistä, sillä testaajien määrää ei tarvitse lisätä, ja olemassa olevat testaajat voivat keskittyä kriittisimpiin osa-alueisiin. Vaihtoehtona työntekijät manuaalisesti kääntävät projektin jokaiselle kohdealustalle ja ajavat testit rutiininomaisesti. Tästä syystä, kaikki ominaisuudet, jotka ovat kriittisiä sovelluksen tai pelintoiminnan kannalta tai joiden testaus kuluttaa huomattavasti resursseja, kannattaa automatisoida, jos aikataulu antaa siihen mahdollisuuden tai ei ole varaa ottaa teknistä velkaa. Myös suorituskykytestit kannattaa ottaa sovelluksen alkuvaiheessa käyttöön, jotta tämän kehityksestä on automaattisesti tallennettua tietoa ja sovelluksen suorituskyvyn tippumiseen ehditään puuttua oikealla hetkellä. Testejä siis voi ja kannattaa kirjoittaa suorituskykyä mittaamaan. Täydellisessä maailmassa kaikki toimii salaman nopeasti, mutta todellisuudessa jonkin toiminnon hitaus voi olla hyvinkin ymmärrettävää, esimerkiksi isojen tiedostojen siirto tai kentän lataus.

Testiautomaation muita hyötyjä on esimerkiksi koodin parantunut huolto. Jos uusi muutos hajottaa testin, niin tiedetään, mistä vika löytyy, ennen kuin ongelma tulee ikävästi esille. Näitä testejä kannattaa tehdä miettimällä käyttäjäkokemuksia ja kokonaisuuksia.

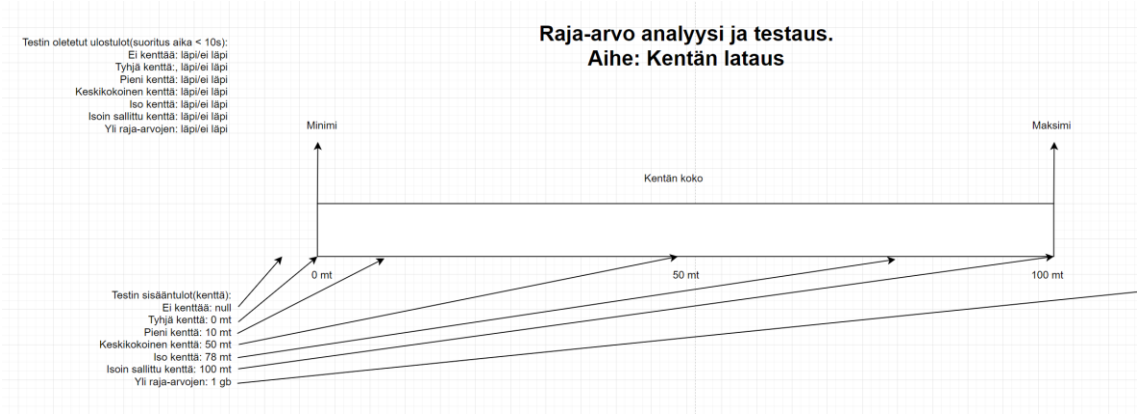
Kuvitellaan tilanne, jossa ollaan tutkittu markkinoiden tarvetta uudelle sovellukselle/pelille. Vastaavanlainen on jo olemassa, mutta se on hyvin kallis ja voisi olla huomattavasti parempi. Jotta pelaajat/käyttäjät saataisiin houkuteltua uuden tuotteen pariin, on sen hyvä olla edeltäjäänsä myös osittain parempi. Tästä syystä määriteltiin projektin käynnistyessä pelille joitain vaatimuksia. Esimerkiksi tilanne, jossa käyttäjän tason lataaminen tai tietokonemallin avaaminen tulee tapahtua alle 10 sekunnissa. Vaatimus määriteltiin kokonaisuutta varten, joka mahdollisesti tarvitsee toimiakseen useiden toimenpiteiden suorituskyvyn arviointia. Suorituskykytestit eroavat yksikkö- ja integraatiotesteistä siten, että näillä mitataan ylös aikaa. Sen vuoksi ko. esimerkissä testit kirjoitetaan siihen tarkoitetuilla yksikkö- ja integraatiotestikehyksillä.

Esimerkki tapauksessa testit voivat paljastaa muistivuodon, joka ajan myötä olisi voinut kasvaa isoksi ongelmaksi, hidastaa tai kaataa sovelluksen. Muistivuoto saadaan esille suorittamalla jokin toiminnallisuus, sen luonti ja mahdollinen muistin vapautus useita kertoja peräkkäin. Muistivuodossa toiminnallisuus varaa tietokoneelta muistia, mutta jostain

syystä, kun muistialuetta ei tarvita, niin sitä ei vapauteta. Automaattisen roskienkerääjän tehtävänä on käyttämättömien muistialueiden vapauttaminen, mutta roskienkeruu löytää viitteitä aktiiviseen koodiin, joten se ei kyseenomaista muistialuetta vapauta. Koodin viallisuus ei välttämättä tule esille suorituskykyä mittaavissa testeissä lainkaan, jos esimerkiksi mitataan vain, montako millisekuntia jokin toiminto kestää.

2.1 Testikehykset ja näiden tarkoitus

Ohjelmistotekniikassa testejä voidaan myös tehdä koodin toiminnallisuuden määrittelyä varten. Kun mietitään testitapauksia ja mitä testien hyväksytyksi saamiseksi tarvitaan, saadaan koodin taso uudelleen käytettävämmäksi ja käyttäjätarinat testattua. Onhan toiminnallisuudella jo uusi käyttäjä, testit. Tämä on mielestäni nopeampi tapa tehdä toiminnallisuuksia, sillä se ei tarvitse suunnitelmien luomista ja hyväksymistä. Se on toiminnallisuus, joka tarvitaan sovellukseen tai peliin. Sitä varten kirjoitetaan toiminnallisuus ja testit. Jos toiminnallisuus hyväksytään tuotteeseen työntekijöiden sekä käyttäjien puolesta, se on koodiriveiltään kasvanut isoksi, voidaan tämän jälkeen miettiä suunnittelumalli, joka sopii parhaiten tilanteeseen. Riskinä on, että tilanne kääntyy toisinpäin. Ohjelmoija ei tiedä mitä halutaan, joten jälki on myös samansuuntaista. Mielestäni hyvä tapa onkin suunnitella käyttöliittymä ja toiminnallisuus erikseen. Mikäli käyttöliittymän muutokset hajottavat toiminnallisuutta, niin testien ansiosta huomataan se parhaimmassa tapauksessa rivin tarkkuudella. Ei ole kysymysmerkki mikä hajosi ja milloin. Työkokemukseni mukaan voi mennä helposti puoli päivää, jos jokin toiminnallisuus ei toimi ja vika ei olekaan uudessa koodissa vaan, että vanha koodi ei sovi täysin yhteen muutosten jälkeen. Tieto, jota vaihdellaan, onkin muuttunut hivenen matkan varrella. Tästä johtuen on tärkeää, että tiedetään kehitettävän ohjelmiston kunto ja saadaan tietoa päätöksentekoa varten. Testit voivat toimia myös spekseinä. Niiden avulla varmistetaan, että koodi toimii tulevaisuudessa ennalta määritellyllä tavalla. Testejä kirjoittaessa olisi hyvä määritellä toiminnallisuuden sisään- ja ulostulot.

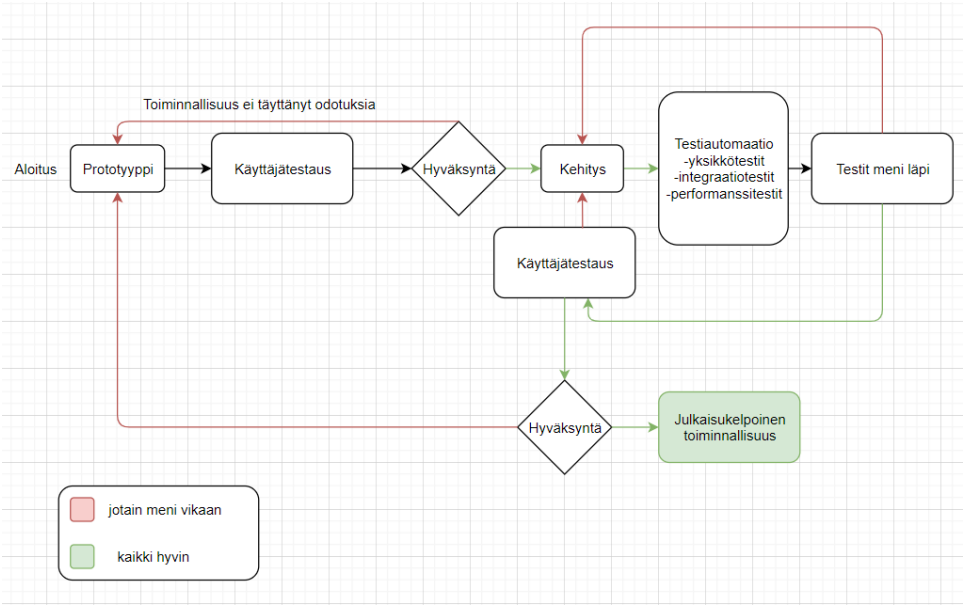


Kuva 3. Kuvassa näkyy, miltä testien sisään- ja ulostulot voivat näyttää. Raja-arvotesteillä käydään testikohde läpi, antaen sille arvoja rajojen alku- ja loppupäästä sekä niiden väliltä. Ajatuksena on, että testataan toiminnallisuudet myös mahdollisissa ongelmatilanteissa. Kuvassa olevassa raja-arvotestimenetelmässä ei usein testata, mitä tapahtuu, jos mennään yli rajojen. Lisäsin poikkeustilanteiden testitapaukset, sillä ongelman hallinnan kannalta on hyvä testata, esimerkiksi, jos kentän lataajalle ei anneta kenttää ollenkaan tai jos kenttä on viallinen. Nämä ovat mahdollisia tilanteita, jos jokin tiedostoista on korruptoitunut tai puuttuu kokonaan. Huomion arvoista on, että vaikka testissä mitataan aikaa ei se silti ole suorituskykyä mittaamaan tehty testi. Suorituskykyä mittaavat testit mittaavat koodin suoritusajan ja tulostaa sen kehittäjän haluamaan paikkaan.

Kun ollaan tekemisissä 3d-järjestelmien kanssa, on tilanne erilainen verrattuna tavallisen ohjelmistoympäristön toimintaan, sillä poiketen perinteisistä ohjelmista 3d-moottorien kanssa ollaan tekemisissä 3d:n piirtämiseen ja fysiikkaan tehdyn kehitysympäristön kanssa. Lisäksi ohjelmointi kieliä voi olla useita samassa projektissa. Tällöin pelkkä koodin toiminnallisuuden toteaminen ei välttämättä riitä, avuksi otetaan ajonaikaiset testit, jolloin moottorin toiminnallisuudet otetaan käyttöön ja testataan, toimiiko ilmiö myös virtuaalimaailmassa.

Virtuaalimaailmassa testit voivat mennä läpi, mutta ohjelma ei toimi halutulla tavalla. Syitä voi olla useita. esimerkiksi päivitetyt rajapinnat, moottori tai asetusten muuttaminen. Miten testataan, huomataanko kappaleen tippuminen ohjelmallisesti, kun fysiikkamoottorin päivitystaajuutta on muutettu? Kappale voi liikkua niin nopeasti, että on hyvinkin mahdollista, että asetukset tai virheet moottorissa jättävät pienen aukon, jolloin pallo voi mennä sen ylittämistä seuraavan toiminnon läpi. Toinen tilanne voi olla, että suunnittelija tai mallintaja on tehnyt sisältöön muutoksia ja ohjelmistotekninen komponentti on syystä tai toisesta jäänyt pois tai asetuksia on muutettu. Tässä tapauksessa perinteisesti

voidaan todeta, että koodi tekee tehtävänsä, mutta ei toimi, sillä kokonaisuus on monimutkainen ja siihen vaikuttaa myös moottorin konfigurointi. Tarvitaan siis tapa testata sitä, että monimutkainen kokonaisuus toimii haluttavalla tavalla.



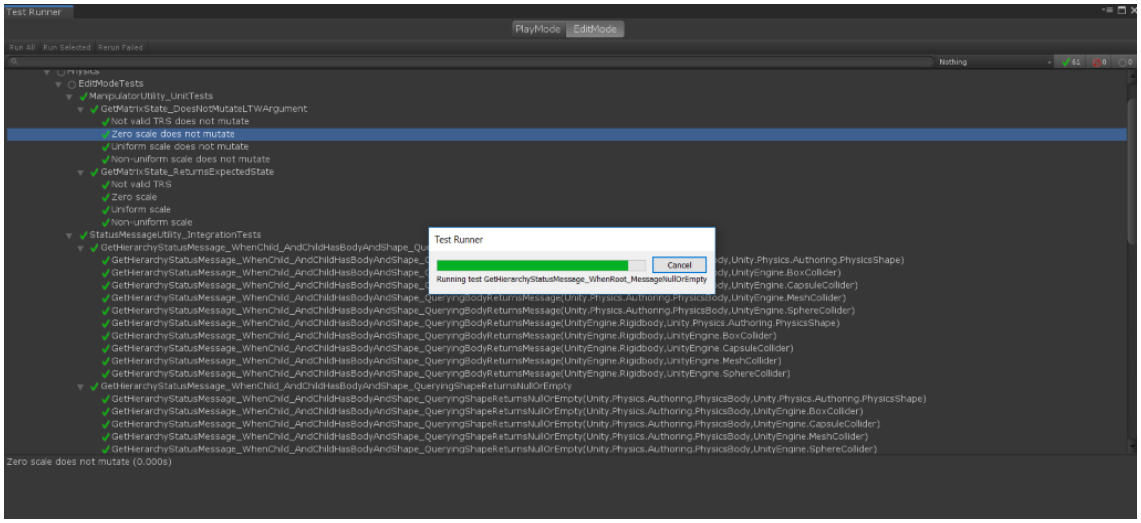
Kuva 4. Kuvasta näkee, kuinka paljon testaamista tarvitaan joissain ympäristöissä ennen kuin jotain voidaan viedä tuotantoon. Koneellinen testaus on tehty tuotannon toiminnoille, jotka ovat usein näkyvissä loppukäyttäjälle tai pelaajalle. Kuva on omaa pohdintaa perustuen työelämän kokemukseen, kuviin netissä, koulutukseen ja pohjamateriaaliin. Isoin kunnia omasta teoreettisesta osaamista menee koululle. Työn käytännönpuoli tulee työelämästä ja siitä, miten itse olen oppinut näkemään asian. Käyttäjä testauksen voisi mahdollisesti ottaa pois, mutta jos ei ole käyttäjiä, joilta saada palautetta, on huomattavasti vaikeampaa saada tehtyä tuotetta, joita uudet käyttäjät haluavat käyttää. Tuotteen tekeminen ilman käyttäjiä, joita vasten testata, on hyvin mahdollista, mutta vaatii luonnollisesti paljon enemmän osaamista alueelta.

Integraatiotestauksen tarve kasvaa, mitä teknisesti vaikeampi peli tai sovellus tarvitaan. Esimerkiksi e-urheilu tai jossain pelissä käytetty työn automatisoitu prosessi, jossa testataan, että automaation välivaiheista tulee odotettu tulos testiaineistolla. On myös pelejä, joissa bugisuudella on tehty taidetta, esimerkiksi Goat Simulator.

2.2 Unity 3D -moottorin testikehysympäristö

Testit ajetaan Unityn editorissa Testinajaja-työkalua käyttäen. Testit voidaan ajaa myös omassa tai Unityn pilvessä. Testien kirjoitus tapahtuu normaalisti ohjelmointiympäristöjä

kuten esimerkiksi Visual studiota tai Rideriä käyttäen. Testit voidaan ajaa kahdessa tilassa: editointitilassa ja ajotilassa. Editointitila on enemmänkin editoripuolen työkalujen toiminnallisuuksien testaamista varten ja pelitila on sitten pelin sisäistä testausta varten. Isoin ero kuitenkin on siinä, että pelitilassa voidaan ajaa ajonaikaisia toimintoja, kuten fysiikkaa ja grafiikkaa. Editointitilassa kannattaa testata asioita, jotka eivät tarvitse pelin käynnistystä.



Kuva 5. Kuvassa Unityn testienajaja ja testejä ajetaan editointitilassa.

Testinajajan avaamalla näkee ja voi suorittaa testit. Tämä paljastaa asioita kuten sen, että testejä löytyy kahdenlaisia. Testejä, jotka vaativat itse sovelluksen suorittamisen ja testejä, jotka voi ajaa editointitilassa, jolloin moottori ei aktiivisesti suorita sovellusta tai peliä.

2.2.1 Test runner ja NUnit

Unity-3D:n testinajajaa käytetään koodin testaamiseen editointi- ja pelitilassa. Se on käyttäjälle pieni ikkuna, jossa näkyvät testien nimet, niiden suoritustila ja nappulat testien ajamista varten. Koodia voidaan testata NUnit-testeillä eri tilanteisiin liittyvillä eri tavoilla. NUnit on yksikkötestikehys kaikille .Net-kielille, se tarjoaa ohjelmointipuolen työkalut testien kehittämistä varten.

NUnit-testien ajaminen Unity-maailmassa tapahtuu Unityn testienajajan kautta. Nunit tarjoaa useita toiminnallisuuksia kuten testileimat, toimenpiteet ja ominaisuudet. Testileimojen avulla pystytään antamaan testeille tietoa testien suoritusta varten ja suodattaa eri alustojen mukaan. Se, kummassa tilassa testejä suoritetaan, on kiinni funktion/testin yläpuolella olevasta leimasta (testitagi) sekä kansiosta, jossa ko. testejä säilytetään.

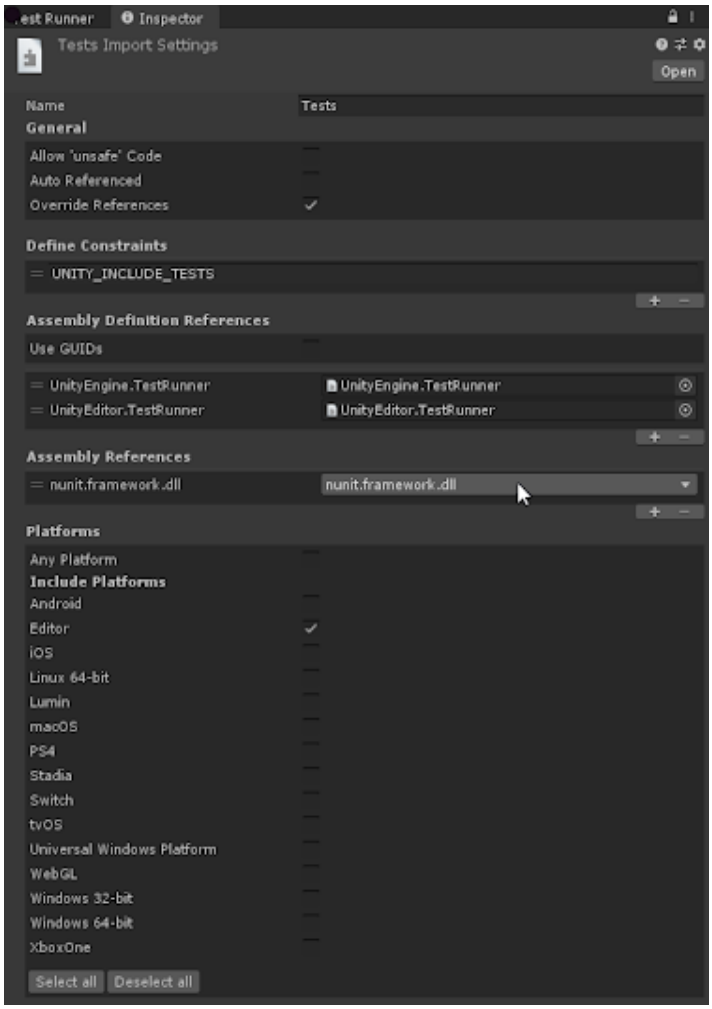
Antamalla koodin kautta testinajajalle ominaisuuksia voidaan vaikka suorittaa toimintoja ennen testejä tai sisällön valmistelua johonkin tiettyyn olosuhteeseen tiettyä operaatiota varten. Unityn testiympäristö ei vielä tue NUnit-kirjaston parametrisoituja testejä, joten parametrisoinnit tapahtuvat oman toteutuksen kautta. Kun Unityn testinajajalle halutaan kertoa, oliko testin tulos oikein, voidaan käyttää NUnit:in “Assert true” -viestiä.
NUnit on hyvin dokumentoitu, joten sieltä kannattaa katsoa enemmän, jos on tarvetta tietää enemmän.

Taulukko 1. Unityn testikehyksen tuettuja ja tukemattomia toimintoja. Sama taulukko löytyy Unityn dokumentaatiosta.

Unity testikehys	**tuettu**
Alusta: Webgl ja WSA	Ei
Muut alustat	Kyllä
NUnit repeat	EI
Parametrisoidut testit	EI
NUnit retry(pelitila)	EI
Parametrisoidut testit (value source)	Kyllä

Unityn testikehys järjestelmä tukee useimpia toimintoja melkein kaikilla alustoilla. Jotkin toiminnallisuudet eivät toimi ja nämä kannattaa ottaa huomioon testejä suunniteltaessa. Toisaalta Unity mahdollistaa testien ajamisen melkein kaikilla alustoilla. Tähän Unity tarjoaa työkaluja. Tekemällä testikokoonpanon käyttäjä voi määritellä alustat, joihin testit kohdistetaan. Myös testien sekaan voi pistää koodirivejä, jotka kohdistavat koodia alustojen mukaan. Joissain tapauksissa jokin toiminto ei ole tuettu muilla alustoilla, joten siihen tarvitaan eri poikkeus. Mielestäni tallaista koodia on vaikeampi seurata ja ymmärtää mitä tehdään. Vielä hankalampi muuttaa, sillä se tarvitsee kyseisen alustan laitteiston, sekä sen tukipaketit tai ympäristön asennettua omalle koneelle. Tällaista koodia tapaa

ostettujen lisäosien lähdekoodissa, sillä lisäosien on yleensä toimittava kaikilla Unityn tukemilla alustoilla, jotta ne myyvät paremmin.



Kuva 6. Kuvassa näkyy testikokoonpanon mahdollisia asetuksia.

2.2.2 Testikehykset

Yleisesti ottaen testikehykset tarjoavat sarjan työkaluja, toimintatapoja ja toimintoja kehittäjille, joiden avulla voidaan toiminnallisuuksia toteuttaa. Kun puhutaan yksikkö- ja integraatiotestauksesta, tarkoitetaan yksikkötestauksella jonkin ilmiön kuten aseen laukaisun testaamista ja integraatiotestauksella sitä, laukeaako se ase halutulla tavalla jossain tietyssä ympäristössä. Nämä ovat vertauskuvallisia esimerkkejä tosielämästä. Testiym-

päristön tehtävä on mahdollistaa nämä edellä mainitut tosielämän esimerkit, jolloin pelimoottori käynnistetään ja aseen laukaus voidaan testien avulla varmistaa toimivan kaikkialla.

Unityn testikehys tekee seuraavat operaatiot, kun testataan pelioliota pelitilassa.

1. Integraatiotestien ajaja luo tyhjän maailman ja tekee tarvittavat toimenpiteet, jotta testit voidaan ajaa.

2. Integraatio- ja yksikkötestit ajetaan määritetyillä alustoilla.

3. Tuotetaan tulos. Testi menee läpi tai epäonnistuu.

4. Tulosten keräys ja tarvittavien kaavioiden ja tiedostojen teko.

5. Julkaistaan tulokset hyvään paikkaan kuten pilveen työntekijöiden saataville.

2.2.3 Yksikkötestikehys

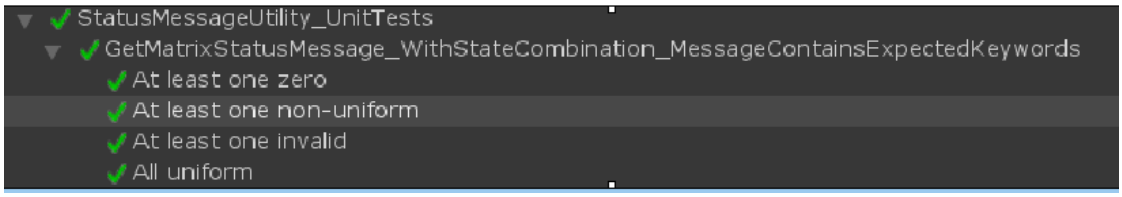
Unity 3D -maailmassa yksikkötestaus keskittyy yksinkertaisten asioiden testaukseen, kuten funktioon tai luokkaan. Unityn nettisivujen mukaan se ei sisällä ulkopuolisia riippuvaisuuksia, eikä ole tekemisissä editorin kanssa tai ota yhteyttä internetin yli, ellei niin määritetä. Unity 3D:n yksikkötestaus tapahtuu pääosin testinajajaa ja Nunit-kirjastoa käyttäen.

Projektin yksikkötestaus keskittyy asioiden kuten vaikka fysiikan ilmiöiden varmistamiseen ja aseiden oikeanlaisen toiminnan toteamiseen. Yksinkertaisina testeinä voidaan käyttää integraatiotestin osia, kuten panos lähtee piipusta tai että panos osuu kohteeseen ja se reagoi jollain tietyllä valmiiksi asetetulla tavalla. Käymme yksikkötestikehyksien ominaisuusia esimerkkien kautta läpi, sillä tällä tavalla päästään hyvin käyttäjätarinoiden pariin ja saadaan myös makua siitä, mihin ja miten testejä voidaan soveltaa.

Unityn GitHub-sivuilta löytyy paljon esimerkkejä hyvistä testitapauksista uusiin teknologioihin liittyen. Käymme nopeasti läpi muutamia koodiesimerkkejä liittyen Unityn eri osa-

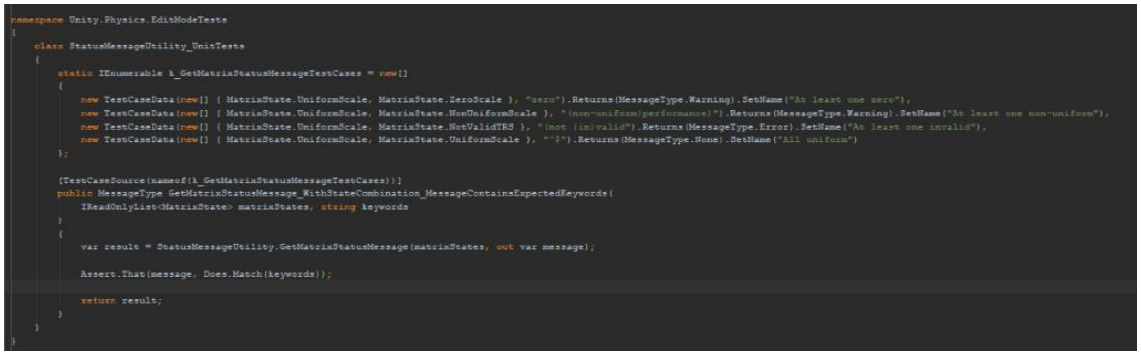
alueisiin kuten korkeansuorituskyvyn fysiikkamoottorin perustoimintojen testaamiseen ja entiteetteihin.

Koodit ja projekti löytyvät GitHubista nimellä Unity Technologies. Otetaanpa yksi koodiesimerkki tarkastelun alle. Testi näyttää seuraavalta itse testinajajassa.



Kuva 7. Testinajajan käyttöliittymä, testien tarkastelua ja ajamista varten. Ikkunan avulla voi myös kätevästi valikoida, mitä testejä ajetaan.

Testi, joka tuottaa kyseisen ikkunan, näyttää seuraavanlaiselta.



```
namespace Unity.Physics.EditModeTests
{
    class StatusMessageUtility_UnitTests
    {
        static IEnumerable k_GetMatrixStatusMessageTestCases = new[]
        {
            new TestCaseData(new[] { MatrixState.UniformScale, MatrixState.ZeroScale }, "zero").Returns(MessageType.Warning).SetName("At least one zero"),
            new TestCaseData(new[] { MatrixState.UniformScale, MatrixState.NonUniformScale }, "(non-uniform|performance)").Returns(MessageType.Warning).SetName("At least one non-uniform"),
            new TestCaseData(new[] { MatrixState.UniformScale, MatrixState.NotValidTRS }, "(not (in)valid").Returns(MessageType.Error).SetName("At least one invalid"),
            new TestCaseData(new[] { MatrixState.UniformScale, MatrixState.UniformScale }, "^$").Returns(MessageType.None).SetName("All uniform")
        };

        [TestCaseSource(nameof(k_GetMatrixStatusMessageTestCases))]
        public MessageType GetMatrixStatusMessage_WithStateCombination_MessageContainsExpectedKeywords(
            IReadOnlyList<MatrixState> matrixStates, string keywords
        )
        {
            var result = StatusMessageUtility.GetMatrixStatusMessage(matrixStates, out var message);

            Assert.That(message, Does.Match(keywords));

            return result;
        }
    }
}
```

Kuva 8. Tarkastelun alla on koodi, jossa testataan matriisien tilanviestitystä.

Kuvassa oleva koodi voidaan rikkoa kahteen osaan.Testidatan luomiseen ja itse testikoodiin. Testidatan luomisessa käytetään Nunit:in TestCaseData-luokkaa. Luokalle annetaan parametrina matriisin tilat sisääntulona ja haluttu testin ulostulo saadaan palautustoiminnallisuuden avulla. Ulostulosta saadaan myös testin nimi, joka voidaan antaa lisäämällä funktio koodirivin loppuun.

```
setName(testinNimi)
```

Esimerkkikoodi 1. Koodi, jolla voidaan antaa testidatalle testin nimet.

Tämän tulos näkyy testinajajan sovellusikkunassa.

Koodin toisessa osassa tämä testidata annetaan testille parametrina “testidata” ja sitä verrataan keskenään parametrina annettuihin avainsanoihin.

Käydään läpi entiteettien jaettujen arvojen testi ja TearDown-ominaisuus.
Seuraavassa esimerkissä käymme nopeasti läpi Unityn entiteetteihin ja entiteettijärjestelmään liittyvien muuttujien testaamisen.

```
NativeArray<int> Source;
NativeArraySharedValues<int> SharedValues;

[TearDown]
public void Cleanup()
{
    if (Source.IsCreated)
    {
        SharedValues.Dispose();
        Source.Dispose();
    }
}
```

Esimerkkikoodi 2. Koodista näkee TearDown-ominaisuuden käyttöä. Ominaisuutta ja sen jälkeistä koodia käytetään muuttujien siivoamiseen muistista testin jälkeen.

TearDown-ominaisuus tarkoittaa, että jokin suoritetaan testin jälkeen ja natiivien taulukkojen muistista poistaminen on hyvä kohde sille, koska muuten nämä jäävät järjestelmään. Automaattinen roskienkeruu ei näitä huomioi, joten mikäli halutaan välttyä muistivuodoilta, nämä olisi hyvä siivota käyttämällä Dispose-komentoa.

```
[Test]
public void NativeArraySharedValuesAllEmptyArray()
{
    PrepareAllSame(0);

    Assert.AreEqual(0, SharedValues.SharedValueCount);
    Assert.AreEqual(0, SharedValues.GetSortedIndices().Length);
}
```

Esimerkkikoodi 3. Koodinpätkässä esitellään yksinkertaisen testin toiminta. [Test] ominaisuutta käytetään kertomaan testikehykselle, että kyseessä on suoritettava testi. Tämän jälkeen testi alkaa ja suoritetaan. Testin alussa usein alustetaan testiympäristö, mikäli testi sitä tarvitsee.

Esimerkkikoodin testi on hyvin yksinkertainen. PrepareAllSame-komento tekee tyhjän jaetun natiivitaulukon ja testissä testataan, että koodi, joka tämän luo, tekee tämän. AreEqual-komento vertaa, että tyhjäntaulukon alkioiden lukumäärä on tasavertainen nollan kanssa.

```
Assert.AreEqual (0, SharedValues.SharedValueCount);
```

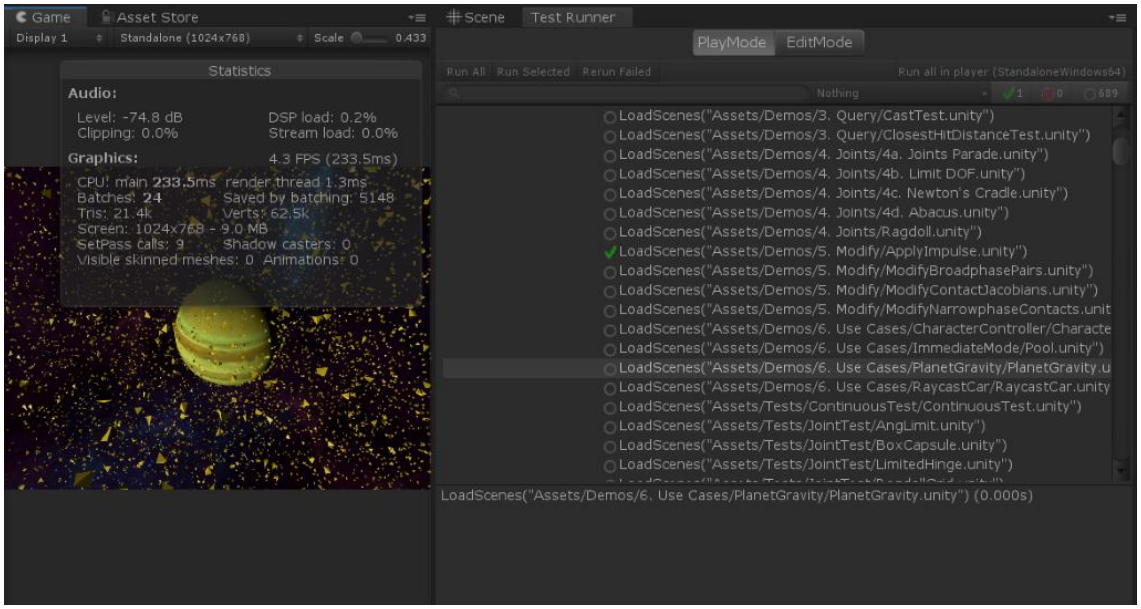
Esimerkkikoodi 4. Tämä koodirivi vertaa jaettua taulukon kokomäärää nollaan ja palauttaa testille joko totuuden tai epätotuuden perustuen siihen, onko taulukon sisältö todella tyhjä.

2.2.4 Integraatiotestikehys

Joskus tai usein ei yksikkötestaus riitä koska rajapintojen ja komponenttien määrä sovelluksessa on kasvanut. Integraatiotestauksessa testataan kokonaisuuksien toimintaa ja yhteensopivuuksia. Integraatiotestit ajetaan pääosin editorissa ajotilassa, eivätkä nämä vaikuta testien ulkopuolelle, sillä Unity-ympäristö osaa karsia testaamiseen liittyvät ohjelmointitiedostot pois peliolioista.

Integraatiotestaus toteutetaan muusta pelisovelluksesta eristetyllä kentällä. Integraatiotestejä, joita tulee mieleen, ovat aseiden toimivuus, kun ammutaan vihollisia tai että pelaajan kohdistuvat voimat tuottavat halutun tuloksen. Integraatiotestit ovat mielestäni vähän mielenkiintoisempia, sillä ne sisältävät useimmiten itse ajotilan käytön.

Ensimmäisessä esimerkissä esitämme “LoadScenes"-testiä. Se ajaa jokaisen fysiikkatestin sisältävän maailman, joten sen avulla saa nopean katselmuksen siitä, mitä Unityn uusi fysiikkamoottori voi tehdä. Esimerkin löytää Unityn GitHubista.



Kuva 9. Testinajajan käyttöliittymä + näkymä testien tarkastelua ja ajamista varten. Kuvassa näkyy, että yksi testi on suoritettu ja toinen on suorituksessa. Testiä voi myös seurata, sillä pelimoottori osaa piirtää kyseisen tapahtuman.

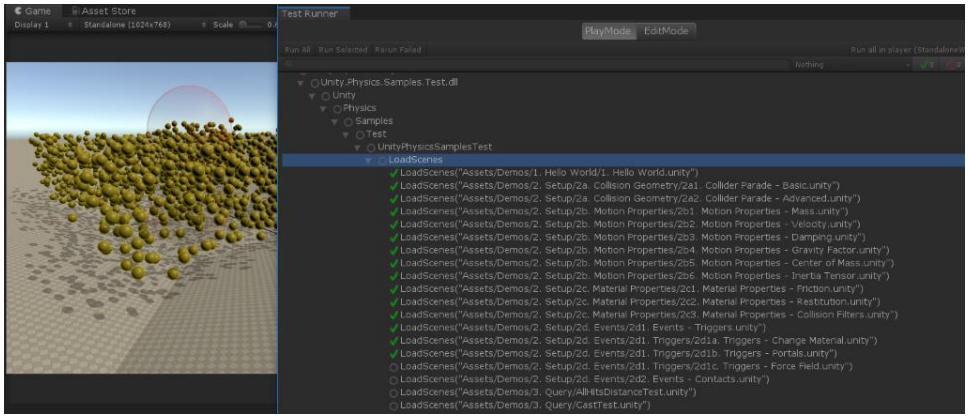
Aikaisemmin puhuttiin, että testit voivat toimia myös spekseinä. Jos halutaan että testit toimivat myös spekseinä, tulisi toimia testit ensin -periaatteella ja miettiä tarkasti, mitä koodilta tarvitaan ulos testeille. Testinajaja listaa hienosti testit helposti luettavaksi kokonaisuudeksi, josta näkee fysiikkamoottorien tärkeiden osien testit. Tätä kannattaakin käyttää eräänlaisena tutustumisena fysiikkamoottorin toimintaan, mikäli on tästä kiinnostunut tai on tarve tietää enemmän.

Seuraavaksi päästään katsomaan koodia kyseisestä testikokoelmasta.

```
#if HAVOK_PHYSICS_EXISTS

[TestFixture]
public class HavokPhysicsSamplesTest : UnityPhysicsSamplesTest
{
    void SetSimulationType(SimulationType simulationType, Scene scene, LoadSceneMode mode)
    {
        var entityManager = World.Active.EntityManager;
        var entities = entityManager.GetAllEntities();
        foreach (var entity in entities)
        {
            if (entityManager.HasComponent<PhysicsStep>(entity))
            {
                PhysicsStep componentData = entityManager.GetComponentData<PhysicsStep>(entity);
                componentData.SimulationType = simulationType;
                entityManager.SetComponentData<PhysicsStep>(entity, componentData);
                break;
            }
        }
    }

    [UnityTest]
    [Timeout(60000)]
    public override IEnumerator LoadScenes
    ([ValueSource(nameof(UnityPhysicsSamplesTest.GetScenes))] string scenePath)
    {
        // Log scene name in case Unity crashes and test results aren't written out.
        Debug.Log("Loading " + scenePath);
        LogAssert.Expect(LogType.Log, "Loading " + scenePath);

        SceneManager.sceneLoaded += (Scene scene, LoadSceneMode mode)
=>SetSimulationType(SimulationType.HavokPhysics, scene, mode);
        SceneManager.LoadScene(scenePath);
        yield return new WaitForSeconds(1);
        UnityPhysicsSamplesTest.EntitiesCleanup();
        yield return new WaitForFixedUpdate();
        LogAssert.NoUnexpectedReceived();
    }
}

#endif
```

Esimerkkikoodi 5. Kuvassa kahdeksan takana oleva koodi.

Esimerkkikoodissa 5 nähdään testiluokka ja itse testi. [TextFixture] ominaisuudella määritellään testiluokka. Annettu luokannimi näkyy myös testienajajassa. Myös testiprojektissa käytetään saman suuntaista taktiikkaa. Sen sijaan, että kirjoitetaan jokaiselle aseelle tai kentälle oma testi, tehdään yksi testi, joka käy kaikki testattavat läpi. Unityn esimerkissä on käytetty hienosti ValueSource-ominaisuutta ja täten saatu parametrisoidut testit mukaan. Testit luodaan ja käydään läpi automaattisesti. Toisaalta testin koodia on huomattavasti vaikeampi tulkita, ja testin kokoaminen on tarvinnut enemmän miettimistä.



Kuva 10. Kuvasta näkee, kuinka testiautomaation suorittaa testejä. Vihreät onnistuneita ja tyhjät pallukat suorittamattomia testejä.

Mikäli tallaisen testin luominen on tuotannossa hyvin dokumentoitu tai ohjelmoija on saatavilla, kun häntä tarvitaan, niin säästetään jonkin verran resursseja tuotannolta.

```
[UnityTest]
[TimeOut(60000)]
```

Esimerkkikoodi 6. Ominaisuuksilla kuvaillaan testikehyksille mitä testiltä tulisi odottaa. Tässä tapauksessa ilmoitetaan testin olevan Unitylle tehty ja että testin suoritus lopetetaan 60 tuhannen millisekunnin jälkeen, jolloin testi epäonnistuu.

Ominaisuuksilla voidaan kertoa, että testi suoritetaan ajotilassa. Jos testi ei suoriudu 60 sekunnissa, suoritus katkaistaan. Tällöin testi merkitään epäonnistuneeksi. Tätä on hyvä käyttää tilanteissa, joissa on mahdollista, että testin odotetut arvot eivät täyty ja testi jää näitä odottamaan. Näissä tapauksissa testi jää ikuisesti suoritettavaksi.

```
[ValueSource(nameof(UnityPhysicsSamplesTest.GetScenes))]
```

Esimerkkikoodi 7. ValueSource-ominaisuuden avulla saadaan tuki parametrisoiduille testeille.

Yllä parametrina on annettu funktio palauttaa kaikki rakennusasetuksissa määritellyt tasot/pelikentät merkkijonolistana testille. ValueSource-atribuutin avulla voidaan antaa parametreja testille. Testin onnistumisen määrittelyyn testi käyttää koodia:

```
LogAssert.Expect(LogType.Log, "Loading " + scenePath);
```

Esimerkkikoodi 8. Testissä käytetty tapa, jolla voidaan antaa testinajajalle käsky odottaa tekstiä.

Funktiolla annetaan oletus, että tämä viesti tulee kirjanpitoon esille jossain välissä. Mikäli näin ei tapahdu, niin testi epäonnistuu. Tämä on minulle uusi tapa toteuttaa testiautomaatio, sillä olen tottunut tuttuun. Testi antaa komennon rajapinnalle ja sen antamaa tulosta verrataan odotettuun arvoon, mikä on mielestäni se yksinkertaisin tapa tehdä testejä. Integraatiotesteissä on myös se puoli, että jos rajapintoja on useita, voi olla, että tarvitaan odottelua. Odottelun voi toteuttaa palauttamalla tyhjää, tämän jälkeen seuraava suoritus tapahtuu, kun nykyinen kuva on piirretty loppuun. Odottelu ei tuki sovelluksen toimintaa, sillä jos tätä silmukassa toteutetaan niin ohjelmalla on useasti enemmän aikaa piirtää kuvia kuin suorittaa koodia. Huomionarvoista odottavissa testeissä on, että testin paluu arvona on IEnumerator-rajapinta, joka tarvitaan odottelua ja taustalla ajoa varten. Tästä käyttäjän ei tarvitse huolehtia, vaan tämä tapahtuu pinnan alla ja testienajaja on se, joka testit käynnistävät.

2.3 Yleistä tietoa Unityllä testaamisesta

Tuotannossa voi ajan vuoksi olla hankalaa testata jokainen funktio. Myös Unityn oma komponenttiarkkitehtuuri voi tehdä testien kirjoittamisesta erilaisempaa kuin muissa ympäristöissä. Ajan myötä voidaan huomata useita asioita kuten sen, että kaikkea ei voi tai edes kannata testata, joten testit kannattaa rajata johonkin. Testien kirjoittaminen voi aiheuttaa huomattavan määrän lisätyötä, mikäli testit vaativat omaa toiminnallisuutta testiympäristön luomiseen tai jos testejä lisätään jo toimivaan toiminnallisuuteen ja se aiheuttaa uusia muutoksia.

Testit siis kannattaa aluksi rajata, vaikka projektin sisäisiin julkisiin rajapintoihin tai johonkin yleisesti käytettyihin toimintoihin, johon koko sovellus nojaa. Oman työelämän kokemukseni mukaan voin neuvoa määrittelemään sovelluksen/pelin toiminnallisuudet niiden kriittisyyden mukaan ja arvioimaan, kuinka paljon jonkin osan hajoaminen voi vaikuttaa sovelluksen kehityskustannuksiin tai menetettyihin tuloihin. Tämä on helpommin sanottu kuin tehty, sillä hyvin iso osa tästä tiedosta tulee kokemuksen kautta. Tässä tilanteessa on hyvä käyttää käyttäjä- ja pelitarinoita, joiden avulla voidaan kuvitella pelaajalle tai käyttäjälle aiheutunut harmi ja jatkoseuraukset.

Kun ohjelmoidaan rajapintoja vasten, voidaan testit kirjoittaa myös rajapintoja eikä rajapintojen sisäisiä toiminnallisuuksia vastaan. Testien kirjoittamiseen käytettyä aikaa voidaan minimoida myös käyttämällä kirjastoja, joilla on jo kattavat testit. Testejä katselemalla saa selville hyvin, miten koodia tulisi käyttää ja missä sen heikot kohdat ovat. Kuten aikaisemmin mainittiin, testejä voidaan kirjoittaa sitä varten, että huomataan, mikäli jokin kriittinen osa ohjelmasta hajoaa. Tällöin viallista koodiriviä ei tarvitse etsiä, sillä järjestelmä osaa näyttää sen etukäteen. Omassa tapauksessa on aina pieni pelko käytössä olevan teknologian tai toiminnallisuuden heikkoudesta, minkä mahdollisesta hajoamisesta olisi hyvä olla tietoinen. Pahimmassa tapauksessa on pelko myös siitä, että työkaverin muutos muualla hajottaa oman vastuualueen toiminnon.

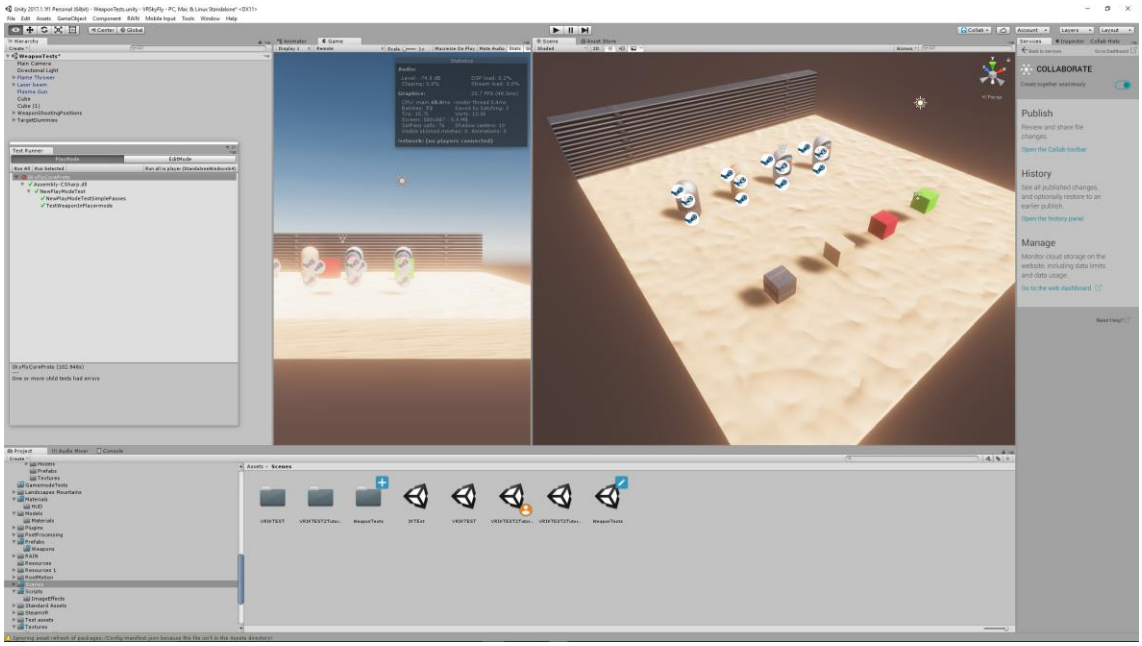
Sovelluksen/pelin testien olemassaolo voi perustua tietoon siitä, että ohjelmiston kerrosarkkitehtuurin alapuolella tapahtuu jatkuvia isoja muutoksia, kun tekniikkaa viedään eteenpäin. On siis mahdollista kirjoittaa testejä joitain todella isoja kokonaisuuksia varten. Esimerkiksi meneekö testi läpi, jos nosturille annetaan hyvin pitkä sarja komentoja tai pysyykö sovellus käynnissä ja hyvässä kunnossa, jos sitä pitää 10 päivää auki? Tällöin testin odotettuja tuloksia voidaan mitata vaikka tarkkailemalla, mitä ympäristössä tapahtuu. Sen sijaan, että mitataan ohjelman jokaista muuttujaa, katsotaankin vain yksinkertaisesti sitä, mitä käyttöjärjestelmä kertoo sovelluksen muistinkäytöstä. Tällöin ei saada selville, mikä aiheutti vian. Sen sijaan saadaan tieto siitä, että jokin monimutkainen käyttäjälle näkyvä kokonaisuus ei toimi. Testejä voidaan myös käyttää ei-ohjelmallisten toimintojen eheyden varmistamiseen. Esimerkiksi kun sovelluksen tai pelien parametreja on muutettu, niin kulkeeko tämä laite tai otus vielä sen reitin, minkä sen on tarkoitus kulkea.

3 Testien ajo ja testistrategia

Lopputyön projektin testit ajetaan pelitilassa. Strategiana on, että lähdetään liikkeelle ajatuksesta, että on paljon pelinosia, jotka vuorovaikuttavat keskenään, joko suorasti tai epäsuorasti. Suora vaikuttaminen siis on tässä yhteydessä sitä, että kaksi testattavaa vuorovaikuttavat suoraan toisten kanssa, ja epäsuora siitä, että on jokin kolmas kappale välissä tai jotain on jo ehtinyt tapahtua. Pelissä on iso kasa pelaajan aseita ja työkaluja. Jokaisen testaaminen aina isojen muutoksien jälkeen on huomattavan iso tehtävä, ja osa

testeistä voi jäädä tekemättä, sillä pelin ominaisuuksien testaaminen ei ole ihan niin systemaattista pelin interaktiivisen luonteen vuoksi.

Tästä syystä projektissa tehtiin ampumaradan tyylinen testiympäristö, jossa testataan kaikki aseet ohjelmallisesti peräkkäin valittuja elementtejä ja olioita vastaan.



Kuva 11. Kuva testikentästä, maalitauluista ja laatikot kuvaavat pisteitä, joista ase laukaistaan. Kuvassa näkyy myös testinajaja ja kaksi testiä joista toinen ajaa projektin testin. Toinen on Unityn esimerkki testi. Eli jokaisen kuution kohdalla on piste, josta ase laukaistaan kapselin muotoista fysiikkaobjektia kohti. Kun projektiili osuu kapseliin, se heilahtaa projektiilin voimasta.

Projektin testikohteina on kohteita, jotka on merkattu eri pelielementeiksi. Näihin elementteihin kuuluu Maa, kivi, metalli, esineet, vihollinen ja pelaaja. Aseiden toimivuus testattiin yksikkötesteillä, eli lähteekö projektiili piipusta. Aseiden vaikutus muihin elementteihin suorasti ja epäsuorasti testattiin integraatiotesteillä.

Esimerkkikoodissa 10, on esimerkki ohjelmakoodista, jolla voi käydä läpi testiprojektin testattavat aseet. Lyhyesti koodi luo ympäristön ja hakee sieltä kaikki aseet. Sen jälkeen koodi sijoittelee aseet paikoilleen vuorotellen ja aktivoi aseeseen liitetyn asejärjestelmän.

```
SetupScene ();

//After setting up scene wait for program to draw 1 frame.
yield return null;

//Fetch all the necessary objects and locations
List<GameObject> weapons = GameObject.FindGameObjectsWithTag("Weapon").ToList
();
List <Transform> weaponsShootingPositions = GameObject.Find("WeaponShooting-
Positions").transform.GetComponentsInChildren<Transform>().ToList ();

//This chunk of code goes through every weapon and location. Test firing at
each position
foreach (GameObject weapon in weapons) {
     WeaponSystem weaponSystem = (WeaponSystem)weapon.transform.GetComponent
("WeaponSystem");
     foreach (Transform shootinPosition in weaponsShootingPositions) {
          weapon.transform.position = shootinPosition.position;
          if (weaponSystem != null) {
               weaponSystem.ActivateWeapon ();
               //make 300 frame delay after weapon is fired
               for(int i = 0; i < 300; i++){
                    yield return null;
               }
               weaponSystem.DeactivateWeapon ();
          }
     }
}
```

Esimerkkikoodi 9. Projektin työssä aseiden testauksen suorittava testikoodi. Testiautomaation koodi on kirjoitettu C# ohjelmointikielellä.

Kun esimerkki 10 koodin testisilmukkaa toistetaan tarpeeksi, niin saadaan osa pelin pelimekaanista toiminnallisuutta testattua pienessä ajassa. Testausta voitaisiin vielä tehostaa tekemällä algoritmi sitä varten, että jokainen kombinaatio saadaan suoritettua. Esimerkiksi jos on peliobjekti, joka muuttaa niiden elementtien ominaisuuksia, joihin se osuu ja tämän jälkeen vaihdetaan asetta, joka myös muuttaa arvoja. Muuttuuko jää tulesta vedeksi ja vesi jääksi edelleen muutoksien jälkeen? Mitä jos tulee pelimekaniikkaan päivitys, jossa muutetaan muuttujia reilummiksi aloittelijoille ja siinä on lämpötiloja muutettu pakosta aika voimakkaalla otteella? Näillä testeillä voidaan myös tällainen pelimekaaninen oikku poimia eikä tule yllätyksenä vastaan. Käyttötapauksia löytyy muitakin.

Toimintoja kehittäessä testit edellä -käytännöllä voidaan testiautomaation avulla nopeuttaa prosessia ja tehdä siitä mielenkiintoisempaa. Kolikon kääntöpuolena on testiautomaation päivittäminen. Aikaa kuitenkin säästyy huomattavasti pidemmällä aikavälillä, ja koodin laatu paranee, koska on mietittävä yhteensopivuuksia, ohjelmallista testausta ja reunaehtoja tarkemmin. Samalla voidaan myös eri algoritmien reunaehdot testata tehokkaasti. Tämä kannattaa tehdä sen jälkeen, kun on prototyyppi hyväksytty ja lyöty lukkoon. Prototyyppien on tarkoitus valmistua nopeasti.

Testien kirjoittaminen voidaan perustella tulevaisuuden työn välttelyllä, luotettavuudella ja tuloksellisuudella. Erilaisia sudenkuoppia on tietämyksen puute ja siitä aiheutuvat komplikaatiot. Esimerkiksi ei tunneta testiautomaation syvällistä tarkoitusta ja hyötyjä, joten näitä ei ikinä saada tuotantoon ja testiautomaation ylläpidosta tulee aikasyöppö. Testit eivät välttämättä maksimoi hyötyjä. Jos testejä tehdään pelkästään, koska niin on käsketty. Tämän takia on tärkeää, että on kuva siitä mitä arvoa testien automatisoinnista on liiketoimintamielessä. On paljon pelejä, jotka on jollain tasolla menestyneitä, mutta eivät täytä laadukkaan ohjelmiston merkkejä. On myös selkeätä, että mitään ohjelmistoteknisesti uutta peliä tai sovellusta ei voida tehdä ilman moderneja teknologioita tai isoa tukea.

Unityllä on itsellään käytössä yksikkö- ja integraatiotestit, ja mitä olen nähnyt, niin jokaisessa ryhmässä tai organisaatiossa on ihmisiä, jotka jossain vaiheessa alkavat pitämään Unityä rikkinäisenä ja käyttökokemukseltaan alhaisena kokemuksena. Testeillä ei siis välttämättä kyetä aluksi täysin varmistamaan tuotteen jatkuvaa toimivuutta, mutta ohjelmiston tilaa pystytään analysoimaan. Kun annetaan uusi päivitys, jossa on kriittisiä korjauksia ja ohjelmiston osia on hajalla, niin voidaan julkaisun muistiinpanoihin ja ilmoitukseen lisätä, mitkä asiat on korjattu ja mitkä ominaisuudet ovat hajalla, jotta seuraavaan versioon siirtyvät voivat arvioida, tuhlaavatko he aikaa ja resursseja, kun he päivityksensä lataavat. Seuraavana kysymyksenä voidaankin pitää sitä, mikä on se prosentuaalinen ohjelmiston eheys, jolloin voidaan tuotteen päivitys julkaista. Tässä asiassa kannattaa tarkkaan puntaroida tunnettujen asiakkaiden ja käyttäjien etuja. Mikäli päivityksestä on enemmän hyötyjä kuin haittoja, voidaan päivitys epävirallisena vaihtoehtoisena päivityksenä tarjota käyttäjille, jotka aktiivisesti seuraavat tuotteen tilaa oman kilpailukyvyn ylläpitämisen vuoksi. Tai jos päivitykseen tekemä korjaus tuo heille merkittävää hyötyä ja päivityksen haittapuolet eivät ole käytössä. Mainittakoon vielä, että testit voidaan

myös ajaa useille eri alustoille, kuten Androidille, IOS:lle, webille, PC:lle, Macille ja Linuxille.

Testien suoritusnopeuden voi muuttaa isommaksi ilman, että testien tulokset muuttuvat, mikäli testattavat asiat tapahtuvat ns. fiksatussa ajassa. Tällöin suoritetaan toimintoja annettu määrä sekunnissa ja tätä voi myös muuttaa ajanhallinnasta. Se on myös suotavaa ajankäytön kannalta, kun on kyllästynyt testien seuraamiseen, eikä ole nähtävää muutenkaan.

3.1 Testistragedian valinnan vaikutukset koodin pohjasuunnitteluun

Jotta testit voitiin ajaa ja testiautomaation ylläpitämisen ajalliset resurssit saatiin optimoitua, niin aseiden pohjamekaniikka oli mietittävä uudelleen. Lopputuloksena saatiin projektissa asemekanismi, joka toimii joka aseessa, vaikka aseen pohjatoteutus olisi erilainen. Mekanismi on koodattu rajapintoja vasten, mikä tarkoittaa, että sama komento toimii kaikissa aseissa jotka, ko. rajapinnan toteuttavat. Tämän toteutuksen vuoksi testiautomaatiota ei tarvitse päivittää jokaisen uuden asetyypin tullessa peliin ja pienimmillään riittää, että se toteuttaa tarvittavat rajapinnat.

3.2 Testien ajo editorissa

Testien ajo tapahtuu editorissa testinajajan ikkunan kautta. Testejä voidaan ajaa yksittäin tai kaikki kerralla -menetelmällä. Tämä on hyvä tapa tutustua projektiin ja katso miten toiminnallisuuksia käytetään. Kun jakaa projektia netissä kaikki pääsevät näkemään koodin toimintaa sovelluksen/pelin ulkopuolelta. Koodin ajaminen editorissa on kuitenkin manuaalista ja epäkäytännöllistä. Etuja tästä kuitenkin löytyy. Samalla tavalla kuin muutenkin, voidaan editori ikkunan kautta klikkailla maailmassa olevia malleja ja katsella, mitä niiden sisällä tapahtuu, ilman keskeyttämättä sovelluksen toimintaa tai vaihtamatta ikkunaa. Joissain ympäristöissä testien ajaminen on mahdollista ainoastaan päättömänä, jolloin se mitä testin sisällä tapahtuu, on täysin mustalaatikko kehittäjälle. Kun testejä ajetaan päättömänä tarkoittaa se sitä, että itse sovelluksen moottori, joka pyörittää sovellusta ei ole kytketty ikkunaan. Editori-ikkunan kautta testien ajamisella vältetään kaikenlaisten viritelmien teko ja testit voidaan rakentaa testikoodin varaan. Vaihtoehtona

on rakentaa kaikesta testattavasta toiminnallisuudesta päätön versio ja kenttä, jossa voi korjata ja huoltaa testattavia toiminnallisuuksia. Jos testi ei toimi, suoritetaan koodi ja katsotaan aktiivisesta sovelluksesta mitä siellä menee vialle. Tätä olen jonkin verran tehnyt työelämässä Babylon.js-moottorin ja ava-testikehyksen avulla.

3.3 Testien ajo komentoriviltä

Työssä ei komentoriviä käytetty, mutta sen pikainen läpikäynti on kuitenkin tarpeen. Unity pystytään käynnistämään tilassa, jossa testejä ajetaan komentoriviltä seuraavilla parametreilla:

- -runTest (Ajaa projektin testit)
- -testPlatform (Alusta jolla testit halutaan ajaa. Eli osx, win, linux jne.)
- -testResults (Minne testien tulokset tallennetaan?)

Esimerkit (Otettu unityn dokumentaatiosta)

```
>Unity.exe -runTests -projectPath PATH_TO_YOUR_PROJECT -testResults
C:\temp\results.xml -testPlatform editmode
```

Esimerkkikoodi 10. Komento, jolla voidaan käynnistää Unity testinajomoodissa ja julkaista tulokset .xml tiedostoon.

3.4 Pilvessä

Versionhallinta päivittää uusimmat muutokset ja antaa käskyn ajaa testit komentoriviltä. Pilvessä ajaminen voidaan toteuttaa muutamalla tavalla. Ensimmäinen ja huomattavasti helpoin tapa on käyttää Unityn valmiita palveluita. Toinen tapa on hivenen vaikeampi. Tässä rakennetaan oma ympäristö käyttäen versionhallintaohjelmistoa, kuten Gitiä, Plastic SCM:ää tai Perforceä. Tämä voidaan yhdistää johonkin palveluun, kuten Microsoftin Azureen, joka tarjoaa erilaisen sarjan työkaluja koodinlaadun varmistamiseksi.

3.5 Havainnot ja kritiikki

Testienajo editorin kautta riittää hyvin jo koodaamisen apuna. Pilvessä ajon ominaisuutta voidaan käyttää pelisovelluksen kääntöautomaation tukena. Testien kirjoittaminen itsessään tarvitsee jo kokemusta. Intermodel-nimisessä EU-projektissa kokeilin testit edellä -tapaa. Ensimmäiset testit tuottivat tulosta, mutta kun testit olivat valmiit, niin tuli tämän jälkeen isoja muutoksia testattavien koodien toiminnallisuuteen. Tästä syystä integraatiotestit olisi tarvinnut koodata uudelleen ja niiden toimivuus varmistaa. Testeistä oli kuitenkin apua, ja lopputulos oli huomattavasti parempi. Kyseenomainen toiminnallisuus ei mennyt kertaakaan rikki koko loppuprojektin aikana. Ainut mitä olisi voinut tehdä paremmin, olisi ollut suunnitella rajapinnat tarkemmin ja ottaa arkkitehtuuriin tulevat muutokset huomioon. Testien ajossa on samat ongelmat kuin muissakin ympäristöissä. Miten ihmisen sormien ja hiiren painalluksia sekä käyttäytymistä voidaan mallintaa tehokkaasti ja luotettavasti?

Nykyään on mahdollisuus pilvipalveluihin, joissa voidaan testata koneoppimisen avulla käyttäjän painalluksia sovelluksissa. Tällöin ei testata sovelluksen sisäisiä toimintoja vaan sitä, miten sovellus reagoi, mikäli käyttäjä navigoi johonkin tiettyyn pisteeseen sovelluksessa. Nämä toimivat hyvin samaan tapaan kuin Robot Framework nimellä tunnettu testikehysympäristö, mutta tehokkaampia, sillä niitä tehostaa koneoppiminen. Koneoppimisen avulla voidaan tehdä asioita, jotka ovat ihmiselle helppoja, mutta vaikeita koneelle. On esimerkiksi vaikea simuloida ihmistä, joka on tottunut käyttämään nykyisiä järjestelmiä ja ihmisiä, jotka eivät ole koskaan käyttänyt näitä, antamalla koneelle samat rajoittuneet kyvyt kuin näillä ihmisillä ilman taitoa käyttää niitä uudessa ympäristössä. Näin ollen voitaisiin sanoa että, Unityssä on hyvä perustestikehysympäristö, mutta ei kuitenkaan kaikista kehittyneintä, sillä se on rajoittunut yksikkö- ja integraatiotesteihin. Tätä puutetta on paikattu koneoppimiseen perustuvaa GameTune-toiminnallisuutta, jossa koitetaan sovelluksen tai pelin käytön aikana löytää hyvät asetukset käyttäjälle.

4 Testiprojekti ja havainnot

Testiprojektia varten tehtiin peliprojektiin oma kenttä testailua varten. Testikoneisto ja sitä varten tehty koodi alustaa ensimmäisenä ympäristön ja ajaa testit tämän jälkeen.

Testien suoritusta voidaan seurata editorista, mikäli testaamiseen liittyy interaktiivisuutta kuten fysiikkamoottorin käyttöä. Testiprojektin testistrategiaa suunniteltaessa kiinnitettiin huomiota siihen, miten voidaan toiminnallisuuden, ohjelmoinnin sekä testien puolella saavuttaa helppo ja vaivaton tapa testata tehtyjä toiminnallisuuksia. Esimerkiksi kaikissa aseissa on sama laukaisumekaniikka. Kun aseella ampuminen tapahtuu, kytketään siinä oleva komponentti päälle. Kun komponentti aktivoituu, se alustaa muuttujat ampumista varten. Tämän jälkeen komponentin koodi suoritetaan. Kun aseen ampuminen on ohi, se kytketään pois päältä. Komponentin koodi pois kytkemisen yhteydessä ajaa komennon, joka hoitaa tarvittavat lopputoimenpiteet. Nämä voidaan tehdä jokaiselle aseelle erikseen ilman, että nämä muutokset vaikuttavat testiympäristön toimintaan.

Koko aseen testaaminen tapahtuu siis sillain, että kyseinen komponentti pistetään vaikka 2 sekunniksi päälle ja tarkastetaan, mitä tapahtuu. Tämän jälkeen kytketään pois päältä ja siirrytään seuraavaan aseeseen.

Projektin testeissä ei ollut tarvetta testata yksityiskohtaisemmin aseiden toiminnallisuutta esimerkiksi antamalla aseille satunnaisia arvoja. Testit saa muutettua nopeammiksi, kunhan testit on toteutettu valmiiksi määritellyssä ajassa. Testejä siis pystytään ajamaan vaikka 100-kertaisella nopeudella.

4.1 Tuki, helppokäyttöisyys ja testikohteet

Tukea saa netistä useissa eri muodoissa, myös maksullinen tuki on mahdollista. Internetistä löytyy apua riittävästi käyttämällä työn tekstistä löytyviä termejä. Dokumentaation lisäksi YouTubesta löytyi videoita, joista oppi paljon uutta. Työelämässä Unityn testiautomaatiota on tullut käytettyä EU:n Intermodel-projektissa, jossa testiautomaation avulla varmistettiin nosturin toiminta. Tätä testisarjaa ajatettiin isojen muutoksien jälkeen, jotta saatiin selville, mikäli muutokset vaikuttavat nosturin käyttäytymiseen. Kun tuotteen toimintakyky ja luotettavuus kasvavat voidaan projektista tehdä kunnianhimoisempi ilman pelkoja, että projekti sortuu kunnianhimoisten haaveiden seurauksena. Kun projektin ajajat voivat nähdä koodaajien työn eheyden ja kokonaisuuden käyttöliittymän kautta, voivat myös työkaverit nukkua yönsä paremmin.

4.2 Ongelmat, riskit ja rajoitteet

Yleisimpiä ongelmia on testiautomaation rakentamiseen ja ylläpitoon kuluva aika. Se maksaa itsensä takaisin, mutta siltikään se ei joka projektissa kannata. Testiautomaatiota on vaikeata suositella, vaikka pelitalolle, joka on onnistuneesti tehnyt pelejä vuosikymmeniä ilman ja eikä sillä ole halua alkaa tällaista rakentamaan. Tässä tilanteessa pitäisikin ehkä tutkia, miten kyseistä organisaatiota voitaisiin parantaa. Jokaisessa projektissa on omat ongelmansa. Näiden löytäminen ja automatisointi voi parantaa kaikkien elämää.

Projektissa ongelmia tuli testien kirjoittamisen yhteydessä. Editoritestejä siis testejä joka ei käynnistä itse sovellusta ei voi ajaa fysiikoiden kanssa. Myös joitain muita ominaisuuksia ei voi editointitilassa käyttää, kuten ajastettuja testejä. Ajastimen voi tosin kiertää.

```
yield return null;
```

Esimerkkikoodi 11. Käyttämällä ko. koodiriviä voidaan Unityn testinajaja pakottaa odottamaan yhden kuvan piirtämisen ajan. Ilman pääsäikeen lukitsemista.

Pelitilassa eli tilassa, jolloin sovellus käynnistetään rajoitteita ei tullut vastaan, aika testien kirjoittamiseen oli optimoitavissa. Testiautomaation kääntöpuolena voidaan pitää sitä, että muutoksia pelimekaniikkaan lykätään sillä perusteella, että se vaatisi isoja muutoksia testiautomaatioon tai että iso osa ajasta menee hukkaan, kun testiautomaatioon tarvitaan perusteellisia muutoksia jatkuvalla syötöllä. Molemmat voidaan ennakoida ja minimoida hyvällä suunnittelulla. Pienenä riskinä on, että jokin pohjatoteutus muuttuu merkittävästi, kuten pelimoottorin rajapinnat tai NUnit.

5 Yhteenveto

Yksikkö- ja integraatiotestien toimivuutta ja hyödyllisyyttä on testattu. Testit ovat todistetusti hyödyllisiä ja testejä voi tehdä normaalin ohjelmoinnin ohessa (testit edellä) tai jälkeen. Testien suorittaminen on helppoa, testit edellä tai jäljessä tuottavat parempaa ja

uudelleenkäytettävämpää koodia. Myös pelin tai sovelluksen ominaisuudet on suunniteltava siten, että ne täyttävät skenaarion kaikki vaatimukset sillä tasolla, että toiminnallisuus voidaan koneellisesti testata. Toiminnallisuuden koodaaminen on hitaampaa kuin nopea testailu, mutta kuitenkin nopeampaa pidemmällä aikavälillä, sillä koneellisesti suoritettua testiä voidaan nopeuttaa tai siirtää sen suoritus toiselle koneelle. Testien hankaluus voi johtua siitä, että testit edellä lähestymistapa tarvitsee sen, että ohjelmoijalla on jo tarvittava tieto testien kirjoittamiseksi. Tämän takia moni aloittelija ei kirjoita testejä, kun heillä ei ole osaamista määritellä koodia tai kuvitella kokemuksen kautta saatuja tilanteita. Kun koodaaminen tapahtuu aloittelijan innolla, niin tapahtuu se usein omassa tapauksessa kokeilemalla mitä tapahtuu sekä kopioi ja liitä menetelmällä paremman puutteessa. Kynnys opetella testiautomaatio samalla, kun opettelee koodaamaan ja kun on jo tottunut koodaamaan ja pärjäämään ilman testejä, on iso. Mikäli testien teko tulee oikeasti tarpeesta, näitä ongelmia ei ole. Tästä syystä onkin hyvä miettiä testattavia kohteita ja hyötyjä etukäteen. Tarvetta usein löytyy oikeasti haastavissa projekteissa, ja testiautomaatio todennäköisesti riippuen ongelmasta, voi estää ongelmien kasaantumisen. Testikohteina oli ilmiöitä, joiden toiminnan takaaminen on tapauskohtaisesti melkein tärkeintä ammuntapelissä, joka nojautuu vahvasti fysiikkamoottorin käyttöön ja on tarkoitettu moninpeliksi. Testikohteiden muuttuminen monimutkaisimmiksi nostaa testaamisen tasolle, jossa monen ihmisen on tarkistettava aseisiin kohdistuvat muutokset. Kuinka monta mahdollista suoraa ja epäsuoraa yhdistelmää syntyy ja kuinka monta pystytään ihmiskäsin tarkasti testaamaan?

Kun työskennellessä pääsee napin painalluksella testaamaan jonkin toiminnallisuuden eri pelinelementtejä kohtaan nopeasti, tarkasti ja helposti, on erittäin positiivinen asia. Mielestäni ei ole esteitä sille, että mennään testit edellä lähestymistapaa käyttäen, kunhan ympäristö sallii. Testiautomaation päivittäminen ja ylläpito isommalla projektilla on vielä kokeilematta. Se kuitenkin on selvää, että omilla päätöksillä voidaan tätäkin parantaa. Näin tehtiin myös projektissa, jossa aseiden laukaisu logiikka suunniteltiin siten, että ne sopivat yksinkertaiseen testiympäristöön ja nojautuen Unityn omaan komponentti arkkitehtuuriin. Mikäli testattavia olioita ja kohteita on paljon, niin tällainen tehostettu testiympäristö voi tehdä ison pelin haaveesta pienen tiimin todellisuutta.

Lähteet

Unity Technologies. Testieditorin dokumentaatio. https://docs.unity3d.com/2017.4/Documentation/Manual/testing-editortestsrunner.html. Luettu 2018-2020. Päivitetty 18.3.2020.

Pohja-aineistoa ohjelmistokehyksille. Dirk Riehle. Framework Design: A Role Modeling Approach. Ph.D. Thesis, No. 13509. Zürich, Switzerland, ETH Zürich, 2000. https://riehle.org/computer-science/research/dissertation/diss-a4.pdf. s. 1-20.

Dokumentaatio. GitHub. https://github.com/nunit/docs/wiki/TestCaseData. Luettu 2018-2020. Päivitetty 11.2.2019.

Ideoita kuviin. https://commons.wikimedia.org/wiki/File:Traditional-Shift-Left.jpg. Luettu 2019. Päivitetty 20.3.2015.

Testikehyksien dokumentaatiota. Verkkoaineisto. GitHub. https://github.com/nunit/docs/wiki/TestFixture-Attribute. Luettu 2018-2020. Päivitetty 7.7.2017.

Tietoa NUnit testikehyksestä. Verkkoaineisto Nunit. http://nunit.org/. Luettu 2018-2020. Päivitetty 13.3.2020.

Unityn puolen testikehyksien dokumentaatio. Verkkoaineisto Unity Technologies. 2019 https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/index.html. Luettu 2018-2020. Päivitetty 23.8.2019.

Yleistä tietoa ohjelmistotestauksesta. Verkkoaineisto. Edureka. 2019. https://www.edureka.co/blog/types-of-software-testing/. Luettu 31.3.2020. Päivitetty 22.5.2019.

Yleistä tietoa ohjelmistotestauksesta, raja-arvot. Verkkoaineisto. CS Helsinki. https://www.cs.helsinki.fi/u/aptuovin/testaus/Ohj_testaus_2013_6.pdf. Luettu 14.4.2020. Päivitetty. 4.4.2013.