Tampereen ammattikorkeakoulu

Jatkuva integraatio, jatkuva toimitus ja yksikkötestaus esimerkillä

Veikka Hupanen

OPINNÄYTETYÖ
Toukokuu 2023

Tietotekniikka
Ohjelmistotekniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikka
Ohjelmistotekniikka

HUPANEN VEIKKA:
Jatkuva integraatio, jatkuva toimitus ja yksikkötestaus esimerkillä

Opinnäytetyö 28 sivua, joista liitteitä 3 sivua
Toukokuu 2023

Tässä opinnäytetyössä tarkastellaan mitä jatkuva integraatio (continuous integration) ja jatkuva toimitus (continuous delivery) tarkoittavat, miten ne toimivat käytännössä ja miten yksikkötestaus saadaan niihin integroitua. Työn osana ja esimerkkinä toimii selainpelin kehitysprojekti, jossa peli kehitetään Unity-pelimoottorilla ja julkaistaan WebGL-ohjelmointirajapinnalla pelattavaksi selaimeen.

Työn tarkoituksena on havainnollistaa jatkuvan integraation ja toimituksen tärkeyttä ja hyötyjä sovelluskehityksessä sekä esitellä keskeisiä työkaluja, joita käytetään jatkuvan integraation ja toimituksen toteuttamisessa. Tärkeitä työkaluja, joita hyödynnetään esimerkkiprojektin kehityksessä ja ylläpitämisessä ovat Git-versionhallintajärjestelmä ja GitHub-verkkoalusta. Lisäksi käsitellään yksikkötestausta osana jatkuvaa integraatiota ja toimitusta sekä niiden toteuttamista Unity-pelimoottorissa C#-ohjelmointikielellä.

Tuloksena opinnäytetyöstä saadaan käsitys, mitä jatkuva integraatio, jatkuva toimitus ja yksikkötestaus ovat ja miksi niitä käytetään. Työstä saa hyvän pohjan laajasti käytetyistä ympäristöistä ja työkaluista edellä mainittujen menetelmien toteutuksessa. Esimerkkipelin kehityksen vaiheita seuratessa saa käytännön esimerkin, miten menetelmiä ja työkaluja käytetään yksinkertaisessa projektissa.

Asiasanat: jatkuva integraatio, jatkuva toimitus, yksikkötestaus, unity, testaus

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in ICT Engineering
Software Engineering

HUPANEN VEIKKA:
Continuous Integration, Continuous Delivery and Unit Testing with an Example

Bachelor's thesis 28 pages, appendices 3 pages
May 2023

This thesis examines what continuous integration and continuous delivery is, how they are used in practice and how unit testing can be integrated with them. As a part of the thesis, an example project in the form of a browser game is developed using the Unity game engine and deployed with the WebGL API to be playable using a web browser.

The goal of the thesis is to illustrate the importance and benefits of continuous integration and delivery in a software project, as well as showcase several popular tools that are used to implement continuous integration and delivery. Some important tools which are used in the example project are the Git version control system and the GitHub web platform. In addition, the thesis examines the use of unit testing as a part of continuous integration and deployment as well as their implementation in the Unity game engine using the C# programming language.

As a result of the thesis, we can get an understanding of what continuous integration, continuous delivery and unit testing is and why they are used. The thesis gives good base knowledge for using widely used development environments and tools to implement the processes. Following the development of the example project gives a practical demonstration of how the methods and tools can be used in a simple project.

Key words: continuous integration, continuous delivery, unit testing, unity, testing

SISÄLLYS

LYHENTEET JA TERMIT

CI	Continuous Integration (Jatkuva integraatio)
CD	Continuous Deployment (Jatkuva toimitus)
Unity	Unity Technologies -yrityksen kehittämä pelimoottori
C#	Microsoftin kehittämä oliopohjainen ohjelmointikieli, jota käytetään Unity-pelimoottorissa
Git	Linus Torvaldsin kehittämä hajautettu versionhallinta-järjestelmä
GitHub	Web-palvelu, joka tarjoaa mm. Git-versionhallintajärjes-telmää varten tietovaraston säilytyspaikan
Tietovarasto (repository)	Versionhallinnassa tiedostojen muutosten ja historian säilytyspaikka.
Yksikkötestaus	Yksinkertaisin testauksen muoto, jossa testataan yksit-täisiä metodeja tai funktioita
WebGL	JavaScript-kieleen perustuva ohjelmointirajapinta, jolla voidaan renderöidä 2D- ja 3D-grafiikoita selaimessa
GitHub Pages	GitHub-palvelu julkaisemaan verkkosivuja tietovarastoa käyttäen
Kanban-taulu	Projektinhallintatyökalu seuraamaan projektin tehtävien kulkua
Unity-skripti	C#-tiedosto, jonka avulla toteutetaan toiminnallisuutta pelin objekteille
SSH	Secure Shell on protokolla, jonka avulla salataan tieto-liikennettä
GameCI	Avoimen lähdekoodin työkalu jatkuvan integraation ja toimituksen toteuttamiselle pelien kanssa

1 JOHDANTO

Jatkuva integraatio ja toimitus (CI/CD) on tärkeä osa-alue nykyaikaista ohjelmistokehitystä, mikä tuo monia hyötyjä kehitystiimille ja projektille. Testaus ja varsinkin yksikkötestit ovat keskeinen osa CI/CD -prosessia.

Ensimmäisessä osassa käsitellään CI/CD:n ja yksikkötestauksen terminologiaa, hyötyjä, haittoja sekä sen vaatimia työkaluja. Git ja GitHub tulevat tutuksi CI/CD:n toteuttamisessa. Toisessa osassa yksinkertaisen selainpelin kehitys Unity-pelimoottorin avulla toimii esimerkkinä käytännön sovelluksesta ja miten se vaikuttaa kehitysprosessiin. Kehityksessä käytetään yksikkötestejä koodin laadun varmistamiseksi ja esitellään, miten niitä tehdään.

Tavoitteena on avata miksi ja miten CI/CD:tä sekä yksikkötestejä voi käyttää ohjelmistokehityksessä helposti ymmärrettävän esimerkin ja vaiheiden avulla. Esimerkkiprojektin vaiheita voidaan seurata oman projektin kehityksen avustamiseksi.

2 KÄSITTEET

2.1 CI/CD-prosessi

Jatkuva integraatio (engl. continuous integration, CI) ja jatkuva toimitus (continuous delivery, CD) ovat osa niin sanottua CI/CD-prosessia, joka on yksi DevOpsin ja modernin ohjelmistokehityksen pääpilareita. CI/CD:ssä yhdistetään kehitys, testaus ja toimitus suoraviivaiseen prosessiin, joka voi antaa monia hyötyjä projektin kehityksessä.

2.1.1 Jatkuva integraatio

Jatkuva integraatio on ohjelmointikehityksessä periaate, jossa ohjelmiston kehittäjät yhdistävät tuottamansa koodin versionhallintajärjestelmää käyttäen (tässä projektissa Git) tyypillisesti päivittäin. Tämän jälkeen koodi käännetään ja testataan automaattisesti, jotta mahdolliset virheet ja bugit huomataan ja voidaan korjata. Automaatio voi myös huomauttaa koodin tyylistä ja muista projektissa käytettävistä käytännöistä. (Fowler 2006; Rehkopf n.d.)

Ilman jatkuvan integraation käyttämistä kehittäjien on manuaalisesti kommunikoitava jokaisesta koodimuutoksesta, joka vaikuttaa ohjelmistoon. Tämä taas vaatii enemmän koordinointia ja byrokratiaa, nostaa projektin kuluja, aiheuttaa hitaampia julkaisuja ja lisää virheiden mahdollisuuksia. Isommissa projekteissa ja laajemmissa koodikannoissa nämä haitat kasvavat eksponentiaalisesti. (Rehkopf n.d.)

Jatkuvan integraation toteuttamisessa voi tulla haasteeksi testaus, sillä testien on oltava tarpeeksi kattavia, jotta virheet löytyvät ja ohjelmiston laatu pysyy hyvänä. Kaikkia virheitä ei ole todennäköisesti mahdollista löytää isommissa järjestelmissä, mutta testikattavuus on silti pyrittävä pitämään mahdollisimman hyvänä. Testien kehitykseen ja ylläpitoon kuluu lisäaikaa ja resursseja, joihin on varauduttava projektin suunnittelussa ja aikatauluttamisessa.

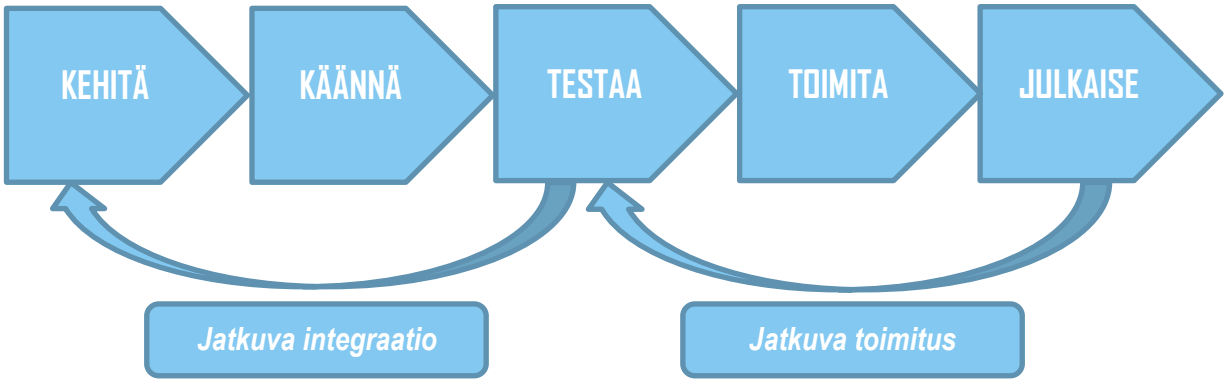
2.1.2 Jatkuva toimitus

Jatkuva toimitus on jatkuvan integraation seuraava vaihe ja nämä ovatkin kytköksissä toisiinsa. Kolmas vaihe, joka usein jätetään pois, on jatkuva julkaisu (continuous deployment). Kun puhutaan näiden menetelmien käytöstä yhdessä, käytetään usein nimitystä CI/CD-putki (pipeline).

Jatkuvassa toimituksessa kehitystiimi pyrkii siihen, että tuote on julkaisuvalmis jokaisen uuden koodin integroimisen jälkeen, koodissa ei ole virheitä, se on mennyt läpi testauksesta ja kehittäjät voivat manuaalisesti julkaista uuden version napin painalluksella. (Singh, 2023)

Jatkuva toimitus tehdään harvemmin kuin jatkuva integraatio. Jatkuvaa integraatiota tehdään oikeaoppisesti päivittäin tai monta kertaa päivässä, kun taas jatkuvassa toimituksessa varmistetaan, että koodi on julkaisuvalmis, mutta julkaisua voidaan tehdä silloin kun koodimuutokset ovat riittävät ja siitä olisi hyötyä käyttäjälle. Julkaistussa ohjelmistossa käyttäjät eivät välttämättä halua jatkuvia pieniä päivityksiä, varsinkin jos alhaallaoloaikoja vaaditaan päivitysten asentamiseen. Näin ollen järkevät väliajat päivityksille on otettava huomioon. (Pittet n.d.)

Harvemmin käytetty jatkuva julkaisu on kuin jatkuva toimitus, mutta uudet muutokset julkaistaan automaattisesti ilman kehittäjän hyväksynnän tarvetta, kunhan testit ovat menneet läpi. Tätä voidaan käyttää, jos halutaan jatkuvaa palautetta asiakkaalta pienistäkin muutoksista. (Pittet n.d.)



KUVIO 1. Jatkuvan integraation ja toimituksen vaiheet

2.2 Testaus

Testaus on yksi tärkeimmistä osa-alueista toimivan sovelluksen kehittämisessä. Testauksessa varmistetaan, että sovellus toimii halutulla tavalla, ilman koodivirheitä ja spesifikaatioiden mukaisesti. Testaus auttaa havaitsemaan koodin ongelmat ajoissa ennen kuin niiden päälle rakennetaan muita osia, jotka vaatisivat mahdollisesti lisäresursseja korjaamiseen jälkikäteen tai pahemmassa tapauksessa ennen kuin tuote toimitetaan asiakkaalle. Kriittisissä sovelluksissa ja ohjelmistoissa huomaamatta jääneet virheet voivat tuottaa valtavia taloudellisia vahinkoja tai jopa ihmishenkien menetystä, esimerkiksi terveydenhuollon laitteissa. (Rana 2022; Hamilton 2023)

2.2.1 Automaattinen testaus

Sovellusten testaus manuaalisesti vaatii usein huomattavasti resursseja, aikaa sekä henkilöstöä. Inhimillisten virheiden mahdollisuus myös nousee, kun testejä joudutaan tekemään käsin. Tämän takia varsinkin jatkuvassa integraatiossa ja toimituksessa pyritään automatisoimaan valtaosa testauksesta. Automaattisessa testauksessa kirjoitetaan valmiita testiskriptejä, joiden avulla testausohjelma tietää, mitä sovelluksessa on tarkoitus testata ja mikä testin tuloksen pitäisi olla. Testit voivat kattaa pieniä luokan metodeja koodissa tai monimutkaisempia toimintojen ketjuja käyttöliittymässä. Automaatio varmistaa, että testit suoritetaan samalla tavalla joka kerta, luotettavammin kuin manuaalisesti, kunhan testiskriptit on kirjoitettu hyvin. (Pittet, n.d.)

2.2.2 Yksikkötestaus

Yksikkötestaus, jota tässä työssä käytetään, on testauksen osa-alue, jossa keskitytään testaamaan pieniä osia koodista, kuten yksittäisiä metodeja tai luokkia. Yksikkötestien on tarkoitus olla pienikokoisia, jotta ne pyörisivät hyvin ja nopeasti automaattisessa testiympäristössä. Yksikkötesteihin joudutaan usein tekemään teko-olioita mallintamaan oikean koodin osia, mikäli oikeita olioita ei olisi käytännöllistä toteuttaa testiympäristössä tai jos niitä ei olla vielä luotu. Tämmöinen voisi olla esimerkiksi tekotietokannan luominen, joka ei häiritsisi oikeaa tietokantaa testauksessa.

Yksikkötestin kirjoittamisen vaiheet ovat yksinkertaiset. Ensin valitaan funktio tai metodi, jota halutaan testata. Sen jälkeen tehdään testiympäristö, jossa on tehty tarvittavat riippuvuudet funktion toimintaan mahdollisesti teko-olioilla tai muilla simuloivilla komponenteilla. Kun ympäristö on kunnossa, annetaan halutut syötteet, joilla testin tulisi toimia oikeassakin tilanteessa ja määritellään mitä tulosten tulisi olla. Lopuksi lisätään kutsu metodille tai funktiolle annetuilla syötteillä ja varmistetaan, että tulokset ovat samat kuin mitä odotettiin. Testiä säädetään, kunnes se läpäistään ja voidaan siirtyä seuraavaan testiin.

Yksikkötestejä ei voi kuitenkaan käyttää isompien kokonaisuuksien testaukseen samanaikaisesti ja pienemmissäkin projekteissa voi jäädä testitapauksia, joille ei ehditä tai ole resursseja kehittää yksikkötestejä niiden määrän takia.

3 TYÖKALUT JA KEHITYSYMPÄRISTÖ

3.1 Unity-pelimoottori

Unity on yksi käytetyimmistä ja monipuolisimmista pelimoottoreista nykypäivänä. Sen on kehittänyt Unity Technologies vuonna 2005 ja sitä on päivitetty jatkuvasti, jotta se pysyisi teknologian kärjessä. Unity käyttää ohjelmoinnissa C# -kieltä ja sillä voi kehittää ohjelmistoja Android-, iOS-, Windows-, MacOS- ja Linux-käyttöjärjestelmille sekä konsoleille. Myös HTML5 -tuki on olemassa, jota käytetään tässä projektissa selaintoiminnallisuuden toteuttamiseksi. Unitylla voi tehdä 2D- tai 3D -pelejä yhtä helposti. (Schardon, 2023)

3.2 Git-versionhallintajärjestelmä

Git on ilmainen ja avoimen lähdekoodin hajautettu versionhallintajärjestelmä. Sen kehitti Linus Torvalds vuonna 2005 auttamaan Linux-ytimen kehityksessä. Versionhallintajärjestelmä seuraa ja pitää kirjaa tiedostoihin tehdyistä muutoksista, mikä mahdollistaa helpomman kehittäjien yhteistyön ja kehityksen samassa koodikannassa.

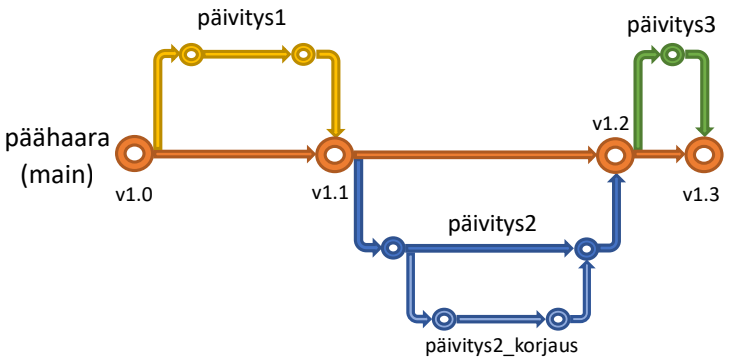
Gitissä ja versionhallinnassa yleisesti keskeisessä asemassa on koodi- tai tietovarasto (engl. repository, repo), jossa säilytetään muutokset ja niiden historia. Hajautetussa, toisin kuin keskitetyssä versionhallintajärjestelmässä, jokaisella kehittäjällä on oma tietovarasto, eivätkä kehittäjät ole riippuvaisia keskitetystä palvelimesta tai jatkuvasta nettiyhteydestä. Versionhallintajärjestelmän avulla kehittäjä voi tarkastella omia tai muiden tekemiä muutoksia ja halutessaan helposti palauttaa tiedostojen versiot edellisiin esimerkiksi ohjelmiston toiminnallisuutta rikkoneen koodimuutoksen jälkeen.

Versionhallintajärjestelmistä Git valittiin koska se on tehokas, joustava ja eniten käytetty järjestelmä. Laajan käyttäjäyhteisön vuoksi sillä on reilusti resursseja ja hyvää dokumentaatiota auttamaan mahdollisissa ongelmissa.

3.2.1 Haarat versionhallinnassa

Yksi Gitin keskeisimpiä ominaisuuksia ovat haarat (branches), jotka ovat viittaus tiettyyn kohtaan Gitin versiohistorian aikajanassa. Haaran luomisen jälkeen muodostuu uusi muutosten historian seuranta, joka ei seuraa päähaarassa (master/main branch) tapahtuvia muutoksia eikä päähaara seuraa uudessa haarassa tapahtuvia muutoksia. Haara mahdollistaa koodin muuttamisen ilman että se vaikuttaa muihin haaroihin. Muutosten jälkeen lähetetään haara etätietovarastoon, jossa sitä voi tarkastella ja yhdistää (merge) päähaaraan, kun ristiriitoja haarojen välillä ei löydy. Haarojen avulla voi olla monta kehittäjää tekemässä eri ominaisuuksia samaan aikaan häiritsemättä tuotannossa olevaa koodia.

Gitin kanssa on kaksi päätapaa tuoda muutokset yhdestä haarasta toiseen, yhdistäminen (merge) ja uudelleenjärjestäminen (rebase). Yhdistämistä käyttäessä, kun muutokset halutaan saada takaisin päähaaraan, jää kehityshaaran historia ja olemassaolo näkyviin projektin historiaan. Lisäksi se tuottaa jokaisella yhdistämisellä tiedon historiaan, joka voi vaikeuttaa historian lukemista ja ymmärtämistä. Uudelleenjärjestämisessä kun kehityshaara tuodaan päähaaraan, kehityshaaran muutokset muutetaan näyttämään siltä, että muutokset olisivat aina tapahtuneet päähaarassa, eikä kehityshaaraa olisi ollut olemassa. Tämä voi selventää projektin muutosten historiaa tekemällä siitä lineaarisen ja välttämällä tietoa yhdistämisistä.



KUVA 1. Esimerkki Git-haarautumiskaaviosta yhdistämisellä

3.2.2 Gitin toiminta ja komennot

Versionhallinnan käyttö alkaa tietovaraston alustamisesta. Gitin käyttö tapahtuu komentorivillä komentoja antamalla tai nykyään monissa kehitystyökaluissa käyttöliittymän avulla. Komennolla ’git init’ voidaan tehdä uusi Git-tietovarasto sillä hetkellä olevaan kansioon. Jos taas halutaan etätietovarastosta itselle paikallinen versio, käytetään ’git clone’ komentoa. Kun luodaan haara, jossa voi tehdä muutoksia päähaaraan koskematta, käytetään joko ’git branch <nimi>’ -komentoa tai ’git checkout -b <nimi>’ -komentoa, jolla myös siirrytään haarasta toiseen. ’git add’ lisää tiedostojen muutokset niin sanotulle valmistelualueelle (staging area), jossa voidaan tarkastella haluamiaan muutoksia ja niitä vastaavia versioita Gitin historiassa ennen lopullista tallentamista. Komennolla ’git commit’ tallennetaan muutokset paikallisen Git tietovaraston historiaan ja mukaan lisätään viesti, joka kertoo lyhyesti mitä muutoksia tehtiin.

Kun halutaan puskea muutokset omasta paikallisesta tietovarastosta etävarastoon, tehdään ’git push’ -komento ja muutosten tuominen etätietovarastolta toimii ’git pull’ -komennolla. Voidaan myös käyttää vetopyyntöjä, joista selitetään myöhemmin, tuomaan muutokset turvallisemmin etätietovarastoon. Haarojen yhdistämisen voi tehdä Gitin avulla ’git merge’ -komennolla, mutta tässä projektissa käytetään GitHub-verkkoalustan toimintoja yhdistämiseen.

3.3 GitHub-verkkoalusta

GitHub on versionhallintaa tukeva avoimen lähdekoodin verkkopohjainen alusta. GitHubissa voi tallentaa omaa koodia ilmaiseksi pilveen ja tarkastella muiden luomia projekteja. Tietovarastoissa näkyy kaikki kooditiedostot ja dokumentaatio sekä niiden versiohistoriat. Tärkeä ominaisuus GitHubissa on mahdollisuus helposti kloonata toisten julkisia tietovarastoja itselleen, jolloin niitä voi muokata omassa ympäristössä ja ehdottaa muutoksia sen ylläpitäjille.

Vetopyyntö (pull request) on myös keskeinen ominaisuus, jonka avulla ehdotetaan muutoksia tiedostoihin, jonka jälkeen tietovaraston jäsenet voivat tarkastaa

koodimuutokset ja hyväksyä tai hylätä ne. Isommissa tietovarastoissa vetopyynnöillä on lähes aina useampi kehittäjä, jotka käyvät muutokset läpi ennen päätöksentekoa. Näin vältetään ongelmallisen koodin joutuminen päähaaraan.

GitHub toimii nykyään sosiaalisena verkostona monille kehittäjille ja siellä voi seurata toisia kehittäjiä ja osallistua moniin eri projekteihin tai näyttää omat projektit muille helposti koottuna. Github tarjoaa myös monia hyödyllisiä työkaluja projektienhallintaan, joista useita käytetään tämän työn kehityksessä, kuten Kanban taulu -tyylistä ”Projects” työkalua, jonka avulla voi seurata tehtävien kulkua ja vaiheita.

3.3.1 GitHub Actions -palvelu

GitHub Actions on jatkuvan integraation ja toimituksen alusta, jonka avulla voi automatisoida ohjelmiston rakennuksen, testauksen ja toimituksen. Actions-palvelun avulla voi automatisoida ja pyörittää muitakin työnkulkua helpottavia tapahtumia GitHubissa, mutta tässä projektissa käytetään vain sovelluksen automaattista rakennusta, testausta ja toimitusta selaimeen pelattavaksi. GitHub tarjoaa ilmaisen virtuaalikoneen CI/CD-prosessia varten, mutta tarvittaessa voi myös käyttää omia virtuaalikoneita tai pilvipalveluita. CI/CD-putken toteutusta varten löytyy ilmaisia käyttäjien ja GitHubin julkaisemia pohjia erilaisille työprosesseille (workflows), joilla saa automatisoitua eri tehtäviä. Tässä työssä käytetään avoimen lähdekoodin GameCI-pohjaa automatisoimaan Unity-testit, jotka toimivat eri lailla kuin muiden ohjelmistojen testit.

3.3.2 Github Pages -verkkosivu

GitHub tarjoaa ilmaisen Pages-verkkoisännöinnin tietovarastoille, johon voi julkaista ja saada näkyviin oman projektin. Jokaiselle tietovarastolle GitHubissa voi olla yksi oma sivu. Pages integroituu helposti GitHubin ja Gitin kanssa ja se tarjoaa HTTPS-salauksen parempaa tietoturvaa varten.

4 PELIN KEHITYS

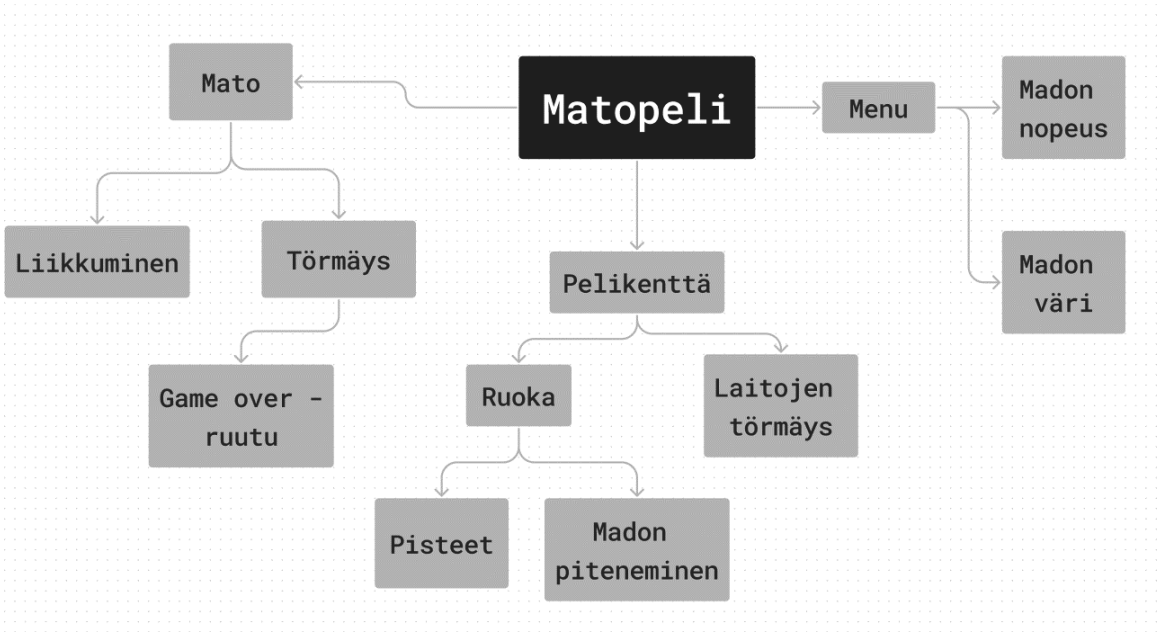
4.1 Esimerkkipeli

Työn käsitteiden havainnollistamiseksi tehtiin peli, joka toteutetaan Unity-pelimoottorilla. Peli rakennetaan WebGL (Web Graphics Library) -tuen kanssa, joka mahdollistaa 2D- ja 3D-grafiikoiden renderöinnin selaimessa ilman lisäosia. Näin voidaan pelata peliä selaimessa ilman esivaatimuksia. Pelin tyypiksi valittiin klassinen matopeli, jossa liikutetaan matoa ympäri pelikenttää ja kerätään ruokaa, joka pidentää matoa ja lisää pisteitä. Peli loppuu, jos mato osuu itseensä tai kentän reunoihin.

Pelin kehityksessä käytetään jatkuvaa integraatiota ja toimitusta helppoon päivittämiseen ja testien pyörittämiseen. Yksikkötestit takaavat koodin laadun ja pelin toimivuuden projektissa. GitHub-verkkoalusta toimii etätietovaraston säilytyspaikkana sekä tarjoaa ilmaiset testien ja julkaisun automatisointityökalut. GitHub Pages -sivuille julkaistaan peli, jossa sitä voi pelata.

4.2 Pelin suunnittelu

Jotta voidaan saada käsitys, millainen peli tai projekti ylipäätään toteutetaan, tarvitaan suunnitelma, jossa havainnollistetaan mitä vaatimuksia projektilla on ja miten ne toteutetaan. Kun kyseessä on pienempi esimerkkipeli, suunnitelman ei tarvitse olla laajamittainen ja pelin vaatimukset toteutetaan yksinkertaisella visuaalisella kartalla, jossa tehtävät ovat jaettu pieniin osiin.



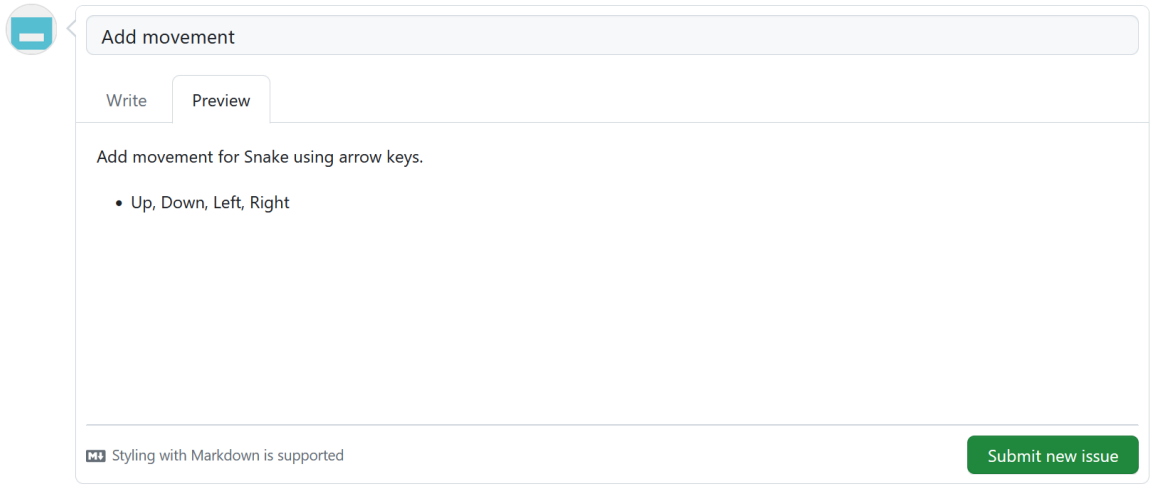
KUVIO 2. Pelin suunnitelma

4.2.1 Git- ja GitHub-integraatio

Jotta saadaan jatkuvaa integraatiota ja julkaisua käytettyä, joudutaan luomaan GitHub-tietovarasto projektille sekä integroimaan se Gitin kanssa. Tietovarasto luodaan GitHubissa ja tuodaan omalle koneelle, tosin sen voi myös luoda omalla koneella ja viedä GitHubiin. Kun tietovarasto on tehty, kirjoitetaan readme-tekstitiedosto, jossa voidaan kertoa hieman projektista mahdollisille tietovaraston lukijoille ja osallistujille. Tietovaraston saamiseksi omalle koneelle tehdään kloonaus Gitin avulla. Tässä käytetään SSH (Secure Shell) -protokollaa salaamaan oman tietovaraston ja GitHubin etätietovaraston välinen liikenne. ’git clone’ -komennolla saadaan tehtyä tietovarasto GitHubissa olevien tiedostojen perusteella. Kun tietovarasto on kopioitu omaan hakemistoonsa, voidaan luoda uusi haara ja alkaa tehdä muutoksia tiedostoihin, jotka Gitin avulla pusketaan GitHub-etätietovarastoon. GitHubissa voi tehdä vetopyynnöt muutoksille ja yhdistää ne päähaaraan ja siitä eteenpäin tuotantoon.

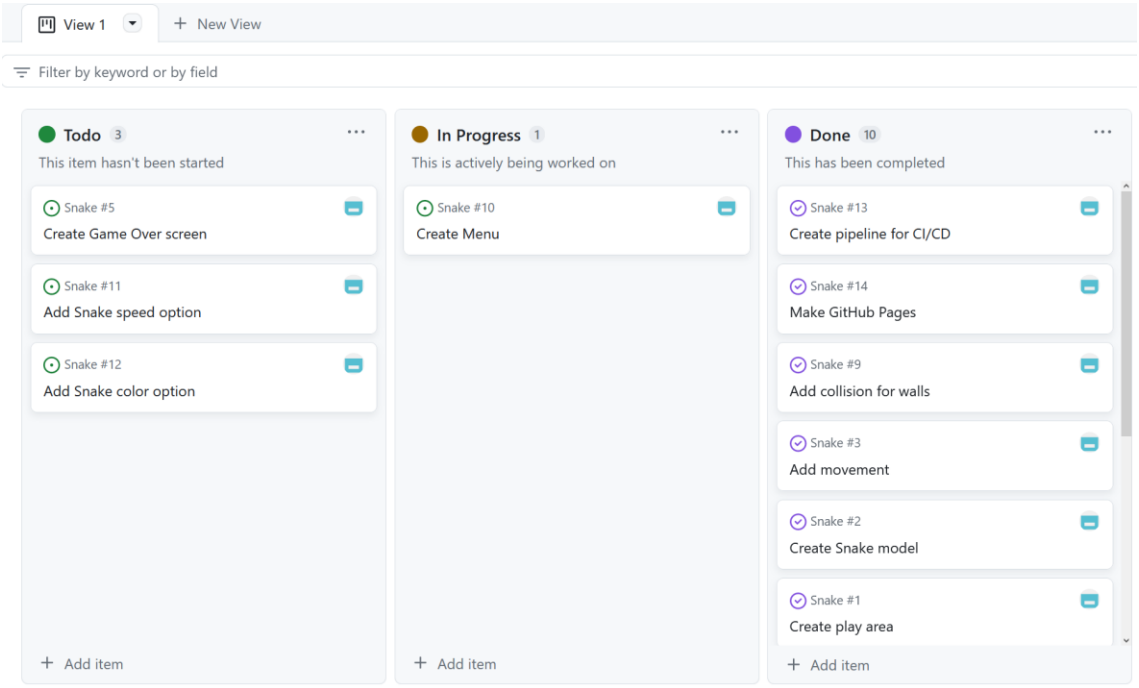
Jokaiselle pelin ominaisuudelle, joka on alustavasti mukana suunnitelmassa, tehdään oma GitHub-issue tai -tehtävä, jossa kerrotaan mitä ominaisuus pitää sisällään ja annetaan tarvittaessa lisätietoa. Nimeksi annetaan helposti tunnistettava ja lyhyt kuvaus, josta näkee nopeasti ja karkeasti mitä kyseinen ominaisuus si-

sältää. Tehtävät voi jakaa eri kehittäjille ja niistä näkee helposti mikä tehtävä kuuluu kenellekin. Tehtäviin lisätään tässä projektissa myös pelin ulkopuoliset tehtävät, kuten automaattisen testauksen ja julkaisun tekeminen sekä Pages-osuuden luominen. Tehtävät ovat pilkottu jo suunnitteluvaiheessa riittävän pieniksi, jotta versioita voidaan julkaista nopeasti ja saadaan mahdollista palautetta lyhyin väliajoin.



KUVA 2. GitHub tehtävän luominen

Pelin kehityksen helpottamiseksi käytetään GitHubin tarjoamaa Kanban-tyylistä taulua, jolla seurataan projektin tehtävien kulkua ja valmistumista. Taulussa on kolme osiota, joiden kautta tehtävät etenevät. Ensin tehtävät menevät ’aloittamattomat’ sarakkeeseen, josta voi nähdä kaikki aloittamattomat tehtävät ja valita sopiva työstettävä ominaisuus. Kun tehtävä on aloitettu ja sille on annettu vastuussa oleva kehittäjä, se siirretään ’työn alla’ osioon. Valmiit tehtävät menevät ’valmis’ sarakkeeseen.



KUVA 3. Projektin taulu GitHubissa

4.3 Pelin perusteiden kehitys

Jatkuvan integraation ja toimituksen käyttö vaatii ensin jotain mitä integroida ja toimittaa, joten muutama pelin ominaisuus tehdään ensin valmiiksi yksikkötestien kanssa, jotta automaattisen testauksen ja julkaisun putki voidaan saada toimimaan. Ensimmäiseksi tehtiin kolme oleellisinta ominaisuutta: pelikenttä, 2D-malli madosta (käärmeestä) ja liikkuminen.

Pelille tehtiin tekstuurit eli visuaaliset osat yksinkertaisella pikselitaidetyylillä. Tekstuureita voi myös ostaa tai hankkia ilmaiseksi netistä tai Unityn sisältökaupasta, mutta tässä esimerkissä ne tehtiin itse.

4.3.1 Unity-pelimoottorilla kehittäminen

Unity-pelin kehittämiseksi ensin ladataan Unity ja asennetaan se WebGL-tuen kanssa. Kun pelitiedostot ovat luotu ja pelimoottori toimii, tuodaan tekstuuritiedostot moottoriin. Tekstuureille asetetaan oikeat mittasuhteet ja tarkistetaan, että ne sopivat yhteen. Pelikenttä lisätään kehitysalueelle (Scene) ja luodaan sille törmäyksen tunnistava komponentti. Madon mallille, tällä hetkellä vain pää, lisätään

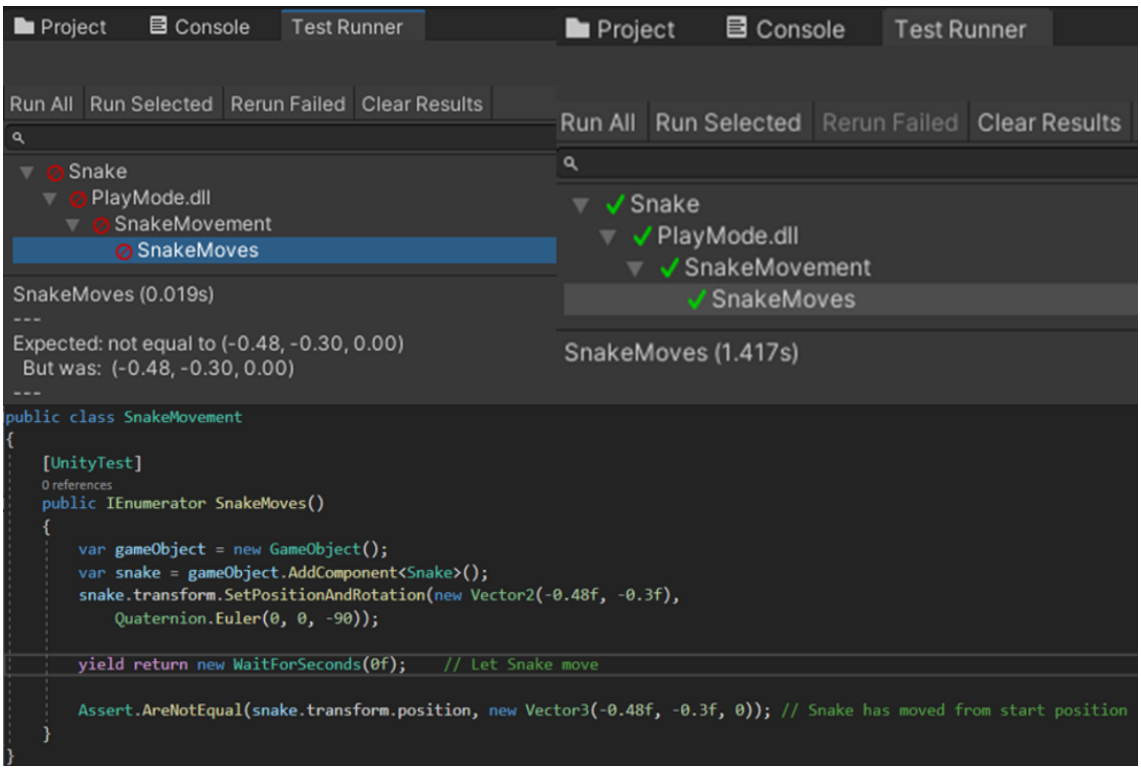
myös törmäys ja liikkumiskomponentti. Jotta saadaan ohjelmoitua madon toimintaa, tehdään C#-skriptitiedosto ja liitetään se madon malliin. Skriptitiedostossa ohjelmoidaan madon luominen pelin alkaessa sekä liikkuminen käyttäjän syötteillä eli nuolinäppäimillä. Ks. Liite 1.

4.3.2 Unity-yksikkötestit

Unityn yksikkötestit poikkeavat luonnollisesti muista yksikkötesteistä, sillä ne ovat kehitetty Unity-moottorin pyöritettäviksi, ja ne voidaankin pyörittää moottorin sisällä. Unityssa on kaksi erilaista testityyppiä: EditMode-testit sekä PlayMode-testit. EditMode-testeissä testit ajetaan ilman pelin pyörittämistä, joten ne ovat nopeampia mutta niillä ei voi testata esimerkiksi komponenttien interaktiota. Sillä suurin osa pelin toiminnallisuuksista vaatii eri osien toimimista yhdessä, useimmat testit ovat toteutettu PlayMode-testeilla.

Testeissä joudutaan luomaan oliot, joita halutaan testata oikeiden olioiden perusteella. Testeihin lisätään vain tarvittavat komponentit ja oliot, joilla haluttu testi saadaan läpäistyä, jotta testien koko ja resurssivaatimukset eivät paisu. Aina kannattaa ensin pyörittää epäonnistuva testi, jotta tiedetään, että testi ei läpäise automaattisesti. Tämä myös varmistaa, että testaus itsessään toimii, eikä testin kääntämisessä ole virheitä.

Alla olevassa kuvassa nähdään madon liikkumista testaava yksikkötesti. Testissä alustetaan ainoastaan mato-olio, sillä muuta testin ei tarvitse testata. Madolle asetetaan tietty aloituspiste ja suunta. ’WaitForSeconds’ -funktion avulla odotetaan, että madolla on aikaa liikkua pois sen aloituspisteestä. Testin tuloksena pitäisi olla totta väite, että madon testin lopussa oleva sijainti ja sen aloitussijainti eivät ole samat. Ensin kuitenkin asetetaan odotusfunktioon aika nollaan, jotta tiedetään, että testi ei läpäise vahingossa. Jos madolla ei ole aikaa liikkua, sen pitäisi olla samassa pisteessä kuin mistä se aloitti ja testi epäonnistuu. Kun odotusfunktio saa oikean arvon ja testin väittämä on totta, tiedetään että testi ja madon liikkumismetodi toimivat.



KUVA 4. Madon liikkumisen Unity-yksikkötestin ajaminen ja testin koodi

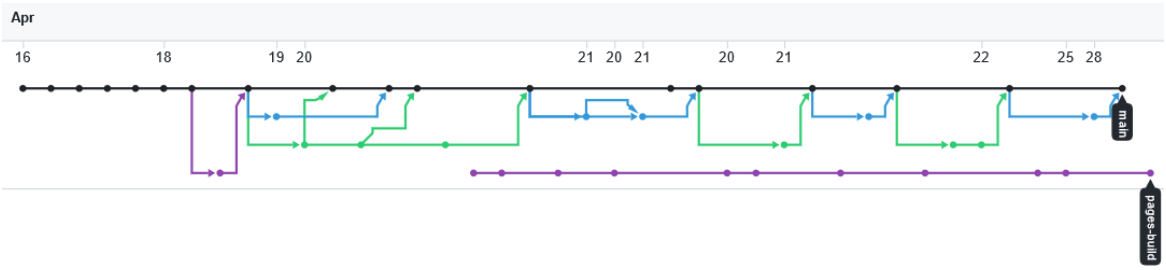
4.3.3 Git-versionhallinnan käyttö

Jotta voidaan tehdä turvallisesti muutoksia versionhallintaa käyttäen, annetaan komentorivillä projektin hakemistokansiossa komento ’git checkout -b <haaran nimi>’. Tämän jälkeen kaikki muutokset tapahtuvat ainoastaan uudessa haarassa eivätkä häiritse päähaaraa.

Kun kehityksessä oleva ominaisuus on valmis ja halutaan saada haara puskettua tietovarastoon GitHubissa, lisätään ensin tiedostot, joiden muutokset halutaan ottaa mukaan seuraavaan commit-käskyyn oman tietovaraston valmistelualueelle ’git add .’ -komennolla, jossa piste tarkoittaa jokaista hakemistossa olevaa tiedostoa. Kun on varmistettu, että kaikki muutokset ovat kunnossa ja tallennettavissa esimerkiksi ’git status’ komentoa hyödyntämällä, tehdään muutokset Git historiaan ja lähetetään ne etätietovarastoon ’git commit’ ja ’git push’ komennoilla.

4.3.4 GitHubin käyttö versionhallinnassa

Kun muutokset ovat saatu Gitin avulla puskettua etätietovarastoon, GitHubissa näkyy uusi haara, jota voi tarkastella ja josta tehdään vetopyyntö yhdistää muutokset päähaaraan. Vetopyyntöön voidaan asettaa tarkastushenkilö(t), joka käy läpi muutokset ja hyväksyy vetopyynnön. Vetopyyntöä hyväksyessä voidaan valita GitHubissa yksi kahdesta tavasta yhdistää uusi haara päähaaraan, joko yhdistäminen (merge) tai uudelleenjärjestäminen (rebase). Tässä projektissa valitaan yhdistäminen, sillä se on aloittelijaystävällisempi tapa ja tietovaraston historian siistinä pitäminen ei ole prioriteetti.



KUVA 5. GitHub-tietovaraston haarojen aikajana projektin edetessä

4.3.5 Automatisointi

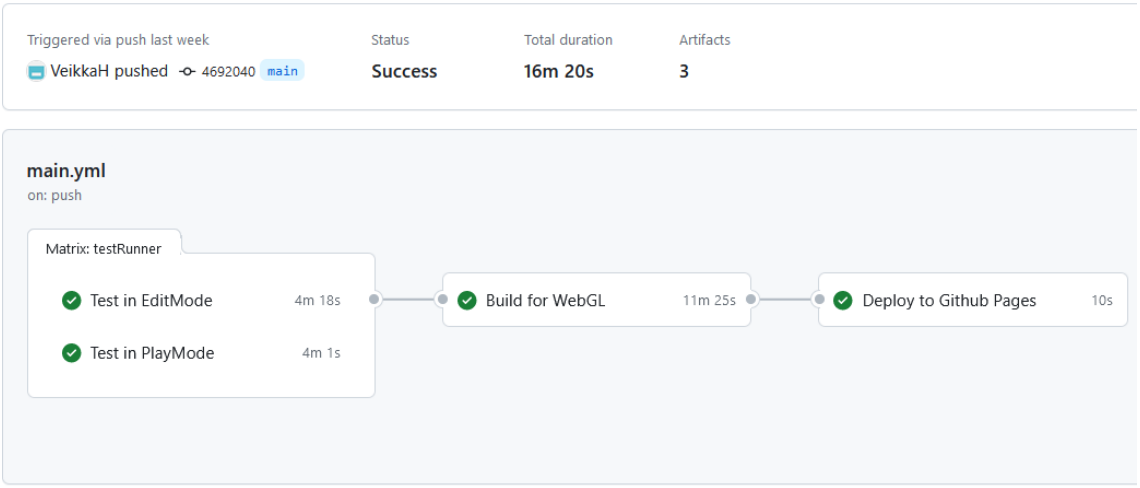
Testien ja pelin rakentamisen automatisointi on tärkein tehtävä jatkuvan integraation ja toimituksen toteuttamisessa. Koska Unity-testien automatisointi ei toimi kuten muut yksikkötestit ja ne tarvitsevat rakennetun pelin sekä testiympäristön toimiakseen, käytetään avoimen lähdekoodin GameCI-työkalua apuna.

Ensin täytyy saada Unity-lisenssikoodi, jonka saa Unityn sivuilta. Lisenssin avulla GameCI voi rakentaa Unity-moottorista version testiympäristössä ja pyörittää testit. GitHub Actions toimii työprosessitiedostolla, joka on kirjoitettu YAML-merkintäkielellä. Tiedosto on tallennettu tietovarastoon, josta GitHub sen hakee ja toteuttaa. Tiedostossa voi määrätä milloin työprosessi alkaa, esimerkiksi ainoastaan muutosten työntämisellä päähaaraan. Testien pyöritysympäristön voi valita sekä suorittaa muiden kehittämiä työprosesseja tiedostosta käsin.

Projektin YAML-tiedostossa käytetään hieman muokattua GameCI-mallipohjaa, johon on lisätty resursseja ja aikaa säästäviä lisäyksiä. Työprosessi muokattiin alkamaan vain ’push’-tapahtumalla. Tiedosto hakee Unity-lisenssin GitHubissa säilytetyistä tiedoista. Verifioinnin jälkeen työprosessi suorittaa kolme työtä:

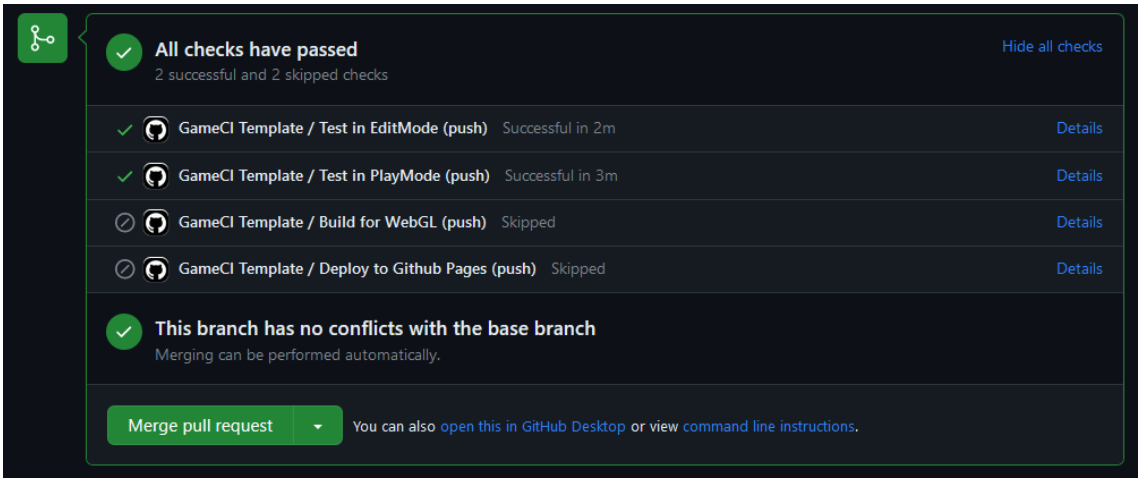
Unity-yksikkötestien ajamisen, WebGL-koontiversion (build) rakentamisen ja GitHub Pages -sivulle pelin julkaisun. Viimeiset kaksi työtä suoritetaan vain, jos muutokset tapahtuvat päähaarassa. Näin ei tuhlata resursseja turhaan rakentamiseen eikä sivuille julkaista mahdollista virheellistä versiota pelistä. Ks. Liite 2.

Alla olevassa kuvassa näkyy GitHub Actions -työprosessi toteuttamassa pelin testauksen ja julkaisun. Ensin prosessi pyörittää pelin yksikkötestit ja varmistaa niiden suoriutumisen hyväksytysti. Jos testit läpäisevät, rakennetaan versio pelistä, joka pyörii WebGL-ohjelmointirajapinnan avulla. Kun peli on rakennettu, se julkaistaan GitHub Pages -sivulle. Ylhäältä näkee prosessin keston.



KUVA 6. GitHub Actions -työprosessi testaa, rakentaa ja julkaisee pelin

Koska testit pyörivät automaattisesti myös kehityshaaroissa, voidaan nähdä ennen muutosten yhdistystä, toimiiko versio vai ei suoraan GitHubissa. Kehittäjien ei tarvitse tuoda versiota omaan tietovarastoon, pyörittää Unity-moottoria ja tarkistaa manuaalisesti, jos testit toimivat.

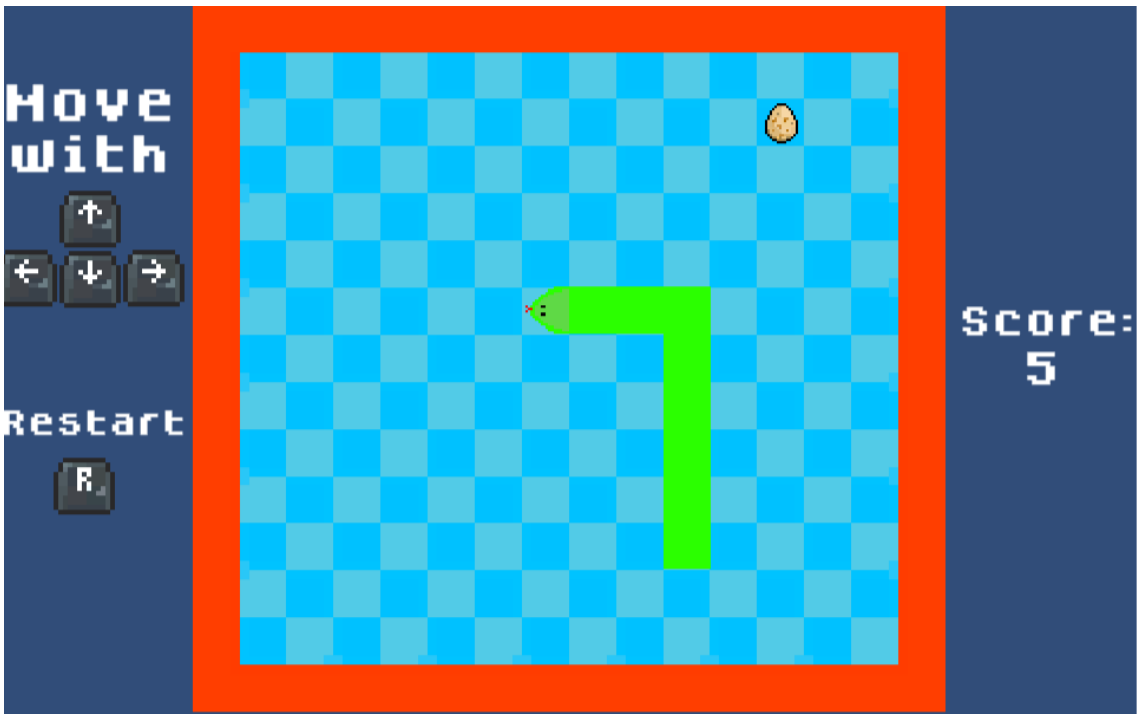


KUVA 7. Testien läpäisy ennen vetopyynnön hyväksymistä GitHubissa

4.4 Pelin viimeistely

Kun automaattinen testaus, rakennus ja toimitus on saatu toimimaan, loppuprojektin toteutus helpottuu merkittävästi. Jokaiselle pelin ominaisuudelle tehdään uusi haara, kehitetään toiminnallisuus ja testit ja pusketaan GitHub-etätietovarastoon. GitHubissa nähdään, toimiiko testit ennen kuin yhdistetään kehityshaara päähaaraan ja voidaan sen jälkeen tuoda muutos päähaaraan huoletta, jolloin GitHubin työprosessi rakentaa uuden peliversion ja julkaisee sen pelattavaksi Pages-sivulle.

Peliin lisättiin ruokaolio tekstuurin kanssa, joka luodaan pelikentälle satunnaiseen sijaintiin, jonka mato voi syödä ja kasvaa. Ruokaan törmätessä ruoka katoaa ja syntyy uudelleen toiseen satunnaiseen paikkaan. Törmäys madolla toimii itsensä sekä pelikentän laitojen kanssa. Toteutettiin myös pisteiden seuraaminen sekä ohjeet madon liikuttamiseen.



KUVA 8. Lopullinen versio pelistä

Peli pelattavissa osoitteessa: https://veikkah.github.io/Snake/
GitHub tietovarasto: https://github.com/VeikkaH/Snake

5 POHDINTA

Opinnäytetyössä käsiteltiin jatkuvan integraation ja toimituksen periaatteita ja hyötyjä, sekä yksikkötestausta ja niihin liittyviä työkaluja. Työstä saa toivottavasti helposti ymmärrettävän kattauksen kyseisistä aiheista.

Opinnäytetyön toisessa osiossa kehitettiin peli esiteltyjen menetelmien perusteella. Projektin kehityksestä saa mallinnettavan suunnitelman ja oppaan samantyyppisen tai samoja menetelmiä käyttävän projektin toteuttamiseen. Varsinkin aloitteleva ohjelmistokehittäjä voi saada hyviä esimerkkejä ja käytäntöjä, joita voi hyödyntää omassa kehityksessä. Suositut ja monissa oikeiden työpaikkojen käytössä olevat työkalut ja alustat tulevat tutuiksi selitysten ja kuvien kanssa.

Peliprojektin kehityksessä auttoi aiemmin saatu kokemus kyseisten menetelmien ja alustojen käytöstä, mutta silti joutui tekemään työtä faktojentarkistuksessa esittelyosiossa sekä Unity-pelimoottorin ja C#-kielen opettelemisessa. Joitain ominaisuuksia jäi pelin lopullisesta versiosta puuttumaan pelin kehitykselle annetun ajan takia ja koska tarvittavat kehitysvaiheet ja menetelmät saatiin jo esiteltyä.

LÄHTEET

Fowler, M. 2006. Continuous Integration. Verkkosivu. Viitattu 9.3.2023. https://martinfowler.com/articles/continuousIntegration.html

Rehkopf, M. n.d. What is continuous integration? Verkkosivu. Viitattu 9.3.2023. https://www.atlassian.com/continuous-delivery/continuous-integration

Singh, G. 2023. Continuous Integration and Continuous Delivery | Complete Guide. Verkkosivu. Viitattu 10.3.2023. https://www.xenonstack.com/blog/continuous-integration-and-delivery

Pittet, S. n.d. Continuous integration vs. delivery vs. deployment. Verkkosivu. Viitattu 10.3.2023. https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

Schardon, L. 2023. What is Unity? – A Guide for One of the Top Game Engines. Verkkosivu. Viitattu 10.3.2023. https://gamedevacademy.org/what-is-unity/

Hamilton, T. 2023. What is Software Testing? Definition. Verkkosivu. Viitattu 13.3.2023. https://www.guru99.com/software-testing-introduction-importance.html

Rana, K. 2022. What is Software Testing? The Complete Software Testing Guide. Verkkosivu. Viitattu 13.3.2023. https://artoftesting.com/what-is-software-testing

Pittet, S. n.d. The different types of software testing. Verkkosivu. Viitattu 13.3.2023. https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing

LIITTEET

Liite 1. Pelin ensiominaisuuksien C#-skriptitiedosto

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

Unity Script (1 asset reference) | 13 references
public class Snake : MonoBehaviour
{
    public Vector2 direction;
    Unity Message | 0 references
    void Start()
    {
        reset();
    }

    2 references
    void reset()
    {
        transform.SetPositionAndRotation(new Vector2(0f, 0f),    // Set start point, f for float
            Quaternion.Euler(0, 0, -90));                        // Point Snake right
        direction = Vector2.right;
        Time.timeScale = 0.1f;
    }
    Unity Message | 0 references
    void Update()       // Called every frame
    {
        getUserInput();
    }

    1 reference
    void getUserInput()
    {
        if (Input.GetKeyDown(KeyCode.UpArrow))
        {
            direction = Vector2.up;
            transform.rotation = Quaternion.Euler(0, 0, 0);
        }
        else if (Input.GetKeyDown(KeyCode.RightArrow))
        {
            direction = Vector2.right;
            transform.rotation = Quaternion.Euler(0, 0, -90);
        }
        else if (Input.GetKeyDown(KeyCode.LeftArrow))
        {
            direction = Vector2.left;
            transform.rotation = Quaternion.Euler(0, 0, 90);
        }
        else if (Input.GetKeyDown(KeyCode.DownArrow))
        {
            direction = Vector2.down;
            transform.rotation = Quaternion.Euler(0, 0, -180);
        }
        else if (Input.GetKeyDown(KeyCode.R))
        {
            reset();
        }
    }
    Unity Message | 0 references
    void FixedUpdate() // Called every fixed framerate frame, affected by Time.timescale
    {
        moveSnake();
    }
    1 reference
    void moveSnake()
    {
        float x = transform.position.x + direction.x;
        float y = transform.position.y + direction.y;
        transform.position = new Vector2(x, y);
    }
}
```

Liite 2. GitHub Actions -työprosessin YAML-tiedoston osia GameCI-mallipohjaa käyttäen

```yaml
name: GameCI Template

on: push

env:
  UNITY_LICENSE: ${{ secrets.UNITY_LICENSE }}

jobs:
  testRunner:
    name: Test in ${{ matrix.testMode }}
    runs-on: ubuntu-latest
    strategy:
      fail-fast: false
      matrix:
        testMode:
          - EditMode
          - PlayMode
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Run tests
        uses: game-ci/unity-test-runner@v2
        id: testRunner
        with:
          testMode: ${{ matrix.testMode }}
          checkName: ${{ matrix.testMode }} test results

  buildWebGL:
    if: github.ref == 'refs/heads/main'
    needs: testRunner
    name: Build for WebGL
    runs-on: ubuntu-latest
    strategy:
      fail-fast: false
    steps:
      - uses: game-ci/unity-builder@v2
        with:
          targetPlatform: WebGL

      - uses: actions/upload-artifact@v2
        with:
          name: build-WebGL
          path: build/WebGL

  deployPages:
    if: github.ref == 'refs/heads/main'
    needs: buildWebGL
    name: Deploy to Github Pages
    runs-on: ubuntu-latest
    steps:

      - name: Deploy
        uses: JamesIves/github-pages-deploy-action@4.1.4
        with:
          branch: pages-build
          folder: build/WebGL
```