REAKTIIVINEN OHJELMOINTI PELIKÄY-TÖSSÄ

Atso Sariola

Opinnäytetyö
Marraskuu 2017
Tietojenkäsittely
Pelituotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Pelituotanto

SARIOLA, ATSO:
Reaktiivinen ohjelmointi pelikäytössä

Opinnäytetyö 42 sivua
Marraskuu 2017

Tämän opinnäytetyön tarkoitus oli toteuttaa kaupallisen Unity3D-moottoria käyttävän videopelipilotin keskeiset ohjelmointitarpeet reaktiivista ohjelmointia hyödyntäen UniRx-kirjastolla. Pilotti tehtiin Red Stage Entertainment Oy:n toimeksiantona. Opinnäytetyön tavoite oli selvittää reaktiivisen ohjelmoinnin soveltuvuutta Unity3D-pelikehitykseen.

Opinnäytetyössä perehdyttiin reaktiiviseen ohjelmointiin teknologiana ja selvitettiin sen peruskäyttöä helmikaavioiden avulla. Lisäksi työssä kuvataan videopelipilotin puitteissa esiin tulleita ongelmia ja niiden ratkaisuja.

Opinnäytetyössä havaittiin, että reaktiivinen ohjelmointi soveltuu Unity3D-pelikehitykseen hyvin. Tekniikkaa ei kuitenkaan voida helposti ottaa käyttöön yrityksissä tekniikan omaksumiseen vaadittavan opetteluajan vuoksi. Työn yhteydessä tuotetun videopeliprojektin keskeiset ohjelmointitarpeet toteutuivat, vaikka pilotti ei valmistunut kokonaisuudessaan.

Asiasanat: Unity3D, reaktiivinen ohjelmointi, ReactiveX, pelit

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Game Development

SARIOLA, ATSO:
Reactive Programming in Game Development

Bachelor's thesis 42 pages
November 2017

The purpose of this thesis was to implement a commercial Unity3D-engine based videogame using reactive programming techniques and the UniRx-library. The objective was to examine the viability of reactive programming for Unity3D game development.

The first chapters introduce reactive programming and the ReactiveX libraries. The chapters explain what reactive programming is, what are its essential concepts and how they interact with each other.

The later chapters provide a look into the problems faced during the implementation of a Unity3D game and their solutions using reactive techniques and tools. Reactive event streams are visualized using pearl diagrams.

The example situations in this thesis describe typical solutions to reactive programming challenges one might face when developing a Unity3D-game. Based on this project, reactive programming is a good fit for game development, but it is not necessarily easy to introduce into any existing project thanks to its steep learning curve.

Key words: Unity3D, reactive programming, ReactiveX, games

SISÄLLYS

1 JOHDANTO ... 6
2 Reaktiivinen ohjelmointi ... 8
2.1 ReactiveX ... 8
2.1.1 Observable ... 8
2.1.2 Observablen lämpötila ... 9
2.1.3 Operaattoreista ... 10
2.1.4 Luovat operaattorit ... 10
2.1.5 Seulovat operaattorit ... 11
2.1.6 Muovaavat operaattorit ... 13
2.1.7 Yhdistävät operaattorit ... 14
2.1.8 Muista operaattoreista ... 14
2.2 UniRx ... 15
3 Rx Unity3D-projektissa ... 16
3.1 Peruskäyttö ... 16
3.2 Komponenttien viestintä ... 22
3.3 Syötekäsittely ... 24
3.4 Syötteen hahmonsovitus ... 33
4 Päätäntö ... 40
LÄHTEET ... 42

ERITYISSANASTO

Unity3D	Suosittu 3D-pelimoottori, jolla opinnäytetyön projekti on toteutettu
peliobjekti, objekti	Unity3D:n 3D-tilassa oleva kappale, jolla voi olla komponentteja
komponentti	Unity3D-skriptauksen perusyksikkö, joiden sisällä lähes kaikki Unity3D-projektien C#-koodi suoritetaan
tagi	Unity3D:n toiminto, jolla peliobjektille voidaan antaa tekstimuotoinen tunniste
asynkroninen	Ohjelman operaatio, jonka suorittaminen ei tapahdu metodin kutsumishetkellä
funktio	Looginen yksikkö, joka määrittelee jonkinlaisen operaation
C#	Unity3D-kehityksessä käytetty imperatiivinen ohjelmointikieli
geneerinen	C#-kielen toiminto, joka sallii luokan tai metodin toteuttamisen käsitellystä tyypistä riippumattomalla tavalla
rajapinta	C#-kielen rakenne, jolla määritellään rajapintaa käyttävien luokkien julkisia metodeja ja kenttiä niin, että toteutus jää silti itse luokan vastuulle
abstrakti luokka	Rajapintaa muistuttava C#-kielen rakenne, jossa osa luokan toiminnoista on jätetty määrittelemättä. Luokkaa voidaan käyttää vain, kun luokan perijä toteuttaa luokan abstraktit osat
ketjutettava metodi	Metodi, jota kutsutaan olion kautta ja joka palauttaa kyseisen olion palautusarvona niin, että palautusarvosta on mahdollista kutsua uudelleen samaa tai muuta ketjutettavaa metodia.
event-rakenne	C#-kielen avainsana ja toiminto, jolla luokat voivat viestiä keskenään
callback-metodi	Oliona käsiteltävä metodi, joka voidaan antaa metodiparametrina muun luokan tai metodin kutsuttavaksi

1 JOHDANTO

Vuosikymmenten saatossa ohjelmointitekniikat ovat kehittyneet moneen suuntaan. Vaikka ohjelmointiparadigmoja ja niiden ympärille rakennettuja kieliä on kymmeniä, on Unity3D-pelien kehityksessä vallitseva imperatiivinen ohjelmointitapa monille ohjelmoijille ainoa tuttu tekniikka. Imperatiivisessa ohjelmoinnissa ohjelmoija määrittää rivi riviltä prosessin, jonka tietokone sitten suorittaa vaihe vaiheelta halutun lopputuloksen aikaansaamiseksi (Microsoft 2012).

Imperatiivisen ohjelmoinnin vastineena toimii funktionaalinen ohjelmointi (Microsoft 2012). Funktionaalisessa ohjelmoinnissa ohjelmoija keskittyy tiedon muovaamiseen funktioilla (Microsoft 2012). Ongelmat tiivistyvät siihen, mitä tietoa ongelman ratkaisemiseen tarvitaan ja millä funktioilla tieto voidaan muualta muovata ongelman vaatimaan muotoon (Microsoft 2012). Tieto siis määritellään suhteessa muuhun tietoon, jolloin kokonainen ohjelma voidaan nähdä yhtenä massiivisena funktiona.

Unity3D:n käyttämä C# on pääsääntöisesti imperatiivinen kieli (Microsoft 2012). Se ei kuitenkaan estä funktionaalista ohjelmointia muistuttavien tekniikoiden hyödyntämistä. Reaktiivinen ohjelmointi tuo funktionaalisen tiedon muovaamisen edut imperatiivisten kielien ulottuville. Reaktiivisessa ohjelmoinnissa tietoa mallinnetaan tapahtumina, joita voidaan muovata funktionaalisesti suhteessa muihin tapahtumiin (Lew 2017). Sen sijaan että ohjelmoija määrittäisi todellisia arvoja, ohjelmoija määrittää tapahtumavirtoja, jotka mallintavat kaikille mahdollisille arvoille suoritettavia funktionaalisia operaatiota (Staltz 2016).

Vaikka reaktiivinen ohjelmointi on verkkopalveluiden kehittämisessä suosittu tekniikka (IBM 2017), ei sitä pelialalla juurikaan tunneta. Opinnäytetyön tavoite on selvittää, soveltuuko reaktiivinen ohjelmointi Unity3D-pelikehityksen haasteisiin. Tarkoitus on toteuttaa Red Stage Entertainment Oy -yritykselle Skábma-peliprojektin pilotin keskeiset ohjelmointitarpeet reaktiivista ohjelmointia käyttäen.

Red Stage Entertainment Oy on luovaan tarinankerrontaan erikoistunut mediatuotantoyritys, jolla on kokemusta muunmuassa lyhytelokuva-, teatteri- ja animaatiotuotannosta.

Opinnäytetyöni yhteydessä toteutettava peliprojekti Skábma - Polar Night on yrityksen ensiaskel pelituotantoon (Red Stage Entertainment 2017).

Skábma - Polar Night on kolmannessa persoonassa kuvattu sanattomaan tarinankerrontaan keskittyvä peli, joka kertoo noitarummun löytäneen saamelaislapsi Ailun seikkailusta luonnon henkien kanssa. Pelille keskeisenä teemana on saamelaiskulttuurin ja mytologian kuvaaminen tarkasti ja kunnioittavasti.

Pelaaja ohjaa pelissä Ailua halki pohjoisen metsien, ratkoen ongelmia ja taistellen paho-ja henkiä vastaan noitarummulla loitsien. Projektin tavoite ei ole toteuttaa koko peliä, vaan ainoastaan pelin perusmekaniikat ja muutamia alueita sekä haasteita.

2 Reaktiivinen ohjelmointi

2.1 ReactiveX

Reaktiivinen ohjelmointi on perinteisesti tapahtunut funktionaalisilla ohjelmointikielillä (Gallagher 2016). Reaktiivisia tekniikoita voidaan kuitenkin hyödyntää myös imperatiivisilla kielillä käyttämällä juuri kyseistä tarkoitusta varten suunniteltua ReactiveX-rajapintaa. ReactiveX, tai lyhyesti Rx, määrittelee joukon asynkronisten tietovirtojen mallintamiseen ja muovaamiseen tarkoitettuja tyyppejä ja metodeja. Näiden määritelmien pohjalta on ReactiveX-toteutuksia tehty kymmenille eri kielille. Tämän opinnäytetyön puitteissa käytössä on kuitenkin vain C#:lle laadittuja toteutuksia. Kielestä riippumatta kaikissa toteutuksissa on tiettyjä yhtäläisyyksiä. Observable on poikkeuksetta tietovirtaa kuvaava tyyppi, ja suuri osa operaattoreista on toteutettu kielelle kuin kielelle. Operaattorien nimissä saattaa kuitenkin olla kielikohtaista vaihtelua. Osa toteutuksista myös sisältää lisäoperaattoreita tai karsii vähemmän käytettyjä operaattoreita pois.

2.1.1 Observable

ReactiveX-ohjelmoinnin perusyksikkönä toimii Observable (ReactiveX). Observable käsitteenä on yksinkertaisimmillaan funktio, joka vastaanottaa tietoa jonkinlaisesta tietolähteestä ja tarjoaa ulostulona uudenlaista tietoa (Lesh 2016). Se siis määrittelee jonkinlaisen funktion sisääntulevan tiedon käsittelemiseksi.

ReactiveX:n kontekstissa Observable on objekti, joka määrittelee sisääntulevaa tietoa käsittelevän operaation. Operaatio voi tuottaa paitsi uusia tietotapahtumia, myös poikkeustapahtumia tai päätäntätapahtuman. Päätäntätapahtumalla Observable voi ilmoittaa kuuntelijoilleen, ettei uutta tietoa ole enää tulossa. Vastaavasti poikkeustapahtumilla kuuntelijat saavat tiedon funktiossa tapahtuneista poikkeustilanteista (Campbell 2012). Kuuntelijat voivat kuunnella erikseen ulostulevaa dataa, poikkeustapahtumia sekä päätäntätapahtumia. Näin kuuntelijat voivat toteuttaa monimutkaisiakin toimintoja, joihin perinteinen tapahtumakäsittely ei normaalisti pystyisi. Observablet kuljettavat aina vain yhtä tietotyyppiä. Tästä syystä Observable on määritelty geneerisenä luokkana. Esimerkiksi string-tyyppisiä tapahtumia tuottava Observable määritellään siis Obser-

vable<string> tyypiksi. Hyvien ohjelmointitapojen mukaisesti Observableja käsitellään yleensä IObservable<T>-rajapinnan kautta. Kuuntelijoita varten on olemassa IObserver<T>-rajapinta joka määrittelee käsittelijät tieto-, poikkeus- ja päätäntätapahtumille, mutta käytännössä sitä ei tarvitse lähes koskaan toteuttaa itse (Campbell 2012). Vaikka IObservable<T> määrittelee Subscribe-metodin joka ottaa parametrina IObserver<T> instanssin, kuunnellaan Observablea yleisemmin Subscribe-metodin vaihtoehtoisella versiolla, jossa tieto-, poikkeus- ja päätäntätapahtumien käsittelyfunktiot annetaan Observablelle parametreina (Campbell 2012). Ne määritellään tyypillisesti lambdalausekkeita käyttäen. Tällöin Subscribe-metodi luo sisäisesti Observerin, joka käyttää annettuja funktioita tapahtumien käsittelyyn. Subscriben kutsuminen palauttaa myös IDisposable-rajapintaan käärityn tyypin, joka edustaa kuuntelijan kuuntelusuhdetta Observableen. IDisposablen Dispose-metodia kutsumalla voimme lopettaa Observablen kuuntelun kyseisen kuuntelijan osalta.

2.1.2 Observablen lämpötila

On hyvä huomata, että Observablen määrittämää funktiota ei välttämättä ikinä suoriteta vaikka Observable saisikin sisääntulevia tietueita. Observablen suorittaminen on riippuvainen siitä, että Observablella on myös kuuntelijoita (Sexton 2013). Tämän takia esimerkiksi sivuvaikutuksia aiheuttavat Observablet on suunniteltava tarkkaan niin, etteivät järjestelmän muut osat ole riippuvaisia Observablen sivuvaikutuksista. Yleisesti ottaen sivuvaikutuksia tulisi välttää, mutta aina se ei ole mahdollista (Campbell 2012).

Observable on mahdollista myös muovata muotoon, jossa sisääntulevat tietueet käsitellään ja välitetään eteenpäin riippumatta siitä, onko Observablella kuuntelijoita. Tällöin puhutaan ns. kuumasta Observablesta (Sexton 2013). Kuuma Observable välittää tietotapahtumansa kaikille kuuntelijoille jaetusti. Vastaavasti Observablea joka generoi uudet tietotapahtumat jokaiselle kuuntelijalle erikseen kutsutaan kylmäksi Observableksi (Sexton 2013).

Observablen lämpötilaa voidaan muovata operaattoreilla. Hyvä esimerkki tästä on Publish-operaattori. Esimerkkimme pohjana toimii luova Range-operaattori. Range-operaattori palauttaa normaalisti kylmän Observablen, joka tuottaa tietotapahtumina jokaisen kokonaisluvun annettujen parametrien välillä sekä päätäntätapahtuman. Nämä

tapahtumat tuotetaan kuitenkin jokaiselle kuuntelijalle erikseen, riippumatta siitä milloin kuuntelija alkaa Observablea kuuntelemaan. Syöttämällä Rangen palauttama Observable Publish-operaattorille, saamme Observablen joka tuottaa kaikki arvonsa vain kerran. Jos kuuntelijat eivät ole tuolloin kuuntelemassa Observablea, jäävät tapahtumat kokonaan huomiotta (Campbell 2012). Jotta kuuntelijat olisi mahdollista rekisteröidä kuuntelemaan Observablea ennen tapahtumien alkamista, muuntaa Publish-operaattori Observablen IConnectableObservable-rajapintaan käärittyyn ConnectableObservable-tyyppiin (Sexton 2013). IConnectableObservable määrittää normaalin IObservable-rajapinnan lisäksi Connect-metodin, jota kutsumalla voimme kytkeä Observablen päälle. Täten voimme siis rekisteröidä kuuntelijoita ConnectableObservableen ennen Connect-metodin kutsumista. Näin varmistumme siitä, että kuuntelijat kuulevat kaikki ConnectableObservablen tuottamat tapahtumat. Subscribe-metodin tavoin Connect-metodi palauttaa IDisposablen, jonka Dispose-metodin kutsuminen kytkee ConnectableObservablen jälleen pois päältä (Campbell 2012).

2.1.3 Operaattoreista

Rx:lle keskeinen idea on Observablejen muovaaminen ja yhdistely uusien Observablejen tuottamiseksi. Täysin reaktiivisessa ohjelmassa kaikki tieto leviää halki järjestelmän tietovirtojen halki, eikä järjestelmän osien tarvitse säilöä hetkellistä tilaa. Observablet sallivat tällaisen tietovirtojen koostamisen operaattoreilla. ReactiveX:n X eli eXtensions viittaa juuri Observablea laajentaviin ketjutettaviin metodeihin, joita käyttäen Observableja voi muovata uusiksi Observableiksi (ReactiveX). Näitä operaattoreita voidaan karkeasti luokitella toimintatapojensa perusteella. Luokista yleisimmät ovat luovat, seulovat, muovaavat sekä yhdistävät operaattorit. On myös paljon muita erityisempiin käyttötarkoitukseen laadittuja operaattoreita, joiden luokitteleminen olisi vaikeaa tai mahdotonta.

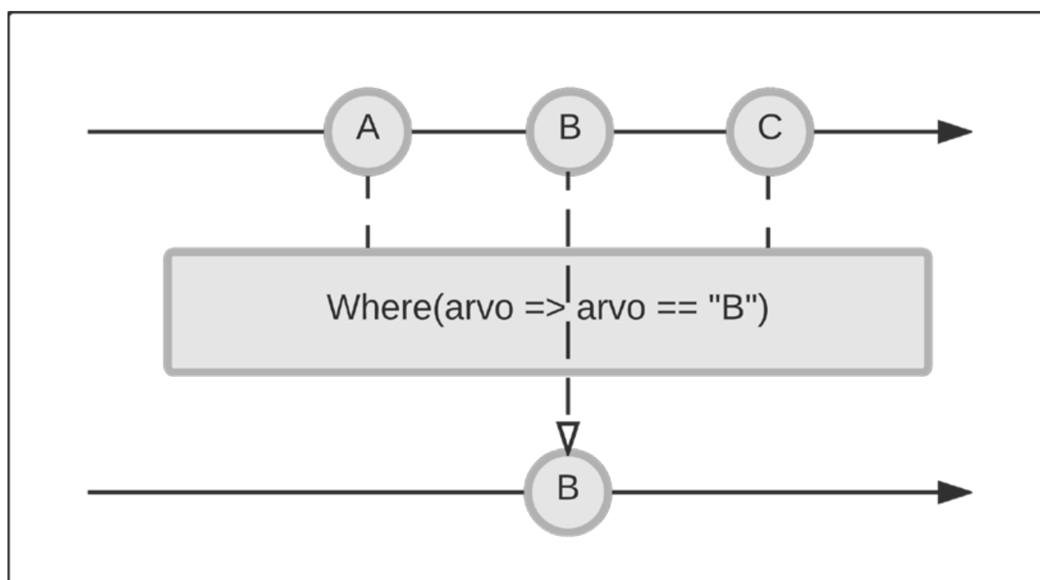
2.1.4 Luovat operaattorit

Luovat operaattorit ovat nimensä mukaisesti operaattoreita, jotka luovat uusia Observableja. Observableja ei C#:ssa koskaan luoda new-avainsanalla, vaan Observablen staattisilla tehdasmetodeilla tai muista tietotyypeistä luovilla laajennusmetodeilla.

(Campbell 2012) Vaikka luovia operaattoreita on paljon, jo muutama yleisimmistä operaattoreista riittää suureen osaan käyttötarpeista. Tämän opinnäytetyön puitteissa luovia operaattoreita ei juurikaan käytetä, sillä suuri osa Unity3D:n sisäisistä toiminnoista voidaan muuntaa suoraan Observable-tapahtumavirroiksi.

2.1.5 Seulovat operaattorit

Seulovien operaattoreiden tehtävä on karsia muovattavan Observablen tapahtumia jonkinlaisten ehtojen perusteella (Campbell 2012). Seulovista operaattoreista ehdottomasti yleisin on Where-operaattori.



Kuvio 1. Where-operaattori

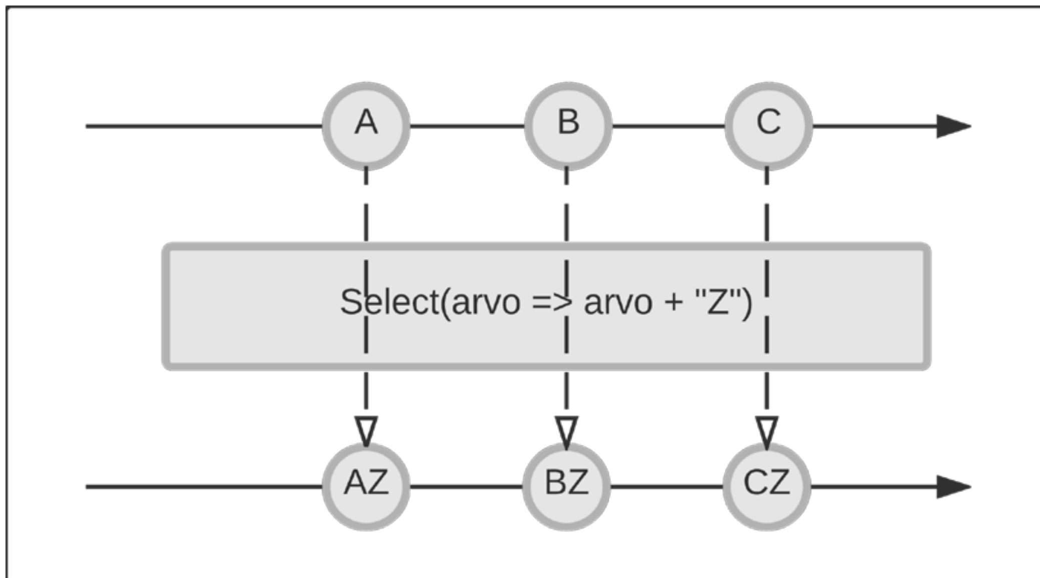
Where-operaattorille annetaan parametrina ehtofunktio, joka suoritetaan jokaiselle tapahtumavirran välittämälle tietotapahtumalle. Mikäli ehto toteutuu, välitetään tapahtuma eteenpäin. Muutoin tapahtuma karsiutuu pois. Ehtofunktion ei tarvitse käsitellä tapahtuman sisältämää tietoa, vaan se voi esimerkiksi hyödyntää ohjelman muuta tilaa. Näin Where-operaattoria hyödyntämällä normaalilla totuusarvomuuttujalla voidaan kontrolloida, välittääkö Observable arvoja kuuntelijoilleen. (Campbell 2012)

Seulovia operaattoreita on monia muitakin, mutta niiden käyttötarkoitukset ovat usein tarkempia ja rajoittuneempia. Jotkin seulovista operaattoreista myös kontrolloivat pää-

täntätapahtuman lähettämistä, kuten esimerkiksi Take-operaattori, joka poimii parametrina annetun määrän tapahtumia virrasta ja päättää sitten Observablen.

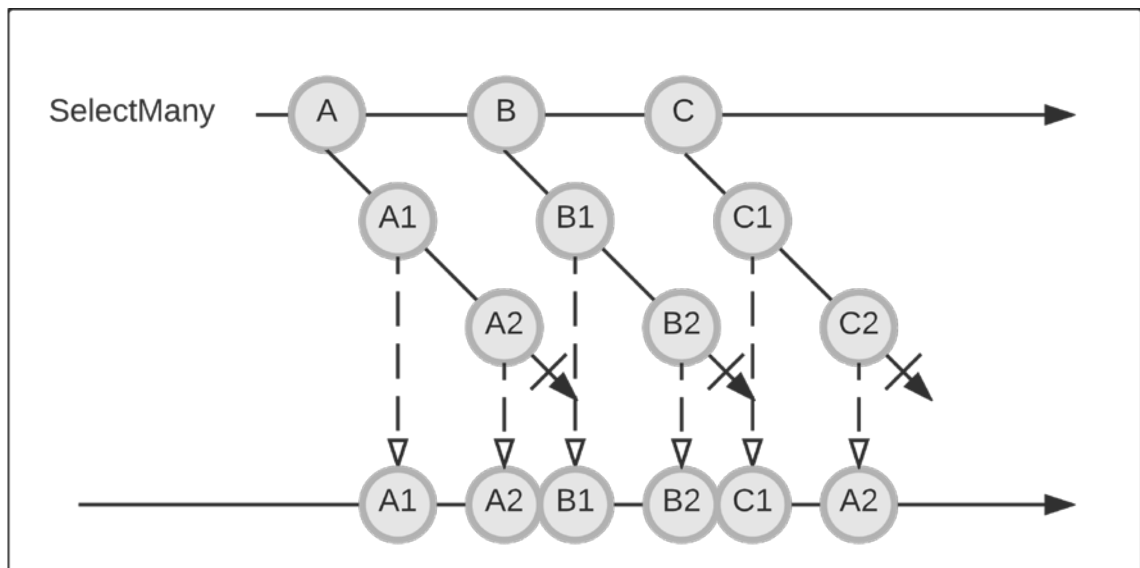
2.1.6 Muovaavat operaattorit

Jotkin operaattorit muuntavat Observablen tuottamia tapahtumia muunlaiseen muotoon. Näitä operaattoreita kutsutaan muovaaviksi operaattoreiksi (ReactiveX). Muovaavista operaattoreista Select ja SelectMany ovat erityismaininnan arvoisia yleiskäyttöisyytensä vuoksi.



Kuvio 2. Select-operaattori

Select-operaattori suorittaa parametrina annetun muunnosfunktion jokaiselle tapahtumavirran välittämälle tietotapahtumalle. (Campbell 2012) Muunnosfunktio voi toteuttaa monimutkaisiakin muunnoksia ja tarpeen tullen esimerkiksi kutsua tapahtumien arvoilla metodeja, välittäen palautetun arvon jälleen eteenpäin kuuntelijoilleen.



Kuvio 3. SelectMany-operaattori

Joskus Observablen tapahtumien on kätevää sallia suorittaa omia tapahtumavirtojaan, jotka saattavat palauttaa useita tapahtumia. SelectMany-operaattori on luotu juuri tällaisia tilanteita varten. SelectManyn avulla sisäkkäiset Observable-tapahtumavirrat voi litistää yhdeksi tapahtumavirraksi. (Campbell 2012) Kuvion 3 esimerkissä jokaisella tapahtumalla aloitetaan uusi virta, joka tuottaa kaksi uutta juuritapahtumaan perustuvaa tapahtumaa.

2.1.7 Yhdistävät operaattorit

Yhdistävät operaattorit ottavat useamman Observablen, ja tuottavat niistä yhden Observablen (ReactiveX). Vaikka yhdistäviä operaattoreita on monia, ei tämän projektin puitteissa ole käytetty muita kuin Merge-operaattoria. Merge-operaattoria tarkastelen lähemmin 3.3 luvussa.

2.1.8 Muista operaattoreista

ReactiveX sisältää kaikki operaattorien variaatiot mukaanlukien satoja operaattoreita. Kaikkien operaattoreiden läpikäyminen tämän opinnäytetyön puitteissa on täysin mahdotonta. Operaattoreita kannattaakin lähestyä työkalupakkina, josta voi ongelmakohtaisesti etsiä tarpeeseen soveltuvat työkalut. Myöhemmissä luvuissa esittelen tarpeen tul-

len ongelmakohtaisesti uusia operaattoreita. ReactiveX:n operaattorien lisäksi Rx.Net-sisältää joukon omia operaattorilisäyksiään ja seuraavassa luvussa käsiteltävä UniRx tuo mukanaan yhä lisää Unity3D-tarpeisiin keskittyviä operaattoreita.

2.2 UniRx

Unity3D:n käyttämä Monon versio ei sisällä Observableja tai muitakaan ReactiveX:n osia, eikä .Net-käyttöön normaalisti tarkoitettu Rx.Net toimi ongelmitta Unity3D-projekteissa.. Suuri osa kirjastosta on jouduttu uudelleentoteuttamaan Unity3D-käyttöä varten. Japanilaisen Grani, Inc. yrityksen Yoshifumi Kawai on toteuttanut avoimen lähdekoodin projektina UniRx-kirjaston juuri tätä tarkoitusta varten. UniRx tarjoaa suuren osan Rx-toiminnoista Unity3D:n Mono-version puitteissa toteutettuna. Lisäksi UniRx sisältää joukon erityisesti Unity3D-käyttöön suunnattuja operaattoreita ja luokkia (Kawai 2016).

Eräs tärkeimmistä UniRx:n tarjoamista toiminnoista on Unity3D:n valmiit toiminnot Observableiksi muuntavat laajennosmetodit. Näiden avulla Unity3D:n komponenttien tarjoamia toimintoja voi käsitellä tietovirtoina kuten mitä tahansa muutakin tietolähdettä ReactiveX-ohjelmoinnissa. Tämä tekee Unity3D:n kanssa työskentelystä Rx-tekniikoin saumatonta, sekä tekee koodista huomattavasti luettavampaa ja ytimekkäämpää.

UniRx helpottaa myös Observablejen sekä kuuntelijoiden elinkaarien hallintaa sallimalla käyttäjän sitoa Observablen päätännän Unity3D-peliobjektin elinkaareen (Kawai 2016). Kun peliobjekti tuhotaan, voidaan objektin komponentissa elävä Observable päättää automaattisesti. Vastaavasti myös kuuntelijoiden elinkaaret voidaan sitoa peliobjekteihin. Näitä tekniikoita käyttämällä säästyy paljon päänvaivaa silloin, kun kuuntelua ja Observableja hyödyntäviä peliobjekteja lisätään ja poistetaan pelin ajon aikana dynaamisesti.

3 Rx Unity3D-projektissa

3.1 Peruskäyttö

UniRx:ää käyttäessä ensimmäinen suuri muutos on Unity3D:n tyypillisten metodien poistuminen käytöstä lähes kokonaan. Koska reaktiivisessa koodissa tapahtumiin pohjautuvat reaktiot julistetaan deklaratiivisesti, ei reaktiivisesti toteutetuissa Unity3D-komponenteissa käytetä juurikaan komponenttien valmiiksi tarjoamia tapahtumametodeja initialisaatiometodeja lukuun ottamatta. Awake-, Start-, OnEnable- ja OnDisable-metodit riittävät lähes kaikkiin tarpeisiin, sillä kaikki muut Unity3D-komponenttimetodit toimivat pelkkinä tapahtumavirtalähteinä käyttäjän omalle logiikalle.

Update-metodi on perinteisesti Unity3D-komponenttien ydin. Reaktiivisilla tekniikoilla sekin mallinnetaan Observablena, joka lähettää tapahtuman jokaisen ruudunpäivityksen yhteydessä. UniRx:n tarjoamat Unity3D-sidonnaiset komponenttien laajennusmetodit tuottavat useimmat Unity3D-tapahtumien seuraamiseen tarvittavat Observablet ohjelmoijan puolesta, jolloin ohjelmoijan tehtäväksi jää vain muovata kyseiset Observablet vastaamaan toteutettavana olevan toiminnallisuuden määrittelyä.

```
public class Destroyer : Triggerable
{

        public float DestroyAfterMilliseconds;
        public bool TriggerOnAwake;

        private bool _destroyed;

        private void Awake ()
        {
                if (TriggerOnAwake) Trigger();
        }

        public override void Trigger()
        {
                OnTriggered();
        }

        private void OnTriggered()
        {
                Observable.Timer(TimeSpan.FromMilliseconds(DestroyAfterMilliseconds))
                        .Subscribe(_ => OnDestroy())
                        .AddTo(this);
        }

        private void OnDestroy()
        {
                if (_destroyed) return;
                Destroy(this.gameObject);
                _destroyed = true;
        }
}
```

Koodiesimerkki 1. Destroyer-luokka

Otetaan esimerkkinä Destroyer-luokka. Luokan tarkoitus on tietyn millisekuntimäärän (DestroyAfterMilliseconds) jälkeen tuhota objekti, johon Destroyer-komponentti on kiinnitetty (this.gameObject). Jätämme Destroyerin aktivoimisen muiden luokkien vastuulle abstraktin Triggerable-luokan kautta. Kaikki Triggerablet toteuttavat Trigger-metodin. Aktivoituessaan (OnTriggered) luokka luo uuden Observablen luovalla Timer-operaattorilla. Palautettu Observable tuottaa määritetyn millisekuntimäärän jälkeen yhden tietotapahtuman ja päättää itsensä. Tietotapahtumalle määritetään Subscribe-metodilla kuuntelija, joka kutsuu OnDestroy-metodia. AddTo varmistaa, että kuuntelu

lopetetaan. Koko luokan edustaman tapahtuman toiminnallisuus tiivistyy näin kolmeen selkeästi luettavaan riviin.

```
public class TraditionalDestroyer : Triggerable
{
    public float DestroyAfterMilliseconds;
    public bool TriggerOnAwake;

    private bool _destroyed;

    private float _elapsedTime;
    private bool _isTiming;

    private void Awake()
    {
        if (TriggerOnAwake) Trigger();
    }

    public override void Trigger()
    {
        OnTriggered();
    }

    private void OnTriggered()
    {
        _elapsedTime = 0;
        _isTiming = true;
    }

    private void Update()
    {
        if (_isTiming) _elapsedTime += Time.deltaTime;
        if (_elapsedTime >= DestroyAfterMilliseconds) OnDestroy();
    }

    private void OnDestroy()
    {
        if (_destroyed) return;
        Destroy(this.gameObject);
        _destroyed = true;
    }
}
```

Koodiesimerkki 2. Destroyer-luokka toteutettuna perinteisesti

Vastaava toiminto toteutettaisiin tyypillisesti laskemalla kulunutta aikaa Update-metodilla, kuten koodiesimerkissä 2. Update-metodissa toteutettuna käyttäjä joutuu itse toteuttamaan kuluneen ajan seurannan, jolloin luokan luettavuus kärsii. Luokan tehtävä on tuhota objekteja, joten ajoitukseen liittyvät yksityiset muuttujat johtavat lukijaa harhaan. Muuttujiin pääsee käsiksi myös tarpeettomasti koko luokka, tehden näin luokan koodista virheherkempää. Luettavuuden ja koodirakenteen tilan eristämisen parantamiseksi ajan seurannan voisi toteuttaa omassa luokassaan, joka ilmoittaisi esimerkiksi C#:n event-rakenteella ajan päättymisestä. Silloin ratkaisu vastaisi lähemmin toteutettua reaktiivista ratkaisua, kuitenkin ilman reaktiivisten tietovirtojen muovattavuutta ja yhdisteltävyyttä. Ajastin toisi silti luokkaan tarpeetonta tilaa, jonka Observablet eristävät itse tapahtumavirran sisäiseen toimintaan.

Unity tarjoaa myös Coroutine-toiminnon pidempiaikaisten operaatioiden ajamiseen. Coroutine-ratkaisu vastaisi toteutetulta logiikaltaan Update-metodissa ajettua, mutta toisi mukanaan monia rajoituksia. Coroutine ei voi esimerkiksi palauttaa arvoja Observablejen tietotapahtumien tapaan, vaan joutuisi joko käyttämään callback-metodeita tai kutsumaan perinteisiä event-tapahtumia.

Update- tai coroutine-ratkaisut eivät ole merkittävästi reaktiivista ratkaisua monimutkaisempia, koska kyseessä oli hyvin yksinkertainen toiminto. Ero koodin ymmärrettävyydessä ja monimutkaisuudessa korostuu kuitenkin jos luokan toimintavaatimuksia lisätään vähänkin. Mikäli esimerkiksi vaatisimme, että tuhottavan objektin koskettaminen kursorilla ennen ajastimen päättymistä peruu ajastimen, joutuisimme lisäämään reaktiiviseen ratkaisuun vain yhden TakeUntil-operaattorin. Vastaava lisäys Update- tai coroutine-ratkaisuihin vaatisi OnMouseEnter-metodin toteuttamista sekä jonkinlaisen kommunikaatiomuuttujan lisäämistä. Ratkaisun luettavuus heikkenisi jokaisesta vastaavasta lisäyksestä.

```
public class CollisionTriggerer : MonoBehaviour
{
	public List<Triggerable> Triggerables;
	public string TriggeringTag;

	public GameObject SpawnOnImpact;

	private bool _triggered;

	private void Awake()
	{
		this.OnTriggerEnterAsObservable()
			.Where(_ => !_triggered)
			.Where(other => other.CompareTag(TriggeringTag))
			.Subscribe(other => OnTriggered(other))
			.AddTo(this);
	}

	private void OnTriggered(Collider other)
	{
		_triggered = true;

		if(SpawnOnImpact != null)
			Instantiate(SpawnOnImpact, other.transform.position, other.transform.rotation);

		foreach (var triggerable in Triggerables)
		{
			triggerable.Trigger();
		}
	}
}
```

Koodiesimerkki 3. CollisionTriggerer-luokka

Destroyer-luokan Trigger-metodia voi kutsua Triggerable-abstraktion kautta esimerkiksi CollisionTriggerer. CollisionTriggerer-komponentin tehtävä on kutsua jokaisen komponenttiin kiinnitetyn Triggerable-komponentin Trigger-metodia silloin, kun CollisionTriggerer-komponentin sisältävä objekti koskettaa tietyn Unity3D-tagin sisältävää objektia, tosin vain silloin jos toimintoa ei ole vielä aiemmin aktivoitu.

Määritämme jälleen Awake-metodissa deklaratiivisesti komponentin toiminnot. Koska tahdomme jonkin asian tapahtuvan silloin kun objekti koskettaa toista objektia, otamme muovaamisen lähtökohdaksi OnTriggerEnterAsObservable-metodin palauttamaa Observablea. OnTriggerEnterAsObservable-metodin palauttama Observable tuottaa tietotapahtumina kosketettujen objektien Collider-komponentteja. Emme ole kiinnostuneita kaikista kosketuksista, joten jatkamme seulomalla tapahtumavirtaa. Tapahtumien seulomisen toteutamme Where-operaattorilla. Ensin seulomme välitetystä tiedosta riippumatta kaikki tietueet, joiden vastaanottohetkellä CollisionTriggerer on jo laukaistu. Sen

jälkeen seulomme välitetyistä Collider-komponenteista vain ne, joiden Unity3D-tagi vastaa TriggeringTag-muuttujalla käyttäjän määrittämää tagia. Vasta seulomisen jälkeen määritämme Subscribe-operaattorilla Triggerablejen laukaisulogiikan tapahtuvaksi. Koko toiminto tiivistyi jälleen viiteen selkeästi luettavaan koodiriviin.

```
public class TraditionalCollisionTriggerer : MonoBehaviour
{
    public List<Triggerable> Triggerables;
    public string TriggeringTag;

    public GameObject SpawnOnImpact;

    private bool _triggered;

    private void OnTriggerEnter(Collider other)
    {
        if (!_triggered)
        {
            if (!other.CompareTag(TriggeringTag))
                OnTriggered(other);
        }
    }

    private void OnTriggered(Collider other)
    {
        _triggered = true;

        if (SpawnOnImpact != null)
            Instantiate(SpawnOnImpact, other.transform.position, other.transform.rotation);

        foreach (var triggerable in Triggerables)
        {
            triggerable.Trigger();
        }
    }
}
```

Koodiesimerkki 4. CollisionTriggerer toteutettuna perinteisesti

Kuten koodiesimerkistä 4 huomaa, perinteinen toteutus ei eroa juurikaan reaktiivisesta toteutuksesta tämäntyyppisessä ongelmassa. Näin yksinkertaisessa tilanteessa perinteinen toteutus on mielestäni jopa selkeämpi kuin reaktiivinen toteutus. Reaktiivista toteutusta on tällaisessa tilanteessa kuitenkin helpompi skaalata luokan tarpeiden kasvaessa. Ohjelmoijan on siis hyvä valita tekniikkansa luokan käyttötarkoituksen ja jatkokehityssuunnitelmien mukaan.

3.2 Komponenttien viestintä

Edellisissä koodiesimerkeissä tarkastelimme komponentteja, joissa Observableilla mallinnettiin vain sisäisiä interaktioita. Projektissa tuli usein vastaan tilanteita joissa luokkien vastuualueet jakautuivat tapahtumavirtojen tuottajiin ja kuluttajiin. Joissain tapauksissa kokonaiset luokat myös muovasivat sisääntulevia virtojaan uusiksi virroiksi tarjotakseen niitä edelleen ulospäin muille komponenteille.

Pelaajan hahmo koostuu useasta komponentista, joiden välinen kommunikaatio tapahtuu juuri reaktiivisesti tietovirroilla. Kompositiosta vastaa yksittäinen pohjakomponentti, joka liittää muiden komponenttien Observablet yhteen. Kokeilin projektissa muutamaa erilaista tapaa viestiä Observablejen avulla.

```
Input.MovementVector
        .Subscribe(vec => Movement.SetMovementInputVector(vec))
        .AddTo(this);
Input.AimVector
        .Subscribe(vec => Movement.SetAimVector(vec))
        .AddTo(this);

Animation.RegisterDrumStream(Input.DrumStream);
Animation.RegisterMovementStream(Input.MovementInput);
Animation.RegisterAltFireStream(Input.AltFireStream);

Health.CurrentHealth
        .SubscribeToText(HealthMeter)
        .AddTo(this);
Health.CanTakeDamage
        .SubscribeToText(InvulnerabilityIndicator)
        .AddTo(this);
```

Koodiesimerkki 5. Tapoja kytkeä Observableja yhteen

SubscribeToText-metodi on UniRx:n tarjoama helppokäyttötoiminto, jolla Observablen voi liittää suoraan Unity3D:n Text-komponenttiin. Text-komponentti on tekstin piirtämiseen käytetty käyttöliittymäelementti. SubscribeToTextiä käyttämällä Observablen tuottamat tapahtumat sidotaan suoraan ruudulla näkyvään tekstikenttään. Toiminto tekee käyttöliittymäkoodista selkeämpää ja vaivattomampaa. Koodiesimerkeissä 6 ja 7

vastaava toiminnallisuus on toteutettu perinteiseen tapaan event-rakenteella, jolloin ohjelmoijan on itse huolehdittava tekstin päivittämisestä ja tapahtumien kuuntelun päättämisestä komponentin tuhoutuessa.

```
Health.CurrentHealthEvent += UpdateHealthDisplay;
Health.CanTakeDamageEvent += UpdateCanTakeDamageDisplay;
```

Koodiesimerkki 6. Event-rakenteen kuuntelusuhteen kytkeminen

```
public void UpdateHealthDisplay(int newHealth)
{
        HealthMeter.text = newHealth.ToString();
}

public void UpdateCanTakeDamageDisplay(bool newState)
{
        InvulnerabilityIndicator.text = newState.ToString();
}

private void OnDestroy()
{
        Health.CurrentHealthEvent -= UpdateHealthDisplay;
        Health.CanTakeDamageEvent -= UpdateCanTakeDamageDisplay;
}
```

Koodiesimerkki 7. Kuunteluun reagoiminen ja kuuntelun lopettaminen

Kahden muun koodiesimerkissä 5 käytetyn kytkentätavan ero on hienovaraisempi. Ensimmäisessä versiossa Input-luokan tarjoamia Observable-tapahtumavirtoja kuunnellaan jo välikätenä toimivassa kytkentäkomponentissa ja Observablen tuottamilla arvoilla kutsutaan muiden komponenttien metodeja. Siirrymme siis asynkronoidusta datavirrasta synkronoituun koodiin. Tämä menetelmä voi olla hyödyllinen esimerkiksi silloin, jos välitämme tietoa ei-reaktiiviselle rajapinnalle. Kuuntelijasuhde jää myös itse kytkentäkomponentin vastuulle, jolloin myös kuuntelun lopettaminen on kyseisen komponentin tehtävä. Koodiesimerkeissä 6 ja 7 perinteisin menetelmin luotu kuuntelusuhde on suora vastine tämän kaltaiselle kytkennälle.

Useimmissa tapauksissa kuitenkin toinen koodiesimerkissä 5 tehty kytkentämalli on järkevämpi. Esimerkissä annamme Input-luokan tarjoamat Observablet Animation-luokalle suoraan, jolloin Animation-luokka voi jatkaa edelleen tapahtumavirran muo-

vaamista omiin tarpeisiinsa. Myös kuuntelu jää silloin animaatiokomponentin vastuulle, selkeyttäen komponenttien vastuunjakoa. Joissain tapauksissa voisi olla järkevää jättää kuuntelun hallinta yhä kytkentäkomponentin vastuulle. Koska kuuntelusuhdetta kuvastaa IDisposable-rajapinta, voisimme palauttaa kuuntelusuhteen palautusarvona kytkentäkomponentin hallittavaksi. Skábma-projektin puitteissa sille ei kuitenkaan ollut tarvetta. Koska C#:n event-rakennetta ei voi vastaavalla tavalla välittää parametrina luokalta toiselle, vaatisi toiminnallisuuden toteuttaminen perinteisellä tavalla event-rakenteen käärimistä parametrina käytettävään luokkaan. Tämänkaltainen luokka vastaisikin käytännössä kaikesta lisätoiminnallisuudestaan karsittua Observablea.

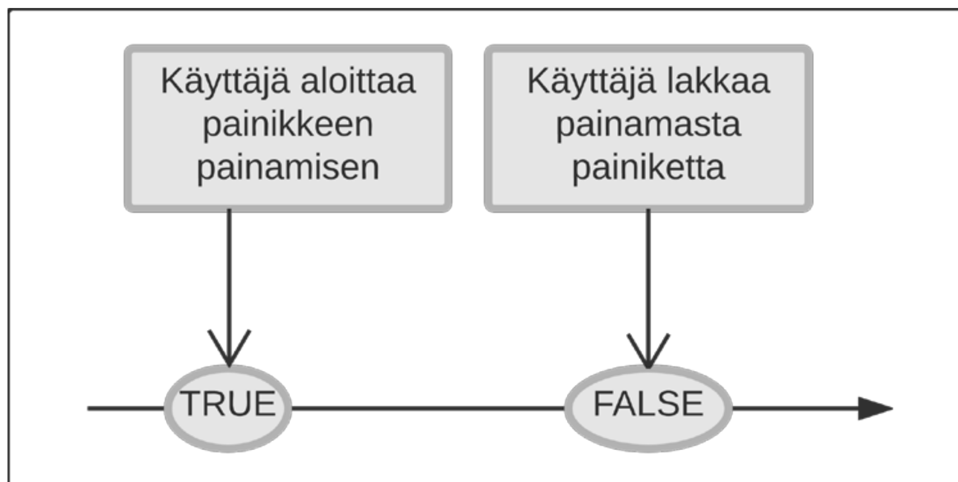
3.3 Syötekäsittely

Käyttäjän syöte koostuu monista eri toiminnoista. Pelaaja voi liikkua kahdessa ulottuvuudessa, tähdätä kahdessa ulottuvuudessa, valita kahden eri loitsimistilan välillä sekä loitsia rumpua rummuttamalla. Näiden Observablejen tuottaminen on jaettu omiin metodeihinsa, joita kutsutaan Awake-metodissa koodiesimerkin 8 mukaisesti.

```
MovementVector = ConstructMovementVectorObservable();

AimVector = ConstructAimVectorObservable();

DrumStream = ConstructDrumStreamObservable();

AltFireStream = ConstructAltFireObservable();
```

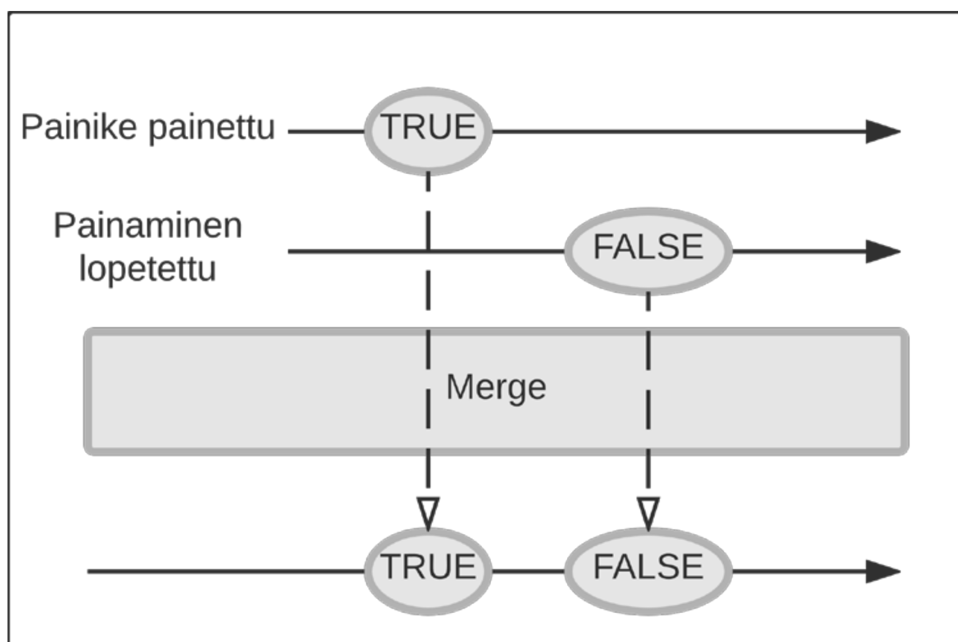
Koodiesimerkki 8. Input-luokan tapahtumavirtojen initialisaatio

Jokainen neljästä Observablesta sisälsi omat haasteensa. Loitsutilasyötettä kuvaava AltFireStream oli syötekäsittelyvirroista yksinkertaisin. Aloitamme muodostamalla mielikuvan halutusta Observablesta.



Kuvio 4. AltFireStream-virran toivottu rakenne

Unity3D:ssä syöte luetaan tyypillisesti ruudunpäivityksen yhteydessä Update-metodia käyttäen. Käytämme siis sitä pohjana ja seulomme sen mukaan, onko käyttäjä painanut painikkeen pohjaan tai lopettanut painikkeen painamisen. Lähestymme ongelmaa kahtena erillisenä tapauksena, ja yhdistämme sitten molemmat Observablet halutun lopputuloksen tuottamiseksi.



Kuvio 5. Merge-operaattorin käyttö AltFireStream-virrassa

Näin toteutettuna ongelmaksi jää ainoastaan näppäinpainallusvirtojen muuntaminen totuusarvoiksi. Se onnistuu yksinkertaisesti Where- ja Select-operaattoreita käyttäen. Where-operaattorilla seulomme päivitystapahtumat joilla näppäimen tila muuttuu, Select-operaattorilla muunnamme kyseiset päivitystapahtumat totuusarvotapahtumiksi.

Lopulliseksi Observablen tyypiksi tulee täten IObservable<bool>, eli totuusarvojen tapahtumavirta.

```
private UniRx.IObservable<bool> ConstructAltFireObservable()
{
    return
        Observable.Merge(
                this.UpdateAsObservable()
                .Where(_ => Input.GetKeyDown(AltFireKey))
                .Select(_ => true),
                this.UpdateAsObservable()
                .Where(_ => Input.GetKeyUp(AltFireKey))
                .Select(_ => false))
        .DistinctUntilChanged();
}
```

Koodiesimerkki 9. AltFireStream-virran luomismetodi

Lopussa käytetty DistinctUntilChanged-operaattori varmistaa, että tapahtuma tuotetaan vain jos tuotettu arvo on eri kuin edellinen tuotettu arvo. Sillä siis varmistetaan, että tapahtuman kuuntelijat reagoivat vain arvon muuttuessa. Näin säästymme tarpeettomilta reaktioilta arvoihin, jotka eivät todellisuudessa muuta pelin tilaa millään tavoin.

Samoilla tekniikoilla voimme tuottaa myös liikkumista ja tähtäämistä kuvaavat tietovirrat. Vaikka molemmissa tapahtumien tuottamiseen vaadittu logiikka on monimutkaisempaa, ei itse Observablejen tuottaminen poikkea aiemmista esimerkeistä lainkaan.

```
private ReadOnlyReactiveProperty<Vector2> ConstructAimVectorObservable()
{
    return this.UpdateAsObservable()
        .Select(_ => _inputsEnabled
            ? GetAimAngle()
            : new Vector2())
        .DistinctUntilChanged()
        .ToReadOnlyReactiveProperty();
}

private ReadOnlyReactiveProperty<Vector2> ConstructMovementVectorObservable()
{
    return this.UpdateAsObservable()
        .Select(_ =>
        {
            var unscaledVec = new Vector2(Input.GetAxisRaw("Horizontal"), Input.GetAxisRaw("Vertical"));
            var scale = Mathf.Clamp01(unscaledVec.magnitude);

            return _inputsEnabled
                ? unscaledVec.normalized * scale
                : new Vector2();
        })
        .ToSequentialReadOnlyReactiveProperty();
}
```

Koodiesimerkki 10. Tähtäys- ja liikkumisvirtojen luomismetodit

Kuten koodiesimerkistä 10 näkee, kaikki liike- tai tähtäysvektorien tuottamiseen tarvittava logiikka voidaan asettaa Select-operaattorin sisään, jolloin jokainen UpdateAsObservablen tuottama tapahtuma muunnetaan kyseisen ruudunpäivityksen vektorimuotoiseksi syötteeksi. Muunnan molemmat IObservable<Vector2>-virrat lopuksi ReactiveProperty-rakenteiksi. ReactiveProperty on UniRx:n tarjoama tyyppi, joka säilyttää viimeisimmän arvonsa luettavana nykyisenä arvonaan. Se on siis Observable, jolla on käsitys nykytilasta. Rakenne on kätevä silloin, jos tapahtumavirran arvoja tahdotaan lukea muutoinkin kuin virtaa kuuntelemalla. Normaalisti ReactiveProperty sallii myös arvonsa muuttamisen, jolloin taustalla oleva Observable tuottaa lisätapahtumia. ReadOnlyReactivePropertyä käyttämällä tämä toiminnallisuus voidaan estää. ReadOnlyReactiveProperty lähettää tapahtuman vain jos arvo muuttuu, DistinctUntilChanged-operaattorin tavoin. Liikkumisvektoria tuotettaessa käytetty ToSequentialReadOnlyReactiveProperty-operaattori poistaa kyseisen toiminnallisuuden käytöstä. Tapahtuma siis tuotetaan myös arvon päivittyessä niin, ettei arvoon tulee muutosta.

```
private Vector2 _movementVectorField;
public event Action<Vector2> MovementVectorEvent;
public Vector2 MovementVectorProperty
{
    get { return _movementVectorField; }

    private set
    {
        if (MovementVectorEvent != null) MovementVectorEvent.Invoke(value);
        _movementVectorField = value;
    }
}

private Vector2 _aimVectorField;
private Vector2 _previousAimVector;
public event Action<Vector2> AimVectorEvent;

public Vector2 AimVectorProperty
{
    get { return _aimVectorField; }

    private set
    {
        if (value != _previousAimVector)
        {
            _previousAimVector = _aimVectorField;
            _aimVectorField = value;
            if (AimVectorEvent != null) AimVectorEvent.Invoke(value);
        }
    }
}
```

Koodiesimerkki 11. Liikkumisen ja tähtäyksen property-rakenteet

```
private void UpdateMovement()
{
    var unscaledVec = new Vector2(Input.GetAxisRaw("Horizontal"), Input.GetAxisRaw("Vertical"));
    var scale = Mathf.Clamp01(unscaledVec.magnitude);

    MovementVectorProperty = _inputsEnabled
        ? unscaledVec.normalized * scale
        : new Vector2();
}

private void UpdateAim()
{
    AimVectorProperty = _inputsEnabled
        ? GetAimAngle()
        : new Vector2();
}
```

Koodiesimerkki 12. Liikkumisen ja tähtäyksen päivitysmetodit

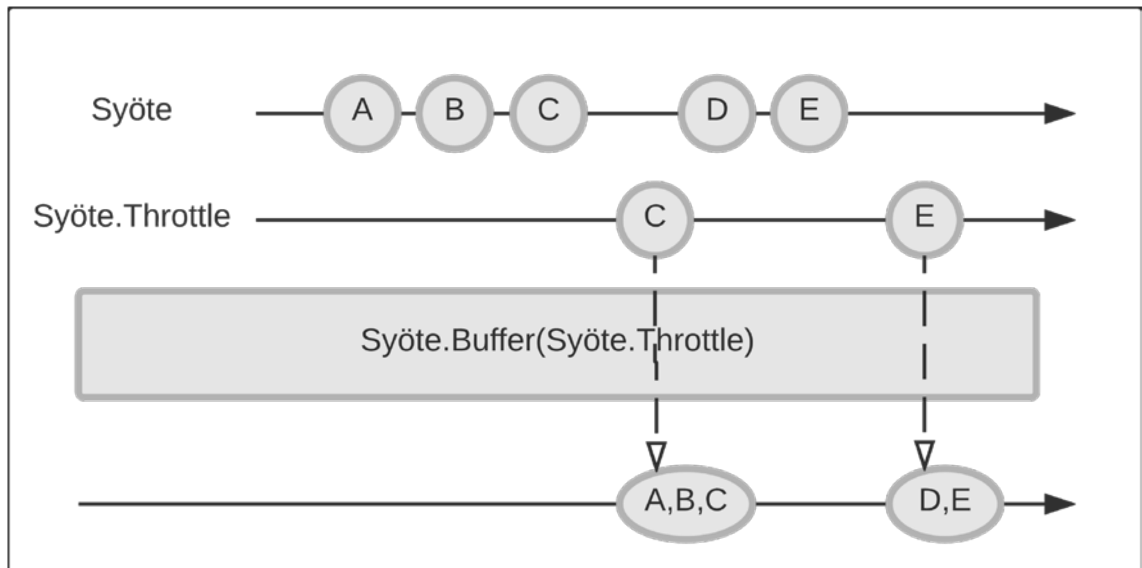
ReactivePropertyt vastaavat nimensä mukaisesti perinteistä C#:n property-rakennetta. Toiminnallisuuden voi toteuttaa manuaalisesti koodiesimerkin 11 tapaan. DistinctUntilChanged-operaattoria vastaavan toiminnallisuuden toteuttaminen vaatii esimerkissä nähtävän AimVectorPropertyn tapaan edellisen arvon säilyttämiseen käytettävää lisämuuttujaa. Arvojen päivittämiseksi syötteen tilaa on kysyttävä Update-metodissa esimerkiksi kutsumalla koodiesimerkin 12 mukaisia metodeja.

Skábma-projektissa pelaajalla on kaksi rumpupainiketta, vasen ja oikea. Syötteet muunnetaan loitsutilasyötettä mallintavan AltFireStreamin tavoin enumeraatioarvoiksi DrumType.Left ja DrumType.Right. Näiden syötteiden tuottamaa väliaikaista Observablea kutsutaan koodiesimerkeissä nimellä unfilteredDrumStream. Molempia painikkeita samanaikaisesti painettaessa tuotetaan kolmas arvo, DrumType.Both. Haastavaksi ongelman tekee se, ettei käyttäjän voida odottaa painavan molempia näppäimiä täsmälleen samalla hetkellä, vaan käyttäjälle on annettava joidenkin millisekuntien verran joustovaraa syötteessä. Joustovaran määrittely on pelisuunnittelijoiden vastuulla, joten millisekuntiaikaa edustaa koodissa julkinen SimultaneousDrumInputBufferMs-kenttä. Mikäli Left- ja Right-syötteiden välissä on alle SimultaneousDrumInputBufferMs:n arvon verran aikaa, lasketaan ne yhdeksi DrumType.Both syötteeksi.

Ratkaisu ongelmaan löytyy ReactiveX:n aikaa käsittelevistä operaattoreista. Buffer-operaattori jakaa tapahtumavirran tapahtumajoukkoihin jonkinlaisen aikamääreen perusteella. Buffer-operaattorilla voisi esimerkiksi ryhmitellä sekunnin välein kaikki viimei-

sen sekunnin aikana tulleet tapahtumat yhdeksi tapahtumajoukoksi. Se siis muovaa IObservable<T>-virran IObservable<IList<T>>-virraksi, eli virraksi, jonka tapahtumat ovat listoja tapahtumista. Rumpusyöteongelman ratkaisemiseksi tulisi ryhmittelyikkunan määrittyä sen mukaan, onko ryhmiteltävä tapahtumavirta tuottanut uusia tapahtumia SimultaneousDrumInputBufferMs-kentän määrittämän ajan sisällä.

Eräs Buffer-operaattorin versio niputtaa tapahtumia ryhmiin parametrina annetun toisen Observablen tuottaessa arvoja. Annamme Buffer-operaattorille parametrina saman syötevirran, mutta Throttle-operaattorin läpi muunnettuna. Throttle-operaattori lykkää tapahtumien tuottamista, kunnes parametrina annettu aikaväli on kulunut ilman uusia tapahtumia. Yhdistämällä Throttle-operaattorin Buffer-operaattoriin, voimme siis ryhmitellä kaikki tapahtumat, jotka tapahtuvat alle määritetyn ajan sisällä toisistaan.



Kuvio 6. Buffer-operaattori Throttle-operaattorilla ryhmitellen

Näin käsiteltynä tuotettu syöte voidaan muuntaa lopulliseen muotoonsa Select-operaattorilla. Mikäli Buffer-operaattorin tuottamassa listassa on sekä Left- että Right-syötteet, tuotamme lopulliseen virtaan DrumType.Both-arvon. Lopullinen koodi on nähtävissä koodiesimerkissä 13.

```
private UniRx.IObservable<DrumEvent> ConstructDrumStreamObservable()
{
    var unfilteredDrumStream =
        Observable.Merge(
            this.UpdateAsObservable()
                .Where(_ => _inputsEnabled)
                .Where(_ => Input.GetButtonDown("Fire1"))
                .Select(_ => new DrumEvent()
                {
                    Type = DrumType.Left,
                    AltFire = Input.GetKey(AltFireKey)
                }),
            this.UpdateAsObservable()
                .Where(_ => _inputsEnabled)
                .Where(_ => Input.GetButtonDown("Fire2"))
                .Select(_ => new DrumEvent()
                {
                    Type = DrumType.Right,
                    AltFire = Input.GetKey(AltFireKey)
                }));

    return unfilteredDrumStream
        .Buffer(
            unfilteredDrumStream
                .Throttle(TimeSpan.FromMilliseconds(SimultaneousDrumInputBufferMs)))
        .Select(inputs => ProcessGroupedDrumInput(inputs));
}
```

Koodiesimerkki 13. DrumStream-virran luominen

Reaktiiviset tekniikat ovat omiaan tämänkaltaisiin tilanteisiin, sillä perinteisin tekniikoin toteutettuna ohjelmoijan täytyy manuaalisesti tallettaa syötteet välimuistiin, tarkkailla edellisestä syötehetkestä kulunutta aikaa ja hallinnoida välimuistiin tallentamisen tilaa. Koodiesimerkissä 14 vastaava syötekäsittely on toteutettu perinteiseen tapaan.

```
private List<DrumEvent> _inputBuffer = new List<DrumEvent>();
private float _elapsedTimeWithoutInput;
private bool _buffering;

private void UpdateDrumming()
{
    if (_inputsEnabled)
    {
        if (Input.GetButtonDown("Fire1"))
        {
            _elapsedTimeWithoutInput = 0;
            _buffering = true;
            _inputBuffer.Add( new DrumEvent()
            {
                Type = DrumType.Left,
                AltFire = Input.GetKey(AltFireKey)
            });
        }

        if (Input.GetButtonDown("Fire2"))
        {
            _elapsedTimeWithoutInput = 0;
            _buffering = true;
            _inputBuffer.Add( new DrumEvent()
            {
                Type = DrumType.Right,
                AltFire = Input.GetKey(AltFireKey)
            });
        }

        _elapsedTimeWithoutInput += Time.deltaTime;
        if (_buffering && _elapsedTimeWithoutInput >= SimultaneousDrumInputBufferMs)
        {
            _buffering = false;
            if (DrumInputEvent != null)
                DrumInputEvent.Invoke(ProcessGroupedDrumInput(_inputBuffer));
            _inputBuffer.Clear();
        }
    }
}
```

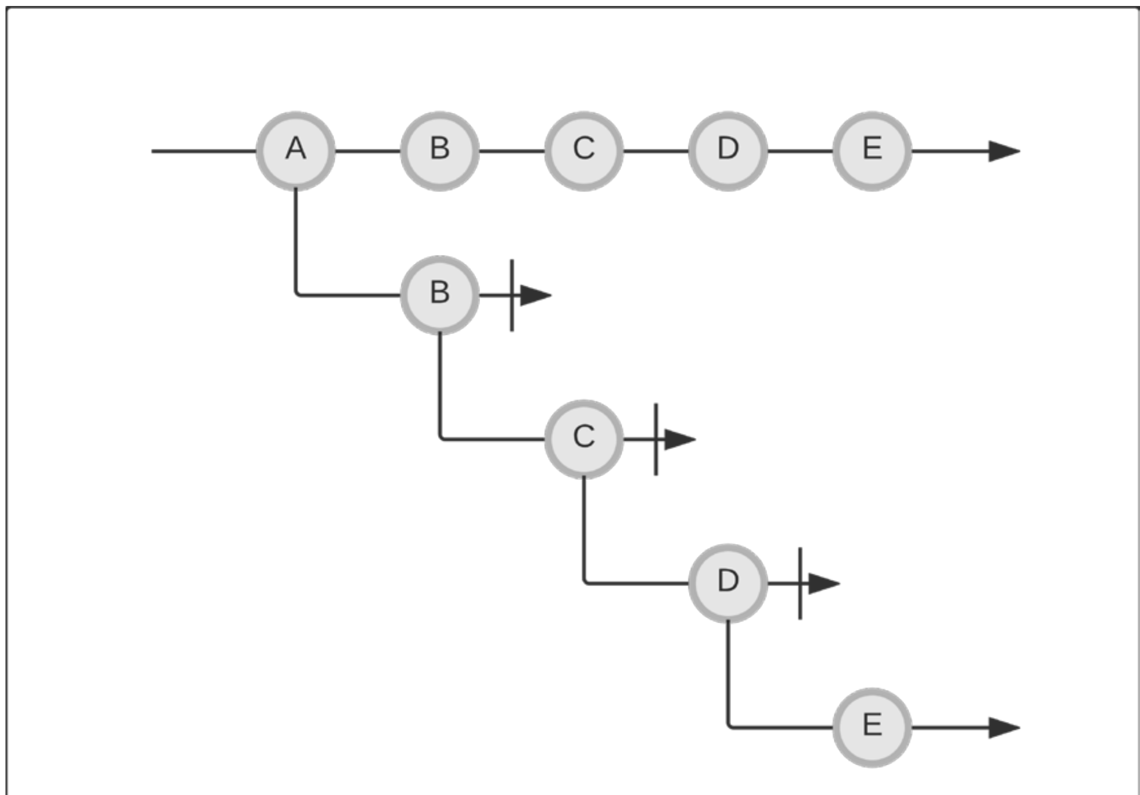
Koodiesimerkki 14. Rumpusyötteen käsittely perinteisin tekniikoin

3.4 Syötteen hahmonsovitus

Tapahtumavirtojen muovaus voi monimutkaisemmissa ongelmissa vaatia paljon luovuutta ja kykyä hahmottaa virran rakennetta, kuten seuraavasta tapauksesta ilmenee. Projektissa pelaaja voi tehdä loitsuja syöttämällä tiettyjä näppäinyhdistelmiä riittävän nopeasti. Pelaajan näppäinkomentoja käsiteltiin jo valmiiksi Observablena, joten esiin nousi tarve tunnistaa pelaajan syötteen tuottamassa tapahtumavirrassa ennalta määrättyjä syöteketjuja. Koska pelaajan on syötettävä loitsuyhdistelmät nopeasti, oli toiminnon myös tunnistettava ettei kahden virrassa esiintyvän merkin välillä ole liian pitkää aikaväliä. Se, että syöte oli juuri pelaajan tuottamia näppäinpainalluksia oli ongelman kannalta yhdentekevää, joten ongelmaksi määrittyi geneerisen hahmonsovittimen implementointi.

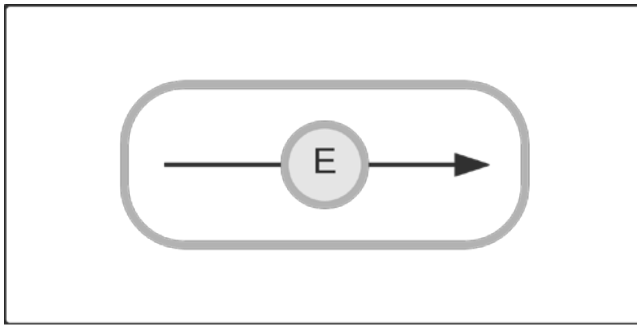
Ongelmaa määritellessä osa implementaatioratkaisuista nousi esiin luonnollisesti. Koska syöteketjut ovat suunnittelijoiden määriteltävissä koodin ulkopuolella, piti ratkaisun toimia millä tahansa syöteketjulla pituudesta tai tietotyypistä riippumatta. Ainoana teknisenä rajoituksena oli se, että itse syötteen tietotyypin oli oltava jokin yhtäläisyysverrattava tyyppi. Muutoin emme voisi millään tavoin tarkistaa onko pelaajan antama syöte yhtäläinen haetun syöteketjun yksittäisen tietueen kanssa. Käytännössä tämä tarkoittaa sitä, että geneerisen luokan on haettava vertailtavan tyypin yhtäläisyysvertailija EqualityComparer<T>.Default() metodilla. Tällä ratkaisulla referenssityypit jotka eivät toteuta omaa vertailijaansa implementoimalla System.IEquatable<T> rajapintaa tulevat vertailluiksi pelkän Equals-metodin pohjatoteutuksen perusteella, mikä pahimmillaan tarkoittaa pelkän objektista generoidun tarkistearvon vertailua. Joissain tilanteissa tämä saattaa olla haluttu toimintamalli, toisissa taas tärkeämpää olisi tietää vastaako objekti sisäiseltä tilaltaan vertailtavaa objektia. Koen kuitenkin että luokka on joustavampi tällä tavoin. Jos vertailun toteutuksella on merkitystä, on vertailtavan luokan itse kannettava vastuu vertailutavasta.

Alkuun muodostamme mielikuvan halutusta tietovirrasta. Kuvat mallintavat esimerkkitilannetta, jossa tarkoitus on tunnistaa syöteketju ABCDE.



Kuvio 7. Syötekäsittelyvirran rakenne

Kuten kuviosta 6 näkee, tietovirrat B, C ja D tietueille ovat identtiset ja voidaan toteuttaa samalla koodilla. Tietueet E ja A, eli haetun syöteketjun ensimmäinen ja viimeinen tietue ovat erikoistapauksia. Ne on käsiteltävä erikseen. Koska Observablen koostaminen muovaa itse lopullisen Observablen tiedonkäsittelyprosessia, oli geneerisestä syötelistasta List<TInput> mahdollista koostaa kuvanmukainen syötekäsittelijä-Observable aggregaation avulla. Aggregaatio-operaatio käy läpi kokoelman arvot suorittaen jokaiselle arvolle jonkinlaisen käsittelymetodin, joka voi hyödyntää aiempaa kumulatiivista arvoa ja palauttaa uuden kumulatiivisen arvon. Tämän ongelman puitteissa tarvitsemme lisäksi aggregaatiossa valinnaisen metodin juuriarvon saamiseksi. Juuriarvo toimii kumuloitavan arvon alkuarvona. Juuriarvomme tuottaa koodiesimerkin 15 mukainen metodi, joka koostaa syöteketjun viimeisen eli esimerkkitapauksessamme E-tietueen tunnistavan Observablen.

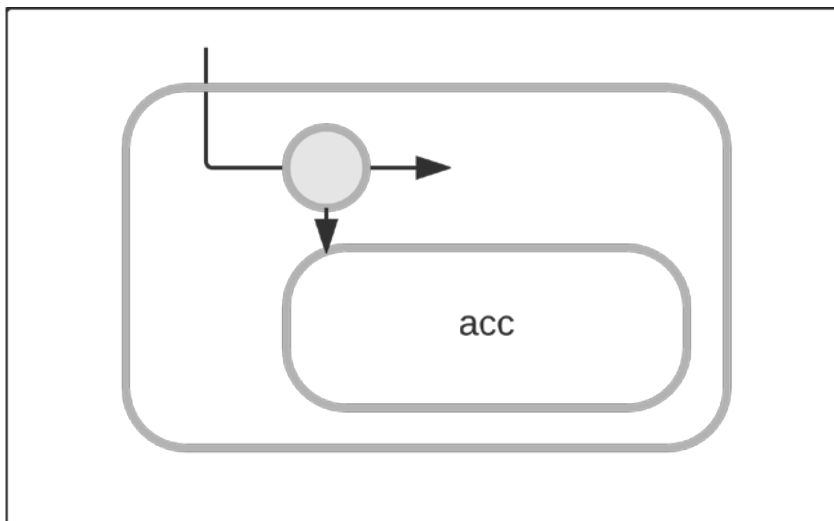


Kuvio 8. Aggregaation juuriarvo

```
private UniRx.IObservable<TInput> GenerateSeedObservable(
    UniRx.IObservable<TInput> inputStream,
    IEqualityComparer<TInput> comparer)
{
    return inputStream
        .Timeout(TimeSpan.FromMilliseconds(TimeoutMs))
        .CatchIgnore()
        .Take(1)
        .Where(input => comparer.Equals(input, InputPattern.Last()));
}
```

Koodiesimerkki 15. Juuriarvon luomismetodi

Akkumulaatiossa käytämme koodiesimerkissä 16 kuvattua metodia, joka koostaa kumulatiivisesta Observablesta uuden Observablen. Tämän Observablen tehtävä on tarkistaa, vastaako seuraava saapuva syöte odotettua syötettä, sekä kyseisen ehdon toteutuessa haarautua uuteen kumulatiivisen Observablen mallintamaan tietovirtaan.

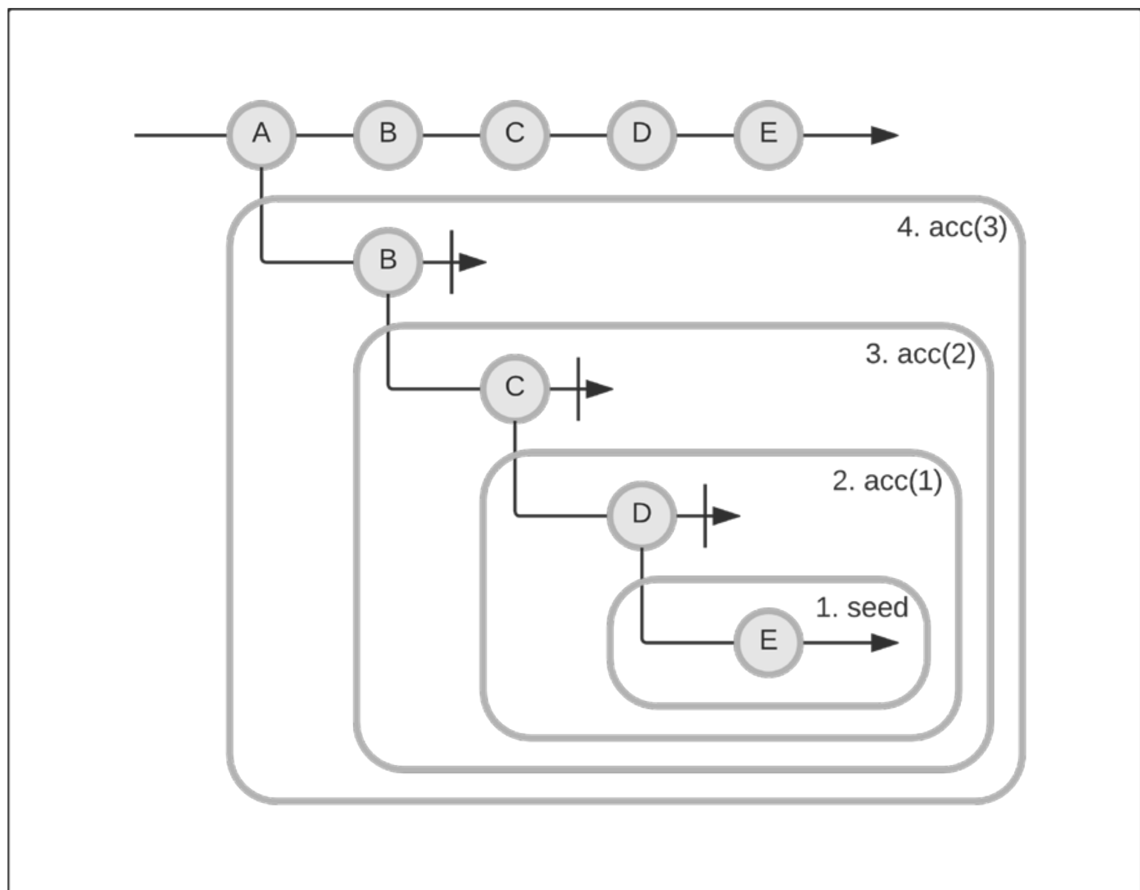


Kuvio 9. Aggregaation akkumulaatioarvo

```
private UniRx.IObservable<TInput> GenerateAccumulatorObservable(
    UniRx.IObservable<TInput> inputStream,
    IEqualityComparer<TInput> comparer,
    UniRx.IObservable<TInput> accumulator,
    TInput next)
{
    return inputStream
        .Timeout(TimeSpan.FromMilliseconds(TimeoutMs))
        .CatchIgnore()
        .Take(1)
        .Where(input => comparer.Equals(input, next))
        .SelectMany(accumulator);
}
```

Koodiesimerkki 16. Akkumulaatioarvon luomismetodi

Käytyämme läpi kaikki haetun syötteen tietueet ensimmäistä lukuunottamatta aggregaatiolla, päädymme Observableen joka poimii syötevirrasta seuraavan syötteen ja vertaa sitä haetun syöteketjun vastaavaan tietueeseen. Mikäli tietue ei vastaa syötettä tai aikakatkaisu-operaattori viestittää poikkeustapauksesta ennen uuden syötteen saapumista, päättää Observable itsensä haarautumatta eteenpäin. Jos syöte vastaa haettua tietuetta, haarautuu Observable jälleen odottamaan seuraavaa tietuetta vastaavaa syötettä.



Kuvio 10. Aggregaation rakenne havainnollistettuna

Aggregaation ulkopuolella käsittelemme toisen erikoistapauksen, eli ensimmäisen haetun syötearvon tunnistamisen. Tämä toteutuu yksinkertaisesti aloittamalla Observablen määrittely syötevirrasta, ja haarautumalla aggregaatiolla tuotettuun monikerroksiseen Observableen aina, kun haetun syöteketjun ensimmäinen tietue esiintyy syötevirrassa.

Toteutettu ratkaisu toimii, muttei katkaise syöteketjujen tunnistamista onnistuneen syötteen päätteeksi. Esimerkiksi haettaessa syöteketjua ABAB, tarvitsee käyttäjän onnistuneen syötteen päätteeksi syöttää vain AB, jotta sama syöteketju tunnistettaisiin uudelleen. Sama ongelma esiintyy haettaessa useaa eri syöteketjua, joiden merkeissä on päällekkäisyyttä. Syötteet ABCD ja ABCDE tunnistetaan molemmat peräjälkeen samalla käyttäjäsyötteellä. Joissain käyttötarkoituksissa tämä voi olla hyväksyttävää tai toivottua, mutta peliin suunnitellun käyttötarkoituksen puitteissa onnistuneen syöteketjun olisi tarkoitus katkaista saman ja muiden haettujen ketjujen tunnistaminen. Ongelman ratkaisu oli syöttää tunnistamiseen käytettyyn syötevirtaan katkaisumerkkejä. Tämä toteutuu yhdistämällä syötevirtaan paikallisesti toinen syötevirta ja lähettää onnistuneen syöteketjun tunnistamisen päätteeksi liitettyyn syötevirtaan katkaisumerkki. Itse katkaisu-

merkki riippuu syötteen tyypistä ja käyttötarkoituksesta. Tästä syystä hahmonsovittimen luomisparametreihin oli lisättävä uusi parametri, jolla luokan käyttäjä voi määritellä käytettävän katkaisumerkin. Koska katkaisumerkin on pelin tarpeiden puitteissa katkaistava myös muiden syöteketjujen tunnistaminen, lisäsin kokonaisratkaisuun uuden käsitteen: MatchingGroup<T>, eli tunnistusryhmä. Hahmonsovittimelle voi antaa vapaaehtoisena parametrina tunnistusryhmän. Kaikki samaa tunnistusryhmää käyttävät sovittimet jakavat yhteisen katkaisumerkkivirran.

Lopputulos oli toimiva ja toteutti kaikki määritellyt tarpeet. Toteutuksessa on kuitenkin suunnitteluvirheitä, joita en projektin aikataulun puitteissa huomannut tai ehtinyt korjata. Suurin näistä oli käyttäjän määrittämä katkaisumerkki. Katkaisumerkin tarkoitus on toimia symbolina, jota ei ikinä esiinny syöteketjuissa. Tiedostin ongelman jo luokkaa toteuttaessani ja kokeilin ongelman ratkaisua Nullable-tyypillä. Nullable on tyyppi, joka sisältää arvon ja tiedon siitä, onko arvo null, eli tyhjä. Esimerkiksi kokonaislukutyypin arvo ei voi normaalisti olla tyhjä. Tällaisissa tilanteissa Nullable voi auttaa mallintamaan itse arvon puuttumista. Jos hahmonsovitin käärii syötteen sisäisessä logiikassaan Nullable-tyyppiin, voidaan null-arvoa käyttää katkaisumerkkinä. Tämä ratkaisu monimutkaistaisi ratkaisua huomattavasti, sillä referenssityypit tukevat null-arvoa eikä niitä täten voi kääriä Nullableen. Tämäkin ongelma olisi ratkennut paikallisella geneerisellä datarakenteella. Rakenteen tarvitsisi sisältää vain syötteen arvo, sekä totuusarvoinen tieto siitä onko kyseinen syöte katkaisumerkki. Näin kaikki syötteet tyypistä riippumatta voitaisiin käsitellä uuteen datarakenteeseen käärittynä eikä käyttäjän tarvitsisi itse määritellä katkaisumerkkiä tyyppikohtaisesti erikseen.

```
public UniRx.IObservable<TOutput> MatchPattern(UniRx.IObservable<TInput> inputStream,
    TInput noopValue,
    MatchingGroup<TInput> matchingGroup = null,
    Func<TInput, TOutput> nonmatchingInputPassthroughFunction = null)
{
    var eqComparer = EqualityComparer<TInput>.Default;
    var backchannel = matchingGroup != null ? matchingGroup.SharedBackchannel : new Subject<TInput>();

    var noopMergedInputStream = inputStream.Merge(backchannel);

    UniRx.IObservable<TOutput> returnStream;
    if (InputPattern.Count > 1)
    {
        returnStream = noopMergedInputStream
            .Where(input => eqComparer.Equals(input, InputPattern.First()))
            .SelectMany(
                InputPattern
                    .Skip(1)
                    .Reverse()
                    .Skip(1)
                    .Aggregate(
                        GenerateSeedObservable(noopMergedInputStream, eqComparer),
                        (acc, next) =>
                            GenerateAccumulatorObservable(noopMergedInputStream, eqComparer, acc, next)))
            .Select(_ => OutputValue)
            .Do(_ => backchannel.OnNext(noopValue));
    } else if (InputPattern.Count == 1)
    {
        returnStream = noopMergedInputStream
            .Where(input => eqComparer.Equals(input, InputPattern.Single()))
            .Select(_ => OutputValue)
            .Do(_ => backchannel.OnNext(noopValue));
    }
    else
    {
        returnStream = Observable.Never<TOutput>();
    }

    return returnStream;
}
```

Koodiesimerkki 17. Hahmonsovittimen luomismetodi kokonaisuudessaan

4 Päätäntö

Lähtökohtanani oli selvittää soveltuuko reaktiivinen ohjelmointi tyypillisiin peliohjelmoinnissa vastaan tuleviin ongelmiin. Koska itse toteutettu peli oli kolmiulotteinen yksinpeliseikkailu, rajoittuivat kohdatut ongelmat erityisesti pelaajan syötteen käsittelyyn sekä hahmojen liikkeisiin ja vuorovaikutuksiin. Erityisesti reaktiivisen ohjelmoinnin perinteiseen käyttötarkoitukseen eli verkkoyhteyksien hallintaan ja koordinoimiseen liittyvät ongelmat jäivät tämän projektin puitteissa tutkailtavien ongelmien ulkopuolelle

Reaktiiviset ohjelmointitekniikat soveltuivat ratkaisemaan kaikki vastaan tulleet ohjelmointiongelmat. Useimmissa projektiin toteutetuissa ratkaisuissa Rx-koodi johti lyhyempiin ja lueattavampiin luokkiin. Tyypillisessä Unity3D-koodissa tapahtumien seulominen ehtojen mukaan, ongelmalle relevantin tiedon valitseminen ja itse tapahtumaan reagoiminen toivotulla toiminnallisuudella sekoittuu toisiinsa sisäkkäisten if-lauseiden ja paikallisten muuttujien sekasotkuksi. Reaktiivissa ratkaisuissa seulomiseen ja tiedon valintaan liittyvä logiikka jäi selkeästi tapahtumavirran määrittelyn vastuulle, kun taas tapahtumavirran tapahtumiin reagoivien metodien vastuulle jäi tehdä vain tapahtuman tuottamat seuraukset. Koodin lukijalle on selkeää mitkä osat koodista koskevat käsiteltävän tiedon valintaa ja mitkä itse käsittelyä.

Tekniikassa oli kuitenkin myös huonot puolensa. Monimutkaisemmissa tilanteissa kuten hahmonsovitusongelmassa tuotetun Observablen rakenne oli hyvin vaikea hahmottaa, eikä itse koodin lukeminen selkeytä asiaa juurikaan. Vaikka ratkaisu itsessään oli elegantti, ei koodia voisi helposti luovuttaa muiden kehittäjien käsiin hankalan rakenteen vuoksi. Ongelmaa voisi lievittää tarkalla kommentoinnilla, tosin tarkan kommentoinnin tarve on itsessään merkki siitä, ettei koodin luettavuus ole riittävää.

Monimutkaisten Observable-rakenteiden ongelmaa lisää ReactiveX:n oppimiskäyrä. Tekniikoiden oppimiseen ja uuden ajattelutavan sisäistämiseen kuluu helposti vähintään yksi kuukausi, ellei enemmän riippuen ohjelmoijan oppimisnopeudesta. Järkevän ratkaisun löytäminen oli monissa ongelmissa useamman kuukauden kokemuksenkin jälkeen kovan pohdinnan takana. Koska reaktiiviset tekniikat eivät ole yleisesti käytössä pelikehityksessä, on kyseisiä tekniikoita osaavien kehittäjien löytäminen projekteihin vaikeaa. Perustasolla reaktiivisen koodin luettavuudesta hyötyminen vaatii kuitenkin ymmärryksen vain yleisimmistä operaattoreista ja tapahtumavirtojen perustoiminnasta.

Suurin hyöty pienimmällä vaivalla saavutetaankin kokemuksieni perusteella vain muutaman päivän oppimisella. Tällöin ReactiveX on parempi jättää monimutkaisemmista ongelmista pois ja rajoittaa vain luvun 3.1 koodiesimerkkien tapaisiin selkeämpiin tapahtumien seulomiseen liittyviin käyttötapauksiin.

Oppimiskäyrästä huolimatta reaktiiviset tekniikat soveltuivat pelikehityksen tarpeisiin erinomaisesti ja tulen niitä käyttämään projekteissa jatkossakin. Vaikka olin käyttänyt Rx:ää ja osasin perusteet jo ennen opinnäytetyön toteutusta, koen syventäneeni osaamistani reaktiivisten tekniikoiden osalta tasolle, jota tekniikoiden ammattiosaajalta voitaisiin odottaa. Itse Skábma-projekti ei tähän vielä pääty, mutta toteutin opinnäytetyöni puitteissa mielestäni onnistuneesti projektin keskeiset perustoiminnot ja valmistelin näin perustukset suurempaa projektia varten.

LÄHTEET

Campbell, L. Introduction To Rx. Luettu 10.7.2017. http://www.introtorx.com/

Gallagher, M. What is reactive programming and why should I use it?. Luettu 24.11.2017. https://www.cocoawithlove.com/blog/reactive-programming-what-and-why.html

IBM, 2017. How (and why) Reactive microservices are mainstreaming with enterprise developers. Luettu 20.10.2017. https://developer.ibm.com/dwblog/2017/reactive-microservices/

Kawai, Y. 2016. UniRx - Reactive Extensions For Unity. Luettu 5.7.2017. https://github.com/neuecc/UniRx

Lesh, B. 2016. Learning Observable By Building Observable. Luettu 16.11.2017. https://medium.com/@benlesh/learning-observable-by-building-observable-d5da57405d87

Lew, D. 2017. An Introduction to Functional Reactive Programming. Luettu 24.11.2017. http://blog.danlew.net/2017/07/27/an-introduction-to-functional-reactive-programming/

Microsoft, 2012. Functional Programming vs. Imperative Programming. Luettu 5.7.2017 https://technet.microsoft.com/en-us/library/bb669144(v=vs.110).aspx

ReactiveX. Luettu 27.6.2017. http://reactivex.io/

Red Stage Entertainment. Luettu 28.10.2017. http://www.redstage.fi/

Sexton, D. 2013. Hot and Cold Observables. Luettu 16.11.2017. http://davesexton.com/blog/post/Hot-and-Cold-Observables.aspx

Staltz, A. 2016. The Introduction to Reactive Programming you've been missing. Luettu 24.11.2017. https://gist.github.com/staltz/868e7e9bc2a7b8c1f754