Opinnäytetyö (AMK)

Tietojenkäsittelyn koulutusohjelma

Sähköisen liiketoiminnan järjestelmät

2015

Lauri Luukkanen

PHASER – PELIOHJELMOINTIKEHYKSEN ARVIOINTIA

TURUN AMMATTIKORKEAKOULU
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietojenkäsittelyn koulutusohjelma | Sähköisen liiketoiminnan järjestelmät

Kesäkuu 2015 | 34

Päivi Nygren

Lauri Luukkanen

PHASER – PELIOHJELMOINTIKEHYKSEN ARVIOINTIA

Tämän opinnäytetyön tavoitteena on tutustua Phaser – peliohjelmointikehyksen käyttöön HTML5- ja JavaScript-ohjelmointikielillä toteutetuissa peleissä.

Tätä tarkoitusta varten luotiin muutama yksinkertainen verkkoselaimessa toimiva peli. Näistä peleistä opinnäytetyöhön valittiin kaksi, joissa käsitellään pelien perustoimintoja, kuten animaatio, törmäyksen tunnistus ja peliobjektien liikkuminen.

Pelit ohjelmoitiin Komodo Edit 8.5 – tekstieditorilla. Lopputuloksena syntyi kaksi peliä, joiden kehitys ja toiminnot on kuvattu tässä raportissa. Pelit eivät ole valmiita tai julkaisukelpoisia, mutta havainnollistavat Phaserin käyttöä.

ASIASANAT:

Peliohjelmointi, peliohjelmointikehys, Phaser

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Business Information Technology | e-Business Systems

June 2015 | 34

Päivi Nygren

Lauri Luukkanen

EVALUATING PHASER GAME PROGRAMMING FRAMEWORK

The goal of this thesis is to look into and evaluate the use of the Phaser game programming framework in games that are programmed in HTML5 and JavaScript.

For this purpose a few browser games were created. Two of these games were selected to be used in this thesis. Creating the games involved the use of basic game elements such as animation, collision detection and the movement of the game objects.

The games were programmed with the Komodo Edit 8.5 text editor. The end result was two games. The development and functionalities of these games are described in this report. The games are not complete or ready for release but give some guidelines in the use of Phaser.

KEYWORDS:

Game programming, framework, Phaser

SISÄLTÖ

1 JOHDANTO **6**

2 PHASER-PELIMOOTTORI **7**
2.1 Yleistä 7
2.2 Phaser 7
2.2.1 Fysiikkamoottori 9
2.2.2 Assets 10

3 ESIMERKKISOVELLUKSET **12**
3.1 Asteroid dodger -peli 12
3.1.1 Raketti-peliobjekti 13
3.1.2 Asteroidi-peliobjektit 15
3.1.3 Ammuslaatikko-peliobjekti 15
3.1.4 Ohjus-peliobjektit 16
3.1.5 Törmäyksen tunnistus Arcade-fysiikkamoottorilla 18
3.1.6 Partikkeliefektit 22
3.2 Ledge jumper -peli 24
3.2.1 Pelihahmon animaatio 24
3.2.2 Pelihahmon liikkuminen pelimaailmassa 26
3.2.3 Pelimaailma 28
3.2.4 Pelin äänet 31

4 YHTEENVETO **32**

LÄHTEET **34**

LIITTEET

Liite 1. Asteroid Dodger – pelin ohjelmakoodi.
Liite 2. Ledge Jumper – pelin ohjelmakoodi.

KUVAT

Kuva 1. Phaser-pelin rakenne 8
Kuva 2. AABB - törmäyksen tunnistus 9
Kuva 3. Pelihahmo png- ja jpeg-tiedostomuotoa lähteenä käyttäen 10
Kuva 4. Preload-funktio 11
Kuva 5. Sprite-peliobjektin kääntyminen origonsa ympäri 14
Kuva 6. Spriten bodyn muokkaus 21
Kuva 7. Sprite-peliobjektin kallistuminen suhteessa bodyyn 22
Kuva 8. Asteroidin hajoamista kuvaava partikkeliefekti 23
Kuva 9. Pelihahmon luomiseen käytetty sprite sheet 25
Kuva 10. Kuvaus canvasia suuremmasta pelimaailmasta 29
Kuva 11. Tile sprite 30

1 JOHDANTO

Tässä opinnäytetyössä arvioidaan Phaser – pelimoottoria pelinkehityksen apuvälineenä. Tätä tarkoitusta varten tein muutamia HTML5 -pelisovelluksia, jotka käyttävät Phaser-kirjastoa. Tarkoitus ei ollut tehdä loppuun saakka viimeisteltyä peliä, vaan lähinnä yksinkertaisia pelinsovelluksia, joissa testataan Phaserin tarjoamia ominaisuuksia. Tekemäni sovellukset testasivat pääasiassa törmäyksen tunnistusta sekä peliobjektien liikkumista. Sovellukset kuitenkin sisälsivät testaamiensa ominaisuuksien osalta runsaasti yhteneväisyyksiä. Siksi päädyin valitsemaan niistä kaksi, joissa käyttämäni toiminnot tulevat esille.

Aiheen valintaan vaikuttivat pitkäaikainen kiinnostukseni peleihin ja ammattikorkeakouluopintojen myötä herännyt kiinnostus ohjelmointiin. Tutustuin pelinkehitykseen alun perin C# -ohjelmointikielellä, mutta halusin tätä opinnäytetyötä varten perehtyä verkkoselaimessa toimivien pelien kehitykseen. Vaihtoehtoja arvioidessani päädyin valitsemaan työkaluiksi HTML5:n ja JavaScriptin, koska ne toimivat yleisimmin käytetyissä verkkoselaimissa, kuten Microsoftin Internet Explorer, Mozilla Firefox ja Google Chrome, ilman asennettavia lisäosia.

Opiskelin JavaScript-ohjelmointikielen perusteita W3Schools.com sivuston avulla.(W3Schools 2015). Tutustuttuani HTML5-pelien kehitykseen kiinnostuin pelimoottoreista ja niiden tarjoamista mahdollisuuksista. Lopulta rajasin tämän opinnäytetyön aiheeksi Phaserin, pelien kehitykseen tarkoitetun avoimen lähdekoodin kehyksen, ominaisuuksien arvioinnin.

2 PHASER-PELIMOOTTORI

2.1 Yleistä

Pelimoottori on ohjelmistokehys, joka on suunniteltu pelien tekemistä varten. Sen ydintoiminnallisuuksiin sisältyy yleensä muun muassa renderointi, fysiikkamoottori, äänet, animointi ja tekoäly. Valmiin pelimoottorin käyttäminen pelin kehittämisen pohjana nopeuttaa kehitysprosessia, koska ne tarjoavat toiminnallisuuksia hyödynnettäviksi sen sijaan, että koko ohjelmarunko pitäisi kirjoittaa alusta asti itse. (Game engine 2014).

Peliohjelmoinnin yhteydessä renderoinnilla (rendering) tarkoitetaan prosessia, jossa kuva tuotetaan näytölle mallista.(Rendering 2015). Tämä malli on käytännössä informaatiota, jota tietokone pystyy lukemaan, esimerkiksi kuvatiedosto tai vektoridataa.

Fysiikkamoottorin tehtävä peleissä on simuloida fyysisiä ilmiöitä. Tällä voidaan tarkoittaa esineiden liikkumisen ja törmäysten mallintamista sekä erilaisten fyysisten voimien, kuten painovoiman ja kitkan vaikutuksen laskentaa.(Kanber 2012). Fysiikkamoottori voi myös olla varsinaisesta pelimoottorista irrallinen kirjasto, jota pelimoottori hyödyntää.

2.2 Phaser

Html5/JavaScript pelimoottoreita on saatavilla useita eri vaihtoehtoja. Valitsin Phaserin, koska sen ominaisuuden riittivät hyvin tavoitteisiini ja sillä on myös runsaasti selkeitä esimerkkejä ja dokumentoinnin tarjoava sivusto. Phaser on Photon Storm Ltd:n kehittämä avoimen lähdekoodin peliohjelmointikehys.

```
<script type="text/javascript">
  var game = new Phaser.Game(800, 600, Phaser.AUTO, '',
                                { preload: preload, create: create, update: update });

    function preload(){
        game.load.image('background', 'assets/Background.png');
        game.load.spritesheet('player', 'assets/Jumper.png',16,40);
        game.load.audio('jump', '../Sounds/Jump.mp3');
    }

    function create(){
        game.world.setBounds(0, 0, 2400, 600);
        game.physics.startSystem(Phaser.Physics.ARCADE);
    }

    function update(){
        game.physics.arcade.collide(player, platforms);
    }
</script>
```

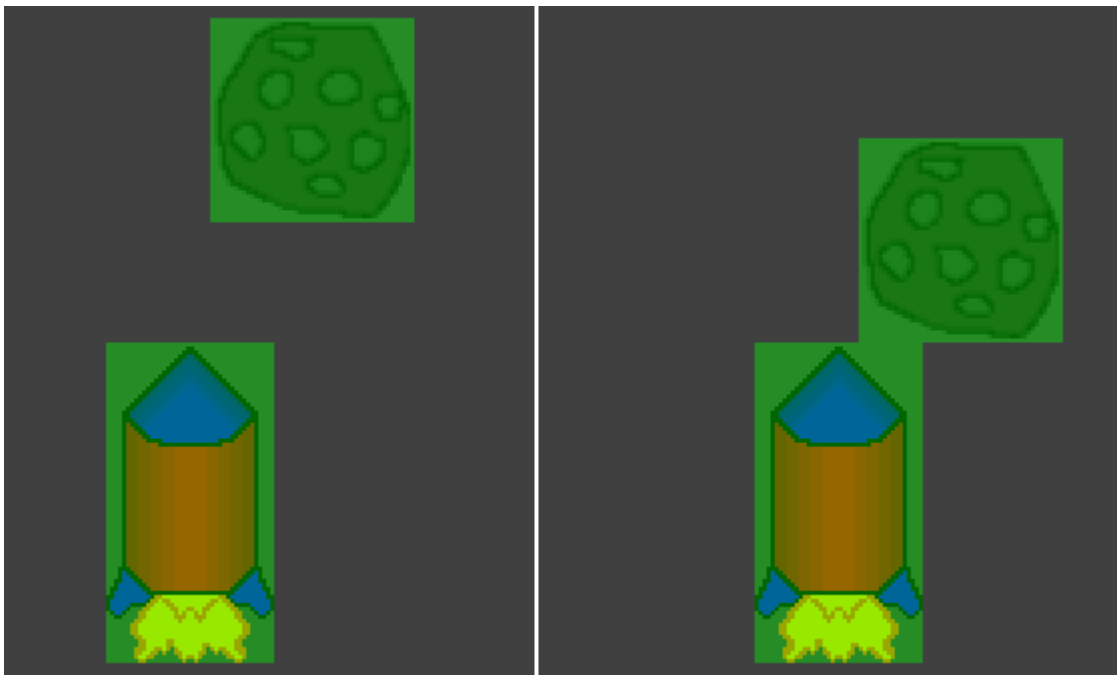
Kuva 1. Phaser-pelin rakenne

Yllä olevassa kuvassa on esitettynä Phaser-pelin rakenne. Ensin luodaan instanssi Phaser.Game-objektista, jolle annetaan muutamia parametreja. Kaksi ensimmäistä parametria määrittävät käytettävän canvas-elementin leveyden ja korkeuden pikseleinä. Ne siis määrittävät pelialueesta näkyvän osan koon, varsinaisen pelialueen koko voidaan kuitenkin määritellä suuremmaksi. Seuraava parametri määrittää käytettävän kontekstin renderoinnille. Tässä vaihtoehtoina ovat AUTO, WEBGL, CANVAS, ja HEADLESS. AUTO-vaihtoehto yrittää ensisijaisesti käyttää WebGL:ä, mutta mikäli selain ei tätä tue, käytetään Canvasia. WebGL on ohjelmointirajapinta, joka mahdollistaa 3D-grafiikan käytön selaimessa ilman asennettavia lisäosia (Khronos group 2011). HEADLESS-vaihtoehdolla renderointia ei tehdä lainkaan. Neljäs parametri määrittää mihin DOM-elementtiin Phaserin luoma canvas-elementti sisällytetään. Arvoksi on annettu tyhjä merkkijono, jolloin se sisällytetään verkkosivun runkoon. Viides parametri on objekti, joka sisältää viittaukset Phaserin perusfunktioihin. Muita mahdollisia parametreja ovat state, transparent, antialias sekä physicsConfig, joita ei kuitenkaan tässä yhteydessä ole käytetty.

Preload on ensimmäinen funktio, jota kutsutaan. Sitä tarvitaan pelissä käytettävien tiedostojen, kuten kuvien ja äänien lataamiseen. Phaser hoitaa preload-funktion kutsumisen automaattisesti. Create-funktiossa luodaan pelin tarvitsemat objektit ja määritetään niiden ominaisuudet. Sitä kutsutaan automaattisesti preload-funktion tultua valmiiksi. Update-funktiota kutsutaan jokaisella kuvalla eli 60 kertaa sekunnissa kuvataajuuden ollessa 60. Siinä määritetään pelin tapahtumat kuten hahmojen liikkuminen ja törmäykset.(Photon Storm 2013).

2.2.1 Fysiikkamoottori

Phaser-kirjasto sisältää kolme eri fysiikkamoottoria: Arcade, P2 ja Ninja. Arcade -fysiikkamoottori käyttää AABB (axis alligned bounding box) – törmäyksen tunnistusta, jossa törmäykset tunnistetaan x- ja y-akseleiden suuntaisten suorakulmioiden välillä (kuva 2). Ninja ja P2 ovat ominaisuuksiltaan monipuolisempia sisältäen muun muassa tarkempia törmäyksen tunnistuksen muotoja, mutta ne ovat samalla raskaampia käyttää.

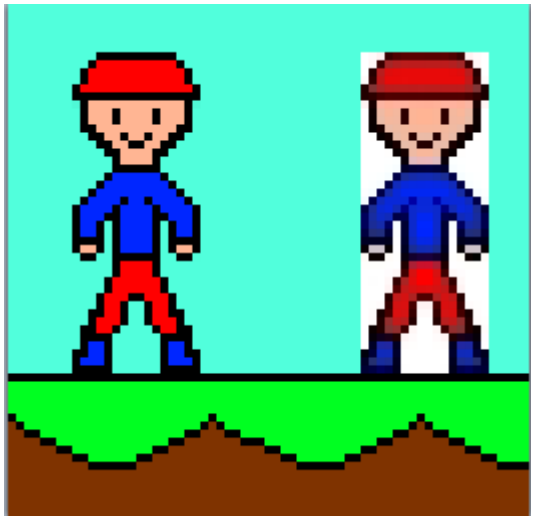


Kuva 2. AABB - törmäyksen tunnistus

Valitsin omiin testeihini Arcade-fysiikkamoottorin, joka on vaihtoehdoista kevyin ja yksinkertaisin, siten siis sopiva Phaserin käytön aloittavalle.

2.2.2 Assets

Kun Phaser-peliä piirretään canvas-elementille, käytetään lähteenä yleisesti jpeg- tai png-formaatissa olevaa kuvatiedostoa. Kuva 3 havainnollistaa eroavaisuuksia näiden kahden tiedostomuodon välillä.



Kuva 3. Pelihahmo png- ja jpeg-tiedostomuotoa lähteenä käyttäen

Näistä kahdesta vaihtoehdosta png – formaatti on monikäyttöisempi, koska se on häviötön ja tukee kuvan läpinäkyviä osia. Esimerkiksi pelin taustamaisemiin soveltuvaa jpeg – formaattia en omissa pelisovelluksissani käyttänyt lainkaan.

Preload-funktio lataa kaikki siinä määritellyt ”assetit” eli pelin käyttämät ulkoiset tiedostot ja varastoi ne välimuistiin. Nämä tiedostot voivat olla muun muassa kuva-, ääni-, JavaScript-, tai JSON-tiedostoja.

```
function preload(){
    game.load.image('ground', 'assets/Ledge.png');
    game.load.image('water', 'assets/Water.png');
    game.load.image('background', 'assets/Background.png');
    game.load.spritesheet('player', 'assets/Player.png',16,40);
    game.load.image('drop', 'assets/Drop.png');
    game.load.audio('jump', '../Sounds/Jump.mp3');
    game.load.audio('splash', '../Sounds/Splash.mp3');
    game.load.audio('bg', '../Sounds/bgMusic.mp3');
}
```

Kuva 4. Preload-funktio

Kun asset ladataan ja varastoidaan välimuistiin, sille täytyy määritellä tiettyjä parametreja. Yhteistä kaikille kuvassa 4 havainnollistetuille ladattaville tiedostoille ovat ensimmäiset kaksi parametria: avain ja URL (Uniform Resource Locator). Avain on merkkijono, jolla voidaan myöhemmin ohjelmassa tunnistaa kyseinen asset ja URL kertoo ohjelmalle käytetyn tiedoston sijainnin.

Eri asseteilla on myös muita mahdollisia parametreja. Tästä hyvänä esimerkkinä toimii sprite sheet. Sprite sheet on useasta samankokoisesta framesta eli kuvasta koostuva kuva. Tämän tyyppistä tiedostoa ladattaessa täytyy edellä mainittujen parametrien lisäksi määrittää yksittäisen framen korkeus ja leveys. Myös framejen lukumäärä, marginaalit ja mahdollinen tyhjä tila framejen välissä voidaan syöttää parametreiksi.

Yleinen käyttötarkoitus sprite sheetille on tuottaa animaatiota. Sprite sheetin framet voivat esimerkiksi kuvata pelihahmon eri asentoja juostessa. Vaihtelemalla näitä frameja määritellyssä järjestyksessä saadaan aikaan animaatio pelihahmon juoksusta.

Kun tarvittavat tiedostot on varastoitu välimuistiin, niitä voidaan käyttää peliobjektien luomiseen create-funktiossa.

3 ESIMERKKISOVELLUKSET

Tutustuakseni Phaser-pelien kehitykseen käytännössä tein muutaman sovelluksen. Lähtökohtaisesti suunnittelin millaisen pelin haluan tehdä ja ryhdyin tämän jälkeen selvittämään toteutusta Phaserin puitteissa. Peligenreistä esimerkkeihini valikoitui ammuntapeli ja tasohyppely, koska niistä minulla oli selkeä mielikuva millaisen pelin halusin tehdä ja Arcade-fysiikkamoottori soveltui niiden toteutukseen hyvin.

Työvälineinä käytin Phaser 2.3.0 kirjastoa, Komodo Edit 8.5-tekstieditoria, Paint.NET-kuvankäsittelyohjelmaa, Mongoose Free 5.5-verkkopalvelinta, Google Chrome-selainta kehittäjäntyökaluineen sekä Phaserin dokumentointia.(Photon Storm 2015).

Pelien ohjelmakoodin kirjoitin suoraan html-tiedostoon, en ryhtynyt kirjoittamaan erillistä JavaScript-tiedostoa. Jotta Phaser-kirjastoa voitiin hyödyntää, piti pelien html-tiedostoihin sisällyttää viittaus siihen:

```
<script type="text/javascript" src="../phaser.js">
</script>
```

3.1 Asteroid dodger - peli

Ensimmäiseksi testitapaukseksi ryhdyin suunnittelemaan ammuntapeliä, jossa oli tarkoitus ohjata avaruusalusta, kerätä ammuksia sekä väistellä tai tuhota vastaan tulevia kohteita. Aloitin työt miettimällä mitä pelissä tulisi näkyä. Tässä vaiheessa oleelliset peliobjektit olivat pelaajan ohjaama raketti, asteroidit, ammukset ja kerättävät ammuspakkaukset.

Pelin kehittelyssä seuraava askel oli peliobjektien lisääminen peliin ja niiden toimintojen ohjelmointi. Ideana oli, että rakettia ohjataan ainoastaan vaakasuunnassa, asteroidien ja ammuspakkauksien lähestyessä pystysuunnassa. Jotta peliobjektien käyttäytymistä päästiin hallitsemaan, piti käynnistää fysiikkamoottori create-funktiossa:

```
game.physics.startSystem(Phaser.Physics.ARCADE);
```

3.1.1 Raketti-peliobjekti

Kun raketin tarvitsema kuvatiedosto oli määritetty preload-funktiossa, voitiin luoda raketille sprite:

```
rocket = game.add.sprite(400, game.world.height - 150,
'rocket');
```

Parametreiksi syötettiin sijainti x-akselilla (400), sijainti y-akselilla (game.world.height -150) sekä käytettävän assetin avain.

Jotta pelin fysiikanmallinnusta voitiin hyödyntää raketin käyttäytymisessä, piti fysiikkamoottori aktivoida myös kyseisen peliobjektin osalta:

```
game.physics.arcade.enable(rocket);
```

Näin saatiin käyttöön tähän spriteen yhdistetty runko (body), johon peliobjektin fyysiset toiminnot tuli kohdistaa. Lisäksi luotiin objekti näppäimistön suuntanäppäimille, jotta liikkeen suuntaa voitiin ohjata:

```
cursors = game.input.keyboard.createCursorKeys();
```

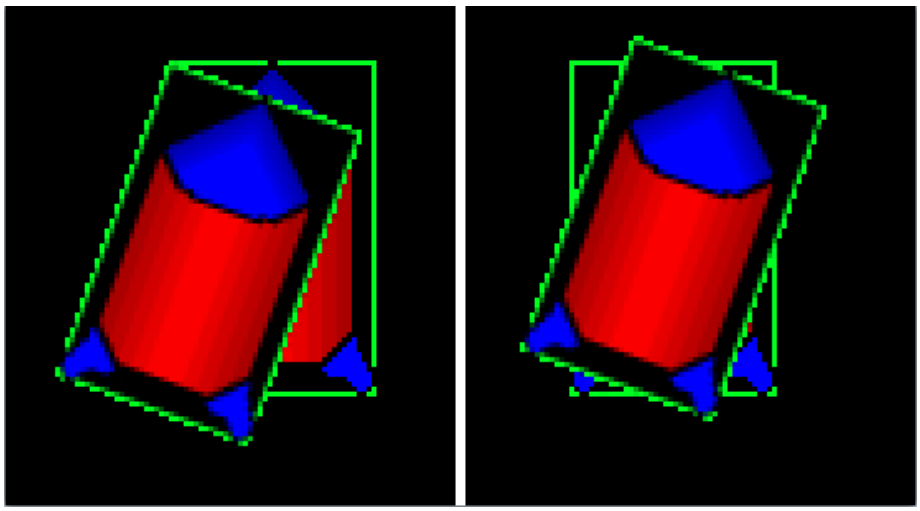
Raketin liikuttaminen vaakasuunnassa saatiin mahdolliseksi hyödyntämällä rungon nopeus (velocity) ominaisuutta x-akselilla ja raketin kallistuminen kulkusuuntaansa spriten kallistus (angle) ominaisuudella. Raketin kulkusuunta ja kallistus ohjelmoitiin määräytymään vasemman ja oikean suuntanäppäimen painalluksen mukaan. Ohjelmakoodi on esitetty alla:

```
rocket.body.velocity.x = 0;
rocket.angle = 0;
```

```
if (cursors.left.isDown){
        rocket.body.velocity.x = -300;
        rocket.angle = 340;
}
else if (cursors.right.isDown){
        rocket.body.velocity.x = 300;
        rocket.angle = 20;}
```

Raketin kallistumisessa oli kuitenkin ongelma. Piste jonka ympäri kääntyminen tapahtui, sijaitsi spriten vasemmassa yläkulmassa raketin ulkopuolella. Jotta raketti saatiin kallistumaan luonnollisemmin, piti spriten origo siirtää sen keskelle:

```
rocket.anchor.set(0.5, 0.5);
```



Kuva 5. Sprite-peliobjektin kääntyminen origonsa ympäri

Raketin ei kuitenkaan haluttu voivan poistua pelialueen ulkopuolelle. Tämä voitiin estää asettamalla spriten runko törmäämään pelialueen reunoihin:

```
rocket.body.collideWorldBounds = true;
```

3.1.2 Asteroidi-peliobjektit

Pelin asteroidien käyttäytymistä suunnitellessa oleellista oli saada luotua useita samankaltaisia peliobjekteja, jotka kuitenkin sijainniltaan ja nopeudeltaan erosivat toisistaan. Tähän soveltui erinomaisesti Phaserin tarjoama group-peliobjekti. Sen avulla voitiin luoda ryhmä ja aktivoida body niiden osalta:

```
asteroids = game.add.group();

asteroids.enableBody = true;
```

Tämän jälkeen voitiin liittää siihen haluttu määrä, tässä tapauksessa 12, asteroidi-peliobjekteja sekä määrittää näiden ominaisuudet, kuten sijainti x- ja y-akselilla, käytettävän assetin avain sekä nopeus, for-silmukalla:

```
for (var i = 0; i < 12; i++){

        var asteroid = asteroids.create(Math.random()
        * 750, Math.random() * (-300), 'asteroid');

        asteroid.body.velocity.y = 170 + Math.ran-
        dom() * 100;

}
```

3.1.3 Ammuslaatikko-peliobjekti

Ammuslaatikoiden suunniteltiin olevan peliobjekteja, jotka ilmestyvät tasaisin aikavälein ja joita keräämällä pelaajan on mahdollista kartuttaa ammusvarastoaan. Ajoitettu ammuslaatikoiden luominen onnistui ajastimen avulla. Ensin kirjoitettiin funktio, joka lisää peliin ammuslaatikko-peliobjektin näkyvän pelialueen yläpuolelle, x-akselilla satunnaiseen kohtaan:

```
function spawnCrate(){

          crate = game.add.sprite(200 + Math.random() *
          575, -50, 'crate');

          game.physics.arcade.enable(crate);

          crate.body.velocity.y = 150;

}
```

Tämän jälkeen lisättiin ajastin, joka kutsuu edellä mainittua funktiota 10000 millisekunnin välein:

```
timer = game.time.create();

timer.loop(10000, spawnCrate, this);

timer.start();
```

3.1.4 Ohjus-peliobjektit

ohjuksien lisäämiseen käytettiin jälleen group-peliobjektia, jonka lisääminen tapahtui samoin kuin asteroidien kohdalla. Mutta yksittäisten ohjusten lisääminen tähän ryhmään tehtiin eri tavalla:

```
missiles.createMultiple(15, 'missile');

missiles.setAll('checkWorldBounds', true);

missiles.setAll('outOfBoundsKill', true);
```

Missiles-ryhmän createMultiple-metodiin syötettiin parametreina ohjusten lukumäärä (15) sekä käytettävän assetin avain. Lisäksi setAll-metodin avulla jokaisen ryhmän jäsenen checkWorldBounds ja outOfBoundsKill parametrit saivat arvon tosi (true). Näin ohjukset tunnistavat pelialueen rajat ja ”kuolevat” ylittäessään ne.

Jokaisen ohjuksen exists-parametri on ryhmään lisättäessä epätosi (false) ja ne eivät siis ole aktiivisia (ovat ”kuolleita”) ennen kuin pelaaja ampuu ohjuksen. Ylös-

nuolinäppäimen painaminen ohjelmoitiin kutsumaan ohjusten ampumisen hoitavaa funktiota (shoot), mikäli pelaajalla on ohjuksia vielä jäljellä (ammo > 0). Jotta ohjusten laukaisu saatiin toimimaan siten, että jokaisella napin painalluksella lähtee vain yksi ohjus, lisättiin muuttuja readyToFire. Tämän muuttujan oletusarvo on tosi, se muuttuu ylös-näppäintä painettaessa epätodeksi ja näppäimen noustessa takaisin ylös taas todeksi. Ohjuksen laukaisu onnistuu vain, jos sen arvo on tosi:

```
if (cursors.up.isDown && readyToFire == true &&
ammo > 0) {

          shoot();

          readyToFire = false;

}

if (cursors.up.isUp) {

          readyToFire = true;

}
```

Shoot-funktio ohjelmoitiin ampumaan ohjus, mikäli kaikki peliin lisätyt ohjus-peliobjektit eivät ole jo aktiivisina, eli ainakin yksi ohjus on ”kuollut” (missiles.countDead() > 0). Tällä tavalla voidaan kierrättää samoja ohjus-peliobjekteja, jolloin saadaan säästettyä resursseja. Group-peliobjekti on käytännössä taulukko, josta tässä tapauksessa shoot-funktio ottaa aina ensimmäisen ”kuolleen” käyttöön. Aina kun ohjus ylittää pelialueen rajat, palautuu se jälleen käytettäväksi (outOfBoundsKill). Shoot-funktio sijoittaa ammuttavan ohjuksen lähtöpisteen x- ja y-akselilla raketin mukaisesti ja määrittää sen nopeuden y-akselilla. Shoot-funktion ohjelmakoodi on esitettynä alla:

```
function shoot() {
        if (missiles.countDead() > 0) {
                var missile = missiles.get
                FirstDead();
                missile.reset(rocket.x,    rocket.y
                -8);
                missile.body.velocity.y = -200;
                ammo--;
                text.setText('Ammo: ' + ammo);
        }
}
```

3.1.5 Törmäyksen tunnistus Arcade-fysiikkamoottorilla

Oleellinen osa tätä esimerkkisovellusta oli testata Phaserin törmäyksen tunnistusta (collision detection). Sitä hyödynnettiin lopulta kohteiden tuhoamisessa, esineiden keräämisessä ja peliobjektien kierrätyksessä. Valitsemani Arcade-fysiikkamoottorin käyttämä AABB törmäyksen tunnistus oli yksinkertainen käyttää, mutta sen puutteet tulivat myös selkeästi esille.

Törmäyksen tunnistus kohdistuu spriten runkoon (body), joka on oletusarvoisesti spriten kokoinen suorakaide. Esimerkkisovelluksessa tunnistus tehtiin jokaisessa tapauksessa samalla tavalla:

```
game.physics.arcade.collide(missiles,
asteroids,explodeAsteroid);
```

game.physics.arcade.collide-metodille syötettiin parametreina objektit, joiden välillä törmäystä tunnistettiin (missiles, asteroids) ja funktio, jota törmäyksen sattuessa kutsuttiin (explodeAsteroid).

Pelin asteroideja, joita luotiin rajallinen määrä, kierrätettiin pelialueen alareunan ulkopuolelle sijoitetulla pelialueen levyisellä peliobjektilla (recycler). Asteroidin törmätessä siihen, kutsuttiin funktiota (recycleAsteroid), joka siirtää asteroidin pelialueen yläreunan ulkopuolelle ja muokkaa sen sijaintia x-akselilla sekä nopeutta:

```
function recycleAsteroid (r, a) {
        a.reset(a.x = Math.random()*750, a.y =
        Math.random()*(-300));
        a.body.velocity.y = 170 + Math.random()*
        100;
}
```

Normaalisti törmäyksen tunnistus välittää peliobjektit kutsumalleen funktiolle siinä järjestyksessä, missä ne on tunnistuksessa määritelty. Kuitenkin jos törmäyksen tunnistus tehdään ryhmän (group) ja spriten välillä, on sprite aina järjestyksessä ensimmäinen. Siksi yllä kuvatussa funktiossa on parametrina asteroidin (a) lisäksi myös recycler-peliobjekti (r), vaikka sillä ei funktion kannalta ole mitään käyttöä.

Vaihtoehtoisesti asteroidien kierrätyksen olisi voinut toteuttaa myös määrittämällä asteroidi-peliobjektien checkWorldBounds-parametri todeksi (true) ja lisäämällä onOutOfBounds-tapahtuma (event) kutsumaan asteroideja kierrättävää funktiota:

```
asteroid.checkWorldBounds = true;
asteroid.events.onOutOfBounds.add(recycleAsteroid);
```

Tässä tapauksessa recycleAsteroid-funktio muokattaisiin ottamaan vastaan vain yksi parametri, eli asteroid-peliobjekti.

Yksi pelin toiminnoista oli kerätä talteen ammuslaatikoita. Ammuslaatikon ja raketin kohtaamiseen sovellettiin alustavasti samaa törmäyksen tunnistusta kuin muihinkin. Tämä kuitenkin osoittautui ongelmalliseksi törmäyksen sysätessä raketin aina pelialueen alareunaan.

Ongelman korjaamiseen kokeiltiin kahta vaihtoehtoa. Ensimmäisenä hyödynnettiin raketin bodyn ominaisuutta immovable:

```
rocket.body.immovable = true;
```

Antamalla tälle ominaisuudelle arvo tosi (true), törmääminen muihin peliobjekteihin ei liikuttanut rakettia. Vaihtoehtona tälle tarkasteltiin collide-metodin vaihtamista overlap-metodiin:

```
game.physics.arcade.overlap(rocket,crate,
collectCrate);
```

Toteutukseltaan overlap-metodi oli samanlainen kuin collide-metodi, mutta se tarkistaa vain peliobjektien päällekkäisyyttä, ei aiheuta niiden välillä fyysistä törmäystä.

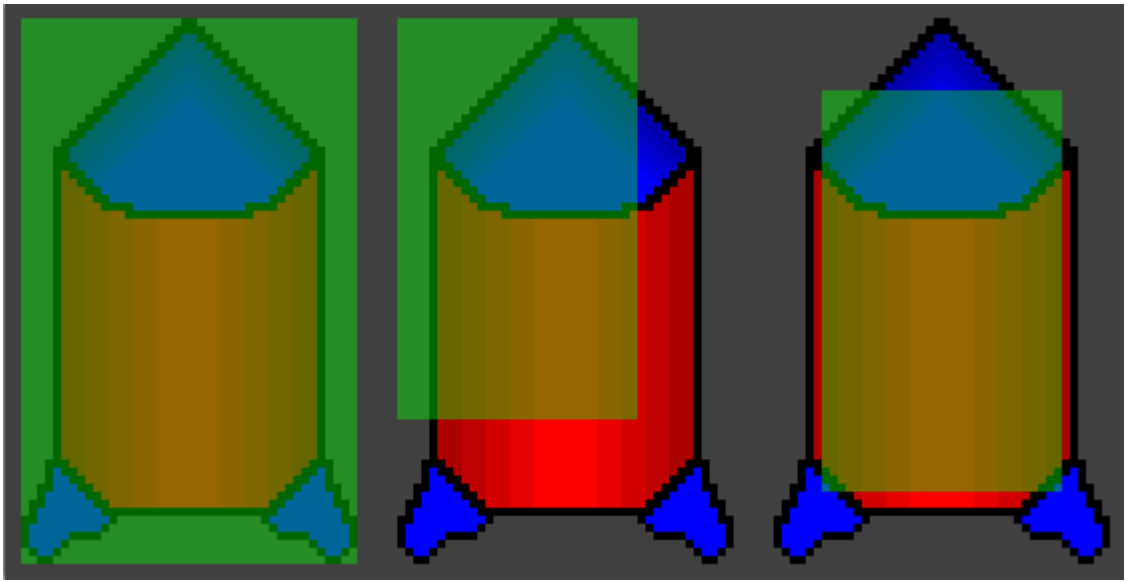
Peliä testattaessa törmäykset eivät olleet kovin täsmällisiä. Ongelman selvittämiseksi Phaserista löytyy kuitenkin toiminto, jolla saatiin spriten body näkyväksi:

```
game.debug.body(rocket);
```

Bodyn ollessa näkyvä, ongelman syyt olivat selvästi havaittavissa. Ensinnäkin spriten näkyvä osa, tässä tapauksessa raketti, ei kattanut koko spriten alaa. Tällöin peliobjektiin jäi pelissä näkymätön osa, joka kuitenkin tunnistaa törmäyksen. Tämä ongelma voitiin osittain korjata säätämällä bodyn kokoa:

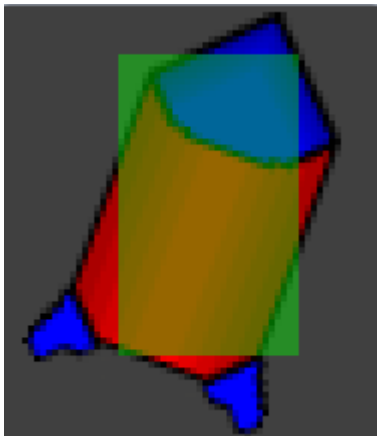
```
rocket.body.setSize(30,50);
```

Bodyn setSize-metodille syötettiin parametreiksi spriten mittoja pienemmät leveys (30) ja korkeus (50). Pienennetty body saadaan siirrettyä spriten keskelle joko muuttamalla spriten origo spriten keskelle tai syöttämällä setSize-metodille leveyden ja korkeuden lisäksi offsetX- ja offsetY-parametrit, jotka muuttavat bodyn sijaintia suhteessa spriteen. Bodyn koon ja sijainnin muokkaus on havainnollistettuna kuvassa 6.



Kuva 6. Spriten bodyn muokkaus

Törmäyksen tunnistuksen toinen ilmeinen ongelma liittyy siihen, että bodyn asento on sidottu x- ja y-akselien mukaiseksi. Vaikka spriten asentoa voidaan muuttaa, pysyy bodyn asento samana (kuva 7).



Kuva 7. Sprite-peliobjektin kallistuminen suhteessa bodyyn

Tätä ongelmaa ei oikeastaan Arcade-fysiikkamoottorin puitteissa voitu korjata. Vaihtoehtoina oli joko jättää spriten kääntäminen pelistä pois, muokata käytettävät kuvat muodoltaan mahdollisimman lähelle neliötä ongelman minimoimiseksi tai vaihtaa kattavamman törmäyksen tunnistuksen sisältävään fysiikkamoottoriin.

3.1.6 Partikkeliefektit

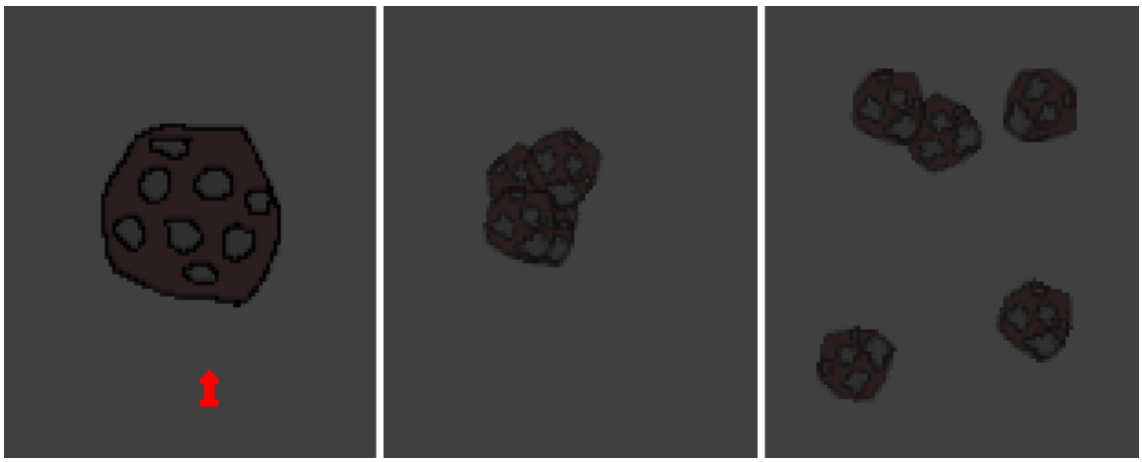
Viimeisenä tähän sovellukseen lisättiin asteroidi-peliobjektin ja ohjus-peliobjektin törmäykseen efekti kuvaamaan asteroidin hajoamista. Tämä toteutettiin lisäämällä peliin partikkeliemitteri (emitter), joka lähettää sijainnistaan partikkeliobjekteja:

```
emitter = game.add.emitter(0,0,50);
emitter.makeParticles('asterPart');
```

Emitterille voidaan määrittää parametreiksi x- ja y-koordinaatit sekä partikkelien maksimimäärä, joka vaikuttaa siihen kuinka monta partikkelia kyseisestä emitteristä voi olla kerrallaan näkyvissä pelissä.

Ohjuksen ja asteroidin törmätessä kutsuttavan funktion explodeAsteroid sisällä sijoitettiin emitteri kyseisen asteroidin kohdalle. Emitterin explode-metodilla saatiin emitteri lähettämään valittu määrä partikkeliobjekteja ympärilleen kuvaamaan asteroidin hajoamista (kuva 8).

```
emitter.x = target.body.x+25;
emitter.y = target.body.y+25;
emitter.explode(1500,5);
```



Kuva 8. Asteroidin hajoamista kuvaava partikkeliefekti

Emitterin explode-metodille annettiin parametreiksi lifespan eli elinikä (1500 ms) sekä quantity eli lukumäärä (5).

3.2 Ledge Jumper - peli

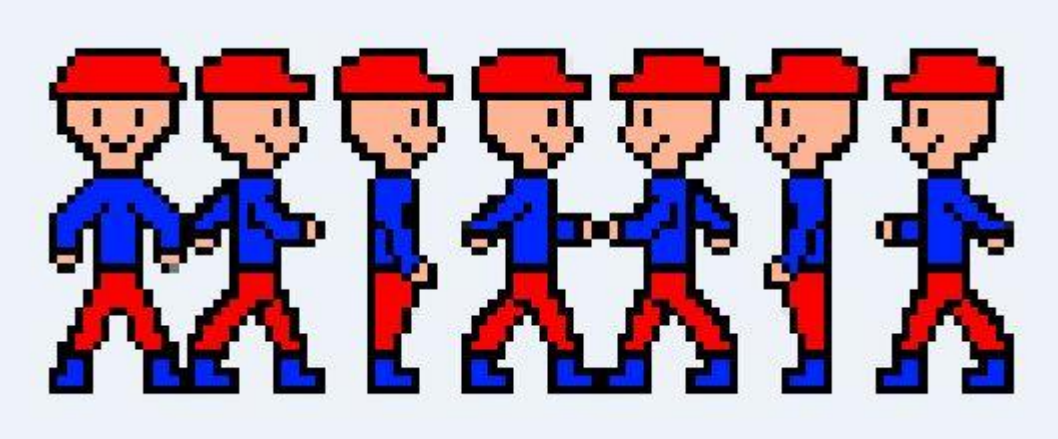
Toisessa esimerkkisovelluksessa tavoitteena oli testata pelihahmon animaation toteuttamista, fysiikkamoottorin hyödyntämistä pelihahmon liikkumiseen sekä äänien käyttöä pelissä. Pelissä oli tarkoitus myös testata sellaisen pelimaailman käyttöä, joka on suurempi kuin mitä ruudulla kerrallaan on näkyvissä. Tätä tarkoitusta varten suunniteltiin tasohyppelypeli, jossa ohjattava hahmo juoksee ja hyppii alustalta toiselle.

3.2.1 Pelihahmon animaatio

Ensimmäinen askel animaation lisäämisessä peliin oli sprite sheet assetin lisääminen:

```
game.load.spritesheet('player',
'assets/Player.png', 16, 40);
```

Sprite assetin lisäämisessä esiteltyjen avaimen (player) ja sijainnin (assets/Player.png) lisäksi sprite sheetin parametreiksi annettiin myös yksittäisen framen leveys (16) ja korkeus (40) pikseleinä. Kuvatiedosto jota käytettiin sprite sheetiin, on esitettynä kuvassa 9.



Kuva 9. Pelihahmon luomiseen käytetty sprite sheet

Tämä sprite sheet koostuu seitsemästä framesta. Ensimmäinen frame (0) kuvaa paikallaan seisovaa hahmoa, kolmea seuraavaa framea (1,2,3) käytetään animaatiossa pelaajan liikkuessa oikealle ja kolmea viimeistä (4,5,6) pelaajan liikkuessa vasemmalle.

Edellisen esimerkkisovelluksen tapaan peliin lisättiin peliobjekti pelaajan ohjaamalle hahmolle, aktivoitiin fysiikkamoottori, ja asetettiin objekti pysymään pelimaailman rajojen sisällä. Aiemmasta poiketen lisättiin vielä hahmon liikkumiseen käytettävät animaatiot:

```
player.animations.add('left',[4,5,6,5],8, true);
player.animations.add('right',[1,2,3,2],8, true);
```

Parametreiksi annettiin animaation nimi, käytettävät framet, animaation nopeus (fps, frames per second) sekä loop parametrille annettiin arvo tosi (true), jotta animaatiota toistetaan niin kauan kuin hahmoa liikutetaan.

Hahmon liikkuminen ja animaatiot yhdistettiin vielä näppäinkomentoihin ja hahmon pysähtyessä animaatio asetettiin pysähtymään sekä peliobjekti käyttämään sprite sheetin ensimmäistä framea:

```
if (cursors.left.isDown){
          player.body.velocity.x = -150;
          player.animations.play('left');
}
else if (cursors.right.isDown){
          player.body.velocity.x = 150;
          player.animations.play('right');
}
else{
          player.animations.stop();
          player.frame = 0;}
```

3.2.2 Pelihahmon liikkuminen pelimaailmassa

Yllä kuvatussa ohjelmakoodissa pelihahmon liikkuminen tapahtuu muuttamalla player-peliobjektin rungon nopeutta x-akselilla vasemman tai oikean suuntanäppäimen ollessa painettuna. Tällä saatiin aikaan karkea liikkuvuus, jossa hahmo on joko paikallaan tai liikkuu maksiminopeutta näppäinkomentojen mukaiseen suuntaan.

Tasohyppelypelissä oleellinen osa pelin mekaniikkaa on hahmon hyppääminen. Tämän testaamista varten pelimaailmaan luotiin peliobjekteja, joiden päällä pelihahmon oli tarkoitus kulkea. Ensimmäinen tällainen objekti sijoitettiin pelaajan alle pelimaailman alkuun ja loput tasaisin välimatkoin vaihtelevalle korkeudelle:

```
platforms = game.add.group();
platforms.enableBody = true;

var ledge = platforms.create(0, 350, 'ground');
ledge.body.immovable = true;
```

```
for (i = 0, x = 300; i < 8; i++, x+=300) {

          ledge   =   platforms.create(x,(Math.random()
          *150) + 250, 'ground');

          ledge.body.immovable = true;

}
```

Näiden peliobjektien bodyn immovable-ominaisuudelle annettiin myös arvo tosi, jotta törmäykset pelihahmon kanssa eivät siirtäisi niitä.

Jotta pelihahmo saatiin kulkemaan näiden alustojen päällä, piti peliin lisätä vielä kaksi toimintoa. Ensimmäisenä asetettiin fysiikkamoottorin painovoimaominaisuus (gravity). Painovoima voitiin asettaa koskemaan koko pelimaailmaa:

```
game.physics.arcade.gravity.y = 400;
```

Tämä tosin aiheutti sen, että pelimaailman kaikki objektit putosivat alas. Ongelman korjaaminen oli mahdollista asettamalla niiden objektien, joiden haluttiin pysyvän paikallaan, allowGravity-ominaisuus epätodeksi:

```
ledge.body.allowGravity = false;
```

Tämän esimerkkisovelluksen tapauksessa oli kuitenkin vain yksi liikkuva peliobjekti, joten yksinkertaisempi ratkaisu oli määrittää painovoima koskemaan ainoastaan tätä kyseistä peliobjektia:

```
player.body.gravity.y = 400;
```

Tällöin painovoima ei koske pelimaailman muita fysiikanmallinnuksen vaikutuksen alaisia objekteja.

Seuraavaksi lisättiin törmäyksen tunnistus pelihahmon ja alustojen välille, jotta hahmo ei putoaisi niistä läpi:

```
game.physics.arcade.collide(player, platforms);
```

Edelliseen esimerkkisovellukseen verrattuna törmäyksen tunnistus oli nyt yksinkertaisempi. Tässä tapauksessa ei ollut tarpeen määrittää kutsuttavaa funktiota, ainoastaan peliobjektit joiden välillä törmäys tuli tunnistaa.

Pelihahmon hyppääminen toteutettiin lisäämällä näppäinkomento ylös-suuntanäppäimelle:

```
if (cursors.up.isDown && player.body.touch
ing.down){

          player.body.velocity.y = -350;

}
```

Näppäimen ollessa painettuna ja peliobjektin alareunan ollessa kosketuksissa alustan kanssa, peliobjektin nopeutta y-akselilla muutettiin. Pelihahmon painovoima vaikuttaa tämän nopeuteen y-akselilla. Näin hypyn nopeus hidastuu ja hahmo putoaa lopulta alas.

3.2.3 Pelimaailma

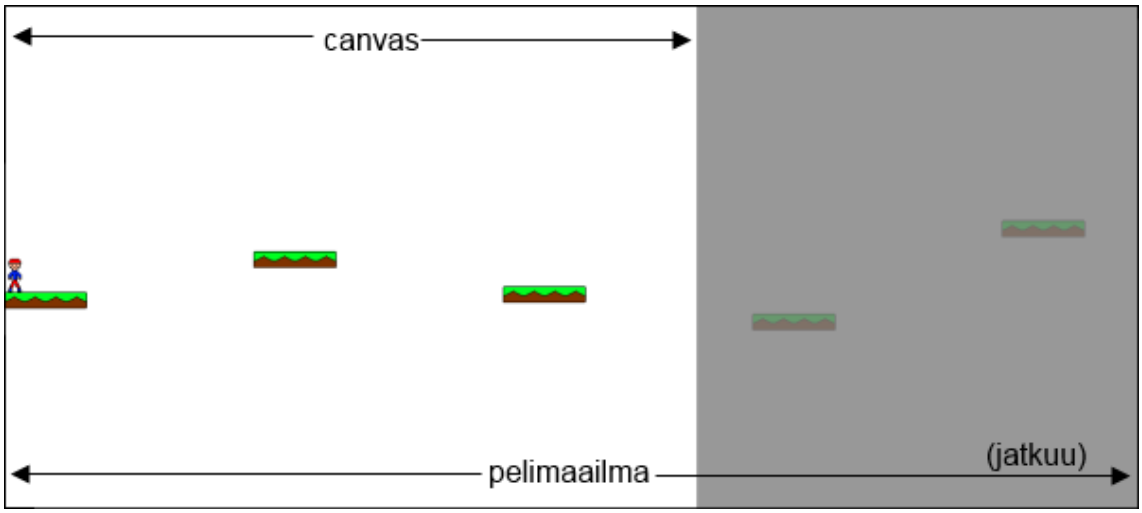
Tähän esimerkkisovellukseen haluttiin tehdä pelimaailma joka on rajallinen, mutta kuitenkin suuremi kuin mitä ruudulla näkyy. Ruudulla näkyvän alueen koko oli jo määritetty ohjelmakoodin alussa Phaser.Game –objektia luotaessa:

```
var game = new Phaser.Game(800, 600, Phaser.AUTO,
'', { preload: preload, create: create, update:
update });
```

Canvas jolle peli piirretään, määritettiin siis 800 pikseliä leveäksi ja 600 pikseliä korkeaksi. Tämän lisäksi asetettiin pelimaailman rajat create-funktiossa:

```
game.world.setBounds(0, 0, 2400, 600);
```

Parametreina pelimaailman rajoja asettaessa ovat vasemman yläkulman x- ja y-koordinaatit (0,0) sekä leveys (2400) ja korkeus (600) pikseleinä. Pelimaailmaa ja sen näkyvissä olevaa osaa havainnollistetaan kuvassa 10.



Kuva 10. Kuvaus canvasia suuremmasta pelimaailmasta

Seuraavaksi pyrittiin saamaan pelin näkymä seuraamaan pelihahmoa. Se onnistui Phaserin kamera-objektin avulla. Phaser luo automaattisesti kamera-objektin, joka toimii pelaajan näkymänä pelimaailmaan ja vastaa kooltaan canvasia. Kameran follow-metodia käyttämällä saatiin kamera seuraamaan haluttua peliobjektia:

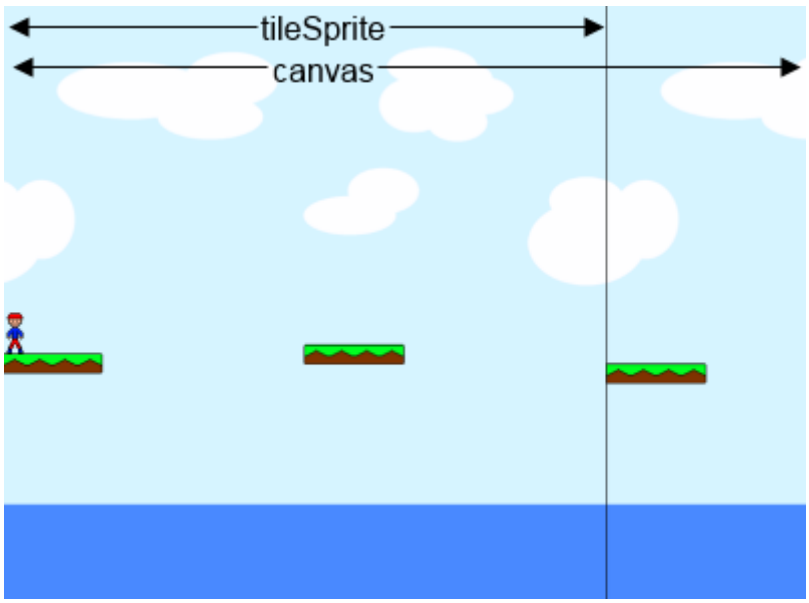
```
game.camera.follow(player);
```

Follow-metodin parametriksi syötettiin peliobjekti, jota halutaan kameran seuraavan (player). Näin kamera liikkuu pitäen pelihahmon kuvan keskellä, rajoittuen kuitenkin pelimaailman rajoihin.

Pelimaailmalle tehtiin vielä taustamaisema. Sen sijaan, että olisi tehty pelimaailman kokoa vastaava tausta, käytettiin TileSprite-peliobjektia (kuva 11):

```
game.add.tileSprite(0,0, 2400, 600, 'background');
```

Parametreiksi annettiin sijainti x- ja y-akselilla, leveys, korkeus sekä käytettävän assetin avain.



Kuva 11. Tile sprite

TileSprite-objektille annetut korkeus ja leveys ovat pelimaailman mitat (2400x600). Käytetyn kuvatiedoston koko on pienempi kuin pelinäkymä, mutta tileSprite toistuu automaattisesti täyttäen annetun pinta-alan.

3.2.4 Pelin äänet

Viimeisenä osana tätä esimerkkisovellusta testattiin äänien käyttöä pelissä. Peliin oli tarkoitus lisätä ääniefektit hyppäämiselle ja veteen putoamiselle sekä taustamusiikki.

Kun tarvittavat assetit oli ladattu preload-funktiossa, voitiin niistä luoda ääni-objektit:

```
jump = game.add.audio('jump');
splash = game.add.audio('splash');
bgMusic = game.add.audio('bg');
```

Jotta taustamusiikki (bgMusic) saatiin toistuvaksi, annettiin sen loop-ominaisuudelle arvoksi tosi:

```
bg.loop = true;
```

Lisäksi ääniefekti asetettiin soimaan halutussa tilanteessa ääniobjektin play-metodilla:

```
jump.play();
```

4 YHTEENVETO

Phaser-peliohjelmointikirjaston käyttömahdollisuudet peliohjelmoinnissa ovat huomattavasti laajemmat kuin mitä tässä opinnäytetyössä on käsitelty. Aihe pyrittiin rajaamaan siten, että siinä käsitellään toiminnallisuuksia, joilla pääsee pelin kehityksen alkuun Phaserin puitteissa.

Esitellyt testipelit testasivat tämän peliohjelmointikehyksen fysiikanmallinnusta pelihahmojen liikkumisen ja törmäyksen tunnistuksen osalta, pelihahmon animaation käyttöä sekä ääni- ja kuvatiedostojen hyödyntämistä pelissä. Fysiikanmallinnuksessa käytettiin Arcade-fysiikkamoottoria, joka on Phaserin kolmesta vaihtoehdosta kevyin ja nopein. Se toimi testipeleissä melko hyvin, vaikka siinä ilmenikin pieniä puutteita törmäyksen tunnistuksessa sekä peliobjektin kääntämisessä.

Valmiin pelimoottorin käyttämisessä oli selvät etunsa ja siitä on etenkin kokemattomalle ohjelmoijalle huomattava apu. Kun toiminnallisuudet ovat pelikehyksessä valmiina, on pelien tekeminen helppo aloittaa suppeammallakin ohjelmointikielen tuntemuksella. Itse en ollut tutustunut oikeastaan lainkaan JavaScript-ohjelmointikieleen ennen kuin aloitin opinnäytetyöprojektin.

Kattavan dokumentoinnin merkitys tuli projektin aikana selkeästi esille. Phaserin sisältämien perustoimintojen käyttö oli dokumentoinnin ja esimerkkien avulla selkeää ja kaikki käyttämäni toiminnot toimivat toivotulla tavalla. Ainoa tilanne, jossa ilmeni ongelma, jolle en löytänyt järkevää selitystä, oli ääniä käyttäessä. Mutta sekin korjaantui, kun vaihdoin Phaserin 2.2.2 version projektin aikana julkaistuun 2.3.0 versioon.

Tämä opinnäytetyö havainnollisti minulle itselleni miten pelimoottorin hyödyntäminen helpottaa ja nopeuttaa pelinkehitysprosessia. Sen tuloksia voisivat mahdollisesti hyödyntää myös muut pelien tekemisestä kiinnostuneet.

Aiheena Phaser oli mielenkiintoinen ja varmasti jatkan perehtymistä siihen harrastusmielessä. Jatkossa olisi kiinnostavaa suunnitella ja toteuttaa valmis pelikokonaisuus Phaserilla ja mahdollisesti myös julkaista se.

LÄHTEET

Game engine. Wikipedia. Viitattu 27.4.2015. http://en.wikipedia.org/wiki/Game_engine

Kanber, B. 2012. How physics engines work. Viitattu 29.5.2015. http://buildnewgames.com/gamephysics/

Khronos Group 2011. Getting started. Viitattu 29.5.2015.https://www.khronos.org/webgl/wiki/Getting_Started

Photon Storm Ltd. 2015. Phaser 2.3.0 API Docs. Viitattu 27.5.2015. http://phaser.io/docs

Photon Storm Ltd. 2013. Tutorial: Making your first Phaser game. Viitattu 29.5.2015. http://phaser.io/tutorials/making-your-first-phaser-game

Rendering (computer graphics). Wikipedia. Viitattu 27.4.2015. http://en.wikipedia.org/wiki/Rendering_%28computer_graphics%29

W3Schools. JavaScript Tutorial. Viitattu 1.6.2015. http://www.w3schools.com/js/default.asp

Asteroid Dodger – pelin ohjelmakoodi

```
<!DOCTYPE html>
<head>
    <meta charset="UTF-8" />
    <title>Asteroid Dodger</title>
    <script type="text/javascript" src="../phaser.js"></script>
    <style type="text/css">
        body {
            margin: 0;
        }
    </style>
</head>

<body>
    <script type="text/javascript">
        var game = new Phaser.Game(800, 600, Phaser.AUTO, '', { preload: preload, create: create, update: update });

        function preload() {
            game.load.image('space', 'assets/Space.png');
            game.load.image('asteroid', 'assets/Asteroid.png');
            game.load.image('recycler', 'assets/Recycler.png');
            game.load.image('rocket', 'assets/Rocket.png');
            game.load.image('asterPart', 'assets/AsteroidPart.png');
            game.load.image('crate', 'assets/Crate.png');
            game.load.image('missile', 'assets/Missile.png');
        }

        var rocket;
        var asteroids;
        var recycler;
        var cursors;
        var space1;
        var space2;
        var crate;
        var missiles;
        var emitter;
        var ammo = 5;
        var timer;
        var text;
        var readyToFire = true;

        function create() {
            game.physics.startSystem(Phaser.Physics.ARCADE);

            timer = game.time.create();
            timer.loop(10000, spawnCrate, this);
            timer.start();

            space1 = game.add.sprite(0, 0, 'space');
            game.physics.arcade.enable(space1);
            space1.body.velocity.y = 150;

            space2 = game.add.sprite(0, -650, 'space');
            game.physics.arcade.enable(space2);
            space2.body.velocity.y = 150;


            recycler = game.add.sprite(0,game.world.height + 50, 'recycler');
            game.physics.arcade.enable(recycler);
            recycler.body.immovable = true;

            missiles = game.add.group();
            missiles.enableBody = true;
            missiles.createMultiple(15, 'missile');
            missiles.setAll('checkWorldBounds', true);
            missiles.setAll('outOfBoundsKill', true);
```

```
    rocket = game.add.sprite(400, game.world.height - 150, 'rocket');
    rocket.anchor.set(0.5, 0.5);
    game.physics.arcade.enable(rocket);
    rocket.body.immovable = true;
    rocket.body.setSize(30,60);
    rocket.body.collideWorldBounds = true;

    asteroids = game.add.group();
    asteroids.enableBody = true;

    for (var i = 0; i < 12; i++){
        var asteroid = asteroids.create(Math.random() * 750, Math.random() * (-300), 'asteroid');
        asteroid.body.setSize(40,40,5,10);
        asteroid.body.velocity.y = 170 + Math.random() * 100;
    }

    cursors = game.input.keyboard.createCursorKeys();

    emitter = game.add.emitter();
    emitter.makeParticles('asterPart');

    text = game.add.text(0, 0, 'Ammo: '+ ammo, { font: '24px Arial', fill: '#fa0000', align: 'center' });
}

function update() {

    if (space1.y >= 650) {
        space1.y = -650;
    }

    if (space2.y >= 650) {
        space2.y = -650;
    }

    game.physics.arcade.collide(rocket, asteroids, explodeAsteroid);
    game.physics.arcade.collide(asteroids, recycler, recycleAsteroid);
    game.physics.arcade.collide(missiles, asteroids,explodeAsteroid);
    game.physics.arcade.collide(crate, recycler, destroyCrate);
    game.physics.arcade.overlap(rocket, crate, collectCrate);

    rocket.body.velocity.x = 0;
    rocket.angle = 0;

    if (cursors.left.isDown){
        rocket.body.velocity.x = -300;
        rocket.angle = 340;
    }
    else if (cursors.right.isDown){
        rocket.body.velocity.x = 300;
        rocket.angle = 20;
    }
    if (cursors.up.isDown && readyToFire == true && ammo > 0) {
        shoot();
        readyToFire = false;
    }
    if (cursors.up.isUp) {
        readyToFire = true;
    }

}

function recycleAsteroid (r, a) {
    a.reset(a.x = Math.random() * 750, a.y = Math.random() * (-300));
    a.body.velocity.y = 170 + Math.random() * 100;
}

function explodeAsteroid(projectile, target){
    emitter.x = target.body.x+25;
    emitter.y = target.body.y+25;
    emitter.explode(4500,5);
    target.body.x = Math.random() * 750;
    target.body.y = Math.random() * (-300);
    target.body.velocity.y = 170 + Math.random() * 100;
```

```
            target.body.velocity.x = 0;
            projectile.kill();
        }

        function spawnCrate(){
            crate = game.add.sprite(200 + Math.random() * 575, -50, 'crate');
            game.physics.arcade.enable(crate);
            crate.body.velocity.y = 150;
        }

        function collectCrate(){
            destroyCrate();
            ammo += 5;
            text.setText('Ammo: ' + ammo);
        }

        function destroyCrate(){
            crate.destroy();
        }

        function shoot() {
            if (missiles.countDead() > 0) {
                var missile = missiles.getFirstDead();
                missile.reset(rocket.x, rocket.y -8);
                missile.body.velocity.y = -200;
                ammo--;
                text.setText('Ammo: ' + ammo);
            }
        }
    </script>
</body>
</html>
```

Ledge Jumper – pelin ohjelmakoodi

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="UTF-8" />
   <title>Ledge Jumper</title>
   <script type="text/javascript" src="../phaser.js"></script>
   <style type="text/css">
      body {
         margin: 0;
      }
   </style>
</head>

<body>
   <script language="JavaScript" type="text/javascript">
      var game = new Phaser.Game(800, 600, Phaser.AUTO, '', { preload: preload, create: create, update: update });

      function preload(){
         game.load.image('ground', 'assets/Ledge.png');
         game.load.image('water', 'assets/Water.png');
         game.load.image('background', 'assets/Background.png');
         game.load.spritesheet('player', 'assets/Player.png',16,40);
         game.load.image('drop', 'assets/Drop.png');
         game.load.audio('jump', '../Sounds/Jump.mp3');
         game.load.audio('splash', '../Sounds/Splash.mp3');
         game.load.audio('bg', '../Sounds/bgMusic.mp3');
      }

      var water;
      var platforms;
      var player;
      var cursors;
      var jump;
      var splash;
      var bgMusic;
      var emitter
      var drop;
      var drown = false;

      function create(){
         game.world.setBounds(0, 0, 2400, 600);
         game.physics.startSystem(Phaser.Physics.ARCADE);

         game.add.tileSprite(0, 0, 2400, 600, 'background');

         platforms = game.add.group();
         platforms.enableBody = true;

         var ledge = platforms.create(0, 350, 'ground');
         ledge.body.immovable = true;

         for (i = 0, x = 300; i < 8; i++, x+=300) {
            ledge = platforms.create(x, (Math.random() *150) + 250, 'ground');
            ledge.body.immovable = true;
         }

         player = game.add.sprite(5, 300, 'player');
         game.physics.arcade.enable(player);
         player.body.gravity.y = 400;
         player.body.collideWorldBounds = true;

         player.animations.add('left', [4, 5, 6], 10, true);
         player.animations.add('right', [1, 2, 3], 10, true);

         water = game.add.sprite(0,game.world.height - 100, 'water');
```

```
        game.physics.arcade.enable(water);
        water.body.immovable = true;

        emitter = game.add.emitter(0, 0, 100);
        emitter.makeParticles('drop');
        emitter.setYSpeed(-100, -200);
        emitter.gravity = 400;

        cursors = game.input.keyboard.createCursorKeys();

        game.camera.follow(player);

        jump = game.add.audio('jump');
        splash = game.add.audio('splash');
        bgMusic = game.add.audio('bg');
        bgMusic.volume = 0.5;
        bgMusic.loop = true;
        bgMusic.play();

    }

    function update(){
        game.physics.arcade.collide(player, platforms);
        if (player.exists == true) {
            if (cursors.left.isDown){
                player.body.velocity.x = -150;
                player.animations.play('left');
            }
            else if (cursors.right.isDown){
                player.body.velocity.x = 150;
                player.animations.play('right');
            }
            else {
                player.body.velocity.x = 0;
                player.animations.stop();
                player.frame = 0;
            }

            if (cursors.up.isDown && player.body.touching.down){
                player.body.velocity.y = -350;
                jump.play();
            }
        }

        game.physics.arcade.collide(player, water, drowned);
    }

    function drowned(){
        if (drown == false) {
            drown = true;
            splash.play();
            emitter.x = player.x;
            emitter.y = 500;
            emitter.explode(1500,20);
            player.kill();
        }
    }
  </script>
</body>
</html>
```