TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Handre Elias

Simulatsioonmäng veebilehitsejas

Bakalaurusetöö (9 ECTS)

Juhendaja: Dr. Benson Muite
Kaasjuhendaja: Prof. Eero Vainikko

Tartu 2016

Simulatsioonimäng veebilehitsejas

Lühikokkuvõte:
Antud töö eesmärgiks on luua mäng, mis simuleerib teatud füüsilist nähtust ja mis jookseb veebilehitsejas. Mäng simuleerib veelainet, mis põhineb 1-D lainevõrrandil ning mille graafika on kuvatud WebGL'i poolt. Mängija saab mängu mängida nii hiire kui ka veebikaameraga. Mängija peab tekitama veelaineid, et liigutada paat teatud punktidesse. Hiirega juhtimine toimib läbi hiirevajutuste ning veebikaameraga juhtimine toimib käeliigutuste põhjal. Töös antakse sissejuhatus WebGL'i, seletatakse lahti 1-D lainevõrrand, leitakse sellele lahendus ja räägitakse kuidas mäng seda rakendab. Samuti tuleb juttu meetodist, millega saab video põhjal liikumist tuvastada.

Võtmesõnad: WebGL, 1-D lainevõrrand, simulatsioon, pilditöötlus

CERCS: P130 Funktsioonid, diferentsiaalvõrrandid; P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Simulation Game in a web browser

Abstract: The goal of this thesis is to create a game in which a physical phenomenon is simulated and run in a browser. In the game, water waves are simulated based on 1-D wave equation. The graphics of the game are displayed using WebGL. The game can be played with a mouse or a webcam. The player must make water waves to move a boat to certain locations in the water. The mouse input is used to make water waves at certain locations. The webcam allows for detection of hand movements and makes water waves based on the hand movements. The thesis will give an introduction to WebGL, explains the wave equation, solves it and explains how the game applies it. A method for detecting movement from video is also given.

Keywords: WebGL, 1-D, wave equation, simulation, image processing

CERCS: P130 Functions, differential equations; P170 Computer science, numerical analysis, systems, control

Sisukord

1 Sissejuhatus

Töö eesmärgiks on luua lõbus mäng mis simuleerib füüsikalist nähtust. Graafika kuvamiseks on kasutuses WebGL, mis võimaldab riistvara toelist 3-D graafikat otse veebibrauseris. Võib mõelda, et veebibrauseris simulatsiooni tegemine on halb idee, sest see kipub olema aeglane, kuid tänu WebGL'le võib mõelda vastupidi - see on väga hea idee. Otse veebilehitsejas simulatsiooni tegemine on kasutajale palju mugavam, sest ole vaja midagi alla laadida ja installida, kõik vajalik on hiirekliki kaugusel. WebGL'i tugi on olemas isegi nutiseadmete brauserites, seega on võimalik simulatsiooni jooksutada kaasaskantavates nutiseadmetes.

WebGL'i programmeerimine toimib läbi JavaScript'i ning seega on mäng programmeeritud just selles keeles. Mäng simuleerib veelaineid, mis põhineb 1-D lainevõrrandil. Kuigi see pole kõige täpsem mudel veelainete simuleerimiseks, annab see rahuldava tulemuse. Mängija eesmärk on paati läbi veelainete liigutada teatud punktidesse. Mängija peab tundma õppima 1-D lainevõrrandi iseärasusi ning selle mõju paadile, et saada mänguga hakkama.

WebGL on suhteliselt uus tehnoloogia ja töös annan ma sellele väikse sissejuhatuse ning toon näiteid kuidas sellega graafikat kuvada. Toon näiteid mängu lähtekoodi põhjal ja selgitan miks ma just nii süsteemi püsti panin. Käeliigutuste registreerimine ja kasutamine mängu juhtimiseks on omaette väljakutse ning siin on see samuti põhjalikult kirjeldatud. Räägin pilditöötlusest ja sealt mängu jaoks vajaliku informatsiooni välja lugemisest ning annan näiteid mängu lähtekoodist.

Protsessis valminud mäng on lõbus viis uurida 1-D lainevõrrandit. Töö annab sissejuhatuse WebGL'i ja julgustab seda kasutama graafika kuvamisel, ning annab lugejale aimu kuidas füüsikalist nähtust simuleerida. Tööga on kaasas loodud mängu lähtekood koos selgitavate kommentaaridega.

Töö teises peatükis annan ülevaate mängu loomiseks kasutatud tehnoloogiatest. WebGL'st tuleb täpsemalt juttu kolmandas peatükis. Neljas peatükk on pühendatud 1-D lainevõrrandile. Seal seletan lahti lainevõrrandi, leian sellele lahendi ning räägin kuidas mäng seda simulatsiooniks ära kasutab. Viiendas peatükis seletan lahti mängu erinevad aspektid. Samuti räägin kasutaja hiire ning veebikaamera sisendi töötlemise protsessidest.

2 Kasutatud tehnoloogiad

Käesolevas peatükis annan ülevaate mängu loomiseks kasutatud tehnoloogitest.

2.1 jQuery

Nagu räägib jQuery koduleht: "jQuery on kiire, väike funktsiooniderohke JavaScript'i teek. See teeb toiminguid nagu HTML dokumendi sees liikumine ja manipuleerimise, sündmuste töötlemise, animatsiooni ja Ajax'i palju lihtsamaks kergesti kasutatava API'ga, mis töötab paljudes lehitsejates. Kombineerides mitmekülgsuse ja laiendatavuse, on jQuery muutnud viisi kuidas miljonid inimesed kirjutavad JavaScript'i."[1]

jQuery teek tegeleb mängus kasutaja sisendi töötlemisega, ning HTML elementide manipuleerisega. On võimalik sada hakkama seda ka kasutamata, kuid lisasin selle mugavuse eesmärgil.

2.2 glMatrix

"glMatrix on disainitud teostama vektori ja maatriksi operatsioone rumalalt kiiresti! Käsitsi häälestades igat funktsiooni maksimaalseks jõudluseks ja julgustades tõhusaid kasutusmustreid läbi API konventsioonide, aitab glMatrix sul saada JavaScript'i mootorist maksimumi."(glMatrixi koduleht) [2].

Maatriksite ja vektorite abil saame me mängumaailmas objekte pöörata, skaleerida ning liigutada. Ilmselgelt on see vajalik, sest ilma liikuvate objektideta poleks mäng lõbus.

2.3 WebGL

WebGL (Web Graphics Library) võimaldab rikkalikku riistvara kiirendatud 3D-graafikat HTML5 veebilehitsejates ilma pistikprogrammiteta, sidudes JavaScript'i OpenGL® ES 2.0'ga. Tänu sellele on veebiarendajatel juurdepääs OpenGL-klassi graafikale läbi JavaScript'i ning võimalus vabalt segada 3D-graafikat HTML-ga. [3] WebGL on toetatud 84% maailmas kasutuses olevates lehitsejates. [4] Täpsema ülevaate WebGL'st teeme järgmises peatükis.

3 WebGL

Käesolevas peatükis kirjeldan WebGL'i täpsemalt ja toon koodinäiteid.

3.1 Kuidas see töötab?

WebGL'is on olemas spetsiaalsed programmid mida kutsutakse varjutajateks (*shader*). Varjutajad on kirjutatud keeles **GLSL** (OpenGL ES Shading Language). Varjutajad jooksevad otse graafikaprotsessori peal ning neid on kahte liiki: tipu (*vertex*) ja fragmendi varjutaja. Tipu varjutaja jookseb iga joonistava objekti tipu peal ning fragmendi varjutaja jookseb iga elemendi piksli peal. Lihtne tipu varjutaja näeb välja selline:

```
attribute vec2 aVertex;
void main(void) {
 gl_Position = vec4(aVertex, 1.0, 1.0);
}
```

ning lihtne fragmendi varjutaja:

```
void main(void) {
 gl_FragColor = vec4(1, 0, 0, 1);
}
```

On näha kahte spetsiaalset muutujat nimega *gl_Position* ja *gl_FragColor*. Varjutajate eesmärk ongi nende muutujate seadmine. Mingi kujutise joonistamiseks ongi ainult vaja piksli asukohta ning värvi. Nende kahe varjutajate koostööl tekibki graafika. WebGL'i programmeerija eesmärk ongi varjutajatele andmete sisse andmine ning varjutajatele öelda kuhu midagi joonistada. Varjutajatele andmete sisse andmisest ning nende programmeerimisest tuleb juttu hiljem.

3.2 WebGL ülesseadmine

WebGL'i on lehitsejas üpris lihtne üles seada. Vaatleme järgnevat koodi:

```
<canvas id="container"></canvas>
```

```
var canvas = document.getElementById("container");
var gl = canvas.getContext("webgl");
```

WebGL toimetab HTML'i *canvas* elemendi peal ning see peab olema HTML dokumendis esitatud. *Canvas* ise graafikat luua ei oska, see on lihtsalt konteiner graafika jaoks - sealt ka tema nimi (lõuend). Võib mõelda, et *canvas* on paber ja WebGL on kunstnik.

Nagu näha, JavaScript'is WebGL'i konteksti saamine on väga lihtne. Algul tuleb leida viide *canvas* elemendi peale mis annab meile meetodi "kunstniku" saamiseks. On soov, et "kunstnik" oleks WebGL ja seega anname meetodile sisendi

"webgl". *Canvas* elemendil on ka olemas kontekst "2d", mis on mõeldud 2-D graafika kuvamiseks, kuid antud töös me seda ei kasuta.

On oht, et WebGL'i konteksti küsimine annab tagasi tühja viite. Veebilehitsejad Internet Explorer ja Edge tahavad, et WebGL'i konteksti pärimine käiks argumendiga "experimental-webgl". Kui pärida WebGL'i mõlemat moodi ja tagasi saab ikka tühja viite, ei toeta kasutaja brauser WebGL'i ning võiks kasutajale teada anda, et tema brauser on aegunud.

Kui WebGL'i kontekst on käes on vaja luua programm, mis koosneb tipu ja fragmendi varjutajast. See protsess näeb välja selline:

```
var fragmentSource = $('#fragment-shader').html();
var vertexSource = $('#vertex-shader').html();

var shader = gl.createProgram();
var vertexShader = gl.createShader(gl.VERTEX_SHADER);
var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, fragmentSource);
gl.compileShader(fragmentShader);

gl.shaderSource(vertexShader, vertexSource);
gl.compileShader(vertexShader);

gl.attachShader(shader, vertexShader);
gl.attachShader(shader, fragmentShader);
gl.linkProgram(shader);
```

Fragmendi ja tipu varjutajate lähtekood tuleb anda sõnena. Mängu loomisprotsessis sai lähtekood lisatud otse HTML dokumendi *<script>* elemendi alla ja kasutasin jQuery't, et see lähtekood sõnena kätte saada. Lähtekoodi *<script>* elemendi alla lisades tuleb kindlasti, tema tüüp muuta nii, et veebibrauser ei üritaks seda käivitada.

Nüüd, kui meil on programm koos tipu ja fragmendi varjutajatega olemas, peame WebGL'ile teada andma, et seda programmi tuleb joonistamiseks kasutada:

```
gl.useProgram(shader);
```

Tüüpilised WebGL'i rakendused kasutavad joonistamiseks mitmeid erinevaid varjutaja programme ja selle meetodiga ütleme, millist programmi kasutada.

3.3 WebGL'is joonistamine

Kui meil on olemas varjutaja programm, saame läbi WebGL'li anda sellele juhiseid joonistamiseks. Toome näite järgmise tipu varjutaja põhjal:

```
attribute vec2 aVertex;
void main(void) {
 gl_Position = vec4(aVertex, 1.0, 1.0);
}
```

Nagu näha on varjutajas muutuja *aVertex* mis on tüüpi *vec2* ja mille ees on *attribute*. *Attribute* tähendab, et sellele väärtus tuleb **puhvrist** ning see on erinev iga tipu puhul. Puhvrite kaudu saab varjutajatele ette anda väärtusi, mis muutuvad iga tipu korral. Puhvri loomine on lihtne:

```
var buffer = gl.createBuffer();
```

Kui puhver on loodud, tuleb WebGL'le teada anda, et tahame toimetada just selle puhvriga:

```
// gl.ARRAY_BUFFER = puhver muudab attribuute
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
```

Nüüd saame puhvri andmeid muuta:

```
var objectVertices = [-1.0,  -1.0, 1.0,  -1.0, -1.0, 1.0,];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(objectVertices),
    gl.STATIC_DRAW);
```

Puhver tahab, et andmed oleks *Float32Array* tüüpi ning, eelmise meetodi viimane parameeter ütleb, et puhvri andmeid enam ei muudeta ja seega WebGL oskab mälu optimeerida. Veel oleks vaja WebGL'le öelda, et millisele atribuudile see puhver vastab. Kõigepealt leiame selle atribuudi programmist üles:

```
attribLoc = gl.getAttribLocation(shader, 'aVertex');
// antud atribuut tuleb lubada
gl.enableVertexAttribArray(attribLoc);
```

ja nüüd anname WebGL'le märku, et toimetuses olev puhver annab oma andmed sellele atribuudile:

```
gl.vertexAttribPointer(attribLoc, 2, gl.FLOAT, false, 0, 0);
```

Meetodi esimene parameeter on atribuudi asukoht. Teine parameeter näitab mitu elementi võtta sellest massiivist mis puhvrile sisse sai antud. Kuna tipu varjutaja atribuut oli tüüpi *vec2*, on vaja võtta 2 elementi, millest saab tipu varjutaja poolest töödeldav tipp. Kolmas parameeter näitab andmetüüpi, neljas märgib kas andmeid tuleks normaliseerida või mitte (muuta vahemikku [-1,1]). Viies märgib mitmendast elemendist alustada ning viimane märgib mitu baiti on elementide vahel. Märkides selle 0-ga oskab WebGL seda ise määrata. Nüüd lõpuks saame välja kutsuda järgmise meetodi, mis joonistab kolmnurga:

```
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

Meetodi esimene parameeter märgib joonistamisviisi, teine nihet algusest ning viimane märgib mitu tippu arvesse võtta. Joonistamisviise on mitmeid, ning igal on oma kasutusvaldkonnad. Tulemus on kolmnurk mille värv on määratud fragmendi varjutaja poolt.

WebGL'i koordinaadid on alati vahemikus [-1, 1]. Kui puhvri andmetes on mingi tipp, mis läheb sellest vahemikust välja, pole seda tippu lihtsalt näha. Kuna

meie kolmnurga tipud on määratud WebGL'i koordinaatide äärtes, on kolmnurk sama suur kui *canvas* element. Tipu varjutajas saab objekti suurust ning asukohta vastava matemaatikaga määrata.

Varjutajatele saab veel andmeid anda *uniform*'de kujul. *Attribute* muutus iga tipu puhul, kuid *uniform* on iga tipu puhul sama. Näiteks kui fragmendi varjutajas on olemas muutuja *"unifrom vec3 uColor"* saab sellele väärtusi anda päris lihtsalt. Tuleb leida selle muutuja asukoht:

```
var colorLoc = gl.getUniformLocation(shader, 'uColor');
```

ning selle muutmiseks kasutada meetodit

```
gl.uniform3fv(colorLoc, colorValue);
```

Niimoodi saabki WebGL'ga graafikat luua.

4 Lainevõrrand

Lainevõrrand on väga oluline evolutsiooniline mudel ning seda kasutavad laialdaselt füüsikud ja insenerid, et kirjeldada erinevate lainete levimist [5]. Kuna mängus on vaja simuleerida veelaineid, sobib see võrrand, täpsemini summutatud (*damped*) lainevõrrand, selleks tööks ideaalselt.

4.1 1-D lainevõrrandi koostamine

Oletame, et on kaks inimest ja mõlemad hoiavad kinni mingist nöörist. Üks ühest otspunktist, teine teisest. Nad mõlemad tõmbavad nöörist piisava tugevusega, et nöör oleks õhus. Oletame, et üks nööri hoidjatest liigutab seda kiiresti üles ja alla - tekib nööri läbiv laine.

Vaatleme tekkinud lainet täpsemalt. Laine koosneb kahest kõverjoonest, üks kõverdub algsest nööri positsioonist üles, teine alla. Nööri sisene pinge tõmbab üles kõverdunud osa alla ning alla kõverdunud osa üles. Seda illustreerib joonis 1. Mida suurem on kumerus, seda tugevamini nöör seda tõmbab. Ütleme, et mingi nööri punkti x kaugus tema algasendist on $u(x,t)$, kus t märgib mingit ajahetke. Teeme järelduse, et jõud mis nööri tõmbab on proportsionaalne nööri kumerusega. Nööri kumerust saab määrata teise tuletisega, seega kumerus on $\frac{\partial^2 u}{\partial x^2}$. See on osatuletis kuna nööri kumeruse leidmiseks peab aeg konstantne olema. Oletame, et nööri iga punkt on mingi osake, millel on kindel mass ja mis liigub ainult vertikaalselt. Seega jõud, mis mingile nööri punktile rakendub on $F = ma$, kus m on mass ja a on kiirendus. See tuleneb Newtoni teisest seadusest, mis peaks tuttav olema. Oletame veel, et nöör on ühtse tihedusega, seega iga punkti mass on konstantne. Punkti vertikaalne kiirendus mingis punktis x on $\frac{\partial^2 u}{\partial t^2}$. Kuna me oletasime, et jõud on proportsionaalne kumerusega, saame kokku võrrandi

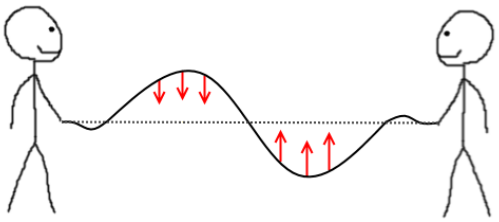
$$\frac{\partial^2 u}{\partial t^2} = k\frac{\partial^2 u}{\partial x^2},$$

kus k on proportsionaalsuse konstant ja mis on positiivne. Seda tavaliselt märgitakse kujul c^2 ja sellest tuleneb laine liikumiskiirus. Lainevõrrandi kohta rohkem informatsiooni saab lehtedelt [6] ja [7].

Tulemuseks ongi lainevõrrandi aga see ei sobi vee simuleerimiseks, sest laine ei kaota energiat. Meil on vaja lisada summutusjõud $2h\frac{\partial u}{\partial t}$, kus h on väike positiivne konstant ja nagu näha, on see proportsionaalne mingi elemendi kiirusega $\frac{\partial u}{\partial t}$. Summutatud lainevõrrand näeb välja järgmine

$$\frac{\partial^2 u}{\partial t^2} + 2h\frac{\partial u}{\partial t} = c^2\frac{\partial^2 u}{\partial x^2}$$

Nüüd on meil olemas vajalik lainevõrrand, jäänud on see veel lahendada.



Joonis 1: Nööri liigutamise tulemusena tekkinud laine

4.2 Lahendamine

Differentsiaalvõrrandeid saab lahendada mitut moodi, kuid siin lainevõrrand lahendatud kasutades lõplike diferentside meetodit (*finite difference method*). Lõplike diferentside meetodi põhiidee on võrrandis esinevate osatuletiste ligikaudne arvutamine teatud punktides. Osatuletised tuleb väljendada ligikaudselt. Rohkem informatsiooni piiratud vahe meetodi kohta leiab [8]. Alguses on laine tasakaalus ehk $u(x,0) = 0$ - iga punkti x kaugus ekviliibriumist on 0. Oletame, et otspunktid on kinnitatud, seega $u(0,t) = 0$ ja $u(L,t) = 0$, kus L on laines esinevate punktide arv. Osatuletiste vastavad ligikaudsed väärtused on järgmised [5]:

$$\frac{\partial u}{\partial t}|_i^n \approx \frac{u_i^{n+1} - u_i^{n-1}}{2\Delta t},$$

$$\frac{\partial^2 u}{\partial t^2}|_i^n \approx \frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2},$$

$$\frac{\partial^2 u}{\partial x^2}|_i^n \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}$$

Asendades lainefunktsioonis tuletised nende ligikaudsete väärtustega on tulemuseks võrrand

$$\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2} + 2h\frac{u_i^{n+1} - u_i^{n-1}}{2\Delta t} = c^2\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}$$

Tekib kahedimensionaalne ruum, kus u_i^n näitab laine kaugust oma nullpunktist ajahetkel n ja vastavas koordinaadi punktis i. Δx ja Δt on vastavalt punktide ja aja muut. Mida väiksem on Δx seda tihedam on laine. Avaldades u_i^{n+1} saame järgmise võrrandi:

$$u_i^{n+1} = \frac{C^2(u_{i+1}^n - 2u_i^n + u_{i-1}^n) + 2u_i^n - u_i^{n-1} + h\Delta t u_i^{n-1}}{1 + h\Delta t}$$

kus $C = c\frac{\Delta t}{\Delta x}$. u_i^{n+1} näitab milline on laine kaugus nullpunktist järgmisel ajahetkel. Kui arvutame iga punkti i jaoks vastavad väärtused antud võrrandiga, saamegi laine.

Kuigi me kirjeldasime nööri, meenutab selle võnkumine veelaineid ja lihtsuse mõttes ütleme, et see lainevõrrand kirjeldab veelainete liikumist.

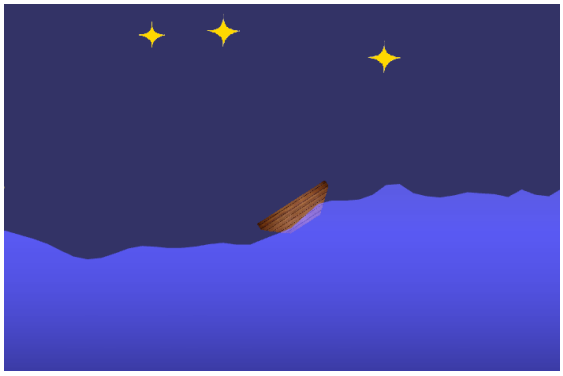
4.3 Kuidas mäng seda ära kasutab?

Nagu eelnevalt kirjeldatud, tagastab lainevõrrand mingi väärtuse, ehk laine kauguse nullpunktist mingis ette antud punktis i. Lihtsuse mõttes ütleme, et nende punktide arv on 100. Mäng loob 100 elemendilise massiivi, kus kõik väärtused on nullid. See tähendab, et laine on siis täiesti statsionaarne - vett pole liigutatud. Lainevõrrandis on vaja ka erinevaid ajahetki laine arvutamiseks, kuid mäng simuleerib lainet iga mängu tsükkel, seega aeg suureneb takistamatult. Kuna lainevõrrand vajab arvutamiseks kahte eelnevat ajahetke, loob programm veel 2 nullidega varustatud massiivi. Iga mängu tsükkel käib programm kõik vee punktid (100) läbi ning arvutab neile uue väärtuse eelneva kahe laineseisu (ajahetke) põhjal. Kui väärtused on arvutatud, tõstab programm lainete väärtused ühe ajahetke tagasi, ehk hetkel käsil olev ajahetk läheb eelmiseks, eelmine üle-eelmiseks. Kui kõik väärtused on nullid, tulevad uued väärtused ka nullid. Kui mingit suvalist vee punkti häiritakse (liigutatakse üles või alla) on lainevõrrandi tööd näha ja tekivad veelained. Sellest täpsemalt räägime järgmises peatükis.

5 Mäng

Mäng kujutab endast väikest paati mida mängija peab juhtima läbi veelainete, kasutades hiire nuppe või veebikaamerat. Mängu eesmärk on paat juhtida ette antud punktidesse võimalikult kiiresti. Paati on vees liikumas näha joonise 2 pealt. Kui mängija suudab paadi juhtida näidatud punkti, suureneb skoor ja tekib uus asukoht ning protsess kordub. Kõik see peab toimima ühe minuti jooksul.

Siin peatükis annan ülevaate mängu sisesest tehnoloogiast läbi lähtekoodi ning seletuste.



Joonis 2: Paat vees

5.1 Vesi

Vesi on selle mängu üks põhilisemaid aspekte.

5.1.1 Vee geomeetria

Vee geomeetria põhineb 1-D lainevõrrandil, mis sai kirjeldatud eelmises peatükis. Lainevõrrandi lahendamine annab meile hulga väärtusi, mida saabki kasutada vee geomeetria joonistamiseks. Lainevõrrandst saadud väärtused tuleb ümber teha mänguruumi koordinaaditeks, millest saabki meie visuaalne vee objekt. Geomeetria on arvutatud järgmiselt:

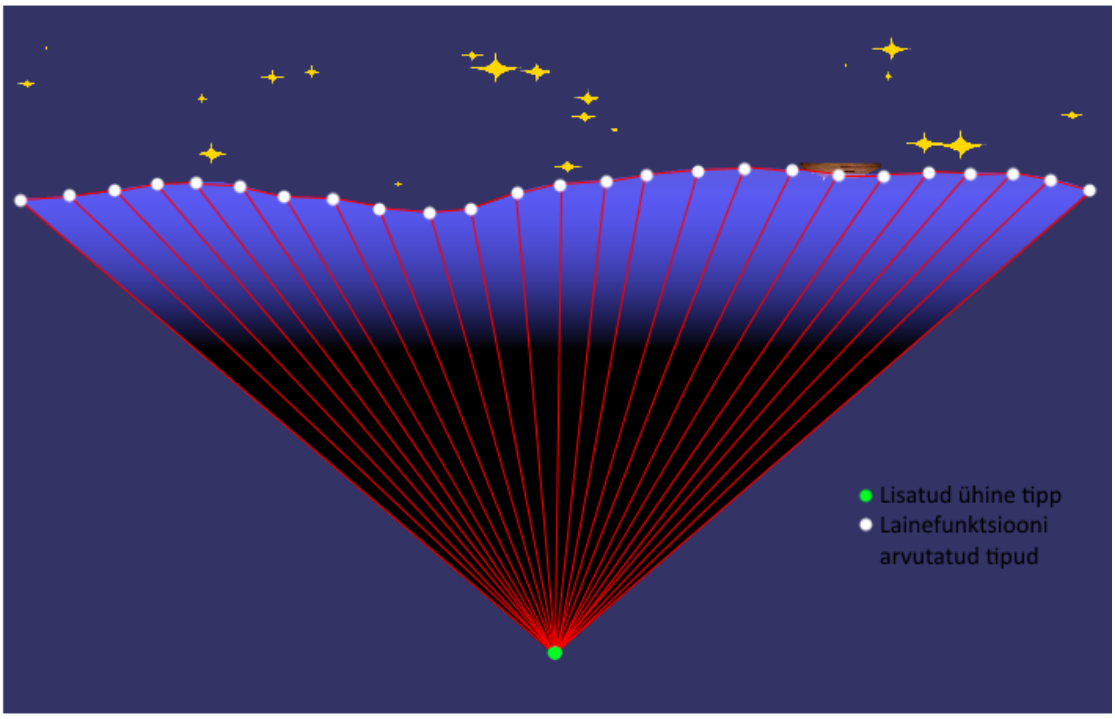
```
/**
 * Calculates and return the wave coordinates using 1d-wave
    equation
 * @param amplifier How many times to increase the wave value
 * @param deltax - Distance between wave points on x-axis
 * @param startLoc - Where the wave starts on x-axis
 * @returns {Array} of coordinates
```

```
 */
function calculateWave(amplifier, deltax, startLoc) {
 var coords = [];
 // lets calculate new displacement for every point x except the
    first and last
 // the first and last are boundaries
 for (var i = 0; i < length-1; i++) {
  var a = i+1;
  // finite difference method for damped wave equation
  var val = (2*v1[a] - v0[a] + C2*(v1[a+1]-2*v1[a]+ v1[a-1]) + h*
     dt*v0[a]) / (1+h*dt);
        v2[a] = val;
        var x = startLoc + i*deltax;
        var y = val * amplifier;

        coords.push(x, y);

 }
 v0 = v1.slice();
 v1 = v2.slice();
 return coords;
}
```

Nagu näha käib funktsioon läbi kõik laine punktid välja arvatud esimene ja viimane ning arvutab igale laine punktile uue väärtuse eelnevate ajahetkede põhjal (eelnevad ajahetked on massiivid v0 ja v1). Iga punktiga arvutatakse uus x ja y kordinaat, mis lükatakse kõikide kordinaatide massiivi. Kui koordinaadid arvutatud, nihkuvad kõik ajahetked ühe võrra tagasi, et järgmisel funktsiooni väljakutsel uusi väärtusi arvutada. Lõpuks tagastab funktsioon arvutatud laine koordinaatide massiivi, millest saab laine kuju. Vett nendest kordinaatidest veel joonistada ei saa. Sellest räägime järgmises sektsioonis.

5.1.2 Vee joonistamine

Kui meil on olemas laine kuju, on selle põhjal vett joonistada suhteliselt lihtne. Nagu eelnevalt mainitud, on WebGL'il joonistamiseks mitu erinevat võimalust ja siinkohal sobib moodus TRIANGLE_FAN. See joonistab kolmnurki mis kõik jagavad ühte tippu. Kuna on olemas laine kuju koordinaadid, on vaja nendele juurde lisada üks tipu koordinaat, millest saab joonistatavatele kolmnurkadele ühiseks. Seda tippu on mõistlik lisada vee keskele ja suhteliselt kaugele vee piirist. Kui see tipp on vee piirile liiga lähedal, võib juhtuda, et tekivad soovimatud defektid, sest kolmnurgad hakkavad üksteist katma. Kui tipp lisatud, tuleb need koordinaadid puhvrisse lükata ja lasta WebGL'il see joonistada. Tulemus on näha joonise 3 pealt.



Joonis 3: Vee joonistamine WebGL-ga

5.2 Paat

Paat on samuti üks väga tähtis mängu element.

5.2.1 Paadi füüsika

Paadi liikumist mõjutab ilmselgelt vesi ja seetõttu peab paat pidevalt jälgima veetaset. Kõigepealt tuleb selgeks teha paadi asukoht veepeal, ehk millise laine punkti peal see asub. See on leitud vastavalt:

```
var curx = parseInt(nx * (boat.locx + waterWidth / 2) / waterWidth
    );
```

Muutuja **boat.locx** on paadi hetke asukoht x-teljestikus (mängu algul on see 0), **waterWidth** on vee laius mänguruumis ja **nx** on vee punktide arv. Kuna vee keskoht on täpselt 0-kordinaadi peal, on vaja hetke paadi asukohale liita pool vee laiust. Kui jagada seda veel vee laiusega, saame protsendilise väärtuse, mis näitab paadi vee läbitust. Korrutades seda veel vee punktide arvuga, saamegi okupeeritud vee punkti. Nüüd leitud punkti abil on võimalik leida paadi asukohas oleva laine nurk radiaanides:

```
var x2 = locx - 1*deltax;
var y2 = v2[curx-1] * amplifier;
var x1 = locx + 1*deltax;
var y1 = v2[curx+1] * amplifier;
var angle = getAngleBetweenPoints(x2, x1, y2, y1);
```

Nagu näha, laine nurk arvutatakse paadi asukohas oleva laine punktist üle-eelmise ja ülejärgmise laine koordinaatide põhjal (paat on umbes 2 punkti jagu lai). Nurga leidmist illustreerib joonis 4. Kui laine vasakpool on kõrgemal kui parempool,

Joonis 4: Nurga leidmine

on nurk positiivne, kui vastupidi, siis on nurk negatiivne. Seda omadust saab ära kasutada laine liikumissuuna määramisel. Kui nurk on positiivne, liigub laine paremale ja kui negatiivne, siis vasakule. Positiivne ja negatiivne nurk on veidi erinevas suurusjärgus ja meil oleks vaja neid modifitseerida:

```
if (angle > 0) {
 angle = Math.PI - angle;
} else {
 angle = (angle + Math.PI)* -1;
}
```

Nüüd on võimalik nurka ära kasutada paadi kiiruse muutmiseks. Kui kiirus on negatiivne liigub paat vasakule ja kui positiivne siis paremale. Sama on ka nurgaga, see tähendab, et on võimalik paadi kiirusele lisada laine nurga väärtus:

```
boat.xspeed += angle / 2500;
```

Nurk on jagatud konstandiga, et kiirus ei kasvaks liiga kiiresti. Kuna vesi raskendab liikumist, tuleb kiirus vähendada takistusjõu võrra:

```
boat.xspeed += -boat.xspeed*boat.drag;
```

Nagu näha on takistusjõud proportsionaalne paadi kiirusega. Kui paadist mööduv laine on liiga väike, ei ületa laine lükkejõud takistusjõudu ning paat ei liigu paigast. Samuti on paadi kiirus limiteeritud mingile kindale arvule (ideaalis laine kiirusele), et paat lõpmata ei kiirendaks. Lõpuks tuleb liita paadi asukohale juurde tema kiirus:

```
boat.locx += boat.xspeed;
```

Kõige selle tulemusega ongi veelainete tõttu liikuv paat.

5.2.2 Paadi joonistamine

Paadi kuju ei muutu ja seega joonistamiseks on vaja teada ainult selle asukohta ning pööret. Kuna need on teada, on vaja WebGL'le teada anda, et see joonistaks paadi asukohale pildi. Pilt tuleb varustada WebGL'le läbi tekstuuride. Tekstuuride kohta leiab informatsiooni lehelt [9].

5.3 Vee manipuleerimine

Mängus saab vett manipuleerida kahte moodi: hiire ning veebikaameraga.

5.3.1 Hiir

Hiirega vee manipuleerimine on suhteliselt lihtne. Iga hiireliigutuse korral salvestab mäng hiire x-koordinaadi. Kui toimub hiirevajutus, leiab mäng vastava vee punkti mängumaailmas:

```
var i = Math.round(nx * (((mousex / canvas.width) * screenWidth) +
    waterWidth / 2 + boat.locx - 2) / waterWidth);
```

Kui vastav punkt on käes, saab selle väärtust mõjutada. Kui vajutati vasakut hiireklõpsu, vähendatakse antud punktis laine väärtust ning kui parem hiireklõps, siis suurendatakse. Järgmisel vee joonistamis kutsel on seda mõju näha - tekib laine.

5.3.2 Kaamera

Kaameraga vee manipuleerimine toimib videopildist liikumise suuna ning kiiruse väljalugemise põhjal. Liikumise lugemine toimub kahe pildi vahe leidmise põhjal. Kahe pildi vahe leidmise idee on võetud Romuald Quantini artiklist [10] ning on veidi modifitseeritud. Kui ühte pilti teisest lahutada, on tulemuseks uus pilt millel on kõik pikslid, mis mõlemal pildil olid samad, mustad ning erinevad pikslid on mingit teist värvi. Kui muuta vastavad teised värvid valgeks, on tulemuseks pilt

millel esinevad ainult mustad ja valged pikslid. Valged pikslid tähendavad liikumist. Kuna veebikaamera võib tekitada palju müra, tuleb lisada kontroll, et pikslid oleksid mingi teatud arvu võrra erinevad. Kui seda arvu suurendada, muutuvad pikslid valgeks suurema liikumise puhul. Uus pilt kahte pildi põhjal on leitud järgmiselt:

```
function threshold(value) {
 return (value > 40) ? 255 : 0;
}

function difference(target, data1, data2, start, end, width,
   height) {
 if (data1.length != data2.length) return null;
 for (var i = 0; i < height ; i++) {
  for (var j = start; j < end; j++) {
   // esimese pildi piksli keskmine v2rv
   var average1 = (data1[4*i*width+4*j] + data1[4*i*width+4*j+1] +
       data1[4*i*width+4*j+2]) / 3;
   // teise pildi piksli keskmine v2rv
   var average2 = (data2[4*i*width+4*j] + data2[4*i*width+4*j+1] +
       data2[4*i*width+4*j+2]) / 3;
   // saa piksli v2rv (kas must v6i valge)
   var pixelColor = threshold(Math.abs(average1 - average2));
   // s2time uue pildi piksli v2rvi
   target[4*i*width+4*j] = pixelColor;
   target[4*i*width+4*j+1] = pixelColor;
   target[4*i*width+4*j+2] = pixelColor;
   // teeme pildi pooleldi l2bipaistvaks
   target[4*i*width+4*j+3] = 128;
  }
 }
}
```

Igale pikslile leitakse uus värv kahe pildi vahe põhjal ning tulemuseks ongi uus must valge pilt. Saadud pilt ei ole käeliigutuste väljalugemiseks hea, sest mängijal võib koos kätega liikuda ka mõni muu kehaosa või riideese. Soovimatute liikumiste välja filtreerimiseks saab kasutada järgmist meetodit:

```
function smoothPixels(data, start, end, width, height) {
 for (var x = 0; x < height; x++) {
  for (var y = start; y < end; y++) {
   var neigbourElementCount = getSurroundingWhitePixelCount(data,
      x, y, end, width, height);
   if (neigbourElementCount < 4) {
    data[4*x*width+4*y] = 0;
    data[4*x*width+4*y+1] = 0;
    data[4*x*width+4*y+2] = 0;
   }
  }}}
```

See kontrollib iga piksli puhul valgete naaberpikslite arvu. Kui valgeid piksleid ümbruses on vähem kui 4, muudetakse see piksel mustaks. Sellist protsessi nimetatakse raku automaatikaks (*cellurar automaton*) ja sellest täpsemalt saab lugeda lehel [11]. Selle meetodi tulemus on näha joonise 5 pealt. Filtreerimata poolel on palju müra. Inimesel oli seljas triibuline särk ning selle väike liikumine oli kohe kahe pildi vahe leidmisel näha. Filtreerides üksikud pildid välja on tulemus kõvasti parem.

Joonis 5: Vasakul filtreeritud, paremal filtreerimata pikslid

Nüüd on vaja leida valgeid piksleid ümbritseva ristküliku. Selle ristküliku sisu ongi liikumine ning selle koordinaate saab kasutada liikumise suuna ja kiiruse välja arvutamiseks. Mäng kontrollib ainult vertikaalset liikumist, seega saab vertikaalset liikumiskiirust arvutada kahe kasti y-koordinaatide põhjal. Kui leitakse uus kast, lahutatakse eelmise kasti y-kordinaadist uue kasti y-koordinaat. Tulemuseks ongi liikumiskiirus. Kui see on negatiivne, toimub liikumine allapoole, kui positiivne, siis vastupidi. Kuna mäng kontrollib nii vasakut kui ka paremat kätt, on liikumine otsitud pildi vasakul ja paremal poolel eraldi. Liikumise otsimist illustreerib joonis 6.

Joonis 6: Liikumise leidmine valgete pikslite arvelt

5.4 Testimiseks kasutatud riistvara ja tarkvara

Mängu on testitud põhiliselt arvutiga, mille operatsioonisüsteem oli Windows 8.1 ja protsessoriks Intel® Core™ i3-3110M. Veebilehitsejaks oli Chrome versiooniga 50. Veel sai testimiseks kasutatud Raspberry Pi versioon 3 ja LG Nexus 5 nutitelefon. Oma limiteeritud jõudlusega sai Raspberry Pi mängu jooksutada suurtel kaadrisagedustel ainult väga väikse resolutsiooniga. Pilditöötlus on väga ressurssinõudev protsess ning see koos mängu joonistamisega sai Pi'le vaevaliseks. Nexus 5 hoidis ilma suure vaevata kõrget kaadrisagedust ilma pilditöötluseta, kuid sellega langes kaadrisagedus märkimisväärselt.

Kui mängu mängida koos veebikaameraga, on vaja seadet suhteliselt võimasa protsessoriga, mis suudaks piisavalt kiiresti videopilti töödelda. Ilma selleta, ei paku mäng seadmele suurt pinget.

6 Kokkuvõte

Käesoleva töö raames valmis arvutimäng mis simuleerib 1-D lainevõrrandit ning mis kasutab graafika kuvamiseks WebGL'i. Töö andis sissejuhatuse WebGL'i ning tõi näiteid sellega graafika kuvamiseks. Räägiti 1-D lainevõrrandist ja selle lahendamisest. Samuti tuli juttu pilditöötlusest ning videost liikumise välja lugemisest.

Töö lugejal peaks olema aimu kuidas toimib WebGL'i, kuidas ära kasutada füüsilist nähtust simulatsiooni/mängu loomiseks ning kuidas toimib liikumise tuvastamine.

Valminud mäng on kindlasti edasi arendatav. Plaanis on lisada mitme mängija võimalus - mängijad mängivad üksteise vastu üritades paati liigutada nende poole. Võitja oleks see, kes paadi vastase poole saab. Samuti oleks vaja graafika ilusamaks teha rakendades WebGL'i suurt potentsiaali. Kindlasti tuleks optimeerida liikumise tuvastamine, et mängu oleks võimalik suurtel kaadrisagedustel mängida ka seadmetel nagu Raspberry Pi.

Viited

[1] "jQuery." https://jquery.com/. (06.05.2016).

[2] "glMatrix." http://glmatrix.net/. (06.05.2016).

[3] "Khronos releases final WebGL 1.0 specification." https://www.khronos.org/news/press/khronos-releases-final-webgl-1.0-specification. (01.05.2016).

[4] "Can I use...support tables for html5, css3, etc." http://caniuse.com/#feat=webgl. (01.05.2016).

[5] F. Izadi ja H. S. Najafi, "Comparison of two finite-difference methods for solving the damped wave equation," *International Journal of Mathematical Engineering and Science*, vol. 3, pp. 35–37, 2014.

[6] P. Dawkins, "Differential equations - the wave equation." http://tutorial.math.lamar.edu/Classes/DE/TheWaveEquation.aspx. (03.05.2016).

[7] "Wave equation." https://en.wikipedia.org/wiki/Wave_equation. (03.05.2016).

[8] H. P. Langtangen, "Finite difference methods for wave motion." http://hplgit.github.io/INF5620/doc/pub/main_wave.pdf, 2013. (03.05.2016).

[9] "WebGL image processing." http://webglfundamentals.org/webgl/lessons/webgl-image-processing.html. (10.05.2016).

[10] R. Quantin, "Javascript motion detection." https://www.adobe.com/devnet/archive/html5/articles/javascript-motion-detection.html. (08.05.2016).

[11] Wikipedia, "Cellular automaton — wikipedia, the free encyclopedia," 2016. (11.06.1016).

Tunnustused

Kolm aastat tagasi Tartu Ülikooli astudes ei teadnud ma arvutitest ega programmeerimisest suurt midagi. Nüüd, kus ma olen lõpetamas bakalaureuseõpet, võin ülima hea meelega öelda, et see on kõvasti muutunud. Ma olen õppinud väga palju, kuid ma tean, et see on tilluke osa selle kõrval mis mul veel jäänud õppida.

Tahaksin tänada oma juhendajaid Benson Muite ning Eero Vainikko, kelle abil ma suutsin oma lõputöö valmis meisterdada. Benson Muite oli alati abivalmis, aitas mul informatsiooni leida ning andis soovitusi töö paremaks muutmisel. Eero Vainikko andis ideid ning aitas mind keeleliselt. Suur tänu teile!

Samuti soovin tänada kõiki oma praktikumijuhendajaid ning lektoreid, kes tegid õppimise väga huvitavaks ja kelle käe läbi olen ma kõik oma väärtuslikud teadmised omandanud. Loodan teie kõigiga koostööd teha ka tulevastel aastatel!

Lisa 1 Lähtekood

Lähtekood on saadaval aadressil `https://kodu.ut.ee/~handre/source.zip` ning on tööga kaasas.

Lisa 2 Mäng

Mäng on mängitav aadressil `https://kodu.ut.ee/~handre/`

Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, Handre Elias,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose **Simulatsioonimäng veebilehitsejas**, mille juhendajad on **Dr. Benson Muite** ja **Prof. Eero Vainikko**,

1.1 reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;

1.2 üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace'i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.

3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, **10.05.2016**