Examensarbete, Högskolan på Åland, Utbildningsprogrammet för Informationsteknik

IMPLEMENTERING & DISTRIBUTION AV ETT 2D-SPEL

John Magnusson

22:2019

Datum för godkännande: 23.05.2019
Handledare: Agneta Eriksson-Granskog

EXAMENSARBETE
Högskolan på Åland

Utbildningsprogram:	Informationsteknik
Författare:	John Magnusson
Arbetets namn:	Implementering & distribution av ett 2D-spel
Handledare:	Agneta Eriksson-Granskog
Uppdragsgivare:	John Magnusson

Abstrakt

Syftet med detta examensarbete är att skapa ett enkelt och intuitivt musikspel där användaren skall använda en datormus i varje hand.

I detta arbete kommer det förklaras hur resultatet har åstadkommits med hjälp av olika designval, kod och programbibliotek.

Detta arbete kommer även förklara förberedelser som krävs innan en eventuell distribution av spelet.

Nyckelord (sökord)
C++, Datorspel, Meddelandesystem

Högskolans serienummer:	**ISSN:**	**Språk:**	**Sidantal:**
22:2019	1458-1531	Svenska	23 sidor

Inlämningsdatum:	**Presentationsdatum:**	**Datum för godkännande:**
17.05.2019	15.05.2019	23.05.2019

DEGREE THESIS
Åland University of Applied Sciences

Study program:	Information Technology
Author:	John Magnusson
Title:	Implementing & distribution of a 2D-game
Academic Supervisor:	Agneta Eriksson-Granskog
Technical Supervisor:	John Magnusson

Abstract

The purpose of this thesis is to create an easy and intuitive music game where the user uses a computer mouse in each hand.

It will be explained in this thesis how the results have been accomplished with the help of different design choices, programming code and libraries.

This thesis will also explain the preparations needed for a potential distribution of the game.

Keywords

C++, Computer Game, Messaging System

Serial number:	ISSN:	Language:	Number of pages:
22:2019	1458-1531	Swedish	23 pages

Handed in:	Date of presentation:	Approved on:
17.05.2019	15.05.2019	23.05.2019

INNEHÅLLSFÖRTECKNING

1. INLEDNING **5**
1.1 Syfte 5
1.2 Metod 5
1.3 Avgränsningar 5
1.4 Uppdragsgivare 5
1.5 Programmeringsspråk & Bibliotek 6
1.5.1 Programmeringsspråket C++ 6
1.5.2 Biblioteket SDL 6
1.5.3 Biblioteket JSON for Modern C++ 6
1.5.4 Biblioteket ManyMouse 6
1.5.5 Spelet DossMauz 6
2. SPELET **7**
2.1 Spelknappar 7
3. PROGRAMSTRUKTUR **9**
3.1 Meddelandesystem 9
3.2 Designdiagram 11
4. SPELMOTOR **12**
4.1 Spel-loop 12
4.2 Användarinput 14
4.3 Grafik 15
4.4 Filhantering 16
4.4.1 Texturladdning 16
4.4.2 Musikladdning 17
4.4.3 Fonter 17
4.4.4 JSON 17
4.5 Kollisionshantering 20
5. DISTRIBUTION **21**
5.1 Copyright 21
5.2 Licenser 21
5.3 Plattformar & Operativsystem 21
6. RESULTAT **22**
7. SLUTSATS **22**
KÄLLFÖRTECKNING **23**

1. Inledning

1.1 Syfte

Syftet med detta arbete är att skapa ett datorspel från grunden. Utöver endast datorspelet behöver också en spelmotor byggas som spelet kan köra på. I slutändan skall spelet kommersialiseras och behöver därför även förberedas inför den framtida distributionen.

1.2 Metod

Arbetet kommer göras i programmeringsspråket C++ och biblioteken SDL, JSON for Modern C++ och ManyMouse kommer användas som hjälpmedel. Grafiken och spelkartorna som spelet skall använda kommer också skapas.

Kunskaperna jag behöver för detta arbete är baserade på högskolestudier i programmering och spelutveckling.

1.3 Avgränsningar

Arbetet kommer endast vara en demoversion bestående av en låt med tillhörande spelkarta. Menyer, funktioner för skapande av spelkartor och annan avancerad funktionalitet kommer göras i framtiden efter detta examensarbete. Musiken till spelet kommer inte göras själv.

1.4 Uppdragsgivare

Då detta arbete är ett eget projekt så är det jag som är uppdragsgivaren.

1.5 Programmeringsspråk & Bibliotek

Ett programmeringsspråk är ett hjälpmedel som programmerare använder sig av för att skapa funktionalitet som en dator kan köra. Ett bibliotek är en samling av sådan funktionalitet.

1.5.1 Programmeringsspråket C++

Programmeringsspråket C++ är ett av de mest använda programspråken för att det går snabbt och enkelt att lära sig som nybörjare. Språket är även väldigt effektivt då det är ett relativt lågnivå-språk och används därför mycket inom spelprogrammering där prestanda är högt prioriterat. (Bjarne Stroustrup 2019)

1.5.2 Biblioteket SDL

Simple DirectMedia Layer, eller SDL, är ett programbibliotek som ger åtkomst till hårdvara såsom mus, tangentbord och skärm oberoende av vilket operativsystem användaren använder. Med hjälp av stödbibliotek används SDL också till fönsterhantering, ljuduppspelning och grafikritning. (Simple DirectMedia Layer 2019)

1.5.3 Biblioteket JSON for Modern C++

JSON for Modern C++ är ett bibliotek som gör det möjligt att läsa från och spara till JSON filer. Detta tillåter dynamisk inladdning av information till programmet; såsom spelkartor och inställningar. (Niels Lohmann 2019)

1.5.4 Biblioteket ManyMouse

ManyMouse är ett bibliotek som läser in och separerar information från flertalet datormöss. Detta gör det möjligt att, till exempel, använda två stycken datormöss per person eller tillåta lokal multiplayer. (Icculus 2019)

1.5.2 Spelet DossMauz

Hädanefter kommer jag referera till spelet som DossMauz, spelet som man spelar med två datormöss.

2. Spelet

DossMauz är ett musikspel där spelaren får testa sin taktkänsla och precisionsförmåga. Under spelets gång kommer det falla färgade block som spelaren skall förstöra med hjälp av datormusen i takt med musiken. Beroende på vilken textur som visas skall den respektive knappen tryckas in vid tillfället som bilden korsar spel-linjen. Figur 1 visar texturerna som spelet använder sig av till de olika knapparna.

2.1 Spelknappar


Left - Denna textur används för musens vänsterknapp


Right - Denna textur används för musens högerknapp

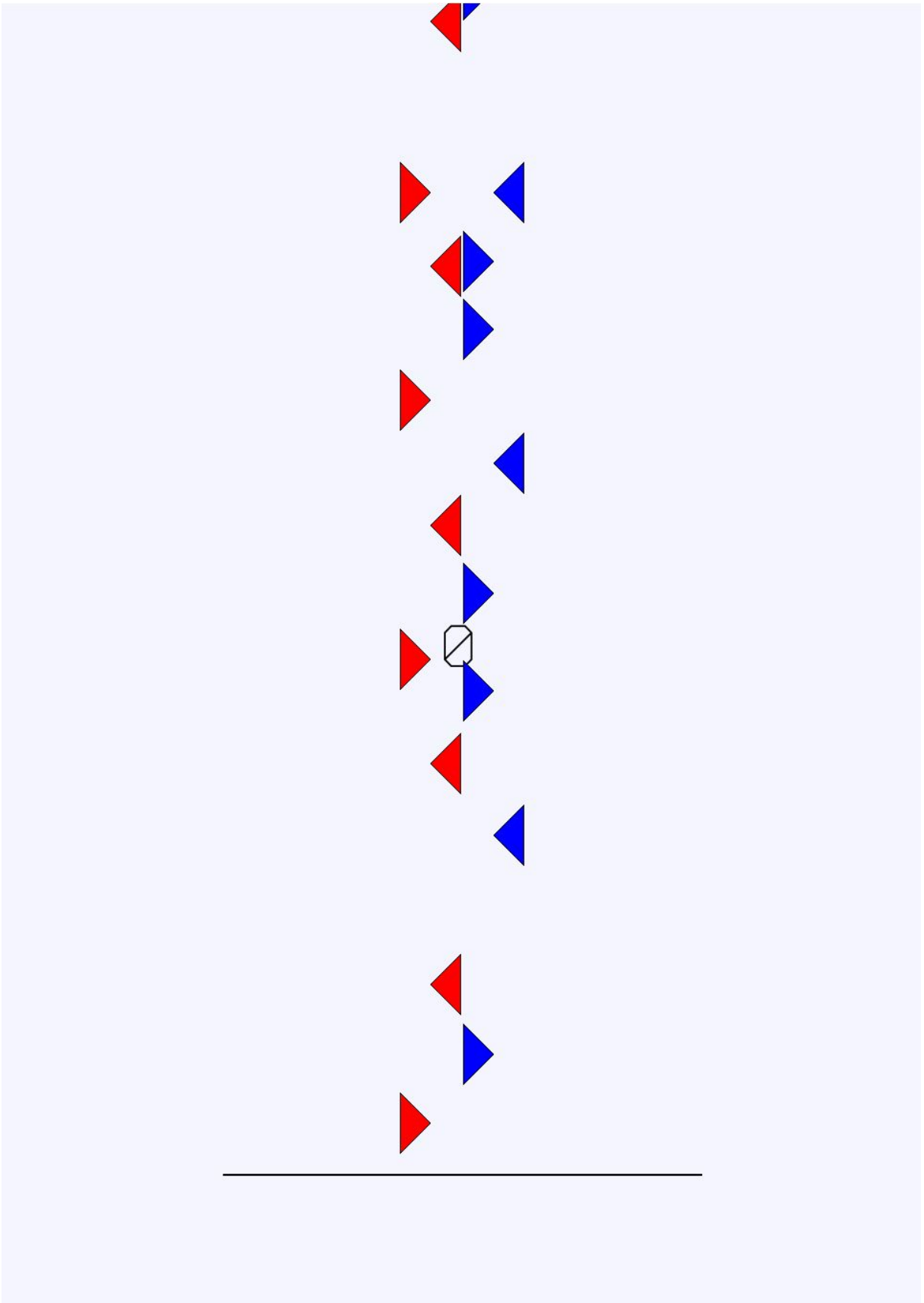

Middle - Denna textur används för musens mittknapp/mushjulsknapp


Up - Denna textur används för musens övre sidoknapp


Down - Denna textur används för musens nedre sidoknapp

Figur 1: Spelets spelknappar

Då spelet kan spelas med flera datormöss kommer varje datormus använda en egen färg för att kunna se skillnad mellan dem. Figur 2 visar en screenshot från en spelkörning.



Figur 2: Dossmauz in action

3. Programstruktur

3.1 Meddelandesystem

Hela programmet använder sig av ett meddelandesystem som bas, där varje kodmodul kan kommunicera med varandra utan att specifikt behöva veta om andra moduler. Meddelandesystemet använder sig av sändare (senders) och mottagare (receivers) där alla mottagare lägger in sig på meddelandesystemet och lyssnar på olika kommandon. Sändare skickar sedan ut meddelanden via dessa kommandon och mottagarna tar emot dem.

Meddelandesystemet har även möjlighet att skicka över meddelanden med valfria variabler så länge dessa har definierats i förväg (Figur 3 & Figur 4). Detta möjliggör ett väldigt brett användningsområde då allt som behövs är ett kommando och medföljande meddelande för att starta vilken funktionalitet som helst i spelet.

```
CREATE_MESSAGE(
    MouseMessage,
    int x int x int,
    std::string x std::string x std::string,
    "device" x "item" x "value"
);

CREATE_MESSAGE(
    KeyMessage,
    std::string x int,
    std::string x std::string,
    "keyname" x "keystate"
);

CREATE_MESSAGE(
    MapMessage,
    std::string x Json,
    std::string x std::string,
    "name" x "json"
);

CREATE_MESSAGE(
    TextureMessage,
    SDL_Texture* x SDL_Rect,
    std::string x std::string,
    "texture" x "rect"
);
```

Figur 3: Definiering av meddelanden

```
CREATE_SUBMESSAGE(Ints, int)
CREATE_SUBMESSAGE(Bools, bool)
CREATE_SUBMESSAGE(Doubles, double)
CREATE_SUBMESSAGE(Strings, std::string)
CREATE_SUBMESSAGE(Events, SDL_Event)
CREATE_SUBMESSAGE(Textures, SDL_Texture*)
CREATE_SUBMESSAGE(Rects, SDL_Rect)
CREATE_SUBMESSAGE(Renderers, SDL_Renderer*)
CREATE_SUBMESSAGE(Jsons, Json)
#undef CREATE_SUBMESSAGE

class SubMessages : public Ints, public Bools, public Doubles, public Strings, public Events, public Textures, public Rects, public Renderers, public Jsons {};

class Message : public SubMessages {
private:
public:
    #define INHERIT(SubMessage...) \
    using SubMessage::setValue; \
    using SubMessage::getValue; \
    using SubMessage::setVector; \
    using SubMessage::getVector;

    INHERIT(Ints);
    INHERIT(Bools);
    INHERIT(Doubles);
    INHERIT(Strings);
    INHERIT(Events);
    INHERIT(Textures);
    INHERIT(Rects);
    INHERIT(Renderers);
    INHERIT(Jsons);

    #undef INHERIT
```

Figur 4: Definiering av variabler till meddelanden

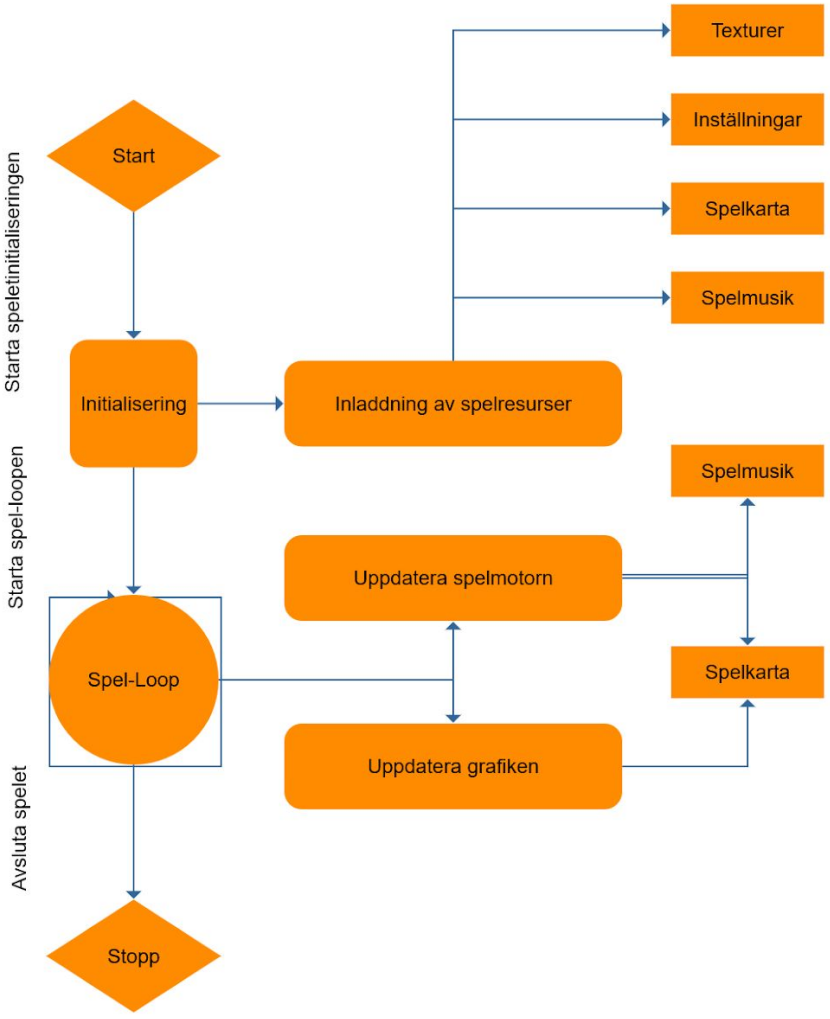
Exempel: Figur 5 visar en klass som både är en sändare och en mottagare. Mottagaren lyssnar på kommandot loadMap och sändaren skickar sedan iväg kommandot Map_Loaded när spelkartan har laddats in.

```
void init(MessageHandler* mh) {
    Receiver::init(mh);
    Sender::init(mh);
    addListeners("loadMap", "saveMap", "removeMap", "printMap");
}

void receive(Message& m, std::string listener) {

    if(listener == "printMap") {
        printMap();
        return;
    }

    StringMessage sm;
    if(sm == m) {
        auto [str] = sm.getValues();

        if(listener == "loadMap") {
            loadMap(str);
            send("Map_Loaded", JsonMessage(mapdata).send());
        }
```

Figur 5: Exempel på hur meddelandesystemet används

3.1 Designdiagram

Detta diagram (Figur 6) visualiserar hur spelet fungerar. Programmet laddar först in alla resurser som behövs till spelkörningen, såsom texturer, inställningar, spelkartor och spelmusik, och startar sedan spel-loopen. Loopen uppdaterar både spelmotorn och grafiken utifrån tiden, spelmusiken och spelkartan. Loopen repeterar sig tills användaren avslutar programmet.



Figur 6: Designdiagram för spelet

4. Spelmotor

Ofta när man skall skapa ett datorspel vänder man sig till redan existerande spelmotorer såsom Unity (Unity 2019), UnrealEngine (UnrealEngine 2019), GameMaker (YoYo Games 2019) eller liknande programvara då de redan sköter allting som ett spel behöver; grafikritning, användarinput, informationshantering, spel-loop och andra platformsspecifika funktioner.

Idén med detta projekt är att skapa allting från grunden och därför skall då också en spelmotor skapas utöver endast spelet. Fördelen med detta är att man lär sig mera exakt vad alla delar i en spelmotor gör och kan skräddarsy motorn att fungera precis som man vill. Nackdelen är att utvecklingstiden ökar avsevärt då man i slutändan egentligen gör två projekt: spelmotorn och spelet.

4.1 Spel-loop

Ett datorspel är en programvara som användaren skall kunna använda i realtid. För att programmet inte skall avslutas så fort det har startats krävs att en loop håller igång spelet tills användaren väljer att avsluta programmet.

En spel-loop skall se till att spelet körs korrekt med konsistenta speluppdateringar och grafikritningar. Spel-loopen skall även se till att programmet inte använder mer datorkraft än vad som krävs, dvs inte köra på full fart hela tiden. (Detta är speciellt viktigt på telefoner).
Spel-loopen jag har skapat använder sig av tidsberäkningar och tidsvariabler definierade i inställningsfilen för att åstadkomma en flytande programkörning.

För varje iteration i loopen som går beräknas tiden som förflutit sedan föregående iteration. Denna deltatid jämförs med ett värde som säger hur ofta speluppdateringar och grafikritningar skall ske och uppdaterar sedan de respektive modulerna för den nuvarande iterationen. Efter att uppdateringarna är gjorda väntar programmet tills det är dags för nästa iteration. Implementationen kan ses i Figur 7.

```
void start() {

    while(running) {

        time.update(1/(double)Zettings["FPSCPU"],1/(double)Zettings["FPSGPU"]);
        if(time.updateCPU()) {
            send("Update_Engine");
        }
        if(time.updateGPU()) {
            send("Update_Graphics");
        }
        time.sleep();
    }
}
```

```
void update(double cpuDelta, double gpuDelta) {

    t = std::chrono::high_resolution_clock::now();
    auto deltacpu = std::chrono::duration_cast<std::chrono::duration<double>>(t - cpuTime);
    auto deltagpu = std::chrono::duration_cast<std::chrono::duration<double>>(t - gpuTime);


    cpuSleepTime = deltacpu.count();
    if(cpuSleepTime > cpuDelta) {
        updatecpu = true;
        cpuTime = t;
    } else updatecpu = false;
    while(cpuSleepTime > cpuDelta) {
        cpuSleepTime -= cpuDelta;
    }
    cpuSleepTime = cpuDelta - cpuSleepTime;


    gpuSleepTime = deltagpu.count();
    if(gpuSleepTime > gpuDelta) {
        updategpu = true;
        gpuTime = t;
    } else updategpu = false;
    while(gpuSleepTime > gpuDelta) {
        gpuSleepTime -= gpuDelta;
    }
    gpuSleepTime = gpuDelta - gpuSleepTime;


    if(cpuSleepTime < gpuSleepTime) sleepTime = round(cpuSleepTime*1000000000);
    if(gpuSleepTime < cpuSleepTime) sleepTime = round(gpuSleepTime*1000000000);

}

void sleep() {
    std::this_thread::sleep_for(std::chrono::nanoseconds(sleepTime));
}

bool updateCPU() {
    return updatecpu;
}

bool updateGPU() {
    return updategpu;
}
```

Figur 7: Spel-loop & Tidsberäkningar

4.2 Användarinput

För att kunna interagera med ett datorspel behöver spelmotorn klara av att ta in och processera input från användaren. Spelmotorn som jag har byggt gör detta med hjälp av två stycken programbibliotek, SDL & ManyMouse.

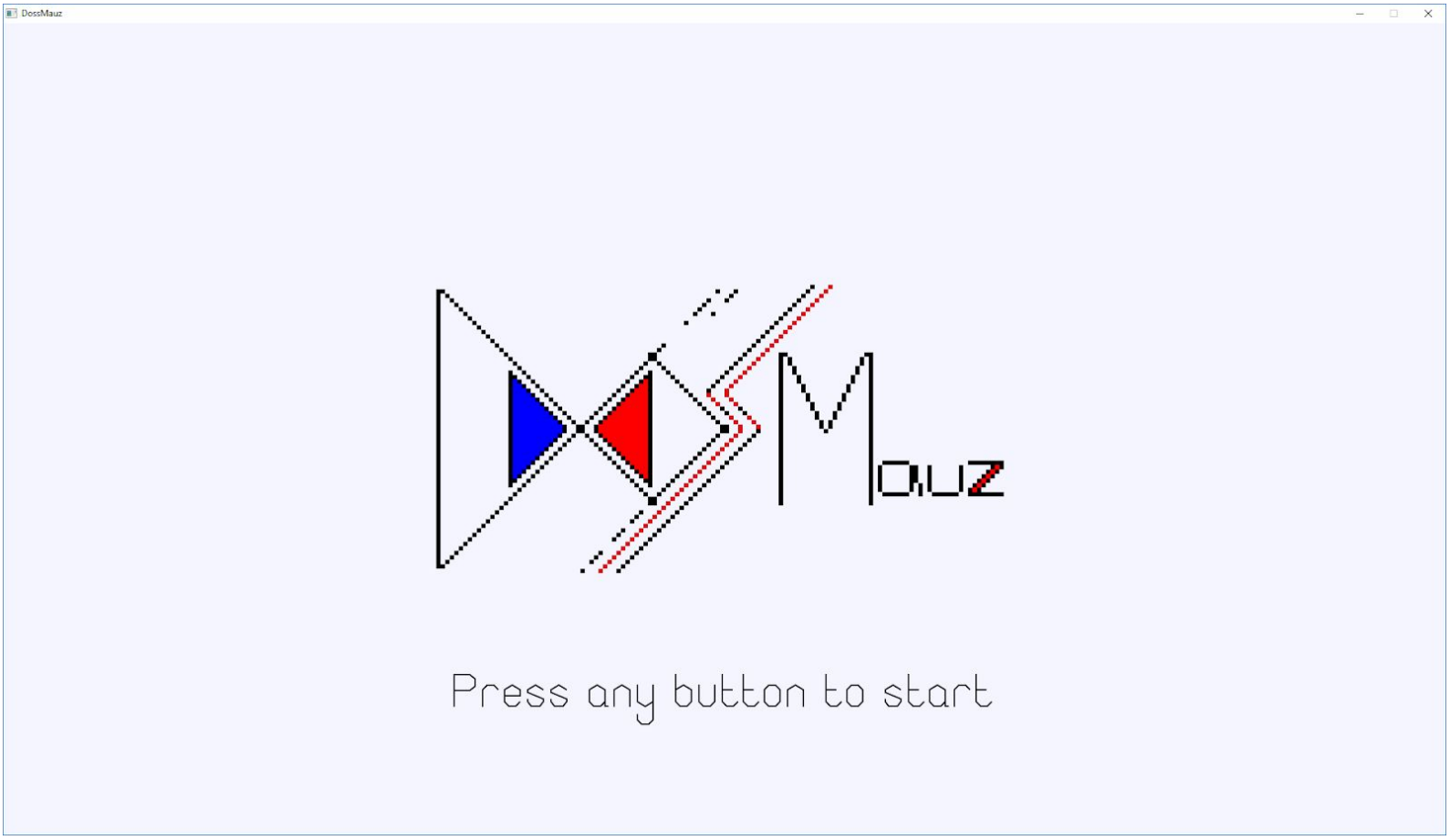
Då dagens operativsystem inte är designade för att kunna hantera flera datormöss har jag tagit biblioteket ManyMouse i bruk som hjälp. ManyMouse läser hårdvaruinformationen direkt från USB-portarna och kan därför kringgå operativsystemens begränsningar. Denna information separeras sedan till meddelanden som programmet kan hantera. Annan input som tangentbord, spelkontroller & touch-skärmar hanterar biblioteket SDL.

Figur 8 visar Windows muspekare (vit) gentemot muspekare som DossMauz skulle kunna använda (röd & blå).

Figur 8: Windows muspekare & DossMauz möjliga muspekare

4.3 Grafik

Majoriteten av alla datorspel som existerar har någon form av grafisk representation. SDL hjälper också till med denna del i form av fönsterhantering och grafikritning. Figur 9 visar startskärmen av spelet där SDL har skapat fönstret, målat upp bakgrundsfärg, skrivit ut text och ritat ut logon.



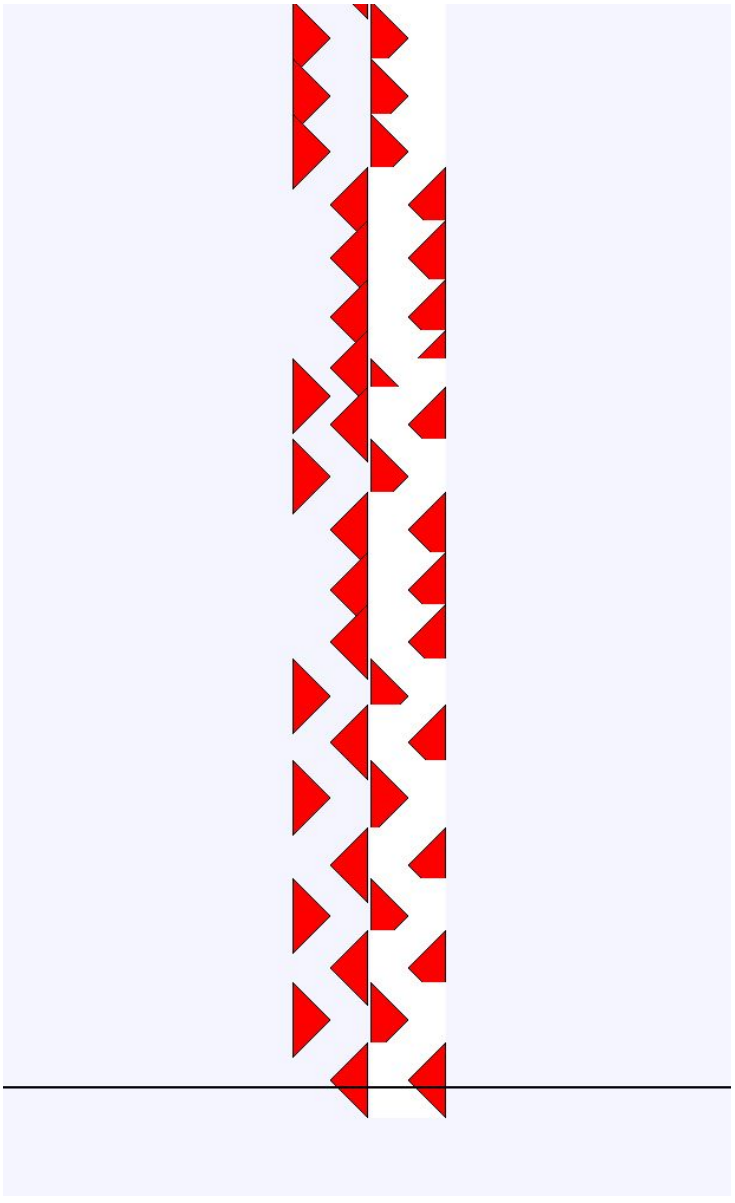
Figur 9: Startskärmen av programmet

4.4 Filhantering

Datorspel behöver oftast ladda in information från olika filer på något sätt. SDL används här också för enkel inmatning av resurser.

4.4.1 Texturladdning

Vikten av att kunna ladda in speltexturer med osynlighet är väldigt hög i datorspel. För att kunna ladda in speltexturer som innehåller osynlighet korrekt i SDL behöver underbiblioteket SDL_image användas. (Simple DirectMedia Layer 2019). Figur 10 visar skillnaden mellan texturer med osynlighet och texturer utan osynlighet.



Figur 10: Exempel på texturer med osynlighet och texturer utan osynlighet

4.4.2 Musikladdning

Då SDL inte nativt klarar av att ladda in eller spela upp musik kan ett annat underbibliotek, SDL_mixer, hjälpa till. (Simple DirectMedia Layer 2019). Då DossMauz är ett musikspel behöver denna funktionalitet stödjas.

4.4.3 Fonter

I Figur 9 kan man under logon se utskriven text. SDL i sig självt har ej denna funktionalitet utan underbiblioteket SDL_ttf måste användas. (Simple DirectMedia Layer 2019). Detta bibliotek laddar in fonter och producerar sedan text som kan skrivas till skärmen.

4.4.4 JSON

För att enkelt kunna ändra på funktionalitet i ett datorprogram kan man använda sig av dynamisk inladdning av information. Ofta är sådan information sparad i något speciellt format såsom XML eller JSON. Designvalet mellan XML och JSON var väldigt enkelt då JSON-kodning är mer lik kodning i C++ än vad XML är, och då jag lyckades hitta ett bibliotek som kunde hantera JSON smidigt. Figur 11 visar skillnad mellan XML- och JSON-kod.

XML

```
<workerinfo>
    <workers>
        <worker>
            <name>John Magnusson</name>
            <age>24</age>
        </worker>
        <worker>
            <name>Jim Magnusson</name>
            <age>21</age>
        </worker>
    </workers>
<workerinfo>
```

JSON

```
{ "workerinfo" :
    {
        "workers" : [
            {
                "name": "John Magnusson",
                "age": 24
            },
            {
                "name": "Jim Magnusson",
                "age": 21
            }
        ]
    }
}
```

Figur 11: XML- och JSON-kod

Mitt program använder då JSON för att enkelt kunna spara olika inställningar i inställningsfilen och alla knapptryckningar för en spelkörning sparas i spelkartorna.

Figur 12 visar inställningsfilen där variabler för spelet finns sparade. Dessa värden inkluderar; bakgrundsfärg, snabbhetsgrad av spelet, loopningsvärden för spel-loopen, linjevariabler för spel-linjen, skärmupplösning, mus-id för ManyMouse, etc.


```
{
    "AIDifficulty": 1,
    "AIFPS": 18,
    "AIMultitasking": 1,
    "AIReflexes": 0.1,
    "BackgroundColor": {
        "A": 255,
        "B": 255,
        "G": 245,
        "R": 244
    },
    "Difficulty": 0.15,
    "FPSCPU": 100,
    "FPSGPU": 200,
    "HitLine": 0.85,
    "LineColor": {
        "A": 255,
        "B": 0,
        "G": 0,
        "R": 0
    },
    "LineSizeX": 0.5,
    "LineSizeY": 2,
    "MonitorRezX": 3440,
    "MonitorRezY": 1440,
    "Mouse1": 1,
    "Mouse2": 0,
    "ResolutionX": 1920,
    "ResolutionY": 1080,
    "TexSize": 64,
    "Title": "DossMauz",
    "TitleText": "Press any button to start"
}
```


Figur 12: Inställningsfilen

Figur 13 visar början av en spelkarta där variabelvärden för noterna finns sparade. “Difficulty” säger hur lång tid spelaren har på sig att trycka på varje not, “id” säger vilken av mössen det gäller, “key” säger vilken musknapp det gäller, “time” säger när denna händelse skall ske i relation med musiken och “value” säger om knappen skall tryckas in eller släppas.


```
"Difficulty": 0.15,
"Notes": [
    {
        "id": 1,
        "key": 0,
        "time": 3.6247278,
        "value": 1
    },
    {
        "id": 1,
        "key": 0,
        "time": 3.8208239,
        "value": 1
    },
    {
        "id": 2,
        "key": 1,
        "time": 4.0550206,
        "value": 1
    },
    {
        "id": 2,
        "key": 0,
        "time": 4.2763755,
        "value": 1
    },
    {
        "id": 1,
        "key": 1,
        "time": 4.3427131,
        "value": 1
    },
```


Figur 13: Exempel på spelkarta

4.5 Kollisionshantering

I många 2D-spel används ofta bilden som kollisionsmätare; då objekten jämförs med varandra på pixelnivå. Denna kollisionsmetod fungerar utmärkt för den typen av spel då det är vad användaren förväntar sig bör ske då bilden är huvuddelen för sådana spel.

På grund av att detta projekt även koncentrerar sig på ljud och inte endast bild kan denna metod producera oväntade resultat då synkronisering mellan ljud och bild aldrig kan bli 100% exakt. När man spelar ett musikspel bör då musiken vara i fokus och därför är metoden som används för kollision baserad på tiden för knapptryckningen och inte den grafiska representationen. Detta designval leder till många positiva aspekter såsom;

- Mindre hänsyn till synkroniseringsfel behövs för programutecklarens del då musiken alltid spelas upp på ett enda sätt gentemot att grafiska bilder kan se lite olika ut beroende på skärmupplösning och texturskalning.

- Spelaren kan koncentrera sig på takten då varje knapptryck kommer registreras på korrekt tid vilket leder till mindre frustration då ljudsynkronisering är mer korrekt än bildsynkronisering som nämnt tidigare.

- Enklare och effektivare programmering då endast tiden behöver jämföras och inte bilden

5. Distribution

När man skapar ett datorprogram som man ämnar att lansera behöver man ha detta i åtanke i varje steg av utvecklingen.

5.1 Copyright

Man behöver vara försiktig med copyright av alla dess slag. Spelresurser såsom texturer, fonter och musik behöver man antingen laga själv för att helt undvika copyright, köpa in dessa resurser av någon eller hitta copyright-free material. Samma sak kan även gälla för programkod och bibliotek.

5.2 Licenser

För varje resurs som inte är gjorda på egen hand behövs det licenser för att få använda. Det gäller att undvika publika licenser som kräver att resten av programmet också lanseras publikt såsom GNU-licenser. För programbibliotek behöver man då se till att de får användas till kommersiella projekt.

5.3 Plattformar & Operativsystem

Då DossMauz är skapat i C++ och endast använder sig av bibliotek med crossplatform-möjligheter kan det lanseras på en rad olika operativsystem som: Windows, Linux, Mac, Android, iOS och till och med konsoler som Nintendo Switch utan större problem. För att kunna sälja denna programvara behöver man också distribuera det på olika plattformar eller butiker som t.ex. Steam, Uplay, Epic Games Store, Origin, Discord, GOG, Google Play, Apple Store eller Amazon Appstore. När man laddar upp spel på dessa plattformar finns det dock krav som måste följas; det får till exempel inte existera virus i programmet och det får heller inte krascha operativsystemet som spelet spelas på genom minnesläckor.

6. Resultat

Efter att ha följt metoderna och hållit mig till avgränsningarna har arbetet blivit lyckat. Ett musikspel skapades med underliggande spelmotor där spelaren kan spela spelet med en datormus i varje hand. Då avgränsningarna hållits är detta endast en demoversion och extra funktionalitet kommer implementeras först i framtiden.

7. Slutsats

Detta projekt har varit en lärorik och intressant upplevelse där jag har fått lära mig mycket om hur en spelmotor fungerar och vikten av att göra bra designval. Då detta var ett eget arbete blev det ganska svårt att kapsla in ämnet och planera hur mycket som skall göras på varje modul och detta har därför lett till mycket längre utvecklingstider än förväntat. Efter detta arbete kommer jag fortsätta bygga på detta projekt tills det blir redo inför en framtida distribution.

KÄLLFÖRTECKNING

Bjarne Stroustrup (2019) C++ 17.05.2019

http://www.stroustrup.com/C++.html

Simple DirectMedia Layer (2019) Homepage 04.04.2019

https://www.libsdl.org/index.php

Simple DirectMedia Layer (2019) SDL_image 17.05.2019

https://www.libsdl.org/projects/SDL_image

Simple DirectMedia Layer (2019) SDL_mixer 17.05.2019

https://www.libsdl.org/projects/SDL_mixer

Simple DirectMedia Layer (2019) SDL_ttf 17.05.2019

https://www.libsdl.org/projects/SDL_ttf

Icculus (2019) ManyMouse 04.04.2019

https://icculus.org/manymouse/

Niels Lohmann (2019) JSON for Modern C++ 04.04.2019

https://github.com/nlohmann/json

Unity (2019) Homepage 22.05.2019

https://unity.com/

Unreal Engine (2019) Homepage 22.05.2019

https://www.unrealengine.com/

YoYo Games (2019) Gamemaker 22.05.2019

https://www.yoyogames.com/gamemaker