LINKÖPINGS UNIVERSITET

# TDP005 Projekt: Objektorienterat system

# Designspecifikation

Författare

Love Bäckman, lovba497@student.liu.se
Gustav P Svensson, gussv375@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2016
Version 1.0

2016-12-18

# Innehåll

# 1 Revisionshistorik

| Ver. | Revisionsbeskrivning | Datum |
|---|---|---|
| 0.1 | Påbörjat utkast | 161123 |
| 0.2 | Första utkast klart | 161125 |
| 1.0 | Designspec. klar | 161218 |

# 2 Detaljbeskrivning av Player

Klassen Player representerar spelarkaraktären som spelaren styr med hjälp av piltangenterna.

Player ärver från klassen Gravitating_Object, som ärver från klassen Movable_Object, som ärver från klassen Simulatable_Object, som ärver från klassen Object. Det vill säga, spelaren är ett simulerbart och rörligt objekt som påverkas av gravitation.

Player's konstruktor tar emot en Vector2f för position, en Vector2f för storlek, en sträng för typ, en Texture-pekare för textur och en float för hastighet.

Samtliga parametrar förutom hastigheten skickas vidare till Gravitating_Object's konstruktor, tillsammans med ett falskt booleskt värde som talar om att spelaren är icke-solid. Hastigheten lagras i Player's m_speed-variabel.

## 2.1 Variabler

Player ärver följande konstant-variabler från sina superklasser (enligt stilen **variabel** - typ av konstant - beskrivning):

**const sf::Texture *m_texture** - textures - textur som spelaren ska ritas ut med

**const std::string m_type** - attributes - identifierar att spelaren är av typ "player"

Player introducerar följande nya konstant-variabler (enligt stilen **variabel** - typ av konstant - beskrivning)

**const float m_speed** - attributes - definierar spelaren bashastighet

Player ärver följande state-variabler från sina superklasser (enligt stilen **variabel** - typ av state - beskrivning):

**bool m_solid** - attributes - om spelaren är solid (alltid falsk)

**bool m_delete_status** - general - om spelaren ska deletas (alltid falsk)

**sf::RectangleShape _shape** - general - shape som används för att få ut position, storlek samt för utritning

Player introducerar följande nya state-variabler (enligt stilen **variabel** - typ av state - beskrivning):

**bool m_jumping** - general - om spelaren hoppar

**bool m_on_ground** - general - om spelaren är på marken

**sf::Clock m_slow_bird_clock** - buffs & debuffs - hur länge sedan det var spelaren kolliderade med en Slow_Bird (eller Bomb_Bird)

**std::unordered_set<const Object*> m_slow_bird_debuffs** - buffs & debuffs - samtliga Slow_Bird's (och Bomb_Bird's) vars debuffs är aktiva på spelaren

**sf::Clock m_boost_bird_clock** - buffs & debuffs - hur länge sedan det var spelaren kolliderade med en Boost_Bird

**std::unordered_set<const Object*> m_bost_bird_buffs** - buffs & debuffs - samtliga Boost_Bird's vars buffs är aktiva på spelaren

**sf::Clock m_nfbb_clock** - buffs & debuffs - hur länge sedan det var spelaren kolliderade med en NFBB

**int m_nfbb_debuffs** - buffs & debuffs - antal aktiva NFBB debuffs

**bool m_quicksand_debuff** - buffs & debuffs - om Quicksand debuffen är aktiv

**bool m_quicksand_collision** - collision - om spelaren kolliderat med ett block av typ Quicksand

**std::string m_oog_action** - out-of-game actions - om spelaren kolliderat med något som har påverkan utanför spelet

## 2.2 Metoder - Arv

Player ärver följande metoder från sina superklasser:

- get_type
- is_solid
- get_delete_status
- get_shape
- simulate
- end_simulate

### 2.2.1 get_type (public)

Parametrar:
Return: *std::string*

Metoden get_type returnerar m_type-variabeln.

### 2.2.2 is_solid (public)

Parametrar:
Return: *bool*

Metoden is_solid returnerar m_solid-variabeln.

#### 2.2.3 get_delete_status (public)

Parametrar:
Return: *bool*

Metoden get_delete_status returnerar m_delete_status-variabeln.

#### 2.2.4 get_shape (public)

Parametrar:
Return: *sf::RectangleShape*

Metoden get_shape returnerar m_shape-variabeln.

#### 2.2.5 simulate (public)

Parametrar: *const int total_simulations, const std::vector<const Object*> &objects*
Return: *std::vector<Object*>*

Metoden simulate utför ett objekts simuleringslogik. För klassen Player innebär detta att förflytta sig samt kolla utföra kollisionskontroller.

#### 2.2.6 end_simulate (public)

Parametrar: *const std::vector<const Object*> &objects* Return:

Metoden end_simulate utför logik som skall utföras efter att ett objekt simulerat färdigt. För klassen Player innbära detta att utföra en sisa kollisionskontroll samt återställa simulation state-variabler.

### 2.3 Metoder - Arv & Override

Player ärver och override:ar följande metoder från sina superklasser:

- prepare_simulate
- handle_moving_collision(const Object *object, const sf::Vector2f &steps)
- handle_static_collision(const Object *object)
- handle_end_collision()
- collision_state_cleanup()

#### 2.3.1 prepare_simulate (public)

Parametrar: *const float distance_modifier, const float gravity_constant*
Return: *int*

Metoden prepare_simulate förbereder förbereder spelaren för simulering genom att räkna ut vilken riktning spelaren ska röra sig åt beroende på vilka tangenter som är nedtryckta, samt hur lång distansen som spelaren ska röra sig blir efter att ha applicerat m_speed-variabeln och en speed_modifier-variabel (som räknas ut beroende på spelarens state).

Resultatet av uträkningarna lagras i m_distance-variabeln. Efter uträkningarna är klara anropas den override:ade metoden, Gravitating_Object::prepare_simulate, med samma parametrar.

Gravitating_Object::prepare_simulate räknar ut gravitationspåverkan och applicerar uträkningen på m_distance-variabeln, varefter den override:ade metoden Movable_Object::prepare_simulate anropas med samma parametrar.

Movable_Object::prepare_simulate applicerar distance_modifier på m_distance och räknar ut hur många simulationer som behöver göras med restriktionen att ett objekt inte kan förflytta sig mer än 24px (det minsta existerande objektet) åt gången. Detta för att undvika missade kollisioner om ett objekt försöker röra sig över 24px. Om objektet försöker röra sig 30px kommer alltså antal behövda simulationer vara 2.

Movable_Object::prepare_simulate returnerar sedan antalet behövda simulationer till Gravitating_object::prepare_simulate, som returnerar vidare till prepare_simulate som returnerar värdet till sin anropare.

#### 2.3.2 handle_moving_collision

Parametrar: *const Object *object, const sf::Vector2f &steps*
Return:

Metoden handle_moving_collision hanterar kollisioner mellan spelaren och objekt som uppstår medan spelaren försöker förflytta sig. Metoden börjar med att anropa sina superklassers implementationer av metoden, för att sedan utföra logik specifik för klassen Player. I metodkedjan för handle_moving_collision hanteras framförallt kollisioner mellan spelare och solida objekt, vid vilka spelarens position justeras så att denne inte kan passera igenom dem.

#### 2.3.3 handle_static_collision

Parametrar: *const Object *object*
Return:

Metoden handle_moving_collision hanterar kollisioner mellan spelaren och objekt som uppstår före och efter att spelaren har förflyttat sig. Metoden börjar med att anropa sina superklassers implementationer av metoden, för att sedan utföra logik specifik för klassen Player. I metodkedjan för handle_moving_collision hanteras kollisioner med andra rörliga objekt, vid vilka spelarens state påverkas.

#### 2.3.4 handle_end_collision

Parametrar:
Return:

Metoden handle_end_collision utför logik som ska utföras efter all annan kollisionslogik baserat på vad som har och inte har kolliderats med under de tidigare kollisionskontrollerna (collision state). Efter logiken utförs anropas superklassernas implementationer av metoden.

#### 2.3.5 collision_state_cleanup

Parametrar:
Return:

Metoden collision_state_cleanup återställer collision state-variabler introducerade i klassen Player och anropar superklassernas implementationer av metoden.

## 2.4 Metoder - Nya

### 2.4.1 get_oog_action (public)

Parametrar:
Return: *const std::string&*

Metoden get_oog_action returnerar m_oog_action-variabeln.

# 3 Detaljbeskrivning av Game_State

Klassen Game_State representerar spelets state, det vill säga alla objekt och variabler som har inverkan på spelet, samt ansvarar för att läsa in och kalla på respektive simulerbart objekts simuleringsfunktion (funktioner som kan förändra spelets state).

Game_State ärver från klassen State.

Game_State's konstruktor tar inte emot några argument, men när den kallas läser den in och lagrar samtliga resources som skall användas i spelet.

## 3.1 Variabler

Game_State ärver följande variabler från sina superklasser (enligt stilen **variabel** - beskrivning):

**std::unordered_map<std::string, sf::Texture*> textures** - texturer som ska användas till objekt

**sf::Texture background**$_texture$ - texturer som ska användas som bakgrund

**sf::Sprite background** - sprite som målas ut som bakgrund

**std::vector<const Object*> objects** - samtliga objekt i nuvarande state

**std::vector<const Object*> texturated_objects** - samtliga texturerade objekt i nuvarande state (objekt som ska ritas ut)

**std::unordered**$_map < std :: string, sf :: Text > text\_objects$ - samtliga text-objekt i nuvarande state (ska ritas ut)

**sf::Font font** - font som ska sättas på text-objekt

Game_State introducerar följande nya variabler (enligt stilen **variabel** - beskrivning)

**float record_time** - rekordtiden för nuvarande bana

**float gravity_constant** - nuvarande banas gravity-konstant

**sf::Clock delta_clock** - tid sedan senaste simulering

**sf::Clock elapsed_time_clock** - tiden sedan inladdning av ny bana

**std::vector<Simulatable_Object*> simulatable_objects** samtliga simulerbara objekt i nuvarande state

**const Player *player** spelarkaraktären i nuvarande state

## 3.2 Metoder - Arv

Game_State ärver följande metoder från sina superklasser:

- get_background
- get_texturated_objects
- ref_text_objects

### 3.2.1 get_background

Parametrar:
Return: *const sf::Sprite&*

Metoden get_background returnerar background-variabeln.

### 3.2.2 get_texturated_objects

Parametrar:
Return: *const std::vector<const Object*>&*

Metoden get_texturated_objects returnerar texturated_objects-variabeln.

### 3.2.3 ref_text_objects

Parameterar:
Return *std::unordered_map<std::string, sf::Text>*

Metoden ref_text_objects returnerar text_objects variabeln som en icke-konstant referens.

## 3.3 Metoder - Arv & Override

Game_State ärver och override:ar följande metoder från sina superklasser:

- prepare_simulate
- simulate
- set_view
- soft_reset

### 3.3.1

Paramterar:
Return:

Metoden _ returnerar _

#### 3.3.2 preprare_simulate

Paramterar:
Return:

Metoden prepare_simulate returnerar förbereder Game_State för simulation genom att återställa statet för att sedan läsa in en ny level (med hjälp av klassen Level_Parser) och sätta nödvändiga variabler (gravity_constant, background, record_time objects, texturated_objects, simulatable_objects, player). Klockorna delta_clock och elapsed_time återställs även.

#### 3.3.3 simulate

Paramterar:
Return:

Metoden simulate arbetar i flera steg.

Första steget är att anropa alla simulerbara objekts prepare_simulate-metod för att förbereda dem inför simulering samt fråga varje objekt hur många simuleringar de kräver. Det högsta antalet krävda simuleringar sparas i den lokala variabeln simulation_cycles.

Nästa steg är att anropa alla simulerbara objekts simulate-metod gånger det högsta antalet krävda simuleringar, där det totala antalet simuleringar skickas med som en parameter så att övriga objekt kan anpassa sin simulering (dividera förflyttningsdistansen med simulation_cycles). Med som parameter skickas även objects-variabeln som objekten kan använda för att kontrollera kollisioner. Om ett objekts simulate-metod inte returnerar en tom vektor insertas alla element från vektorn till en lokal variabel new_objects.

I det tredje steget anropas alla simulerbara objekts end_simulate-metod, där objekten får utföra sin end-of-simulation logik.

I det fjärde steget tas alla objekt som markerat sig själva med m_delete=true bort ur vektorerna simulatable_objects, texturated_objects samt objects. Minnet frigörs även för objektet.

Sedan, i det femte steget, läggs alla objekt i den lokala variabeln new_objekts till i vektorn objects, samt texturated_objects och simulatable_objects om objekten uppfyller respektive vektors krav.

Efter dessa steg kontrolleras sedan om spelaren m_oog_action-variabel=goal, varpå elapsed_time jämförs med record_time för att se om rekordtiden skall skrivas över.

Är inte m_oog_action-variabeln=goal kontrolleras om Escape eller R tryckts ned för att isåfall återgå till menyn respektive starta om banan.

Är inget av ovanstående sant returneras 0, vilket innebär att klassen Engine återigen kommer anropa denna funktion.

## 4 Designbeskrivning

Designen är implementerat på sådant sätt att en spelmotor, Engine, håller reda på vilket state av Menu_State och Game_State som är aktivt och kör det state:ets simulate-metod. Beroende på returnvärdet från denna simulate-metod kan det aktiva state:et ändras. Efter att det aktiva state:et simulerats renderar

sedan Engine state:ets renderbara objekt.

I nuläget är Menu_State enbart en nedskalad version av Game_State som möjliggör att använda en level som meny. Detta fungerar för den nuvarande meny-funktionaliteten, men vi skulle implementera ytterligare funktionalitetet såsom en level-meny medför det vissa design-svårigheter.

När simulate-metoden för det aktiva state:et anropas kommer det state:et sedan fortsätta simulationskedjan genom att kalla på simulerings-metoder för alla simulerbara objekt i state:et.

Dessa simulerings-metoder utför diverse logik såsom hur objektet ska röra sig baserat på buffs/debuffs det har ådragit sig genom kollision med andra objekt, eller om objektet ska spawna ett nytt objekt.

Till ett objekts simulerings-metod skickas en vektorn med pekare till konstanta-objekt in, detta innebär att ett objekt aldrig kan ändra på ett annat objekt. Ett objekt får alltså enbart ändra sina egna egenskaper (samt skapa helt nya objekt genom att returnera en vektor nya objekt).

Denna design tycker vi är bra då det tvingar utvecklaren att ta fram en robust logik för kollisionshändelser där all logik är isolerad till det egna objektet. Det blir annars lätt rörigt om man i kollisionslogiken ändrar på ett annat objekt, som på grund utav den ändringen sedan kommer agera annorlunda än tilltänkt när det objektet i sin tur kolliderar med det tidigare objektet.

För att undvika buggar där kollisioner missas har vi även satt en begränsning att ett objekt inte kan röra sig över 24px (det minsta objektet) per simulation. Om objekt A försöker röra sig 30 pixlar och objekt B försöker röra sig 10 pixlar kommer objekt A i pågående simulering istället röra sig 15 pixlar, och objekt B 5 pixlar. Därpå följer en ny simulering där objekt A och B rör sig de resterande 15 respektive 5 pixlarna.

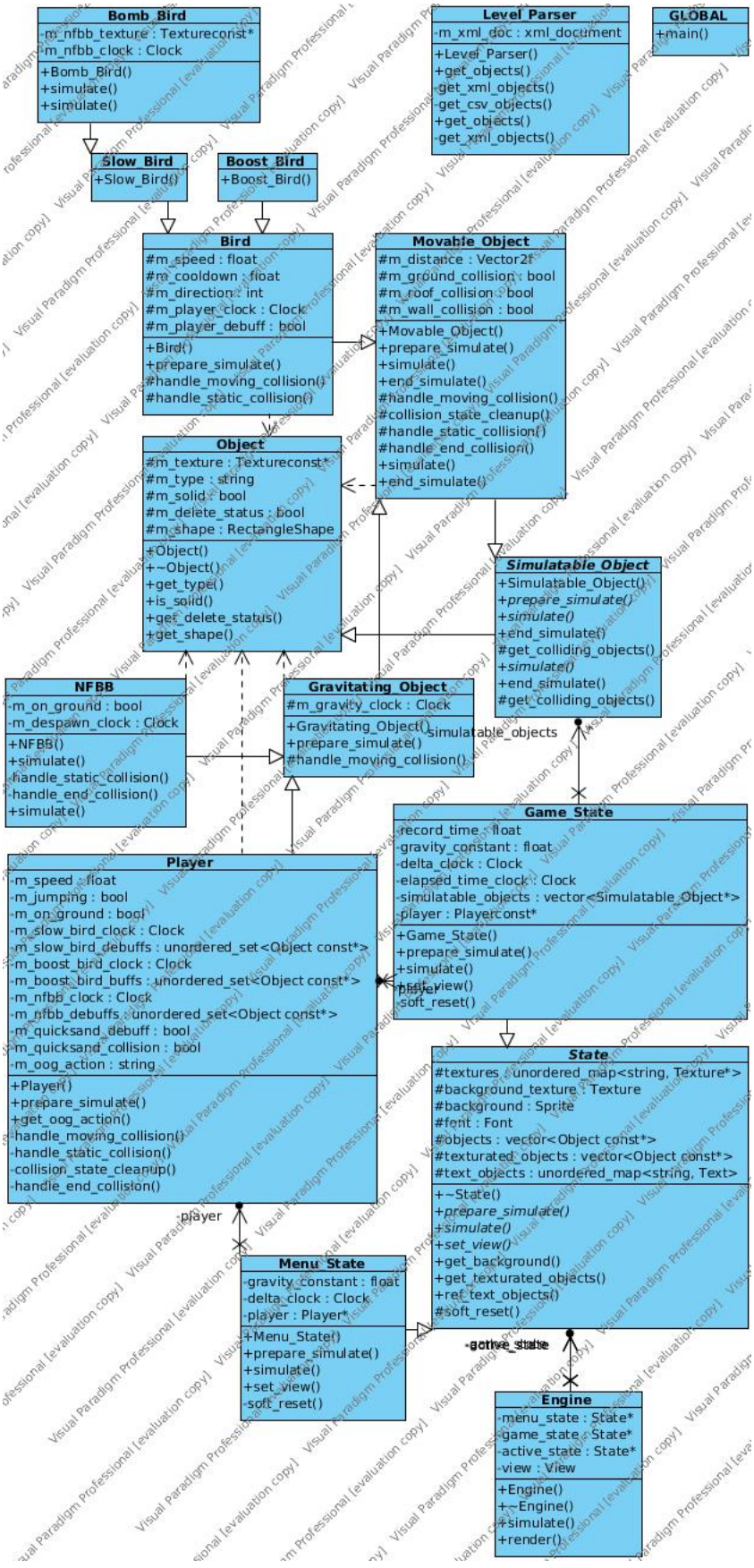
Innan, under (x- och y-förflyttning sker i steg) och efter varje förflyttning kollas sedan kollision. Detta ger en väldigt precis kollisionsdetektion där situationer där objekt som bör kollidera med varandra, men inte gör det, minimeras.

Relationen mellan objekt ser ut som så att ett objekt implementerar en egen version av en simuleringsmetod där den utför diverse logik, varpå den sedan kallar på superklassens implementing av simuleringsmetoden. Detta innebär att all gravitationslogik ligger i klassen Gravitating_Objects simuleringsmetoder, medan exekvering av förflyttningslogik ligger i Movable_Objects simuleringsmetoder.

Klassen Player, som är ett rörligt objekt som påverkas av gravitation, kan således ärva av Gravitating_Object, som ärver av Movable_Object, och sedan enbart implementera enbart den logik som saknas för just klassen Player. Klassen Bird däremot, som är ett rörligt men icke graviterbart objekt, kan ärva direkt från Movable_Object. Sedan kan respektive implementation av fåglar ärva från klassen Bird för att på så sätt ärva all logik som är gemensam för fåglar.

Implementering av kollisionshantering för respektive objekt sker enligt samma kedjemodell som simuleringsmetoderna är implementerade efter.

På nästa sida följer det fullständiga klassdiagramet:


Bomb_Bird
-m_nfbb_texture : Textureconst*
-m_nfbb_clock : Clock
+Bomb_Bird()
+simulate()
+simulate()
Level_Parser
-m_xml_doc : xml_document
+Level_Parser()
+get_objects()
-get_xml_objects()
-get_csv_objects()
+get_objects()
-get_xml_objects()
GLOBAL
+main()
Slow_Bird
+Slow_Bird()
Boost_Bird
+Boost_Bird()
Bird
#m_speed : float
#m_cooldown : float
#m_direction : int
#m_player_clock : Clock
#m_player_debuff : bool
+Bird()
+prepare_simulate()
#handle_moving_collision()
#handle_static_collision()
Movable_Object
#m_distance : Vector2f
#m_ground_collision : bool
#m_roof_collision : bool
#m_wall_collision : bool
+Movable_Object()
+prepare_simulate()
+simulate()
+end_simulate()
#handle_moving_collision()
#collision_state_cleanup()
#handle_static_collision()
#handle_end_collision()
+simulate()
+end_simulate()
Object
#m_texture : Textureconst*
#m_type : string
#m_solid : bool
#m_delete_status : bool
#m_shape : RectangleShape
+Object()
+~Object()
+get_type()
+is_solid()
+get_delete_status()
+get_shape()
Simulatable_Object
+Simulatable_Object()
+prepare_simulate()
+simulate()
+end_simulate()
#get_colliding_objects()
+simulate()
+end_simulate()
#get_colliding_objects()
NFBB
-m_on_ground : bool
-m_despawn_clock : Clock
+NFBB()
+simulate()
-handle_static_collision()
-handle_end_collision()
+simulate()
Gravitating_Object
#m_gravity_clock : Clock
+Gravitating_Object()
+prepare_simulate()
#handle_moving_collision()
simulatable_objects
Game_State
-record_time : float
-gravity_constant : float
-delta_clock : Clock
-elapsed_time_clock : Clock
-simulatable_objects : vector<Simulatable_Object*>
-player : Playerconst*
+Game_State()
+prepare_simulate()
+simulate()
+set_view()
-soft_reset()
Player
-m_speed : float
-m_jumping : bool
-m_on_ground : bool
-m_slow_bird_clock : Clock
-m_slow_bird_debuffs : unordered_set<Object const*>
-m_boost_bird_clock : Clock
-m_boost_bird_buffs : unordered_set<Object const*>
-m_nfbb_clock : Clock
-m_nfbb_debuffs : unordered_set<Object const*>
-m_quicksand_debuff : bool
-m_quicksand_collision : bool
-m_oog_action : string
+Player()
+prepare_simulate()
+get_oog_action()
-handle_moving_collision()
-handle_static_collision()
-collision_state_cleanup()
-handle_end_collision()
player
State
#textures : unordered_map<string, Texture*>
#background_texture : Texture
#background : Sprite
#font : Font
#objects : vector<Object const*>
#texturated_objects : vector<Object const*>
#text_objects : unordered_map<string, Text>
+~State()
+prepare_simulate()
+simulate()
+set_view()
+get_background()
+get_texturated_objects()
+ref_text_objects()
#soft_reset()
Menu_State
-gravity_constant : float
-delta_clock : Clock
-player : Player*
+Menu_State()
+prepare_simulate()
+simulate()
+set_view()
-soft_reset()
active_state
Engine
-menu_state : State*
-game_state : State*
-active_state : State*
-view : View
+Engine()
+~Engine()
+simulate()
+render()

# 5 Externa filformat

De externa filformat som används är vanliga textfiler (.txt) för in- och utläsning av rekord (ett rekord per fil), samt XML-filer genererade av Tiled Map Editor (.tmx) för inläsning av levels. XML-filerna innehåller även inbäddat csv-struktur.

Användning av Tiled Map Editor har varit mycket hjälpsamt för att kunna konstruera banor.