LINKÖPINGS UNIVERSITET

TDP005 Projekt: Obejktorienterat system

Designspecifikation

Författare

Daniel Kaller, danka340@student.liu.se
Emil Gummus, emigu041@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2022
Version 1.0

2022-11-24

1 Innehållsförteckning

Innehåll

1 Innehållsförteckning **1**

2 Revisionshistorik **2**

3 Detaljbeskrivning **2**

4 Diskussion av design **2**

5 Externa filformat **2**

2 Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Första utkast	24/11/22

Tabell 1

3 Detaljbeskrivning

Kollisioner behandlas via att alltid hålla koll på en entitens position via update funktioner. Om en entitet då är på väg att få samma koordinater som en existerande entitet eller objekt tas det upp av check_collision funktionen som finns i alla rörande entiter (spelare och fiender). Vid kollision så kollas det vilka objekt som kolliderade och beroende på vad det är för objekt så sker olika saker.

4 Diskussion av design

Designen är baserad på arv. Genom hela designen är det ett tydligt sträck över hur klassernas funktioner, konstruktorer och variabler samarbetar med varandra. Det finns två viktiga klasser för vårt program, vilket är State och Objects. State är den klassen som presenterar allt i programmet och Objects är grund klassen till alla fysiska objekt i spelvärlden.

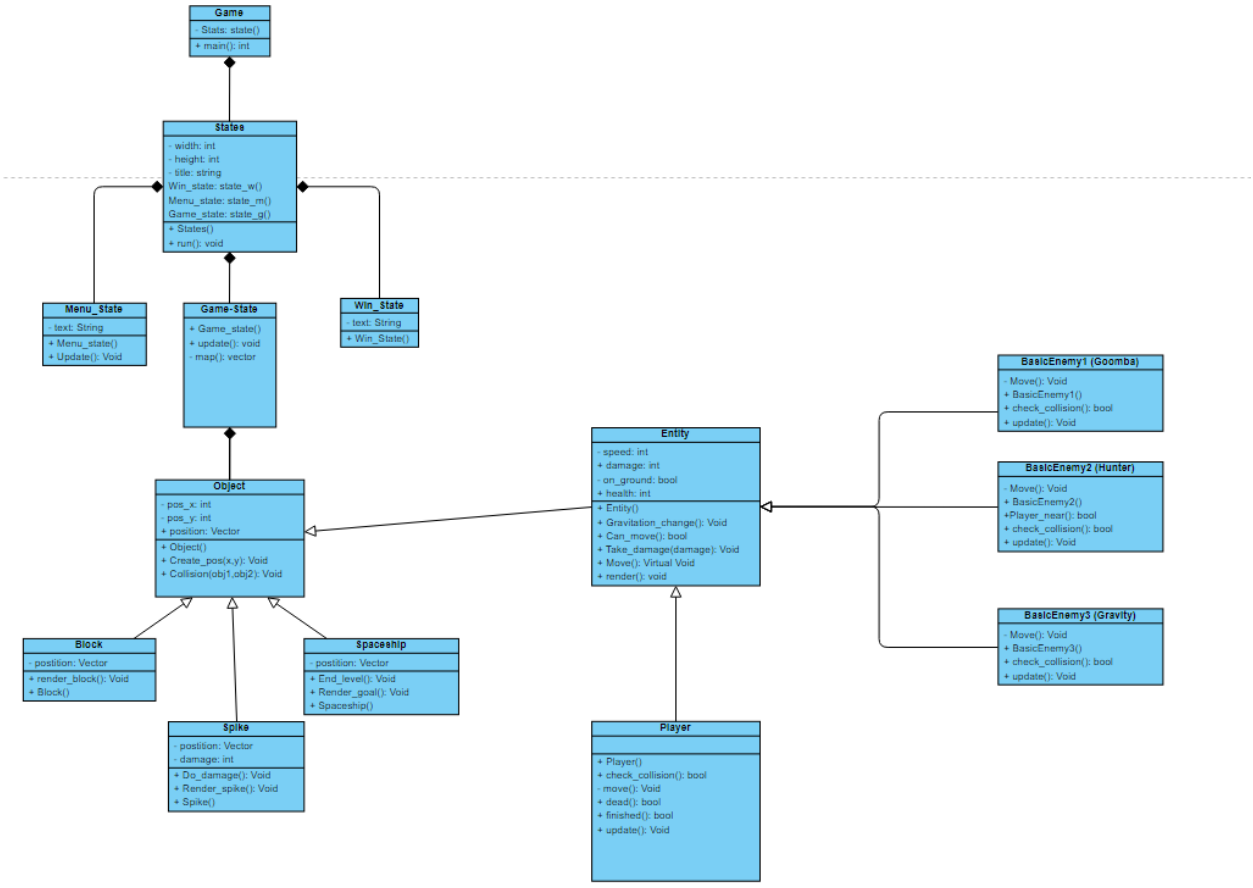
Designen är tänkt att vara konventionell för att lägga till nya objekt i spelet. För objekt har vi en huvudklass som vi kallar för Objects och sedan har vi en under klass till den som heter Entity. För statiska objekt som t.ex en låda skapar man en underklass för Objects och för rörliga objekt som t.ex spelare eller fiende skapas de klasserna under Entity. Anlendningen till att vi inte har en underklass som t.ex static_objects är för att vi inte ser någon användnings potential då Objects klassen redan sparar det som är väsentligt för alla objekt vilket är Position som är en vector där x och y koordinaterna för objektet sparas.

Designens nackdelar är att det kan vara svårare att få in ny funktionalitet som t.ex tid räknare eller att spara hur långt spelaren har kommit på en bana vid en ny start av programmet. Att koppla ihop renderingen av världen samt att få in alla objekt på rätt plats är något som kan bli svårt då vi utgår ifrån att ladda alla objekt från en fil i början av nivån som körs.

I nuläget har vi valt att ha en virtual funktion för move() i klassen Entity. Detta kan skapa ett problem om en programmerare senare vill lägga till en Entity som inte ska röra på sig då vi anger att underklasser till Entity ska innehålla en move funktion. Liknande problem får vi med rörande objekt som en projektil då klassen Entity har vi skapat för levandeöbjekt. Ett alternativ som hade passat bättre för att kunna göra flera olika element/klasser i Entity hade varit att göra den till en klass för rörande objekt, då saker som en projektil hade kunnat skapas som underklass. För vårt program har vi dock inte tänkt göra någon projektil och därför valt att göra klassen Entity enbart till levande objekt.

5 Externa filformat

Spelets olika banor sparas och kan ändras i en fil som heter level.txt. Filen är fylld med banor uppbyggda av olika tecken där till exempel 'B' representerar en mark bit och 'E' representerar en vanlig fiende. Det är från den filen som 'game_state' klassen läser in banorna via att läsa igenom filen och ersätta tecknerna med deras tänkta objekt. Då det finns en klocka som räknar hur lång tid det tar att klara banorna så sparas det snabbaste tiderna och skrivs ut vid ban väljaren. För att göra detta så sparas det snabbast tiderna i en extern fil som kallas fastest_time.txt. När en bana blir avklarad så kollas filen igenom och om den nya tiden är snabbare än den nuvarande snabbaste tiden så updateras filen. Om banan aldrig blivit avklarad sår det Level not cleared".



Figur 1: UML Diagram.