LINKÖPINGS UNIVERSITET

TDP005 Projekt: Objektorienterat system

Designspecifikation

Författare

Martin Kuiper, marku849@student.liu.se
Jim Teräväinen, jimte145@student.liu.se

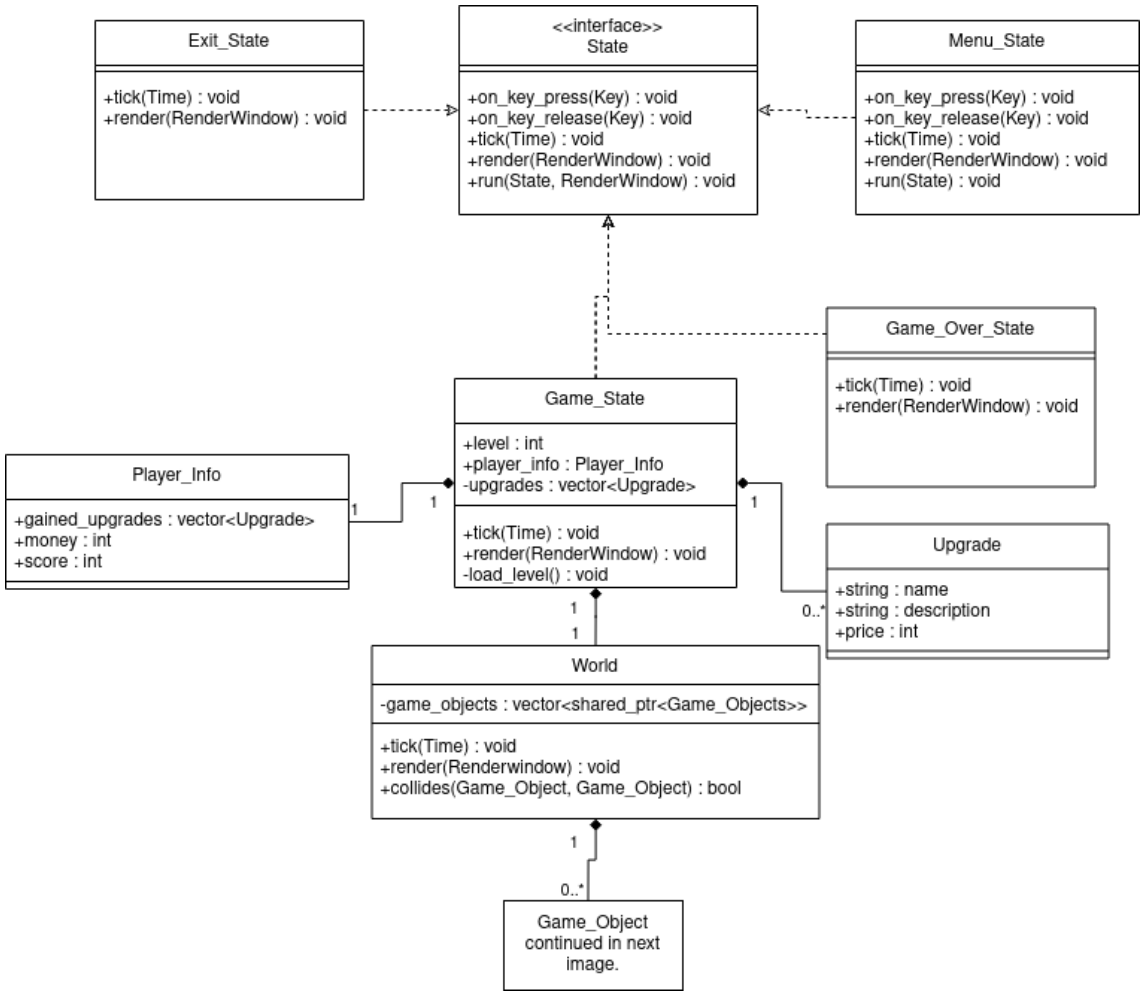
INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2020
Version 1.2

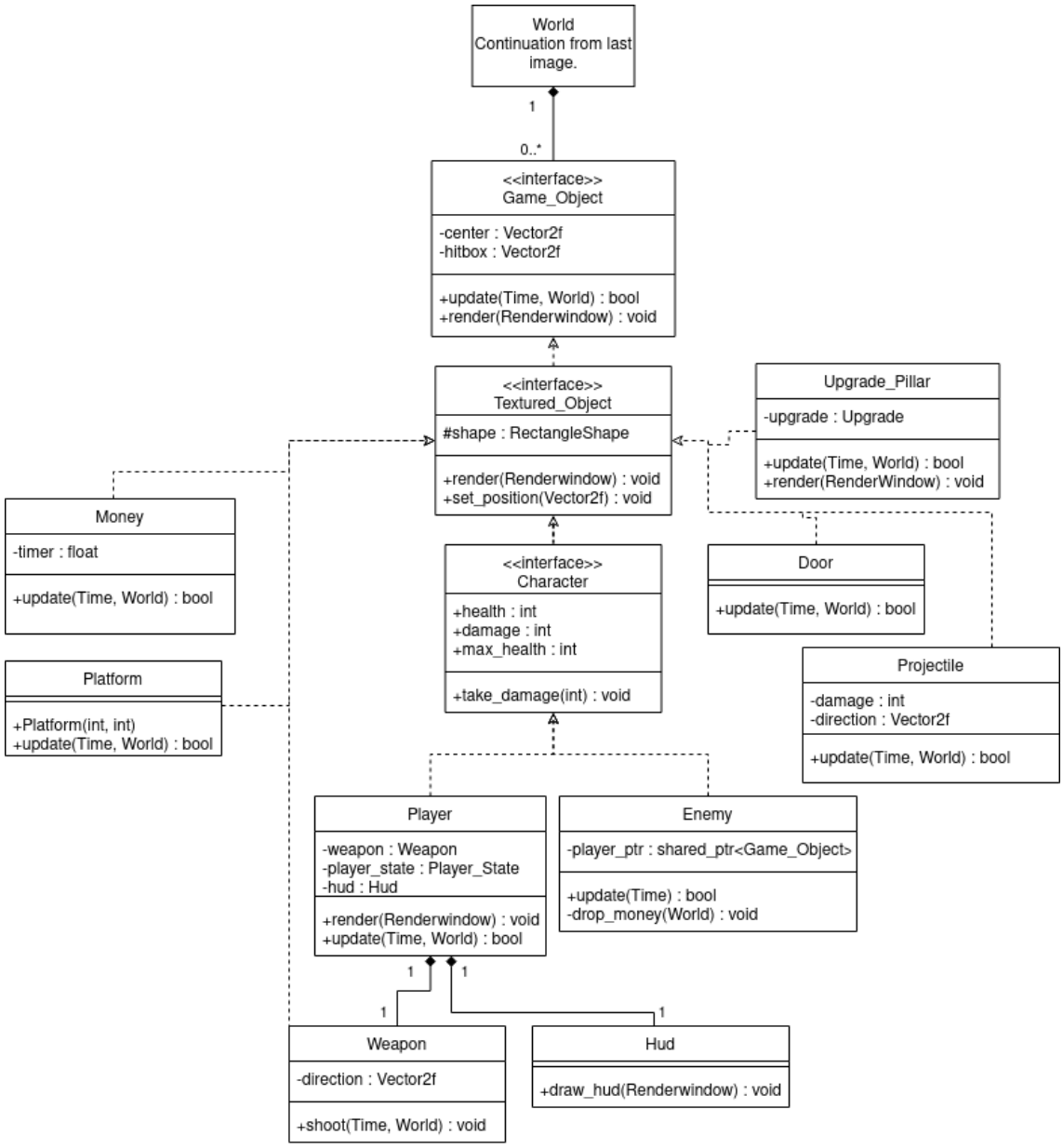
18 december 2020

1 Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.2	Slutgiltig version med ändringar från projektets gång. Klasstrukturen har gjorts om något, och vissa ansvarsområden har flyttats mellan klasser. Fler funktioner har lagts till i klasserna. Upgrade_Station har bytt namn till Upgrade_Pillar. Hud ärver inte längre av Game_Object. Spawner byttes ut mot en funktion i Game_State. UML-diagrammet har delats upp i två delar.	2020-12-18
1.1	Ändringar i UML diagram, tick blev update på många ställen och vissa publika variabler blev private	2020-12-03
1.0	Första utkast för designspecifikationen.	2020-11-27



Figur 1: Övergripande UML diagram för strukturen i spelet fokuserat på State och dess underklasser.



Figur 2: Övergripande UML diagram för strukturen i spelet fokuserat på Game_Object och dess underklasser.

2 Beskrivning av övergripande system

2.1 Stadier

Grunden för spelet ligger i att det skapas olika stadier som sedan körs och hanterar underfunktioner. Dessa stadier representeras i toppen av figur 2 med namn som innehåller 'state'. När spelet startar skapas ett 'Menu_State' där spelaren kan välja mellan olika alternativ. Alternativen leder till olika saker men oftast till att spelet startas igenom att köra 'Game_State'.

'Game_State' kommer att existera under hela körtiden och håller därför koll på viss information som behöver behållas. Detta inkluderar saker som hur många spelplaner spelaren klarat (level) för att skala upp svårighetsgraden. Det håller även koll på viss information som behöver vara konstant när spelaren skapas om för varje spelplan, som hur mycket pengar och vilka uppgraderingar spelaren har. Allting under 'World' kommer att försvinna och skapas igen på nya platser för varje spelplan som laddas in.

2.2 Världen och objekt

Världen representeras av klassen 'World' och hanterar ett obestämt antal 'Game_Object' som alla har en position och 'hitbox'. Dessa objekt bildar tillsammans det komplexa system som är spelet. Alla objekt har en textur och ärver därmed från klassen 'Textured_Object' som kommer hjälpa till att hantera texturerna för objekten. Klassen Game_State har hand om att skapa och hålla koll på finderna som ska dyka upp på spelplanen. Så länge som rätt mängd fiender för nuvarande 'level' inte skapats så kommer den fortsätta att skapa fiender med ett visst tidsintervall, detta också baserat på 'level'.

Klassen 'Character' ärver från 'Textured_Object' och innehåller allt som spelare och fiender har gemensamt, som att de har hälsa och skada. Hälsa representerar hur mycket skada en karaktär kan ta innan den raderas från spelplanen, och i spelarens fall förlorar. Hälsa är individuellt och sparas i alla objekt för sig, det går ned när objektet tar skada. Skada representerar hur mycket hälsa spelaren eller en fiende förlorar när de blir nuddade av något som gör skada.

'Player' representerar spelarkaraktären och hanterar sin rörelse, hur den reagerar på kollisioner, och sitt 'Weapon'. 'Player' kan tillkalla funktionen 'shoot()' i sitt weapon så att det avfyras en projektil i vapnets riktning. Denna projektil ärver sin skada från spelaren så att eventuella uppgraderingar som modifierar spelaren appliceras.

'Enemy' hanterar sin egen rörelse och rör sig mot spelaren för att skada denna.

2.3 Objekt med få funktioner

'Money' lägger till ett visst värde på spelarens pengar när de kolliderar med dem. Vi valde av tematiska skäl att visa pengarna som Själar för spelaren men funktionen är oförändrad.

'Hud' ritar ut hur mycket hälsa och pengar spelaren har så att spelaren enkelt kan se om de är påväg att förlora eller har råd med en uppgradering.

'Platform' hanterar de plattformar som spelaren kan stå på i nivån utöver botten av skärmen.

'Door' skapas på kordinater enligt 'load_level()' när alla fiender besegrats. Genom att interagera med dörren laddas nästa spelplan in.

'Upgrade_Pillar' skapas också enligt koordinater från 'load_level()' när alla fiender besegrats och innehåller en slumpad uppgradering. Denna uppgradering kan läggas till i 'player_info' om spelaren har nog med pengar (själar) för att köpa den och väljer att gör det.

3 Klassbeskrivningar

3.1 Player

Player ska kunna uppdatera sin position beroende på vilka knappar som är intryckta samt vilket state Player har. Player har en överlagrad render()-funktion, då den behöver kunna uppdatera hur den renderas beroende på vad som händer Player-objektet. Funktionen update() kallar alla funktioner som ska köra när spelet uppdateras.

3.1.1 Viktiga datamedlemmar

- state - påverkar hur objektet beter sig när update() kallas.
- weapon - ett objekt av klassen Weapon ägs av Player.
- hud - ett objekt av klassen Hud ägs av Player.
- activate_popup - ett textobjekt som visas när spelaren kan interagera med vissa andra objekt.
- player_info - en referens till Player_Info-objektet som håller reda på saker som spelarens insamlade score och pengar.
- invincible - hur länge Player ska vara osårbar.
- max_jumps - hur många gånger Player kan hoppa utan att nudda marken.
- health - ärvd från Character, minskar när Player tar skada.
- damage - ärvd från Character, påverkar Players vapens skada.
- speed - ärvd från Character, påverkar hastigheten av objektets förflyttning.
- shape - ärvd från Textured_Object, påverkar hur objektets textur ritas ut.
- center - ärvd från Game_Object, bestämmer objektets position på skärmen.
- hitbox - ärvd från Game_Object, bestämmer storleken på objektets kollisionsområde.

3.1.2 Konstruktor

Konstruktorn vi har skapat för player behöver koordinater var objektet ska starta, och en referens till Player_Info-objektet. Resten av datamedlemmarna initieras till standardvärden.

I början av spelet får spelarkaraktären vissa startvärden på sina förmågor från funktionen apply_upgrades. När ett nytt Player-objekt skapas på varje ny nivå tas de köpta uppgraderingarna från Player_Info och modifierar värdena som appliceras av apply_upgrades.

Player-konstruktorn kallar i sin tur Textured_Object-konstruktorn, vilken Player ärver. Den sätter players hitbox automatiskt till samma storlek som texturen. Det är väldigt smidigt i de flesta fall, men då vi vill att Player ska vara animerad och av effektivitetsskäl använda samma textur för alla animationsbilder blir Players hitbox mycket större än önskat. Därför justeras den i Players konstruktor.

Hitboxen justeras även för hattens bredd, då den sticker utanför resten av kroppen.

Funktionen setup_activate_popup() kallas för att ställa in textrutan som ska visas när Player-objektet står framför ett objekt spelaren kan interagera med.

3.1.3 update(Time)

Vid varje uppdatering av spelet kallas Players update-funktion av det ägande World-objektet. Funktionen har i uppdrag att utföra följande:

För att flytta Player kallas move_player().

För att flytta och skjuta vapnet kallas handle_weapon().

För att hantera Players kollisionsbeteende med andra objekt kallas handle_collision().

För att hantera animationerna av Players textur kallas handle_animation().

Datamedlemmen 'invincible' minskas med tiden som passerat sen den senaste uppdateringen om den inte är 0.

Det ägda Hud-objektets datamedlemmar uppdateras så att nuvarande hälsa, pengar och poäng visas på skärmen.

En datamedlem i Player_Info uppdateras så att Game_State vet om spelaren har dött eller inte.

Slutligen returneras spelarens levandestatus till World så att den inte tas bort om den inte är död.

3.1.4 move_player()

För att hantera den horisontella rörelsen kallas funktionen handle_horizontal_move(). Den flyttar karaktären beroende på vilka tangenter som är intryckta.

För att hantera hopp-input kallas handle_jump_input(). Den ser till att när spelaren trycker på hopp-knappen ändras state till 'jumping' om spelaren inte redan hoppat maximalt antal gånger utan att landa.

För att kolla om spelaren ska börja falla från en platform kallas funktionen handle_drop().

Om player_state är satt till 'jumping' kallas jump() för att hantera hopp-rörelsen. Efter en viss tid sätter jump() player_state till 'falling'.

Om player_state är satt till 'falling' kallas fall() för att hantera fall-rörelsen.

Normalt är player_state satt till 'standing'.

3.1.5 handle_weapon()

Players position bestämmer Weapon-objektets position.

Input från tangentbord eller muspekare bestämmer vapnets rotation. Vapnet har en datamedlem 'fire_rate' som påverkar hur ofta det kan avfyras.

Är 'Avfyra vapen'-knappen intryckt och vapnets 'fire_rate' 0 skapas en projektil. Projektilen rör sig i riktningen av vapnets rotation och har en datamedlem 'damage' som sätts till Players 'damage'.

3.1.6 handle_collision(World)

Player frågar World vilka objekt den kolliderar med.

Om den understa delen av Player-objektet kolliderar med den översta delen av en plattform och player_state är 'falling' ändras player_state till 'standing'.

Kolliderar Player med ett Enemy-objekt och 'invincible' är noll kallas take_damage(). Spelaren får även audiovisuell respons på att den tagit skada och 'invincible' ökas med odödlighetsperioden.

Kolliderar Player med ett Upgrade_Pillar- eller Door-objekt visas hjälptexten i 'activate_popup'.

Om 'Bekräfta val'-knappen är intryckt köper spelaren uppgraderingen på Upgrade_Pillar om den har råd och uppgraderingen inte redan är köpt.

Om 'Bekräfta val'-knappen är intryckt vid kollision med 'Door'-objekt signalerar Player via Player_Info till Game_State att nästa nivå ska laddas in.

3.1.7 handle_animation()

Spelarens textur uppdateras baserat på om spelaren aktivt rör sig i x-led eller inte, och vilken riktning den har. Vid rörelse spelas en animationsslinga upp baserat på nuvarande x-position.

3.2 World

World har i uppdrag att hantera alla spelets Game_Object-objekt. World ägs av Game_State, som även kallar vissa av Worlds publika funktioner varje gång spelet ska uppdateras och renderas.

3.2.1 Datamedlemmar

- game_objects - Vector som innehåller smartpekare till alla Game_Object-objekt.
- stored_window - Referens till objektet som hanterar spelfönstret (RenderWindow). Behövs för att hålla rätt på var muspekaren är relativt till fönstret.

3.2.2 Konstruktor

Worlds konstruktor tar in en referens till programfönsterobjektet av klass RenderWindow. Default-konstruktorn är avstängd då stored_window alltid måste initialiseras till rätt värde för att spelet ska fungera.

3.2.3 tick(Time)

Funktionen kallas mellan varje rendering av Game_State, som skickar med tidsskillnaden sedan förra gången funktionen kallats.

Worlds lista av Game_Object-smartpekare loopas igenom och den virtuella funktionen update() kallas i varje objekt. Barnklasserna av Game_Object överlagrar update() och utför de unika handlingar som passar den typen av objekt.

En referens till World-objektet skickas med som parameter i update() för att spelobjekten ska ha tillgång till Worlds medlemsfunktioner.

De får även med tidsskilnaden delta så att de kan justera sitt beteende efter tid, vilket behövs för att förhindra att simuleringen går snabbare eller långsammare beroende på hur snabbt datorn kör koden.

Om World får false i retur av objektets update() vet den att objektet vill raderas. Efter att ha raderat ett objekt minskar word antal gånger loopen ska köras för att unvika segmenteringsfel.

3.2.4 render(RenderWindow)

World loopar efter uppdateringen av alla objekt igenom objekten igen och kallar på deras renderingsfunktion, som är överlagrad på samma sätt som update().

En referens till programfönsterobjektet RenderWindow från SFML-biblioteket skickas till spelobjektet för att det ska kunna rita sig själv på skärmen med hjälp av RenderWindows medlemsfunktioner.

3.2.5 clear_level()

Mellan varje nivå rensas worlds spelobjektslista helt med clear_level() så att nya objekt kan laddas in.

3.2.6 add_front(), add_back() och insert_at()

För att lägga till nya spelobjekt, exempelvis projektiler, kallas någon av tilläggningsfunktionerna.

Då vissa saker ska renderas framför andra saker, som spelarkaraktären framför platformarna, så har vi olika funktioner för att lägga till saker först eller sist i listan.

I ett fall vill vi att uppgraderingspelarna ska läggas framför platformar men bakom allt annat, då kallas funktionen insert_at() av Game_State som själv håller reda på hur många platformar som skapats.

3.2.7 collides() och collides_with()

När ett spelobjekt vill veta om det kolliderar med några andra objekt kallar de funktionen collides_with() och skickar med en referens till sig själva som parameter.

World loopar igenom alla sina spelobjekt och kontrollerar med collides()-funktionen om något objekts hitbox överlappar med det första objektets hitbox. Smartpekare till alla objekt som gör detta läggs till en lista som returneras till spelobjektet.

4 Designens för- och nackdelar

Målet är att hålla designen så simpel som möjligt genom att designa klasser som ärver så mycket funktion som möjligt. I vissa fall kan detta spara mycket jobb då klasserna inte kräver så lång kod. Nackdelen med detta är att designen har mycket beroende av föräldrar i alla klasser och det kan vara svårt att skapa mer nischat beteende.

Då spelet har många komponenter och subsystem som interagerar med varandra kan det lätt bli väldigt komplext och oöverblickbart. För att motverka detta krävs god organisation och bra abstraktionsnivåer.

Om det hade funnits många fler objekt med animationer hade det antagligen varit värt att lägga ut den specialkod som behövs till en separat klass mellan Textured_Object och Character som kunnat heta Animated_Object. Nu bedöms det inte som värt tiden att göra den refaktoreringen.

Systemen för att skapa uppgraderingar och bandesigner har vissa för- och nackdelar som diskuteras i nästa sektion.

En nackdel de båda delar är att det inte finns någon felhantering av filinläsningen, vilket ställer högre krav på användaren att formatera filerna korrekt.

5 Externa filformat

5.1 Bandesign

Funktionen 'load_level()' i 'Game_State' använder textfiler med definerade nivåer. Funktionen läser tecken enligt en fördefinerad struktur för placering av plattformar, spelare, uppgraderingsstationer och avslutningsdörr.

Med detta så kan man enkelt skapa nya nivåer. Detta system har även fördelen att man inte riskerar placera flera saker på samma plats då textfiler inte godkänner flera tecken på varandra.

Nackdelen är att friheten i bandesignen begränsas något.

5.2 Uppgraderingar

Uppgraderingarna läses också in från en textfil enligt en bestämd struktur. Det går enkelt att ändra på pris och hur mycket uppgraderingen ska ändra på spelarkaraktärens förmågor.

Det krävs en något större arbetsinsats med förändringar i koden för att lägga till uppgraderingar som ändrar annan spelardata än de kategorier som redan är implementerade.

Exempelvis, om en uppgradering ska ändra på hur länge en spelare är odödlig efter att ha tagit skada måste förändringar göras i både load_upgrades() i Game_State och apply_upgrades() i Player. Det krävs dock inte stora förändringar i någon av funktionerna och det skulle gå relativt snabbt att implementera.