# Projektbeskrivning

## <Crusader's Conquest>

## 2023-05-11

**Projektmedlemmar:**
Kevin Rintanen Österblad <Kevri781@student.liu.se>

**Handledare:**
Tobias Berglind <Tobbe620@student.liu.se>

## Innehåll

# Projektplan

## 1. Introduktion till projektet

Som projekt tänkte jag utveckla en sidscrollande/shoot-em-up 2D-spel. Ett sidscrollande spel är ett datorspel där spelaren styr en karaktär som rör sig horisontellt genom en spelvärld som visas på en skärm. En "shoot-em-up" är ett datorspel där spelaren kontrollerar en karaktär som ständigt rör sig uppåt eller åt sidan genom en spelvärld fylld med fiender och projektiler. Målet är att skjuta ner så många fiender som möjligt med hjälp av spelarens vapen och undvika att bli träffad av fiendens attacker.

Spelet kommer inte gå ut på något speciellt, utan som spelare kommer man försöka att döda så många fiender som möjligt. För varje fiende man dödar så får man en viss summa pengar beroende på hur svår den är. Pengarna man får använder man för att köpa uppgraderingar till sig själv, t.ex. att utöka antalet liv, hur mycket skada man ger eller ex. hur högt man hoppar. Man kommer också kunna uppgradera och köpa nya fiender som är svårare, men som släpper mer pengar.

Detta projekt har framtagits genom de givna inspirationsprojekten för sidscrollande spel samt för shoot-em-up spel.

## 2. Ytterligare bakgrundsinformation

För att spelet ska vara visuellt tillfredställande så kommer jag att lägga en del tid på grafiken. Jag tänkte exempelvis använda mig utav parallax scrollande bakgrunder. Parallax scrollande bakgrunder är en teknik som används i 2D-spel för att ge en illusion av djup och rörelse i spelvärlden. Det innebär att olika lager av bakgrundsbilder rör sig med olika hastigheter när spelaren rör sig genom spelet. Vanligtvis rör sig bakgrundslagren längs en horisontell axel, där närmare lager rör sig snabbare än lager som ligger längre bort.

Något annat jag tänkte jobba på är att inkludera animationer för fienderna och spelarkaraktären. För att implementera animationer i ett 2D-spel finns det olika metoder och tekniker som kan användas. Just för detta projekt tänkte jag använda s.k. "sprite sheets", vilket är en stor bild som innehåller alla bildrutor för en karaktärs eller ett objekts olika animationer. Genom att byta ut de aktuella bildrutorna i rätt ordning och med rätt tidsintervall kan man åstadkomma en smidig och realistisk animation.

## 3. Milstolpar

| # | Beskrivning |
|---|---|
| 1 | Skapa spelramverk: Skapa en grundläggande struktur för spelet, inklusive hantering av fönster, ljud och spelloop. |
| 2 | Implementera parallax scrollande bakgrunder: Skapa en mekanism för att lägga till och styra olika bakgrundslager som rör sig med olika hastigheter. Se till att de rör sig korrekt baserat på spelarens position och skapar |

| | |
|---|---|
| | illusionen av djup. |
| 3 | Skapa spelkaraktär: Implementera logik för att styra spelarens karaktär och dess rörelse. Lägg till animationer för karaktären, inklusive gång, hopp, och attack. |
| 4 | Skapa fiender och projektiler: Implementera logik för att generera och styra fiender som rör sig och skjuter projektiler. Lägg till animationer för fiender och projektiler för att göra dem levande och reaktiva. |
| 5 | Kollision och skada: Implementera kollisionsdetektion mellan spelarens karaktär, fiender och projektiler. Definiera hur spelaren tar skada vid kollision och hantera eventuella hälsobarer eller livspoäng. |
| 6 | Power-ups och uppgraderingar: Implementera mekanismer för att generera power-ups och uppgraderingar som spelaren kan samla. Definiera hur dessa föremål påverkar spelarens karaktär, till exempel genom att förbättra vapen eller tillfälligt öka spelarens styrka. |
| 7 | Nivådesign och progression: Skapa olika spelområden eller nivåer med olika fiender, hinder och bakgrundsmiljöer. Implementera en progressionssystem där spelaren kan avancera till svårare nivåer efter att ha klarat av tidigare nivåer. |
| 8 | Ljud och musik: Lägg till ljud- och musikeffekter för att förbättra spelupplevelsen. Implementera ljud för skjutande vapen, fiendehantering och bakgrundsmusik som passar spelets atmosfär. |
| 9 | Anpassa användargränssnitt: Skapa ett användargränssnitt med menyer, poängstavlor och eventuella andra spelrelaterade skärmar. Implementera funktioner som att starta om spelet, spara framsteg eller välja olika spelinställningar. |
| 10 | High-score och prestationer: Implementera ett poängsystem och spara spelarens poäng för att möjliggöra high-score-listor. Lägg även till prestationer eller utmaningar som spelaren kan uppnå för att ge extra incitament och återuppspelbarhet. |
| 11 | Boss-strider: Implementera speciella boss-fiender med unika rörelsemönster, attacker och svagheter. Skapa spännande boss-strider som utmanar spelaren och ger variation till spelet. |
| 12 | Finputsning och finjustering: Genomför tester och speltestning för att identifiera buggar och förbättringar. Finputsning och finjustera spelet för att säkerställa en smidig spelupplevelse och balans mellan svårighet och underhållning. |
| 13 | Dokumentation och paketering: Skapa dokumentation som beskriver spelets funktioner, kontroller och spelmekanik för att underlätta för spelare och eventuella framtida utvecklare. Paketera och distribuera spelet för att nå ut till en bredare publik. |

# 4. Övriga implementationsförberedelser

Inför utvecklingen av detta spel tänkte göra lite förberedelser inför hur jag tänkte att klasstrukturen skulle kunna se ut.

Spelet ska vara uppbyggt med hjälp av olika paneler (JPanels) som delar upp allt som man kan navigera genom i spelet, så som huvudmenyn, spelet i sig, och uppgraderingsmenyn. Alla dessa paneler kommer behöva en ”manager” som sköter bytet mellan panelerna.

Spelarna och fienderna kommer att behöva ha egna klasser som håller reda på deras tillstånd, som t.ex. position, antalet liv, hur mycket skada de gör, etc.

Nedan kommer en lista på ett antal klasser som kan komma att behövas. Det kommer förmodligen att tillkomma några klasser under utvecklingens gång.

- GamePanel-klass: En subklass till JPanel som representerar spelområdet där spelet renderas och användarens inmatning hanteras. Innehåller huvudloopen och logik för att uppdatera och rendera spelvärlden.
- Player-klass: Representerar spelarens karaktär. Innehåller logik för spelarens rörelse, kollisioner, skjutande och hälsa.
- Enemy-klass: Representerar fiender i spelet. Innehåller logik för fiendens rörelse, beteende, kollisioner och hälsa.
- Background-klass: Hanterar parallax scrollande bakgrunder. Innehåller logik för att ladda och rendera olika bakgrundslager med olika hastigheter.
- Animation-klass: Hanterar animationer för spelare, fiender och andra spelobjekt. Innehåller logik för att byta mellan bildrutor och skapa en illusion av rörelse.
- PowerUp-klass: Representerar power-ups eller uppgraderingar som spelaren kan samla. Innehåller logik för att ge spelaren förmåner eller förstärkningar.

# 5. Utveckling och samarbete.

Jag kommer att utveckla detta spel på egen hand så tidsanpassning kommer inte att vara ett problem förutom det krockar med andra kurser. Eftersom detta är vårat första ”riktiga” projekt så vill jag ändå lägga en del tid på det då det kommer ge mig stor nytta i att förstå objekt-orienterad programmering på riktigt.

Jag ska försöka göra ett spel som jag själv kommer vilja spela vilket jag tror kommer göra att detta projekt blir väldigt mycket roligare. Men jag måste samtidigt ta hänsyn att jag också har andra kurser att läsa på, så att jag inte hamnar efter i allt annat.

Jag vill göra det möjligt att fortsätta utveckla projektet även efter kursens gång så jag har något kul att göra under sommaren de dagar jag är hemma.

# 6. Implementationsbeskrivning

## *6.1. Milstolpar*

#1 – Delvis avklarad, Inga ljudeffekter finnns i spelet.

#2 – Avklarad

#3 – Avklarad

#4 – Delvis avklarad, Fiender och dess rörelser/beteende (AI) är klart, inga projektiler finns för fiender fiender som ska kunna ha “ranged attacks”.

#5 – Delvis avklarad, spelaren tar skada av att gå in i fiender, men de finns inga projektiler.

#6 – Avklarad

#7 – Delvis avklarad, finns bara en bana för tillfället, har inga planer på att implementera hinder just nu.

#8 – Ej avklarad

#9 – Delvis avklarad, vissa grejer i huvudmenyn är inte klara som load game (det går inte att spara spelprogressionen) och optionspanelen är inte heller klar.

#10 – Delvis avklarad, finns inget poängsystem förutom om man räknar med pengar och uppgraderingar.

#11 – Ej avklarad

#12 – Avklarad

#13 – Avklarad

## *6.2. Dokumentation för programstruktur, med UML-diagram*

- **Övergripande programstruktur**

  Crusader’s Conquest är ett spel som är uppbyggt med flera olika paneler som visar de olika delarna av spelet, som huvudmenyn, spelpanelen eller uppgraderingspanelen. Bytet mellan dessa paneler sköts med hjälp av klassen panelmanager som också instanserar varje panel i konstruktorn.

  Uppgraderingar sköts med hjälp av UpgradesPanel och UpgradesManager. UpgradesPanel är till för att visa upp grafiken för menyn och själva uppgraderingarna sköts med hjälp av UpgradesManager som beräknar alla priser och jämför priserna med spelarens nuvarande pengar vid köp av en uppgradering.

  Spelets framgång sköts av den statiska klassen GameProgress som håller koll på den nuvarande sessionens uppgraderingar samt hur mycket pengar som har

samlats upp. Spelet kan för tillfället inte spara framgång efter att det har stängts ner. Detta är något som ska bli inkluderat i framtiden med använding av JSON data.

Spelet i sig uppvisas med hjälp av GamePanel klassen. GamePanel innehåller också spelets huvudloop. Konstruktorn initierar och konfigurerar olika variabler och objekt som används i spelet, såsom spelbakgrunden, spelaren, fienderna och HUD (heads up display). Metoderna start och stop sköter spelets huvudloop. Start startar en ny tråd (thread) och sätter igång spelets uppdaterings- och ritloop. Stop avslutar dessa loopar och väntar på att tråden ska avslutas. Metoden resetGame återställer spelet till dess ursprungliga tillstånd och skapar nya instanser av spelaren och fienderna. GamePanel sköter också knapptryckningarna som styr spelaren under spelets gång.

**UML**
GameObjects o dess underklasser
Drawable och vilka klasser den implementerar
Panelmanager och underpanelerna

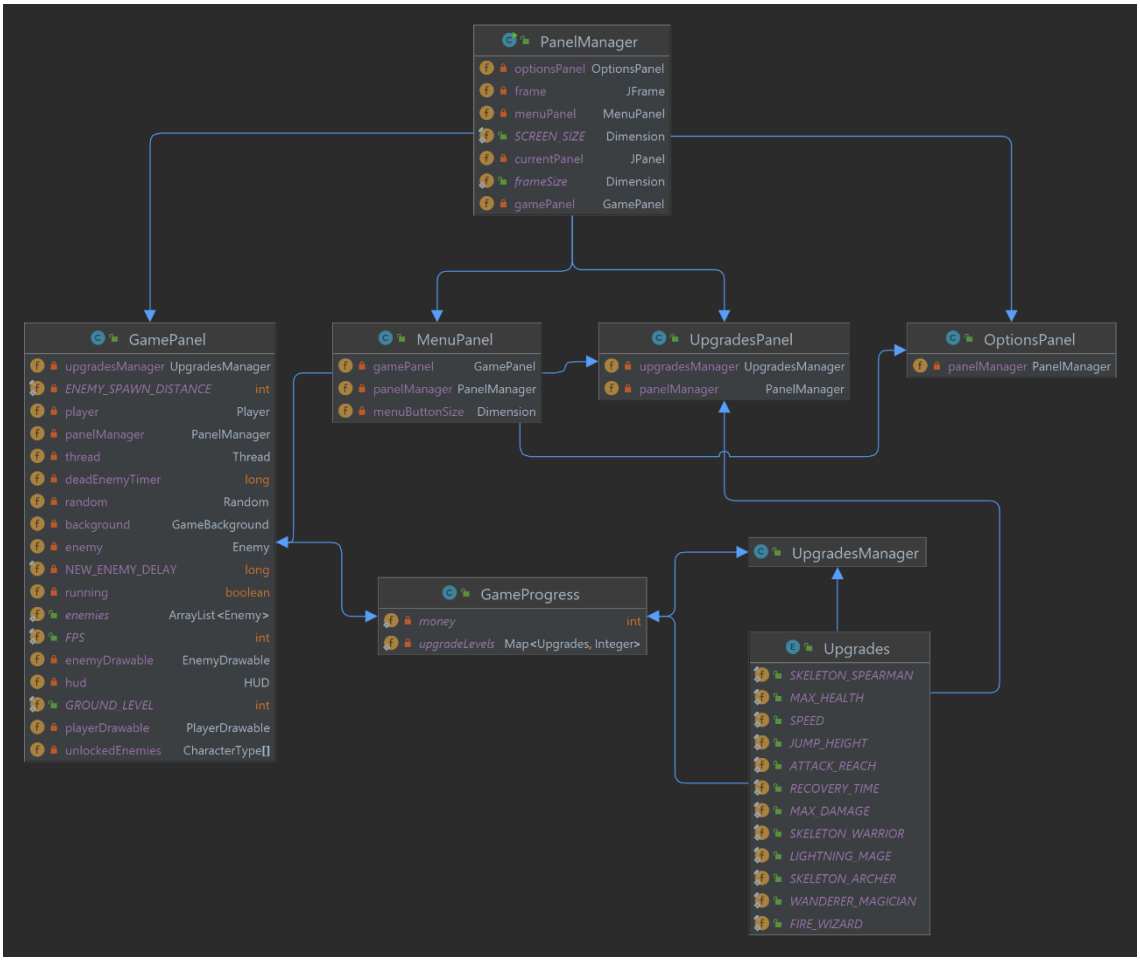
- **Översikter över relaterade klasser**

För att förklara Crusader's Conquests tänkte jag börja från toppen av hierarkin, det vill säga där ifrån spelet startas. Spelet startas med hjälp av klassen PanelManager, det är alltså denna klass som innehåller spelets main metod. Denna klass initierar också alla JPanels (paneler) som går att besöka i spelet i konstruktorn, den initiella panel som visas när spelet startas sätts till MenuPanel. Den innehåller också metoder som gör att byten mellan vilken panel som ska visas kan utföras.

MenuPanel (huvudmenyn) innehåller alla knappar för alla alternativ som ska finnas i huvudmenyn, så som knappen för att starta ett nytt spel, eller att gå till upgraderingsmenyn. Denna klass utökar JPanel och är uppbyggd med boxlayout som "layout manager". Knapparna har varsin "action listener" som använder PanelManagers panelbytes metoder för att kunna byta till det valda alternativet.

UpgradesPanel (uppgraderingsbutiken) hanterar visningen av alla uppgraderingar likt MenuPanel, fast den använder sig utav en gridbaglayout som layout manager. Varje knapp för varje uppgradering i panelen sker med hjälp UpgradesManager som sköter priserna samt checkar ifall upgraderingen kan köpas. Både UpgradesManager och UpgradesPanel använder sig av den klassen GameProgress för att veta hur mycket pengar som har samlats eller vilka uppgraderingar som har köpts hittills. Denna klass innehåller fälten för mängden pengar och köpta uppgraderingar som är statiska för att de ska inte ska starta olika instanser då det bara finns en spelare att hålla koll på dessa egenskaper för.

UML-diagrammet här näst visar den förklarade strukturen. OBS: OptionsPanel är ännu ej klar.



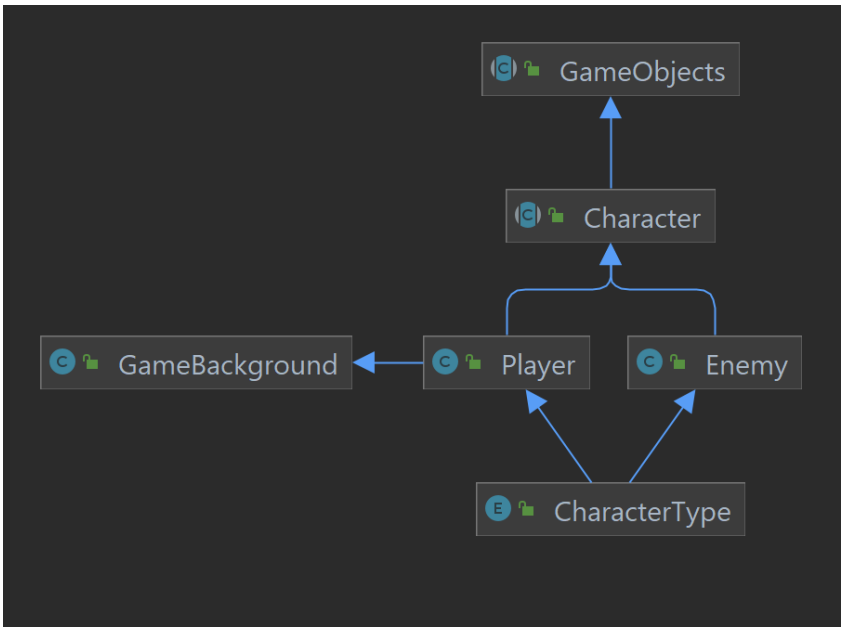

I diagrammet ser man även fälten av varje klass. Upgrades är en enum klass och innehåller alla uppgraderingar som går att köpa. Men här ser man att PanelManager är den klass som sköter vilken panel som ska visas i spelet. Vi ser även lite hur uppgraderingar fungerar.

Näst tittar vi på GamePanel som är panelen där varje GameObject initieras, uppdateras och ritas. Det är alltså här vi har spelloopen som uppdaterar alla spelobjektens position samt ritningen av deras sprites. Varje objekt i spelet är del av typhierarkin under den abstrakta klassen GameObjects. GameObjects sköter det som varje spelobjekt bör ha, det vill säga, x och y position, x- och y-velocitet, bredd och höjd, m.m. Den innehåller också metoder som går att applicera på spelobjekten, som t.ex. applyGravity.

En subklass till GameObjects är den abstraktra klassen Character som utökar GameObjects. Character klassen är till för alla “spelobjekt” som är levande, d.v.s. spelaren och fienderna. Denna klass innehåller fält som definerar levande ting, t.e.x. health, speed eller booleanen dead. Här finns också metoder för springa vänster och höger eller hoppa (moveLeft(), moveRight(), jump()).

Player och Enemy klasserna utökar Character klassen och definerar mer specifika egenskaper för båda två. Player klassen håller reda på spelarens position och kommunicerar med GameBackground för att berätta om spelaren vill röra på sig. För det är inte spelaren som rör sig utan det bakgrunden. Player klassen sköter också interaktioner med fiender i update() metoden. Enemy klassen fungerar likt Player klassen när det gäller position m.m. men den innehåller också metoden Ai som styr fiendernas beteende. Enemy klassen kan

också ta olika CharacterTypes från CharacterType enum klassen. Beroende på vilken den får så får fienden sina egna unika egenskaper i konstruktorn.



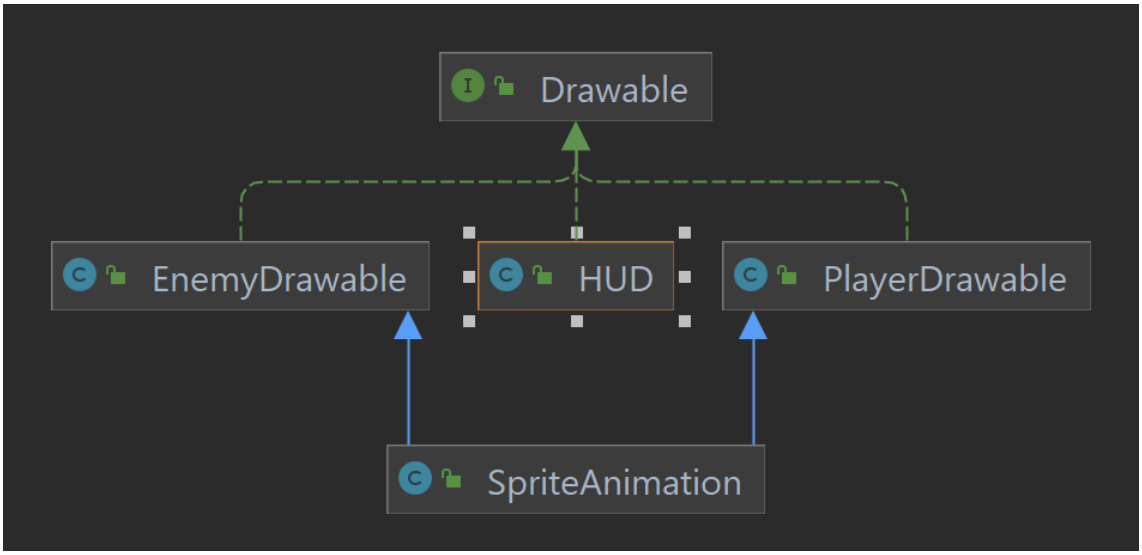

GameObjects sköter alltså inget visuellt utan endast de osynliga egenskaperna för varje spelobjekt så som position, velocitet, bredd och höjd, m.m.

Det ritandet av all grafik sker med hjälp av interface klassen Drawable. I denna typhierarki ingår underklasserna PlayerDrawable och enemyDrawable. Dessa två klasser sköter animationslogiken för spelaren och fienderna. De håller reda på var alla resurser finns för bilderna som ska ritas och sedan ritar spriten på spelarens eller fiendens position. HUD klassen är också del av denna hierarki. HUD'en utökar spelaren och visar information om spelarens hälsa, mängden pengar, m.m.

Beroende på vilken animation som ska ritas så väljs en specifik spritesheet ut från resurs mappen. Denna spritesheet delas upp med hjälp av klassen SpriteAnimation. Konstruktorn i SpriteAnimation klassen laddar in bilden från URL'en som den blir tilldelad. Sedan med hjälp av spritesheetens bredd och höjd räknas antalet bilder ut.

Varje sub-bild är en kvadrat så höjden av spritesheeten definerar sub-bildernas bredd och höjd, alltså blir spritsheetens bredd delad på höjd antalet sub-bilder. Beroende på om karaktären kollar åt vänster eller höger så uppspelningen av sub-bilderna vara reversed då jag har invertat spritesheeten för att få fram bilder på karaktären som är baklänges.

Huvudmetoden i SpriteAnimation är getNextFrame. Den byter till nästa bild i spritesheeten beroende på om variablen animationDelay är mindre eller lika med den senaste tiduppdateringen för att se till att animationen spelas upp i rätt takt. Är det väl dags för nästa sub-bild så adderar den currentFrame variabeln som agerar som sub-bildsindex. Om det är sista bilden i spritesheeten så antingen återställs indexen till första bilden eller så avbryts animationen. Sedan uppdaterar den lastUpdate metoden. Sist räknar den ut x och y för den nuvarande sub-bilden och returnerar den nya utklippta sub-bilden.




Här ser man typhierarkin för alla drawables, gameBackground borde egentligen vara del av denna hierarki, men eftersom det var en av de första klasserna jag implementerade så blev den inte med. Det kanske är fixat i nuvarande version.

# 7. Användarmanual

Spelet startas genom att köra PanelManager klassen genom din IDE. När du fått igång spelet kommer du först presenteras av spelets huvudmeny som innehåller fem olika alternativ: “New Game”, “Load Game” (ej klar), “Upgrades”, “Options” (ej klar) och “exit”. New Game startar ett nytt spel. Load Game laddar in en gammal sparning av spelet. Upgrades leder dig till uppgraderings sidan, där du kan köpa uppgraderingar för dig själv eller uppgradera fienden. Exit avslutar spelet.

Efter att man tryckt på new game hoppar man direkt in i ett nytt spel. Gubben stryrs med hjälp av piltangenterna och mellanslag.

**Kontroller**

- **PIL-UPP** – Spelaren hoppar.
- **PIL-VÄNSTER** – Spelaren springer vänster.
- **PIL-HÖGER** – Spelaren springer höger.
- **MELLANSLAG** – Spelaren slår/attackerar.

Väl inne i spelet så kommer man direkt stöta på en fiende som kommer att försöka döda spelaren. Ifall det är det första spelet som körs så kommer fienden vara en “skeleton warrior” som bara är kapabel till närstridsattacker (andra fiender kan köpas senare). Ditt uppdrag är att döda så många fiender du kan genom att slå dem **MELLANSLAG** tills du dör. För att undvika att dödas så måste spelaren röra på sig med hjälp av piltangenterna: **PIL-UPP, PIL-VÄNSTER, PIL-HÖGER**.

För varje fiende som dödas tjänar spelaren pengar. När du väl dör kommer du presenteras av en “GAME OVER” skärm. För att gå vidare tillbaka till huvudmenyn så trycker man på **MELLANSLAG**.

Efter en lyckad runda kan spelaren med hjälp av sina upptjänade pengar navigera till uppgraderingsbutiken från huvudmenyn genom att trycka på knappen “Upgrades”.

I uppgraderingsbutiken finns det sex olika typer av uppgraderingar för spelaren. “Max Damage”, ökar spelarens skada på fienden. “Max Health”, ökar livet som spelaren börjar med och later en ta mer slag innan man dör. “Speed”, ökar hastigheten som spelaren kan springa. “Jump height”, ökar höjden som spelaren kan hoppa. “Attack Reach”, ökar distansen som spelaren kan träffa en fiende från. Och sist men inte minst “Recovery Time”, ökar tiden som spelaren inte kan ta skada efter att man precis tagit skada. Under varje upgraderingstiteln står det vilken din nuvarande uppgraderings nivå är för den specifika uppgraderingen. Och under det finns knappen som visar priset och köper uppgradering ifall pengarna räcker till. Mängden pengar hittar du över uppgraderingarna benämnt “Balance:”.

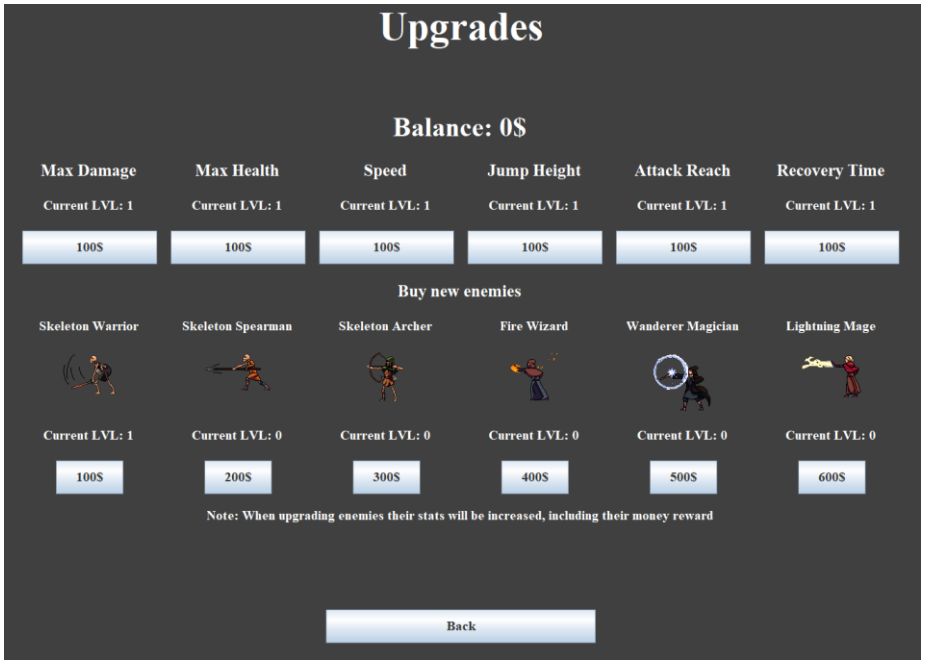
Raden under hittar du titlen “Buy new enemies”. Där hittar du alla sex fiender som går att låsa upp. Som vid spelaruppgraderingarna finner du också nuvarande uppgraderingsnivå på fienderna. Ifall uppgraderingsnivån av en fiende är noll betyder det att fienden inte kommer dyka upp i framtida spel. Tryck på köpknapparna under för att låsa upp dem. Varje gång spelaren uppgraderar en fiende så kommer följande egenskaper för den specifika fienden att uppgraderas: rörelsehastighet, antal liv, given skada samt attackdistans.


Nere i mitten hittar du sedan “Back” knappen som tar dig tillbaka till huvudmenyn. Spelet i nuläget inget riktigt slut, utan det går mest ut på att låsa upp alla uppgraderingar. I framtiden kommer kanske bossar och andra banor att tilläggas.

Bilden till höger -> visar spelets huvudmeny.


Denna bild visar spelet när det är igång.


Såhär ser det ut när en runda är över, när s pelaren har dött.




Här är uppgraderingsbutiken.