Projektbeskrivning

Surviving it with Yuri Doroshenko

2017-03-06

Projektmedlemmar:
Angus Lothian <anglo546@student.liu.se>
Oskar Lundin <osklu414@student.liu.se>

Handledare:
<handledare@ida.liu.se>

Planering

1. Introduktion till projektet

Vi ska utveckla ett spel som handlar om överlevnad. Spelet ska vara i 2D, tile-baserat, och ritas “top-down”, ovanifrån spelaren.

Spelaren rör sig runt i en värld med piltangenterna och interagerar med sin omgivning på olika sätt. Målet med spelet är att utforska och överleva. Det sistnämnda görs genom att hitta mat att äta och vapen att döda fiender med. Spelaren har från början endast hp som går ner när spelaren tar skada, sedan läggs till hunger som går ner konstant och som går upp när spelaren äter. När hunger går ner till noll så börjar man ta hp skada. Sedan läggs även till värme som går ner långsamt och som man tar skada av när den når noll, värmen går upp när man sitter nära en lägereld.

Grunden blir att implementera spelar rörelse, kollisionshantering och en enklare värld för spelaren att rör sig i. I mån av tid lägger vi till fiender, föremål, ett stridssystem med mera.

Det ska finnas både fiender som försöker döda spelaren och passiva djur som inte attackerar spelaren. Exempel på fiender som spelaren kan möta på är en björn eller en varg och exempel på passiva djur är en älg. De dödade djuren ska ge spelaren mat.

Exempel på vapen är svärd, pilbåge och pinne.

Om vi får tid vill vi även implementera slumpmässig terrain generation med Perlin Noise algoritm.

2. Ytterligare bakgrundsinformation

Som sagt är spelet top-down och tile-baserat. Detta innebär att allting ritas ovanifrån. Att det är tile-baserat innebär att spelvärlden är uppbyggd av mindre fyrkantiga brickor. Fördelen med detta är att man kan återanvända samma bilder flera gånger.

Ett exempel på ett sådant spel är The Legend of Zelda: A Link to the Past.

Se bild 1:

Mer info om Perlin noise algoritmen som vi vill använda för slumpmässig terrain generation finns här: https://www.redblobgames.com/maps/terrain-from-noise/

Vi har också tänkt använda A* search algoritm för hur djuren ska kunna hitta runt i världen, mer info om det finns här:
https://www.redblobgames.com/pathfinding/a-star/introduction.html

Vi har också tänkt modella djuren som simpla finite statemachines, mer info om det finns här: https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867

3. Milstolpar

#	Beskrivning
1	Det finns en spelloop där vi uppdaterar logik och ritar saker. Vi kan rita en bild på skärmen på vald position. Vi har en fönsterklass, en ritarklass och en spelklass som hanterar dessa.
2	Den grundläggande klasshierarkin är klar med scener, spelobjekt och tiles. Alla dessa kan ritas på olika positioner. Vi har ett objekt för spelaren som kan styras av tangentbordet och en pekar input som styrs av musen.
3	Spelet har kollisionshantering för tiles. Till exempel finns det vissa tiles som man inte kan gå över. Vi testar med hål tiles som spelaren inte kan gå över.
4	Spelet har kollisionshantering för spelobjekt med hitboxes. Vi testar detta genom att skapa ett trädobjekt som spelaren inte kan gå över.
5	Items finns i spelet. Dessa kan ligga i “containers” som kistor eller i spelarens ryggsäck.
6	Andra spelobjekt, såsom djur och fiender, finns i världen som rör sig omkring fritt. Djuren är programmerade som simpla finite statemachines och de använder A* search algoritm för att ta sig runt i världen.
7	Grundläggande interaktion mellan spelobjekt finns genom meddelande delande. Testas genom att ta skada från något.
8	Konvertering från items till spelobjekt finns (och viceversa).
9	Funktionalitet för strid som bygger på interaktion mellan spelobjekt finns i relevanta spelobjekt som t.ex. i spelaran och i en ny spelobjektstyp björn. Detta kan testas genom att spelaren slåss mot björnen.
10	Vi skapar ett antal flera varelser (både neutrala och fiender), items och växter som ger spelet djup
11	Mer survival element som hunger, kyla, ev. mer, läggs till i spelaren.
12	Grundläggande HUD för hp, hunger, kyla.

13	Grundläggande HUD för inventory.
14	Spelaren kan skapa lägereld.
15	Funktionalitet för serialisering av spelobjekten finns för att kunna spara.
16	Funktionalitet för att kunna ladda sparade spel.
17	Sprites kan använda animationer.
18	Slumpgenererade nivåer med Perlin Noise algorithm.
19	Mer avancerade jakt mechanics som att man följer fotspår.

4. Övriga implementationsförberedelser

Vi tänker ha en spelklass, som innehåller spelloopen. Denna har fält för allting som hör till spelet och uppdaterar dessa. Spelklassen har en fönsterklass, som hanterar fönstret där allting syns.

Allting ritas med hjälp av en ritarklass, som har färdiga funktioner för att rita bilder, text o.s.v. Denna samarbetar med fönsterklassen, förslagsvis genom att allting ritas till fönstret via ritarklassen. För att rita saker skickas ritarklassen till ritafunktionen hos allt som ska ritas.

Spelklassen har också en scenklass som innehåller det som ska synas på skärmen. En scen innehåller en array av Tiles, som motsvarar banan, samt en lista av spelobjekt. Ett spelobjekt är en så kallad aktör i spelet. Det kan vara ett djur, en blomma eller spelaren - allting som kräver mer avancerad funktionalitet och interagerar med saker lite mer än bara genom kollisioner.

För kommunikation mellan olika spelobjekt använder vi oss av ett slags meddelandesystem. Alla spelobjekt har en funktion för att ta emot ett meddelande. Ett meddelande är en klass som har en typ och data. Funktionen receive kan sedan hantera meddelandet som skickas på lämpligt sätt.

5. Utveckling och samarbete

Vi tänker använda oss av Git för versionshantering. För att ha koll på vad som behöver göras använder vi Trello.

I början av projektet tänker vi göra allting tillsammans, för att båda ska få en förståelse för hur allting fungerar. När vi sedan kommer till mer specifika saker kan vi börja dela upp arbetet.

Vi träffas på alla schemalagda tider för att arbeta tillsammans och uppdatera varandra. Annars jobbar vi lite som vi vill. Vi bestämmer vissa tider som vi kör “hackathons” då vi träffas utanför schemalagd tid.

Vi siktar på betyg 5.

Slutinlämning

6. Implementationsbeskrivning

6.1. Milstolpar

Ange för varje milstolpe om ni har genomfört den helt, delvis eller inte alls. (Detta är till för att labbhandledaren ska veta vilken funktionalitet man kan ”leta efter” i koden. Själva bedömningen beror *inte* på antalet milstolpar i sig!)

1. *Det finns en spelloop där vi uppdaterar logik och ritar saker. Vi kan rita en bild på skärmen på vald position. Vi har en fönsterklass, en ritarklass och en spelklass som hanterar dessa.*

Genomfördes helt.

2. *Den grundläggande klasshierarkin är klar med scener, spelobjekt och tiles. Alla dessa kan ritas på olika positioner. Vi har ett objekt för spelaren som kan styras av tangentbordet och en pekar input som styrs av musen.*

Genomfördes helt.

3. *Spelet har kollisionshantering för tiles. Till exempel finns det vissa tiles som man inte kan gå över. Vi testar med hål tiles som spelaren inte kan gå över.*

Genomfördes helt, fast vi testade med sten tiles istället för med hål tiles.

4. *Spelet har kollisionshantering för spelobjekt med hitboxes. Vi testar detta genom att skapa ett trädobjekt som spelaren inte kan gå över.*

Genomfördes helt, fast testades med rävobjekt istället för med trädobjekt vilka lades till senare.

5. *Items finns i spelet. Dessa kan ligga i “containers” som kistor eller i spelarens ryggsäck.*

Genomfördes delvis, Item finns och de kan ligga i ItemContainers som spelarens Inventory, fast inte i några kistor.

6. *Andra spelobjekt, såsom djur och fiender, finns i världen som rör sig omkring fritt. Djuren är programmerade som simpla finite statemachines och de använder A* search algoritm för att ta sig runt i världen.*

Genomfördes helt.

7. *Grundläggande interaktion mellan spelobjekt finns genom meddelande delande. Testas genom att ta skada från något.*

Genomfördes helt.

8. *Konvertering från items till spelobjekt finns (och viceversa).*

Genomfördes inte då vi kom fram till att det vore en för komplex lösning.

9. *Funktionalitet för strid som bygger på interaktion mellan spelobjekt, finns i relevanta spelobjekt som t.ex. i spelaran och i en ny spelobjektstyp björn. Detta kan testas genom att spelaren slåss mot björnen.*

Genomfördes helt, fast vi testade istället genom att slåss mot rävobjekt.

10. *Vi skapar ett antal flera varelser (både neutrala och fiender), items och växter som ger spelet djup*

Genomfördes delvis, vi la till ett antal olika djur som rävar, pingviner, vildsvin och yetis men de är alla fiender. Vi skapade ett antal grundläggande items som kniv, spjut, bär, flinta och stål och gyllene skor. Vi la dock inte till några växter på det sättet som vi tänkte på utan de finns snarare som i bakgrunden på tiles och träd som gameobjekts.

11. *Mer survival element som hunger, kyla, ev. mer, läggs till i spelaren.*

Genomfördes inte. Det enda survival elementet vi implementerade var hp.

12. *Grundläggande HUD för hp, hunger, kyla...*

Genomfördes delvis, vi implementerade grundläggande HUD för hp.

13. *Grundläggande HUD för inventory.*

Genomfördes helt.

14. *Spelaren kan skapa lägereld.*

Genomfördes inte, däremot implementerade vi så att spelaren kan tända eldar med ett flinta och stål itemobjekt.

15. *Funktionalitet för serialisering av spelobjekten finns för att kunna spara.*

Genomfördes inte.

16. *Funktionalitet för att kunna ladda sparade spel.*

Genomfördes inte.

17. *Sprites kan använda animationer.*

Genomfördes helt, även mycket tidigare än planerat.

18. *Slumpgenererade nivåer med Perlin Noise algorithm.*

Genomfördes helt.

19. *Mer avancerade jakt mechanics som att man följer fotspår.*

Genomfördes inte.

20. *Graphics renderas via OpenGL med shapes.*

Genomfördes inte.

Det kan vara värt att nämna att milstolpar 15-20 inte var planerade att hinnas genomföras utan fanns som möjliga mål ifall vi skulle hunnit längre än planerat.

6.2. Dokumentation för programkod, inklusive UML-diagram

6.2.1 Övergripande programstruktur

I programmets “entry point” (finns i klassen Game) sker två saker. Först skapas en instans av Game. Detta initialiserar de fält som är nödvändiga för att spelet ska köras, däribland spelfönstret (GameWindow), världen (Scene), o.s.v. Det är också var mycket av spelets

ihopkoppling sköts, pekare som behövs mellan olika klasser skickas till varandra.

Efter detta kallas metoden Game.start(). Programmet drivs av en en loop som ser till att uppdatera all logik och sedan rita saker på skärmen. Genom att kalla på metoden startas denna loop och spelet är igång. Loopen körs maximalt 60 gånger per sekund. I det fall att loopen skulle gå för långsamt skickar vi med tiden mellan föregående loop och nuvarande in i alla metoder för uppdatering i spelet. På så vis kan vi undvika att spelet beter sig konstigt.

Här följer en mer detaljerad beskrivning (inte komplett:

Objekt som skapas vid start (av Game's konstruktor):

- GameWindow
- Renderer
- Keyboard
- Mouse
- InputHandler
- Camera
- Scene
- Hud (Heads Up Display)

Saker som utförs av spelloopen:

- Uppdateringar
    - InputHandler läser av inputs och ser till att rätt saker sker
    - Scene uppdateras
        - Scenens GameObjects uppdateras
            - Rörelser och diverse interaktioner utförs. Kollisioner hindras.
        - Scenens Camera flyttas så den är placerad rätt
    - Data i Keyboard och Mouse som ska glömmas tas bort
- Ritningar
    - Renderer förbereds för ritning
    - Scene ritas
        - Scenens Tiles ritas
        - Scenens synliga GameObjects ritas
    - Hud ritas
    - Allting visas på skärmen

6.2.2 Översikter över relaterade klasser

Som beskrivet ovan är det klassen Game som hanterar hela spelet. Nedan kommer lite information om alla understående strukturer och system. Vi börjar med Scene och Tile som utgör själva spelvärlden.

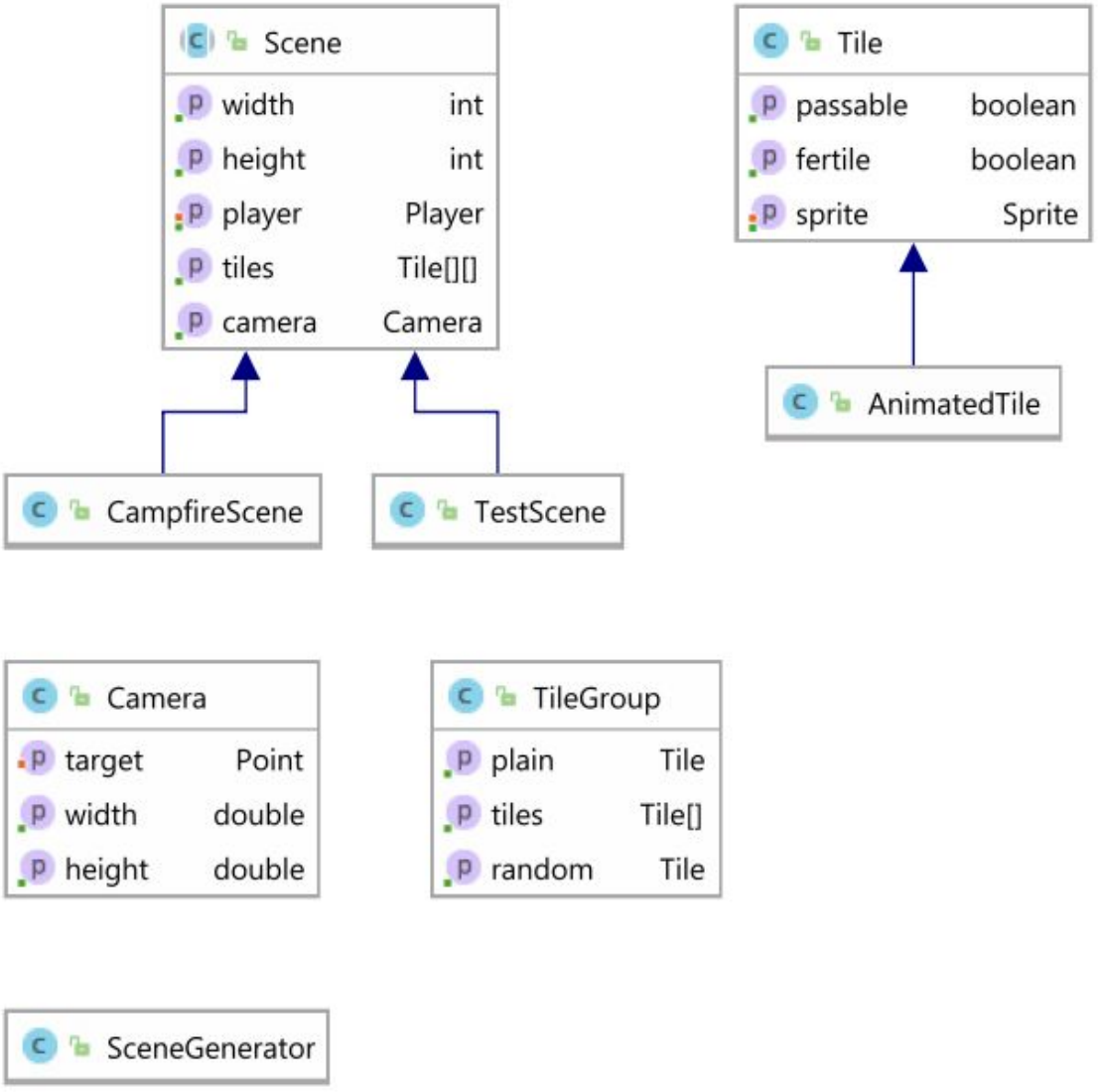


Diagram 1: UML-diagram för Scene, Tile, m.m.

En Scene är i stort sett en tvådimensionell array av typen Tile samt en lista av GameObject (se diagram 2). Alla olika scener bygger på Scene. Som exempel har vi TestScene, som vi har använt för att testa saker under utvecklingen av vårt projekt. För att generera scener kan SceneGenerator användas, som fyller en scen med olika GameObjects och Tiles. En Tile är antingen passerbar eller inte (beskrivs av "passable"). Den kan också vara "fertile", vilket innebär att SceneGenerator genererar vegetation på den.

En TileGroup är helt enkelt en samling av tiles som liknar varandra. Dessa används av SceneGenerator för att se till att Tiles som liknar varandra genereras bredvid varandra.
En AnimatedTile är, som namnet säger, en Tile som har en animerad Sprite (AnimatedSprite).

Kameran används för att se vår Scene och rita det som syns. För att rita används Renderer i paketet "graphics". Klasserna är detta paket är förhållandevis självförklarande och behöver inte beskrivas i detalj.

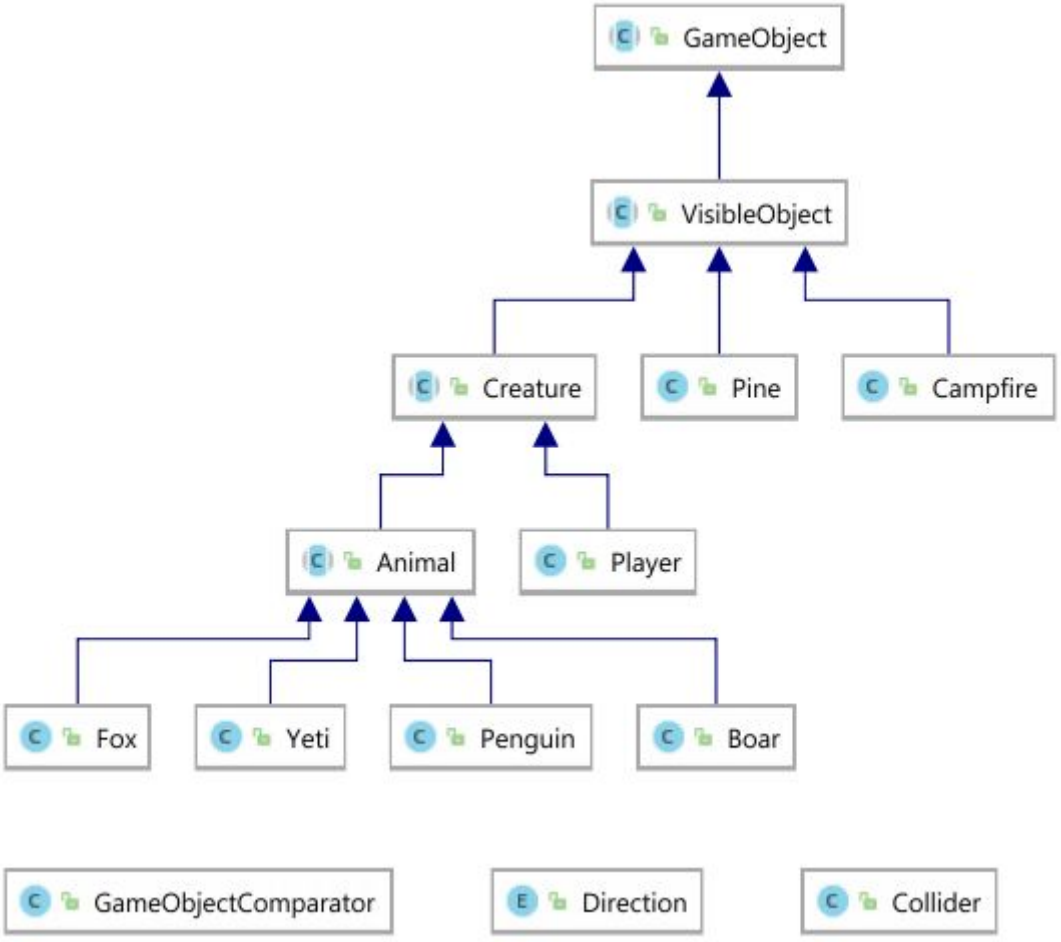


Diagram 2: UML diagram för GameObject och alla relaterade klasser.

Alla aktiva entiteter i spelet är av klassen GameObject. Dessa innehåller en position (x, y), ett fält som beskriver om de är vid liv eller inte, en Scene som de befinner sig i och en Collider som beskriver vilket område de befinner sig i.

Alla synliga GameObject är av typen VisibleObject som också har en Sprite, bilden som ska visas när de ritas på skärmen. Klasser av denna typ är Pine och Campfire. Under VisibleObject har vi även Creature, som representerar lite mer levande varelser. Dessa har hälsopoäng, en hastighet som de rör sig med, en riktning som de rör sig i med mera.

Under Creature har vi spelar klassen Player som har extra funktionalitet för Inventory och Item hantering vilket är unikt för spelaren. Klassen Animal ärver också från Creature och den är superklass över alla djur i spelet. Animal har huvudsakligen en StateMachine som används som djurens beteende i spelet. Klasserna Fox, Yeti, Penguin och Boar ärver från Animal och de skiljs mest med olika data värden.

GameObjectComparator används enbart vid ritning av spelobjekten. Denna sorterar objekten efter deras position i y-led för att se till att objekt som är framför andra ritas över dem. Direction Enum klassen innehåller värden för olika riktningar som Creatures skulle kunna röra sig i och även värden för x, y förändringar som kommer med rörelse i en viss riktning.

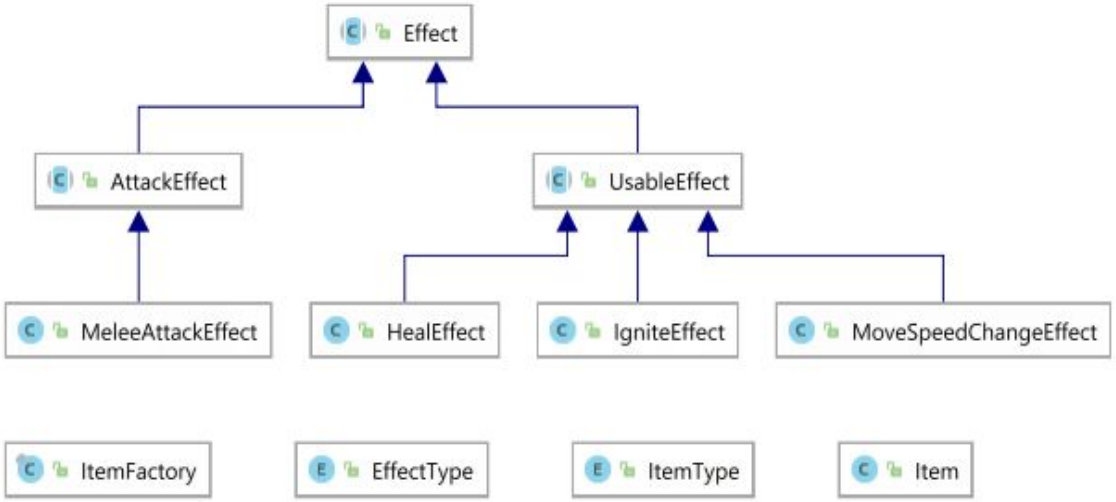


Diagram 3: UML-diagram för Item, Effect och relaterade klasser

När det kommer till föremålen i spelet har vi valt en mer komponentbaserad lösning. Ett Item är kortfattat en samling effekter (Effect) som aktiveras vid olika tillfällen.

Varje Effect har en effekttyp (EffectType) och en pekare till sitt Item. Under detta finns just nu två olika effekter: AttackEffect (aktiveras när föremålet attackeras med) och UsableEffect (aktiveras när föremålet används). MeleeAttackEffect används för föremål som ska orsaka skada på nära håll när de attackeras med. Denna är den enda attackeffekten som har implementerats just nu, men i framtiden kan även andra attackeffekter implementeras så som en effekt för range attacker t.ex..

UsableEffect har många subklasser. HealEffect kan läggas till till ett Item som ska ge användaren hälsopoäng vid användning. IgniteEffect låter föremålet tända lägereldar i närheten och MoveSpeedChangeEffect ändrar på användarens hastighet.

För att skapa ett Item används ItemFactory.createItem(ItemType) som tar in en ItemType. Beroende på ItemType returnerar den ett nytt Item som innehåller rätt effekter.

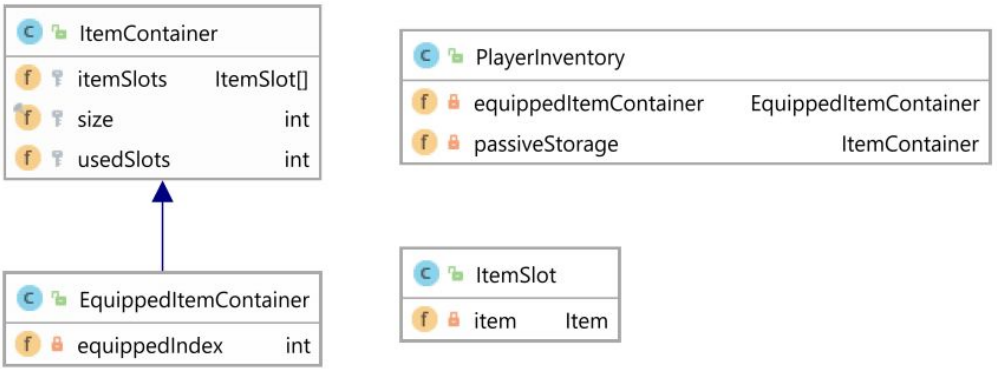


Diagram 4: UML-diagram för ItemContainer och relaterade klasser

Vår lagringsstruktur för föremål är ItemContainer. Varje container har ett antal objekt av typen ItemSlot. En ItemSlot kan innehålla ett Item. Ett exempel på en användning av ItemContainer är spelarens föremål som sparas i PlayerInventory.

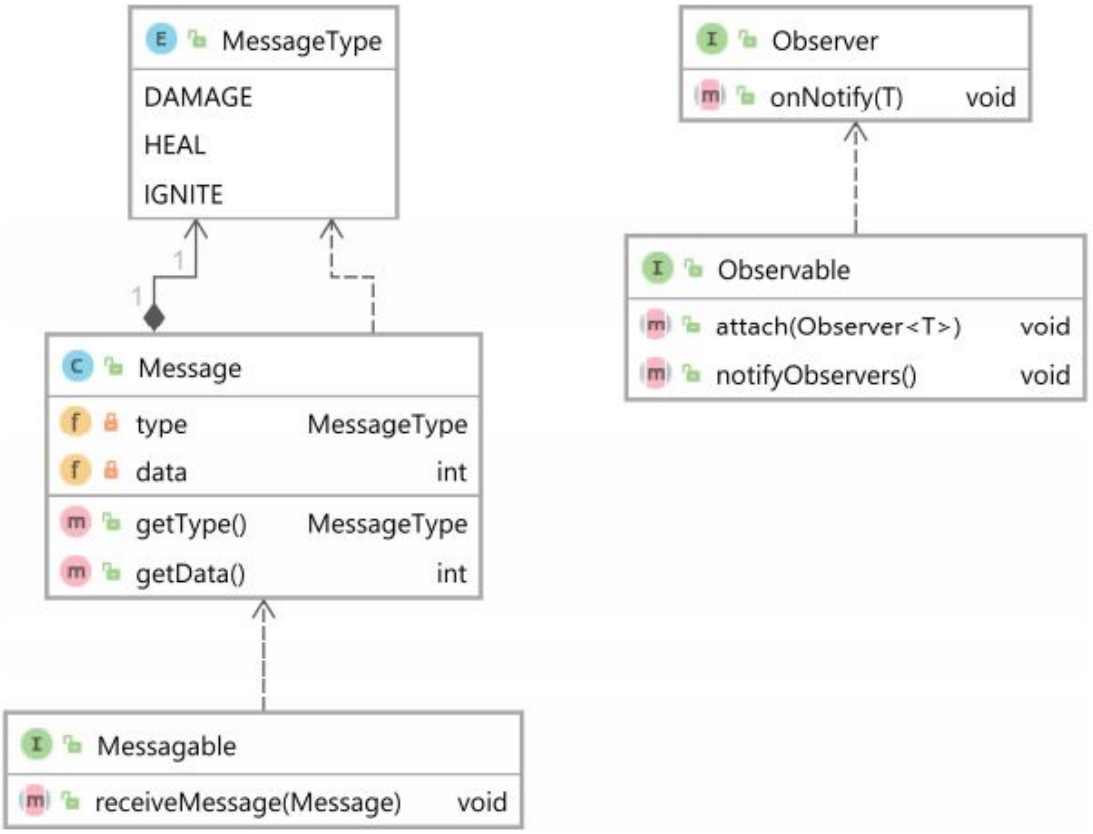


Diagram 5: UML-diagram för de olika kommunikationssystemen i spelet

I alla större projekt behövs robusta system för att sköta kommunikation mellan saker olika delar av programmet. För att göra detta har vi valt att implementera två populära designmönster: *Observatör* och *Meddelanden*.

Objekt som implementerar gränssnittet Messagable kan ta emot meddelanden (Message). Ett Message består av två fält: en MessageType som talar om vad det rör sig om för meddelande och data i form av ett heltal. För att en Creature ska ta skada kan ett meddelande av typen DAMAGE skickas med datan 10. Det enda som krävs är att Creature, i sin receiveMessage(Message) hanterar detta.

Observer och Observable används för att låta ett objekt meddela om det har uppdaterats på något sätt. Ett objekt som implementerar Observable kan ansluta observerare (Observer) till sig. När en Observable vill notifiera sina observerare kallar de på metoden notifyObservers() som för varje Observer ansluten till Observable kallar på onNotify(T). Här kan vår Observer nu uppdatera någonting utefter datan i Observable.

Ett utmärkt användningsområde för detta är vår Head-up-display som hela tiden måste uppdateras när Player uppdateras. För att uppnå detta implementerar Player Observable och vår head-up-display Observer. Player kallar på notifyObservers() när den uppdateras och head-up-display uppdaterar sin information.

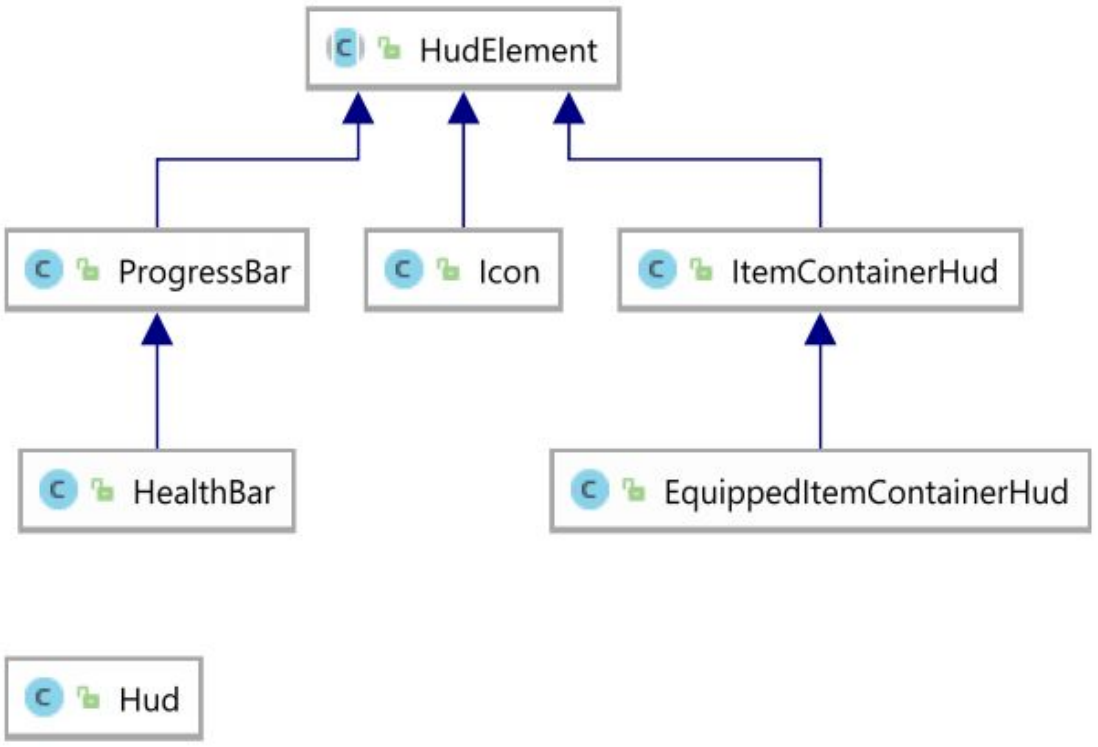


Diagram 6: UML-diagram för de olika spelets heads-up-display

Detta för oss till strukturen för vår heads-up-display. Huvudklassen här är Hud. Hud skapas med en Player som parameter och skapar utifrån denna ett antal element för att visa information om den. Elementen sparas i en lista i Hud och är av typen HudElement. Varje HudElement har en position och storlek på skärmen samt information om hur det ska ritas.

Elementet ProgressBar används för att visa ett förlopp eller liknande. Som exempel har vi HealthBar som visar spelarens hälsopoäng. Elementet Icon är en enkel bild. ItemContainerHud används för att visa en ItemContainer och har subtypen EquippedItemContainerHud som kan visa en EquippedItemContainer (förslagsvis spelarens föremål).

6.3. Användning av fritt material

Vi har inte använt några klassbibliotek i vårt projekt. Vi har dock använt oss av en kodsnutt, en implementation av funktionen "Perlin Noise", skriven av funktionens skapare. Länk till koden finns här: https://mrl.nyu.edu/~perlin/noise/. Vi har modifierat koden en aning för att den ska följa dagens Java-standard lite bättre.

6.4. Användning av objektorientering

1. Objekt/klasser

Objekt och klasser har används överallt i koden, då Java är ett *väldigt* objektorienterat språk. Som exempel har vi klassen GameObject som lagrar ett spelobjekts position, utseende, beteende, m.m. Utan objekt/klasser hade ett spelobjekt kunnat motsvaras av ett id. För varje fält hade det krävts en array eller liknande och id:t hade kunnat vara ett index där fältets värde hittas.

Det sistnämnda må vara snabbare, då man kan placera saker som används tillsammans närmare i minnet. Säg att vi har 500 objekt som ska ritas på skärmen. I en objektorienterad

lösning har objekten kanske ett fält för bilden och positionen, men också 5 andra fält som inte är relevanta för ritandet av objektet. När vi då itererar genom de 500 objekten för att rita dem måste de onödiga fälten passeras varje gång för att hitta bilden och positionen. Detta försämrar prestandan en aning.

Däremot är det mycket svårare att arbeta med och ett utmärkt exempel på hur klasser förenklar saker genom samla data och funktionalitet som hör ihop. Det är svårt att säga vilket som är bättre. I slutändan kommer det till vad man vill prioritera: prestanda eller enkelhet.

En alternativ lösning mot en massa arrays är att helt enkelt använda structs, som finns i de flesta språk. Dock är dessa i jämförelse mycket primitiva och även om man kan uppnå samma funktionalitet kräver det mycket arbete.

2. Konstruktorer

Konstruktorer har används för alla objekt som vi anser behöver initialiseras (vi har också ett antal verktygsklasser med enbart statiska metoder). Ett konkret exempel är alla objekt av typen Sprite (“graphics/Sprite.java”). En Sprite är en yta på ett SpriteSheet (en bild). Den har position som sparas i fälten x och y, en storlek som sparas i fälten width och height samt en pekare till vilket SpriteSheet den finns på. Här är det otroligt smidigt att kunna initialisera alla värden med en konstruktor.

Utan objektorientering och konstruktorer hade dessa värden behöva sparas ett i taget och koden hade blivit mycket svårare att läsa, hantera, skriva. Vissa språk som har structs har dock stöd för så kallade initialization lists, som är mycket lika konstruktorer. Dessa brukar dock inte alls vara lika sofistikerade och det krävs ofta mycket mer kod för att uppnå saker som skulle ta några få rader i ett objektorienterat språk.

3. Typhierarkier

Även typhierarkier har används mycket i vårt projekt. Även här är ett bra exempel GameObject och dess typhierarki. I metoden update(double) i klassen Scene (“scene/Scene.java”) itererar vi igenom alla spelobjekt i scenen och uppdaterar dessa. Trots att objekten är av olika typer (Player, Fox, Boar, e.t.c.) kan samma pekartyp användas för att de ligger under GameObject i hierarkin. Objekten kan behandlas likadant.

Utan objektorientering hade detta varit svårt. Med ett objektorienterat språk kan ett fel i koden som är relaterat till detta upptäckas innan kompilering och körning av programmet. Utan är det svårt och kan resultera i fel under programmets körning. Det blir lättare att man råkar skicka en parameter till en funktion som inte kan hanteras där, men svårt att veta att man har gjort felet innan man kör programmet.

Det krävs i många fall flera implementationer av liknande funktioner och det blir svårt att undvika duplicerad kod. Man kan behöva göra en typecasting och en process som hade varit enkel med typhierarkier blir plötsligt ganska knepig. Här är fördelarna övervägande med objektorienterade språk. En nackdel kan dock vara att man försöker göra typhierarkier av allt och på så sätt gör en uppgift svårare än vad den behöver vara.

4. Subtypspolymorfism

Subtypspolymorfism med sen/dynamisk bindning (för att anropa "rätt" implementation av en metod utan att känna till objektets exakta typ) används exempelvis i GameObject. Klassen GameObject (“gameobjects/GameObject.java”) har en abstrakt metod update(double) som implementeras i subklasserna. När vi uppdaterar alla GameObject i vår scens uppdateringsmetod update(double) (“scene/Scene.java”) bestäms vilken metod som ska anropas dynamiskt.

Genom att använda egenskapen slipper vi ha separata listor för olika typer av objekt som behöver uppdateras. Detta är en enorm fördel. Dock sköts bestämningen av vilken metod som ska kallas på av JVM under programmets körtid vilket kan ses som negativt då det inte är så snabbt som det kunde vara.

Att efterlikna detta beteende utan objektorientering kräver troligen mer kod. I vissa språk kan man uppnå polymorfism på det här sättet men det kräver en hel del arbete. Troligen är det smidigare att lösa problemet på ett helt annat sätt (hur beror på problemet i fråga).

5. Ärvning med overriding

Ärvning med overriding har bland annat används i Player (“gameobjects/Player.java”). Player är en subtyp av klassen Creature, vilken har en funktion setCurrentHealth(int). Denna ändrar på fältet currentHealth i Creature. Som beskrivet längs ner i avsnitt 6.2 har spelet en heads-up-display som måste notifieras då någonting relevant i Player ändras. En av de sakerna är spelarens hälsopoäng. Därför har vi overridat funktionen setCurrentHealth(int) från Creature och skrivit en ny i Player som ser till att heads-up-displayen får reda på att spelarens hälsopoäng ändrats.

Ärvning med overriding är inte riktigt relevant i språk som inte är objektorienterade. Det bästa är nog att inte tänka objektorienterat överhuvudtaget. Om man hamnar i en situation då man behöver “ärvning med overriding” har man troligen strukturerat sitt program fel.

Om man istället pratar om att efterlikna funktionaliteten i sådana språk finns det saker man kan göra. Till exempel kan man använda if-satser för att kolla vilken typ av objekt det rör sig om inuti en funktion. Med många olika objekt blir detta både långsamt och svårt att underhålla. Här är objektorientering en ganska klar vinnare.

6. Inkapsling

När det kommer till inkapsling har vi följt vad som anses vara god vana i Java. Alla fält som hör till klasser är privata (med undantag). För att komma åt fälten utifrån klassen används getters och setters.

Vissa fält har vi valt att göra protected, för att klasser under i hierarkin ska kunna komma åt dessa. Två exempel är width och height i Scene (“scene/Scene.java”). Dessa är med fördel protected då de ofta behövs i subklasserna. Vi har också arbetat för inkapsla så mycket som möjligt och undvikit publika metoder. Till exempel har vi gjort konstruktorer för abstrakta klasser protected (“scene/Scene.java”).

Utan objektorientering är det svårare att inkapsla saker och ting, men ofta görbart. I C kan man till exempel använda void* för att gömma data. Det har dock inte riktigt samma innebörd som i objektorientering och är svårt att jämföra.

6.5. Motiverade designbeslut med alternativ

1. Att använda en arvshierarki träd för våra spelobjekt.

Vi ville åstadkomma en struktur för våra spelobjekt som skulle göra det lätt att skapa nya spelobjekt av olika typer med olika funktionalitet och egenskaper, men som ändå skulle kunna hanteras, interageras med och uppdateras gemensamt genom att lagras i en delad collection.

Vi löste detta genom att skapa en inheritance struktur för våra spelobjekt med GameObject som huvud klass och dess subklasser (se diagram 1). Varje subklass bygger på mer

funktionalitet som tillåter skapningen av enkla och mer komplicerade klasser som Pine, Player och Fox samtidigt som de alla kan sparas i samma Collection i Scene och enkelt kan interagera med varandra.

En rimlig alternativ lösning till detta vore att använda ett Entitiy Component System (ECS) istället för inheritance, där varje spelobjekt skulle vara ett objekt av samma klass Entity som mer eller mindre är en samling av komponenter som bidrar med olika egenskaper och funktionaliteter.

Vi ville från början egentligen lösa problemet med ett ECS då det framkom i olika källor (citation needed??) att det var industristandarden nu för tiden då inheritance kan leda till onödigt komplicerade lösningar och strukturer samt olika sorters problem som för det mesta bara kan lösas med mindre optimala lösningar. Anledningen till att vi tillslut valde en inheritance struktur ändå var att vi inte var familjära med något sådant system tidigare och vi inte kunde hitta några fullständiga källor om hur man skulle implementera och strukturera ett sådant system, vilket gjorde inheritance strukturen till ett enklare val då vi ville komma igång med projektet tidigt och vi båda redan var väldigt bekanta med hur man skulle implementera en inheritance struktur. Efter att vi kom en del in i utvecklingen av vårt projekt så stötte vi på några av de problemen som en inheritance struktur lösning skulle kunna leda till och vi känner nu att det nog vore bättre med en ECS lösning istället.

2. Att använda en finite state machine (FSM) för djurens beteende.

Vi ville åstadkomma en bra struktur på hur djurens beteenden skulle modelleras som skulle vara flexibel och ge realistiskt beteende men inte vara för komplicerad.

Vi löste detta genom att modellera beteendet för alla Animal klasser enligt en FSM där varje Animal objekt äger en StateMachine klass som hanterar uppdateringen och bytandet mellan olika tillståndsklasser vilka implementerar en State interface som vi implementerat. De tillståndsklasser som vi implementerade var AttackState, EscapeState och FollowState vilka utgör djurens olika möjliga tillstånd. De olika tillstånden hanterar detaljerna kring hur djuren beter sig i varje speltick i deras update metod.

En rimlig alternativ lösning hade varit att implementera djurens beteende med en Behavior Tree (BT) ai modell, där djurens beteenden kan representeras av ett riktat träd fyllt med sekvenser av olika villkorskontroller av olika prioritering som bestämmer vad djuret gör i varje tick.

Anledningen till att vi valde en FSM lösning var att det var något vi båda är ganska vana med sen tidigare och att ett beteende modellerat efter en tillståndsmaskin är mycket mer intuitivt än en efter ett träd som följer en föjd villkorskontroller. En FSM lösning passar även mycket bra när beteendet är simpelt och inte för komplicerat då det endast krävs mellan 3-5 states. Vår lösning gör även hanteringen av beteendet enkelt, då om vi vill förändra en specifik del av beteendet så är det bara att ändra i den relevanta State klassen.

3. Att använda java AWT som GUI system.

Vi ville ha ett GUI system som skulle vara simpelt, flexibelt och som skulle ge oss kontroll över ritningen av spelfönstret istället för att vara bundna till ett visst frameworks regler och utan att göra vår kodbas för bloated.

Vi löste detta genom att använda oss AWT frameworket för vår GUI då vår Renderer klass ärver från java.awt.Canvas, som man enkelt kan rita olika former och bilder på vid valda koordinater och med vald storlek genom att använda java.awt.Graphics olika draw metoder. Vår GameWindow klass ärver också från java.awt.Frame klassen vilket utgör själva fönstret av spelet.

En rimlig alternativ lösning som vi funderade emellan vore att använda ett nyare java framework för GUI. Ett specifikt exempel är JavaFX som var huvudkonkurrent till AWT. Det är det senaste GUI ramverket från Oracle och tillåter mer komplex funktionalitet för UI byggnad.

Anledningen till att vi valde AWT över JavaFX var att vi fick intrycket av att det var komplicerat att få upp en simpel 2D canvas att rita direkt på i JavaFX där vi själva kunde styra flödet av programmet och bestämma när det skulle ritas då man verkade tvungen att använda en sorts Timer för JavaFX. Eftersom vi snabbt hittade en lösning i AWT som erbjöd precis det vi sökte så valde vi det. Nu i efterhand så inser vi däremot varför det kanske vore mer optimalt med JavaFX istället då det skulle vara lättare att hantera input direkt på spelobjekt.

4. Att använda pekare till Statiska Sprites och Tiles för rendering.

En fråga som dök upp under utvecklingen var hur vi skulle ladda in resurser till spelet och sedan komma åt dessa under programmets gång. Vi ville ha en smidig lösning som var enkel att använda och inte tog för mycket tid då vi ville fokusera på andra saker.

Vi valde att spara alla resurser som statiska fält i sina respektive klasser. Till exempel sparade vi våra sprites som statiska publika fält i Sprite (delar av en bild) och spritesheets som publik statiska fält i SpriteSheet (själva bilden). På så sätt kan fälten kommas åt smidigt över hela programmet. Framför allt är det väldigt enkelt att använda.

Denna lösning känns tillräcklig då:

- Vi enbart har bilder (bara ett filformat -> inget behov av mer avancerad inladdning)
- Det inte rör sig om så många resurser
- Projektet inte är så storskaligt

En rimlig alternativ lösning är att ha någon slags resurshanterare som vid programstart laddar in alla resurser i minnet. Pekare till resurserna skulle kunna ligga i en HashMap med resurserna som värden och en resursernas namn som nycklarna. Denna lösning hade varit mycket mer flexibel och robust, men tagit extra tid och varit lite överflödig för våra syften. Om vi däremot hade haft flera olika resurser och resurstyper hade det troligen varit mycket bättre. Man hade kunnat automatisera inladdningen för att slippa skriva så mycket kod själv.

5. Att implementera Items som ett entity-komponentsystem.

Vi ville ha ett robust och smidigt system för items som skulle göra det lätt att skapa nya items med olika sorters funktionalitet.

Den första lösningen som vi funderade över var med inheritance där det skulle finnas en superklass Item som alla item klasser ärver från. Det skulle då finnas subklasser för alla olika funktionaliteter som t.ex. Weapon, Tool och Food som sedan skulle kunna ha flera subklasser som lägger på ännu mer funktionalitet.

Den lösningen som vi implementerade i slutet blev däremot ett entitiy-komponentsystem lösning där det finns en Item klass som utgör varje item objekt. Items funktionalitet implementeras i olika Effect klasser som utgörs av en inheritance struktur. Det finns en abstrakt superklass Effect som alla effekter ärver från. Varje Effect har en EffectType enum värde som kategoriserar vilken sorts Effect det är. Item objekt lagrar sina effekter i en dictionary där varje EffectType enum värde kopplas till en lista av effekter av den effekt typen.

Anledningen att vi valde en ECS implementation över inheritance implementationen var att vi fick ett problem att vi ville ge items olika sorters funktionalitet som inte är kopplade med varandra och som skulle behöva ärva från två olika Item subklasser. Därför valde vi en ECS lösning då man kan lägga till vilka effekter som helst av olika typer till samma objekt utan

några sorters problem. ECS lösningen gör det även smidigt att skapa nya items med redan existerande effekter, då det bara krävs att lägga till ett nytt fall i ItemFactory klassens createItem metod istället för att skapa en ny klass.

6. Att hantera positioner i världen och på skärmen som separata x och y värden.

Många objekt i vårt spel har någon form av fält för vilken position de har i x-led och i y-led. För att lagra detta hade vi två alternativ. Det ena var att ha ett fält för x-värde och ett fält för y-värde i varje klass som behövde det. Det andra alternativet var att skapa någon slags klass som håller ett x-värde och ett y-värde (en tvådimensionell vektor) och ge varje objekt som har en position ett fält av den typen.

När vi kom fram till att det sistnämnda var ett alternativ hade vi redan skrivit en hel del kod med separata fält för x och y i varje klass. Vi försökte till en början skriva om koden så att den använde en klass för positioner istället. Detta visade sig vara svårt och kräva mycket arbete, så vi beslutade oss för att fortsätta med de separata fälten.

Med detta uppnådde vi till en början enkelhet men efter ett tag innebar det en hel del duplicerad kod. Till exempel finns det många ställen där vi adderar och multiplicerar positioner med varandra. Med en klass för tvådimensionella vektorer hade detta kunnat göras som en metod istället vilket hade varit kortare och snyggare.

Efter ett tag kom också situationer då vi behövde returnera x- och y-värden tillsammans i metoder. Här var vi *tvungna* att skapa en klass som innehöll dessa två fält (“util/Point.java”). Helt plötsligt hade vi positioner både samlade som vektorer och uppdelade i olika fält. Detta blev inte så snyggt och det hade nog varit bättre att använda en klass för positioner från början.

7. Att lagra Items direkt i ItemSlots i ItemContainer.

Vi ville åstadkomma ett item lagrings system som kunde lagra items.

Till att börja med så implementerade vi ett item lagrings system där varje ItemSlot hade ett ItemStack objekt. Varje ItemStack bestod av ett Item och ett antal av det Item objektet. Det betydde att när man skulle lägga till och ta bort items från en ItemSlot så ändrade man antalet i ItemStack objektet. Detta blev dock intressant kring hur Item pekaren i ItemStack skulle hanteras då det pekar till ett unikt Item objekt som kan ha interna värden lagrat i fields, och då man skulle kombinera två olika ItemStack objekt så kändes det inte som att det fanns en realistisk lösning till vilket utav Item pekarna som skulle behållas.

Tillslut så ändrade vi det till att varje ItemSlot istället direkt pekade till ett Item objekt för att det kändes som att spelet tappade realism i hur Item objekt hanteras med ItemStack lösningen. Dessutom kändes lösningen att en ItemSlot pekar direkt till ett Item som mer passande för ett survival spel då man ska begränsad lagrings storlek i sin Inventory.

8. Att separera rendering från VisibleObjects och istället hantera det i en gemensam Renderer klass.

När det kom till att bestämma vad som skulle rita ut VisibleObjects hade vi två alternativ, varje med olika fördelar och nackdelar. Det första alternativet var att ha en metod draw() i objektet i fråga som i denna ritar ut sig självt. Fördelen med detta är att varje objekt kan rita ut sig på ett unikt sätt. Om till exempel Player hade behövt rita fler saker än sin vanliga bild hade detta varit en fördel. Den stora nackdelen med denna metod är att det leder till mycket duplicerad kod.

Därför bestämde vi oss för att istället sköta ritandet av VisibleObjects helt utanför dem. Scene går igenom alla objekt som syns av Camera och ritar dessa med Renderer. Detta anser vi är

bättre av flera anledningar:

- Alla våra VisibleObjects består av en enskild Sprite i taget. Därför kan alla ritas på samma sätt och koden kan se likadan ut för alla.
- Bättre separation uppnås.
- Om vi behöver ändra hur objekten ritas kan vi göra det på ett och samma ställe.
- Som nämnt tidigare blir det mindre duplicerad kod.

7. Användarmanual

7.1. Starta spelet

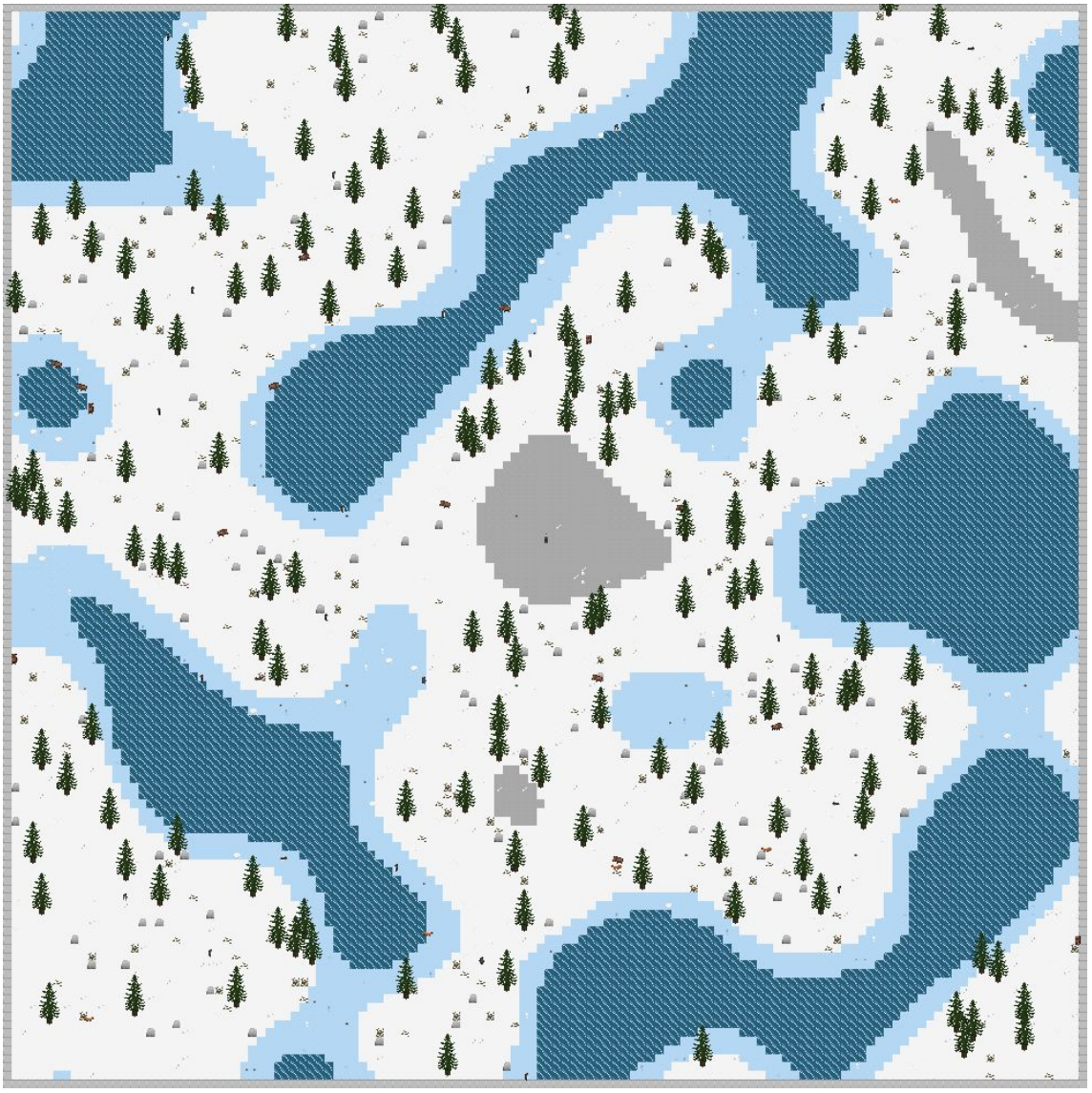
Programmets entry point ligger i klassen “Game” i paketet “survivingit”. Genom att köra denna metod startar man spelet. Altenativt kan man bygga en artifact och köra spelet den vägen. Spelet kan startas i felsökningsläge genom att gå in i klassen “Renderer” (i paketet “survivingit/graphics”) och sätta flaggan “DEBUG” till true.

7.2. Att spela

Spelfönstret

Väl inne i spelet möts man av en gubbe (hädanefter kallad för hjälte) i mitten av skärmen. Hjälten styrs av spelaren och interagerar med världen på olika sätt. För att flytta på hjälten används tangenterna W, A, S och D. Spelets kamera följer hela tiden efter hjälten, men muspekaren kan användas för att se åt olika håll. För att se mer eller mindre av världen kan man zooma in eller ut genom att trycka på pil upp eller pil ner på tangentbordet.

Runtomkring hjälten finns världen. Denna består av snö, is, vatten, granar, djur, m.m. Om man har otur fastnar hjälten vid spelets start. I det fallet är det bara att starta om spelet, då en lösning till detta inte har implementerats. Världen ser olika ut varje gång man startar om. Djuren attackerar andra djur alternativt spelaren om de anses vara svagare och är tillräckligt nära. Detta gör att djuren respektive hjälten tappar hälsopoäng.



Exempel på värld

Nere till vänster finns ett fält som visar hur många hälsopoäng hjälten har kvar. Nere till höger kan hjältens föremål ses. Däribland finns en kniv, ett spjut, ett bär, lite tändstål och ett par gyllene skor. Runt det föremål som är aktivt finns en röd ring. Vilket föremål som är aktivt ändras med hjälp av musens scrollhjul. För att använda ett föremål används vänsterklick och för att attackera med ett föremål används högerklick.

Genom att attackera med kniven skadar hjälten djur som är i mindre närhet mycket och kan på så vis försvara sig på nära håll. Genom att attackera med spjutet så skadar hjälten djur som är i en större närhet än kniven, men gör mindre skada och kan på så sätt försvara sig på längre håll. Genom att använda bäret får hjälten tillbaka några hälsopoäng. Detta kan också uppnås genom att gå till en släckt lägereld och tända den genom att använda tändstålet i närheten av den. En tänd lägereld ger alla varelser i närheten lite hälsopoäng då och då. Genom att använda de gyllene skorna så får spelaren snabbare rörelse hastighet.

Exempel på hjälten i strid (Färglagd, 1959)

När hjälten eller ett djur får slut på hälsopoäng dör de. För hjälten innebär detta att spelet måste startas om för att fortsättas, då funktionalitet för återuppståndelse inte implementerats ännu.

8. Slutgiltiga betygsambitioner

Vi siktar på betyget 5 och känner att vi har uppfyllt alla krav.

9. Utvärdering och erfarenheter

§ *Vad gick bra? Mindre bra?*

○ På det stora hela gick projektet mycket bra. Vi hann göra mycket arbete och är nöjda med det slutgiltiga resultatet.

I efterhand har vi insett att många saker, rent kodmässigt, kan göras annorlunda och bättre. Om vi fick chansen att göra projektet igen skulle vi nog ändra en del saker. Mot slutet av projektet fick vi lite problem med prestandan i spelet som vi hade kunnat slippa om vi gjort annorlunda tidigare. Detta känns dock lite som tanken med projektet. Man kan inte lära sig vad som fungerar bra och dåligt utan att testa.

§ *Vilket material och vilken hjälp har ni använt er av? Har ni gått på föreläsningar? Läst boken? Letat på nätet? Gått på handledda labbar? Ställt många frågor? Vad har ”hjälpt” bäst? Vi vill gärna veta för att kunna vidareutveckla kurs och kursmaterial åt rätt håll!*

○ Till att börja med gick vi på kursens föreläsningar. Dessa var användbara för att förstå hur man ska tänka när man programmerar i Java och hjälpte därför en hel del. Vi har läst mycket på nätet för att försöka komma fram till hur vi borde lösa saker och ting. Utöver detta har vi förlitat oss på tidigare kunskaper.

§ *Har ni lagt ned för mycket/lite tid?*

- Vi anser att vi har lagt ned lagom mycket tid på projektet. Efter att ha jämfört med kurskamraterna kom vi fram till att vi nog ligger bland dem som har lagt ner mest arbete, men vi anser inte att det innebär att vi har lagt ner för mycket.

Vi hann implementera många roliga saker i projektet och det börjar likna en ganska färdig produkt.

§ *Var arbetsfördelningen jämn? Om inte: Vad hade ni kunnat göra för att förbättra den?*

- Vi har haft en jämn arbetsfördelning. Vi har gjort mycket tillsammans och lagt ungefär lika mycket tid på arbetet.

§ *Har ni haft någon nytta av projektbeskrivningen? Vad har varit mest användbart med den? Minst?*

- Projektbeskrivningen har varit hjälpsam, framför allt milstolparna som vi bestämde innan vi började. Det har varit bra att ha en referenspunk för var man befinner sig i projektet och vad som fortfarande har behövt göras. Den minst användbara delen var nog punk 4 (övriga implementationsförberedelser). Med det menar vi inte att den inte var användbar, då det hjälpte att tänka igenom saker lite i detalj innan vi började.

§ *Har arbetet fungerat som ni tänkt er? Har ni följt "arbetsmetodiken"? Något som skiljer sig? Till det bättre? Till det sämre?*

- Vi är nöjda med hur arbetet har fungerat. Vi har sett till att göra stora beslut tillsammans och hela tiden kommunicera för båda ska förstå projektets kod.

§ *Vad har varit mest problematiskt, om man utesluter den programmeringstekniska delen? Alltså saker runt omkring, som att hitta ledig tid eller plats att vara på.*

- Vi har inte haft några större problem med saker utanför det programmeringstekniska. Att hitta plats har gått bra och det har funnits mycket tid.

§ *Vilka tips skulle ni vilja ge till studenter i nästa års kurs?*

- Börja tidigt. Försök ha projektet redo för inlämning en vecka innan slutdatum för att vara säker. Bry er inte för mycket om vad projektet. Se till att göra något som ni tycker är roligt och fokusera på att skriva bra kod.

§ *Har ni saknat något i kursen som hade underlättat projektet?*

- Ingenting särskilt. En sak som hade varit hjälpsam hade varit någon form av exempelprojekt som kan användas som referensram för att se vad som är god praxis när det kommer till att strukturera större program.

§ *Har ni saknat något i kursen som hade underlättat er egen inlärning?*

- Vi tycker att kursen har varit ganska bra på att ge en snabb inledning till Java och på ett förståeligt sätt förklarat de grundläggande koncepten.