LINKÖPINGS UNIVERSITET

TDP005 Projekt: Objektorienterat system

Designspec

Författare

Filip Eriksson, filer358@student.liu.se
Jimmy Björnholm, jimbj685@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

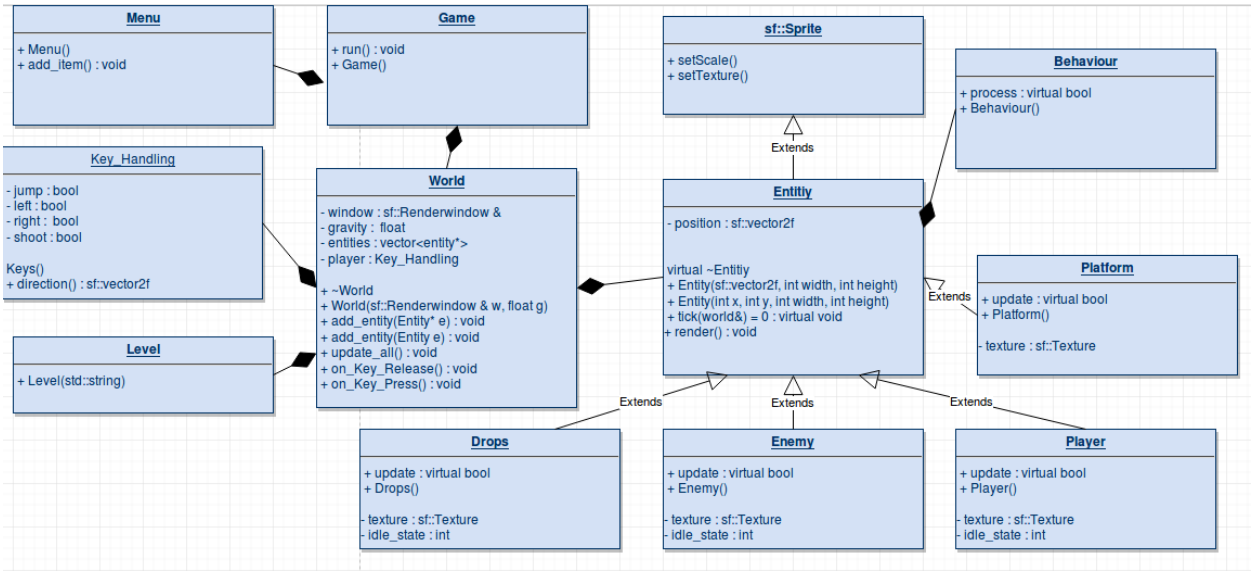
Höstterminen 2017
Version 1.0

2017-11-30

1 Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Första versionen utav designspec klar	2017-11-30

2 UML-Diagram



Figur 1: UML diagram

3 Detaljbeskrivning

3.1 Player

- Namn: Player
- Syfte: Klassen tar hand om spelkaraktärens texturer samt sköter animationer och allt annat som rör spelkaraktären, exempelvis position och hastighet.
- Relationer: Player ärver från Entity. Entity är klassen som hanterar alla objekt i en World. En Entity kan vara både en Player men även en Platform eller Enemy. Player anänder även Behaviour för att avgöra spelmekanik såsom hastighet och om spelkaraktären är död eller inte.

  ```
  Player(Behaviour* b):Entity(b), idle_state{0}{}
  ```

- Konstruktorer: Här ovan är den konstruktor som Player använder. Players konstruktor tar in ett Behaviour och skickar detta vidare till Entity. Player har en idle_state som bestämmer vilken bild player ska ha, detta behövs för animationer. Det är tänkt att konstruktorn senare ska ta in flera argument för exempelvis vilken spritesheet och storlek som ska användas.
- Publika Metoder:

  ```
  bool update(World& w) override {
            killed = behaviour_ptr->process(w, *this);
            return killed;
        }
  ```

  Metoden ovan är för närvarande den enda metoden som Player har, fler metoder kommer implementeras exmpelvis för animationer och för att skjuta eldklot.
- Variabler:

  `sf::Texture texture{};`

  Texture har hand om bilden som visas när spelkaraktären målas ut.

  `idle_state{0};`

  Ovanstående variabel har hand om spelarens nuvarande animations bild.

3.2 World

- Namn: World
- Syfte: Klassen tar hand om världsimuleringen. Den har kolla på alla objekt och får dessa att uppdatera sig och ritar sedan ut dem varje spel tick.
- Relationer: World använder Key_Handling och Level för att kunna hantera input respektive nivå utritande.

```
World::World(sf::RenderWindow &w, float g)
  :window(w), gravity{g}, entities{}{}
```

- Konstruktorer: Här ovan är den konstruktor som World använder. Den tar in 2 argument, ett referens argument till canvaset den ska kunna rita till och en variabel för gravitationen i världen.
- Publika Metoder:

  `~World;`

  World har en default destruktor

  `void add_entity(Entity*);`

  En funktion som lägger till en Entity pekare i entities vektorn för att kunna uppdatera dem senare.

  `void add_entity(Ptr<Entity>);`

  Samma sak förutom att den tar in en shared_ptr istället

  `void update_all();`

  Går igenom och kallar på update på alla objekt i entities

  `void render_all();`

  Gör .draw på alla object i entities

  `void on_Key_Press(sf::Keyboard::Key);`

  Skickar att en key har trycks ner till key_handling

  `void on_Key_Release(sf::Keyboard::Key);`

  Skickar att en key har släppts upp till key_handling
- Variabler:

  `Key_Handling player1{};`

  Har hand om tangent tryckningar och dessas effekter

  `bool run{true};`

  `private:`

  `sf::RenderWindow& window;`

  `float gravity;`

  `std::vector<Ptr<Entity>> entities;`

4 Diskussion

Vi baserade vår design runt tanken att ha en klass som hanterar världen. Våran World klass. Den agerar Observable till alla andra objekt i spelet. Alla andra klasser blir notifierade att de ska uppdatera sig efter det behaviour de har blivit tilldelade. Från denna tanke tog Key_Handling form för att hantera tangent tryckningar.

Vi valde att arbeta så mycket som möjligt med sfml som bas. Därför använder Entity sf::sprite som bas. I den klassen finns positions hantering och textur hantering. Detta gör att vi inte behöver implementera detta själva. Vi planerar också att se om vi kan använda smfls kollissions hantering och därför inte behöva göra den själv.

Sedan valde vi att utveckla en behaviour till varje entity. Dock kan dessa återanvändas så att en viss enemy klass kan dela behaviour. Samma sak med poängbonusar och extraliv.

Nackdelar med våran design är att allting är oerhört beroende av world. Om något ändras i den klassen kan allt annat gå sönder. Därför blir det också problem med includes då kompilatorn skapar include cirklar även om man använder include_guards. Detta komemr vi att kringå genom att använda för deklarationer av klasser.

En annan stor nackdel finns i att vi inte kan påverka utritning etc på en grundligare nivå än den som sfml tillåter. För att kunna göra detta hade vi varit tvugna att skriva mycket av den koden som sfml redan tillhandahåller. Detta kändes onödigt och vi får leva med denna nackdel då. Vi kommer om det någon gång uppstår ett för stort problem med smfl att vara tvungna att gå tillbaka och ändra stora delar av koden om det inte finns något sätt att arbeta runt problemet.

Sedan finns det problem i det faktum att alla platformar och väggar kommer att vara med i entities vektorn och de kommer att kollas för kollision. Detta kommer att sakta ner spelet oerhört och vi behöver hitta ett annat sätt att göra detta.

5 Externa filer

Vi kommer att använda .png som texturer till våra object. Dessa kommer endera att vara hemmagjorda eller royalty fria bilder funna på internet. Dessa laddas in med hjälp av sfml in i en texture class och sparas så att vi ej behöver ladda in dem flera gånger.

Utöver detta kommer vi att använda .level (.txt) filer till våra nivåer så att man kan skapa nya nivåer utan att behöva kompilera spelet igen. Dessa kommer från början att vara uppbyggda med en ratio av 1:1 och om vi har tid så kommer vi att utveckla sätt att komprimera dessa.