LINKÖPINGS UNIVERSITET

TDP003 Projekt: Egna datormiljön

Dokumentmall

Författare

Nils Bark, nilba048@student.liu.se
Hadi Ansari, hadan326@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2020
Version 1.1

2020-12-16

1 Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
0.1	Första utkast skapat	2020-11-27
1.0	Uppdatering av utkast enligt feedback från handledare	2020-12-07
1.1	Sista uppdatering inför inlämning	2020-12-16

2 Beskrivning av Entity

Nedan kommer en lista över alla funktioner och variabler som finns i klassen `Entity`.

2.1 Variabler

- `sf::Vector2f location`: Innehåller två koordinater av typen float som representerar spelarens position på spelplanen.
- `sf::Sprite sprite`: Fungerar som figurens kropp och kan täckas av en textur.
- `float speed`: Används för att bestämma objekts hastighet.
- `int health`: Bestämmer objektets livspoäng.
- `int width och int height`: Bestämmer objektets storlek.

2.2 Funktioner

- `virtual void Tick(sf::Time const & delta, World & world)`: Uppdaterar objektens beteende. Den implementeras av de härledda klasserna och beteer sig där på olika sätt. Till exempel för Big plane så subtraheras x-koordinaten med ett värde som är bundet till hastigheten och tiden som har gått sedan förra uppdateringen.
- `virtual void render(sf::RenderWindow &window) = 0`: Ritar ut spelobjektet på skärmen.
- `bool kill_me()`: Kontrollerar om ett objekt ska förstöras (till exempel om det befinner sig utanför spelplanen) och returnerar true om så är fallet.
- `virtual void collision(vector<entity*> objects, World & world) = 0`: Tar en lista som innehåller entity-pekare till alla andra spelobjekt, utöver det som kallat på funktionen, och kontrollerar vilket av dem objektet har kolliderat med. Har flera olika effekter beroende på vilket objekt som har kolliderat med ett annat. Till exempel förlorar spelaren liv om den kolliderar med en fiende.
- `virtual string get_type() = 0`: Returnerar objektets typ i form av en sträng.
- `virtual sf::Sprite get_sprite() const`: Returnerar objektets sprite.
- `int get_health() const`: Returnerar objektets liv.
- `virtual void freeze()`: Pausar alla spelets timers när spelet går till `pause_state`.

`Entity` är en `abstract class` som alla underståenden klasser i spelet ärver ifrån. Detta innebär att alla dess variabler och funktioner är tillgängliga direkt i de härledda klasserna. `Entity` låter oss därmed uppdatera alla andra objekt i spelet. Detta sker via funktionen `tick()` som tar och uppdaterar positionen för alla objekt på skärmen plus andra beteenden beroende på objektet. `Tick()` är en `virtual` funktion som kan överskrivas i de härledda klasserna. Tack vare tidsberäkningen som sker inuti funktionen är vi oberoende av spelets bildfrekvens och tiden det tar att hantera alla händelser i loopen. Alla datamedlemmar förutom `location` i basklassen kommer att initieras av de härledda klassernas constructor. till exempel kommer spelarens `speed`

att initieras när spelaren skapas och inte innan. Exempel på extra beteenden som uppdateras av tick (utöver uppdatering av position) är till exempel att `Big_Plane` kan lägga till en `enemy_bullet` på spelplanen

3 Beskrivning av Player

Nedan kommer en lista över alla funktioner och variabler som finns i klassen `Player`.

3.1 Variabler

- `float shoot_speed`: Bestämmer tiden som måste passera efter att spelaren har skjutit innan den får skjuta igen.
- `sf::Clock shoot_timer`: Mäter tiden som sen kontrolleras mot `shoot_speed`.
- `sf::Time shoot_time`: Sparar tiden från `shoot_timer`.
- `bool shield`: Är true om spelaren har shield-effekten aktiv.
- `sf::Clock shield_timer`: Räknar tiden från och med att spelaren plockade upp en shield-powerup
- `sf::Time shield_time`: Sparar tiden från `shield_timer`.
- `bool tripleshot`: Indikerar om spelaren har en triple-shot powerup aktiv.
- `sf::Clock triple_timer`: Räknar tiden från och med att spelaren plockade upp en tripleshot-powerup.
- `sf::Time triple_time`: Sparar tiden från `triple_timer`.
- `static bool invincible`: Indikerar om spelarens ödödlighetsperiod är aktiv.
- `sf::Clock invincibility_timer`: Räknar tiden från och med att spelarens odödlighetsperiod aktiverades:
- `sf::Time invincibility_time`: Sparar tiden från `invincibility_timer`.
- `bool give_invincible`: Indikerar om spelaren ska ges sin odödlighetsperiod.
- `bool freeze_state`: Indikerar om spelet är pausat.

3.2 Funktioner

- `string get_shield_time()`: Returnerar den kvarstående tiden från `shield_clock`.
- `bool has_shield()`: Returnerar true om spelaren har en shield aktiv, annars false.
- `static bool is_invincible()`: Kollar om spelaren är odödlig.
- `void cheat()`: Kontrollerar om fusk ska aktiveras (via korrekt tangent).
- `void set_position()`: Uppdaterar spelarens position.
- `void update_status()`: Uppdaterar spelarens powerups. Till exempel om spelaren fortfarande ska ha dem aktiva.
- `void shoot()`: Kontrollerar att spelaren tillåts skjuta, och gör i så fall det.

`Player` representerar det objekt som användaren kommer att ha kontroll över under spelets gång. Det är en härledd klass av `Textured_object` (som i sin tur härleds från `Entity`) och ärver därmed alla datamedlemmar och funktioner därifrån. Den har också en känna-till-relation till `player_bullet` och `Power-up`. Player överskriver `tick()` för att kunna uppdatera sitt specifika beteende. I spelarens konstruktor initieras spelarens

alla datamedlemmar. `tick()` returnerar alltid true när det går att uppdatera spelaren. Annars returnerar den false vilket indikerar att spelaren har förstörts.

Spelarens kollision med andra objekt kontrollerar ständigt under spelets gång via `collision()` från `entity`. Baserat på vilket objekt som spelaren kolliderar med kan flera olika saker hända. Till exempel tar spelaren skada vid kollision med fiender (förutom då spelaren har en shield aktiv), medan den diverse aktiverar effekter vid kollision med power-ups. Den enda kollisionen som ignoreras av spelaren är med dess egna skott (och fienders skott ifall spelaren har en shield aktiv).

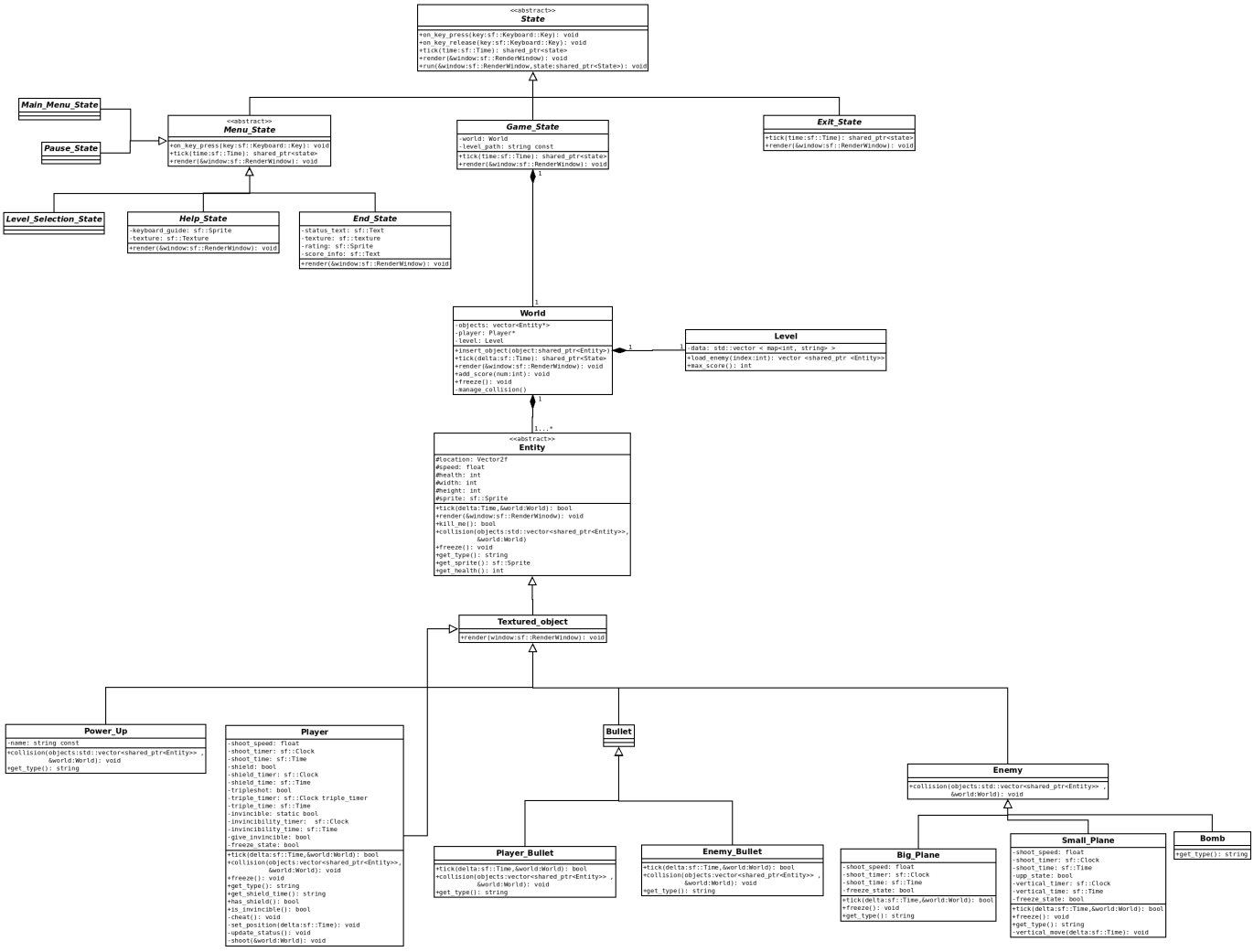
4 Diskussion

Flera olika för- och nackdelar har visat sig under projektets gång. Här kommer vi att nämna och diskutera några av de viktigaste vi har lagt märkte till. Först ut är `collision` som fungerar på ett mycket ineffektivt sätt. Den kontrollerar nämligen vid varje uppdatering om varje objekt har kolliderat med något annat objekt. Den kontrollerar då kollision mot alla andra objekt på spelplanen. Även om det första objektet som kontrolleras är det som kolliderar med fortsätter den att gå igenom alla möjliga objekt. En fördel med vårt system är dock att det låter båda objekten i en kollision påverkas på olika sätt om till exempel båda objekten ska ta skada vid en kollision.

Det sker viss koduppprepning i vår `Enemy`-klass eftersom vi har seperata underklasser för varje fiendetyp. Här kunde vi istället ha designat spelet så att alla, eller åtminstånde de flesta, fiender kan skapas från samma klass. Dock togs detta problemet upp för sent för att kunna åtgärdas på ett tidseffektivt sätt. I nuläget gör denna design det dock lätt att lägga till fler fiendetyper, speciellt ifall de har nya egenskaper.

5 Externa filformat

Vi skapade ett eget filformat .sw (som i praktiken är en enkel textfil) där vi sparar varje nivå. Texten i filen skrivs i formatet: [y-koordinat] [fiendenamn (eller None)]. Detta läses sedan in av programmet som då lägger in alla fiender på spelplanen samtidigt tills de når ett None (då tar det en paus på 2 sekunder). På så sätt kan man skapa vågor av fiender.



Figur 1: UML-diagram över spelet