TDP005 Projekt: Objektorienterat System

Designspecifikation

Författare
Maximilian Bragazzi Ihrén, maxbr431@student.liu.se
Markus Lewin, marle943@student.liu.se
David Spove, davsp799@student.liu.se

Höstterminen 2014
Version 1.2

1. Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Skapad inför inlämning	141215
1.1	Utvecklat diskussion, gjort om diagram	150113
1.2	Utvecklat diskussion med lite över ett halvt A4.	150119

2. Detaljbeskrivning av Player-klassen

2.1. Syfte

Syftet med Player-klassen är representera det objekt spelaren styr med hjälp av tangentbordet.

2.2. Arv

Player-klassen ärver av klasserna CollisionBox och Entity, som i sin tur implementerar objekt av typen Sprite och Weapon.

CollisionBox används av Player-klassen för att kontrollera alla kollisioner, t.ex. mellan ett Player-objekt och ett Bullet-objekt, eller ett Player-objekt och ett Pickup-objekt. Player-klassen ärver följande funktioner av CollisionBox-klassen:

- bool is_in(SDL_Rect const& rect1, SDL_Rect const& rect2) – Denna funktion returnerar true om parametrarna kolliderar med varandra.
- bool is_outside_of(SDL_Rect const& rect1, SDL_Rect const& rect2) – Denna funktion returnerar true om parametrarna ej rör vid varandra.

Player-klassen ärver av Entity-klassen så att den har tillgång till funktioner som är liknande mellan Player-klassen och Enemy-klassen. Player-klassen ärver följande funktioner av Entity-klassen:

- void render(SDL_Renderer* const& r, bool const& debug) – Denna funktion ritar ut spelaren. Om debug är aktiv ritas spelarens aura och hitbox även ut.
- void shoot(vector<Bullet*>& bullets, int const& shot_direction) – Denna funktion anropas när spelaren försöker skjuta sitt vapen. Skotten som avfyras läggs till i parametern ”bullets”. Parametern shot_direction anger åt vilket håll spelaren är vänd åt vid anropet, vilket bestämmer skottriktningen.
- SDL_Rect get_hitbox() - Denna funktion returnerar spelarens hitbox.

2.3. Beskrivning av konstruktor

Player-klassens konstruktor tar in ett x värde och ett y värde, tillsammans med en SDL_Renderer-pekare. X och Y-värdena är två positiva heltal som representerar koordinaterna som spelaren startar på.

SDL_Renderer-pekaren används av Entity-konstruktorn, vilket i sin tur använder det för att initiera

bilderna som representerar spelaren.

I konstruktorn sätts även spelarens rörelsehastighet (”movespeed”) till 3, och byter färg på spriten som den ärver av Entity till spelarens satta färger. Den sätter även spelarens skjutfördröjning till 0, så att spelaren kan börja skjuta så fort den kommer in i spelet.

2.4. Beskrivning av publika metoder

Player-klassen har följande publika metoder:

- void input(bool KEYS[], vector<Bullet*>& bullets, SDL_Rect playscreen) – Denna funktion läser in spelarens input med hjälp av parametern ”KEYS”, vilket lagrar en bool för varje ASCII-värde beroende på om tangenten var intrycket eller ej.

  T.ex. blir KEYS[97] ”true” om spelaren har tryckt på tangenten ”a”.

  Parametern ”bullets” används endast av ”shoot”-funktionen som Player-klassen ärver av Entity-klassen (se punkt 2.2).

  Paramtern ”playscreen” är enbart rutan som representerar spelplanen. Den används för att se till att spelaren inte går utanför spelplanen.

- void update(long long score, int armor, bool rapidfire, bool slowfire) – Denna funktion uppdaterar spelarens huvudfärg beroende på ”score”-parametern, och spelarens axelfärg beroende på paramtern ”armor”, vilket är hur många armor-uppgraderingar som spelaren har för tillfället.

  Den ser även till att spelarens sprite står i rätt position.

  Parametrarna ”rapidfire” och ”slowfire” används för att uppdatera spelarens skjutfördröjning, vilket bestämmer hur snabbt spelaren får skjuta.

- void render(SDL_Renderer* const& r, bool const& debug) – Denna funktion ritar ut spelarens sprite, och även spelarens hitbox och ”aura” om parametern ”debug” är ”true”.
- SDL_Rect get_aura() - Denna funktion returnerar spelarens ”aura”, vilket är ett område som fiender försöker att undvika.
- void set_weapon(int new_type) – Denna funktion ändrar spelarens vapen till den inmatade parametern (Vapnets typ representeras av en int).
- bool is_hit(vector<Bullet*>& bullets) – Returnerar true om en av kulorna i parametern ”bullets” kolliderar med spelarens hitbox. Om en kula kolliderar tas även kulan bort ur vektorn.
- void reset() - Denna funktion kallas när spelaren har dött, så att om du vill spela igen under samma session kommer allt vara som det var den första gången.

2.5. Beskrivning av medlemsvariabler

Player-klassen innehåller följande medlemsvariabler:

- int movespeed – Lagrar i pixlar per frame hur snabbt spelaren rör sig, med ett default-värde på 3.

- double shoot_timer – Spelaren får enbart skjuta när denna variabel är lika med eller mindre än 0. När spelaren har skjutit sätts variabeln till 2, och minskar med ett visst antal varje gång ”update”-funktionen kallas (se punkt 2.4).

Följande variabler ärvs av Entity-klassen:

- double x och double y – Dessa variabler representerar spelarens koordinater.
- int facing – Denna variabel representerar åt vilket håll spelaren står.
- Sprite sprite – Detta objekt representerar spelarens synliga form, och håller även reda på spelarens animation.
- Weapon weapon – Detta objekt representerar spelarens nuvarande vapen, vilket kallas på då spelaren ska skjuta (se punkt 2.2, ”shoot”).

3. Detaljbeskrivning av PlayState-klassen

3.1. Syfte

Syftet med PlayState-klassen är att samla det kontinuerliga spelets objekt på ett ställe där de på ett enkelt sätt kan anropa t.ex. input, update och render-funktionerna för de objekten.

Det tar in spelarens inmatningar och uppdaterar spelet därefter.

3.2. Arv

PlayState-klassen ärver av klassen GameState, vilket är en abstrakt klass som representerar alla ”states” i spelet.

Av GameState-klassen ärver PlayState-klassen följande abstrakta funktioner, vilka används för att kunna anropa alla olika ”states” grundfunktioner på ett samlat sätt:

- virtual void input(bool KEYS[], bool const& mousedown, SDL_Renderer* const& r) = 0
- virtual int update(long long& score, SDL_Renderer* const& r, bool cheats[]) = 0
- virtual void render(SDL_Renderer* const& r, bool const& debug) = 0
- virtual void reset(SDL_Renderer* const& r) = 0

Se punkt 3.4 för en detaljerad beskrivning av vad dessa funktioner gör i PlayState-klassen.

3.3. Beskrivning av konstruktor

PlayState-klassens konstruktor tar in en SDL_Renderer-pekare och en TTF_Font-pekare.

SDL_Renderer-pekaren används för att skapa Player-objektet, hud:et och golvbilden. Den används även för att initiera dörr-variablerna.

TTF_Font-pekaren används även den för att initiera hud:et.

Konstruktorn initerar även alla medlemsvariabler (se punkt 3.5).

3.4. Beskrivning av publika metoder

PlayState-klassen innehåller följande publika metoder:

- void input(bool KEYS[], bool const& mousedown, SDL_Renderer* const& r) – Denna funktion används för att läsa in spelarens inmatningar, och PlayState-klassen anropar Player-klassens input-funktion, vilken uppdaterar därefter (se punkt 2.4).
- int update(long long& score, SDL_Renderer* const& r, bool cheats[]) – Denna funktion anropar update-funktionen för alla objekt i PlayState-klassen. Den kollar även om det ska ”spawnas” nya fiender, och gör det i så fall. Den kontrollerar även om spelaren har dött, och hoppar till GameOverState om så är fallet.

  Funktionen hanterar även powerups och poängräkning.

  Update-funktionen returnerar det ”statet” som ska användas härnäst, t.ex. returnerar den en int som representerar PlayState om spelet ska fortsätta, och om spelaren dör returneras en int som representerar GameOverState.
- void render(SDL_Renderer* const& r, bool const& debug) – render-funktionen anropar alla objekts render-funktioner, så att de ritas ut på skärmen.

3.5. Beskrivning av medlemsvariabler

PlayState-klassen innehåller följande medlemsvariabler:

- double enemy_spawn_timer – Denna variabel minskar med 0.1 varje frame, och när den når 0 och antalet fiender på banan är mindre än 10 så ”spawnas” en fiende bakom en slumpmässig dörr. Efter att en fiende har ”spawnats” så sätts variabeln till 20, vilket motsvarar ungefär 2 sekunder.
- double karma_tick_timer – Denna variabel minskar även den med 0.1 varje frame, och när den når 0 så ökar spelarens poäng med 1 (vilket är dåligt, då spelarens mål är att få så lite poäng som möjligt), och variabeln sätts till 10.
- bool is_dead – Denna variabel blir sann om spelaren blir skjuten eller knivhuggen utan att ha några armor-uppgraderingar kvar. Om den är sann så ändras statet till GameOverState.
- Player player – player-variabeln representerar den användarstyrda karaktären. Den förklaras djupare under punkt 2.
- vector<Bullet*> bullets – bullets-vektorn representerar alla kulor som för nuvarande finns på spelplanen.
- vector<Enemy*> enemies – Denna vektor representerar alla fiender som för tillfället är levande.
- vector<Pickup*> pickups – pickups-vektorn representerar alla pickup-objekt som för tillfället ligger på spelplanen.
- vector<Door*> doors – doors-vektorn representerar de tre dörrar som finns på spelplanen.
- HeadUpDisplay hud – hud-variabeln representerar hud:et, dvs. information till spelaren angående poäng, powerups och vilket vapen som spelaren har.

- Image floor – floor-variabeln innehåller en bild som ritas ut i det område som spelaren kan röra sig i.

4. Diskussion

Vårat spel är state-baserat, vilket fungerar speciellt bra för 2D-spel. Vi har även brutit ner så mycket kod som möjligt i mindre klasser, vilket ökar förståelsen av spelets logik.

Som ni kan se på bilden nedan så har vi en hel del klasser med allmänna funktioner som de flesta andra klasser kan använda sig av, t.ex. CollisionBox, Image och Text. CollisionBox ärvs t.ex. av Player, Enemy, Pickup och Bullet. Alla dessa klasser har behov av kollisionshantering, vilket CollisionBox tillhandahåller med sina funktioner (via arv).

De många klasserna bidrar till en högre abstraktionsnivå, vilket skapar en bättre översikt på hela projektet, vilket således innebär att ändringar inom en klass inte påverkar de andra. Detta förbättrar även vidareutvecklingsförmågan inom detta projekt.

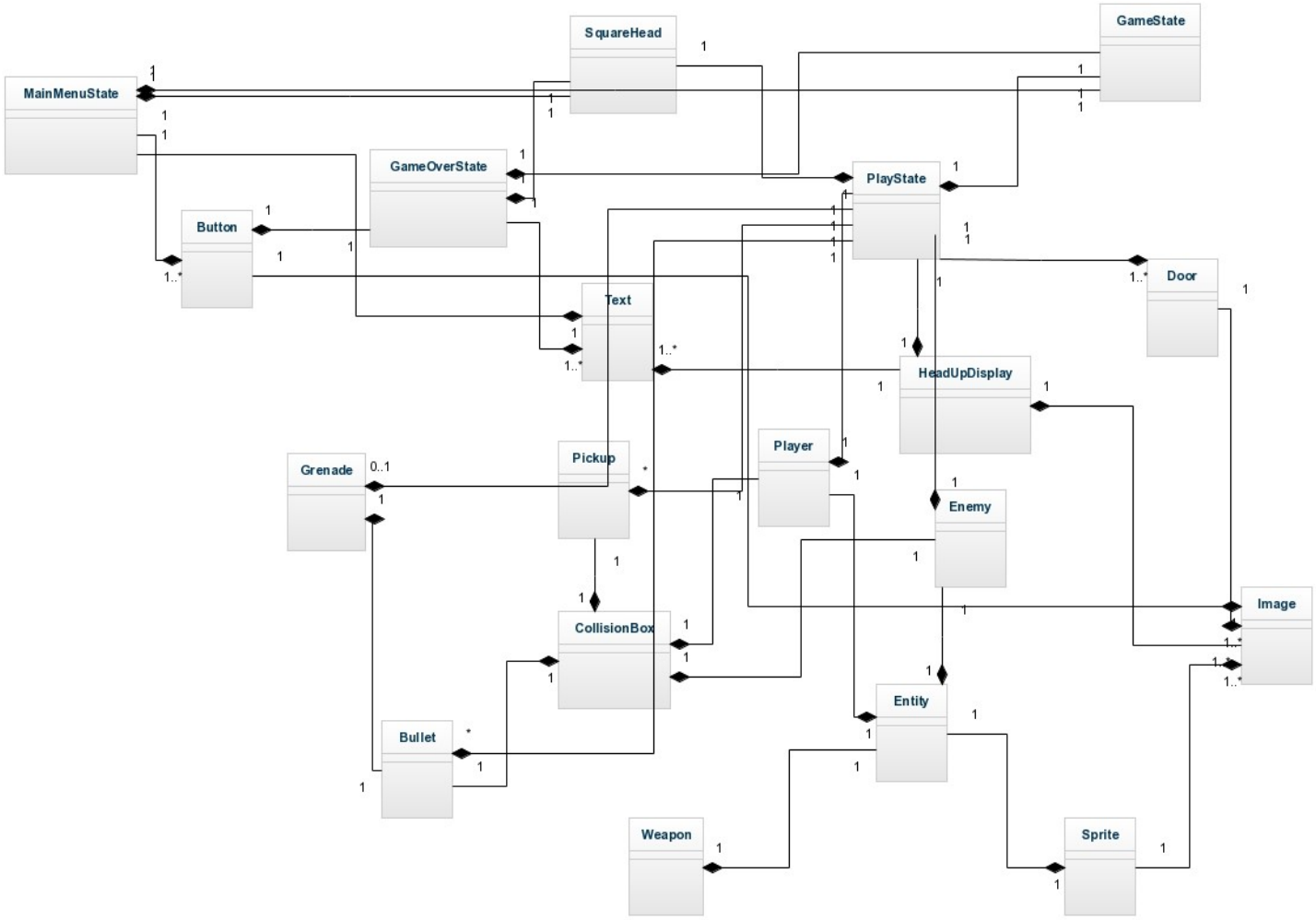
En nackdel med detta tillvägagångssätt är dock att mycket mer kod skrivs än vad som skulle ha behövts. Trots våra försök skulle även mängden coupling kunna minskas.

Antalet const& (konstanta referenser) i vår kod kan man nu i efterhand se blev en aning överdrivet, särskilt vid inparametrar till funktioner och klasser då till exempel en const& till en boolean eller integer tar upp ungefär lika mycket plats i minnet som en kopia av samma variabel.

Någonting vi skulle kunna gjort annorlunda är att skicka in en pekare till SquareHead-objektet som används till alla states istället för att skicka in flera olika variabler från objektet, som till exempel renderern, cheats och input-hanteraren. Vi skulle istället kunna ha get-funktioner för alla variabler vi behövde. Detta skulle ha kunnat öka kodens läsbarhet, och det skulle även göra det mycket enklare att lägga till nya input-variabler, vilket i nuläget skulle innebära att vi behövt gå in i GameState och dess underklasser för att ändra input-parametrar.

Vårat program saknar, på både gott och ont, en hierarkisk struktur. Detta skapar en viss svårighet att till exempel läsa och förstå spelets klasstruktur, vilket illustreras av diagrammet härnedan. Detta ger dock fördelar i form av att de individuella klasserna blir lättare att förstå. Ett sätt att förändra denna struktur skulle kunna vara att minska klassernas associationer med varandra genom att skicka dem som pekare från huvudklasserna istället för att importera klasserna direkt.

Någonting som skulle kunnat förenkla arbetet vore om vi hade kommenterat funktioner och klasser direkt när de skrevs istället för att göra detta i efterhand. På så sätt skulle kommentarerna reflektera ursprungliga syfte bättre, då koden låg färskt i minnet. Som det nu var blev det en hel del återläsande av koden, vilket tog upp för mycket extra tid. Dock hjälpte detta vid kodgranskningen, då vi nyligen hade läst igenom all kod.



5. Externa filformat

I vårt projekt använder vi endast tre olika sorters externa filer, nämligen bilder (.png), en textfil (.txt) som lagrar highscore-listan och en font av TTF-format.

Värdena i highscore-listan lagras i formatet ”[poäng]|[namn]”. Listan läses in ifrån filen varje gång spelaren kommer in i GameOverState, och skrivs sedan över med de nya värderna då spelaren går ifrån det statet.

Många av bilderna är endast 1x1 pixel stora (och expanderas sedan av SDL till rätt storlek). Detta sparar plats i datorns minne när spelet körs.

Fonten läses in två gånger under programmets gång (för de olika storlekarna som används).

En nackdel med detta upplägg är att vi har inläsningar från hårddisken flera gånger under spelets gång, vilket är något mindre effektivt än att ha bilderna lagrade i minnet.