Designspecifikation

TDP005 Projekt: Objektorienterade System

Elias Lund Lilja IP1 - elili552

Harald Vogel IP1 - harvo506

2. Beskrivning av klasser

2.1 Player

Syftet med player klassen är att ha en separat klass för spelaren där allt som rör spelaren kan hanteras. Den håller isär spelaren från resten av programmet vilket ökar läsbarheten och ger programmet en bra struktur. ett annat syfte med player klassen är att vi kan hantera spelarens egna kollisioner med andra objekt samt dess rörelser och hur spelaren påverkas på grund av yttre faktorer så som, skada.

Player är kopplat till klasserna arrow, sprite och level. Arrow gör så att spelaren kan skapa pil-objekt när användaren trycker på blanksteg och därmed avfyra en pil. Arrow använder i sin tur spelaren för att hämta spelarens position och kan därför anpassa sin startposition och rikting från spelaren. Från sprite får spelaren generella funktioner och variabler som gäller för alla sprites, saker som kollisionshantering för spelfönstrets begränsing, sprite sheets och positions- samt riktnings variabler m.m. Till level är spelaren kopplad för att spelaren skall kunna användas på en bana men det är level som skapar spelaren.

Spelarens konstruktor ser ut så här:

Player(Game game, Level* level, std::string playername);*

*Player(const Player&) = delete;**
*Player& operator=(const Player&) = delete;**

Den tar emot en spel-pekare, level pekare och ett playername. Spel pekaren gör så att spelaren kommer åt de funktioner som finns för att ladda in sin sprite sheet och vi kommer också åt fönsterstorleken. Level-pekaren används främst för att hämta några av de värden som skall användas till att initiera spelarens variabler och dessa beror på vilken nivå spelet är på. En lättare nivå kanske ger oss mer livspoäng medans en svårare kan ge oss lägre hastighet eller mer/mindre skada. Till sist kommer playername vilket inte används i nuläget men finns där ifall man någon gång kommer vilja skilja på flera spelare, t.ex. vid highscore eller om vi hade haft flera spelare samtidigt.

Anledningen till de sista två konstruktorerna är för att vi vill förbjuda kompilatorn att skapa egna kopierings konstruktorer för att kopiera en spelare, eller tilldela en spelare en annan. Vi har valt att bara ha en spelare, och det ska bara finnas en instans, och de får aldrig kopieras.

Spelarens funktioner:
move: Spelarens funktion för att förflytta spelaren beroende på vilken riktning den har.

takeDamage: En simpel funktion som subtraherar från spelarens livspoäng den skada som ges som funktionsargument.
events: Events kontrollerar tiden vid varje anrop och sätter den interna “cooldown” timern, vilket kommer begränsa hur ofta vi kan skjuta pilar. Funktionen har också ansvar för att kontrollera spelares livspoäng, sätta riktningen på de pilar som avfyras och sätta riktningen på spelare beroende på vilket håll vi vill gå åt. Events hanterar också på ett lämpligt sätt ändringen av spel “state” om vi har avlidit.

collision: Här sker kollisionshanteringen för spelaren och den kollar så att spelaren inte har krockat med väggen eller zombies. vid kollision bestäms också vilken riktning vi “studsar” bort från de objekt vi kolliderat med.

destruktor: Tar bort spelarens objekt när spelaren dör, funktionen ser också till att frigöra de resurser som spelaren skapat och förstöra eventuella pilar eller zombies som fortfarande existerar på spelytan då spelaren förstörs/dör.

Spelarens variabler:

Uint32 startTime, Uint32 arrowCooldownTime och Uint32 currentCooldownTime:
Dessa variabler är till för att begränsa hur ofta spelaren kan avfyra pilar. Vi kollar om currentCooldownTime (som kommer ifrån den nuvarande tiden - starttime) inte är mer än arrowCooldownTime när vi avfyrar en pil.

Uint8* keystates:
Keystates är en samling av keystates i SDL där vi kan lagra om de olika knapparna på tangentbordet är i nertryckt läge eller ej. Vi kan sedan undersöka keystates[knapp_namn].

list<Arrow*> arrows:
En lista som innehåller alla pilobjekt som spelaren har skapat.

SDL_Surface* arrowSheet:
pilarnas spritesheet

Game* game:
En game-pekare så vi kan komma åt game(för att ladda spritesheets eller hämta skärmbredd m.m)

Level* level:
En level-pekare för att komma åt level där vi kan hämta health, damage och speed för de olika nivåerna.

string facing:
En sträng där vi sparar vilken riktning vi har. Det är denna vi utgår ifrån när spelaren skall förflyttas.

string facing_for_arrow:
Liknar facing men denna är en riktning som vi skickar med pilarna så att de vet vilken riktning de skall åt.

2.2 Level

Precis som med spelaren så får programmet mycket bättre struktur med en separat level- klass. Den tar bort mycket funktioner och variabler som annars hade översvämmat game-klassen och det är även mycket praktiskt att kunna skapa nya nivåer samt ta bort dem med levels hjälp. Det är level som skapar spelare och user interface där vi ser spelares livspoäng grafiskt.

Level är en basklass där alla olika nivåer härstammar ifrån. Klassen level ansvarar för att ladda in bilder för nivåns bakgrund, zombie spritesheet, spelmusiken samt tidshantering för de svärmar av zombies som kommer fylla spelytan. Den tillhandahåller också en struct som används för att skapa svårighetsgraden.

Level fungerar på ett sätt då som en sorts mall där vi kan skapa olika nivåer med olika svårighetsgrader.

Level har relationer till allting som har en direkt koppling till spelplanen som t.ex. player, arrow, zombie och ui. Level har också en väldigt stark koppling till game eftersom de samarbetar tätt för att få fram spelet. När game står för själva spelmotorn är det level som får allt att synas på skärmen.

Levels konstruktor ser ut såhär:

Level(Game gamePointer);*

*Level(const Level&) = delete;**
*Level& operator=(const Level&) = delete;**

Den tar endast emot en game pekare vilket ger oss tillgång till games funktioner så som applyImage, playMusic, loadImage m.m. Detta ger oss det vi behöver för att rita ut ett spelobjekt på spelplanen, spela upp musik eller ladda in bilder, helt enkelt.

Anledningen till de sista två konstruktorerna är för att vi vill förbjuda kompilatorn att skapa egna kopierings konstruktorer för att kopiera en level, eller tilldela en level en annan level. Vi har valt att bara ha en level igång samtidigt och det ska bara finnas en instans. De får aldrig kopieras.

Levels funktioner:

virituel destruktor:
Level har en destruktor som frigör zombiespritesheet, spelbakgrunden och spelmusiken då en nivå förstörs.

destruktorn är dessutom virituel för att vi ska kunna gå till de klasser som ärver ur level och frigöra de resurser som de objekt allokerat innan vi kör levels destruktor.

blit:
Använder games applyImage för att rita ut bakgrund, spelare, zombies, ui och pilar.

events:
Events främsta funktion är att hantera skapande utav ett visst antal zombies i ett visst tidsintervall men den kommer också kontrollera antalet zombies som finns ute på spelplanen, räkna ut hur många som kommer att skapas enligt de tidsmönster vi har angivit. events har också som syfte att kolla spelarens egna events samt kolla om vi har döda zombies på spelplanen, om så är fallet tar vi bort dem.

update:
Ropar på de olika objektens move-funktioner.

collision_detection:
Ser till att objekten kollar sina kollisions-funktioner och hanterar kollisionen för pilarna.

Levels variabler:
struct Difficulty:
Denna struct innehåller alla viktiga variabler som kommer att förändras beroende på vilken nivå det är.
I structen finns:

```
unsigned int zombiesTotal,     //max antal zombies per nivå
unsigned int waveSize,                  //zombies som kommer per våg
unsigned int playerDamage,  //hur mycket skada spelaren gör
int playerHealth,         //hur mycket liv spelaren har
int playerSpeed,         //hur snabbt spelaren rör sig
unsigned int zombieDamage,//hur mycket skada zombies gör
unsigned int zombieHealth,   //hur mycket liv zombies har
unsigned int zombieSpeed;   //hur snabbt zombies rör sig
Uint32 waveCooldown;         //hur lång tid det är mellan vågorna
```

Game* game:
En game-pekare så vi kan komma åt game(för att ladda spritesheets eller hämta skärmbredd m.m)

Player* player:
En player-pekare så att vi kan uppdatera spelaren

Ui* ui:
En ui-pekaren som gör att vi kan uppdatera ui.

SDL_Surface* zombieSpriteSheet:
Sprite sheet för zombies.

list<Zombie*> zombies:
En lista som innehåller alla zombies.

SDL_Surface* levelBackground:
Spelbakgrunden.

Uint32 startTime:
En startid som hjälper oss när vi skall beräkna när nästa våg av zombies skall dyka upp.

Mix_Music* game_music;
spelets in-game musik

3. Tankar kring design

Vi har valt en design med en basklass Sprite som alla objekt som kräver en sprite ärver ifrån (spelare, zombies, ui, pilar). Vi har en klass, Game, som hanterar allt som har med spelmotorn att göra. Ifrån game.run som är den huvudsakliga metoden för att starta spelet ropar vi först på loadMenu. loadMenu skapar ett nytt meny objekt som ritar ut en startmeny och när man trycker på play så skapar vi en ny nivå (newlevel) genom att skapa ett objekt ärvt av Level. Level har hand om uppdateringen av dessa levelobjekt.

En fördel med vår lösning är att den var snabb att komma igång med. Det var inte allt för avancerad klass-struktur och det var i början lätt att överblicka varje område och se hur allt fungerade ihop. Detta gjorde det enkelt att koppla ihop en meny och att komma in i hur SDL fungerar och få ihop ett spelbart projekt tidigt i kursen.

Dock märkte vi ganska fort att många klasser (framförallt game) började bli allt större och det blev svårare och svårare att hitta i de större filerna. En annan nackdel är att vissa klasser fick ett allt suddigare användningsområde och det leder till att det är svårt att veta vart saker händer.

Ett exempel är när game hade hand om allt från ljud, musik, meny till hur zombies skulle dyka upp på nivån. Vi försökte lösa det genom att lägga till level-klassen och det fungerar för det mesta väldigt bra. vi har nu helt separerat spelmotorn med hur vi kan skapa nivåer.

Om vi hade gjort om spelet igen skulle vi nog ha valt en lite annorlunda struktur med olika fristående gamestates(menu, play, game over) som hanterar sig själva och inte att allt går via en punkt i mitten (game) vilket gör det rörigt och svårt att förstå. Vi har även förstått användningen med arv och haft fler olika basklasser som grenar ut sig till mindre klasser. Ett exempel är att dela upp sprite i en till klass, movable_sprites, där alla sprites som faktiskt skall röra sig kommer att ärva ifrån. Resten kommer ärva direkt från sprite och slipper då ha en massa onödiga funktioner som inte används.

4. Externa filformat

Vi använder oss av png filer till de texturer vi använder oss av. Vi har också valt att använda oss av true type fonts till de user interface vi skapat till spelet samt musik i mp3 format, wav filer för ljudeffekter. Allt de vi använder oss av externt kräver att vi länkar mot bilbioteken:

- SDL_image
- SDL_mixer
- SDL_ttf.