CPROG Rapport för Programmeringsprojektet

[Gruppnummer: 6]
[Gruppmedlemmar:
Emilia Ruth Sullivan 19970811-3528
Enes Ihsan Celik 19970127-5035
Johan Nordström 20031129-5075

Skriv en kortfattad instruktion för hur programmeringsprojektet skall byggas och testas, vilka krav som måste vara uppfyllda, sökvägar till resursfiler(bildfiler/ljudfiler/typsnitt mm), samt vad en spelare förväntas göra i spelet, hur figurernas rörelser kontrolleras, mm.
Om avsteg gjorts från kraven på Filstruktur, så måste också detta motiveras och beskrivas i rapporten.

Fyll i 'check-listan', så att du visar att du tagit hänsyn till respektive krav, skriv också en kort kommentar om på vilket sätt du/gruppen anser att kravet tillgodosetts, och/eller var i koden kravet uppfylls.

Den ifyllda Rapportmallen lämnas in tillsammans med Programmeringsprojektet. Spara rapporten som en PDF med namnet CPROG_RAPPORT_GRUPP_NR.pdf (där NR är gruppnumret).

1. Beskrivning

Vi skapade ett klassiskt arkadspel i 2D där spelaren tar kontroll över ett rymdskepp för att försvara sig mot fallande fiender. Spelet byggdes ovanpå en generisk spelmotor.

Målet med spelet är att spelaren ska försvara sig mot fallande fiender så länge som möjligt och samla på sig poäng. Spelarens figur(Rocketship) styrs i horisontellt i X-led med vänster och höger piltangent. Spelaren skjuter skott (Bullet) med upp piltangent. Fiender rör sig automatiskt nedåt i Y-led med en slumpmässigt hastighet och startposition. Om en fiende lyckas passera spelaren och når botten av skärmen utlöses en GameOver-händelse.

Vi har följt den rekommenderade filstrukturen med en tydlig separation mellan spelmotorn och den spelspecifika koden.

2. Instruktion för att bygga och testa

- För att bygga projektet krävs en C++ kompilator samt att biblioteket SDL3 är installerat på systemet. Koden kan kompileras med *make* kommandot i projektets rotmapp. Detta kommer att kompilera källkoden. När bygget är klart startas spelet med kommandot *./build/debug/play*
- För att spelet ska fungera korrekt måste biblioteket SDL3 vara installerade. Spelet hämtar sina visuella och textbaserade resurser från följande relativa sökvägar(se också Constants.h)
    - Bilder: resources/images (Innehåller bakgrund, rymdskeppet, kula och fiender)
    - Typsnitt: resources/font (Innehåller .tff fil för texter)

    - Konstanter: include/Constants.h (Samtliga filnamn och grundinställningar finns centraliserade i Constants.h.
- För testning verifieras att rymdskeppet kan styras med piltangenterna och att fiender spawnas korrekt. Testa även att GameOver-rutan dyker upp om en fiende når botten.
- Tryck på uppåt piltangenten för att skjuta fiender för att verifiera att en kula skapas och att fienden försvinner när de träffas. Se också till att poängräknaren(uppe till höger sida av skärmen) uppdateras vid varje träff.

3. Krav på den Generella Delen(Spelmotorn)

3.1. [Ja/Nej/Delvis] Programmet kodas i C++ och grafikbiblioteket SDL används.
Kommentar: Ja, programmet har kodas i C++ och grafikbiblioteket SDL har använts.

3.2. [Ja/Nej/Delvis] Objektorienterad programmering används, dvs. programmet är uppdelat i klasser och använder av oo-tekniker som inkapsling, arv och polymorfism.
Kommentar: Alla våra spelmotor klasser: Engine, Sprite, MoveableSprite och Label använder Objektorienterad programmering. Inkapsling, arv och polymorfism används. Värdesemantik är förbjuden.

3.3. [Ja/Nej/Delvis] Tillämpningsprogrammeraren skyddas mot att använda värdesemantik för objekt av polymorfa klasser.
Kommentar: Ja, koden är strukturerad så att värdesemantik undviks, istället används referenser och pekare för objekt av polymorfa klasser. Exempelvis så ärver EnemySpawner från demo::Sprite för att kunna implementera metoder som tick och draw istället för att objekt kopieras med värdesemantik.

3.4. [Ja/Nej/Delvis] Det finns en gemensam basklass för alla figurer(rörliga objekt), och denna basklass är förberedd för att vara en rotklass i en klasshierarki.
Kommentar: Ja, basklassen i vårt program är Sprite

3.5. [Ja/Nej/Delvis] Inkapsling: datamedlemmar är privata, om inte ange skäl.
Kommentar: Inkapsling är genomfört i princip alla klasser genom att datamedlemmarna är privata eller skyddade genom arv. Åtkomst till data medlemmar sker även via funktioner.

3.6. [Ja/Nej/Delvis] Det finns inte något minnesläckage, dvs. jag har testat och försökt se till att dynamiskt allokerat minne städas bort.
Kommentar: Minnesläckor har undviks genom att använda pekare så som std::shared_ptr för spelobjekt som är de dynamiskt allokerade objekten. Även exempelvis SDL texturer städas bort manuellt i labell.cpp.

3.7. [Ja/Nej/Delvis] Spelmotorn kan ta emot input (tangentbordshändelser, mushändelser) och reagera på dem enligt tillämpningsprogrammets önskemål, eller vidarebefordra dem till tillämpningens objekt.
Kommentar: Spelmotorn tar emot input via SDL och skickar vidare tangentbord händelser till objekt som MoveableSprites. Dessa anropas via onKeyUP, onKeyLeft och onKeyright.

3.8. [Ja/Nej/Delvis] Spelmotorn har stöd för kollisionsdetektering: dvs. det går att kolla om en Sprite har kolliderat med en annan Sprite.
Kommentar: Ja, spelmotorn har stöd för kollisionsdetektering om en Sprite sker med en annan och detta sker i spelmotorn genom att kontrollera om två MoveableSprites kolliderar. Då anropas respektive objekt kollisionshantering.

3.9. [Ja/Nej/Delvis] Programmet är kompilerbart och körbart på en dator under både Mac, Linux och MS Windows (alltså inga plattformspecifika konstruktioner) med SDL och SDL_ttf, SDL_image.
Kommentar:Ja, spelet och spelmotorn använder inga plattformsspecifika konstruktioner och vi har under utvecklingen testat programmet är kompilerbart och körbart både under Mac och MS Windows.

4. Krav på den Specifika Delen(Spelet som använder sig av Spelmotorn)

4.1. [Ja/Nej/Delvis] Spelet simulerar en värld som innehåller olika typer av visuella objekt. Objekten har olika beteenden och rör sig i världen och agerar på olika sätt när de möter andra objekt.
Kommentar: Ja, spelet simulerar en värld som innehåller olika typer av visuella objekt. Objekten har olika beteenden och rör sig i världen på olika sätt när de möter andra objekt. Spelet innehåller objekt som *Rocketship*, *FallingEnemy* och *Bullet*. De har unika beteenden(till exempel *FallingEnemy* faller, skott rör sig upp) och interagerar via kollisionshantering där objekt förstörs vid kontakt.

4.2. [Ja/Nej/Delvis] Det finns minst två olika typer av objekt, och det finns flera instanser av minst ett av dessa objekt.
Kommentar: Det finns flera olika klasser som *FallingEnemy* och *Bullet*. Genom klassen *EnemySpawner* skapas kontinuerligt flera instanser av *FallingEnemy* under spelets gång.

4.3. [Ja/Nej/Delvis] Figurerna kan röra sig över skärmen.
Kommentar: Alla objekt ärver från *MoveableSprite*. Spelaren styrs via tangentbordet medan fiender och kulor har automatiserad rörelse som uppdateras i varje frame via *tick*()-metoden.

4.4. [Ja/Nej/Delvis] Världen (spelplanen) är tillräckligt stor för att den som spelar skall uppleva att figurerna förflyttar sig i världen.
Kommentar: Ja, spelet använder en fönsterstorlek(definieras i *Constants*) som tillåter rörelse i både X och Y-led. Spelet tillåter rörelse i X-led eftersom spelaren kan trycka på höger och vänster piltangent för att ändra skeppets X-värde, vilket gör att Rocketship-figuren förflyttas horisontellt över skärmen. Spelet tillåter också rörelse i Y-led genom att fiender och Bullet(kulor) automatiskt uppdaterar sina Y-värden. Fiender faller nedåt och skott rör sig uppåt. Bakgrunden rullar dessutom(via Background-klassen) vilket förstärker känslan av att rymdskeppet färdas genom rymden.

4.5. [Ja/Nej/Delvis] En spelare kan styra en figur, med tangentbordet eller med musen.
Kommentar: Ja, Rocketship styrs av spelaren i sidled längs X-axeln med hjälp av vänster och höger piltangent. Engine fångar upp tangenttryckningar och uppdaterar skeppets position före varje rendering.

4.6. [Ja/Nej/Delvis] Det händer olika saker när objekten möter varandra, de påverkar varandra på något sätt.
Kommentar: Ja, genom *Engines* kollisionsloop upptäcks överlappningar. När en Bullet träffar en *FallingEnemy* flyttas fienden tillbaka till toppen(respawn). Dessutom interagerar objekten med spelplanens gränser, om en *FallingEnemy* når botten av skärmen triggas ett GameOver-tillstånd som avslutar spelet.