CPROG Rapport för Programmeringsprojektet

[Gruppnummer: 73 ]

[Gruppmedlemmar:

Alfred Berggren, 9710040693

Jesper Nilsson, 9709308978

Kristian Remnelius, 9512073298 ]

* Beskrivning

Github repository för projektet: <https://github.com/alfredberggren/cprog_projekt>

**Motorn**

Motorn består av tre huvudsakliga klasser: *GameEngine, AssetManager* och *Sprite.* Tanken är att *GameEngine* ska sköta den huvudsakliga spelloopen, och anropa *AssetManager* som innehåller medlemsfunktioner för hantering av alla dynamiska entiteter som laddats in i spelet.

*Sprite* är en abstrakt klass, och innehåller basmedlemmar som ett spel måste implementera för att kunna hanteras av spelmotorn. *AssetManager* behandlar klasser av typen *Sprite,* och exekverar funktioner på denna basklass som är dynamiskt bunden och därför kan överlagras i implementationer av klassen.

Motorn har möjligheten att i en *Map* ta in *SDL\_KeyCode*’s som nycklar med funktionspekare som värden, så att en utvecklare enkelt kan definiera beteenden på *Sprites* för nedtryckning av olika tangenter. Exempel på användning av denna beskrivs i detalj i kodens kommentarer. Boostfunktionen som beskrivs i spelbeskrivningen nedan, implementerades exempelvis med denna funktion, genom att skicka in *KeyCode* för blanksteg som nyckel och funktionen som ökar spelarens hastighet som värde i mappen som skickas in till spelmotorn. Motorn pollar events för knapptryck och utför funktionerna associerade med vissa knapptryck i mappen *key\_to\_function\_map.*

Andra *EventTypes* än tangentbordsnedtryck implementeras genom att hårdkoda virtuella funktioner i *Sprite* som exekveras när den *EventType* associerad med funktionen pollas. Ett exempel är *SDL\_MOUSEMOTION* som, när den pollats av motorn, utför funktionen ”*mouse\_moved”* i alla *Sprites.* Då är det upp till spelutvecklaren att implementera denna funktion för objekt som behöver hantera musrörelser, i vårt spel är denna klass *Player.* Det finns därför i nuläget två olika lösningar på hur events ska hanteras av motorn, den ena med funktionspekare och den andra med polymorfism. Vi är medvetna om att detta blir förvirrande för en användare av motorn, men i brist på tid får vi nöja oss med denna lösning.

Klassen *Camera* är också en del av motorn, och används som wrapperklass för en *SDL\_Rect* som representerar den del av spelet som objekt skall renderas i relation till. En utvecklare kan själv ange om kameran skall centreras runt något objekt eller om den skall vara stillastående. Detta kan utföras genom att i subklasser till *Sprite* överskugga metoderna *get\_rendered\_x(), get\_rendered\_y(), get\_rendered\_w()* samt *get\_rendered\_h()* för att själv beskriva hur objekt av klassen ska visas på skärmen. Detta är skilt från *Sprites* faktiska x, y, w och h- värden. När motorn renderar objekt kommer funktionerna ovan användas för att hämta objektens positioner på skärmen. Om inget specifikt beteende definieras för metoderna kommer alla objekt renderas med standarddefinieringar i *Sprite*, som returnerar objektens sanna positioner i spelet. I vårat spel väljer vi att rendera alla objekt relativt en kamera som hålls centrerad runt spelaren.

*LevelBackground* representerar spelets bakgrundsbild. Den är separerad från *Sprite* eftersom att den inte kan renderas på samma sätt som *Sprite.*

**Spelet**

Spelet går ut på att ”äta upp” andra karaktärer i spelet. Man kan bara äta upp en annan spelare om man är större än den. Man äter upp den andra spelaren genom att kollidera med den andra spelaren. Rörelse utförs genom att röra muspekaren, spelaren rör sig alltid mot muspekaren och kan stanna genom att sätta muspekaren i mitten av spelaren. Det finns också ”mat” utplacerad på kartan, maten kan inte röra sig och placeras ut på kartan slumpmässigt, spelare kan äta maten för att öka i storlek och för att ”ladda” sin boost som kan användas genom att trycka på blanksteg. För att boosta krävs det att man äter 10 st matbitar, när man sedan använt sin boost måste man äta 10 nya för att ladda upp den igen.

Klassen *Character* representerar alla typer av rörliga spelare. *Character* har en abstrakt metod *char\_move()* som förflyttar karaktärer i ett mönster som bestäms av subklasser till *Character.* Klassen *Player* representerar den spelbara karaktären, och rör sig alltid i en riktning mot muspekaren, om inte muspekaren är i mitten av spelartexturen. Klassen *NPC* rör sig i huvudsak mot det närmsta objekt som är mindre än de själva, men avviker från detta mönster ifall den närmsta entiteten är större än de själva, i vilket fall de försöker rymma. Fiendernas tillstånd kan ses från vilken textur som de vid en given tidpunkt renderar (när de flyr används en textur där de ser rädda ut o.s.v.)

Alla ”*GlobuleGobbleSprite”* (våra spelspecifika sprites)i spelet renderas relativt ett kameraobjekt. Kameran centreras alltid runt spelaren så länge denna är vid liv, när spelaren dör så börjar kameran följa den som karaktär som dödat spelaren som kameran senast följde.

* Instruktion för att bygga och testa

Spelet har byggts med den bifogade Makefilen. Kommandot ’make’ i katalogen där Makefilen finns kommer att bygga spelet till en exekverbar fil vid namn ”play”, som hamnar i katalogen /build/debug/.

* Krav på den Generella Delen(Spelmotorn)
* [ Ja/Nej/Delvis ] Programmet kodas i C++ och grafikbiblioteket SDL2 används.  
  Kommentar: Ja
* [ Ja/Nej/Delvis ] Objektorienterad programmering används, dvs. programmet är uppdelat i klasser och använder av oo-tekniker som inkapsling, arv och polymorfism.  
  Kommentar: Ja
* [ Ja/Nej/Delvis ] Tillämpningsprogrammeraren skyddas mot att använda värdesemantik för objekt av polymorfa klasser.  
  Kommentar: Ja
* [ 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
* [ Ja/Nej/Delvis ] Inkapsling: datamedlemmar är privata, om inte ange skäl.  
  Kommentar: Ja
* [ Ja/Nej/Delvis ] Det finns inte något minnesläckage, dvs. jag har testat och sett till att dynamiskt allokerat minne städas bort.  
  Kommentar: Ja
* [ Ja/Nej/Delvis ] Spelmotorn kan ta emot input (tangentbordshändelser, mushändelser) och reagera på dem enligt tillämpningsprogrammets önskemål, eller vidarebefordra dem till tillämpningens objekt.  
  Kommentar: Ja
* [ Ja/Nej/Delvis ] Spelmotorn har stöd för kollisionsdetektering: dvs. det går att kolla om en Sprite har kolliderat med en annan Sprite.  
  Kommentar: Ja
* [ Ja/Nej/Delvis ] Programmet är kompilerbart och körbart på en dator under både Mac, Linux och MS Windows (alltså inga plattformspecifika konstruktioner) med SDL 2 och SDL2\_ttf, SDL2\_image och SDL2\_mixer.  
  Kommentar: Delvis, vi har inte haft möjlighet att testköra spelet i Linux.

* Krav på den Specifika Delen(Spelet som använder sig av Spelmotorn)
* [ 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
* [ Ja/Nej/Delvis ] Det finns minst två olika typer av objekt, och det finns flera instanser av minst ett av dessa objekt.  
  Kommentar: Ja
* [ Ja/Nej/Delvis ] Figurerna kan röra sig över skärmen.   
  Kommentar: Ja
* [ Ja/Nej/Delvis ] Världen (spelplanen) är tillräckligt stor för att den som spelar skall uppleva att figurerna förflyttar sig i världen.  
  Kommentar: Ja
* [ Ja/Nej/Delvis ] En spelare kan styra en figur, med tangentbordet eller med musen.  
  Kommentar: Ja
* [ Ja/Nej/Delvis ] Det händer olika saker när objekten möter varandra, de påverkar varandra på något sätt.  
  Kommentar: Ja