Individuell inlämningsuppgift
Kurs: SP6
Period: vt 2012

David Rogers

Vår motor är baserad på en kärna kallad **GameCore** med flera singleton sub-system runt sig. Denna kärna har likheter med hur Gregory beskriver Ogre::Root [1]. Tanken är att ett spel ärver av GameCore som sedan sköter initialiseringen av alla sub-system när den körs. Enklast är helt enkelt att den klass som ärver av GameCore har main metoden i sig och kör sig själv där. Spelutvecklaren kan också använda denna klass till att göra det överliggande arbetet som kan tänkas behövas. Jag kan tänka mig att man t.ex. lägger in defaultbindings för input i denna klass (som sedan går att binda om). Tanken med att göra det såhär är att det skall vara enkelt att börja bygga ett spel. Man ärver av GameCore och kör run på den så har du redan grunden till ett spel. Sedan behöver man bara bygga en klass som ärver av GameState som är där logiken och resurserna som är unika för spelet ligger. GameStates är inte tänkt att vara en singleton utan det är mer som en del att av ett helt spel. Det är i viss mån som en *World Chunk*[2]. GameCore skrevs egentligen av båda gruppmedlemmarna medans GameState skrevs i huvudsak av min gruppmedlem (Niklas Skande) så jag refererar till hans rapport för mer ingående information.

Objekt i vår motor sköts via **GameObject**. Denna klass är abstract och är tänkt att fungera som grund för både dynamiska och statiska objekt så som Gregory beskriver dom[3]. Vi skiljer mellan dynamiska och icke-dynamiska objekt med en boolean i GameObject. Anledningen till att vi valde att inte dela upp dessa i flera klasser är att dom har såpass många likheter i vår motor att det inte skulle vara någon större mening.
GameObject är i sig själv indelad i tre olika klasser dock. En är såklart GameObject som står som referens till objektet i sig men det finns också Sprite som är den grafiska representationen av objektet som har en referens till en bild och också till en Animation, som är en klass vi skrivit själva. Vi valde också att lägga in ett värde för ett objekts gravitation i själva objektet. Detta värde är av en klass vi själva skrev kallad Vector2 som helt enkelt är en två-dimensionell vektor. Detta betyder att olika objekt kan påverkas olika mycket av gravitation i både X- och Y-led. Vi ansåg att detta medförde så mycket valfrihet som möjligt för spelutvecklaren medans vi samtidigt håller det väldigt simpelt.
Sprite har som sagt en referens till en bild och till en Animation som är en klass vi själva skrev för att ha hand om just animationer. Det är helt enkelt en sprite animation med flera bilder lagrade som Gregory beskriver i sektionen om *Cel Animation*[4]. Den tar en deltatid i varje update anrop i Sprites update metod och kollar mot tiden för att byta bild, som spelutvecklaren angivit, och sen byter om den uppnåtts. Vi valde att separera all grafik i en egen klass mest för att göra det så lätt och obegripligt som möjligt.

---

[1] Gregory, J. (2009). *Game Engine Architecture* (pp. 202-204).
[2] Gregory, J, pp. 693-694.
[3] Gregory, J, pp. 690-696.
[4] Gregory, J, p. 492.

Body är den tredje klassen som är tänkt att representera ett spelobjekt för fysikmotorn. Det är mer eller mindre vad Gregory beskriver när han skriver om *Collidable Entities*[5].Den har en referens till en rektangel som är den fysiska formen våra objekt har, detta är då en typ av två-dimensionell AABB som beskrivs i kursboken[6]. Vi valde att göra det så grundläggande dels för att det är enkelt men också för att det är mycket mindre beräkningskrävande än mer komplexa former. Det tillåter oss också att använda Javas egna rektangel klass som har metoder för att underlätta kollisionsdetektering.
Här har vi gjort flera större ändringar från vår första verision av vår spelmotor. I den första verisionen var spelobjekten nämna till Sprite och hade en referens till en Body. Förutom den osedvanligt korkade namnet så betydde det att ett objekt *måste* ha en grafisk representation vilket är något vi absolut inte ville ha. Vi valde ganska snart att göra en övergripande abstract klass för spelobjekt som sedan *kunde* ha grafisk och/eller fysisk representation. Anledningen till att vi gjorde så från början var mest dålig planering och att vi inte ändrade det i tid för första deadline var mest p.g.a. tidsbrist. Vi hade tyvärr vissa problem med buggar i andra sub-system som vi prioriterade att få rätt på. Varför vi gjorde ändringarna är i sig ganska uppenbara, vi ville helt enkelt inte ha en så rigid struktur på hur ett spel måste byggas utan ge mer valmöjligheter åt spelutvecklaren. Vi lade också till en funktion för att ”flippa” en bild. Denna hade helt enkelt glömts bort i första verisionen och består av två booleans i Sprite, en för att flippa bilden på den horisontella axeln och en på den vertikala axeln. Grafikmotorn gör sedan en koll om bilden skall flippas och lägger på den transform och förflyttning som behövs.

Fysikmotorn sköter i vår spelmotor faktiska förflyttningar samt detektering och hantering av kollisioner.Detta är också en klass vi båda i gruppen har kodat. Jag tänker referera till min gruppmedlems rapport för kollisionshanteringen som han skrev huvuddelen av.
Fysiken i vår spelmotor hanteras av ett sub-system som Gregory beskriver både som *Collision Detection System* och som *Rigid Body Dynamics*[7]. Vi har då alltså som Gregory föreslår slagit ihop dessa två system i en fysikmotor. Eftersom de är så tätt kopplade till varandra kändes det bara naturligt att göra så. Förutom själva förflyttningar och kollisioner sköter vår fysikmotor också appliceringen av gravitation. Som tidigare beskrivet finns värdet för gravitation i ett objekt så vad fysikmotorn gör är att helt enkelt gå igenom alla dynamiska objekt och applicera den angivna gravitationen. Detta medför att gravitationer alltid kommer att hända och om man inte vill att ett objekt ska bli manipulerad av gravitationen kan man helt enkelt sätta detta värde till 0. Det går också att göra mer ”onaturliga” saker som att ha negativ gravitation och gravitation i fel dimension.
I fysikmotorn har vi gjort några ändringar från vår första verision. Vi hittate en bugg som gjorde att objekt ”twitchade” (alltså rörde sig snabbt fram och tillbaka) vid vissa positioner. Detta rättades till. Vi la också till filtrering för kollisioner, en lista med vilka klasser ett objekt kan kollidera med. Som en typ av *Collision Masking* som kursboken tar upp[8].

Loadern är en enkel verision av en *resource manager* som kursboken [9] nämner. Den laddar in både ljud och bilder och skapar de typer som spelmotorn använder. Den kan också ha hand om en path till ljud- och/eller bildfiler. Detta underlättar när man gör själva spelet på så sätt att man slipper skriva ut

[5] Gregory, J, pp. 604-605.
[6] Gregory, J, (2009), *Game Engine Architecture* (pp. 609-610).
[7] Gregory, J, pp. 595-685.
[8] Gregory, J, p. 629.
[9] Gregory, J, pp. 272-301.

en hel path för varje fil och gör också koden mer lättläst. Det medför dock också att om man ska ladda in en fil som inte ligger i den förbestämda strukturen kan det bli lite rörigt and skriva en path. Det finns referenser till alla inladdade resurser i datastrukturer i Loadern för att inte behöva ladda in samma fil flera gånger. Detta är då för att spara på minne och minimera antalet diskåtkomster. Denna struktur skiljer sig en del från den redan nämna resource manager som kursboken gick igenom. De största skillnadena är att vår lösning är mycket mindre funktionell och saknar sätt att ladda in t.ex. composite resources. Vi ansåg att fördelen med dessa extra funktioner var ganska små i en så enkel 2D-motor att de inte vägde upp den tidsåtgång det tog att göra dem.
När Loadern laddar in ljudfiler så anropar den en metod i SoundManager som gör den faktiska inladdningen av filen och skapandet av ljudet i det format som används i motorn. Detta görs för att Loadern inte ska behöva ha hand om ljudformat utan det är något som SoundManager och SoundClip ska sköta.

Ljud sköts via vad som bäst beskrivs som en wrapper till Javas standardbibliotek javax.sound.sampled. Den förenklar användingen av ljuden dock och ett ljud existerar i motorn som ett SoundClip. Detta SoundClip har ett antal enkla metoder för att spela, stoppa och återuppta ljudet. Vi gjorde så för att använda javax.sound.sampled direkt är lite bökigt och vi ville förenkla användningen av ljud så mycket som möjligt.
Det finns några fler metoder för ökad funktionalitet. Dessa metoder är loop() och setVolume() och gör ungefär det dom är nämnda till. För att ändra ljudets volym tar setVolume() emot en double mellan 0.0 och 1.0 där 0.0 representerar helt tyst och 1.0 full volym. Uträkningen för decibel är tagen från ett exempel från en hemsida[10] mest eftersom ingen av oss i gruppen har någon större koll på ljudteknik. Funktionen för loop() sätter helt enkelt att ljudet ska loopa oändligt. Dessa extra funktioner är till för att införa lite extra val för spelutvecklaren, loop är ju mer eller mindre ett krav i spel och att ändra volymen bidrar till väldigt mycket lättare jobb för utvecklare. Funktionerna är helt enkelt till för att göra jobbet att bygga ett spel lite enklare.
Vi valde att inte ha med stöd för komprimerat ljud för att det skulle kräva att vi använda bibliotek som inte ingår i Javas standard och det är något som vi tidigt bestämde oss för att undvika så gott det går. MIDI-ljud har inte stöd för att vi visste att vi inte skulle använda det men mest p.g.a. tidsbrist.

Många klasser är i sig skrivna av oss båda i gruppen. De klasser som är angivna som skrivet av en av oss är i grunden skriven av den personen men har i största sannolikhet detaljer ändrade i sig av den andra gruppmedlemmen. Vi valde tidigt att inte hålla alltför hårt på detta eftersom vi dels bara är två i gruppen men också för att vi inte ville att någons arbete skulle stoppas p.g.a. en ändring som måste göras i en klass som inte är ens ”egen”. Så länge vi hade koll på vilka ändringar vi gjorde och kunde motivera de för varandra accepterade vi att vi gjorde ändringar där det behövdes. Vi har båda också mer eller mindre bugg testat alla klasser gemensamt.

[10] http://www.exampledepot.com/egs/javax.sound.sampled/Volume.html.