**Individuell inlämningsuppgift**

**Kurs: SP6**

**Period: vt 2012**

**Niklas Skande**

Vår motor är byggd på en kärna kallad GameCore som vi sedan har kopplat flera singleton sub-system runt. Detta i likhet med vad Gregory beskriver Ogre gör med Ogre::Root[[1]](#footnote-1). Tanken med det hela är att man enbart ska behöva ärva våran GameCore och sedan köra den med run() för att få alla sub-system att startas automatiskt. Klassen som ärver av GameCore är då den klass som också bestämmer vad för skärmupplösning som skall användas och vilka GameStates som skall finnas. Den kan även skapa KeyBindings m.m.

**GameStates** är något som krävs för att det hela skall köras dock. Ett GameState kan tänkas vara en specifik bana eller menyer till exempel. Denna klass finns för att förenkla spelutvecklarens jobb genom att automatiskt kunna byta GameState och ha hand om hantering av tömmning av minne tex. Dessutom ger GameState klassen tillgång till endel sub-system som kan vara till användning. Tillexempel kan man komma åt det Graphics2D object som kommer ritas ut nästa frame. På det Graphics2D object som tas emot kommer redan alla object vara utritade så det som ritas på Graphics2D objectet här kommer automatiskt hamna ovan på allt annat vilket är perfect för att skapa interface vilket det hela är tänkt för.

Den klass som ärver av GameState kommer få ett anrop via runLoop() varje frame så på detta sättet kan spelutvecklaren skapa spel logiken för spelet.

Det är även i den klassen dom olika spel objekten skapas och därför vi har gjort hantering för att tömma minnet på spelobjekt som inte längre skall användas tex.

Det är dock inte nödvändigt med fler än ett GameState men då får spelutvecklaren ta hand om hantering av olika funktioner som medföljer automatiskt om man har flera klasser som ärver av GameState, som tex paus av spelet när man går mellan menyer om menyerna är en separat GameState och spelet.

GameStates är alltså inte tänkta att vara singletons utan mer som en *World Chunk*[[2]](#footnote-2).

GameState klassen har fått några ändringar då vissa funktioner inte blev helt klara till första inlämningen.

**ScreenManager** är en klass som har hand om sjävla fönstret som spelet ritas ut på, och även hantering för fullscreen och återställning av skärmen till de inställningar som var innan man ändrade inställningarna på skärmen till de spelet använder.

För att köra i full screen krävs att skärmen man har kan klara av den upplösning, bit depth och refreshrate man vill ha. För att ta reda på om slärmen klarar dessa använde vi oss av en klass som kallas DisplayMode vilken finns med javas standard bibliotek som innehåller dessa. Det går sedan med hjälp av javas standard bibliotek att ta reda på vilka DisplayModes en skärm stöder och på så sätt kan ScreenManager kolla så skärmen klarar av de inställningar som frågas efter.

Det som ska ritas ut på fönstret ritar man sedan på ett Graphics2D object som man hämtar från ScreenManagern.

För att slippa flickering och tearing av object så använder ScreenManagern sig av javas BufferStrategy. Detta är något Gregory beskriver när han pratar om *The Frame Buffer*[[3]](#footnote-3).

Det går ut på att man ritar klart en bild i en buffer som man sedan låter skärmen rita ut. Medans skärmen ritar ut den bilden ritar man nästa frames bild i en ny bild i en buffer i bakgrunden. När skärmen har ritat klart bilden och bilden i bakgrunds buffern är klar så byter buffern på bilderna så den bild som tidigare var i bakgrunden är den som ska ritas på skärmen och den som var på skärmen tömms på grafik för att rita ut ny grafik på i bakgrunden medan skärmen ritar ut den nya bild den fick.

**GraphicsEngine** är en klass som är ganska intimt relaterad till ScreenManagern men kännde att jag ville skillja på dessa olika klasser. Resultatet är att det enda GraphicsEngine gör är att rita ut den bild dom GameObjects som ska ritas ut har på den position objecten har.

För att få fram den possition som objekten ska ritas ut på hämtar vi helt enkelt objektets position. Men för att kunna röra på skärmen så har vi gjort så man kan sätta ett objekt som skärmen ska följa. Detta gör att GraphicsEngine får räkna ut en ny position utifrån det objekt som har satts som focusObject där den sedan ritar ut bilderna för all objekt. Om man inte vill ha det objekt skärmen följer precis i mitten av skärmen kan man också sätta offset variabler i X och Y led.

Alla objekt har en variable som säger om den ska flippas horizontellet eller vertikalt vilket GraphicsEngine själv gör automatiskt utan att man behöver ladda in bilder för objekt i olika riktningar. Detta sker med hjälp av javas AffineTransform.

**InputManager** är den klass som har hand om HID delen av motorn. För att få möjligheten att mappa om key bindings så har vi gjort på det sätt som Gregory beskriver under avsnittet *Input Re-Mapping*[[4]](#footnote-4)*.*

Det InputManagern gör är att när man trycker på en knapp kollar den i en HashMap om det finns en entry med den knappen som key. HashMappen innehåller GameAction klasser som är den klass som hanterar olika states för om en knapp är tryckt eller inte bland annat.

Om det finns en entry med den knappen som tryckts som key sätter den det GameAction object som är value för den keyn till pressed.

InputManagern innehåller även funktioner för att ta bort alla key bindings, ta bort en specifik GameActions key bindings, ta bort en viss keys key binding och sätta key bindings. Eftersom denna klass är väldigt tätt relaterad till GameAction kommer jag prata lite om hur det hela funkar där också.

**GameAction** klassen är våran version av det Gregory pratar om när han pratar om att ge varje funktion ett unikt id under kapitlet *Input Re-Mapping*[[5]](#footnote-5).

GameActions är helt enkelt ett den klass spelutvecklaren kollar för att se om en specifik GameAction ska ske, så som att hoppa eller gå åt höger till exempel.

Så spelutvecklaren skapar ett GameAction med namnet Jump och binder det till space knappen genom att skicka Jump GameActionet och javas KeyEvent.VK\_SPACE till InputManagern. När spelaren sedan trycker på space kommer GameAction Jump att anropas via InputManagern och dess state kommer ändras till pressed. När spelet sedan kollar om spelaren ska hoppa kollar det om GameAction Jump är tryckt eller inte.

GameAction har också lite olika funktioner för hur det ska reagera på knapptrykningar. Man kan välja om GameAction ska reagera bara på ett knappnedtryck och inte på om knappen är nedtryckt eller om den ska reagera på att kanppen är nedtryckt för varje frame/loop.

**PhysicsEngine** klassen sköter hantering av förflyttning samt detektering och hantering av kollisioner.

Denna klass har både jag och Devid, min grupp kamrat skrivit på. Det jag skrivit är förflyttningen, kollisions detektering och korrektion av possition efter kollision så objekt inte hamnar i varandra.

Vi valde att göra fysikmotorn så enkel som möjligt för att kunna bygga på den efteråt om vi fick tid till det. Vilket vi inte gjorde. Detta gjorde att våran fysikmotor enbart hanterar kollision mellan AABB objekt (Axis-Aligned Bounding Boxes[[6]](#footnote-6)). Fysiken i vår spelmotor hanteras alltså av ett sub-system som Gregory beskriver både under *Collision Detection System* och under *Rigid Body Dynamics[[7]](#footnote-7)*.

Eftersom fysik och kollision är så tätt relaterade till varandra valde vi att hantera båda i våran fysikmotor.

Kollisioner kollas genom att kolla om två objekts bodies intersects eller inte. Om de gör det skapas en intersection yta som fysikmotorn sedan använder för att beräkna hur långt tillbaka objektet behöver flyttas. När det har beräknas kollar fysikmotorn varifrån kollisionen sker och flyttar sedan objektet enbart tillbaka i den riktningen.

Så om kollisionen sker ovanifrån kommer objektet enbart flyttas nedåt när possitionen rättas till för att inte hamna i det objekt den krockade i. Detta för att spara förflyttningen på den andra ledden ( i detta fall i sedleds ).

Förflyttningen av objekt sker genom en beräkning på objektens variablar force och gravity. Objektens force är för att hantera till exempel när spelaren vill gå åt höger så sätter han spelar objektets force till något positivt på X-axeln och fysikmotorn kommer sedan flytta spelaren åt höger. Att objekten själva har en variabel för gravitation gör också att man kan välja vad varje objekt ska ha för gravitation. Så ett objekt kan ha en gravitation som är åt höger ett åt vänster och ett nedåt till exempel utan att det skulle skapa några problem.

Objekt har även en typ av *Collision Masking* som Gregory tar upp i kursboken[[8]](#footnote-8). Detta för att vi ska kunna styra vilka objekt som skall krocka med vilka.

Information om vad som fixades och vad som har lagts till har min gruppkamrat skrivit om i sin rapport.

**GameObject, Sprite, Body**

GameObjects var vid första inlämningen en enda klass som hette Sprite som vi båda skrivit. Till denna inlämning har den förändrats och delats upp till flera klasser av David så han skriver om dessa.

1. Gregory, J. (2009). *Game Engine Architecture* (pp. 202-204). [↑](#footnote-ref-1)
2. Gregory, J, pp. 693-694. [↑](#footnote-ref-2)
3. Gregory, J. (2009). *Game Engine Architecture* (pp. 440-441). [↑](#footnote-ref-3)
4. Gregory, J, pp. 363-365 [↑](#footnote-ref-4)
5. Gregory, J, pp. 363 [↑](#footnote-ref-5)
6. Gregory, J. (2009). *Game Engine Architecture* (pp. 609). [↑](#footnote-ref-6)
7. Gregory, J, pp. 595-685. [↑](#footnote-ref-7)
8. Gregory, J, p. 629. [↑](#footnote-ref-8)