Gravity Surge

Nicky van Urk

(Realisatie)

Inhoudsopgave

Inleiding

In dit document beschrijf ik mijn aanpak hoe ik problemen in mijn code oplos oftewel ‘bugs’ herstellen. Ook beschrijf ik twee programmeer ideeën die ik heb geïmplementeerd met bijbehorende beschrijving.

Realisatie

Voordat ik begin met programmeren zorg ik er eerst voor dat ik een goed idee heb van wat er gemaakt moet worden. Dan zorg ik er voor dat ik alles in kaart breng door een ontwerp document te schrijven die ik gebruik tijden de gehele realisatie periode. Dit document is mijn referentie materiaal en zorgt er dat er precies gemaakt wordt wat er moet worden gemaakt.

Vervolgens gebruik ik dit ontwerp document om zo in grote lijnen mijn programma uit te stippelen en probeer ik effectieve manieren te verzinnen hoe ik iets zou kunnen realiseren om deze vervolgens te gaan implementeren. Hierbij gebruik ik een methode genaamd ‘bottom-up design’ dit houdt in dat ik eerst kleine programma’s / methodes schrijf en die later aan elkaar knoop om zo een groter geheel te vormen..

Elementen die ik heb leren maken zijn onder andere:

Beweging
Besturing
Levels laden
Gamestate Manager (zie pagina x)
Zwaartekracht (zie pagina x)
Enzovoort...

Vervolgens tijden het bouwen van de applicatie debug ik mijn programma elke keer als ik wat nieuws heb geschreven. Hierna kijk ik of alles werkt zoals ik het voor ogen heb en probeer ik het te ‘breken’ en als dit lukt herstel het in mijn code, oftewel een bug fix. Na het schrijven van nieuwe code debug ik mijn programma nogmaals en test ik het opnieuw tot dat er maar één optie overblijft en dat is de juiste manier van werken die ik als programmeur voor ogen heb. Dit doe ik eigenlijk de hele periode totdat de applicatie is voltooid.

Als de applicatie is voltooid test ik alles grondig, dit is eigenlijk zoveel mogelijk verschillende dingen uitproberen en als er iets vreemds gebeurt herstel ik dit alsnog. Ook kijk ik naar het geheugen wat wordt gebruikt tijdens het draaien van de applicatie, ik zorg ervoor dat ik geen memory leaks heb, oftewel geheugen van de computer die aan wordt gemaakt maar niet meer wordt vrijgegeven. Als ik als programmeur helemaal tevreden ben is de applicatie voltooid en wordt de bijbehorende documentatie verder uitgebreid.

Gamestaten

Concept

Gamestaten houdt in dat in het spel er altijd maar één staat actief is. De staten in mijn game bestaat onder andere uit:

Splash
Menu
Level select
Game
...

Als er van gamestaat wordt gewisseld wordt de vorige staat gewist of de staat blijft bestaan en wordt gepauzeerd. Een gamestaat die blijft bestaan kan ook weer worden geactiveerd en behoud al zijn gegeven, dit wordt gebruikt zodat de speler het spel zou kunnen pauzeren en weer verder zou kunnen spelen.

De meerwaarde van deze methode is dat je heel makkelijk nieuwe gamestaten kan aanmaken. Omdat er in de code maar weinig verschil bestaat tussen twee gamestaten of tien gamestaten.

Implementatie

De gamestaten worden geïmplementeerd door middel van twee classes genaamd:

State
StateManager

Op de volgende pagina volgt een uitgebreid voorbeeld van deze twee classes:

```
Class State
{
Public:
    State(stateManager, window, replace)

    Virtual pause()
    Virtual resume()

    Virtual update()
    Virtual draw()

    Unique_ptr<State> next()

    Bool isReplacing()

Protected:
    StateManager& m_stateManager
    RenderWindow& m_window

    Bool m_replacing

    Unique_ptr<State> m_next
}
```

In deze state class staan een paar functies en variabelen, de functies zijn over het algemeen '**virtual**'dat wil zeggen dat als ik deze class **inherit** met een class bijvoorbeeld genaamd 'MenuState' dat als er in de class 'MenuState' functies staan met de zelfde benaming als deze functies in de State class, de virtual functies in de State class worden overschreden met de functies uit de 'MenuState', het voordeel hiervan is dat voor elke state deze zelfde virtual functies zullen worden overschreden en daar kun je weer logica op uitvoeren.

De **next()** functie returns variabel **m_next,** een **unique_ptr** van type **State**.

De **isReplacing()** functie returns variabel **m_replacing**, een **true** of een **false**.

De overige variabelen zorgen dat er getekend kan worden naar het scherm door middel van de **draw()** functie en zorgt ervoor dat de **StateManager** kan worden aangeroepen in de class waar **State** wordt in **inherit** om zo een nieuwe state aan te roepen.

```
class StateManager
{
Public:
    StateManager()

    Run()

    NextState()
    LastState()

    Update()
    Draw()

    Running()
    Quit()

    Build(stateManager, window, replace)

Private:
    Stack<unique_ptr<State>> m_states

    Bool m_resume
    Bool m_running
}
```

De **StateManager** class zoals het woord al doet vermoeden zorgt ervoor dat alle staten zich makkelijk laten beheren.

De **StateManager()** functie en genaamd de **constructor** en initialiseert variabelen **m_resume** en **m_running** naar **false**.

De **run()** functie neemt een **unique_ptr**<**State**> parameter zet het variabel **m_running** naar **true** en ‘pushed’ de **unique_ptr**<**State**> in de **Stack**. Deze functie wordt gebruikt om de eerste gamestaat te initialiseren.

De **NextState()** functie zorgt voor de logica om naar een volgende bestaande staat te gaan en bepaald of de huidige staat word verwijderd of gepauzeerd door middel van de **isReplacing()** functie in de **State** class.

De **LastState()** functie gaat naar een bestaande staat dat op de positie staat in de Stack voor de huidige staat en verwijderd dan de huidige staat.

De **Update()** en **Draw()** functies roepen de gelijknamige functies aan van de staat dat op dat moment actief is.

En tot slot de **Build()** functie roept nieuw geheugen aan en in dat geheugen wordt een gamestaat opgeslagen en returns dan een **unique_ptr** naar deze locatie.

Vervolgens roep ik bij de start van het programma de eerste gamestaat aan. En roep ik bij elke update **NextState()** aan, deze bepaald dan wat er moet gebeuren: doe niks oftewel blijf in de huidige gamestaat of als er een nieuwe gamestaat wordt aangemaakt ga naar die nieuwe gamestaat en beslist dan of de huidige gamestaat wordt verwijderd of gepauzeerd.

Vervolgens in die eerste staat die wordt aangemaakt (denk aan een class: IntroState) creëer ik op een bepaald moment een nieuwe gamestaat (denk aan class: MenuState) en die wordt dan automatisch geactiveerd door middel van de beschrijving hierboven. Elke staat heeft zijn eigen variabelen, update() en draw() functies.

Zwaartekracht

Een beschrijving hoe ik de zwaartekracht in mijn game bereken.

Formule

De bijpassende formule waar mijn game op gebaseerd is is een formule van Newton second law of motion namelijk: F = MA waarbij A = F/M

Implementatie

F bereken tussen twee objecten.

Object1
Object2

Waar een object een planeet is of een ruimteschip oftewel de speler

F = massa * versnelling.

vx = object2.M * Afstand tussen object2 en object1 op de x-as.
vy = object2.M * Afstand tussen object2 en object1 op de y-as.

Waarbij vx en vy staat voor de velocity per as.
Deze velocity per as wordt elke update bij elkaar opgeteld.

Deze formule heb ik als volgt geïmplementeerd:

massa/ (d*d) * (object2 - object1) / d

D = Afstand tussen object1 en object 2 (denk aan lange zijde driehoek).
En Dx = Afstand tussen object1 en object 2 op de x as.
En als we zeggen dat D gelijk is aan 100 en Dx gelijk aan 75 is mijn formule als volgt:

20000 / (100*100) * (75) / 100 = 1.5

Zoals gezegd, deze uitkomst wordt elke update bij elkaar opgeteld...
Dus bij deze specifiek update wordt er 1.5 bij vx opgeteld.

Deze updates worden 60 keer per seconde uitgevoerd.

Dus als de positie niet zou veranderen (wat wel gebeurt maar om het simpel te houden)

Dan heb je 1.5 * 60 = 90 pixels per seconde op de x as dat zou betekenen dat het ruimteschip in deze situatie zich naar rechts verplaats op de x as.
Dit doe je ook voor de y as en de velocity wordt elke keer anders want bij elke update veranderd de afstand tussen de twee objecten dat wordt vergeleken. Hoe kleiner de afstand hoe groter de velocity wordt toegevoegd per update.

En hier mijn letterlijke code:

```
m_ship.vx += (float)(p.getMass() / (d*d) * (p.getGlobalPosition().x - m_ship.x) / d);
m_ship.vy += (float)(p.getMass() / (d*d) * (p.getGlobalPosition().y - m_ship.y) / d);
```

Conclusie

Tot slot de conclusie van de realisatie periode. De gehele periode is vrij soepel verlopen en heb ik veel dingen geleerd. Af en toe zat ik even vast met een probleem maar al snel kwam ik met een oplossing en dan wordt het kijken of ik met een betere oplossing kan komen, vervolgens de juiste oplossing implementeren. Ik ben tevreden over de gehele realisatie periode.