Technisch Verslag Thema opdracht Game

Net Attack

Stefan de Beer // 1684137
Paul Ettema // 1677247
Jeroen Kok // 1666479
Brian Duncker // 1677303
Bart van der Kolk // 1663533
Amrit Malhi // 1691806

21/2/17 // V0.2

Index

Inleiding

Dit spel is opgesteld als thema opdracht gedurende het tweede jaar van de studie Technische Informatica. De opdracht was om in teamverband een spel te creëren volgens de Scrum methode. Het project maakt gebruik van CMake en is geschreven in C++.

Het doel van het verslag is om de developers die gebruik willen maken van dit project, als basis voor hun eigen project, een leidraad te geven in hoe het project is opgebouwd. Met deze informatie moet het mogelijk zijn om verder te werken aan dit project.

Functioneel ontwerp

Features

- Vier speelbare karakters.
- Kan singleplayer en coop worden gespeeld.
- Een wereld waarin alle nodige karakters tegelijk in spelen.
- 2D Side Scrolling Platformer.
- Een aparte story voor elk karakter, die langzaam samenkomen tot één groot geheel.
- Puzzels oplossen om de levels te beïnvloeden om het level te behalen.

Speler Motivatie

Je bereikt het einde van het verhaal door verschillende uitdagende puzzels op te lossen.

Genre

De genre is een 2D platformer en puzzel game.

Mechanics

In het spel kun je soms pas ergens voorbij komen als dat deel veranderd is door de acties van een ander karakter met een speciale ability. Deuren en platforms kunnen worden veranderd door logische poorten en schakelingen op te lossen.

Unique Selling Points

- Wat een karakter verandert in de wereld, verandert ook voor de andere karakters.
- Multiplayer.
- Uitdagende minigames.
- Multiple rooms.
- 2D side on.
- 4 player co-op.
- Meerdere unieke karakters met speciale vaardigheden.
- Pixel graphics.

Doel Platform

Platform: PC met ondersteuning voor Windows en Linux.

Ontwerpdoelen

- De puzzels moeten uitdagend zijn.
- De veranderingen aan de wereld door de andere karakters moeten duidelijk merkbaar zijn.
- Een simpele physics engine voor de karakters.
- Meerdere werelden, die levels hebben, die op hun beurt rooms hebben die met elkaar verbonden zijn.

Technisch Ontwerp

De architectuur van het project berust op het idee dat alles in het spel een object is. Een object kan moveable, collisionable, controllable of drawable zijn. Het object kan ook meerdere van de opgesomde zijn. Om een voorbeeld te noemen; een character is moveable, collisionable en controllable. Dit betekent dat een character kan bewegen, kan botsen met andere objecten en bestuurd kan worden door de speler. Hierdoor is het gemakkelijk om een veelzijdig element toe te voegen.

Level

Een level is opgebouwd uit een set van bepaalde soorten objecten die uitgelezen worden uit een tekstbestand(.txt). In dit tekstbestand definieer je het soort object dat je wilt hebben gevolgd door de:
Locatie, grootte, maximum horizontale snelheid, maximum verticale snelheid, kleur/opaciteit en textuur van het object.
Het soort object bepaalt ook of het object een van de hierboven genoemde eigenschappen bevat. Er is hier gekozen om de gegevens direct achter elkaar te zetten omdat dat het makkelijkst te implementeren is en dit voor de developers is om test levels te kunnen maken dus hoeft het niet super mooi eruit te zien.

Main

In de main van het project komt alles bij elkaar. De main zorgt ervoor dat de levels worden uitgelezen uit de tekstbestanden en dan die op het scherm worden geschetst als de speler de keuze maakt in het main menu om het spel te starten. Verder zorgt de main ook voor wat overige dingen zoals het switchen tussen karakters, de achtergrond muziek, het pauze menu voor als je in game bent en de ending credits voor als je klaar bent met het spel.

Audio

De audio wordt in de main geregeld. Ten eerste werd er gekozen om een aparte audio engine klasse aan te maken maar door de manier waarop SFML audio laat werken is het niet mogelijk om je eigen subklasse te maken van de SFML audio klasse. Hierdoor wordt de audio direct in de main ingeladen en aangeroepen buiten de loop zodat het los van de staat van het spel door blijft draaien. Ook is er momenteel 1 audiobestand van ongeveer 5 minuten lang dat oneindig door loopt.

Menu

Het menu is opgebouwd met een generieke menu klasse die buttons bijhoudt en een functie heeft om deze makkelijk te kunnen centreren op het scherm. Dit is zo gemaakt zodat het makkelijk is nieuwe menu’s toe te voegen en dat deze dan direct gebruikt kunnen worden door muis en toetsenbord zonder dat het veel moeite kost te implementeren. De menu’s die aangemaakt worden hoeven dan alleen add button aan te roepen om nieuwe buttons toe te voegen en center buttons als deze in het midden moeten worden gecentreerd. Het tekenen en afhandelen van de muis en controller acties wordt dan door de menu superklasse allemaal geregeld.

Collision

De collision is opgebouwd uit een collision superklasse. Bij elke klasse die van deze superklasse erft, kan er collision gedetecteerd worden.

Deze klasse heeft 3 functies:

- (**detect_collision**) Een functie om te controleren of het object in een ander object is. Deze functie word gebruikt om te controleren of de collision detection toegepast moet worden.
- (**detect_collision_position**) Een functie om te kijken of het object in een bepaalde richting kan. Deze functie word gebruikt om te controleren aan welke kant de collision plaatsvindt. Deze functie kan op 1 of op een lijst van 'collisionables' gebruikt worden.
- (**detect_collision_detection**) Een functie die gebruikt wordt aan welke kant van het object de collision plaatsvindt. Deze functie verplaats het object in elke richting om zo te kijken of in die richting de collision plaatsvindt.

De functie ‘handle_collision’ wordt niet meer gebruikt.

Bijna alle functies van collisionables zijn uitgewerkt in physics. Dit is niet optimaal want hierdoor is elk 'physics object' ook een 'collisionable'. Wij hebben dit gedaan omdat 'collisionable' de position in physics moet kunnen gebruiken. Wij hebben hier nog geen oplossing voor zonder veel te moeten veranderen.

Het huidige collision systeem is een uitbreiding en verbetering van het oude systeem. De oude implementatie was erg gelimiteerd en had alleen de functie ‘detect_collision’. Hierdoor moest de ‘character’ klasse zelf controleren waar de collision plaats vond. Dit probleem hebben wij opgelost door de richting van de collision te implementeren in de ‘collisionable’ klasse. Hierdoor kan elk ‘collisionable’ object opvragen of er collision is en waar deze collision plaatsvindt. Ook kan er nu teruggegeven worden met wat voor object de collisie plaatsvindt door het soort object terug te geven. Dit kan gebruikt worden om bijvoorbeeld te kijken of een character een killbox raakt,of een wall.

Physics

In de huidige implementatie van de physics wordt er in de update loop gekeken naar alle objecten die physics hebben. Van elk wordt er gekeken naar hun snelheid, daarna wordt deze snelheid aangepast; genormaliseerd, gelimiteerd, ect. Daarna wordt er gekeken of heb object kan worden verplaatst, kan dit niet, dan wordt de snelheid aangepast zodat het object niet meer botst met een ander object.
Het voordeel van deze manier is dat er per update gekeken wordt welke bewegingen er toegepast moeten worden. Hierdoor heeft het bewegen een natuurlijk gevoel. Ook is het de bedoeling alles physics related een uniforme code te geven.

Hiervoor om te kunnen prototypen werd het bewegen van het personage afgehandeld als de toetsen werden ingedrukt, werd in de update loop alleen gekeken naar de zwaartekracht en waren de verticale en horizontale 2 verschillende variabelen.

Realisatie

Om aan de slag te gaan zijn de volgende packages vereist:

- SFML (2.4.1)
- CMake

Verder is het aangeraden om Git te gebruiken voor versiebeheer en een stabiele IDE naar keuze zodat je makkelijker code kan schrijven.

Een ontwerp cyclus gaat als volgt te werk. Na het toevoegen van een bepaalde functie of feature in een .cpp of .hpp bestand in de /src/ map sla je die op. Vervolgens navigeer je naar de build map in het project (als deze niet bestaat kan je een lege aanmaken). Vanuit de terminal run je dan het commando “cmake ..” en daarna “make”. Dit zorgt ervoor dat de makefile wordt aangeroepen en dat de code wordt gecompileerd. Om de code te runnen voer je het commando “./thema-game” uit.

Er is gekozen voor de Personal Computer als platform omdat dit het grootste platform is voor games. Hierbij zullen de bestuur systemen Windows en Linux ondersteund worden.

Elke keer als het level wordt gestart wordt er een zogeheten ‘factory’ aangeroepen. Deze leest een tekstbestand en op basis daarvan maakt hij een level.
Zo kunnen gebruikers ook hun eigen levels maken. Ook is er en levels.txt file waarin alle levels staan. Om een level toe te voegen moet de directory van het level in deze lijst van levels worden geplaatst.

De syntax van een rectangle in een level.txt is:
rectangle positie.x positie.y size.x size.y r g b texture(optioneel).

Van een circle
circle positie.x positie.y size r g b texture(optioneel)

Voor een wall:
wall positie.x positie.y size.x size.y r g b texture(optioneel)

Voor een killbox
killbox positie.x positie.y size.x size.y r g b texture(optioneel)

Van een character*:*
character postitie.x positie.y size.x size.y loopsnelheid spronghoogte r g b texture(optioneel)

Om de bewegingen en tekenen in het spel af te handelen gebruiken wij 2 loops;
De eerste is die alle key-input afhandeld. Er wordt gekeken welke relevante toetsen zijn ingedrukt en daarbij worden functies van character aangeroepen. Bijvoorbeeld als je op de linker-pijl drukt, wordt de snelheid van het character verhoogd.

Daarna wordt er door een lijst met alles wat kan bewegen geïtereerd. Alles wat kan bewegen wordt verplaatst - mocht een object stil staan dan wordt hij niet verplaatst. Hierin wordt de luchtweerstand, botsingen en zwaartekracht afgehandeld. Deze verplaatsing is nog niet te zien.
De vorige twee loops worden net zolang herhaald totdat een bepaalde tijd is verstreken. Als deze tijd is verstreken zal er worden geïtereerd door een lijst van Drawables. In deze loop wordt alles getekend wat uiteindelijk op het scherm komt te staan.
Door deze verdeling wordt de GPU niet overbelast - enkel 60 keer per seconden. Maar het spel voelt nog steeds responsive, omdat zolang 1/60 van een seconden niet is verstreken, alle input wel wordt afgevangen.

level_button en level_lever werken niet. Om deze werkend te maken moeten ze toegang hebben tot een level object om daar shapes te kunnen veranderen of toevoegen.

Evaluatie

Problemen

- We hadden van te voren op zijn minst een generiek klassendiagram moeten maken om overzicht te behouden.
- De mogelijkheid was niet opengelaten om buttons en levers te implementeren. Hierdoor is het niet mogelijk geweest om de buttons en levers goed te laten werken. Dit kwam mede door het gebrek aan een klassendiagram en duidelijke documentatie dat van te voren gemaakt had moeten worden om vast te stellen wat precies nodig was.

Oplossingen

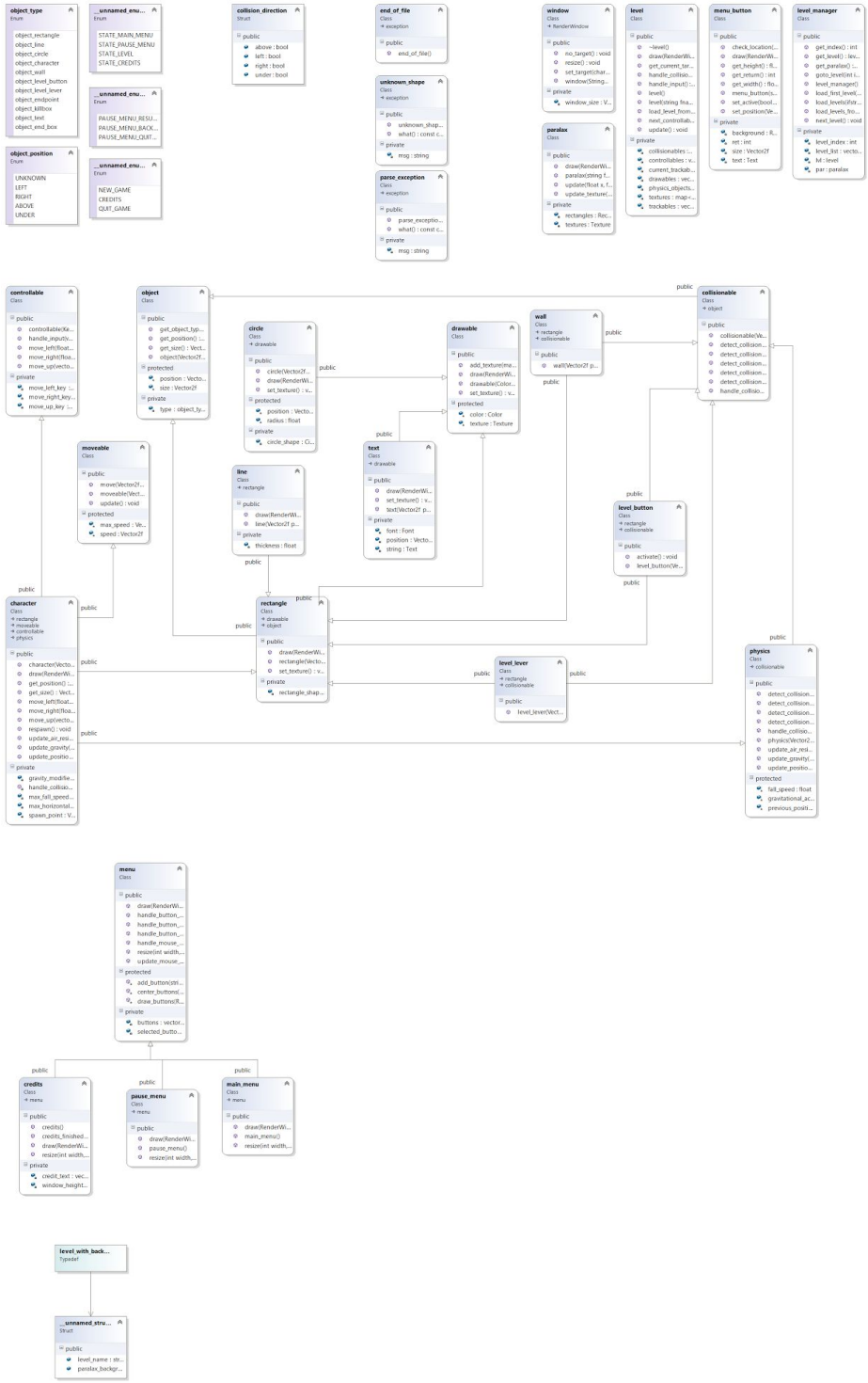
- Het missen van de klassendiagram had opgelost kunnen worden door voor of tijdens de eerste sprint een klassendiagram te creëren en deze actief aan en bij te houden.
- Voor de buttons en levers had er tijdens de tweede sprint reken gehouden moeten worden zodat deze zonder de hele code overhoop te halen te implementeren.

Conclusies en aanbevelingen

Wat zijn de conclusies en aanbevelingen op functioneel vlak?

- Het creëren van een klassendiagram voordat de werkweek begint en met voortschrijdend inzicht brainstormen.
- Betere communicatie over wat sommige functies moeten kunnen leveren om daar andere functies op toe te passen.

Bijlagen



KlasseDiagram: