UNIVERSITEIT GENT

FACULTEIT INGENIEURSWETENSCHAPPEN EN ARCHITECTUUR

Project Programmeren

Slime Volley in C++

Begeleiders

ir. Jeroen van der Hooft

ir. Pieter Bonte

ing. Merlijn Sebrechts

dr. ir. Femke De Backere

prof. dr. Bruno Volckaert

Verantwoordelijke lesgever

prof. dr. ir. Filip De Turck

Contact

pgm@lists.ugent.be

2de Bachelor Ingenieurswetenschappen:

Computerwetenschappen en Elektrotechniek

Academiejaar 2016-2017

Inhoudsopgave

1 Introductie

Hieronder wordt kort het doel van het project geschetst en wordt een inleiding gegeven over een aantal aspecten die typisch aan bod komen bij game programming. Neem deze bij voorkeur volledig door, zodat er voldoende achtergrondkennis is om aan de eigenlijke opgave te beginnen.

1.1 Doel van het project

Het doel van het project is om een variant op het spel Slime Volleyball te implementeren in C++, en dit in groepen van twee personen. Het project heeft als doel om zoveel mogelijk aspecten van softwareontwikkeling aan bod te laten komen. Hierbij denken we behalve het programmeren zelf aan het gebruik van de IDE, de integratie van bestaande libraries, het ontwerp van algoritmen en het efficiënt samenwerken aan een project. Besteed genoeg tijd en aandacht aan dit project, want het gaat om 3 van de 20 punten voor het vak Programmeren die gewonnen, maar dus ook verloren kunnen worden.

1.2 Het programmeren van games

Het programmeren van games verschilt op een aantal vlakken van het programmeren van traditionele applicaties. Een overzicht volgt hieronder.

1.2.1 De game loop

Vanuit het standpunt van de programmeur is de game loop de belangrijkste component van een spel. De game loop zorgt ervoor dat het spel vlot blijft lopen, onafhankelijk van het feit of de gebruiker al dan niet input genereert. Meer traditionele softwareprogramma's als tekstverwerkers of browsers reageren enkel op de input van de gebruiker. Spellen daarentegen blijven voortdurend werken, al dan niet met input van een gebruiker. Een game loop in pseudocode zou er als volgt kunnen uitzien:

Algorithm 1 Een voorbeeld van een game loop.

```
1: while user does not exit do
2:     handle user input
3:     run AI
4:     move objects
5:     resolve collisions
6:     draw graphics
7: end while
```

Hierbij wordt achtereenvolgens gecontroleerd welke input er gegeven is, wordt de logica van het spel en de artificiële intelligentie (AI) van de vijand uitgevoerd, wordt de beweging van verschillende objecten uitgevoerd, worden eventuele botsingen afgehandeld (cfr. Subsectie 1.2.3 en 3.5.5) en worden de objecten opnieuw getekend. De game loop kan verder verfijnd worden naarmate de ontwikkeling van het spel verloopt, maar de meeste spellen zijn op dit basisprincipe gebaseerd.
Game loops kunnen anders geïmplementeerd worden afhankelijk van het platform waarvoor ze ontwikkeld zijn. Een spel voor game consoles kan bijvoorbeeld alle processing resources gebruiken naar eigen wens, terwijl een spel

voor Windows uitgevoerd moet worden binnen de beperkingen van de process scheduler. Verder is het zo dat de meeste spellen multi-threaded zijn, zodat de berekening van de AI losgekoppeld kan worden van de generatie van bewegende beelden in het spel. Dit creëert een kleine overhead, maar zorgt ervoor dat het spel efficiënter uitgevoerd kan worden op hyper-threaded of multicore processoren. Nu de computerindustrie focust op CPU's met meerdere cores en threads, wordt dit steeds belangrijker.

1.2.2 Bewegende beelden

Het visuele aspect is bij games zeer belangrijk: er moet een vloeiend beeld weergegeven worden, wil men het spel er goed doen uitzien. De theorie rond hoeveel beelden per seconde het menselijk oog minimaal moet zien om iets als een vloeiende beweging waar te nemen, is veel ingewikkelder dan meestal wordt aangenomen. Er zijn een aantal vuistregels die in de meeste gevallen lijken te kloppen:

- Hoe groter het aantal frames per seconde (FPS), hoe vloeiender de beweging;
- Hoe kleiner het verschil tussen opeenvolgende frames, hoe vloeiender de beweging.

Het is gemakkelijk om de snelheid van de game loop ook in frames per seconde uit te drukken, zodat er een maat is om mee te vergelijken. Het aantal frames per seconde duidt dan eigenlijk aan hoeveel keer per seconde de toestand van het spel aangepast wordt. Het aantal gegenereerde FPS is enerzijds gelimiteerd door de hardware van de computer, maar anderzijds ook door hoe zwaar de game loop wordt. Wanneer een bepaalde actie bijvoorbeeld een groot aantal berekeningen vraagt, zal de iteratie van de game loop langer duren en zal het aantal zichtbare FPS tijdelijk dalen.

1.2.3 Collision detection

Bij games wordt dikwijls gesproken over "collision detection", het detecteren van een botsing tussen twee of meerdere objecten. In elke stap in de game loop moet er, voor het hertekenen van het nieuwe frame, gecontroleerd worden of er geen botsing veroorzaakt werd door het uitvoeren van de bewegingen van alle verschillende objecten. Dit kan bijvoorbeeld het tegen elkaar plaatsen zijn van de twee objecten in plaats van "in elkaar", of een botsing zijn die de objecten de tegenovergestelde richting opstuurt. Eens deze botsing gedetecteerd en opgelost is, kan het frame hertekend worden.
In een tweedimensionale ruimte valt detectie van botsingen goed mee, maar in driedimensionale ruimtes kan collision detection snel geavanceerd worden. Er zijn veel artikels en boeken geschreven rond dit onderwerp en elk van deze artikels en boeken vereist een goede wiskundige kennis. In dit project zal collision detection zich beperken tot de tweedimensionale variant.

1.3 Entity-component-system framework

Uit ervaring hebben we geleerd dat een klassieke objectgeoriënteerde aanpak niet altijd de beste resultaten oplevert bij het ontwerpen van een game. Abstracties die het ene moment logisch lijken, kunnen later een hindernis vormen bij het toevoegen van nieuwe functionaliteiten. Voorbeelden hiervan zijn:

- Tekenen naar het scherm gebeurt door elke klasse een render-methode te laten implementeren, maar dit zorgt ervoor dat dit aspect van het spel verspreid wordt over de volledige codebase, wat het moeilijker onderhoudbaar en uitbreidbaar maakt;

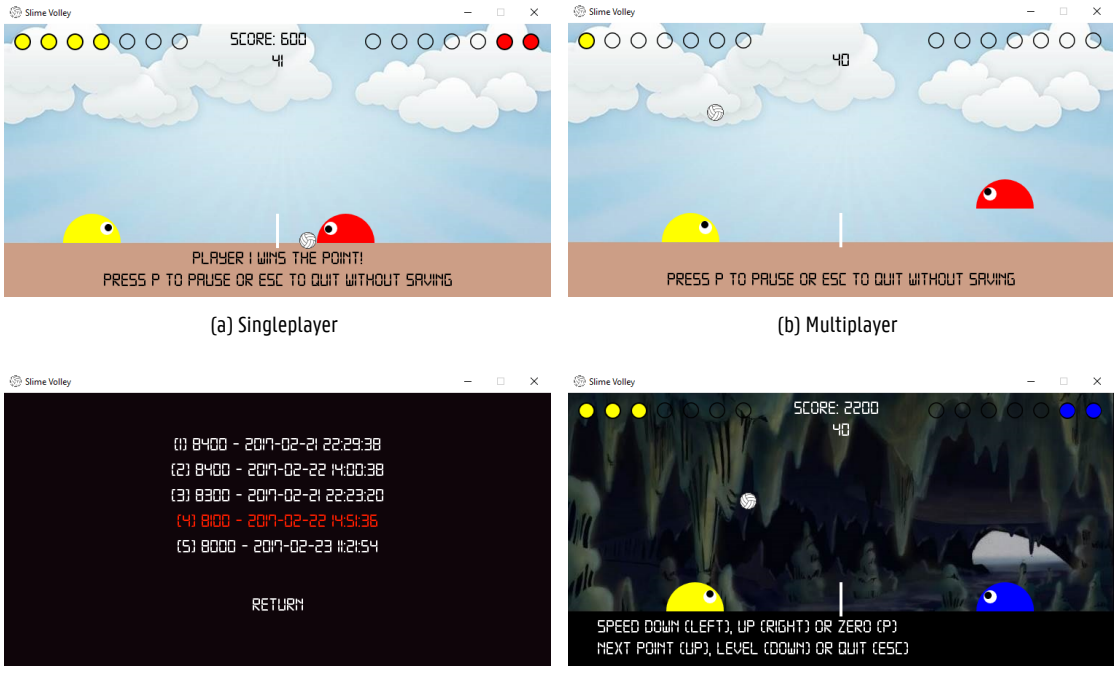
- Generieke functionaliteit wordt geïmplementeerd in klassen waarvan de objecten kunnen overerven, zodat een speler bijvoorbeeld overerft van de klasse InputController om te kunnen reageren op input. Maar wat als het gedrag at runtime moet veranderen op basis van de toestand, bijvoorbeeld wanneer de speler tijdelijk bediend dient te worden door de AI in een bepaalde cutscene?

Bovenstaande problematiek wordt ook erkend in de spelindustrie en een van de voorgestelde oplossingen is het gebruik van een entity-component-system framework. Bij deze data-driven aanpak worden de verschillende spelobjecten niet meer voorgesteld als afzonderlijke klassen, maar als generieke entiteiten die samengesteld worden aan de hand van een bepaald aantal componenten. Deze componenten bevatten heel specifieke data die kunnen gekoppeld worden aan een object (bijvoorbeeld de positie en snelheid van een speler). Entiteiten en hun bijhorende componenten zijn dus pure datacontainers en implementeren geen functionaliteit of gedrag. Dat is de verantwoordelijkheid van de verschillende systemen die zullen inwerken op de geregistreerde entiteiten. Voorbeelden hiervan zijn een systeem dat bewegingsvectoren toepast en een systeem dat entiteiten op het scherm kan tekenen. Voor meer details en het precieze gebruik van een entity-component-system framework, verwijzen we naar de artikels in http://www.richardlord.net/blog/what-is-an-entity-framework en http://www.richardlord.net/blog/why-use-an-entity-framework.

2 Opgave

2.1 Het spel

Het doel van het project is om een variant op het spel Slime Volleyball te implementeren. Zoals bij volleybal steeds het geval is, is het de bedoeling om punten te scoren door de bal te laten botsen op het veld van de tegenstander. De linkse speler begint steeds met op te slaan, en het spel is gedaan zodra een van de twee spelers zeven punten weet te scoren. Er wordt zowel een singleplayer als een multiplayer voorzien, zodat het spel zowel tegen de computer als tegen een echte tegenstander gespeeld kan worden. In de singleplayer zijn er drie mogelijke levels, waarbij de speler in levels 1, 2 en 3 respectievelijk 200, 400 en 600 punten krijgt voor elk gemaakt punt, en 100, 200 en 300 punten verliest voor elk punt tegen. Er wordt een lijst bijgehouden met daarin de vijf hoogst behaalde scores, en het is mogelijk om een van de scores aan te klikken en een volledige replay te zien te krijgen van het spel. Een aantal screenshots van onze geïmplementeerde versie zijn te zien in Figuur 1, en een video van het spelverloop werd als bijlage voorzien bij dit document.



(a) Singleplayer

(b) Multiplayer

(c) Highscores

(d) Replay

Figuur 1: Vier verschillende use cases van het spel.

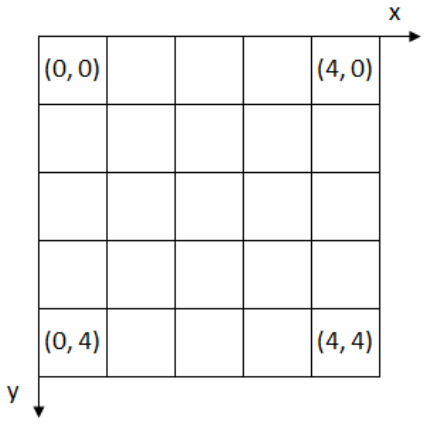
2.2 Allegro library

Om het visuele aspect van het spel af te handelen, zal er gebruik gemaakt worden van de Allegro library. Om deze library aan te roepen, wordt een AllegroLib klasse voorzien die de nodige funtionaliteiten abstraheert. Dit betreft alles in verband met initialisatie van Allegro, het opzetten van input, timing en events. Tracht zeker de code eens te bekijken, zodat het duidelijk is wat deze klasse precies aanbiedt. Meer informatie over de API en documentatie omtrent de Allegro library (5.0.10) is terug te vinden via http://liballeg.org/a5docs/5.0.10/ en https://www.allegro.cc/ (community-based).

3 Implementatie

3.1 Het coördinatenstelsel

Aangezien Slime Volley een eenvoudig tweedimensionaal spel is, kan elk object op het scherm voorgesteld worden door een stel van (x, y) coördinaten. Het gebruikte coördinatensysteem van Allegro is equivalent aan dat van Java Swing, waarbij de x-coördinaat toeneemt naar rechts en de y-coördinaat toeneemt naar beneden (zie Figuur 2).

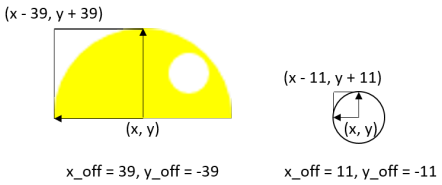


Figuur 2: Visualisatie van het Allegro assenstelsel.

Omdat er in het spel een zekere hoeveelheid wiskunde en fysica aan bod komt, is het niet onlogisch om over te stappen op een klassiek assenstelsel met een toename van de x- en y-coördinaat naar rechts en naar boven respectievelijk. Dit vraagt uiteindelijk slechts een kleine aanpassing bij het weergeven van de entiteiten: y afbeelden op $y_{max} - y$. Om de logica verbeterbaar te houden, wordt de volgende afspraak gemaakt: wanneer een speler zich op de grond bevindt (i.e. niet in de lucht springt), geldt er dat $y = 0$. De verschillende afmetingen worden volledigheidshalve weergegeven in Figuur 3. Merk op dat $y_{min} = 0$ enkel slaat op het "werkvenster" waarin de slimes en de bal zich bewegen: er is nog een "vloer" van 75 px voorzien waarin instructies of berichten geplaatst kunnen worden.

Figuur 3: Visualisatie van de afmetingen van het spel.

Bij het weergeven van afbeeldingen (bitmaps) in Allegro, worden de (x, y) coördinaten van de linkerbovenhoek gebruikt. Wanneer de coördinaten (0, 0) opgegeven worden, wordt de afbeelding in het Allegrostelsel in de linkerbovenhoek weergegeven. In ons klassieke assenstelsel, zou dit overeenkomen met de coördinaten (0, 299).
Voor de bewegende objecten in het spel (i.e. de bal en de slimes), is het echter makkelijker om te werken met de x- en y-coördinaten van het middelpunt van de (halve) cirkel. Op die manier hebben de bal en de gele slime in Figuur 3 dezelfde x-coördinaat, wat het maken van berekeningen vereenvoudigt. Gezien de grootte van de meegegeven afbeeldingen (cfr. Tabel 2), komt dit overeen met een translatie van (39, -39) en (11, -11) voor de slimes en de bal respectievelijk, zie Figuur 4. Om rekening te houden met deze translatie, kan er gebruik gemaakt worden van de functie `void Graphics::DrawBitmap(Sprite sprite, float dx, float dy, float cx, float cy)`, zie Subsectie 3.10.5. Merk op dat de onderkant van een slime op deze manier inderdaad overeenkomt met een y-waarde van 0, en dat er nooit een x-waarde lager dan 39 of hoger dan 710 kan voorkomen.



Figuur 4: Translatie van de (x, y) coördinaten van een slime en de bal.

3.2 De klasse Game

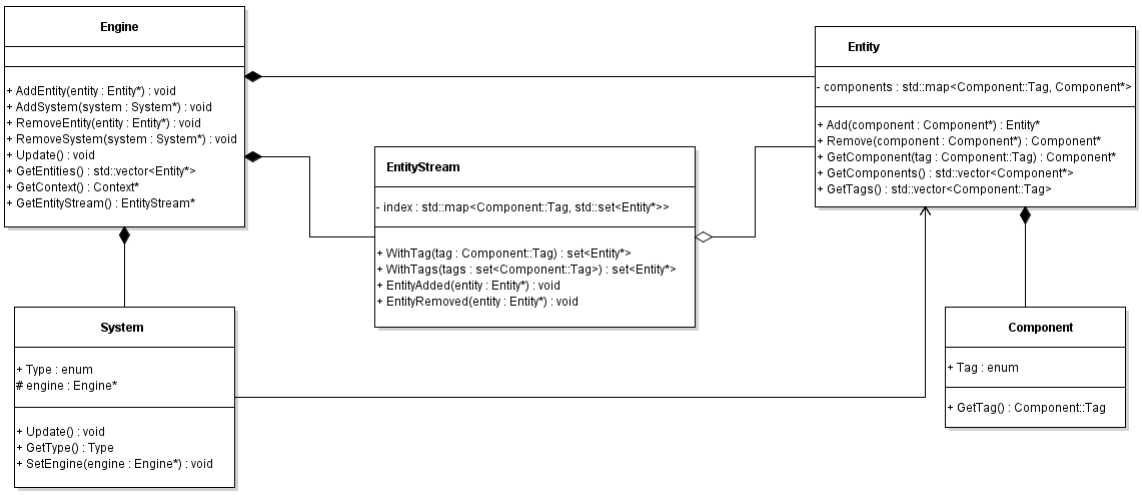
De implementatie van de klasse Game werd reeds voorzien. Deze initialiseert de Allegro Library en laadt vervolgens de verschillende afbeeldingen (ook wel sprites genoemd), lettertypes en highscores in vanuit de map assets. Eenmaal de functie Run() opgeroepen wordt, wordt er een eenvoudige game loop opgestart waarin het startmenu wordt weergegeven. Door middel van muisinput van de gebruiker, kunnen vier verschillende opties gekozen worden:

- Singleplayer: start een spel tegen de computer (klasse GameSingle)
- Multiplayer: start een spel met twee spelers (klasse GameMulti)
- Highscores: geef de vijf hoogst behaalde scores weer
- Quit: sluit het spel af

Wanneer de highscores gekozen worden, wordt een nieuw menu getoond met daarin de vijf hoogst behaalde scores, die elk aangeklikt kunnen worden om de replay van een spel te tonen (klasse GameReplay). Afhankelijk van de teruggegeven waarde van de Run()-functies van deze drie types spellen, zal een nieuw level of spel gestart worden, of wordt het startmenu opnieuw weergegeven. Elk van de klassen GameSingle, GameMulti en GameReplay zijn gebaseerd op een entity-component-system framework, waarover hieronder meer informatie volgt.

3.3 Entity-component-system framework

Meer informatie over een entity-component-system framework was te vinden in de eerder vermelde bronnen in Sectie 1.3. Voor dit project werd reeds een basisversie geïmplementeerd, die wordt weergegeven in Figuur 5. Het framework bestaat uit een aantal belangrijke klassen, die hieronder kort overlopen worden.



Figuur 5: Het te implementeren entity-component-system framework.

3.3.1 Component

Een Component is een heel eenvoudige data-container. De bedoeling is conceptuele eigenschappen in een klasse te steken die overerft van de klasse Component. Voor de eenvoud mogen de attributen in dit project publiek gehouden worden, aangezien deze dikwijls zullen moeten uitgelezen worden.

3.3.2 Entity

Een Entity stelt om het even welk object voor in het spel. Entiteiten worden opgevuld door een verzameling Components toe te voegen die de eigenschappen van het object voorstellen. De belangrijkste methoden van de klasse Entity zijn:

- `Entity* Add(Component* component)`: Voegt een Component toe via een pointer naar deze component. Een pointer naar de Entity wordt teruggegeven om via een builder pattern verder te kunnen werken (https://en.wikipedia.org/wiki/Builder_pattern), maar dit is geen verplichting;
- `Component* Remove(Component* component)`: Verwijdert een Component via een pointer naar de te verwijderen component. Een pointer naar de verwijderde Component wordt teruggegeven (verwijderen betekent niet vernietigen);
- `Component* GetComponent(Component::Tag tag)`: Geeft een pointer naar een Component terug door opgeven van zijn Tag (enum-waarde);
- `std::vector<Component*> GetComponents()`: Geeft een vector van Component pointers terug die wijzen naar alle Componenten van deze entiteit;
- `std::vector<Component::Tag> GetTags()`: Geeft een vector terug met alle Tags (enum-waarden) van de Componenten van deze entiteit.

3.3.3 System

Een System stelt een bepaald aspect van het spel voor, zoals beweging, collision detection en printen naar het scherm. Alle systemen zullen overerven van de hoofdklasse System, en zullen een of meerdere entiteiten verwerken in elke iteratie van de game loop. System beschikt over de volgende methoden:

- `virtual void Update()= 0`: Dit is een methode die verplicht te implementeren is door alle overervende System klassen, waarin alle logica van het systeem uitgevoerd wordt. Deze methode wordt eenmaal per game loop iteratie aangeroepen;
- `virtual Type GetType()= 0`: Dit is een methode die verplicht te implementeren is door alle overervende System klassen, die een unieke enum-waarde (Type) teruggeeft voor het desbetreffende systeem;
- `void SetEngine(Engine* _engine)`: Dit is een eenvoudige setter waarmee de protected variabele engine ingesteld wordt om een pointer naar de engine beschikbaar te hebben in alle overervende systemen.

3.3.4 Engine

De Engine is de belangrijkste klasse van het framework, waarin alle klassen functioneel samenkomen. Zowel Entities als Systems worden toegevoegd aan de engine. De Engine heeft een Update()-methode die zal opgeroepen worden voor elke iteratie van de game loop, waarin elk systeem wordt aangeroepen. Systems zullen via de EntityStream (zie Subsectie 3.3.5) de nodige entiteiten kunnen opvragen om deze te verwerken. Indien nodig kunnen systemen ook met andere systemen communiceren door ze op te vragen bij de Engine. De klasse Engine beschikt over de volgende methoden:

- `void AddEntity(Entity* entity)`: Voegt een Entity toe aan de Engine via een pointer;
- `void RemoveEntity(Entity* entity)`: Verwijdert een Entity van de Engine via een pointer;
- `void AddSystem(System* system)`: Voegt een System toe aan de Engine via een pointer;
- `void RemoveSystem(System* system)`: Verwijdert een System van de Engine via een pointer;
- `void Update()`: Roept om de beurt de Update()-methode aan op alle Systems die momenteel aan de Engine toegevoegd zijn;
- `std::vector<Entity*> GetEntities()`: Geeft een vector met pointers naar alle Entities uit de Engine terug;
- `Context* GetContext()`: Geeft een pointer naar de Context klasse terug, zie Subsectie 3.10.1);
- `EntityStream* GetEntityStream()`: Geeft een pointer terug naar het EntityStream object.

3.3.5 EntityStream

Aan de EntityStream kan een set van Entities opgevraagd worden door op te geven over welke Componenten ze moeten beschikken. Een RenderSystem zal alle Entities willen die beschikken over een Component van het type Sprite. EntityStream moet dus op de hoogte zijn van alle Entities die toegevoegd en/of verwijderd worden van de

Engine. Dit is mogelijk door in de klasse Engine bij toevoegen of verwijderen van een Entity de EntityAdded en EntityRemoved-methoden van EntityStream op te roepen. Deze zullen de interne mapping in een `std::map` van (sleutel) Component::Tag naar (waarde) `std::set<Entity*>` aanpassen. EntityStream beschikt over de volgende methoden:

- `std::set<Entity*> WithTag(Component::Tag tag)`: Geeft een set terug met pointers naar de Entities die toegevoegd zijn aan de Engine én die een component met de opgegeven tag bevatten.
- `std::set<Entity*> WithTags(std::set<Component::Tag> tags)`: Geeft een set terug met pointers naar de Entities die toegevoegd zijn aan de Engine en over alle opgegeven componenten beschikken.
- `void EntityAdded(Entity* entity)`: Wordt aangeroepen door de Engine bij toevoegen van een Entity. Een pointer naar de Entity wordt aangeleverd via het argument. Daarmee wordt de interne mapping aangepast, zodat de WithTag(s)-methoden correct kunnen gebruikt worden.
- `void EntityRemoved(Entity* entity)`: Wordt aangeroepen door de Engine bij verwijderen van een Entity. Een pointer naar de entiteit wordt aangeleverd via het argument. Daarmee wordt de interne mapping aangepast, zodat de WithTag(s)-methoden correct kunnen gebruikt worden.

3.4 Componenten

De verschillende componenten hangen af van de verschillende functionaliteiten van het spel en de manier waarop erop ingespeeld wordt. In de voorbeeldoplossing werden de componenten in Tabel 1 gebruikt. De component sprite wordt gebruikt door alle entiteiten die een afbeelding nodig hebben (zoals de slimes, de bal, en het net). De component motion wordt toegekend aan alle entiteiten die kunnen bewegen, volgens een bepaalde snelheid en versnelling (zie Subsectie 3.5.4). De component player slaat op de twee slimes, waarvan de ogen bewegen op basis van waar de bal zich bevindt (zie Subsectie 3.5.6. De component point wordt gebruikt door entiteiten die de tussenstand tussen de twee spelers aanduiden, met een maximum van zeven punten elk. De component ball wordt gebruikt als tag, i.e. om de overeenkomstige entiteit gemakkelijk op te vragen.
Bij wijze van voorbeeld zou het creëren van een entiteit van het type ball er als volgt uit kunnen zien:

```
Entity* ball = new Entity();
ball->Add(new ComponentSprite(Graphics::SPRITE_BALL, 150, 11, 738, -11, 150, 11, 288,
    11));
ball->Add(new ComponentMotion(0, 0, 0, -1));
ball->Add(new ComponentBall());
engine.AddEntity(ball);
```

De bal zal initieel weergegeven worden in het punt (150, 150), en stilhangen in de lucht. Doordat er echter een valversnelling is, zal de verticale snelheid toenemen naar onder (negatieve versnelling in het klassieke assenstelsel) en zal de bal naar beneden vallen.

3.5 Systemen

Ook de verschillende systemen hangen wederom af van de verschillende functionaliteiten van het spel. In dit project worden de onderstaande systemen onderscheiden, elk met een aantal constanten die in Tabel 2 gedefinieerd worden.

Component	Attributen	Verklaring
Sprite	sprite	Afbeelding van het object
	x	De x-waarde in het assenstelsel uit Figuur 3
	x_min	Minimale x-waarde
	x_max	Maximale x-waarde
	x_off	Offset x-waarde met betrekking tot Figuur 4 (i.e. 39 en 11)
	y	De y-waarde in het assenstelsel uit Figuur 3
	y_min	Minimale y-waarde
	y_max	Maximale y-waarde
	y_off	Offset y-waarde met betrekking tot Figuur 4 (i.e. -39 en -11)
Motion	v_x	Horizontale snelheid
	v_y	Verticale snelheid
	a_x	Horizontale versnelling
	a_y	Verticale versnelling
	j_y	Nieuwe verticale snelheid wanneer een object zijn minimale y-waarde bereikt
Player	player_id	ID van de speler (1 voor links, 2 voor rechts)
	pupil_x	De x-waarde van de pupil
	pupil_y	De y-waarde van de pupil
Point	player_id	ID van de speler waar het punt bij hoort
	point_id	ID van het gescoorde punt waar de component bij hoort
Ball	N/A	N/A (geen attributen nodig)

Tabel 1: Componenten van de voorbeeldoplossing.

Houd er rekening mee dat de volgorde van het toevoegen van systemen aan de Engine van belang is; zo zal het tekenen van het frame telkens de laatste stap van de iteratie zijn.

3.5.1 InputMulti

Dit systeem past de snelheid van de slimes aan naargelang de input van de gebruiker. Om respectievelijk naar links, rechts of omhoog te gaan, wordt de linkse slime bewogen met A, D en W voor Qwerty of Q, D en Z voor Azerty, de rechtse slime met de pijlen naar links, rechts en omhoog.

3.5.2 InputSingle

Idem als voor InputMulti, maar nu wordt enkel de linker slime bewogen met de pijlen naar links, rechts en omhoog.

3.5.3 AI

Dit systeem beweegt de rechtse slime op basis van een vooropgestelde logica. Omdat dit een belangrijke component is, wordt de logica in wat meer detail besproken in Subectie 3.9.

3.5.4 Motion

Dit systeem past de positie aan van alle entiteiten met de Component motion. In elke iteratie worden de horizontale en verticale snelheid (uitgedrukt in pixels per frame) aangepast volgens de gedefinieerde versnelling (uitgedrukt in pixels per frame^2), om vervolgens de nieuwe x- en y-waarde van de entiteit te berekenen.

3.5.5 Collision

Dit systeem controleert op mogelijke botsingen van entiteiten met elkaar of met de muur, de grond en het net. De volgende botsingen moeten opgelost worden:

- Een botsing tussen een speler en de muren of het net, met behulp van de variabelen x_min en x_max;
- Een botsing tussen een speler en de vloer, met behulp van de variabele y_min;
- Een botsing tussen de bal en de muren, met behulp van de variabelen x_min en x_max;
- Een botsing tussen de bal en het net;
- Een botsing tussen de bal en een speler, reeds geïmplementeerd wegens spelspecifiek (implementatiedetails hoeven niet bestudeerd te worden).

Merk op dat de positie en de afmetingen van het net beschreven staan in Tabel 2.

3.5.6 Eyes

Dit systeem plaatst de pupillen van een slime op de juiste plaats, gegeven de posities van de slime en de bal. Hiertoe kan de richtingscoëfficiënt (rico) berekend worden tussen het centrum van het oog en het centrum van de bal (deze eerste kan bepaald worden via een eenvoudige translatie van de (x, y)-coördinaten van de slime, cfr. Tabel 2), om vervolgens de coördinaten van de pupillen op de rechte te bepalen.

3.5.7 Output

In een singleplayer spel pusht dit systeem elk frame de nieuwe coördinaten van de twee slimes en de bal naar een lijst van Coordinates. Zodra het level afgelopen is, worden deze weggeschreven naar het bestand "./assets/high-scores/{start_time}_{level}.txt", met start_time het aantal seconden sinds epoch en level het huidige level (beide gedefinieerd in de hulpklasse Context, cfr. Subsectie 3.10.1. Wanneer het spel net begonnen is, zullen de eerste vijf lijnen er bijvoorbeeld als volgt uitzien (x speler 1, y speler 1, x speler 2, y speler 2, x bal, y bal):

```
150 0 594 0 150 133.125
150 0 588 0 150 132.375
150 0 582 0 150 131.250
150 0 576 0 150 129.750
150 0 570 0 150 127.875
```

De linker slime blijft staan (1-2), terwijl de rechter slime naar links beweegt (3-4) om een betere ontvangstpositie aan te nemen. De bal beweegt verticaal naar beneden (5-6), vooraleer te botsen met de linkse slime. Deze bestanden zullen ingelezen worden om een replay mogelijk te maken van de vijf spellen met de hoogste score. Merk op dat coördinaten enkel bijgehouden mogen worden wanneer ze verschillend zijn van de voorgaande, en dus niet wanneer het spel bijvoorbeeld gepauzeerd is.

3.5.8 StateMulti

Dit systeem behandelt de toestand van het multiplayer spel wanneer het spel actief (niet gepauzeerd) is. Wanneer een punt gescoord wordt, wordt een teller gebruikt om het spel tijdelijk te freezen. Op deze manier is het visueel duidelijk dat er een punt gescoord werd, en dat de spelers best hun toetsen kunnen loslaten. Zodra deze korte periode voorbij is, worden de spelers teruggezet op hun beginpositie en wordt de opslag toegekend aan de speler die het laatste punt won. Wanneer een van de spelers echter zeven punten gescoord heeft, moet er een melding verschijnen (geactiveerd door het Render-systeem, zie Subsectie 3.5.12) en moet er gewacht worden op input van de gebruiker: een spatie om opnieuw te beginnen, ESC om terug te gaan naar het startmenu.

3.5.9 StateSingle

Idem als voor StateMulti, maar nu voor het singleplayer spel. De mogelijke opties aan het einde van een spel hangen nu af van winst of verlies in het level (doorgaan naar het volgende level of opnieuw beginnen respectievelijk). Bovendien moet er voor elk punt voor of tegen de speler een update van de score gebeuren (+200 * level en -100 * level respectievelijk).

3.5.10 Replay

Dit systeem leest een bestand met coördinaten in en pusht deze naar een lijst van Coordinates. Afhankelijk van de afspeelsnelheid, die geregeld kan worden met de pijlen naar links en rechts, worden geen, een of meerdere frames uit de lijst verwijderd en worden de posities van de slimes en de bal aangepast. Net zoals in StateSingle moeten het aantal gescoorde punten en de score aangepast worden wanneer een punt gemaakt wordt. Met behulp van de pijl naar boven moet bovendien geskipt worden naar het volgende punt (i.e. vanaf de opslag), met behulp van de pijl naar onder naar het volgende level (als er een is). Deze laatste twee functionaliteiten kunnen zeer eenvoudig voorzien worden, zoals aangetoond in de reeds geïmplementeerde methoden GoToNextPoint() en GoToNextLevel(). Merk op dat de replay level per level gebeurt: zodra een GameReplay afgewerkt is, zal een nieuwe opgestart worden als er een volgend level bestaat.

3.5.11 Points

Dit systeem itereert over alle entiteiten van het type Point, en past de afbeelding aan van een lege naar een ingekleurde cirkel voor elk gescoord punt.

3.5.12 Render

Het systeem Render zal voor elk frame de weergave op het scherm voor zich nemen. Wanneer er getekend wordt, moet eerst en vooral het scherm leeggemaakt worden met `Graphics::Instance().ClearScreen();`, zodat op een leeg, zwart canvas kan begonnen worden met tekenen. Vervolgens worden alle entiteiten met component Sprite getekend, samen met de pupillen voor beide spelers. Ook de score en de bekomen waarde voor de frame rate worden naar het scherm geprint, samen met de instructies voor het pauzeren of afsluiten van het spel. Wanneer een punt gemaakt werd, wordt feedback gegeven naar de gebruiker (zoals “Nice point, Player 1!”) en wanneer een level of spel afgelopen is, worden de juiste instructies getoond (zoals “Press space to continue to next level or ESC to quit”). Als laatste stap wordt `Graphics::Instance().ExecuteDraws();` aangeroepen. Elke tekenoperatie wordt namelijk

naar een bitmap in het geheugen geschreven, en bij oproepen van ExecuteDraws() wordt de bitmap in het geheugen gewisseld met de bitmap op het scherm. Deze techniek, beter gekend als buffering, zorgt ervoor dat er tijdens het renderen geen flikkering in het scherm ontstaat.

3.6 De klasse GameMulti

Deze klasse voorziet een multiplayer spel. Bij initialisatie worden de benodigde systemen toegevoegd via de AddSystems()-methode en worden de nodige entiteiten aangemaakt in de MakeEntities()-methode. De systemen werden reeds voorzien in game_multi.h, de entiteiten zijn zelf nog toe te voegen.
In de game loop in de Run()-methode wordt er telkens een nieuwe iteratie gestart via de AllegroLib singleton. Hierna moet het huidige Allegro event opgevraagd worden, zodat de juiste actie ondernomen kan worden. Wanneer het een event betreft dat wijst op het indrukken van een toets op het toetsenbord, wordt de overeenkomstige boolean getoggled in de hulpklasse Context (zie Sectie 3.10). Wanneer de toets losgelaten wordt, wordt deze geuntoggled. Wanneer het event wijst op het verstrijken van de game state timer (het spel zelf moet geüpdatet worden), wordt een update uitgevoerd op de engine. Merk op dat er op dit moment nog een aantal zaken aangesproken worden via de Graphics klasse, om een statisch beeld weer te kunnen geven. Deze code moet uiteraard uitgevoerd worden door het Render systeem, zodat de Graphics klasse enkel van hieruit aangesproken zal worden.
Wanneer het spel gedaan is, worden de systemen verwijderd en worden alle aangemaakte entiteiten vernietigd met behulp van de RemoveSystems() en RemoveEntities()-methoden.

3.7 De klasse GameSingle

De implementatie van deze klasse is vrij gelijkaardig aan die van de klasse GameMulti. Er zullen een aantal extra systemen toegevoegd moeten worden, en wat de entiteiten betreft zal de sprite van de tegenstander afhangen van het huidige level. De belangrijkste wijziging is dat de AI ingevoegd moet worden, die besproken wordt in Sectie 3.9. Merk op dat de Run()-methode nu een geheel getal teruggeeft dat afhankelijk is van het verloop van het spel en de beslissing van de speler:

- De speler won het level en wil naar het volgende level: 1
- De speler won het level en wil of kan (einde spel) niet naar het volgende level: 2
- De speler verloor het level en wil niet opnieuw beginnen: 2
- De speler verloor het level en wil opnieuw beginnen: 3

De teruggegeven waarde wordt gebruikt door de klasse Game, om te beslissen wat er na het beëindigen van het singleplayer spel moet gebeuren.

3.8 De klasse GameReplay

De implementatie van deze klasse is gelijkaardig aan de twee bovenstaande. Tracht zelf te achterhalen wat er verwacht wordt in welke situatie.

3.9 De AI

De AI is de logische component die de tegenstander zal aansturen bij het spelen van een singleplayer spel. In de klasse SystemAI wordt in pseudocode uitgelegd wat er verwacht wordt van de tegenstander in levels 1 en 2. Een derde level kan als uitbreiding geïmplementeerd worden, maar dit is geen verplichting. Hieronder wordt kort de flow omschreven van wat er gebeurt in levels 1 en 2, al zal veel pas duidelijker worden bij het doornemen van de pseudocode.

3.9.1 Level 1

De logica is opgesplitst in vier delen: de opslag, de positionering, het springen en de return. In level 1 is er maar één enkele opslag, waarbij er in een enkele stap naar rechts gegaan en gesprongen wordt. Wanneer de bal in het spel is en in het linkervak lijkt te vallen, tracht te slime een basispositie aan te nemen. Wanneer de bal op het eigen veld lijkt te vallen, zijn er een aantal scenario's waarin de slime zal springen (bijvoorbeeld wanneer de bal van uiterst rechts over het net gespeeld moet worden, en er dus extra snelheid nodig is). Daarnaast kan de slime ook naar links of rechts bewegen om de bal beter over te kunnen spelen, wat in bepaalde gevallen nodig zal zijn. Om gefundeerde beslissingen te nemen, zal de functie `XBallBelow(double y_target)` gebruikt worden om te bepalen op welke positie de bal zich zal bevinden wanneer zijn verticale positie voor het eerst lager wordt dan y_target. Op basis van deze positie bepaalt de slime of hij een bepaalde beweging moet inzetten naar de bal. Merk op dat er een variabele state gebruikt wordt om een opslag te geven. Deze variabele houdt bij waar in het punt de slime zich bevindt, bijvoorbeeld om te weten wanneer hij bepaalde acties moet doen die tot een geslaagde opslag leiden.

3.9.2 Level 2

In level 2 worden er twee welbepaalde nieuwe opslagen ingevoerd en kiest de slime een betere basispositie uit om aces kort achter het net te vermijden. Verder is de implementatie vrij gelijkaardig aan level 1.

3.9.3 Level 3

Probeer, als uitbreiding, een moeilijker level te maken. Denk hierbij aan nieuwe opslagen, een betere basispositie, een betere definitie van wanneer een bal in het eigen veld botst en nauwer gedefinieerde gevallen waarin er gesprongen mag worden.

3.10 Hulpklassen

3.10.1 Context

Dit is een hulpklasse waar een aantal variabelen in staan die aangepast en opgevraagd kunnen worden. Het bijhouden van de overkoepelende game state is daar slechts één voorbeeld van. Merk op dat, aangezien Context meegegeven wordt aan een Game en vervolgens aan een Engine, de variabele context via de engine opgevraagd kan worden in elk systeem.

3.10.2 Color

Dit is een hulpklasse die toelaat een kleur voor te stellen, ofwel als RGB met waarden van 0-255, ofwel als RGBA met waarden van 0-1.

3.10.3 Tags

Dit is een hulpklasse die toelaat om aan de hand van een builder pattern een set van Compontent::Tag tags op te stellen. Dit kan handig zijn bij het opvragen van Entities aan de EntityStream klasse, door bijvoorbeeld het commando `std::set<Component::Tags> tags = (Component::Sprite).And(Component::Motion).List();` te gebruiken om alle entiteiten te verkrijgen die bewegen.

3.10.4 AllegroLib

AllegroLib is een wrapper rond de Allegro bibliotheek. Allegro laat toe om op relatief eenvoudige wijze naar het scherm te tekenen en een game te ontwikkelen. Via AllegroLib werd de library nog sterk vereenvoudigd, en het is dan ook de bedoeling dat er gebruikt gemaakt wordt van deze wrapper waar mogelijk. Allegro-specifieke methoden, die typisch beginnen met "al_" moeten dus meestal niet gebruikt worden. De volgende methoden werden voorzien:

- `void Init(int _screen_width, int _screen_height, float _fps)`: Initialiseert de Allegro bibliotheek;
- `void StartLoop()`: Deze methode start de achterliggende timer die 40 keer per seconde de game loop laat uitvoeren. Ze wordt dus opgeroepen net voor het starten van de game loop;
- `void StartIteration()`: Deze methode wacht totdat het volgende Allegro event ontvangen wordt. Ze wordt als eerste opgeroepen in een iteratie van de game loop;
- `void Destroy()`: Ruimt alle interne structuren van Allegro op, en wordt als laatste methode opgeroepen in het programma;
- `ALLEGRO\_EVENT GetCurrentEvent()`: Staat toe om het laatste Allegro event op te vragen om het te kunnen verwerken.
- `bool IsWindowClosed()`: Geeft aan dat het laatste opgevangen Allegro event al dan niet een window close event was en dat dus het scherm werd gesloten door middel van het kruisje rechts boven;
- `bool IsTimerEvent()`: Geeft aan dat het laatste opgevangen Allegro event al dan niet een timer event was;
- `bool IsKeyboardDownEvent()`: Geeft aan dat het laatste opgevangen Allegro event al dan niet een keyboard event was waarbij de toets ingedrukt werd;
- `bool IsKeyboardUpEvent()`: Geeft aan dat het laatste opgevangen Allegro event al dan niet een keyboard event was waarbij de toets losgelaten werd;
- `bool IsMouseMoveEvent()`: Geeft aan dat het laatste opgevangen Allegro event al dan niet een muisgerelateerd event was waarbij de muis bewogen werd;

- `bool IsMouseMoveEvent()`: Geeft aan dat het laatste opgevangen Allegro event al dan niet een muisgerelateerd event was waarbij er geklikt werd met de muis;
- `void ShowError(string msg)`: Print een foutmelding msg naar het scherm.

Merk op dat AllegroLib als singleton werd geïmplementeerd. Opvragen van de instance doe je dus steeds met de statische methode `AllegroLib::Instance()`, waarop de nodige methoden opgeroepen kunnen worden.

3.10.5 Graphics

De klasse Graphics voorziet een aantal methoden om verschillende fonttypes en afbeeldingen in te laden, en om tekst en afbeeldingen weer te geven op het scherm. Ook deze klasse werd als singleton geïmplementeerd. De volgende methoden werden voorzien:

- `void LoadFonts()`: De fonts worden ingeladen en opgeslagen in een lokale variabele;
- `void LoadSpriteCache()`: De sprites worden ingeladen in een vector van ALLEGRO_BITMAP*, die als cache gebruikt kan worden. De vector is geïndexeerd door middel van een enum met de namen van de sprites;
- `void UnLoadFonts()`: Verwijder de ingeladen fonts;
- `void UnLoadSpriteCache()`: Verwijder de ingeladen sprites;
- `void ExecuteDraws()`: Wissel de bitmap in het geheugen met die op het scherm;
- `void ClearScreen()`: Leeg het scherm door het zwart te schilderen;
- `void SetBackground(Sprite sprite)`: Stel de achtergrondafbeelding in;
- `void DrawBackground()`: Teken de huidige achtergrondafbeelding;
- `void DrawBitmap(Sprite sprite, float dx, float dy, float cx, float cy)`: Teken de sprite, waarbij de coördinaten dx en dy overeenkomen met de offset ten opzichte van de linkerbovenhoek van het scherm, en de coördinaten cx en cy overeenkomen met een offset ten opzichte van de linkerbovenhoek van de afbeelding;
- `void DrawString(string str, float dx, float dy, Color c, Align align)`: Teken een string op positie (dx, dy) in de kleur c en met de opgegeven uitlijning.

3.11 Constanten

Tabel 2 definieert de constanten die gehanteerd worden in dit project. Alle maten van snelheid en versnelling worden uitgedrukt in de eenheid "per frame". De header Constants.h kan ingeladen worden om deze variabelen eenvoudiger aan te passen. Merk op dat het toegelaten en zelfs aangeraden is dit bestand verder uit te breiden, zodat aanpassingen eenvoudiger uitgevoerd kunnen worden.

Verklaring	**Waarde**
Afmetingen van de bal	23 x 23 (straal 12)
Verticale startpositie van de bal	133.5
Horizontale versnelling van de bal	0
Verticale versnelling van de bal	-0.375
Afmetingen van de slimes	79 x 40 (straal 40)
Horizontale startpositie van de linkse slime	150
Horizontale startpositie van de rechtse slime	600
Horizontale versnelling van de slimes	0
Verticale versnelling van de slimes	-0.750
Snelheid naar links of rechts wanneer de slime beweegt	6
Initiële snelheid naar boven wanneer de slime omhoog springt	11.625
Offset van het centrum van het oog voor de linkse slime	(20, 20)
Offset van het centrum van het oog voor de rechtse slime	(-20, 20)
Afmetingen van het net	4 x 47
Coördinaten van het net	(373, 39)
Freeze time na het scoren van een punt [s]	1.2
Aantal te scoren punten per spel of level	7
Minimale afspeelsnelheid replay	0.5
Maximale afspeelsnelheid replay	4

Tabel 2: Constanten in het spel.

4 Bestanden

Een deel van de code wordt bij de opgave meegegeven. Het is de bedoeling de bestanden waar nodig aan te passen om zo tot een werkende oplossing te komen. De onderstaande bestanden zijn geheel of gedeeltelijk gegeven.

4.1 Headerbestanden

De headerbestanden omvatten onder meer de verschillende componenten en systems, zoals reeds besproken in Sectie 3. Het wordt aangeraden de signatuur zoveel als mogelijk te behouden, al is het toegestaan om kleine aanpassingen te doen wanneer nodig. Zorg er wel voor dat dit duidelijk is door het nodige commentaar bij extra functies en variabelen te voorzien.

4.2 Implementatiebestanden

De implementatiebestanden omvatten de logica voor het spel. De bestanden main.cpp, allegro_lib.cpp, graphics.cpp, game.cpp, entity.cpp, entity_stream.cpp, engine.cpp en context.cpp werden reeds volledig geïmplementeerd, alle andere moeten nog worden aangevuld.

4.3 Assets

De folders fonts en images bevatten de benodigde lettertypen en afbeeldingen om het spel te spelen. De folder highscores bevat een enkel bestand highscores.txt, maar zal uitgebreid worden met bestanden met de coördinaten van de bewegende entiteiten in een singleplayer spel.

5 Compileren

5.1 Windows

5.1.1 Visual Studio 2015

Er wordt aangeraden om Visual Studio 2015 Community te gebruiken op Windows, aangezien de bijhorende compiler gebruikt zal worden bij het verbeteren van dit project. De software kan gedownload worden via de officiële website: https://www.visualstudio.com/downloads/download-visual-studio-vs/. Eenmaal geïnstalleerd, kan de solution file "SlimeVolley/Slimevolley.sln"van het project geopend worden. Hierin werden alle properties van het project juist ingesteld en wordt de Allegro library ingeladen. Merk op dat de 32-bit compiler moet gebruikt worden om het project te compileren met Allegro. Bij het uitvoeren van de gegeven opgavebestanden zal er zonder wijzigingen reeds een beperkte GUI weergegeven worden. Meer details hieromtrent staan beschreven in een aan te raden stappenplan in Sectie 6.

5.1.2 Stand-alone executable

Eens het spel af is, kan het leuk zijn het te exporteren en spelen op een andere computer. Op de volgende manier kan je een executable maken die stand-alone uitgevoerd kan worden:

- Build het project in Release mode in plaats van Debug mode;
- Kopieer de executable uit de Solution/Release folder naar een locatie L naar keuze;
- Kopieer de map assets in de Project folder naar L;
- Kopieer allegro-5.0.10-monolith-md-debug.dll en msvcr120d.dll uit allegro/bin naar L;

De executable zou nu zonder problemen moeten uitvoeren. Alle bestanden en folders in L kunnen gezipt worden, zodat ze overgezet kunnen worden naar een andere computer.

5.2 Linux

Tijdens dit project bieden wij geen ondersteuning voor niet-Windows systemen. Studenten die dit echt willen, kunnen hun project maken in Linux, maar moeten erop toezien dat de code compileerbaar en uitvoerbaar moet zijn met de compiler aangeboden door Visual Studio.

5.2.1 Installeren van Allegro

Om Allegro te compileren op Linux, moet Allegro 5.0 eerst geïnstalleerd worden. Hoe Allegro installeren hangt af van de specifieke Linux distributie. Voor Ubuntu probeer je best eerst deze methode: https://wiki.allegro.cc/index.php?title=Install_Allegro_from_Ubuntu_PPAs. Als dat niet lukt, kan je proberen Allegro zelf te compileren (bijvoorbeeld via volgende instructies: https://wiki.allegro.cc/index.php?title=Install_Allegro5_From_Git/Linux/Debian).

5.2.2 Compileren van de code

Om de code te compileren moeten er enkele packages, zoals de compiler, voorhanden zijn. Op vele Linux systemen zal dat al het geval zijn. Op Ubuntu kunnen ze, indien nodig, geïnstalleerd worden via "sudo apt-get install build-essential pkg-config". Ga vervolgens naar de map waar Allegro geïnstalleerd is en voer daar "make" uit. Dit zal automatisch de installatie van Allegro detecteren en alles compileren. Merk op dat, om het project zelf te kunnen compileren op Linux, bepaalde functies (zoals fopen_s en localtime_s) aangepast zullen moeten worden.

6 Stappenplan

Hieronder wordt een voorstel gegeven voor een stappenplan voor dit project. Het is niet verplicht dit plan te volgen, maar het kan een leidraad zijn om het project tot een goed einde te brengen. Er wordt aangeraden eerst de multiplayer te maken, vervolgens de singleplayer en ten slotte de replay. Wat het entity-component-system framework betreft, moeten enkel de spelspecifieke systemen nog geïmplementeerd worden. Om dit zo overzichtelijk mogelijk te houden, wordt er best systeem per systeem gewerkt, zoals hieronder weergegeven.

1. Controleer dat de gegeven solution correct ingeladen kan worden in Visual Studio. Zorg ervoor dat het project compileert en uitvoerbaar is (gebruik de 32-bit compiler om te compileren met Allegro!). Als alles juist is, wordt er bij het opstarten van het spel een startmenu weergegeven, en een statische afbeelding met enkele sprites wanneer op een van de mogelijke opties geklikt wordt.

2. Implementeer de nodige functies in de klasse GameMulti, waarbij voorlopig enkel de variabele van de klasse SystemRender ingeladen wordt. Maak één entiteit aan met de component Bal, twee entiteiten met de component Player, veertien entiteiten met de component Point (tweemaal zeven, het maximum aantal punten per speler) en het net met enkel de component Sprite. Implementeer een eerste versie van de klasse SystemRender, die in elke iteratie alle entities met een component Sprite op de juiste plaats weergeeft. Wanneer dezelfde coördinaten gebruikt worden als in het voorbeeld (tijdelijk in elke game loop), zal een gelijkaardige statische afbeelding weergegeven worden als in stap 1.

3. Implementeer de Update()-functie in de klasse SystemMotion, voeg het systeem toe in de klasse GameMulti en voeg een component Motion toe aan de bal en slimes. Zowel de bal als de slimes zouden nu verticaal naar beneden moeten vallen bij het opstarten van het spel.

4. Implementeer de nodige functies in de klasse SystemCollision, die er onder meer voor zal zorgen dat een slime zijn grenzen respecteert (i.e. op zijn helft van het veld blijft en geen y-waarde onder 0 kan hebben) en dat de bal in het spel blijft. Voeg het systeem toe in de klasse GameMulti en start het spel opnieuw op. De bal zou nu verticaal moeten blijven botsen op de linker slime.

5. Implementeer de Update()-functie in de klasse SystemInputMulti, die het al dan niet ingeduwd zijn van bepaalde toetsen omzet in een snelheid naar links, rechts of onder. Voeg het systeem toe in de klasse GameMulti, en het zou nu mogelijk moeten zijn de slimes te besturen.

6. Implementeer de Update()-functie in de klasse SystemStateMulti, die detecteert wanneer de bal de grond raakt en vervolgens de nodige acties onderneemt. Voeg het systeem toe in de klasse GameMulti, en het zou nu mogelijk moeten zijn om een vroege versie van het spel te spelen. Pas meteen ook de klasse SystemRender aan om, afhankelijk van de huidige game state, een boodschap naar het scherm te printen.

7. Implementeer de Update()-functie in de klasse SystemPoints, die de sprites van de punten aanpast aan de huidige score tussen de twee spelers. Voeg het systeem toe in de klasse GameMulti, en het zou nu mogelijk moeten zijn om de voortgang van het spel visueel te volgen.

8. Implementeer de Update()-functie in de klasse SystemEyes, die de pupillen van de ogen op de juiste plaats zet voor elke slime. Voeg het systeem toe in de klasse GameMulti, en het zou nu mogelijk moeten zijn om een volledige versie van het multiplayer spel te spelen.

9. Doe hetzelfde voor de klasse GameSingle als in stap 3, maar houd rekening met de verschillende entities per level.

10. Implementeer de Update-functies in de klassen SystemInputSingle en SystemStateSingle en voeg alle benodigde systemen toe aan de klasse GameSingle. Het zou nu mogelijk moeten zijn om een singleplayer spel te spelen tegen een niet-bewegende tegenstander.

11. Implementeer een eerste versie van de AI in de klasse SystemAI, gebruik makende van de opgegeven pseudocode. Voeg het systeem toe en werk de implementatie stap voor stap bij. Het eindresultaat is een werkende singleplayer met twee levels en nodige feedback op het scherm.

12. Implementeer de Update()-functie in de klasse SystemOutput, die de (x, y) coördinaten van de twee slimes en de bal wegschrijft naar een outputbestand zodra een level voorbij is.

13. Implementeer de klasse SystemReplay, die een inputbestand inleest naar een lijst van Coordinates en de positie van de drie bewegende entiteiten frame per frame aanpast. Het eindresultaat is een werkende replay van de spellen in de highscores, waarbij de nodige functionaliteiten (trager/sneller afspelen, doorgaan naar het volgende punt of level) aanwezig zijn. **Wanneer deze versie van het spel correct gecompileerd kan worden met Visual Studio en er geen memory leaks aanwezig zijn, voldoet je implementatie aan de voornaamste criteria van dit project.**

14. Implementeer vrijblijvend een aantal uitbreidingen op het spel. Mogelijkheden zijn een extra level in de singleplayer, het gebruik van meerdere slimes of ballen en het toepassen van nieuwe gravitatiewetten. Merk op dat er in dit geval twee versies van het spel ingediend moeten worden: de basisversie uit de voorgaande stap en de uitgebreide versie uit deze stap.

7 Tips en opmerkingen

Hieronder worden een aantal tips en opmerkingen gegeven waar best rekening mee gehouden wordt in dit project:

- Aangezien dit project met twee studenten gemaakt dient te worden, zal een vlotte samenwerking noodzakelijk zijn. Het gebruik van GitHub (https://github.ugent.be/) wordt hierbij aangeraden, zolang de repository privaat(!) is. Een beknopte GitHub tutorial werd voorzien (https://https://github.com/IBCNServices/tutorials/blob/master/Github.md), neem deze zeker even door;
- Begin niet onmiddellijk te programmeren. Lees eerst aandachtig de opgave en denk na hoe je de verschillende problemen zou oplossen;
- Het voordeel van een entity-component-system framework is dat je bepaalde systemen kan uitschakelen tijdens het debuggen door ze tijdelijk niet aan de Engine toe te voegen;
- Maak zoveel mogelijk gebruik van de Standard Template Library (STL) (http://www.cppreference.com/wiki/stl/start);
- Alle opmerkingen bij de practica gelden ook hier: wees zuinig met geheugen, let op voor dangling pointers, geef alle gealloceerde geheugen ook weer vrij, controleer of bestanden wel correct geopend en gesloten worden enzovoort;
- Als programmeerstijl werd de Google C++ Style Guide gebruikt (https://google.github.io/styleguide/cppguide.html). Vele andere opties waren echter mogelijk, een enkel "juistälternatief bestaat niet (zelfs onder de begeleidende assistenten van dit vak bestaan er veel verschillende stijlen). We vragen wel om, in de kader van dit project, de gehanteerde stijl zoveel mogelijk te respecteren.
- Als referentiecompiler wordt Microsoft Visual C++ Community Edition 2015 gebruikt. Zorg er dus voor dat je project daarmee compileert en uitvoerbaar is. Behalve het keyword auto is gebruik van C++11 toegestaan;
- Probeer compiler warnings te vermijden; deze duiden meestal op fouten die Visual Studio C++ voor de programmeur zal oplossen, maar die evengoed vermeden kunnen worden door de code aan te passen;
- Vanzelfsprekend zijn er zaken waar deze opgave je vrij in laat; deze zijn naar eigen inzicht in te vullen. Verduidelijk in elk geval je broncode met commentaar waar dit nuttig is;
- Sluit het spel telkens af met het kruisje op de UI. Wanneer de console zelf afgesloten wordt, zal de Allegro bibliotheek niet altijd correct opgeruimd kunnen worden en kunnen eigenaardige fouten optreden.
- Algemene vragen over het project kunnen steeds gesteld worden via het forum, specifieke vragen via pgm@lists.ugent.be. Wij trachten alle vragen zo snel mogelijk te beantwoorden, maar uiteraard in de mate van het mogelijke.

8 Indienen

Het project wordt gemaakt in groepen van twee studenten, vanaf 31 maart 2017 in te geven via Minerva. Om in te dienen zip je de broncodebestanden (.h en .cpp, eventueel samen met een mogelijke uitbreiding op het spel) samen met een verslag in een bestand project_nr.zip (met nr het Minerva groepsnummer). Dit bestand wordt via de Minerva dropbox ingediend bij Bruno Volckaert en Femke De Backere, en dit ten laatste op zondag 21 mei 2017, 23u59. Het verslag is van het type PDF, is maximaal twee pagina's lang en omvat volgende zaken:

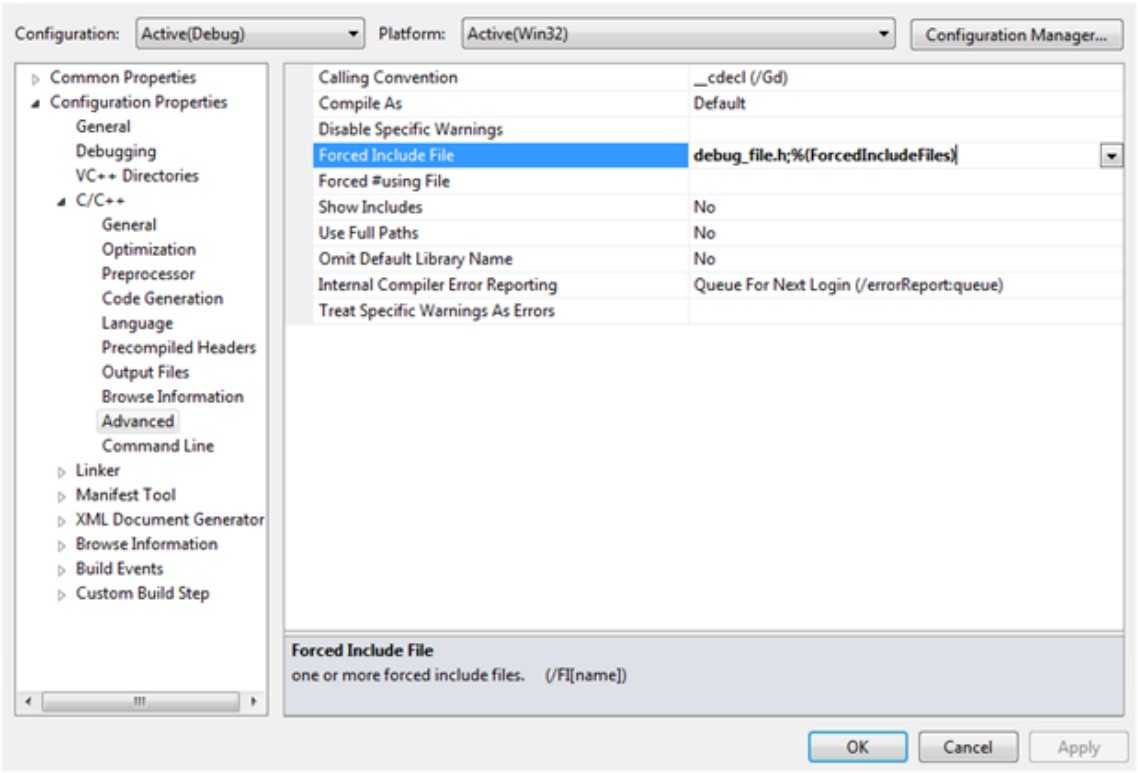
- Naam, voornaam en richting van de groepsleden, plus Minerva groepsnummer
- Een korte uitleg over de belangrijkste ontwerpbeslissingen, zoals de beslissingen van de AI
- Bij het niet (correct) uitvoeren van het programma: waarom?
- De taakverdeling: wie heeft wat gedaan?

Als minimumvoorwaarde wordt vereist dat het project compileerbaar is met de referentiecompiler in Visual Studio 2015, zoniet wordt een nulscore toegekend. Wanneer het programma niet (correct) uitgevoerd kan worden, wordt er gevraagd hierover een korte toelichting te geven in het verslag.

9 Appendix: memory leaks in Visual Studio

Naast de gekende methode voor het vinden van memory leaks (gebruikt in de practica), bestaat er een manier om dit proces ietwat te stroomlijnen. In plaats van de locatie van het geheugenlek te moeten gebruiken om dan tijdens het opnieuw runnen van je programma een breakpoint te zetten met _CrtSetBreakAlloc(long), kan er gedubbelklikt worden op het gevonden lek, om direct naar het bestand en de lijn van de allocatie te gaan.

Via forced include(*) in Visual Studio gaan we bij elk bestand het bestand debug_file.h includen. De forced include functionaliteit in Visual Studio kan teruggevonden worden bij Project Properties (rechtsklikken op het project in de Solution Explorer), onder Configuration Properties > C/C++ > Advanced > Forced Include File (zie Figuur 6.



Figuur 6: Force include in Visual Studio.

Het bestand debug_file.h bevat de volgende regels:

```
#pragma once
#ifdef _DEBUG
        #define _CRTDBG_MAP_ALLOC
        #include <stdlib.h>
        #include <crtdbg.h>

        #ifndef DEBUG_NEW
                #define DEBUG_NEW new ( _NORMAL_BLOCK , __FILE__ , __LINE__ )
                #define new DEBUG_NEW
        #endif
#endif
```

Wat dit bestand exact doet, wordt hieronder lijn per lijn uitgelegd.

```
#pragma once
```

Dit is een compiler-geoptimaliseerd statement om te zorgen dat er geen dubbele includes gebeuren.

```
#ifdef _DEBUG
...
#endif
```

Dit statement zorgt ervoor dat de code enkel uitgevoerd wordt als _DEBUG gedefinieerd is. Bij het builden in Debug mode in Visual Studio wordt automatisch de _DEBUG-vlag gedefinieerd.

```
        #define _CRTDBG_MAP_ALLOC
        #include <stdlib.h>
        #include <crtdbg.h>
```

De includes zijn gekend van de normale manier van debuggen via de crtdbg bibliotheek. Het define statement vervangt de malloc-statements in de code door een malloc met argumenten, waar de constanten __FILE__ en __LINE__ in meegegeven worden. Dit zijn compiler-gedefinieerde constanten die de bestandsnaam en het lijnnummer meegeven waar ze worden opgeroepen.

```
        #ifndef DEBUG_NEW
                #define DEBUG_NEW new ( _NORMAL_BLOCK , __FILE__ , __LINE__ )
                #define new DEBUG_NEW
        #endif
```

Hier gebeurt exact hetzelfde, maar dan manueel bijgevoegd voor het new commando.
Vanaf nu zullen memory leaks op het einde van het programma dubbelklikbaar zijn, zodat rechtstreeks naar het bestand en de regel van de allocatie kan gegaan worden, zonder gebruik te moeten maken van _CrtSetBreakAlloc(long _BreakAlloc).
Om zeker te zijn dat geheugen slechts gedumpt wordt na het eindigen van de main()-methode en dus ook geen statische variabelen die nog in het geheugen zitten op dat moment te rapporteren, gebruik je in plaats van _CrtDumpMemoryLeaks() op het einde van je programma, beter volgende vlag in het begin van je programma:

```
int main()
{
        // Debug flag
        _CrtSetDbgFlag ( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );

        // Programma code
}
```

Merk op dat de Runtime Library (Configuration Properties > C/C++ > Code Generation > Runtime Library) MTd of MDd moet zijn om de _DEBUG vlag te definiëren, en dus het detecteren van memory leaks mogelijk te maken.