

Geavanceerde softwareontwikkeling Project

Survival Shooter Game

Inhoud

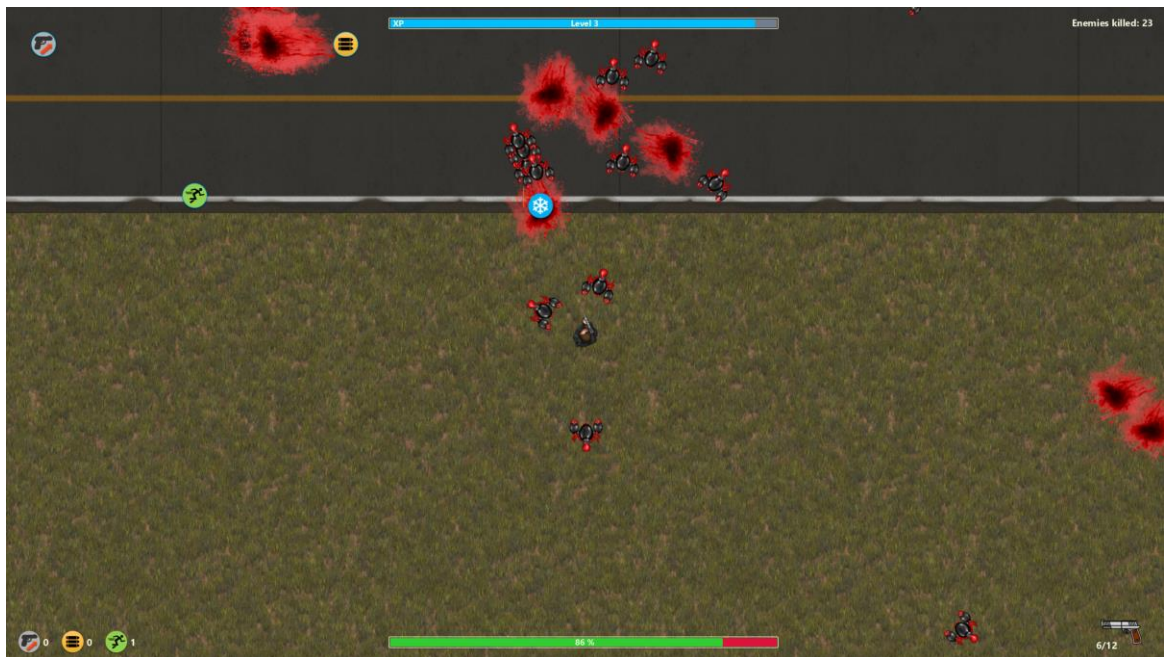
1	Doel	2
2	Opdracht	2
3	Spelconcepten	3
3.1	Wapens.....	3
3.2	Monsters	3
3.3	Level-up mechanisme	4
3.4	Bonussen	5
4	Technisch.....	6
4.1	Java FX & Game development	6
4.1.1	De Game Loop	6
4.1.2	Canvas	7
4.1.3	Hulpklassen	7
4.1.4	Sprites.....	8
4.1.5	Sounds	8
4.2	Entity System Framework	8
4.2.1	Entiteiten	9
4.2.2	Componenten.....	9
4.2.3	Systemen	9
4.2.4	Engine	9
4.3	Wiskundige tips	12
4.3.1	Bewegingsvector bepalen	12
4.3.2	Hoek tussen twee punten berekenen.....	13
4.4	Resources	13
5	Indienen	14

1 Doel

Binnen het geïntegreerd project van het vak Geavanceerde Softwareontwikkeling wordt een applicatie ontwikkeld, gespreid over meerdere labo-sessies heen. Hierdoor worden zowel de ontwikkeltijd als de complexiteit van het eindresultaat hoger en kunnen de verschillende technieken die gedoceerd worden binnen dit vak, toegepast worden in een real-life context.

2 Opdracht

In het project wordt een Survival Shooter spel uitgewerkt dat losjes gebaseerd is op Crimsonland uit 2003. Vanuit bovenaanzicht bestuurt de speler een personage dat met behulp van een gevarieerd wapenarsenaal, probeert te overleven in een wereld waar continue meer en meer monsters op hem af komen. De speler wordt hierbij geholpen door bonus items die met een bepaalde probabilliteit verschijnen wanneer een vijand gedood wordt. Tenslotte krijgt de speler “experience points” (XP) per gedode vijand en zal hierdoor af en toe stijgen in “level”. Dit maakt de speler krachtiger doordat hij dan telkens kan kiezen uit een aantal upgrades. Het doel van het spel is om een zo hoog mogelijk level te behalen.



Figuur 1: Screenshot demo

3 Spelconcepten

3.1 Wapens

De speler start het spel met een handgun, maar kan alternatieve wapens vinden door vijanden te doden. Elk van de alternatieve wapens heeft z'n specifieke sterktes en zwaktes:

Type	Schade	Bereik	Accuraatheid	Herlaadsnelheid	Ammo
Handgun 	Laag	Ver	Goed	Snel	12
Shotgun 	Hoog	Heel kort	Veel Spreiding	Traag	6
Automatic Rifle 	Gemiddeld	Ver	Beetje Spreiding	Snel	30
Grenade Launcher 	Heel Hoog	Gemiddeld	Goed	Traag	5

Jullie zijn vrij extra wapens te introduceren, maar bovenstaande types moeten minstens geïmplementeerd zijn.

3.2 Monsters





Het gedrag van de monsters wordt eigenlijk op een vrij eenvoudige manier geïmplementeerd. Volgens een logaritmisch stijgende functie worden een aantal monsters toegevoegd aan het spel op een geregeld interval. Deze monsters krijgen een willekeurige locatie binnen het speelveld toegewezen en zullen vanaf daar steeds naar de speler proberen toelopen. Verder onderscheiden we drie types van monsters:

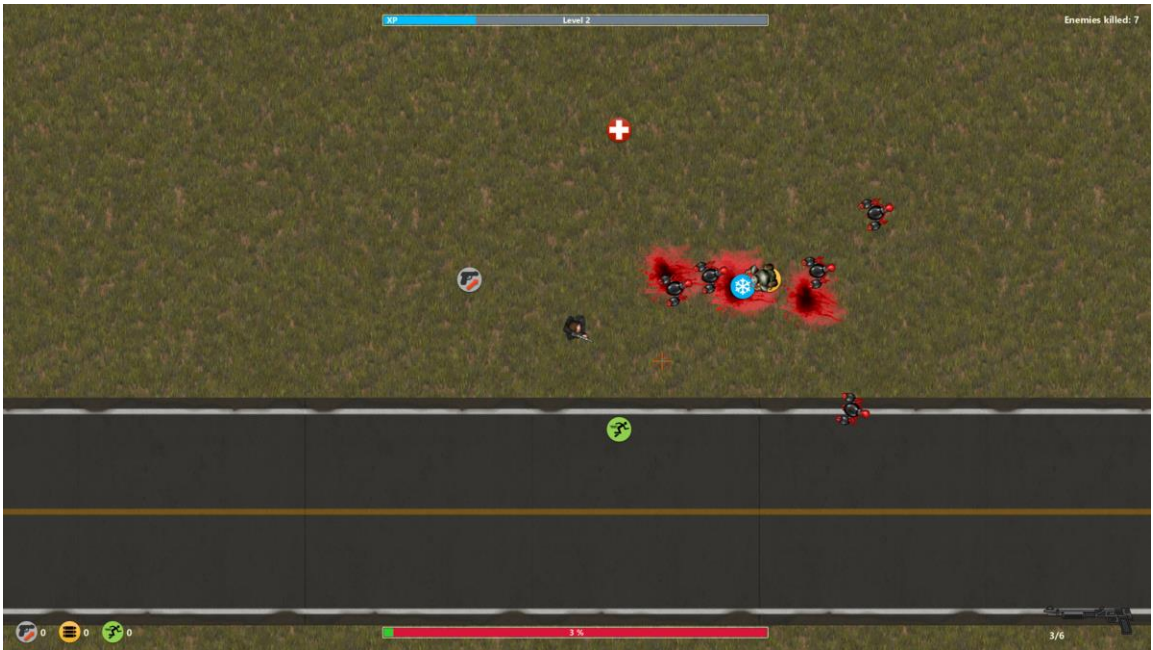
Type	Snelheid	Schade	Aantal Levens
Scout 	Hoog	Klein	Laag
Soldier 	Gemiddeld	Gemiddeld	Gemiddeld
Enforcer 	Laag	Hoog	Heel Hoog

Jullie zijn vrij extra monsters te introduceren, maar bovenstaande types moeten minstens geïmplementeerd zijn.

3.3 Level-up mechanisme

De speler kan stijgen in level door XP te verdienen dat wordt toegekend bij het doden van vijanden. Het benodigd aantal XP om te stijgen naar een volgend level zal steeds stijgen volgens een logaritmische curve. Bij het stijgen in level krijgt de speler keuze tussen de volgende vier upgrades:

Type	Beschrijving
Restore Health 	De speler kan deze upgrade kiezen om z'n leven weer tot het maximum niveau te herstellen. Hierbij moet dus de afweging gemaakt worden of dit nodig zal zijn om te overleven tot aan het volgend level, of als de speler beter kiest voor een wapen- of snelheidsupgrade.
Upgrade Ammo 	De speler kan deze upgrade kiezen om hoeveelheid ammo voor z'n huidig wapen uit te breiden met een bepaald percentage. Hierbij moet dus de afweging gemaakt worden of de speler dit wapen zal behouden, anders is het misschien beter te kiezen om de levens te herstellen of een snelheidsupgrade op te nemen.
Upgrade Damage 	De speler kan deze upgrade kiezen om de hoeveelheid schade van z'n huidig wapen te verbeteren met een bepaald percentage. Hierbij moet dus de afweging gemaakt worden of de speler dit wapen zal behouden, anders is het misschien beter te kiezen om de levens te herstellen of een snelheidsupgrade op te nemen.
Upgrade Speed 	De speler kan deze upgrade kiezen om de snelheid waarmee hij op de map kan rondlopen te verbeteren. Dit helpt om de vijand te ontwijken, maar heeft geen effect op de schade die een speler kan berokkenen.







Figuur 2: Voorbeeld level-up

Op Figuur 2 kan je zien hoe de keuzemogelijkheden voor upgrades rond de speler tevoorschijn komen bij een level-up. Wanneer de speler een upgrade kiest door erover te lopen, verdwijnen de andere mogelijkheden uit de wereld.

3.4 Bonussen

Naast het level-up mechanisme kan de speler zich berusten op tijdelijke bonussen om langer te overleven:

Type	Beschrijving
Freeze Bonus 	Bij het oprapen van deze bonus, worden tijdelijk alle vijanden “bevroren” waardoor ze niet meer kunnen bewegen. Er worden ook geen nieuwe vijanden meer in het spel geplaatst tijdens de duurtijd van de bonus. De speler kan wel nog steeds schieten op bevroren vijanden.
Fire Bonus 	Bij het oprapen van deze bonus, schiet de speler tijdelijk met “vuurkogels”. Voor de handgun, shotgun en rifle varianten, betekent dit dat de projectielen vervangen worden door een vuur-sprite en er meer schade verricht wordt. Voor de grenade launcher wordt de granaat vervangen door een soort molotov dat een grotere explosie tot gevolg heeft. Daarnaast moet de speler niet herladen zolang deze bonus geldt.
Shield Bonus 	Bij het oprapen van deze bonus, krijgt de speler tijdelijk een schild dat alle schade absorbeert. Wanneer de duurtijd van de bonus afgelopen is, gaat de speler verder met evenveel leven als voor het opnemen van de bonus.
Bomb Bonus 	Bij het oprapen van deze bonus wordt op de locatie van de bonus een enorme explosie gecreëerd die veel schade aanricht aan alle vijanden in een bepaalde straal rond deze locatie. De speler zelf verliest hier geen levens door.

4 Technisch

4.1 Java FX & Game development

4.1.1 De Game Loop

De belangrijkste component van eender welk spel – vanuit het standpunt van de programmeur – is de game loop. De game loop zorgt ervoor dat het spel vlot zal draaien, onafhankelijk van het feit of de gebruiker al dan niet input genereert. Een ander belangrijk aspect hierbij is het visuele. 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 te zien, is veel ingewikkelder dan meestal wordt aangenomen. Er zijn een paar vuistregels die in de meeste gevallen (blijken te) kloppen:

- Hoe meer FPS (frames per second) hoe vloeiender het beeld.
- Hoe kleiner het verschil tussen (de beeldinformatie op) de verschillende frames hoe vloeiender de beweging.

Dit is niet in alle gevallen correct, maar voor ons volstaat het om hier zo over te denken.

Het is gemakkelijk om de snelheid van de game loop (logische lus) ook in frames per second uit te drukken, op die manier is er een maat om mee te vergelijken. Frames per second is dan eigenlijk hoeveel keer per seconde de gamestate aangepast wordt.

Het aantal FPS is dus bij games enerzijds gelimiteerd door de hardware van de computer, maar ook door hoe zwaar de game loop wordt. Bij een actie ondernomen door de speler die meer berekeningen vraagt bijvoorbeeld, zal een game loop-iteratie langer duren en dus het aantal zichtbare FPS tijdelijk dalen.

In JavaFX 8 kunnen we gebruik maken van de abstracte klasse `AnimationTimer` om de game loop te implementeren. Deze klasse heeft één enkele abstracte methode die moet geïmplementeerd worden, namelijk `handle(long now)`. Jullie implementatie van deze methode zal dus één enkele iteratie van jullie game loop specificeren. Door dan de `start()` methode op te roepen die al voorzien wordt door de `AnimationTimer` zal JavaFX ervoor zorgen dat vanaf dit moment de methode `handle` 60 keer per seconde opgeroepen wordt (60 fps).

Merk op dat dit zoals gezegd dus kan variëren afhankelijk van hoe zwaar de game loop wordt. Om de mogelijkheid te geven hiervoor te compenseren, beschikt `handle` over het argument `now` dat het tijdstip van het huidige frame in nanoseconden voorstelt. Je zou op basis hiervan een factor kunnen afleiden die je dan bv. kan toepassen op de snelheid waarmee de game objecten bewegen, zodat ondanks een lager aantal frames per seconde het spel correct blijft werken. **Voor deze opgave mogen jullie echter deze correctiefactor negeren!**

4.1.2 Canvas

Natuurlijk moet onze implementatie kunnen tekenen naar het scherm. Voor geavanceerdere spellen worden APIs zoals DirectX of OpenGL gebruikt om op een vrij low-level manier de grafische processor aan te sturen. In dit project maken wij de abstractie door de Canvas klasse van JavaFX te gebruiken, want het tekenen een stuk eenvoudiger (maar ook minder krachtig maakt).

Om een Canvas aan te maken en in te stellen als de scene voor de primaryStage kunnen we volgende code schrijven:

```
public void start(Stage primaryStage) {  
  
    ...  
    Canvas canvas = new Canvas(SCREEN_WIDTH, SCREEN_HEIGHT);  
    Scene gameScene = new Scene(new Group(canvas));  
    primaryStage.setScene(gameScene)  
    ...  
}
```

Aan de canvas instantie kunnen we een GraphicsContext instantie vragen om dan effectief iets te tekenen:

```
GraphicsContext gc = canvas.getGraphicsContext2D();  
gc.setFill(Color.RED);  
gc.setFont(Font.font(24));  
gc.fillText("Hello World!", exampleX, exampleY);
```

4.1.3 Hulpklassen

Stel x, y-coördinaten voor aan de hand van de klasse Point2D en gebruik Rectangle2D om de omvang van een object voor te stellen enerzijds, maar anderzijds ook om botsingen tussen objecten na te gaan (collision detection). Dit zal jullie veel tijd uitsparen doordat heel wat wiskundige berekeningen op deze manier onmiddellijk al geïmplementeerd zijn, bijvoorbeeld het controleren of er een doorsnede is tussen twee rechthoeken:

```
Rectangle2D a = createRandomRectangle();  
Rectangle2D b = createRandomRectangle();  
  
if(a.intersects(b)) {  
    System.out.println("Rectangle intersection detected!");  
}
```


4.1.4 Sprites

Maak gebruik van de klasse `Image` om sprites in te laden en te tekenen, bv.:

```
Image sprite = new Image(new FileInputStream("sprite.png"));
gc.drawImage(sprite, exampleX, exampleY);
```

4.1.5 Sounds

JavaFX beschikt over de klasse `AudioClip` om audio-bestanden in te laden en af te spelen, bv.:

```
AudioClip plonkSound = new AudioClip("http://somehost/path/plonk.aiff");
plonkSound.play();
```

4.2 Entity System Framework

Ervaring leert ons 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 functionaliteit. Voorbeelden hiervan zijn:

- Teken 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 die dit gebruiken kunnen overerven, bv. een speler erft over van de klasse `InputController` om te kunnen reageren op input. Maar wat als het gedrag at runtime plotseling moet veranderen op basis van de toestand? Bv. de speler dient tijdelijk te worden bestuurd door de AI voor een bepaalde cutscene.

Bovenstaande problematiek wordt ook erkend in de game industrie en één van de voorgestelde oplossingen is het gebruik van een Entity System Framework. Bij deze data gedreven aanpak worden de verschillende game-objecten niet meer voorgesteld als afzonderlijke klassen, maar als generieke entiteiten die samengesteld worden aan de hand van een bepaald aantal componenten (m.a.w. Composition over Inheritance). Deze componenten bevatten heel specifieke data die kunnen gekoppeld worden aan een game-object (bv. de positie).

Entities en hun bijhorende componenten zijn dus pure data containers en implementeren geen functionaliteit of gedrag. Dat is de verantwoordelijkheid van de verschillende entity systems die zullen inwerken op de geregistreerde entiteiten. Voorbeelden hiervan zijn een systeem om bewegingsvectoren toe te passen en een systeem dat entities op het scherm kan tekenen.

Een entity system framework bestaat uit vier concepten: Entiteiten, Componenten, Systemen en ten slotte de Engine:

4.2.1 Entiteiten

Entiteiten zullen dus de verschillende game objecten voorstellen (zoals de speler, vijanden, projectielen, etc...). Op een entiteit kunnen volgende methodes opgeroepen worden:

- **void** `add(Component component)` : Voegt een component toe aan deze entiteit.
- **void** `remove(Class<? extends Component> componentClass)` : Verwijdert een component aan de hand van de gegeven klasse.
- `<T extends Component> T get(Class<T> componentClass)` : Geeft een component terug aan de hand van de gegeven klasse.
- **boolean** `has(Class<? extends Component> componentClass)` : Geeft `true` terug als de entiteit over een component beschikt van de gegeven klasse, anders `false`.

4.2.2 Componenten

Componenten kunnen alle klassen zijn die de lege interface `Component` implementeren. Een component instantie zal enerzijds als data container dienen voor een specifieke set aan gegevens (bv. een x, y-locatie) en zal anderzijds ook als identificatiemiddel dienen (wanneer we bijvoorbeeld alle Entiteiten met een `TextureComponent` opvragen om te tekenen naar het scherm).

Typische voorbeelden van componenten zijn een `TransformComponent` (dat de x, y, z-positie van een object voorstelt, samen met de rotatie) en een `AnimationComponent` (dat een bepaalde entiteit van een animatie voorziet).

Het is de bedoeling dat jullie zelf nadenken over de componenten die je nodig hebt.

4.2.3 Systemen

Systemen zijn klassen die het interface `EntitySystem` implementeren en de eigenlijk logica van het game zullen bepalen. `EntitySystem` bevat één enkele methode `update()` die elke iteratie van de gameloop (bv. 60 keer per seconde) zal opgeroepen worden door de Engine (zie volgende sectie).

Een typisch voorbeeld van een Systeem is `MovementSystem` dat elke iteratie alle Entiteiten met een bewegingsvector (opgeslaan in een bepaalde component) zal beschouwen en deze zal toepassen op de huidige positie van de entiteit (ook te vinden in een component).

Het is de bedoeling dat jullie zelf nadenken over de specifieke system die je nodig hebt.

4.2.4 Engine

De Engine brengt alle bovenstaande concepten samen en zorgt dat de functionaliteit voorgesteld door de systemen kan werken. Op de opgegeven implementatie van Engine kunnen volgende methodes opgeroepen worden:

- **void** add(Entity entity) : Voegt een entiteit toe aan de engine dat vanaf nu in aanmerking komt om verwerkt te worden door de geregistreerde systemen.
- **void** remove(Entity entity) : Verwijdert een entiteit uit de engine, deze zal vanaf nu niet meer in beschouwing genomen worden door de systemen.
- Collection<Entity> getEntitiesWithComponents(Class<? **extends** Component>... componentClasses): Geeft een collectie van entities terug die over de gegeven componenten beschikken.
- Optional<Entity> getEntityWithComponents(Class<? **extends** Component>... componentClasses) : Geeft de eerste gevonden entiteit terug die over de gegeven component beschikt als een Optional. Deze methode is bedoeld als een hulpmethode wanneer je werkt met entiteiten waarvan je zeker bent dat er maximaal één van aanwezig is in de engine (bv. de player).
- <T **extends** EntitySystem> T registerSystem(Class<T> systemClass) : Registreert een systeem bij de engine aan de hand van de gegeven klasse. De engine zal zelf een instantie van de klasse aanmaken en de @Inject afhankelijkheden beheren (via de Java Reflection API). De update() methode van deze instantie zal enkel opgeroepen worden wanneer alle afhankelijkheden voldaan zijn. Systemen worden opgeroepen in de volgorde dat ze toegevoegd zijn.
- <T **extends** EntitySystem> T getSystem(Class<T> systemClass) : Geeft het geregistreerde systeem terug op basis van de gegeven klasse.
- **void** unregister(EntitySystem system) : Verwijdert een systeem uit de engine, vanaf dit moment zal het gegeven systeem niet meer opgeroepen worden.
- **void** update() : Triggert een update-operatie en zal hierbij de update() methode van alle actieve geregistreerde system oproepen.

De concrete implementatie van Engine, IoCEngine breidt de standaard Engine implementatie uit met een Inversion of Control (IoC) container. IoC is een design principe waarbij je als developer niet zelf meer op zoek gaat naar welke concrete implementaties voor afhankelijkheden tussen klassen je gebruikt (zoals die vaak als constructor argumenten worden doorgegeven), maar je eigenlijk de IoC container de verantwoordelijkheid geeft om de afhankelijkheden voor je klasse instantie in te vullen. Je kan dit principe zien als een soort verderzetting van het Factory pattern dat ervoor zorgt dat code leesbaarder en makkelijker onderhoudbaar wordt.

Een concreet voorbeeld: dankzij de IoCEngine kunnen we in de concrete EntitySystem implementaties makkelijk toegang krijgen tot andere Systems:

```

public class EnemySystem implements EntitySystem {

    @Inject
    private Engine engine;

    @Inject
    private PlayerSystem playerSystem;

    @Inject
    private BonusSystem bonusSystem;

    @Inject
    private EnemySpawnBehaviour spawnBehaviour;

    @Override
    public void update() {
        Collection<Entity> enemies = engine.getEntitiesWithComponents(EnemyComponent.class);

        //Do enemy logic using injected references
    }

}

```

Zo kan het ook handig zijn veel gebruikte objecten zoals configuratie objecten en de JavaFX grafische context in de IoC container op te nemen zodat deze ook kunnen geïnjecteerd worden in jullie EntitySystem implementaties. Om ons niet vast te zetten op het gebruik van IoC, hebben we hiervoor een aantal extra generieke methodes voorzien in Engine die ook via een alternatieve oplossing zouden kunnen geïmplementeerd worden:

- `registerContext(contextType, context)` : Voegt een object context toe van het type contextType aan de engine. Dit laat toe dat het opnieuw kan opgevraagd worden via de `getContext` methode of kan geïnjecteerd worden in geregistreerde systemen en zorgt er dus voor dat globale objecten (zoals configuratie of de GraphicsContext) makkelijk kunnen gedeeld worden.
- `unregister(Object instance)` : Verwijdert een context object uit de engine, deze zal vanaf nu niet meer beschikbaar zijn via `@Inject` of de `getContext` methode.
- `Set getContext(contextType)`: Geeft een collectie van context objecten terug voor het gegeven context type.

Bijvoorbeeld:

```

//Voeg het Graphics Config object toe als context aan Engine.
engine.registerContext(GraphicsConfig.class, config.getGraphics());

```

Hierdoor wordt het mogelijk dit object te injecteren in een System klasse via `@Inject`:

```

@Inject
private GraphicsConfig config;

```

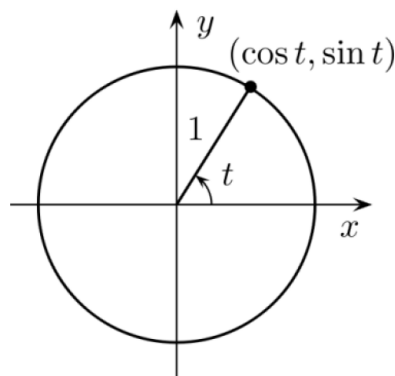
Dit is een heel handige aanpak om zo aan configuratie data te kunnen in om het even welk system, zonder dat je zelf continu referenties moet doorgeven.

4.3 Wiskundige tips

Game Development is een tak van de Computerwetenschappen waar wiskunde ook in de praktijk nog een belangrijke rol speelt. Gelukkig bestaan er heel wat tools en libraries die ons al enig denkwerk kunnen besparen (zie bijvoorbeeld de `Point2D` en `Rectangle2D` klassen van Java FX 8 die zeker van pas zullen komen voor collision detection). Toch zijn er ook voor dit game een aantal zaken die we nog zelf moeten kunnen berekenen.

4.3.1 Bewegingsvector bepalen

Zowel de speler als de vijanden zullen op een bepaald moment over een input hoek beschikken en een bepaalde snelheid waarmee ze in de richting van die hoek moeten bewegen. Dit kunnen we perfect mappen naar een goniometrische cirkel (zie Figuur 3) die eigenlijk een hoek vertaalt naar een vector met magnitude 1 in een orthogonaal stelsel.



Figuur 3: Mapping van een hoek naar een punt via de goniometrische cirkel

Concreet kan de bewegingsvector dus berekend worden als:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos \alpha \\ \sin \alpha \end{bmatrix}$$

Met α de toe te passen hoek in radialen. Het resultaat is een genormaliseerde vector, om ook rekening te houden met de snelheid dienen we dus deze factor nog toe te passen via een scalaire vermenigvuldiging:

$$movementVector = speed \cdot \begin{bmatrix} \cos \alpha \\ \sin \alpha \end{bmatrix}$$

4.3.2 Hoek tussen twee punten berekenen

Bij de implementatie van het spel, moeten we in twee gevallen de hoek tussen twee punten kunnen berekenen:

1. De kijkrichting van de speler bepalen door de hoek te berekenen tussen de positie van de speler en de positie van het vizier ("crosshair").
2. De bewegingsrichting van vijanden bepalen door de hoek te berekenen tussen de positie van een vijand en de positie van de speler (zodat we het effect krijgen dat de vijanden de speler volgen).

Om dit te doen zullen we eerst en vooral de twee punten in beschouwing transleren, zodat het éne punt op de oorsprong komt te liggen en het andere op de originele relatieve afstand (maar deze keer t.o.v. de oorsprong). Dit laat ons toe om een goniometrische cirkel vanuit de oorsprong te gebruiken om de hoek α te bepalen voor dit tweede punt P. We weten namelijk dat de richtingscoëfficiënt van de rechte vanuit de oorsprong, door het tweede punt gelijk is aan $\tan \alpha$, met andere woorden:

$$Py = \tan \alpha \cdot Px$$

Bovenstaande vergelijking kunnen we oplossen naar α :

$$\alpha = \tan^{-1} \frac{Py}{Px}$$

Het is echter wel zo dat deze oplossing enkel geldt wanneer $x > 0$. Wanneer x kleiner of gelijk is aan nul, dient een afgeleide waarde berekend te worden. Om programmeurs het leven makkelijker te maken is echter de `atan2` functie ontwikkeld, die x en y als aparte argumenten aanneemt en zelf zal rekening houden met in welk kwadrant het punt zich bevindt. Op deze manier kunnen we eenvoudig α berekenen als:

$$\alpha = \text{atan2}(Py, Px)$$

4.4 Resources

De opgave voorziet allerhande resource-files voor de verschillende spelconcepten: de achtergrond van de wereld, geluidbestanden voor de wapens en bonussen, sprites voor de diverse wapens, vijanden, explosies, etc...

Het is echter zo dat dit een samenraapsel is van bronnen die gevonden zijn via opengameart.org die hier puur voor educationele doeleinden gebruikt wordt. Sommige van de bronnen kunnen extra vereisten opleggen wanneer afgeleide werken gedistribueerd worden. Gebruik deze dus niet om een versie van jullie implementatie publiek te maken zonder dit verder na te gaan!

5 Indienen

- Het project wordt individueel gemaakt
- Schrijf een verslag bij je oplossing met daarin:
 - Uitleg bij de belangrijkste ontwerpbeslissingen
 - Uitleg bij de gebruikte design patterns
 - Uitleg bij de gekozen componenten en systemen
 - Uitleg bij eventuele extra functionaliteit
- Zip je volledige project samen met het verslag in een bestand naam_voornaam_project_gso.zip
- Gebruik een standaard zip formaat, geen rar of 7-zip
- Zet in je bronbestanden bovenaan in commentaar je naam
- Stuur je oplossing door middel van de dropbox op minerva door naar Bruno Volckaert, Thomas Dupont en Wannes Kerckhove ten laatste op zondag 17 december, 23u59