Documentatie Asteroids v1.0

Gemaakt door: Emiel de Brouwer & Robbert Goey

# Gebruiksaanwijzing Asteroids

Je speelt in een First Person View. Gebruik de WASD-toetsen op het toetsenbord om jezelf te bewegen: druk op “W” om vooruit te lopen, druk op “A” om links of “D” om rechts te keren en druk op “S” om achteruit te lopen.

# HUD

De HUD laat de volgende teksten zien:

* De tijd
  + We gebruiken de tijd op dezelfde manier als hoe we dat gedaan hebben met de Computer Graphics taak met behulp van SFML. De tijd loopt meteen zodra het spel is opgestart. De tijd wordt ook gebruikt als score. Hoe minder lang je erover hebt gedaan hoe beter je score is.
* Uitleg
  + De eerste uitleg in het spel is “hardcoded” erin gezet, dit in verband met het gebruik van \n. Overige uitleg/berichten kunnen via level.txt erin gezet worden als parameter van room. Let wel op: om de volgende hint/bericht te zien, moet er wel op enter gedrukt worden, anders blijft het hangen op de laatste hint/bericht.
* FPS
  + Voor debug mogelijkheden, kunnen we de Frames Per Second tonen, niet te verwarren met First Person Shooter ;)
* Huidige locatie van de speler
  + Voor debug mogelijkheden tonen we de locatie van de speler.

# Requirements

De volgende requirements zijn in de game geïmplementeerd:

* Ruimteschip vliegt door een 3D ruimte met asteroïden
* Asteroïden hebben elastische botsingen met elkaar en “muren”
* Asteroïden worden verwijderd zonder memory leak
* De camera drift soepel achter het ruimteschip (met behulp van behaviour)
* Een octree
* Broad phase controle op botsingen

(nog) niet geimplementeerd:

* Kapot schieten van asteroïden
* Bepalen welke triangle van het ruimteschip wordt geraakt
* Picking
* Narrow Phase Collision Detection

# Gebruikte patterns

## Visitor Pattern

We gebuiken het visitor pattern voor de renderer, de updater en de collisions. Dit zorgt ervoor dat alle render, update en collision operaties bij elkaar in een klasse zitten ipv verspreid over meerdere klasses. Ook een voordeel is mochten we een nieuwe visitor toe willen voegen dan hoeven hiervoor geen bestaande klasses worden aangepast.

We hebben ervoor gekozen om het objectmodel de traversal te laten doen in plaats van de visitor.

Het objectmodel zorgt ervoor dat kinderen ook gevisit worden, dus ook de accept methode van alle kinderen aanroepen. Ook de mesh, behaviour en collider van een object vallen hieronder.

### Rendervisitor

Alle draw methoden uit de Visitable klasses worden vervangen door de visit methodes van de RenderVisitor. Behaviour en Collider hoeven niet te worden gerenderd dus die visit methodes doen niets.

## Builder en factory

Voor de opbouw van de levels gebruiken wij het Builder en factory pattern. De builder leest een textfile uit (“level.txt”, standaard). Vervolgens gebruikt deze builder de factory om alle objecten aan te maken.

## Singleton

Voor de tijd gebruiken we de Singleton Pattern. Hiermee zorgen we ervoor dat de toegang tot de tijd altijd via één object te laten gaan. Hierdoor is de tijd uniek.

## Composite

We gebruiken het composite pattern voor onze gameobjecten. Dit zorgt ervoor dat we eenvoudig gameobjecten kunnen nesten en geen onderscheid hoeven te maken tussen containers en “normale” objecten.

# Ontwerp

TODO: updaten voor asteroids

Het ontwerp is vooral gewijzigd ten op zichte van de eerdere MicroGameEngine door het gebruik van bovenstaande design patterns. Verder hebben we een aantal wijzigingen/toevoegingen gedaan specifiek voor deze game. De Player klasse, wat erft van de camera omdat onze view First Person is. En de EndCube klasse, omdat onze game een keer moet stoppen. Onze scene graph is opgedeeld in Rooms, omdat verschillende ruimtes een requirement was en omdat het een hele handige opdeling is.

## Class Diagram

TODO: updaten voor asteroids



## Sequence Diagram

TODO: updaten voor asteroids



## State Diagram

TODO: updaten voor asteroids



# Specifieke oplossingen

## Camera movement

Om de camera op een juiste manier het ruimteschip te laten volgen hebben we de camera op een vaste afstand op de omtrek van een cirkel met als middelpunt het ruimteschip. De locatie van de camera is dan gebaseerd op de rotatie van het ruimteschip.

De positie van de camera kan berekend worden met de volgende formules:

float camerax = cameradist \* sin(yrot \* PI / 180) + xpos;

float cameraz = cameradist \* cos(yrot \* PI / 180) + zpos;

Waar yrot de rotatie om de y-as is van het ruimteschip. En xpos en zpos respectievelijk de x en z positie zijn van het ruimteschip.

Bron en veel uitgebreidere uitleg: <http://www.gamedev.net/page/resources/_/technical/game-programming/a-simple-third-person-camera-r1591>

## Collision Detection en Response

Zodra een asteroide botst met een andere asteroide of onze muur in de ruimte, draaien we de velocity (snelheid + richting) om. Dit leverde eerst problemen op dat objecten vast blijven zitten in elkaar, ze draaien constant hun velocity om. Om dit te voorkomen hebben we gekozen om bij te houden met welk object de laatste botsing was, om zo maar 1x op deze botsing te reageren. Nadeel hiervan is echter wel dat een botsing tussen meer dan 2 objecten tegelijk nog steeds problemen op kan leveren. Een alternatieve oplossing is om een duwtje te geven in de goede richting, maar omdat je hier voor een minimale snelheid introduceert hebben we voor onze oplossing gekozen.

# Problemen

(nog) niet geimplementeerd:

* Kapot schieten van asteroïden
* Bepalen welke triangle van het ruimteschip wordt geraakt
* Picking
* Narrow Phase Collision Detection

# Conclusie

# Persoonlijke reflectie

## Emiel

Ik vond deze opdracht moeilijker dan verwacht. Ik heb daardoor wel veel geleerd. Maar helaas was onze planning niet op deze moeilijkheid en tegenslagen van ziekte gebaseerd. Hierdoor is het helaas lang niet zover af als ik had gewild. De samenwerking ging wel prima, we konden goed taken verdelen zonder op elkaar te hoeven wachten. En ik ben moe en heb geen zin meer om een lange reflectie te schrijven. bladibla

## Robbert