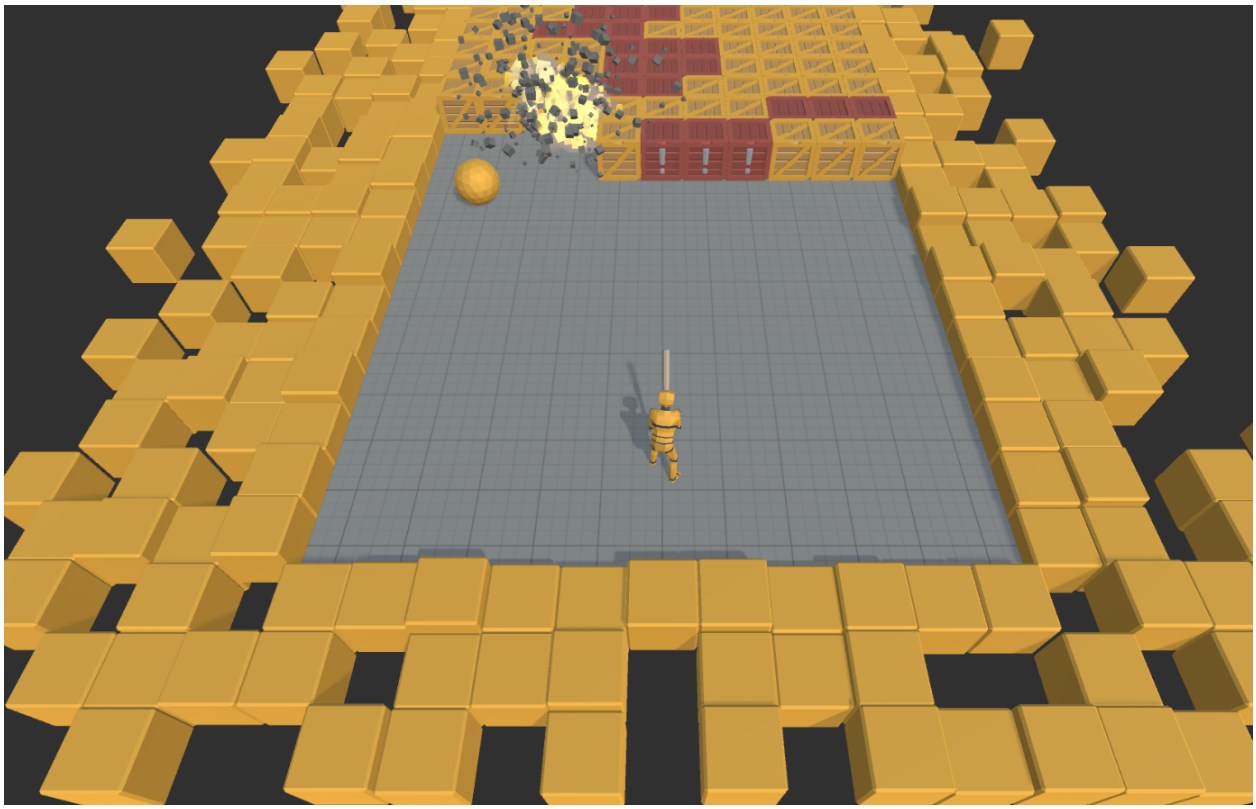
KERNMODULE GAME DEVELOPMENT 1 (RETRO GAME MET TWIST)
ALEX VAN DER LINDEN

SWORDOUT

BREAKOUT, MAAR DAN MET EEN ~~ZWAARD~~ KNUPPEL



Inleiding

‘Swordout’ is een twist op de klassieke game Breakout. In plaats van een paddle die horizontaal beweegt en waar de bal op stuitert zit er in dit spel een character waar je mee kunt lopen en met een knuppel de bal kan slaan.

In het project heb ik gebruik gemaakt van ***interfaces***, ***object pools***, ***event systems***, ***inheritance*** en een hoop ***raycasting***.

Object Pool (PoolManagement.cs)

Om zo min mogelijk tijdens de gameplay in te laden (waar het spel door zou kunnen haperen) gebeurt dit aan het begin van het spel via een object pool. Via de singleton `PoolManagement.cs` is het makkelijk om *GameObjects* via een naam (string) in te spawnen en weer te 'verwijderen'. Bijvoorbeeld:

```
PoolManagement.Instance.Spawn(Block.REGULAR_BLOCK, Vector3.zero);
```

Initialisatie

Allereerst maak ik een `List<PoolObject>` met alle objecten die ik aan wil maken:

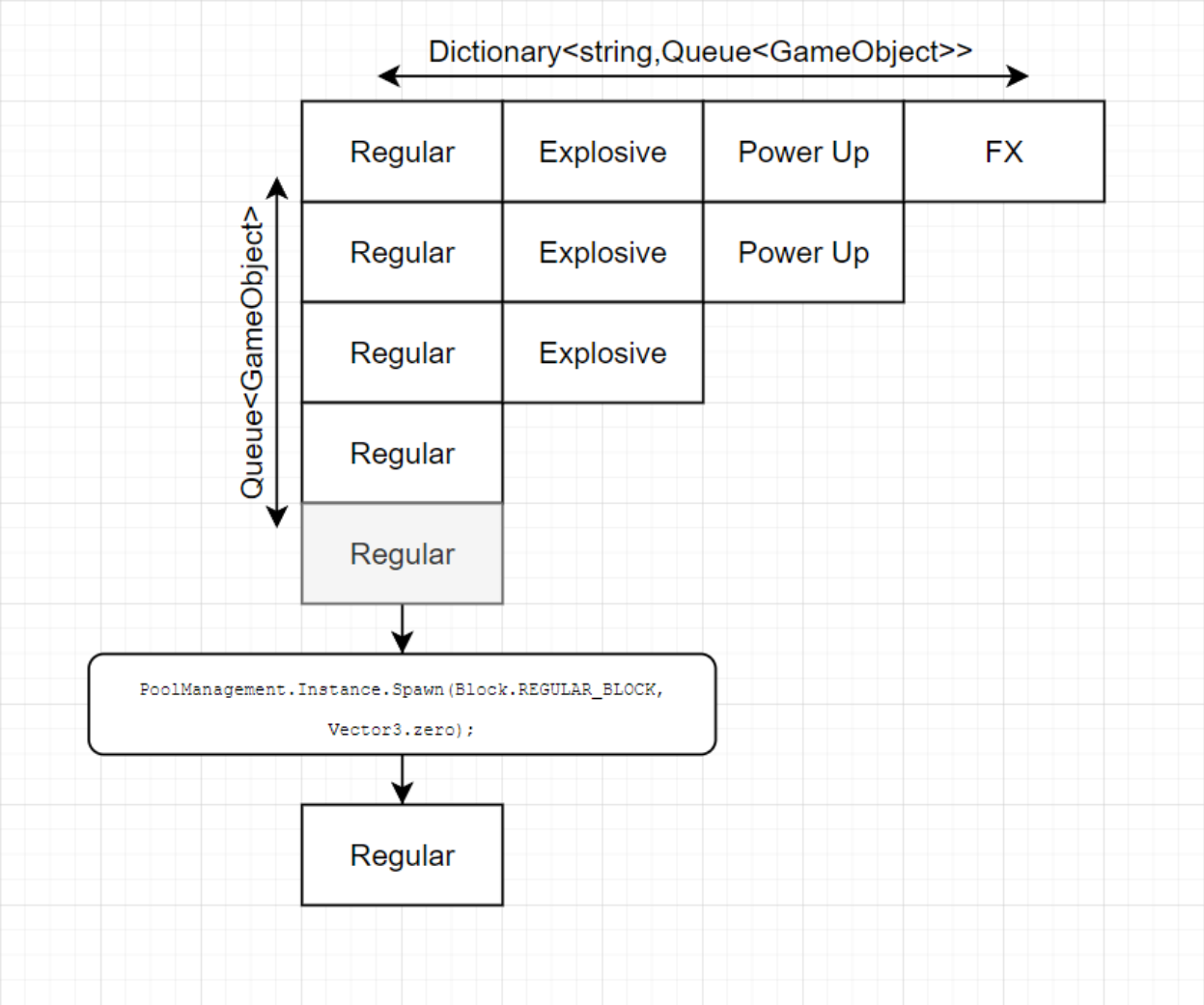
- name:String
- gameObject:GameObject
- amount:int

Vervolgens maak ik een `Dictionary<string, Queue<GameObject>>` aan wat de pool is. De `string` is de naam van het object die ik wil opzoeken en de `Queue<GameObject>` is de queue waar ik het eerstvolgende GameObject met de gegeven naam uit kan halen via het *first in, first out* principe. Alle geïnstantieerde objecten worden op `SetActive(false)` gezet zodat ze onzichtbaar zijn en de code niet draait.

Spawn

Via Spawn(string name, Vector3 position) kun je een object laten spawnen. De functie kijkt eerst of de naam van het object bestaat, daarna of er met die naam nog objecten over zijn en zal vervolgens het object uit de queue halen en het object initialiseren.

Hier een voorbeeld met 5 Regular, 3 Explosive, 2 Power Up blokjes met 1 FX:



Na deze call blijven er nog 4 Regular blokjes over in de queue.

Remove

Via Remove(GameObject gameObject) kun je een object laten ‘verwijderen’. De functie zoekt voor een queue met de naam van het object en zal vervolgens het object in de lijst opzoeken om deze te deactiveren.

Events (EventManagement.cs)

Door verschillende classes te laten subscriben op een event is het makkelijker om uit te breiden op iets wat een object doet. Als een blokje ontploft wil je bijvoorbeeld dat het blokje gedeactiveerd wordt in de object pool, maar ook dat er ergens een score wordt toegevoegd in de UI, dat een ander blokje reageert of dat er een nieuw object spawnt (zoals een effect).

Als een blokje geen levens meer heeft wordt er een event aangeroepen:

```
EventManagement.HitBlockEvent(BaseBlock block, int row, int column);
```

De `GameManagement.cs` class heeft zich subscribed op dat event via:

```
EventManagement.OnHitBlockEvent += OnRemoveBlock;
```

In `OnRemoveBlock` wordt er vervolgens gekeken en gezocht naar het blokje en afgehandeld.

Interfaces

Om hetzelfde gedrag voor meerdere classes te implementeren maak ik gebruik van interfaces. `IHittable` en `IPoolable`.

Inheritance

Sommige classes zoals `BaseView.cs` hebben `abstract` en `virtual` basisfunctionaliteit waar `MainView.cs` en `EndView.cs` gebruik van maakt. Zo hoef ik geen tot minder herhalende code te schrijven. Dit geldt voor meerdere classes (zie UML).

Verbeterpunten

- Code:
    - De inheritance zou verbeterd kunnen worden. Bijvoorbeeld een een Base class die boven de huidige Base classes zoals BaseBlock zit om zo nog ‘generiekere’ code te schrijven.
- Gameplay:
    - Meer dan één level met verschillende patronen van de blokjes.
    - Verschillende power ups en blokjes waardoor het meer Arkanoid wordt.
    - Geluid.

Conclusie

Ten opzichte van mijn eerst ingeleverde opdracht heb ik gebruik gemaakt van meerdere design patterns en een betere basis gelegd om het spel verder uit te kunnen breiden.