Breakin

Casper van Battum | 3031249

Kernmodule 1 | 19-09-2019

Spelconcept

Breakin is het tegenovergestelde van Breakout/Arkanoid. In plaats van dat de blokken boven je zitten, zitten ze nu in een ring in het midden en de speler moet proberen alle blokken te verwijderen. Door de cirkelvorm is het een stuk belangrijker om actief de bal in de goede richting te sturen dan bij de klassieke versie van Breakout. Ook heb je, zoals in Arkanoid, level progressie.

Codestructuur

Ik ben dit project begonnen door eerst de main gameplay te schrijven. Toen we verder kwamen in het vak ben ik steeds meer design patterns toe gaan passen, zoals de ObjectPool, de Singleton en uiteindelijk ook de Finite State Machine.

Main loop

In het begin bestond de code voornamelijk uit MonoBehaviours. De combinatie van hun Update functies bepaalde hoe het spel werkt. Inmiddels wordt de update functie bestuurd door de Finite State Machine. De belangrijke classes hiervan zijn Block (+subclasses), Ball, BatMovement en Spawner. De Ball gebruikt het physics system in Unity om zich te bewegen, en zijn snelheid wordt slechts op twee momenten ingesteld (Lock en Unlock). Deze functies worden getriggerd als de gebruiker op de muis klikt en als de bal het scherm uitgaat/aan een level begint. Dit staat geheel los van de state machine, voornamelijk omdat dit lastig naderhand in states te verwerken was.
BatMovement (en de Bat class) zorgen voor de beweging van het batje. Iedere Update wordt de muisinput opgehaald en omgezet in een positie voor het object. Afgezien van de Update functie die bestuurt wordt door de state machine staat deze class geheel los.
De Spawner is één van de belangrijkste onderdelen van het spel. Deze is verantwoordelijk voor het aanmaken van nieuwe ringen met blokken, maar ook voor het in de gaten houden van deze ringen zodat win- of verliescondities geregistreerd worden. Ook is de Spawner verantwoordelijk voor het doorgeven van de levelinstellingen aan andere classes die deze nodig hebben (de Rings en de Blocks) Het aanmaken van ringen wordt gedaan via de Ring class. Deze erft expres niet van MonoBehaviour zodat je er makkelijk nieuwe instanties van aan kan maken. In de constructor van de Ring worden nieuwe blokken direct aangemaakt en op de juiste plek gezet.
Een Block is enkel verantwoordelijk voor het registreren dat hij geraakt is in de OnCollisionEnter2D functie. Om makkelijk verschillende typen blokken te kunnen maken is deze class abstract, en definieert hij de abstracte functie OnHit die aangeroepen wordt als de bal het blok raakt. Iedere subclass moet definiëren wat er dan gebeurt. Als het blok breekt moet OnHit de functie OnBreak aanroepen, die standaard is geïmplementeerd in de base class. Deze roept vervolgens het event aan dat dit blok kapot gegaan is. De ring kan hier op reageren door te checken of alle blokken kapot zijn. Als dit zo is waarschuwt de Ring op zijn beurt de Spawner weer en dan gaat deze checken of het einde van het level behaald is. Al deze gebeurtenissen worden gecommuniceerd door middel van events.

Object pool

In plaats van Blocks te spawnen voor iedere ring heb ik gebruik gemaakt van een object pool. Hoewel ik deze functie niet gebruikt heb is er in principe ondersteuning voor het hebben van meerdere (types) blokken. Aan een ring zouden meerdere prefabs toegevoegd kunnen worden, en de pool moet hier ondersteuning voor bieden. Ik heb dit opgelost door de pool te implementeren met een dictionary, waarvan de key de prefab is voor een lijst van dat soort blokken. Het voordeel van indexeren op een specifieke prefab tegenover het Type in het algemeen is dat je zo meerdere blokken van hetzelfde type met verschillende instellingen kan gebruiken. De items in de pool moeten een IPoolable zijn. Deze class definieert de property IsActive, die gebruikt wordt om aan te geven dat het object in gebruik is en niet meer ergens anders gebruikt mag worden. Dit staat toe dat iedere partij (de pool, de gebruiker en het object zelf) ten alle tijden het object kan loslaten nadat hete eenmaal geclaimd is. Hoewel het een beetje minder veilig is, levert dit veel flexibiliteit op. Zo kan de pool op deze manier alle objecten in één keer terugclaimen bij een level switch, of kan een ring de blokken vasthouden totdat ze allemaal kapot zijn.
De pool is dynamisch, oftewel hij kan groeien mocht dit nodig zijn. Bovendien is hij geheel generic, wat hem goed hebruikbaar maakt voor andere projecten waar eenzelfde scenario voorkomt.

GameManager

De GameManager en de EventManager ijn het hart van het spel. De GameManager is de enige class die, sinds het gebruik van de FSM, een Unity Update functie heeft. Bovendien bevat deze de referentie naar de state machine die het spel bestuurt en naar de LevelManager die de levels bijhoudt.

StateMachine

De state machine is in principe een algemene state machine. Het zijn de states zelf die bepalen wat de FSM doet. In de constructor wordt een beginstate meegegeven. Deze state kan vervolgen door middel van een event dat iedere state moet hebben (RequestTransition) doorgeven dat er van state gewisseld moet worden. Er moet een nieuwe state meegegeven worden als deze event aangeroepen wordt. Hierdoor kan iedere state zelf bepalen wanneer er gewisseld moet worden en wat de volgende state wordt.

EventManager

Om de states te laten communiceren met de rest van het spel wordt een EventManager gebruikt. Deze statische class bevat statische delegate fields die van overal aangeroepen en waar vanaf overal op geabonneerd kan worden. De bedoeling was om de events in deze class zo generiek mogelijk te houden, alleen om de code die ik al had aan te sluiten op de state machine heb ik een aantal

specifieke events aan moeten maken. De belangrijkste events zijn activate, gameLoop en reset. Deze zijn in feite het equivalent aan Start, Update en OnDestroy in Unity. Dit zorgt ervoor dat het level gestart wordt, draait en opgeruimd wordt. Bovendien moet een level geladen kunnen worden, waarvoor de twee events broadcastLevel en levelStart gebruikt kunnen worden. De eerste geeft de LevelData mee zodat de entiteiten die dit nodig hebben zichzelf goed kunnen instellen. De overige events zijn bijvoorbeeld voor de UI.

UI

De UI van dit spel is heel simpel. Er zijn slechts twee elementen: de scoreteller en een message box. De scoreteller laat de score zien die wordt verstuurd door een intern event in de ScoreManager. De message box laat allerlei berichten zien aan de speler en reageert op de events ShowMessage en HideMessage van de EventManager.

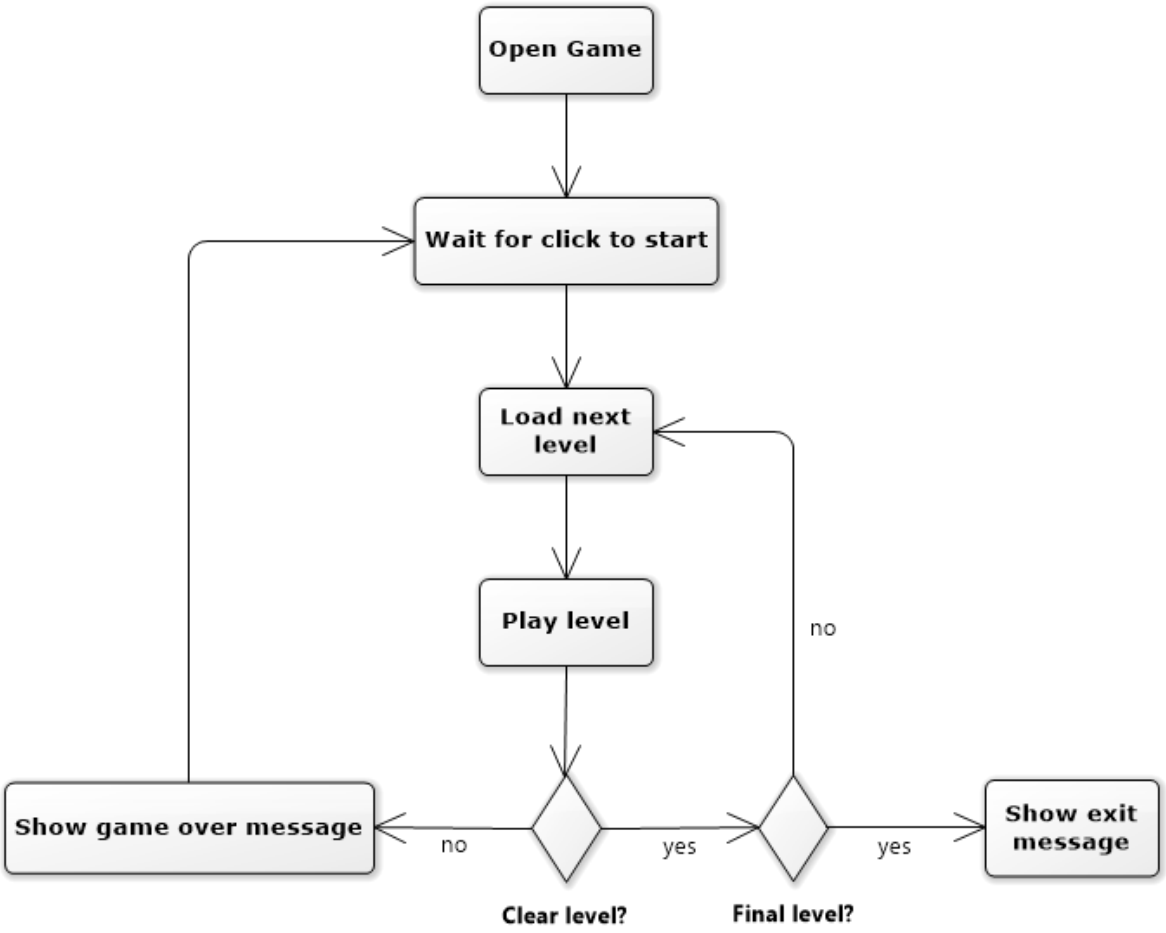
Sound

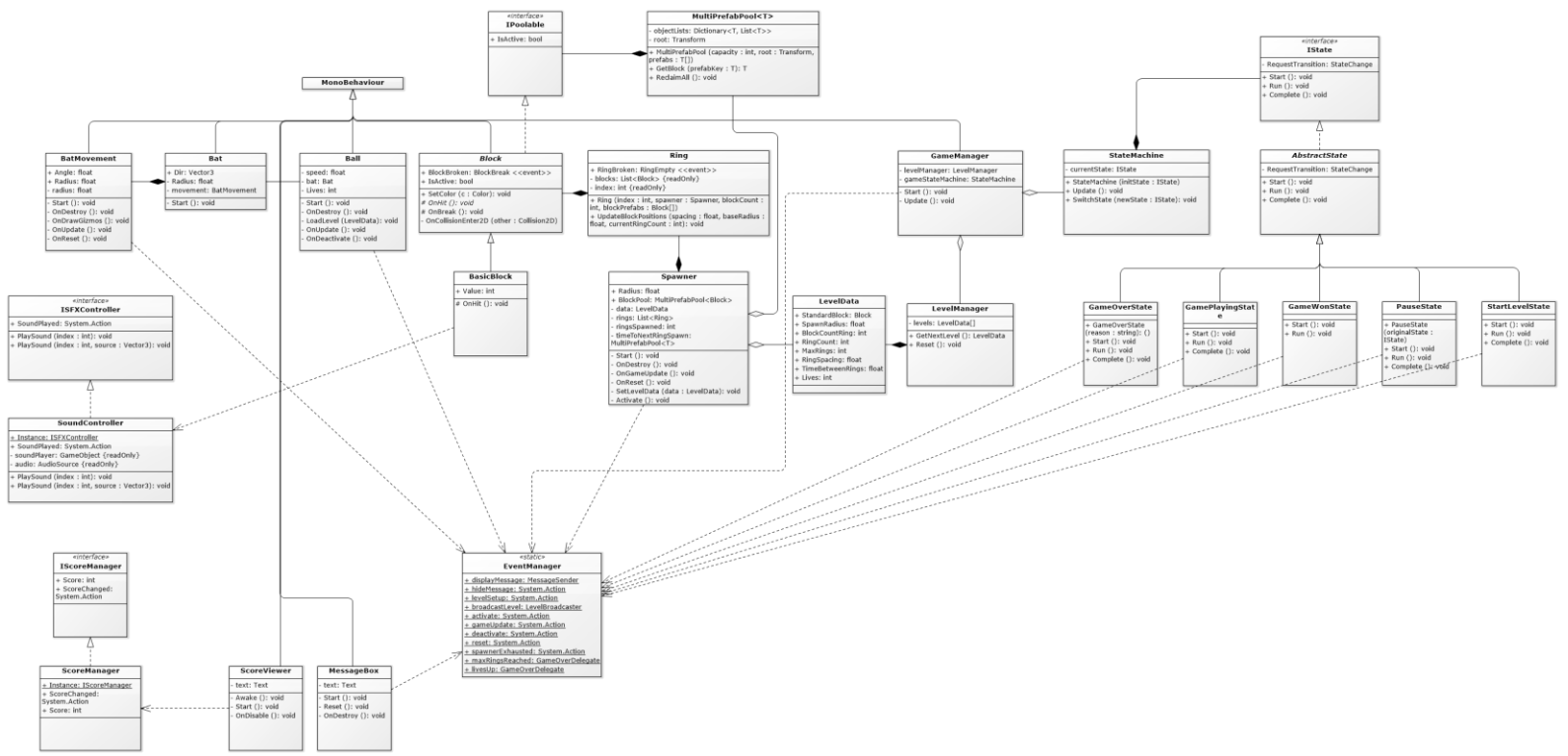
Het geluid wordt geregeld door een losstaande SoundManager. Deze singleton heeft een eigen PlaySound functie die van overal aangeroepen kan worden. De geluiden worden geladen uit een map en kunnen op index aangeroepen worden. De soundmanager maakt een eigen gameobject aan die geluiden af kan spelen. Deze class is volledig losstaand zodat hij makkelijk hergebruikt kan worden.

Reflectie

Ik ben uiteindelijk tevreden over het eindresultaat. De code werkt foutloos en is uit relatief makkelijk uit te breiden. Hoewel niet alles wat ik in het begin bedacht had uiteindelijk gemaakt is heb ik er in veel gevallen wel rekening mee gehouden, en ik ben blij met de hoeveelheid features die er wel in zit.
Waar ik niet trots op ben is de implementatie van de Finite State Machine en de EventManager. Dit vind ik erg rommelig geworden. Hoewel het goed uitgebreid kan worden vind ik het erg onoverzichtelijk. Je kan niet makkelijk de control flow volgen en de EventManager is niet hebruikbaar. De reden dat dit gebeurt is, is dat ik het aan moest sluiten op de code die ik al had. Om het niet nog een keer te laten gebeuren moet ik vanaf het begin programmeren vanuit een state machine.

Activity diagram





https://github.com/Creator13/HKU-Kernmodule1/blob/master/Files/UML%20v3%20(final).png