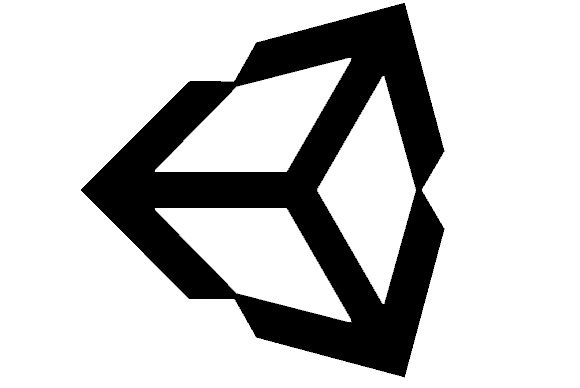
Einddocumentatie GTO7





**Naam:** Frank Severijns  
**StudentNr:** 2383764  
**Klas:** GD7

Datum: 14/01/2016

Table of Contents

[AI 3](#_Toc441490914)

[API 5](#_Toc441490915)

[Multiplatform 6](#_Toc441490916)

[Efficient Level Creation 7](#_Toc441490917)

[Persistence 9](#_Toc441490918)

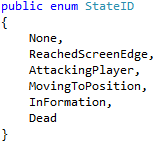
[Localization 10](#_Toc441490919)

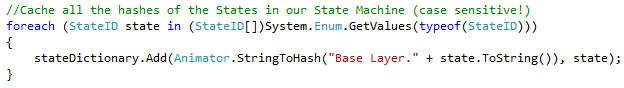
[Audio Management 11](#_Toc441490920)

[Bronnen 12](#_Toc441490921)

# AI

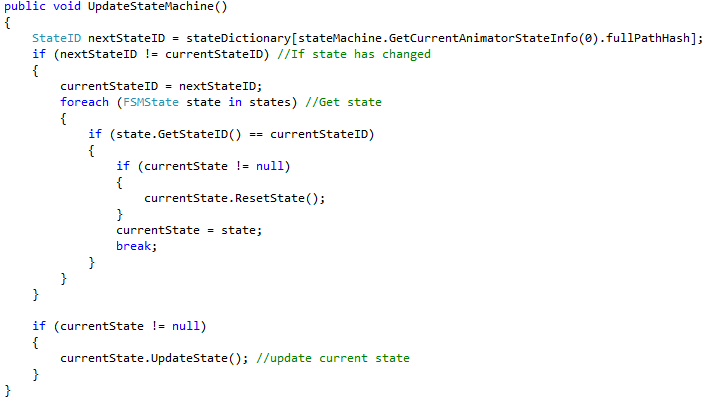
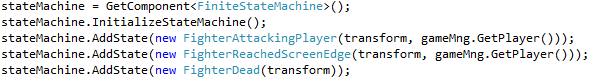
Om de FSM te maken heb ik gebruik gemaakt van het Advanced FSM voorbeeld uit hoofdstuk 2 van GAI7, in combinatie met een tutorial waarin Mecanim gebruikt wordt als Finite State Machine (<http://darkgenesis.zenithmoon.com/mastering-unity-2d-game-developmentai-and-state-machines/>).

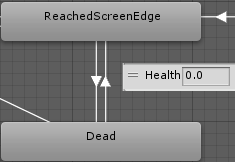
Om te beginnen heb ik een FiniteStateMachine script. In dit script wordt bijgehouden welke state op dit moment actief is. In dit script staat ook een enum waar alle gebruikte states in staan. Het is belangrijk dat deze naamgeving ook precies zo terugkomt in de Mecanim states. Deze StateIDs worden aan een Mecanim state verbonden worden middels een Dictionary.

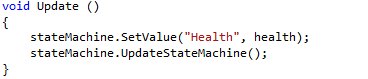
Deze Dictionary heeft een key (int) en een value (StateID). De key van een dictionary item is de hash van de mecanim state naam, de value is de bijbehorende StateID.

Vervolgens heb ik een abstracte klasse FSMState gemaakt. Deze klasse wordt overerft door alle state scripts. Hierin staan een aantal methodes gedefineerd, waaronder de GetStateID() methode, waarin een StateID teruggegeven wordt (in bijvoorbeeld een DeadState script word dan StateID.Dead teruggegeven).

De State scripts worden door iedere agent toegevoegd aan een List in hun FiniteStateMachine component.

In iedere update wordt bijgehouden welke state in Mecanim actief is. Door de actieve state uit te lezen en de hash ervan te gebruiken om de bijbehorende StateID uit de eerder genoemde Dictionary te halen, kan bepaald worden welke State script op dat moment gebruikt moet worden.

Om in mecanim het transitioneren van state naar state bij te houden maak ik gebruik van de mecanim parameters en transition condities. Door bijvoorbeeld de waarde van Health constant up-to-date te houden zorgt mecanim er zelf voor dat wanneer health onder 0 komt, de actieve state transitioneert naar Dead (zolang er in mecanim een transition tussen de states bestaat).



Dit helpt om de states en de transities te visualiseren. Ook zorgt dit ervoor dat ik niet in code hoef te bepalen wanneer ik van state naar state moet transitioneren, dit wordt allemaal automatisch gedaan in mecanim, zolang de parameters en condities goed ingesteld zijn en bijhouden worden.

# API

Ik heb gekozen om de GameJolt API te gebruiken (<http://gamejolt.com/games/unity-api/15887>). De GameJolt API stelt me in staat om scores op te slaan en trophies te maken die de speler kan unlocken.

Om de API te gebruiken heb ik een Unity plugin geimporteerd met daarin een prefab die ik in de menu scene van mijn spel heb geplaatst. De plugin is makkelijk te gebruiken en heeft ook een ingebouwd UI voor o.a. het inloggen van de gebruiker en het bekijken van trophies.

Om de API aan te roepen heb ik een singleton klasse (APIManager) geschreven waarin ik een aantal publieke methodes beschikbaar maak om zo makkelijk vanuit andere scripts de benodigde functionaliteit van de API te gebruiken.   
Om bijvoorbeeld een trophy te unlocken moet er eerst gekeken worden of de trophy niet al unlocked is (anders zal de pop-up van de trophy iedere keer opnieuw tevoorschijn komen), en kan ik daarna pas de trophy daadwerkelijk unlocken. Door dit een enkele keer in mijn APIManager klasse als publieke methode UnlockTrophy(int trophyId) te implementeren hoef ik vervolgens in andere klassen alleen maar APIManager.instance.UnlockTrophy(id) aan te roepen.

De APIManager beheert naast de trophies ook het opslaan en inladen van scores in de DataStore en het inloggen en uitloggen.

Het inlogscherm, trophy scherm en de pop-up die toont dat er een nieuwe trophy unlocked is hebben allemaal een zelfgemaakte UI. 

# Multiplatform

Het spel is compatible met Android. Als het spel op Android gespeelt wordt kan de speler bewegen door op het speelveld te tappen. Het schip volgt de positie van de speler. Terwijl de vinger op het scherm zit schiet de speler ook projectielen af.

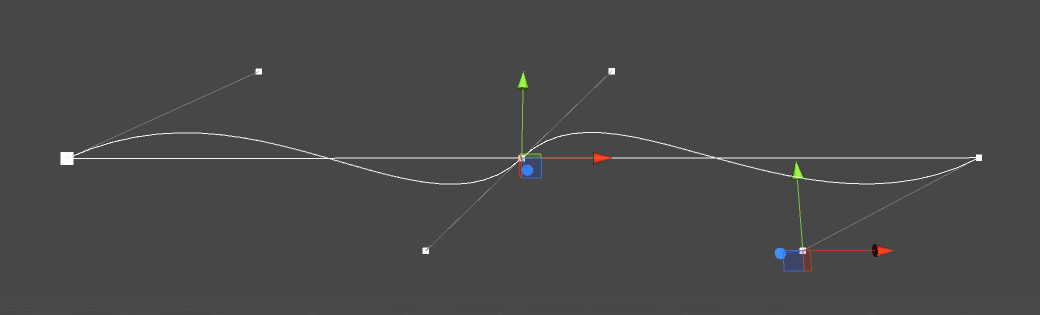
Alle UI componenten zijn gemaakt met de standaard Unity Canvas objecten, en zijn uit zichzelf al compatible met Android.

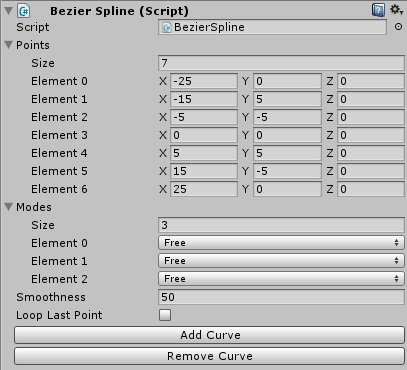
De standaard GameJolt UI is niet compatible met Android, daarom heb ik er een eigen UI voor gemaakt met Unity Canvas objecten.

# Efficient Level Creation

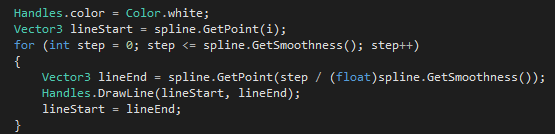
Een van de vijanden in het spel is een “drone”, deze vijand doet niets anders dan in grote groepen in een bepaald patroon over het speelveld bewegen. Om deze patronen te maken heb ik een editor script gebruikt dat ik eerder gebruikt heb voor een ander spel. (<http://catlikecoding.com/unity/tutorials/curves-and-splines/>)

Het editor script zorgt ervoor dat ik vanuit de inspector en in de Unity editor bezier splines kan maken met versleepbare punten. Deze bezier splines kan ik gebruiken als pad die de drone vijanden volgen door telkens naar volgende punt in de spline te bewegen.



De bezier spline zelf wordt gegenereerd in een MonoBehaviour script. In dit script staan SerializeField lists met Vector3 “Points” en “Modes” enums. De Points geven aan waar de sleepbare punten in de spline staan. De Modes geven aan hoe een curve overgaat naar de volgende.

Tussen de points wordt een curve genereerd. De spline kan dan getekend worden door de Handles.DrawLine() methode aan te roepen en een start- en eindpunt in de spline mee te geven als parameter.



Om het instantieren van objecten zoals projectielen efficienter te maken heb ik een instance pooling systeem gemaakt. De instance pools kunnen een dynamische grootte hebben, wat betekent dat er nieuwe instanties bij gemaakt worden als er te weinig in de pool zitten, maar kunnen ook een vast aantal behouden.

Om dit systeem te bouwen heb ik twee scripts gemaakt: ObjectPooler en PooledObjectBehaviour.   
De instances hebben allemaal een PooledObjectBehaviour script (of een script die van dit script overerft).   
In het PooledObjectBehaviour script staan methodes om het gameobject van de instance in een parent te zetten en om het object te disablen.   
In de ObjectPooler wordt een lijst van de instances gemaakt en bijgehouden. Als ik vanuit een ander script een beschikbare instance wil ophalen om te gebruiken heeft de ObjectPooler een GetNextInstance() en een GetNextDynamicInstance() methode met een GameObject returnwaarde.   
De GetNextDynamicInstance() methode instantieert een nieuwe instance als er geen beschikbare instances in de pool zijn.   
De GetNextInstance() methode heeft een boolean parameter, als deze boolean true is zal de methode altijd een GameObject uit de pool returnen, ook al is deze in gebruik. Staat de boolean parameter op false, dan returnt de methode null.  
De GameObject in de returnwaarde kan dan in een ander script (bijvoorbeeld de ProjectileController) geactiveert worden zodat het lijkt alsof er een nieuw projectiel gespawnt is.

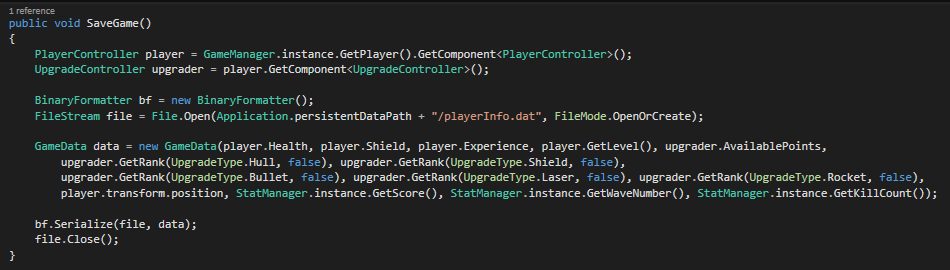
# Persistence

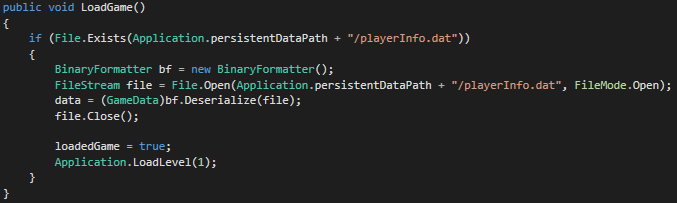
Voor data persistentie maak ik gebruik van GameJolt en de PlayerPrefs om drie verschillende scores op te slaan, namelijk de high score, hoogste “killcount” en de hoogste “wave”.

Om scores op te slaan heb ik een StatManager singleton script geschreven waarin alle scores bijgehouden worden. In dit script wordt ook gekeken of er trophies zijn die unlocked kunnen worden, die vervolgens middels de APIManager unlocked worden.

Tijdens het spelen krijgt de speler een betere highscore door bijvoorbeeld vijanden kapot te maken of experience punten te verzamelen. Door vijanden kapot te maken verhoogt ook de “killcount”. Vijanden komen in “waves”, het nummer van de wave wordt weergeven om te laten zien hoe ver de speler gekomen is. Als de speler een nieuwe highscore haalt, meer vijanden kapot maakt of een latere wave bereikt, moet dit natuurlijk opgeslagen worden.  
Als de speler zich heeft ingelogt op GameJolt wordt gebruik gemaakt van de DataStore functionaliteit om de scores online op te slaan. Tevens worden de scores opgeslagen in de PlayerPrefs, mocht de speler geen internet verbinding hebben of simpelweg niet ingelogt zijn op GameJolt. De volgende keer dat de speler inlogt op GameJolt in het spel, zal de StatManager controleren of de scores in de PlayerPrefs gelijk zijn aan die op GameJolt en vice versa. Zo niet, dan zullen de lagere scores overschreven worden met de hogere scores.

Tijdens het spelen kan de speler progressie opslaan. De data van de speler (health, experience, positie, level, huidige wave, etc.) wordt in een GameData object gezet. Dit object is serializable zodat deze weggeschreven kan worden naar een bestand door middel van een BinaryFormatter en de File.Open() functie.

In het hoofdmenu en in het pauzemenu kan de speler het laatst opgeslagen spel laden. Voordat het spel start worden de gegevens uit het bestand uitgelezen en de relevantie variabelen naar de waardes in het bestand teruggezet. 



# Localization

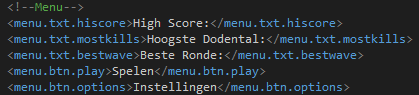
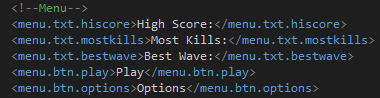
Een van de vrije onderwerpen die ik gekozen heb is localization. Om dit te realiseren heb ik een systeem geimplementeerd waarmee de gebruiker een taal kan kiezen. Alle teksten in het spel worden dan verandert naar de kozen taal.

De uitwerking van de localization is gebaseerd op de techniek die gebruikt wordt in een spel waar ik samen met Tom Vaessen voor het bedrijf Excamedia aan heb gewerkt.

Voor iedere taal waar de speler uit kan kiezen bestaat een .xml bestand met een bepaalde naamgeving (nl\_Text.xml, en\_Text.xml, etc.). In een LanguageController script houdt ik bij welke taal de speler heeft gekozen door de taal prefix in een variabel op te slaan (zoals “nl” of “en”). Met deze prefix kan het juiste .xml bestand uitgelezen worden.

Alle Tekst componenten die ik wil vertalen hebben een script component waarin een “key” string opgeslagen wordt. Deze key wordt gebruikt om de juiste tekst op te halen uit het .xml bestand.





# Audio Management

Om audio af te spelen heb ik een singleton klasse gemaakt waarin ik het afspelen van alle audio afhandel. Hierbij wordt rekening gehouden met het door de speler ingestelde volume.

Als ik bijvoorbeeld een projectiel afschiet wordt de PlaySoundWithRandomPitch functie in de AudioManager aangeroepen. Deze functie vraagt om een AudioClip parameter. Vervolgens krijgt de AudioClip een willekeurige pitch (om variatie in geluid te creëren).

De AudioSource waaruit een AudioClip wordt afgespeelt bevindt zich op een lege GameObject in een instance pool. De lege GameObjecten in deze pool hebben naast een AudioSource ook een script component die ik gemaakt heb, genaamt TimeToLive. Dit script zorgt ervoor dat een gameobject na een bepaalde tijd gedeactiveerd of vernietigd word. Deze tijd wordt ingesteld op de lengte van de audioclip.  
Bij het afspelen van een clip wordt een object uit de pool geactiveerd. Dan worden de clip, het volume en de pitch erop ingesteld en de AudioSource.Play() functie aangeroepen.

Voor geluiden waar geen random pitch bij nodig is bestaat ook de PlaySound() functie, die in alle opzichten hetzelfde doet, maar de pitch met rust laat.

# Bronnen

|  |  |
| --- | --- |
| **Onderwerp** | **Bronnen** |
| AI | Mecanim FSM:  <http://darkgenesis.zenithmoon.com/mastering-unity-2d-game-developmentai-and-state-machines/>  GAI7 FSM voorbeeld: <https://portal.fhict.nl/GDenT/SPECMINOR/Lesmateriaal/2_FSM_Tank_Assets.rar> |
| API | GameJolt API tutorial: <http://loicteixeira.github.io/gj-unity-api/tutorial.html> |
| Persistence | GameJolt DataStores:  <http://loicteixeira.github.io/gj-unity-api/tutorial.html>  Save/Load uit file:  <https://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/persistence-data-saving-loading> |
| Multiplatform | -- |
| Efficient Level Creation | Splines editor script tutorial: <http://catlikecoding.com/unity/tutorials/curves-and-splines/> |