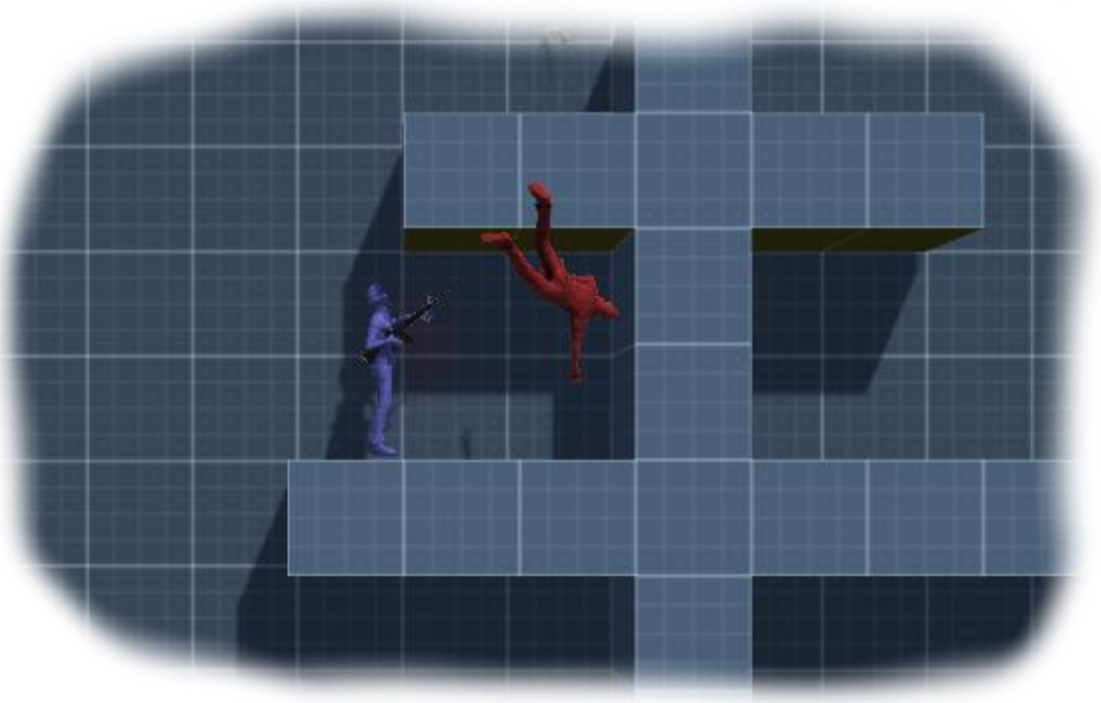

Sidescrolling 2.5D Shooter

Viking Crew Development

1 Introduction

This system aims to be a base project for anyone who wants to create a 2.5D gun'n run game. The scripts follow SOLID-principles as closely as possible in order to make it easily extendible. For example, where possible UnityEvents are used to communicate between scripts so that the scripts do not need to reference each other. If you have suggestions for further improvements in this regard let us know!

1.1 Support

There is a **thread for support** questions at: http://forum.unity3d.com/threads/sidescrolling-2-5d-shooter-support.384397/

If you prefer **support through e-mail** the address is: mike@vikingcrew.net

There is also a thread for **ideas, suggestions and feature requests** at http://forum.unity3d.com/threads/sidescrolling-2-5d-shooter-suggestions.384398/

All ideas are most welcome, regardless of whether you own the package or not! We would also love to hear about your projects. If you have a plan for something you want to make, please do not hesitate to let us know. We can probably give valuable advice on how to get the most out of the package.

Note that you are currently reading the first version of this manual. Any and all feedback on it would be most welcome!

1.2 2D or 3D physics?

In this package we have implemented 2.5D in two versions: One uses 2D physics for all game logic related physics (characters, bullets etc.) and uses 3D physics only for visual effects (particles, ragdolls etc.). The other one uses 3D physics for everything.

Which one you choose to use is up to you and your project requirements. 2D for logic is better performance wise. It might also be easier to separate the logic and the visual effects. In a future update however, we plan to integrate a voxel engine for destructible environments and there is no real high performance 2D equivalent to the 3D mesh collider which makes integrating a voxel engine using marching cubes very hard in 2D. If this is of no interest to you then we recommend you go with 2D physics.

1.3 Multiple (additive) scenes

You do not have to use multiple scenes, the two scenes *2D demo scene* and *3D demo scene* are complete scenes ready to be played, using 2D and 3D physics respectively. In case you prefer to work with additive scenes instead these are included in the scene subfolder *Additive scenes*. Usage of these scenes is described below.

The basic level is a simple level built to work with both 2D and 3D logic. To get a scene playing load the scenes basic level and **one of** *2D logic* or *3D logic* scenes.

Additive scene management is a new feature as of Unity version 5.1 so if you need to brush up on what it means read more here: http://docs.unity3d.com/Manual/MultiSceneEditing.html

2 Characters

The character is controlled through one script, *CharacterController2D.cs.* The ai (*AIControls.cs*) or the player (*PlayerControls.cs*) both call methods in that one script to interact with the world or their equipment.

Character controls are entirely physics based and thus do not use root motion at all.

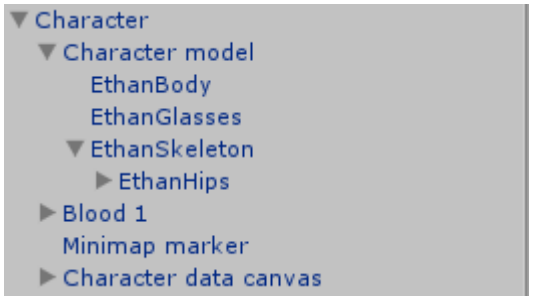

Figure 1 The important child objects of the character prefab

In Figure 1 the important child objects of the character prefab are displayed. Their main responsibilities are:

- *Character* - Parent object, contains most of the logic scripts
- *Character model* - Display the character
- *EthanSkeleton* - Contains the ragdoll and the skeleton hierarchy
- *Blood* 1 - Particle system for splashing blood when getting hit
- *Minimap marker* - A sprite renderer that displays the character on the minimap
- *Character data canvas* - A world space canvas that displays meters for health and reload progress

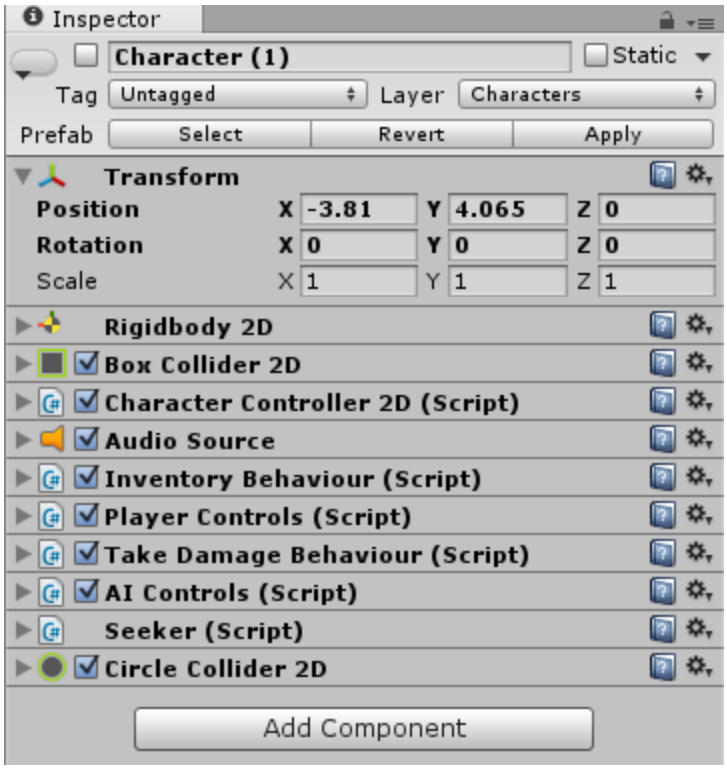

Figure 2 The components of the character parent object. Some of their responsibilities are described in further detail below.

2.1 Creating different looking characters

There is a prefab for characters in the prefab folder called *Character*. If you want to change the character model, then you should copy this prefab and change the child object *Character model* to look the way you want. Note that some manual work will be needed, you will need to set the variables of the script *IK Control* to the hand and head of your new character skeleton in order to have the character grab the weapon properly. You will also need to create a new ragdoll for the character model you are using. How to do that is further described here: http://docs.unity3d.com/Manual/wizard-RagdollWizard.html

2.2 Inventory

The *InventoryBehaviour.cs* script handles all things regarding inventory for the player. Note that it also is responsible for handling some interactions with the current weapon such as reloading.

2.3 AI

The *AIControls* class represents a pretty simple minded state machine based AI. It does two things, it thinks in Think() and determines what actions to take and it acts in Act() where it send signals to CharacterController2D.cs to make the character move the way it planned when thinking.

As thinking is a costly operation which may require pathfinding lots of ray casting and such Think() is put in a coroutine and only runs a few times per second. For more information about pathfinding see chapter 5.

If you want the AI to tell you what it is thinking as debug messages in the console, then set the *Verbose Debug* variable to *true* in the inspector.

2.4 Who's behind the wheel?

The player can be given control of a character by the *TeamBehaviour.cs* script. Currently this happens when the controlled character dies. You could edit the script to allow for changing characters through some other measure if you like.

2.5 What does *Seeker* do?

Seeker is the component that finds a path in the pathfinding graph that should span the world. Read more about it in chapter 5.

3 *Guns*

You want more guns, right? A weapon needs to things: A prefab and a data scriptable object that describes it. The prefab is responsible for things like how the weapon looks and sounds and the scriptable object is responsible for pure data like how many bullets does it have in a mag and how much damage does it cause. The idea here is that you could reuse the prefab for different types of weapons if you like where you could have the same prefab but have it cause different amount of damage depending on weapon quality or something like that.

3.1 Procedure for creating a new weapon

Right click in the project view, select *Create -> Firearm data*. This will create a new *Firearm Data* scriptable object.

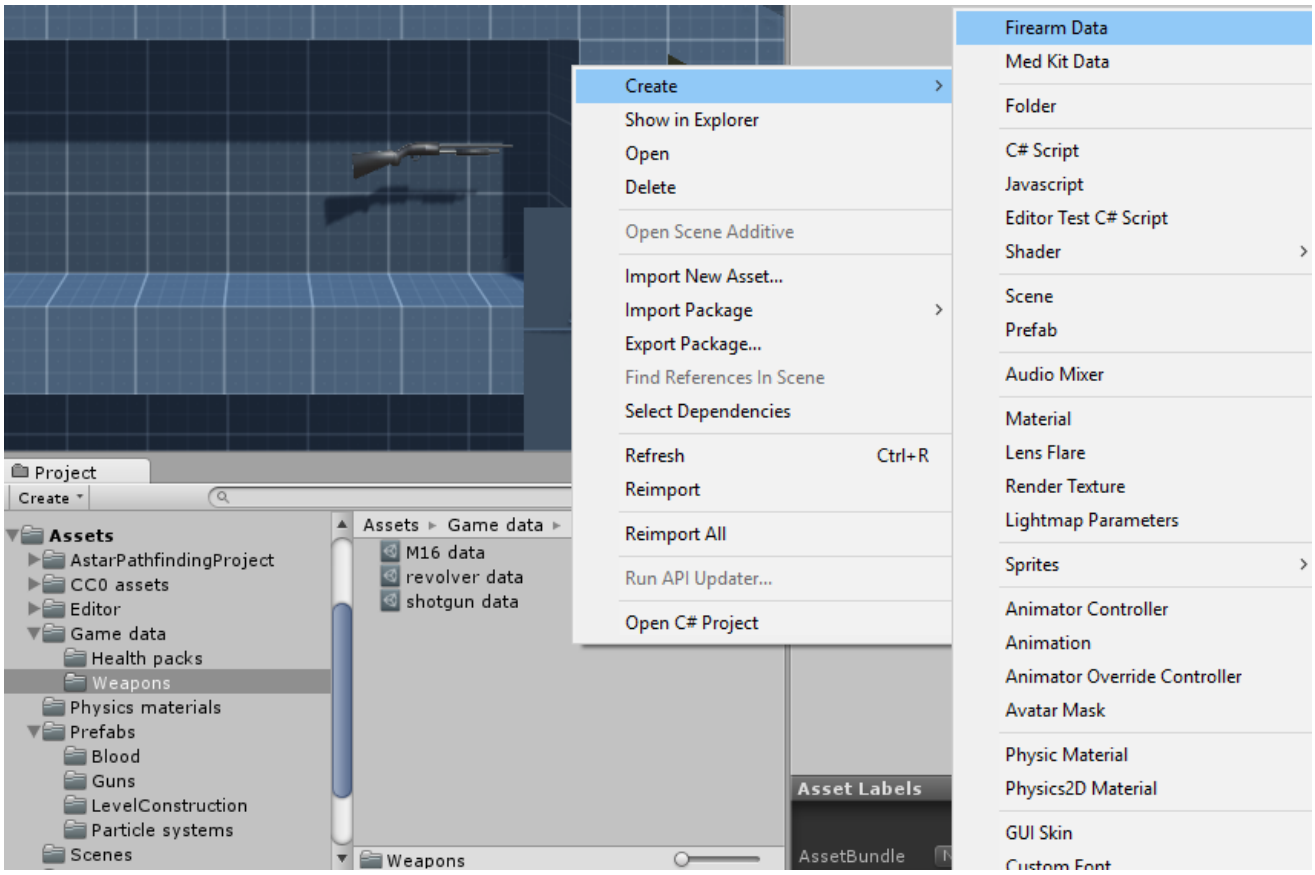

Figure 3 Create Firearm Data scriptable object

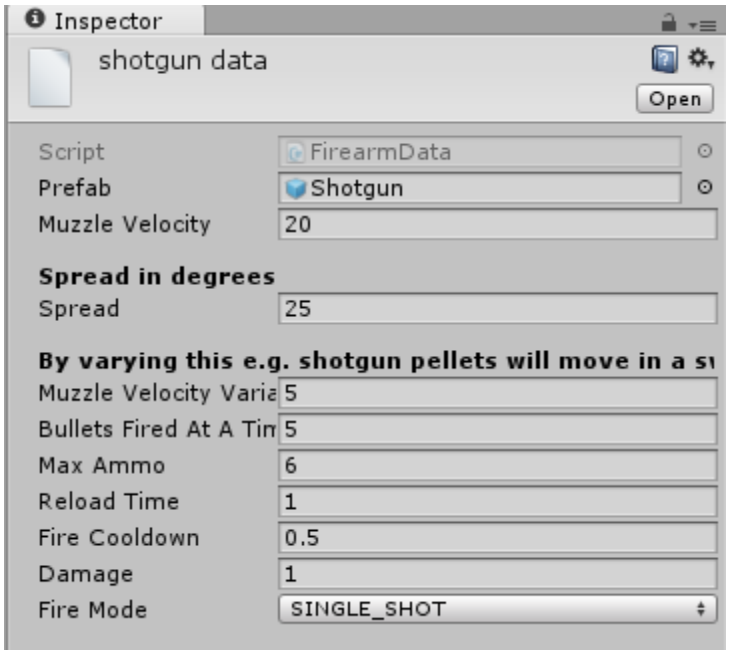

Figure 4 Edit firearm data

Next, duplicate one of the already existing weapon prefabs or create your own. If you create your own, then assign the same components and children as the existing prefabs. Make sure to assign the prefab in the *firearm data* scriptable object.

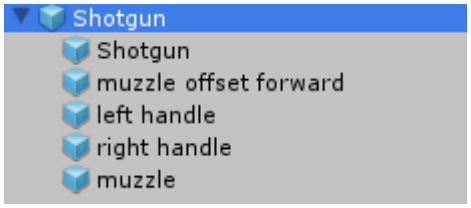

Figure 5 Hierarchy of a firearm prefab

We use *inverse kinematics* (IK) to determine the position of the character's hands. This way we do not need a specific animation for every weapon we create. The *Firearm Behaviour* script needs a reference to the hands if you want their position to be controlled. If you do not, then leave the references to null and the inverse kinematics will ignore them and let the normal animation run its course. This way you could assign the right hand only if you want the character to only hold a weapon with one hand. The *muzzle offset forward* is used to make the character point is head a few meters in front of the barrel to make it look like he is aiming. If you do not want this behavior, then just do no assign that variable.

It should be pointed out that the IK of the Unity Engine are not fantastic. If you want better IK it is strongly recommended that you have a look at the package *Final IK* by Root Motion (https://www.assetstore.unity3d.com/en/#!/content/14290) in the asset store. The principle will still be the same, the end result will just look a lot better.

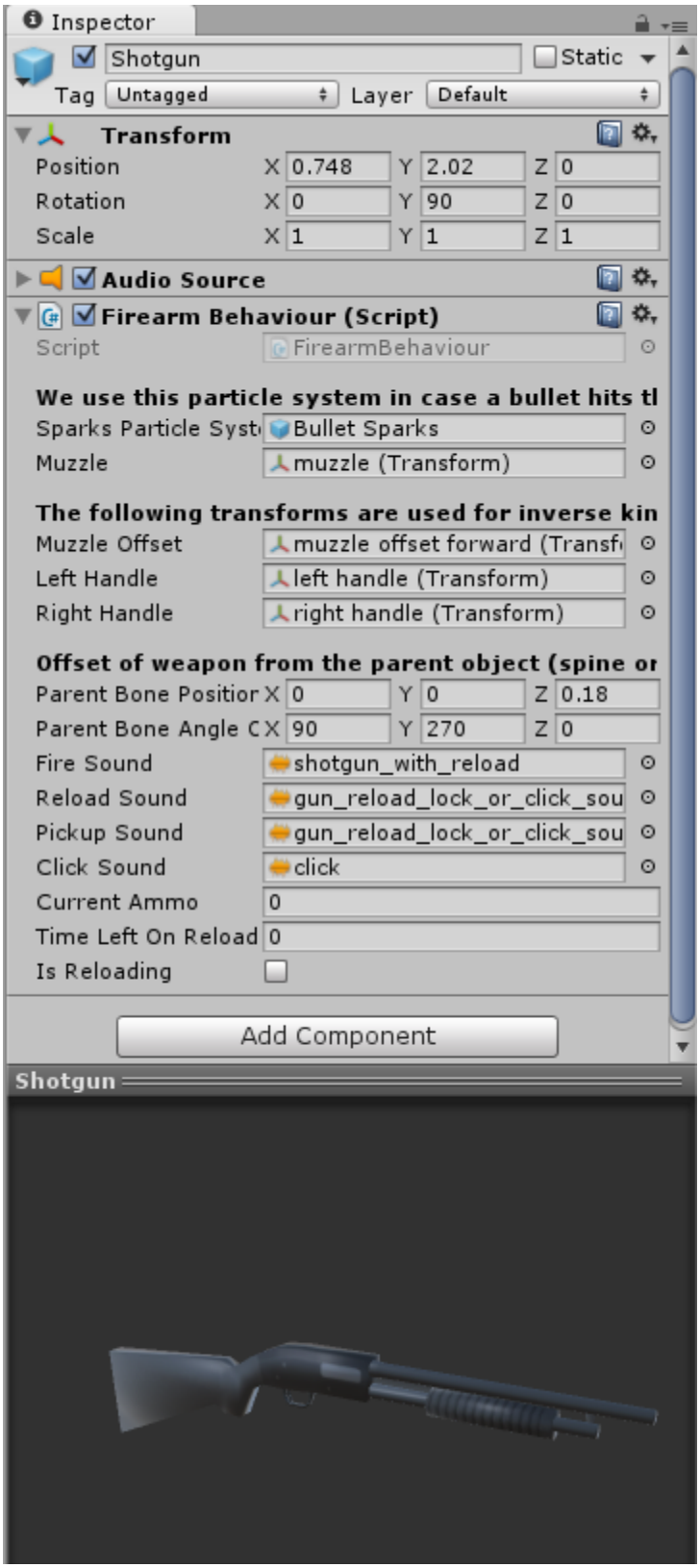

Figure 6 Example of the components of a firearm prefab

3.2 Bullet manager

As there will be tons of bullets flying in this type of game it is important to keep down garbage generation. If we created and destroyed game objects for every time a trigger was pulled the garbage collector would be up in its neck in just bullets. Therefore, when a bullet "disappears" for some reason (maybe entering a body or striking a wall) it is recycled by the *Bullet Manager* and will be reused next time someone fires a weapon. Note that if you want to add a new type of bullet, like exploding rockets, arrows or thrown axes, then you should add a new object pool to the bullet manager for that type of bullet.

If this is a new concept to you then you may want to read up on it:

http://docs.unity3d.com/Manual/UnderstandingAutomaticMemoryManagement.html

4 World

Pathfinding for jumping platforms is more complex that one would think. In this system we have tried to create as simple a system as possible for this. In order to allow for pathfinding, we must build the world in such a way that it can be interpreted by the pathfinder.

Specifically, you should have a look at the building blocks in the Prefabs/LevelConstruction-folder. Note that the *Viking Prototype Cube*-prefab has a script called *GenerateWaypoints.cs*. This script uses the two variables *Start Point* an *End Point* and places waypoints along a line between these two points for the pathfinder to use. This is to let the pathfinder figure out where there are platforms that characters can run on. If you want to place something in the world that the ai should not run on, then you do not need this script. You could instead manually place *Waypoint*-prefabs anywhere you want the AI to be able to run but this becomes tedious fast… Make sure that the waypoint intervals are shorter than the *Max Distance* variables set in the *Jump Graph* of the A* pathfinder or they will not be connected. Don't make them *too* short or there will be too many connections in the graph to make it efficient.

Note that the shader used by the material for the construction prefabs allows you to rescale the objects and maintain texture uvs in world coordinates. You can probably use this using your own prefabs and textures as well.

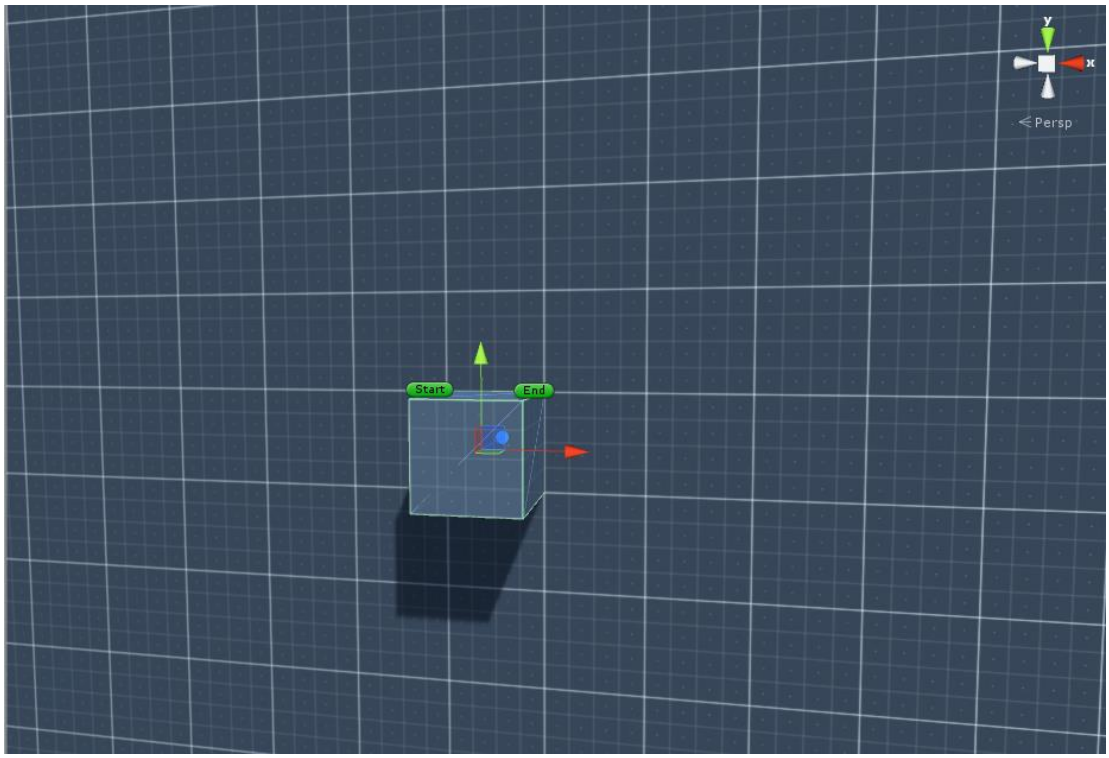

Figure 7 A Viking Prototype Cube placed in the editor

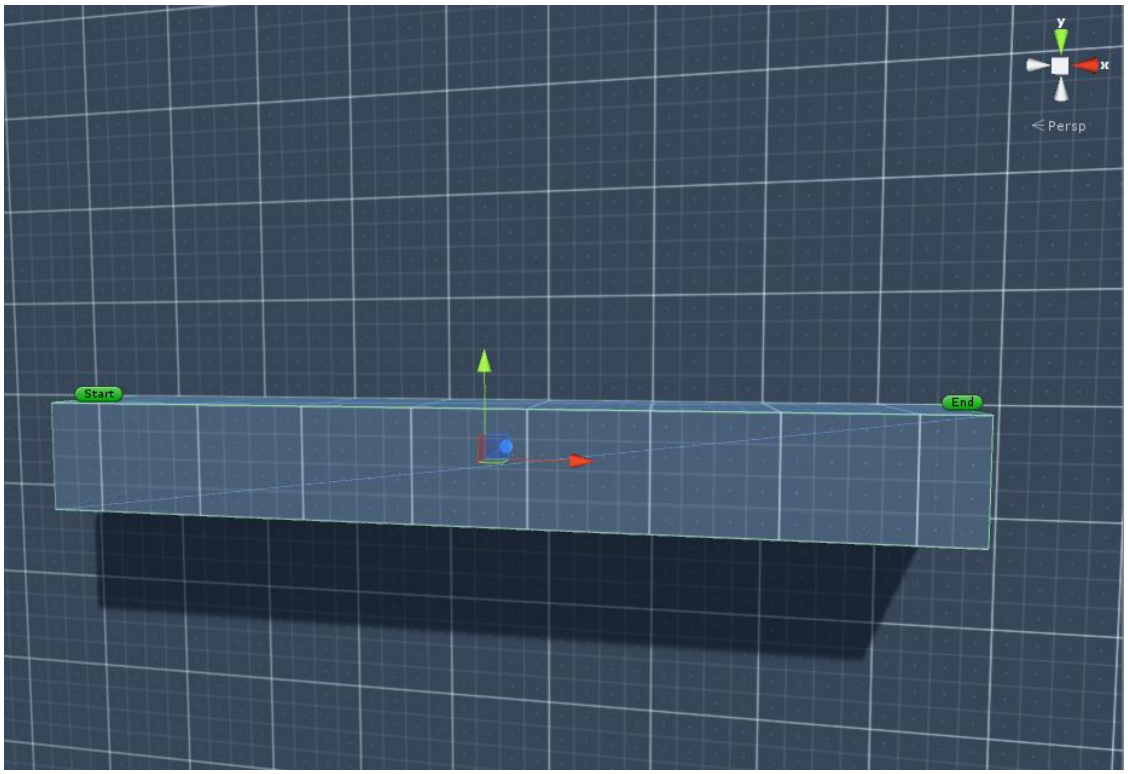

Figure 8 The same cube with the scale along x-axis increased. Note how the texture adapts according to world coordinates.

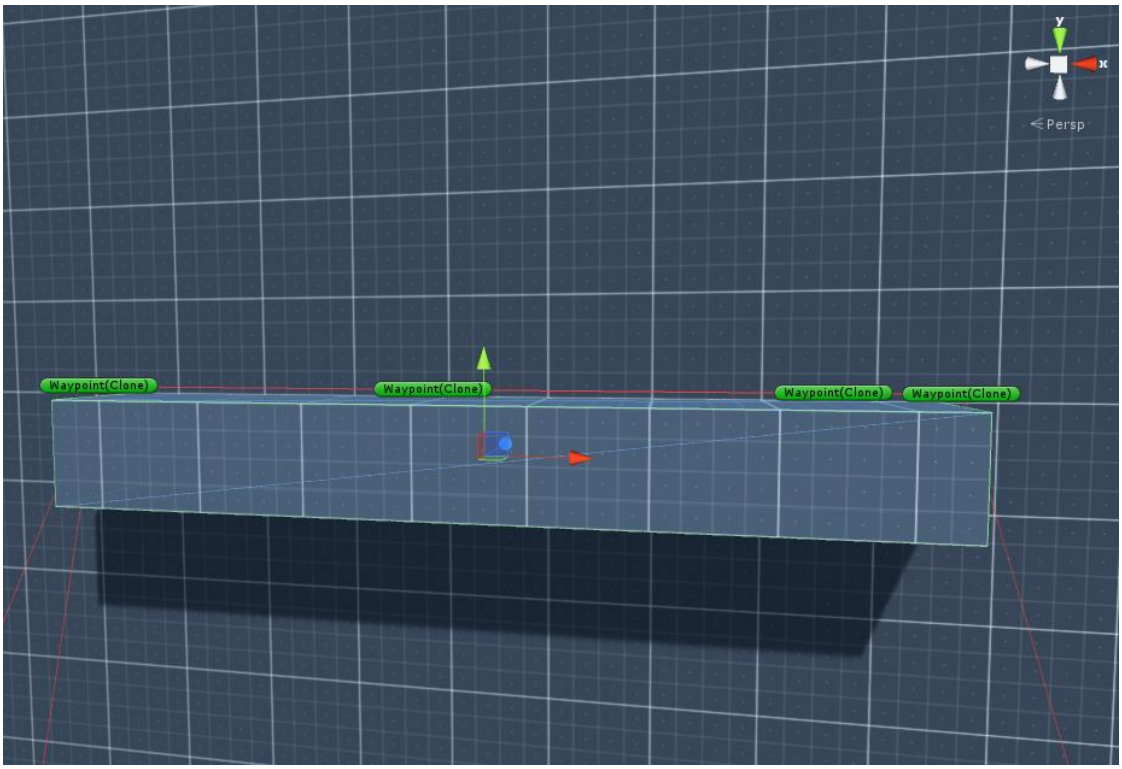

Figure 9 The same cube now during gameplay. Note how waypoints have been automaticlly placed along its walkable surface. If you look closely you can see how the red lines show traversable edges in the graph.

5 Pathfinding

Aron Granberg has kindly permitted the free version of his awesome *A-star Pathfinding Project* to be included in this project. If you own the pro-version of that asset and want to use it instead, then read the instructions in the *JumpGraph.cs*-script. For more information about the *A-star Pathfinding Project* check out http://arongranberg.com/astar/

In this project we have implemented a new type of graph for pathfinding where actors need to jump between platforms. You can find the implementation details in *JumpGraphs.cs.* It introduces two new features; a waypoint node will only be connected to a node above it if it is no higher than the variable *Max Jump Height* of the Jump Graph in the inspector. Vice versa there is a max drop height that characters will allow to avoid jumping from too high heights. Nodes are also only connected if there is no 2D collider between them.

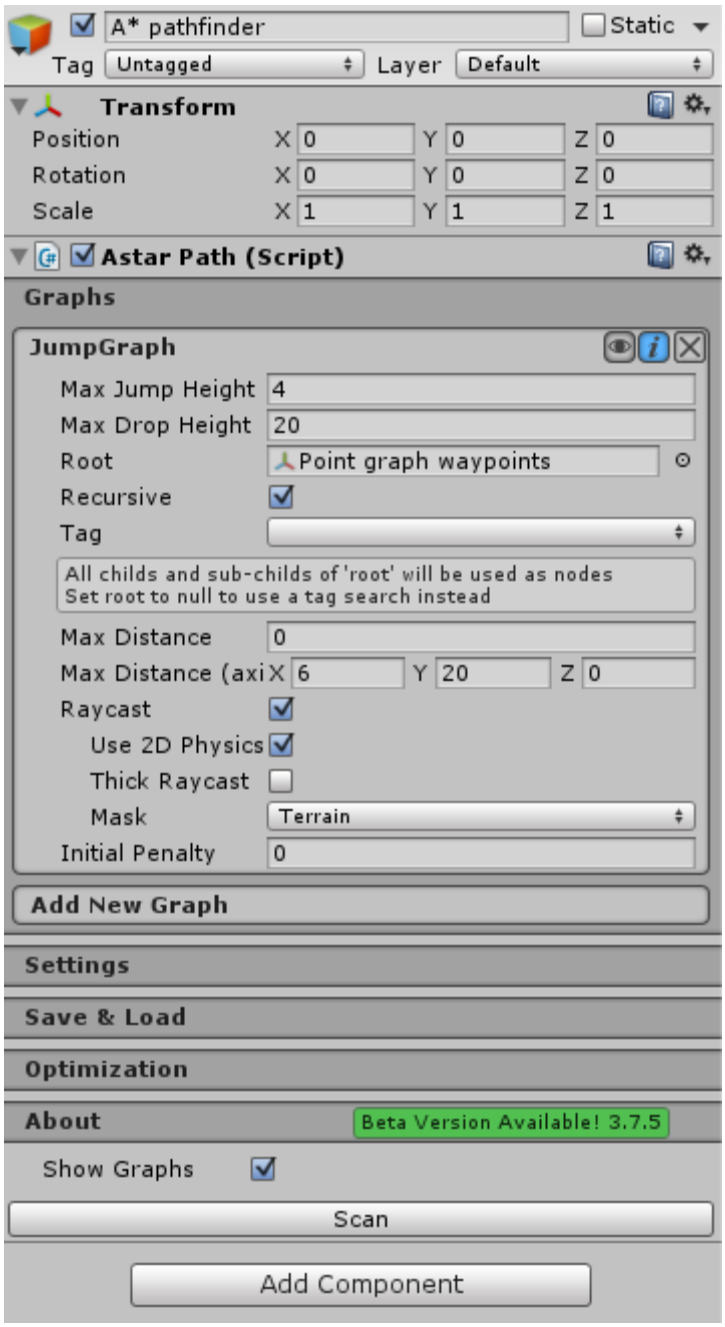

Figure 10 You can set variables like max jump height in the inspector. Note that the graph will use both jump and drop height and max distance to determine connectivity.

6 Planned features

There are many ideas for future updates. Some of them are:

- Wall climb
- Wall jump
- Jetpack
- Grapple hook
- Weapon recoil and reloading animations

- More guns: Machine guns, rocket launchers, hand grenades etc.
- X-Box (etc) controller support
- More object pools for characters, weapons etc for increased performance
- Multiplayer, local or online
- UMA 2.0 integration
- Voxel world package integration for destructible terrain (almost done, just needs more testing)
- Playmaker support
- Integration of other packages such as Final IK, PuppetMaster, DunGen etc.

There is a fine balance when adding features in keeping the package clean and simple to use and the awesomeness new features would bring. If you as user or presumptive customer have any feelings that you'd like to share, then please do so through either of the channels mentioned in the introduction.