Unity 3D Third-Person Game Foundation Kit
-User Manual-

Contents:

Versions:

Current Version = 1.5

Version Differences and Previous Versions:

Consult the documentation for notes on previous versions and version differences/additions.

F.A.Q.

I just bought the asset and I hit play to run the game in the editor. Why can I not control the character and what are all these errors related to Input?
This is due to not having the proper Input Manager asset file. Unfortunately I cannot bundle this with the asset so please follow the link in the **Support** section of this user manual to download the one I have created. Simply put it in the 'ProjectSettings' directory for you Unity project and replace the existing file.

I'm getting errors related to the GamePadInputs.cs script, why is this?
This is due to either there not existing an instance of this script in your scene, or it is in the wrong place. Make sure you have a game object with the tag “Handler” and have the script attached to that object. There is a 'Handler' prefab in the prefabs folder, supplied with this asset that comes ready prepared.

The targeting feature isn't working when I use the triggers on my gamepad and I'm sure that the function is switched on in the characters inspector. What gives?
This is because you're using a controller that reads triggers as buttons rather than axes (or vise versa). Go to your instance of the GamePadInputs.cs script in your scene and tick/untick the box in the inspector that reads 'TriggersAreButtons'.

My project won't build. Why?
The Unity Submission guidelines require 'graceful error handling' and this means pausing the Unity Editor (through script) to prevent null reference exceptions (as assets aren't allowed to have these in any case). Therefore, the scripts in the assets contain references to the Editor and when a project is being built it cannot reference the editor. Therefore, some lines of code have to be removed on the users end. To see which, please refer to the **When It's All Working** section of this user manual.

The F.A.Q. Didn't help me with my problem. Where do I turn?
Head over to the **Support** section of this user manual and find my contact information there. I'll get back to you very quickly and we'll sort something out together. :)

This asset isn't to my liking. Can I get a refund?
Refer to the previous question. And do not be hasty to give my asset a low rating on the asset store; whether it's due to bugs, the asset not working like you want it to, some features that you desire aren't present, or any other criticism, I will try to solve this issue and it may prove to be a 5 star asset for you in the long run (and I'd definitely appreciate a good review)!

There's something I think should be added to a future update. Can I say?
Half of the features in this asset are due to feedback/criticisms so your voice will not go unheard if there's something you want to see but the asset doesn't currently offer it.

I absolutely love this asset! What can I do to show my appreciation?
I'm glad that you've gotten so much use out of this asset and I can't wait to see what amazing work you might build on top of it. If you're dying to show your appreciation, leave

a rating on the assets store page, drop a review or even contact me to just share your excitement! I always have time to get back to anyone that gives their time to contact me.

Overview of Scripts:

Here are a list of scripts present in the asset:

-ThirdPersonController.cs [ANNOTATED]
-CameraFollower.cs [ANNOTATED]
-GamePadInputs.cs
-HUD.cs [ANNOTATED]
-Inventory.cs [ANNOTATED]
-Item.cs
-NPC.cs [ANNOTATED]
-OnClickNPC.cs [ANNOTATED]
-Pickup.cs
-TargetNumber.cs
-WaterForce.cs [ANNOTATED]
-BillBoard.cs

NOTE: The scripts aboved marked with '[ANNOTATED]' are fully annotated; there are comments after the '//' syntax on most of the lines (or at least before code segments) to explain what that line/section of code does and how it functions. The other scripts are sectionised and slightly annotated at most or are too short and simple to require explanations.

ThirdPersonController.cs:

This script is the one attached to the character that the player will play as. This handles the movement of the character, the controlling of the characters animations, and essentially every feature mentioned in the *Documentation.pdf* file. The script is sectionized clearly so that you (the user) can make adjustments to particular mechanics without altering/interfering with other mechanics. The script requires (in terms of components) a Rigidbody, a Capsule Collider and a Sphere Collider; the script also requires (in terms of other Transforms) the character model game object, Root Bone game object, Head Mesh game object and the bone that will act as the position for the first person mode. Each of these things can be dragged into the inspector and all of the features are able to be turned on or off through the inspector. I will explain what these transforms are and why they're necessary in the 'Character Prefab' section.

CameraFollower.cs:

This script is the one attached to the Main Camera of the scene; the one that you (the user) may wish to have follow the character. This simply handles the way that the camera is moved and controlled in all four of its modes (behind, targeting, free and first-person-view (FPV)) and the handling of repositioning the camera with obstructions and its collisions with walls. This script only requires the Camera component of the Main Camera itself and will instantiate any child transforms to the camera game-object.

GamePadInputs.cs
This script has been added to make using inputs easier for if there is a gamepad connected or the user just wishes to use a mouse and keyboard. It works by having its own values such as 'pressAction' and 'holdAction' and has these two values for each button, and also takes the values of any axes used. So if a mouse button is pressed or the corresponding gamepad button is pressed, it will read it as 'pressButton'. The inclusion of this script has changed every instance of 'Input.GetX' to 'GamePadInputs.PressButton' or '.HoldButton' or '.SpecificAxis'. It makes writing inputs easier in the scripts and doesn't require the user to check if it's a mouse/keyboard button/key or a gamepad button.

HUD.cs
This script takes the GUI Canvas that the user would want to use for their HUD, takes the specific GUI Elements (Hearts (Image), bars/meters (Slider[]), Money (Text) and hotkeys (Button)) and allows the user to set quantities, maximum quantities, spacing in positions, speeds, values, etc. and will draw the entire HUD with the specified conditions at runtime.

Inventory.cs
This script takes the GUI Canvas that the user would want to use for their Inventory (pause screen), takes the specific GUI Elements (Items (Button), Previous Page (Button), Next Page (Button) and Item Information (Text)) and allows the user to, again, set quantities, maximum quantities, spacing in positions, speeds, values, etc. and will draw the entire Inventory screen with the specified conditions at runtime.

Item.cs:
This script is to be attached to every item prefab as it holds values essential for use with the inventory system. It keeps the values of the item name, its icon, its quantities, its max quantities and if it uses quantities.

NPC.cs:
This script is to be attached to every NPC that the user wants the character to interact with. It requires the NPC game object to have a Sphere Collider trigger. It takes the GUI Canvas that the user will want to display text boxes (dialogue) on, takes specific GUI Elements (Text Box (Image), Replies (Button)) and allows the user to set quantities, spacing values, dialogue content, replies, resulting dialogue, text speed, etc. and will draw everything at runtime with the specified conditions.

OnClickNPC.cs:
Works as a relay to check which GUI button has been pressed at a specified time so that clicking on a reply (to an NPC) will alert the NPC.cs script which reply button specifically has been pressed.

Pickup.cs
This script is to be attached to every game-object that the user wishes to use as a pick-up item. The script keeps a value of what it will replenish and the amount to be replenish. This can be anything from health, money, arrows or bombs and whatever the user wishes to add. It also includes a function to hide the pickup after it has been 'collected' and then respawn after a short while; this is mainly used for testing purposes. It also has the pickup rotate in place for visual purposes.

TargetNumber.cs:
This script is to be attached to every game-object with the tag 'Target' (basically every object that you would like the character to target/focus on). It simply keeps an interger to give that target a unique identifier which is set and used by the ThirdPersonController script.

WaterForce.cs:
This script is required on any body of water that the user wants to create. It affects what state the character is in, how the camera displays an underwater effect and any forces that are acted upon the character while submerged in water.

BillBoard.cs:
This script is attached to the target ridicule and 'mouse icon grab' prefab that appear (move) in certain circumstances. This script simply keeps the transform facing the camera at all times to give the illusion of it being an on-screen ridicule.

Where to use the Scripts and their Requirements:

Introduction:
So now that you're acquainted with the scripts and their general purpose, this script will explain in which cases should the scripts be use. What game objects require them and why. And **pay attention to the tags** that certain game-objects have as this is vital for some of the scripts to locate the others.

ThirdPersonController.cs:
Only needs to be attached to the character, the game-object that the user wants the player to control. Other scripts will look for the game-object with the tag “Player” and that game-object is where the scripts will look for an instance of ThirdPersonController.cs. As for wanting to use multiple characters, you need only have the other scripts search for specific instances of it; however, as a feature, it is planned as a **potential** addition to version 2.0.

CameraFollower.cs:
Only needs to be attached to the main camera. Other scripts will look for the game-object with the tag “MainCamera”.

GamePadInputs.cs
Only needs to be attached to the 'Handler' game-object. The game-object needs to have the tag “Handler”.

HUD.cs
Only needs to be attached to the 'Handler' game-object. The game-object needs to have the tag “Handler”.

Inventory.cs
Only needs to be attached to the 'Handler' game-object. The game-object needs to have the tag “Handler”.

Item.cs:
Attach it to every prefab/instance of a game-object that you would want to be an 'item'. The game-object doesn't need any specific tag, collision layer or name to function properly.

NPC.cs:
This script is to be attached to every NPC that the user wants the character to interact with. The game-object needs to have the tag “NPC” and each NPC needs to have a different 'name'.

OnClickNPC.cs:
This script has to be attached to the Reply GUI Button on the NPC GUI Canvas (if being used) as it needs to reference the NPC.cs script. No need for a specific tag/name/layer.

Pickup.cs
This script is to be attached to every game-object that the user wishes to use as a pick-up item. It doesn't require any specific tag/name/layer, however it is beneficial to have them on the collision layer 'Pickups'.

TargetNumber.cs:
This script is to be attached to every game-object that the user wants to be focused on when targeting. The game-objects must have the tag “Target”.

WaterForce.cs:
This script is to be attached to every game-object that the user wishes to use as water. It doesn't require any specific tag/name/layer, however it is beneficial to have them on the collision layer 'Water'.

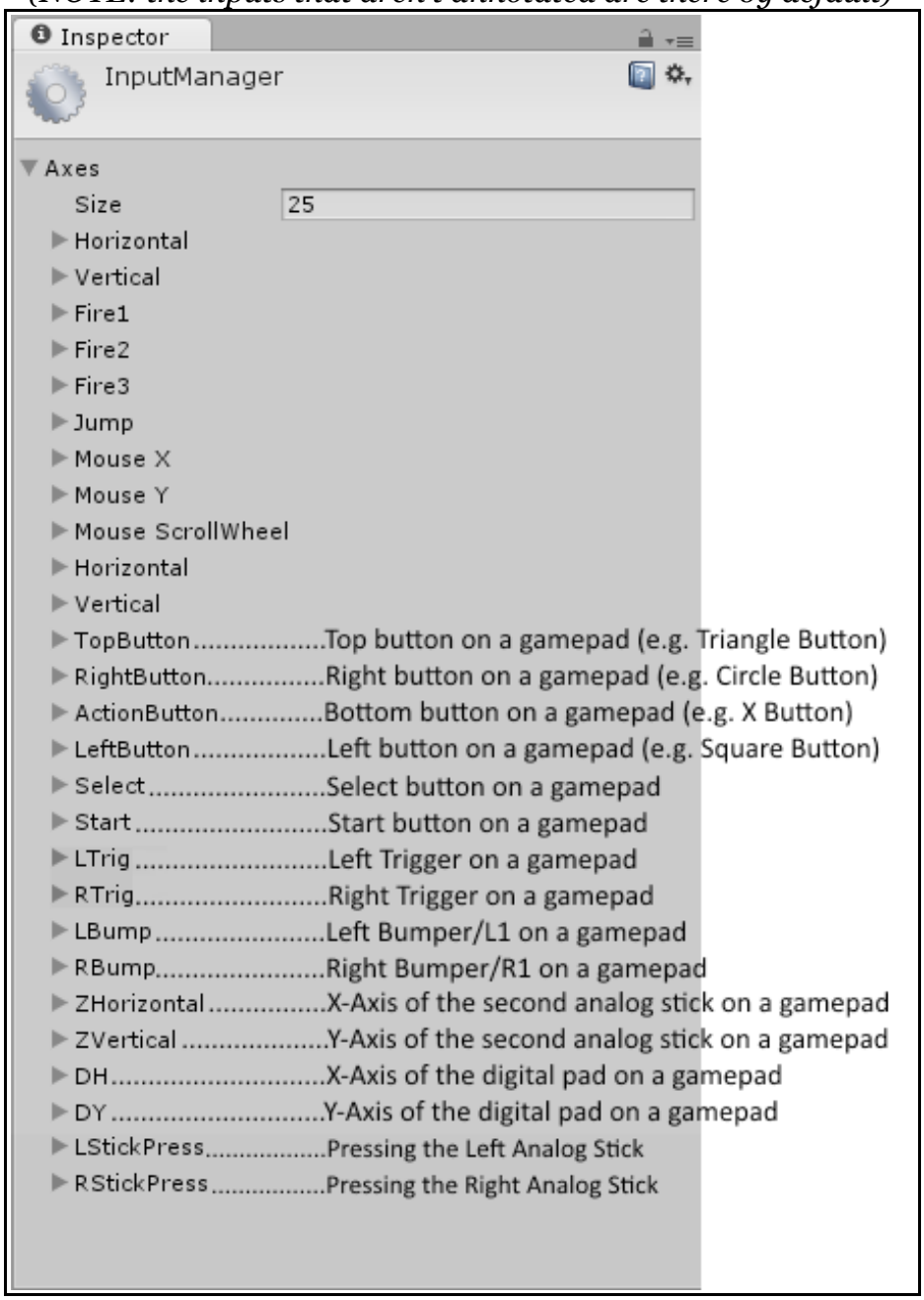
BillBoard.cs:
Completely optional script; simply attach it to any object that the user may wish to have facing the camera at all times.

Input Manager:

A download link for my input manager settings asset is in the 'Support' section at the end of this User Manual.

Introduction:
The input manager has added inputs to it that are made for use with a gamepad. This set up may not be completely compatible with your gamepad so you can make the correct adjustments. This image of the input manager better explains which input refers to what button on a gamepad:

(NOTE: the inputs that aren't annotated are there by default)

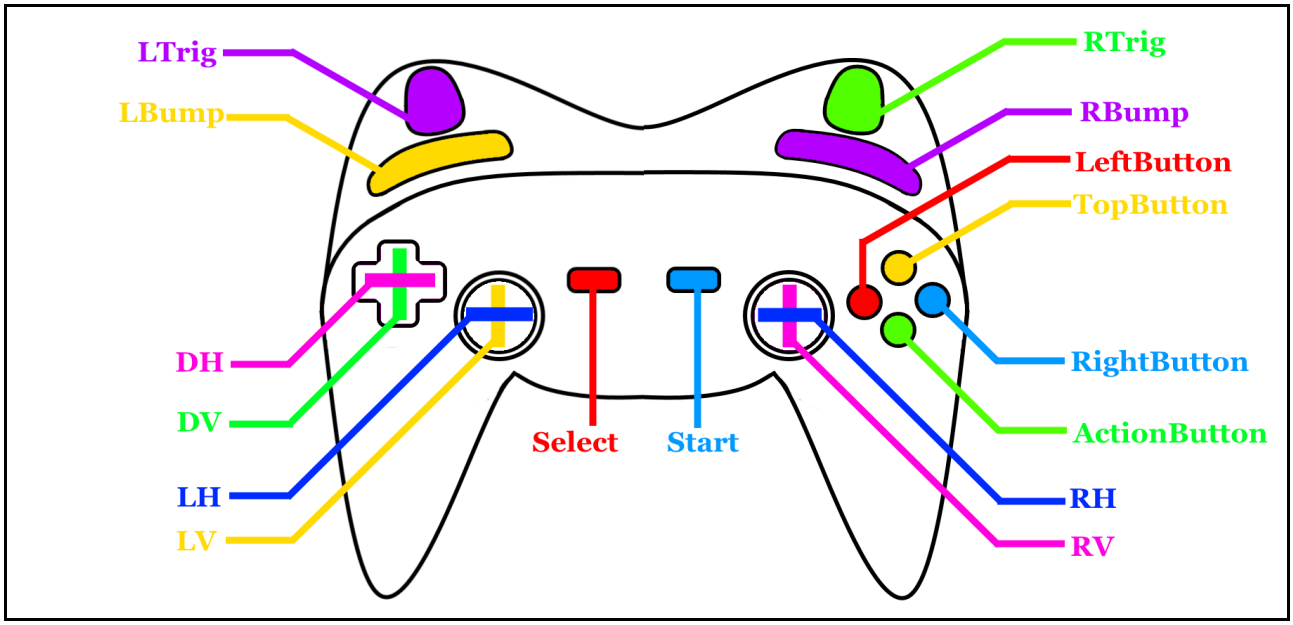


The only ones of these **new** inputs that are used in any of the scripts are:

-ActionButton (for rolling/jumping in target mode; alternate button is 'Mouse Button 0'),
-LeftButton (for jumping on command; alternate button is 'Space'),
-Triggers(Left)(for using the target camera mode; alternate button is 'Mouse Button 1'),
-LBump (for toggling first-person-view; alternate button is 'Mouse Button 3'),
-ZHorizontal/ZVertical (for looking around in **first person**; alternate input is 'Mouse X'/'Mouse Y'),
-Zvertical (for zooming in/out on the character in **free mode**; alternate input is 'Mouse ScrollWheel');

You may need to set the Triggers to be used as Buttons rather than as Axes. If you have changed it to Buttons, find your instance of GamePadInputs.cs in your scene and (in the inspector) tick the box next to 'TriggersAreButtons'.

Input Manager Visual Representation



Collision Layers and Tags:

Introduction:
There are a few Collision Layers and Tags that need to be used in this asset and have to be assigned properly for each game-object that they're assigned to to interact with other game-objects and their scripts properly.
NOTE: Make sure that these are assigned to the corresponding layer numbers as stated below. Otherwise you will need to change lines in the ThirdPersonController script and make sure that each layer number in the IgnoreLayer() methods is set to the correct one (e.g. if it says (10, 11) and your 'Player' layer is 19 and your 'Slopes' layer is 20, change it to (19,20)). To find the lines that need altering, simply search (Ctrl+F) “IgnoreLayerCollision” and you shall find each instance of the method.

Collision Layers [“Layer Name” (“Layer Number”)]:
-Player (10),
-Collisions (9),
-Slopes (11),
-Water(4) (Default layer, not referenced in script),
-Pickups(12);

Player:
The collision layer that the character and all of its child transforms are on. This should ONLY be assigned to the character as there are times when the collision between the 'Player' layer and the 'Collisions'/'Slopes' layers is turned off and the character is programmed to stay stationary at these points.

Collisions:
The collision layer given to any mesh that you (the user) want to act as ground that can be traversed on (or it can be a wall; the character will not walk on it unless it's within the specified 'walkableSlopeTolerance' in the characters script (is changed on a slider in the inspector).

Slopes:
Acts just like the 'Collisions' layer, however if the character lands on a mesh of this layer (or attempts to walk onto it), they will slide down the mesh (or be pushed away from it/slip back slightly) and will remain uncontrollable until the character seizes contact with the mesh. The gradient of the mesh is irrelevant; walls can also be of this collision layer and shouldn't affect climbing ledges, walking against the walls, etc. But it is recommended that all walls are of layer 'Collisions' and slopes only used on inclines that the user may not want traversable.

Pickups:
Use for any pickups if you want any collectables to interact with other collision layers differently.

Tags:
-Player,
-Target,
-Ladder/Fence,
-Box,
-Handler,
-NPC;

Player:
The tag given to the character game-object. Only has to be assigned to the parent transform as that is the one with the script attached and is the start of the address that cameras script uses to find the character in the scene.

Target:
The tag given to any object that you, the user, wants the character to focus on when targeting. The transform that is tagged this should be the one with the TargetNumber script attached to it.

Ladder/Fence:
The tags given to either of the two climbable objects used in this asset: Ladder is used on the ladder object which only has the character move up and down on. Fence is used on the fence object which the character can move around on in all four directions and go across gentle (not sharp) corners on the fence.

Box
The tag given to any object that you want to be an interactive 'box' that can be pushed/pulled. This will really only work on cubes, though I recommend checking the 'The Box Prefab' section of this User Manual to understand how the current box prefabs are designed and how they might direct you to make your own (if you do so wish). Basically, make sure that any interactive box has this tag.

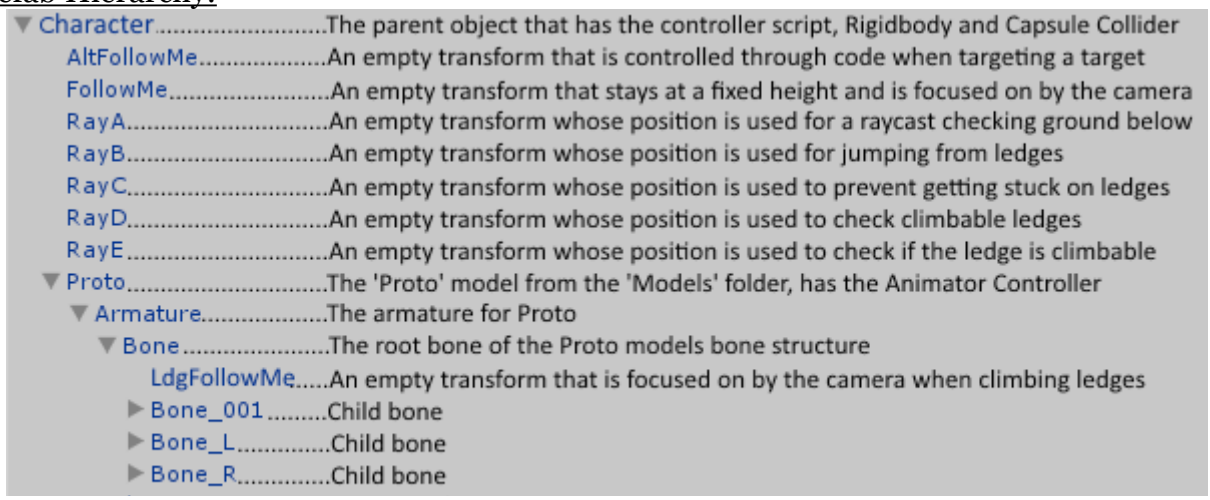
Handler
The tag given to the Handler object that contains the 'Handler.cs' script which handles the inventory and HUD. Other scripts (such as 'Pickup.cs') will search for an instance of the script through this tag so make sure that your Handler object has this tag.

NPC
The tag given to any object that the user would want to have as an NPC. The object should contain the 'NPC.cs' script which handles NPC interactions. Make sure that each object instance with this tag has a different name.

The Character Prefab

Introduction:
The character prefab (in the folder marked 'Prefabs' > 'Character') is the one named 'Character'.

Prefab Hierarchy:



NOTE: All the child transforms (apart from “Proto”) are created in the ThirdPersonControllers Start() method. The image above was taken after the child transforms have been created and the game is running. Naturally, the Character prefab only contains the 'Character' parent and the 'Proto' child (also, the 'LdgFollowMe' child is created after running Start() and isn't there in the prefab file). As of version 1.5, the character object also contains a child known as 'BlobShadowProject' (Standard Asset).

Also, the other child objects of 'Proto' (basically, all apart from the 'Armature' child) have been omitted from this image. The other child objects are the individual meshes that make up the character model (e.g. Head, Torso, Waist, etc.). More about them beneath, under the title 'Proto Model Meshes'.

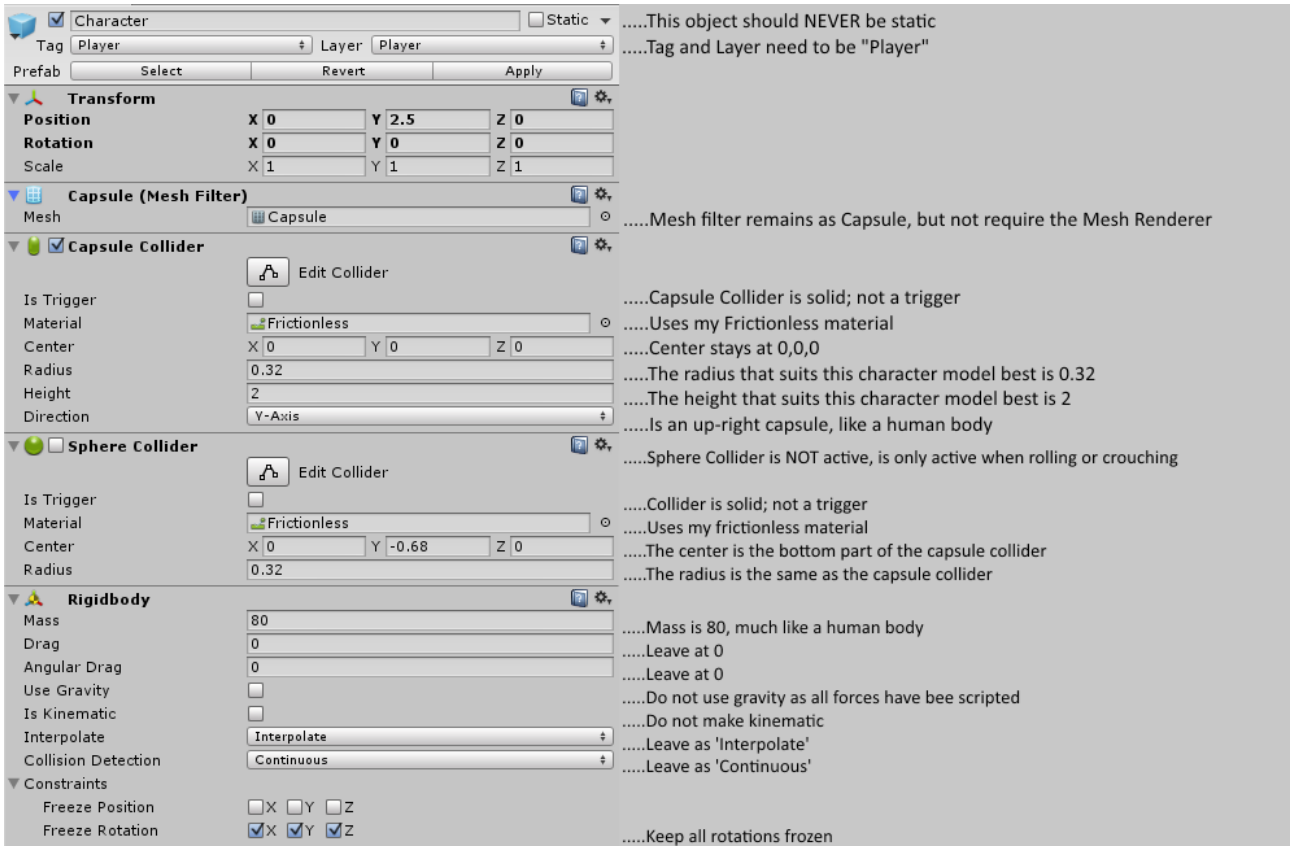
Both the camera script and the controller script will look for certain child transforms (mainly the FollowMe's and Ray ones). It is essential that if you, the user, wants to remove any of these transforms in the Start() method, then the appropriate lines of code will have to be edited (e.g. wherever there is a FindChild method, the path will have to be changed). However, simply using the prefab as is, only to include your own model and animations, there's no need to worry about changing any of the paths in the code.

The 'Proto' game-object also holds the Animator component, and hence the Animator Controller that the character uses and the one that the controller script references to. It is important that the animator remains on the models game-object, not the parent 'Character' object.

Proto Model Meshes
There are 17 separate meshes that make up the 3D character model, eached named after their respective body part. They are all child objects to 'Proto'. There is one mesh in particular that is important to how the code functions, this is the 'Head' mesh (named

"Cube_001" in previous versions (versions 1.2 and 1.0)). It is important that this mesh keeps its name as the Camera script will look for this mesh when going into FPV. The reason for this is that when the camera goes into FPV, the head mesh is made 'invisible' so that it doesn't clip through the camera.

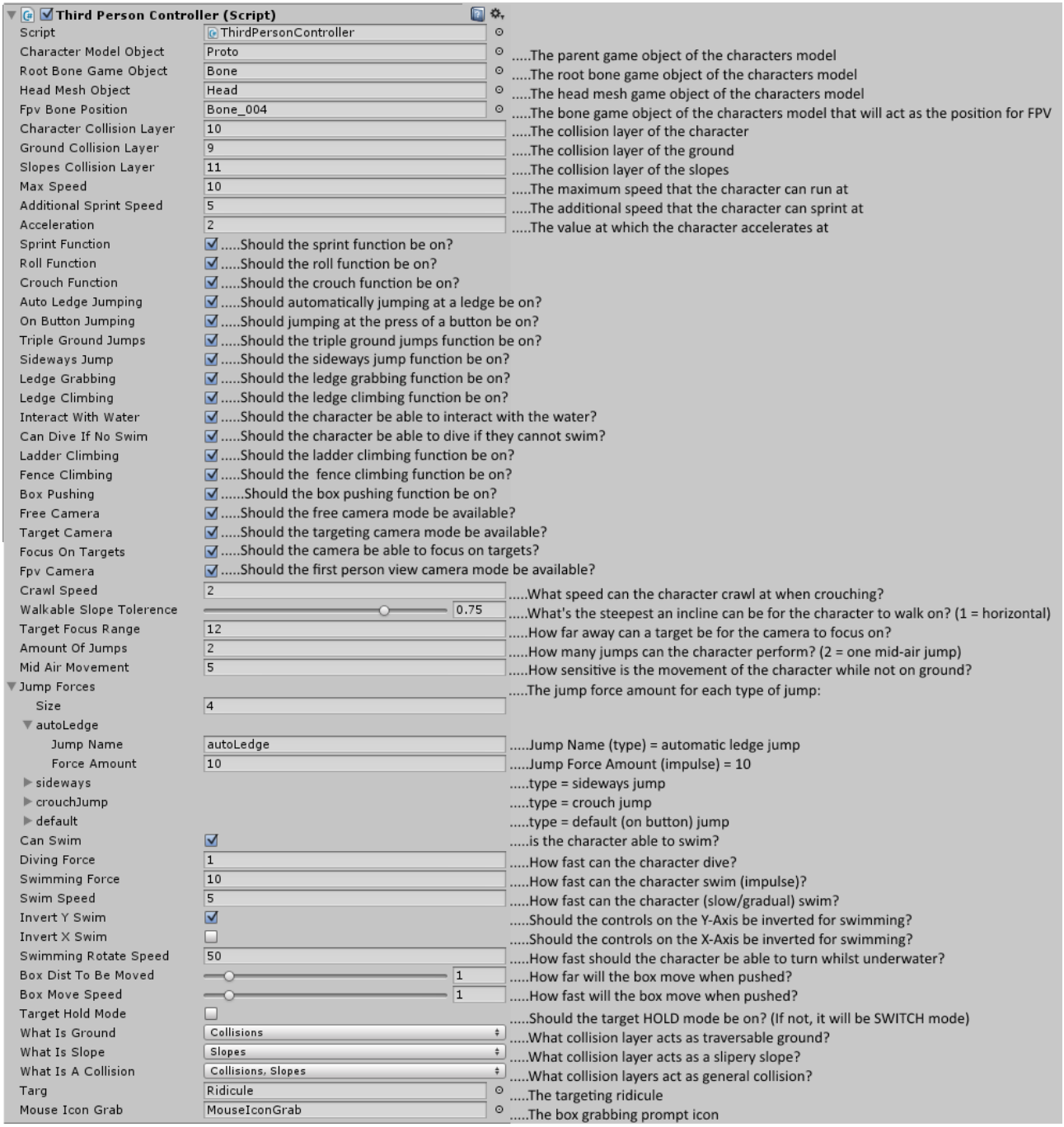
Components:



When wanting to have **a character with a different size**, only change the height and radius of the Capsule Collider and make sure your character model fits the collider. **Do not change the scale of the character** as this is different to changing the size of the collider and how the script scales all the raycasts. For example, if you want your character to be twice as tall as the default prefab, change the capsule height to 4. Do not change the transform scale y-value to 2 as the collider will remain the same size. Do not worry about changing the sphere colliders properties as these are handled in the script, relative to the size of the capsule collider.

All of the forces applied in the script are multiplied by the mass of the character. When making a smaller/bigger collider, be sure to scale the mass down/up (respectively) to give a more realistic feel. The default values of the prefab are to emulate the realistic weight/feel of a typical adult in terms of meters (height) and kilograms (mass).

Keep in mind that when making a smaller character (for example), you may want the character to move slower as a whole. Therefore, the thresholds in the animator (when it comes to speed) may have to be lowered (in this case) so that the character plays their running animation when at their maximum speed.



The top four variables can simply just be dragged into the Inspector from the scene view/hierarchy. No need to change any paths in code to find certain transforms since version 1.5.

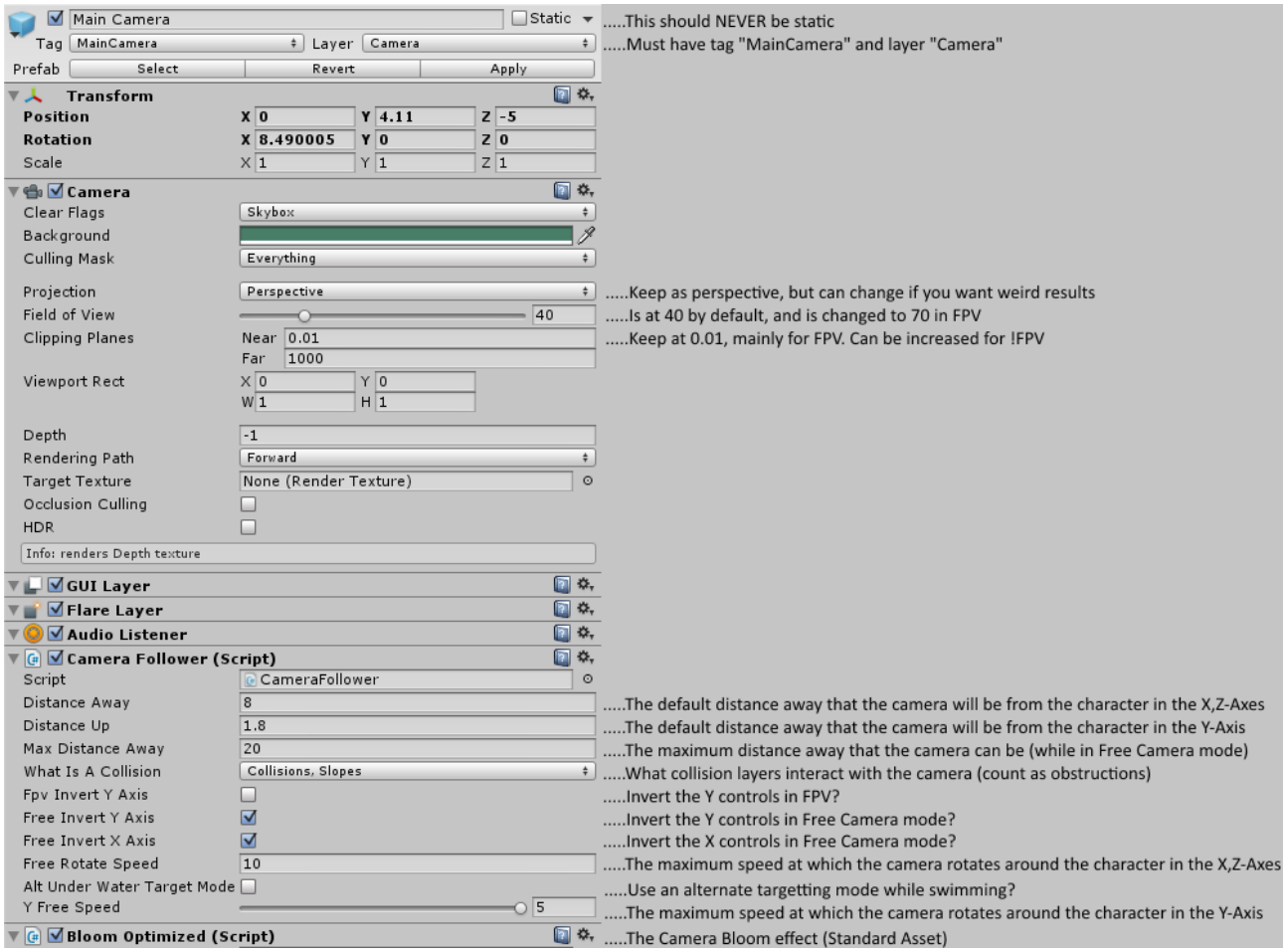
Make sure the collision layers match the actual IDs set in the Editor. To find out what the ID is for each layer, select the 'Layers' tab near the top-right of the Editor and then choose 'Edit Layers'.

The large selection of booleans regarding all the features is how the user can turn specific features on or off. No need to change areas on the code since version 1.5 as it can all be handled here in the Inspector.

The Camera Prefab

Introduction:
The camera prefab (in the folder marked 'Prefabs') is the one named 'Main Camera'.

Components:



The 'Bloom Optimized' script can be removed.

The Ladder and Fence Prefabs

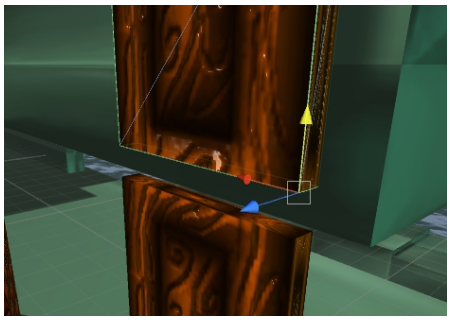
Introduction:
There are two climbable objects in this asset: the Ladder and the Fence. Each prefab comes as a single block tile and has a corresponding tag: “Ladder” or “Fence”.

Note: The character will only attach onto the FRONT of the ladder tile, not on any other side.

How to build a Ladder:
Simply use the 'Ladder' prefab in the 'Prefabs' folder and basically use them as building blocks. Try to make the ladder at least three tiles high so that it functions well (and realistically speaking, makes for a useful ladder). To do this simply, follow these steps:

1-In the Unity editor, select the following cursor/selection option:

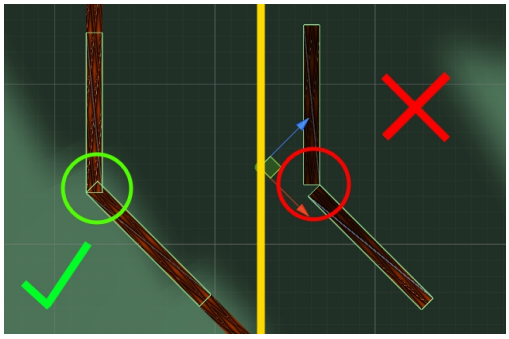
2-Select one of your ladder tiles and hold the 'V' button. This enables 'Vertex Snapping'.



3-Hover around one of the corners of the tiles and then left-click (and hold) and drag it over to the corner of another tile that you want to connect it to.

How to build a Fence:
The same principles apply as for making a ladder, however the tiles can go side by side. Additionally, since version 1.5, the fence can just be a large mesh/cube and is not limited to be made of tiles. If you wish for the fence to curve around, that is very do-able.
For building a fence out of tiles, simply pick a tile and rotate it by the desired amount on the Y-axis.

When joining two tiles of two different y-rotations, be sure to join them at the correct verticies. Make sure they connect at a point that doesn't leave a seam. This is what I mean:



The Water Box and Water Cube Prefabs

Introduction:
The Water Box and Water Cube prefabs are basically the same thing, just bodies of water. One is a large box collider with the Unity 5.1 standard asset 'Advanced Water' plane at the top, and another is basically a large cube made of water. Both function the exact same way by using the 'WaterForce.cs' script.

NOTE: This functionality is still subject to alterations/modifications in future versions.

How Do I Make A Body Of Water?
Easily just drag out one of the two prefabs and place it where ever you would like to have a body of water. Do not, however, place them together like tiles as this currently does not work when multiple water boxes/cubes are connected. The individual boxes/cubes can be placed anywhere and at any height or size, so it isn't that constricting at all.

Take notice of the WaterForce.cs script in the inspector; you have 'Force Direction', 'Force Amount', 'WaterBox ID' and 'inThisWaterBox'. **The only one you need to pay any attention to is 'WaterBoxID'**. When adding in a new water box/cube, make sure to change this field to any integer that isn't zero and isn't already the waterBoxID of another body of water (basically, make sure that no two bodies of water in one scene have the same waterBoxID). This just helps the character script keep track of what body of water they're currently in.

The WaterForce.cs script uses an optional Unity Editor feature to make being underwater more realistic; this is the 'Fog' feature that becomes a dense, blue fog to simulate the thickness/depth of water while swimming. The 'Fog' feature has to be checked in the Unity editor, and is found in **Window > Lighting >> Scene [Tab] > Fog .** Having this unchecked shouldn't cause any errors, but will not give any underwater fog effect. This is usually a good idea for making large underwater segments as the fog will mean less rendered objects in the cameras sight.

The Box Prefab

Introduction:
The box prefab is a cube with a Box Collider, “Box” tag and is of layer “Collisions”. The ThirdPersonController.cs script handles all the box's movements and hence doesn't require a script itself. It uses a rigidbody component so that it moves at a constant velocity and will interact with any colliders along the way. It also moves at a set distance each time so that it gives precise movements (the speed and distances can be set in the characters inspector in the editor).

How to use the Box
It's simple enough; just drag it onto the scene, make sure it's resting on flat ground. The character, when up against the side of a box, will have a prompt appear above them. If the player holds down the action button, the character will grab the box and be able to push it or pull it.

What to consider
The boxes only work well when they're moving across flat ground; they are not made to work on bumpy terrains or over multiple meshes of different heights. They also do not fall when pushed off a ledge (though this is something that is easily implemented and something that may appear in future releases).

The boxes search for obstructions at the foot of the box. If you're pushing it, it will search ahead of the box; if pulling, it will search behind the character (again, from the foot of the box). It will search just far enough so that it knows if can make the entire distance and be pushed up against a wall (it searches a bit further if it is being pulled as to account for the character since we don't want them being squashed up against a wall). Therefore, the boxes should be placed in 'pens', much like in the demo scene, where three are placed in a small square made up of small blocks.

The boxes have a Box Collider that is just barely smaller than their mesh (as you'll see in their inspector, the size of the collider is (0.9, 1, 0.9). The 'y' height stays the same, but the 'x' and 'z' sizes are smaller so that the meshes of the boxes can line up against each-other perfectly but also so that the colliders don't brush up against each other and cause problems with moving the boxes. Using the vector-snapping method that is shown in the 'ladder and fence prefabs', the boxes can be placed corner-to corner, side-by-side, while their colliders do no touch. Do not stack the boxes vertically.

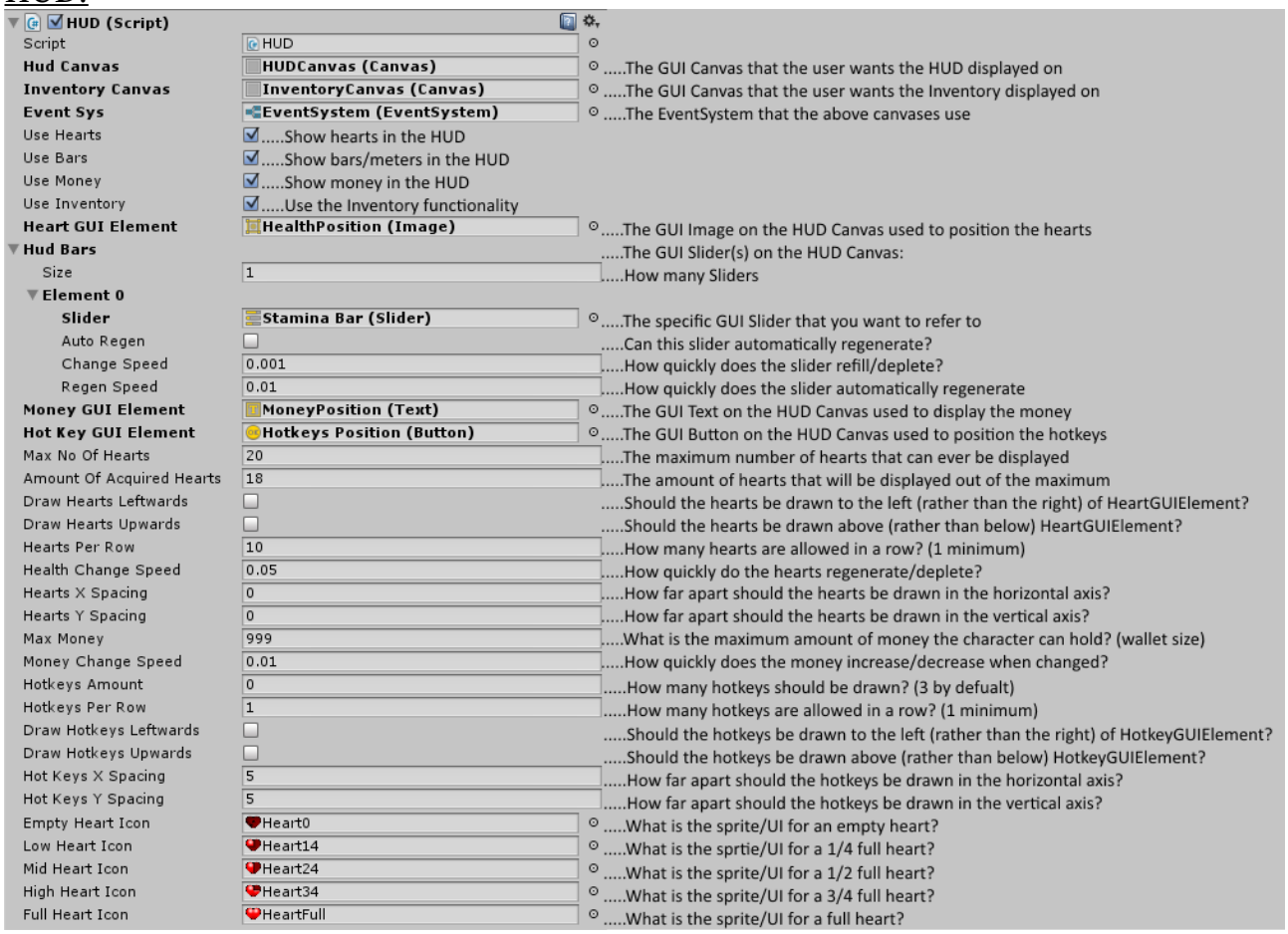
The Handler Prefab

Introduction:

The handler prefab is an object that the user would want to remain in the scene at all times and not physically interact with any other objects in the scene. Attached to it are the HUD.cs, Inventory.cs and GamePadInputs.cs scripts which other objects will be searching for. The HUD.cs and Inventory.cs scripts all handle the HUD, Inventory, health, stamina and money elements and keep them all centralised in one place. Use this to customise your HUD or inventory.

It is critical that your Handler game-object has the tag "Handler".

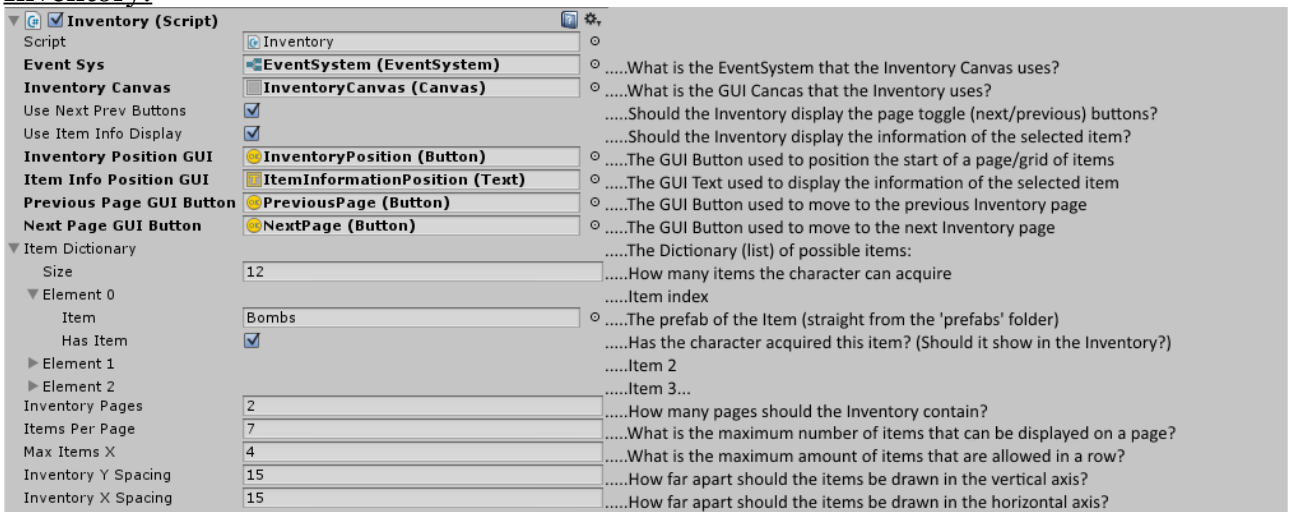
HUD:



For instructions on how to use the HUDCanvas in conjunction with the HUD.cs script, please check the '**HUDCanvas Prefab**' section of this user manual.

The HUD.cs script takes in the canvas the user wants the HUD to be displayed on, takes certain GUI Elements and then draws them to the specifications that the user wants at runtime. For example, if the user wants to add another Bar (Slider), select 'Hud Bars', increase the size by one, drag in the Slider GUI Element from the physical canvas, select whether they want it to automatically regenerate and then set the change/regeneration speeds.
Or, if the user wants the hearts to be displayed in two columns rather than two rows, simply set 'Hearts Per Row' to be '2' so that it will draw them all in two columns rather than two rows.

Inventory:



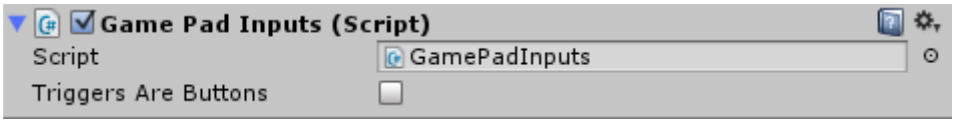
For instructions on how to use the HUDCanvas in conjunction with the HUD.cs script, please check the '**InventoryCanvas Prefab**' section of this user manual.

This functions similarly to the HUD section of this prefab. The Item Dictionary is the list of items that the character can possibly acquire. These items will be drawn on to 'Inventory Canvas' when the canvas is displayed (when the game is paused). Each item has its own specific slot so the first element of the dictionary (here in the Inspector) will be in the first slot, the second will be in the second slot, and so on. Here in the dictionary you can change whether that item has been acquired or not (hasItem) and this can be changed at any time. If the item has not been acquired then it will not be displayed on the canvas. The items passed into the dictionary must be Item Prefabs (which you can learn about in the **Item Prefab** section of this user manual).

The items are displayed on pages and it is up to the user whether they want the items being displayed upon multiple pages or just on a single page. If there is more than one page, then the player can make use of the 'Next/Previous Page Buttons' that navigate through the pages.

The user can select how many items are shown on each inventory page, how many items can be displayed in a row and whether to show the information of the currently selected item.

GamePadInputs



Many of the other scripts will look for the GamePadInputs.cs script which should be in your Handler game-object (the one with the tag “Handler”).

The 'Triggers Are Buttons' boolean can be set true if the gamepad that player is using uses buttons as triggers rather than as axes.

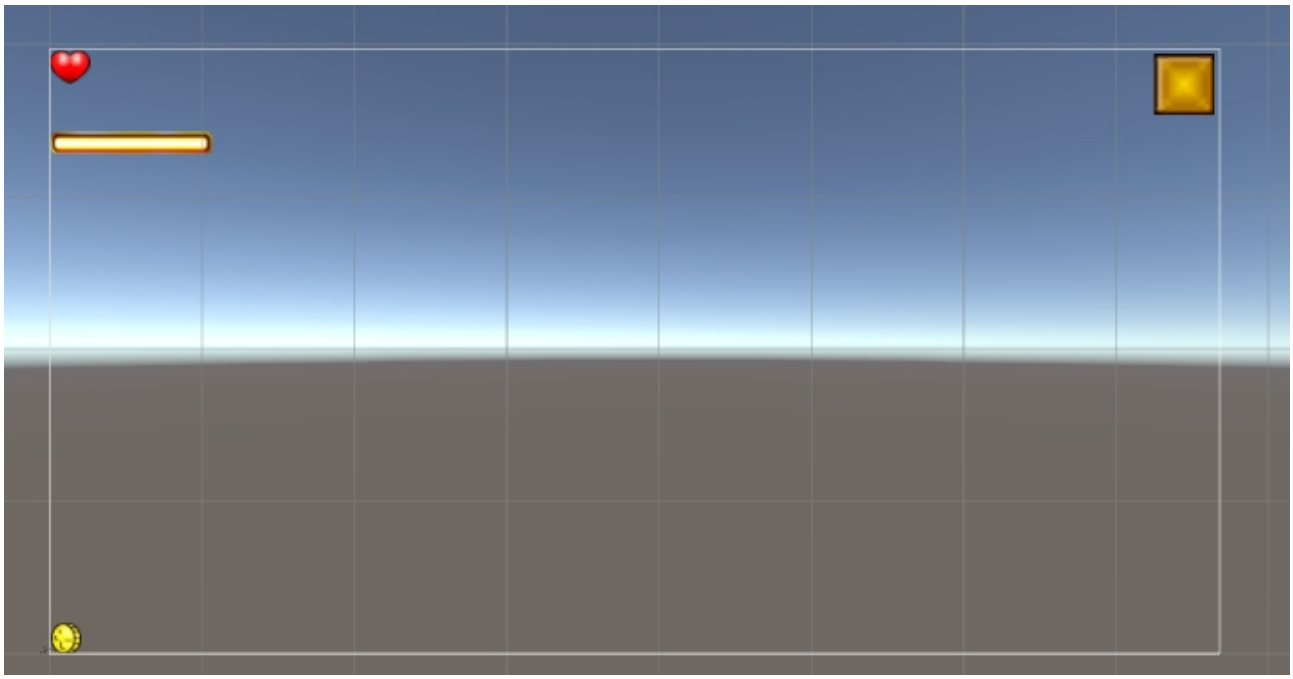
The Event System Prefab

Introduction:
The Event System prefab that's included in the asset is just a general Event System that has been configured to work in tune with other components of this asset. Since the asset uses GUI Canvases and GUI Buttons, it is necessary that an Event System exist in the scene. It isn't terribly necessary that the settings of the Event System are exactly what they're set to be in the prefab, but it has been set to use Inputs mentioned in the Input Manager that this asset uses and is set to 'Send Navigation Events' which is needed to navigate between reply buttons (in NPC interactions) and between the items in the Inventory.

How to use it:
Simply have it exist in your scene and drag it into instances of the HUD.cs, Inventory.cs and NPC.cs scripts so that they know which event system to use.

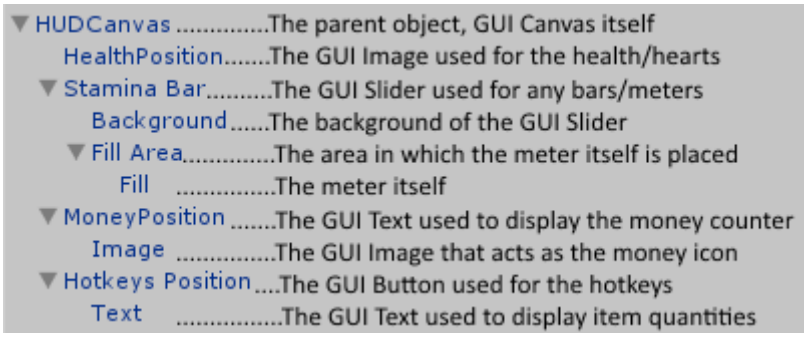
The HUD Canvas Prefab

Introduction:



The canvas itself can be set to however the user wants. The canvas (in the inspector) can be set to work with whatever the user needs in their game and so it can have different scaler settings, rendering modes, etc. All that the scripts that I have supplied require (even then, it's if you want them displayed) are a GUI Image for health (the heart in the top left), a GUI Slider (or more) for stamina, a GUI Text for displaying money and a GUI Button for drawing hotkeys. The rest of the design choices are up to the user and what works best for them.

Hierarchy:

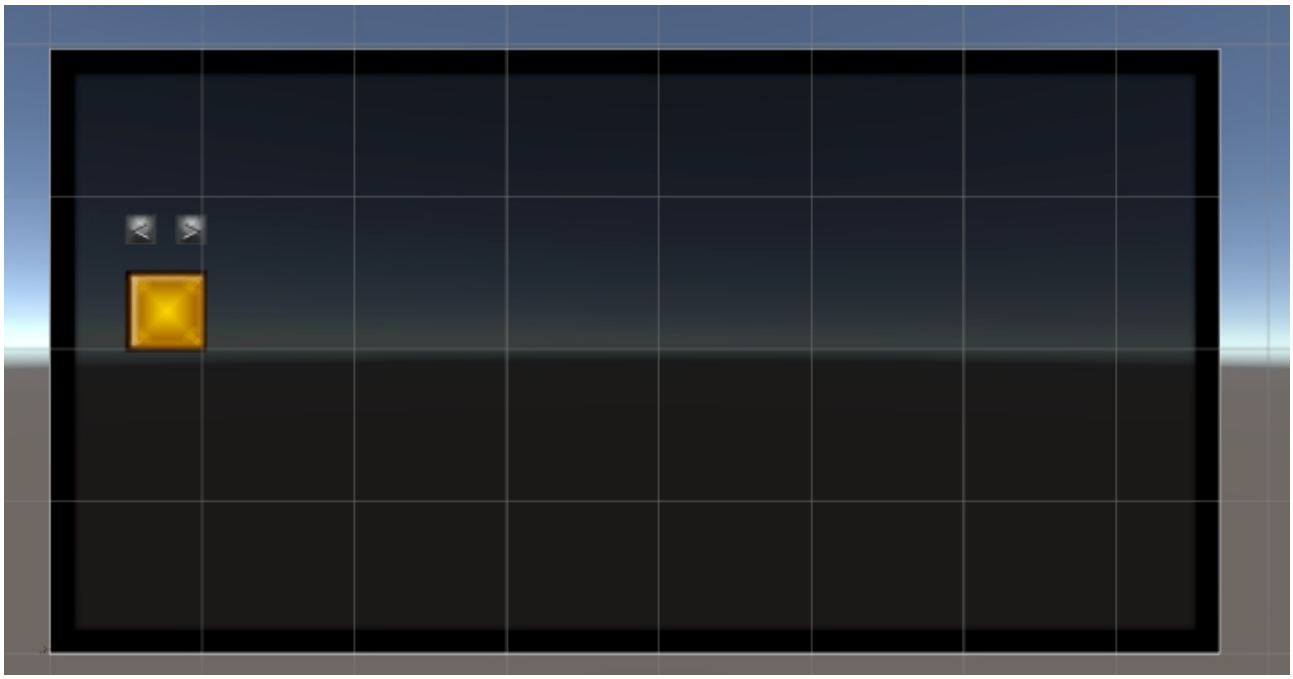


HealthPosition (Image), Stamina Bar (Slider), Money Position (Text), Hotkeys Position (Button) and the HUDCanvas (Canvas) itself are all that you need to drag into HUD.cs on your Handler game-object for the HUD system to work.

If you don't want some of these displayed, best leave them un-deleted and just untick the appropriate boxes under HUD.cs on the Handler game-object in the Inspector. This will exclude certain elements from being drawn.

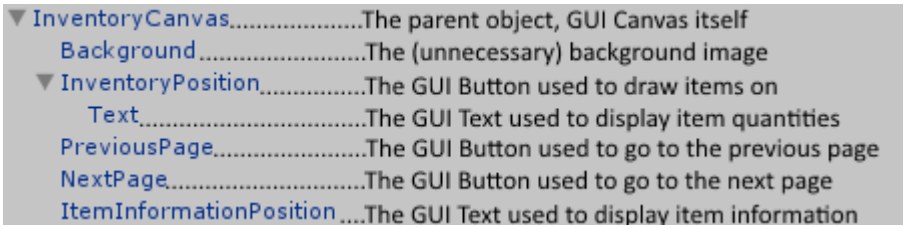
The Inventory Canvas Prefab

Introduction:



The same principles as with the HUD Canvas prefab go for this prefab. All the elements can be repositioned anywhere and the canvas itself can be set to have any type of scaler or rendering mode that the user wishes. All that Inventory.cs does is takes in GUI objects and gives them functions and draws out what the user has specified. All this canvas needs is a position for items (Button), a 'previous' button (Button), a 'next' button (Button) and a text area for item information (Text).

Hierarchy



Inventory Position (Button, Text), PreviousPage (Button), NextPage (Button) and ItemInformationPosition (Text) and the InventoryCanvas (Canvas) itself are all that the user needs to drag into Inventory.cs on the Handler game-object (with the tag “Handler”) for the Inventory system to display correctly.

If you do not want to use the Inventory feature, simply untick 'Use Inventory' under HUD.cs on the Handler game-object and when the player presses start, the game will only pause but not display the Inventory and the hotkeys will not be drawn on the HUD at all.

The Item Prefab

Introduction:
The Item Prefab, well, prefabs are basically 'representations' of items. They do not initially exist within the scene but are objects with the 'Item.cs' script attached to them which carry their item name, item information (that is displayed on the inventory menu), item icon (that is also displayed on the inventory and on the hotkeys), whether the item uses quantities, and, if so, the quantity and maximum quantity.

How do I create an Item?
Simply duplicate one of the item prefabs (recommended: the 'Item (Generic)' prefab) and change the fields in the inspector. Have fun with it and be creative.

How do I use this item within my game?
In the Handler object that you have added to your scene (read the '**The Handler Prefab**' section of this User Manual if you haven't already) you will see a field named the 'Item Dictionary' as part of Inventory.cs. Click on its drop-down arrow and either change one of its elements to your new item, or increase the number of elements (the 'Elements' field) to add it to your existing item dictionary. Think of this as the characters arsenal. Then, once it's added, simply decide whether the character 'hasItem' or not. This way, in your game, you can just set it to true once the character *finds the item* for it to appear in the inventory. While the item is not 'hasItem', its corresponding space in the inventory menu will be blank.

How do I actually USE the item?
When the item is showing in the inventory, you can either select it by left-clicking on it OR you can use the D-Pad on your gamepad (or alternatively the IJKL keys) and then press the corresponding button on your gamepad/key for it to be assigned to that button (the current button mapping is 'Right Bumper' for the first hotkey, 'Top Button' for the second and 'Right Button' for the third; or alternatively use the '1', '2' and '3' keys). Once the item is assigned to a hotkey, unpause the game and use that hotkey by pressing '1', '2' or '3' or the corresponding buttons on the gamepad. If the item doesn't have a quantity then nothing will happen; if it does have a quantity then it will decrease by one.

Why doesn't the item actually do anything?
It is up to the user to decide what happens with these weapons (well, this is how it stands prior to version 2.0). The UseItem() method of the Inventory.cs script starting at line 202 is where the items are being 'used' and I have annotated where each item is being used and where to insert custom item usage code. My tip is to create a prefab with a rigidbody, instantiate it here at the characters position and set its velocity in the direction of the characters forward. Experienced coders should have fun putting in what they like here, but beginners may want to wait for Version 2.0.

Anything else I should know?
Prior to version 1.5, the items being 'used' affected the prefabs themselves. Now, however, the items are instantiated as objects in the scene (though you cannot physically interact with them), so changes to any values are run time are no longer persistent and do not affect the prefabs themselves.

The Pickup Prefab

Introduction:
The Pickup prefab is basically a collectable that will increase the amount of certain values. They each contain the 'Pickup.cs' script which carries 'pickup type' and the 'value'. Then, once its collider trigger has been triggered, it will disappear and add its value accordingly (and for the sake of testing, it will reappear after five seconds).

How does I make my own?
Just drag the prefab into your scene view or the hierarchy, then change its 'pickup type' to the name of an item (it has to be EXACTLY the same as the 'item name' field of the item whose quantity the user wishes to replenish) or to 'money' or 'health'. Then enter a value and this will work accordingly as soon as the character walks through it.

Why does it reappear?
I mainly added that in for testing, just so that the editor game window didn't have to keep being reset to test the same pickup. I have clearly annotated what should be removed in the 'Pickup.cs' script for this to stop happening. Remove any instances of the 'cannotPickUp' field and remove the use of the 'RespawnPickUp' IEnumerator function.

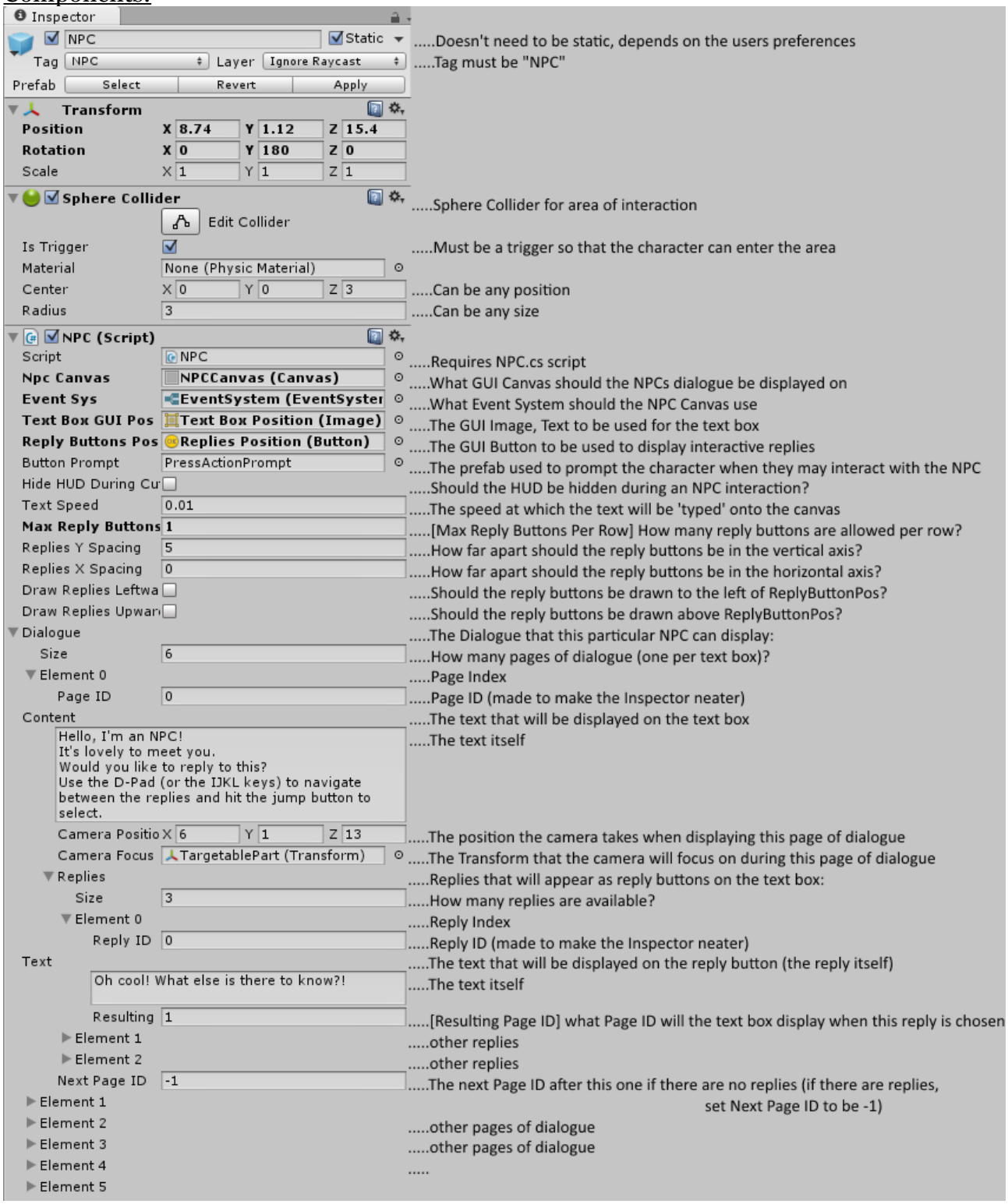
The NPC Prefab

Introduction:

The NPC (Non-Playable Character) prefab is a game-object that the character can interact with to display text boxes containing dialogue. This prefab uses the NPC.cs script which should be attached to every character that the user would like to use as an NPC.

The prefab interacts with the NPC Canvas (explained further in the **NPC Canvas Prefab** section below) to display the dialogue, replies, etc.

Components:



When in the Sphere Collider trigger, the NPC.cs script checks that the character is facing the NPC game-object and displays a prompt ('Button Prompt') when they are able to interact with the NPC.

'Dialogue' and 'Replies' are custom classes that act as the dialogue content. This allows the user to create all of their cutscenes purely through the Editor as they can write the content, possible replies, the resulting page IDs (e.g. clicking on Reply 2 has the text box display page 3), what the camera looks at during this particular dialogue page and where the camera is exactly.

How do I create my own NPC?
Simply drag in the prefab or just create a game-object with a model (and what not) and attach the NPC.cs script to the object and a Sphere Collider trigger to act as the area of interaction with the NPC. The NPC.cs script in the Inspector is where all the magic happens.

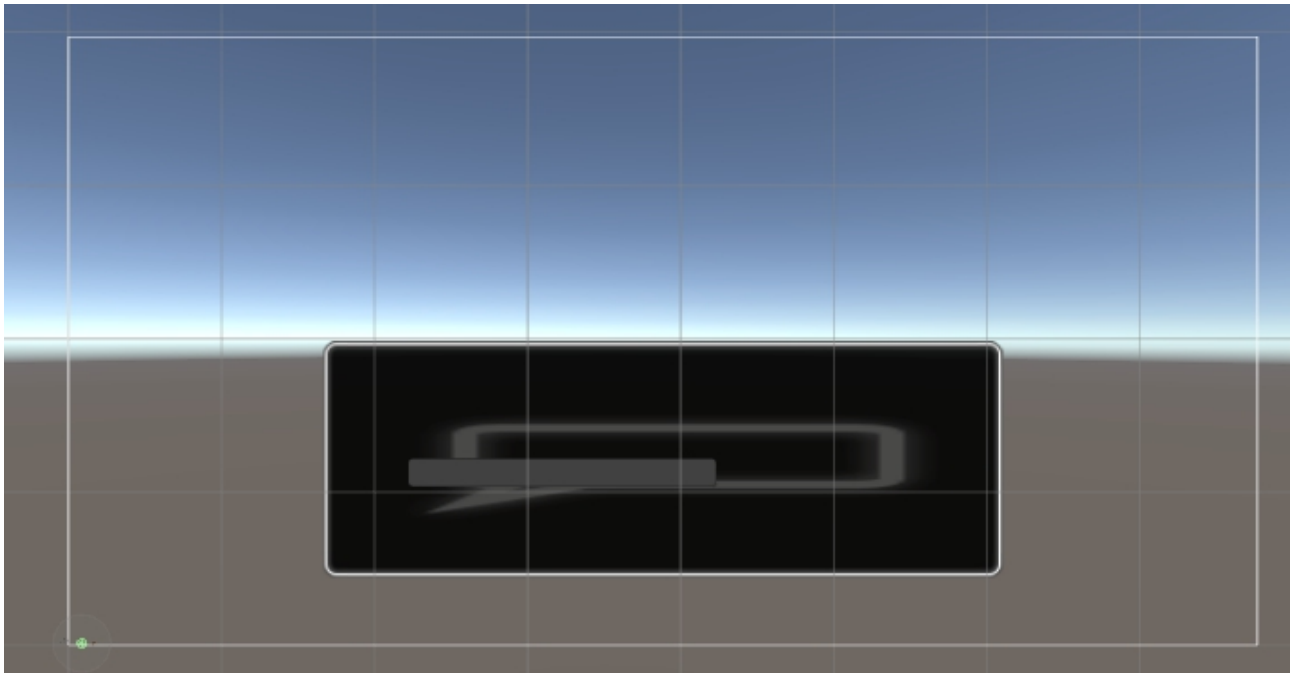
So how do I make my own dialogue for the NPC?

1. Drop down the arrow next to 'Dialogue' in the Inspector. Increase its size to 1, and there, you've made a 'page'.
2. Set the 'Page ID' of this page to be anything higher than '-1'.
3. Fill out the 'Content' part to be whatever text you would like to be displayed in the text box. Settings regarding font should be set on the GUI Element itself.
4. Set the camera position to be any position in world space.
5. Drag in any Transform from the scene view that you would want the camera to look at while displaying this page of dialogue.
6. If you want this to be the last page of dialogue then set the 'Next Page ID' to be '-1'. If you want it to go to another page but not have any replies, set the 'Next Page ID' to be the 'Page ID' of another page of dialogue (to create another page of dialogue, increase the size that was mentioned in Step 1 and give the new page a different 'Page ID'. If you want this page of dialogue to have replies, then click the drop down arrow next to 'Replies' for this page and increase its size to as many number of replies as you wish.
7. [Read on only if you've made a reply] Set the 'Reply ID' to a number higher than '-1'.
8. Set the 'Text' to be the reply itself: The text that will be displayed on the reply button in the text box.
9. Set the 'Resulting Page ID' to be the 'Page ID' of one of the pages existing in this 'Dialogue'.

Please continue on to the '**NPC Canvas Prefab**' to learn how to use the NPC Canvas with the an NPC.

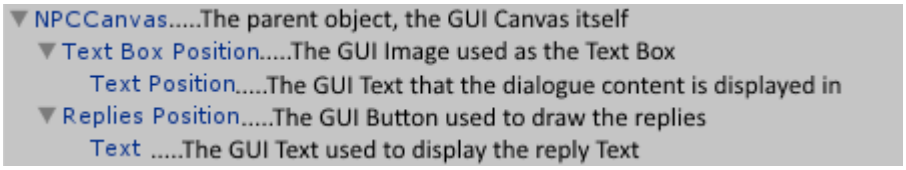
The NPC Canvas Prefab

Introduction:



The same principles as with the HUD Canvas prefab go for this prefab. All the elements can be repositioned anywhere and the canvas itself can be set to have any type of scaler or rendering mode that the user wishes. All that NPC.cs does is takes in GUI objects and gives them functions and draws out what the user has specified. All this canvas needs is a position for the Text Box (Image, Text), and a button for the Replies (Button, Text).

Hierarchy:



Text Box Position (Image, Text), Replies Position (Button,Text) and the NPCCanvas (Canvas) itself are all that the user needs to drag into NPC.cs on any NPC game-object for the dialogue to display correctly on it.

Read the '**NPC Prefab**' section of this user manual to understand how to use the NPC.cs script on an NPC game-object to display onto the NPCCanvas.

If the user doesn't wish to display the HUD during an interaction with an NPC, simply check the 'Hide HUD' boolean on an NPC.cs script in the Inspector on an NPC to do so.

If the NPC functionality isn't desired at all, simply don't include any NPC game-object, nor the canvas in the scene.

Useful Script Functions

Introduction:
Not everything can be done in the Unity Editor and so when it comes to making your game, there are certain methods/functions in the scripts to help make the game-making process a bit easier.

ThirdPersonController.cs
[BOOL] isPaused : Set true if you would want the character to remain still.

HUD.cs
[VOID] AcquireNewHeart() : Add a new heart to the HUD and 4 to the maximum health.

[VOID] UpgradeMoney(int newAmount) : Increase the wallet size by 'newAmount'.

[VOID] HideHUD() : Will stop the HUD from displaying.

[VOID] DrawHUD() : Will display the HUD.

[IEnumerator] ChangeHealth(int amount) : Change the health by 'amount' (can be positive or negative).

[IEnumerator] ChangeHUDBarValue(int hudBarIndex, int amount) : Change a specific bar in the HUD by 'amount'.

[IEnumerator] RegenerateStamina(int hudBarIndex) : Have a certain hud bar completely regenerate.

[IEnumerator] ChangeMoney(int amount) : Change the amount of money by 'amount'.

Inventory.cs
[VOID] UseItem(string itemName) : The user would need to add their own code here in this method if 'itemName' == 'name of an item'.

Importing Your Own Character

Introduction:
As the user, if you're happy with the character prefab as it is and you're ready to put in your own character model, then this is how you import your own character. There are two ways to do this; one involving your own imported/ready model (easiest method) or by using the .blend file I have included in a .zip package in this asset.

TIP: Be sure that for each individual mesh for your character model that it is set to 'Update Off Screen' in the Inspector in the Skinned Mesh Renderer component. And check that your Animator is set to 'Always Update'. This tip is just to make sure that First-Person View mode functions properly.

Option 1: Swap in your own imported model
New to version 1.5, you can simply just set your model (along with its armature and meshes parented to it) to be a child object to the character game-object. Then, just drag in the appropriate Transforms into the Inspector for ThirdPersonController.cs and you're ready to go! Just be sure to give your model the 'PlayerAnimController' in the folder 'Animations'>'Player' and check that each of the Animations (in the Animator window in the Editor) are the correct animations (head to the '**Using Your Own Animations**' section of this user manual to check that you've got all the animations you need).
No need to alter any of the lines of code like in the previous versions.

Option 2: Edit the .blend File (Basic knowledge of Blender needed)
Since Blender is a free modelling software, anyone can use this method. The .blend file for the Proto model is zipped up in the asset package. Simply unzip it and open the file in Blender. Change the meshes to your own meshes/character and parent the bones to the mesh (in Object mode, have the mesh and armature selected, press Ctrl+P and select with 'automatic weights'). In the case of more than one mesh (like mine), select all the meshes.

When ready, go into object mode and press A to select everything. Then go to export the file as .fbx. On the far left of the screen, under 'Export FBX', check 'Selected Objects', then, above 'Apply Modifiers', select (out of the 5 options) 'Armature' **and** 'Mesh'. Finally, make sure 'Include Animation' is checked and 'Include Default Take' is unchecked. If your mesh is using a texture, make sure to save your texture image file in the 'Textures' project folder.

Import the .fbx file into Unity. Select the .fbx file in the 'Project' window and look at the inspector. Under the 'Model' tab, set the scale to '20' (or whatever gets your model to appear properly in the game). Under the 'Animations' tab, make sure that the correct animations are looping (head to the '**Using Your Own Animations**' section of this user manual and follow the steps there).

Click apply and the 'Character' prefab's model-child-transform should be updated. If not, set the .fbx file as a child object to the character prefab, in replacement of the original 'Proto' child object. Give the new model-child object an Animator Controller, and in the Inspector, make the Controller field to 'PlayerAnimController' in the folder 'Animations'>'Player'.

Finally, check the animator window to make sure that the states have the correct animations and now everything should be working with your own new character model.

Using Your Own Animations

Introduction:
The animations used in this asset were made to fit in with the programming. Using your own animations may have some be too short or too long for the action that the character is doing (mainly referring to the ledge-climbing animations); also, there may be animations that you may not want in the asset and there are some animations that work differently to others. This section will go over the timing that the script uses, where to find certain animations in the code and which animations function differently to others.

List of animations:
Here are a list of all the different animations that the character uses (ones marked with an '(L)' are looped):

-Backflip	-GrabbingLeft (L)	-RunRight (L)
-Backwalk (L)	-GrabbingRight (L)	-SideHopLeft
-ClimbDown	-GrabPull	-SideHopRight
-ClimbDown2	-GrabPush	-SidestepLeft (L)
-ClimbLeft	-GrabStill (L)	-SidestepRight (L)
-ClimbLeft2	-Idle (L)	-SlowSwim(L)
-ClimbRight	-Jump	-Surfaced
-ClimbRight2	-JumpToGrab	-SwimFlip
-ClimbStill (L)	-LowJumpLdg	-SwimImpulse
-ClimbStill2 (L)	-LowLdg (not actually used)	-SwimStill (L)
-ClimbUp	-MidLdg	-Sliding (L)
-ClimbUp2	-Paddle (L)	-Walk (L)
-Diving(L)	-PullUpLdg	-Brake
-Fall (L)	-Roll	-Crawling (L)
-Floating(L)	-RollKnockBack	-CrouchDown
-GrabBeneath	-Run (L)	-Crouching (L)
-Grabbing (L)	-RunLeft (L)	-DoubleJump
-Hardland	-RunningLand	-SecondJump
-Sprint (L)	-SprintLeft (L)	-SprintRight (L)
-SidewaysJump		

NOTE: Under the 'Animations' tab in the inspector when viewing an imported model, select 'Import Animation' and make sure that the correct animations are set to loop.

Animation Timing:
At the bottom of the ThirdPersonController script are two IEnumerator Co-Routines: Delay() and LedgeHandler(). Delay() sets certain bools to false after a certain amount of time, and after that time, the character performs a different action and so plays a different animation. LedgeHandler() positions the collider at the position of the model after playing a specific ledge climbing animation. These use the 'WaitForSeconds(float amount)' functions which are set to specific amounts which are in time with the animations.

If you use your own animations/edit existing ones and are longer than the original animations, you may have to tweak these times in order to match the length of your animations.

Animation Types:
There are three different ways that the character is animated. One is using a stationary animation while the character moves through script. Another is using a blend tree to blend between different animations that a variants of the same action. The other is using an animation that moves away from the character's collider which stays still.

The stationary animation is by far the most common type used in this asset. Since the majority of the movement is physics based or is scripted a certain way, all the animation has to do is perform the action in one spot while the collider moves it (the model) around (e.g. Jumping, falling, rolling, etc.).

The blend tree animations use the same logic as the one mentioned above but blend between different speeds/directions of the same action. The grabbing movement (staying still, grabbing along the ledge to the left/right) and the locomotion animations (walking to running, depending on speed; running while leaning left or right depending on how sharply the character's turning).

Finally is when the animation moves independently from the collider; this is only used on the ledge climbing animations (midLdg, pullUp and grabBeneath). The collider is programmed to stay perfectly still and the animation of the model moves beyond the position of the collider. When it reaches the end of the action, the collider is moved straight to the position of the model (to the models root bone).

What you need to keep in mind:
So when you're making your own animations, be sure to keep all of the animations apart from midLdg, pullUp and grabBeneath stationary. Keep the root bone in one place, or at least keep it very close to the original position (it can move out slightly, such as the rolling animation which is lower than the original position and the running animations which dart backwards briefly to give the illusion of stepping).

Standard Assets Used:

Introduction:
Some parts of this asset now include uses/edits of the free Standard Assets that are available to Unity 5.1 owners.

Assets Used:

>Effects-
    >Image Effects-
        >Scripts-
            -BloomOptimized.cs
            -PostEffectsBase.cs
        >Shaders-
            >_BloomAndFlares-
                -MobileBloom
    >Projectors-
        >Materials-
        >Prefabs-
            -BlobShadowProjector
        >Shaders-
        >Textures-

>Environment
    >Water
        >Water [Folder]
        >Water4 [Folder]

Uses
The Image effects are used by the Main Camera and are not edited/modified in any way.
The Environment assets are used in the 'Water Box' prefab to create the water mesh (along with all its effects) to sit atop an empty collider with the attached 'WaterForce.cs' script.
The BlobShadowProject is used as a secondary shadow for the character for better FPV jumping precision.

Miscellaneous Project Files:

Scenes Folder:
This folder will have a demo scene in it, with a working character and camera in a test stage to play about in.

Materials Folder:
This contains materials applied to the meshes in the scene to give some detail to all of the blocks and shapes. It also has the material that the Ridicule uses.

Physics Materials Folder:
This contains the physics material applied to the characters capsule collider. It's just a material with zero bounce and friction to ensure that the character moves only in the way it's programmed and isn't acted upon by unnecessary physics.

Shader Folder:
Contains a customer shader, used for the WaterCube prefab so that the water on the outside of the cube can also be seen from the inside.

When It's All Working...:

Is there anything I can get rid of?
Of course there is! Wherever you see an if-statement checking if an object == null, then leads to code that uses Debug.Break() and Application.Quit(), you can get rid of these as these are only nullpointer counters that were made for ease of use for you, the user. You can keep them in there, but if you plan on building the project **there's something you must get rid of**. Search for all the lines that include (not including speech marks):

"REMOVE THIS FOR BUILDING"

Make sure to remove the lines between these phrases containing references to the Unity Editor as a built project is not using the Unity Editor and will fail to build if this line remains. This is only used to stop the editor from playing once there has been an error.

While you can now turn off every feature purely through the Inspector, you can also remove any of the ledge-climbing, grabbing, jumping, rolling actions in the ThirdPersonController.cs script. Most of the code specific to the actions is sectionised, so it should be easy to get rid of a vast majority of each feature one chunk at a time. There's usually a chunk in Update() and a chunk in 'FixedUpdate()'. Just hit Ctrl+F and search for any key words and it should take you to the chunks. There will also be bools scattered throughout the code so be sure to search for (for example) 'rollDelay' and delete any instances of it in the scripts if you want to remove the rolling feature.

Use this asset to create your own platformer, fan game, spin-off, first-person-shooter, any game you can think of! I hope this helps you create your dream game, and remember that my contact information for support is stated in the Documentation PDF file. This script is free to use in a commercial product; it is bound by the license agreement that is given to all Unity assets on the asset store and is ONLY to be used by the person that has bought the asset, even if it has been heavily modified. This asset is not to be redistributed in any form.

Have fun and go wild! If you come up with any cool projects using this asset, I'll be more than happy to hear from you.

Support / Helpful Links:

Input Manager Asset Download Link:
http://www.weebly.com/uploads/5/1/7/2/51725187/inputmanager.asset

Contact Me:
http://zeliggames.weebly.com/contact.html