# uSurvival Documentation


(Asset Store) (Support Email) (Support Discord) (Forum)

(Online Documentation)

# Table of Contents

## Getting Started

To see **uSurvival** in action, simply open the included scene from **uSurvival/Scenes**, press **Play** in the Editor and select **Server & Play**. Run around a bit, pick up some weapons and drag them to your Hotbar at the bottom of the screen to try them. Kill some Zombies, climb the ladder, etc.

Once you got a feeling for it, now it's time to get some basic project overview and learn some new stuff. Recommended first steps:

- Inspect the Prefabs/Entities/Players/Player prefab. Take a look at all the components to see how much you can modify easily, without writing any code.
- Look at the Resources/Items folder and play around with the items. You can modify stats and icons easily without writing any code. Duplicate the Banana and modify it to be an apple. Then select one of the Item Spawners in the Scene and assign it.
- Inspect the Hierarchy/Scene. Add a Zombie by duplicating one, then move it somewhere else. Move around some of the Environment. Go to Window->Navigation and rebake the Navmesh afterwards. Zombies need it to move around.
- Read through the UNET manual to understand Unity's Networking system.
- Read the rest of this documentation to understand all the components.

## Recommended Unity Version

Right now, the recommended Unity version is Unity 2018.2. You should not use an **older version** because Unity is not downwards compatible.

You can use **newer versions** at your own risk. uSurvival is a big project that uses a lot of different Unity features, so whenever Unity introduces a new bug, we feel the effect very significantly. In theory, any newer Unity version should work fine if (and only if) Unity didn't introduce new bugs. That being said, it's common knowledge that Unity always introduces new bugs to new versions.

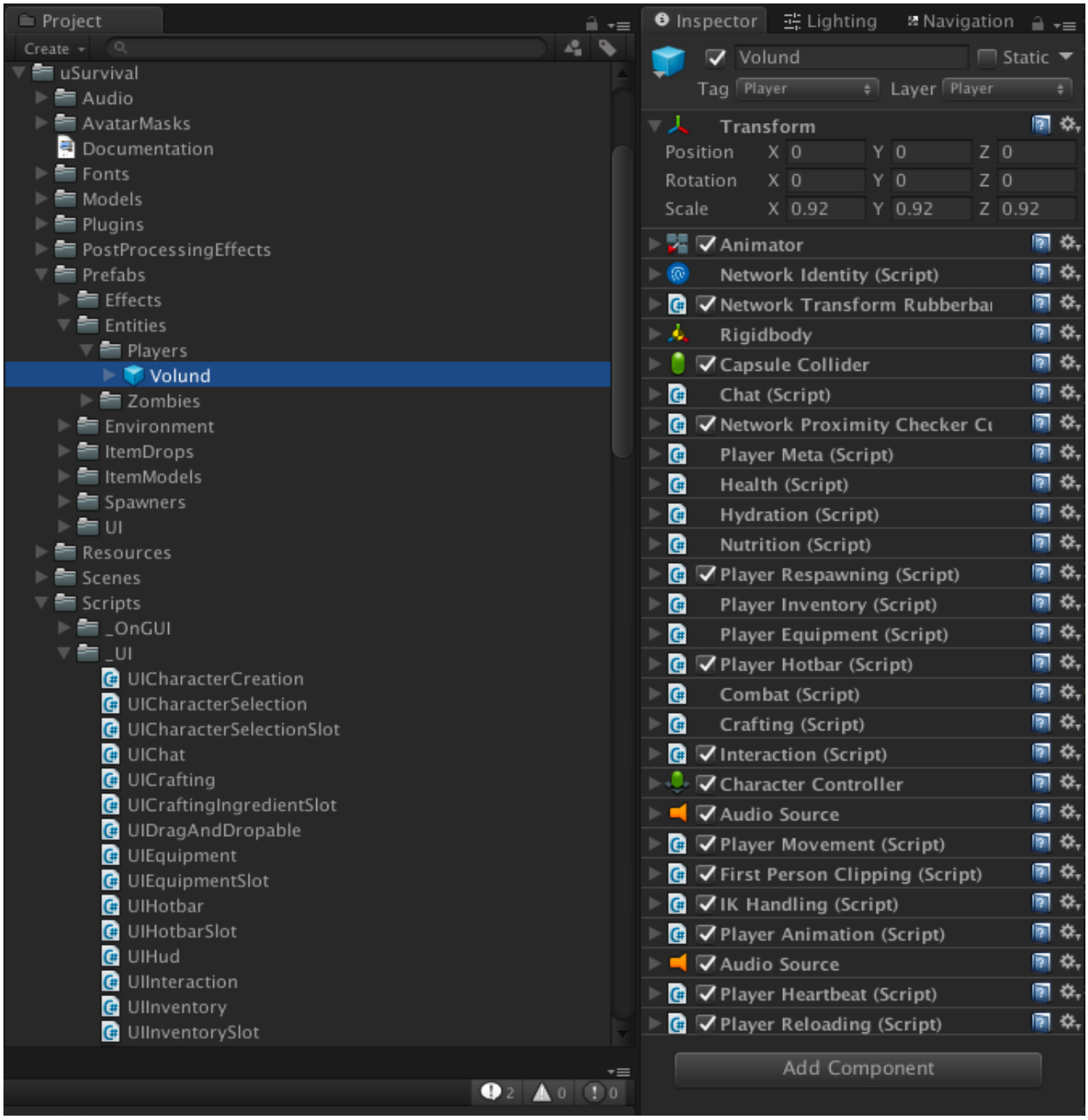
During the past few years working on uMMORPG, the process was to download Unity, encounter bugs, report them, upgrade Unity to the newer version where the bugs were fixed, only to encounter new bugs, and so on. It was a never ending cycle of upgrading headaches, with only the occasional stable version in between.

During GDC 2018, Unity announced the **LTS release cycle**. LTS stands for long term support. Unity LTS versions are supported for 2 years and will only receive bug fixes, without introducing any new features (and hence new bugs).

Words can hardly express how significant LTS versions are for a multiplayer game. You should absolutely use LTS at all times, otherwise your players will suffer from bugs and all hell will break loose.

## Players

The Player prefab(s) can be found in the Prefabs/Entities/Players folder:




Players have a whole lot of components attached to them, generally one per feature.

You can modify a whole lot without writing any code, simply browse through the components in the Inspector to get a feel for it.

You can of course add your own components to player prefabs too.

## Player Movement

The PlayerMovement and NetworkTransformRubberbanding components are the only two components that are client authoritative. The reason is very simple too: evaluating every movement request on the server first would cause too much of a delay to feel smooth on the client.

Players are moved with Unity's CharacterController.

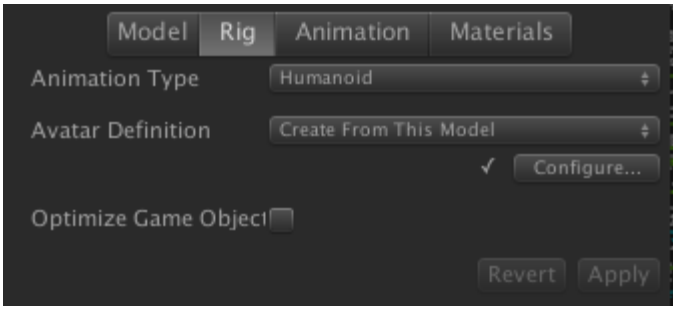
- They are not NavMeshAgents because then they couldn't jump or climb ladders.
- They are not Rigidbody driven because physics movement is incredibly difficult to get right, even in single player games.

If you ever played games like Counter Strike or Quake then you will greatly appreciate the CharacterController movement. It's not 100% physically correct, but it sure feels really, really great.

## Creating new Players & Modifying Player Models

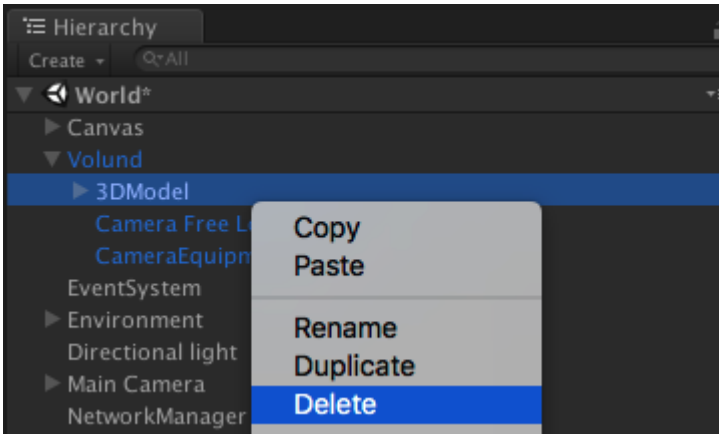
The player model is easy to modify. All we have to do is swap out the 3D model and assign a couple of components, so that uSurvival knows where the hand/head/etc. are. Let's do it step by step with the [Space Robot from Unity].

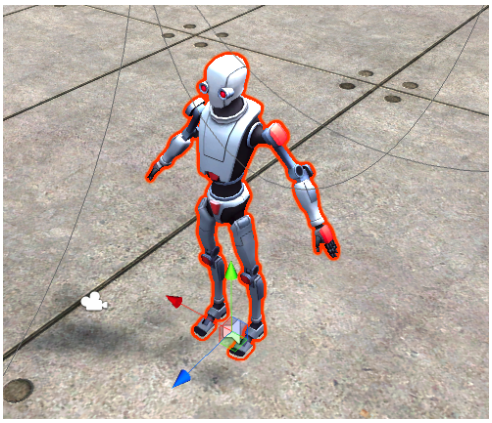
1. **Import** the model from the Asset Store:
   a. 

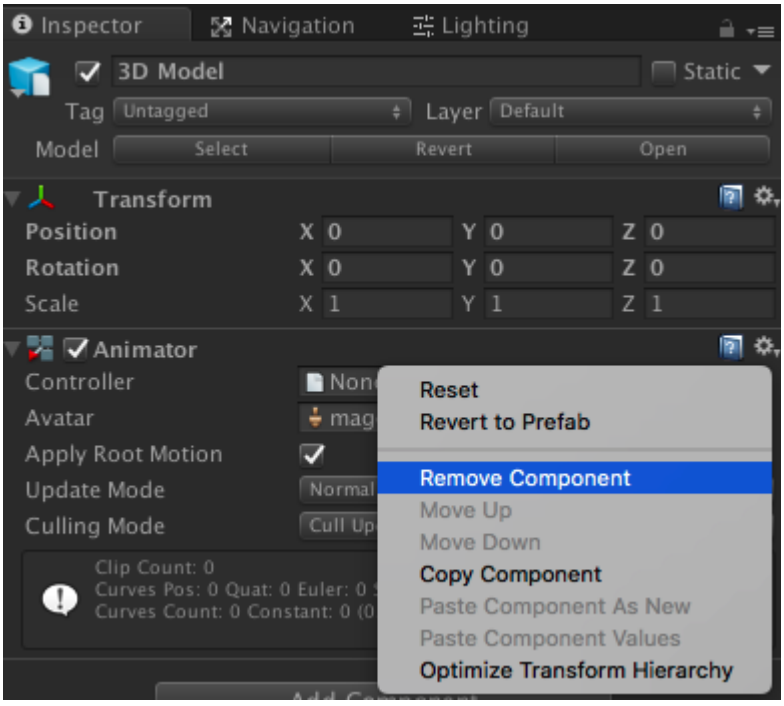
2. Find it in the project area, then in the Inspector select **Rig** -> **Animation Type**, set it to **Humanoid** and press **Apply**. Otherwise we can't retarget animations.
   a. 

3. Drag the existing player prefab into the scene:

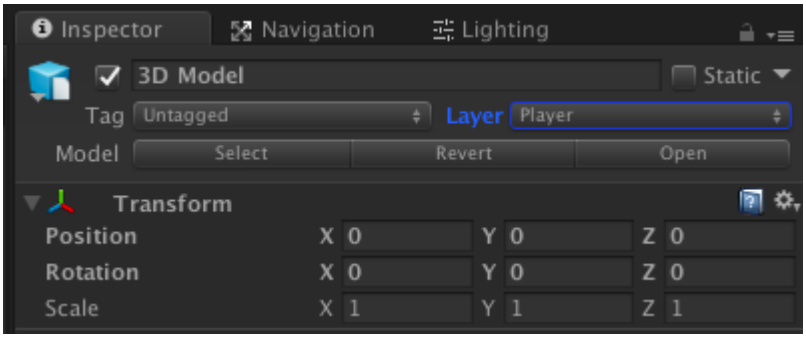
a. 

4. Select it in the Hierarchy, delete the old 3D Model:
   a. 

5. Drag the new one into the Volund GameObject, rename it to **3DModel** for consistency
6. If the model appears too big, then select the file again and modify the **Scale Factor**. In this case, the default scale works fine:
   a.
   b. 
7. Remove the Animator component from the model:

a. 


8. Set the Layer to Player

a. 

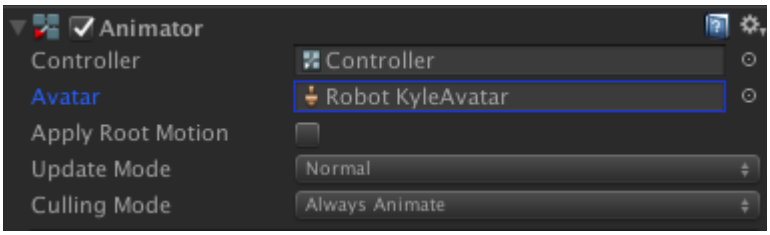

9. Find the **Hips** (or Pelvis) Bone in the Hierarchy.

a. 

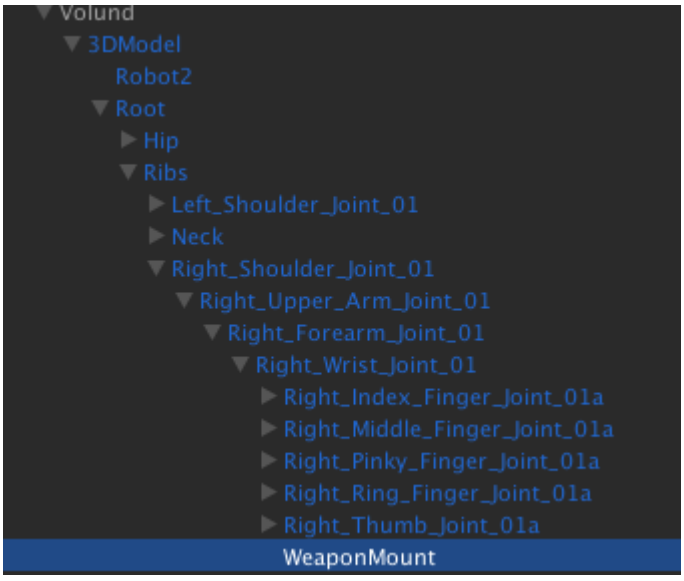

10. Add a **CapsuleCollider** component to it. Set **Direction** to which ever **Axis** lets you scale the height vertically *(X-Axis in our case)*, enable **IsTrigger** and **scale** it so that it fits the model:

a. 

11. Add an **AudioSource** component to the Hips *(for footstep sounds)*.
12. Select the root GameObject *(Volund)*, find the **Animator** component and drag the new model's avatar into the avatar field:

a. 

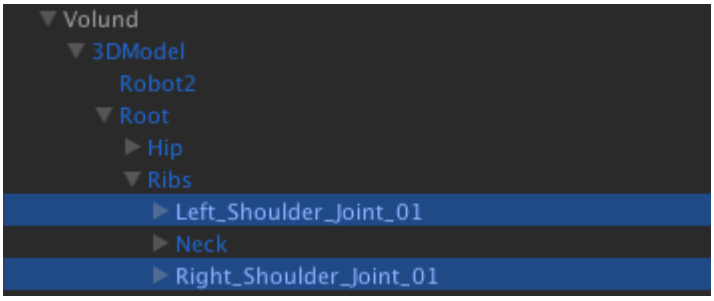

13. Assign the **PlayerEquipment** component's **Weapon Mount** slot to an empty WeaponMount GameObject inside the player's right hand. If it doesn't have one, then add it:

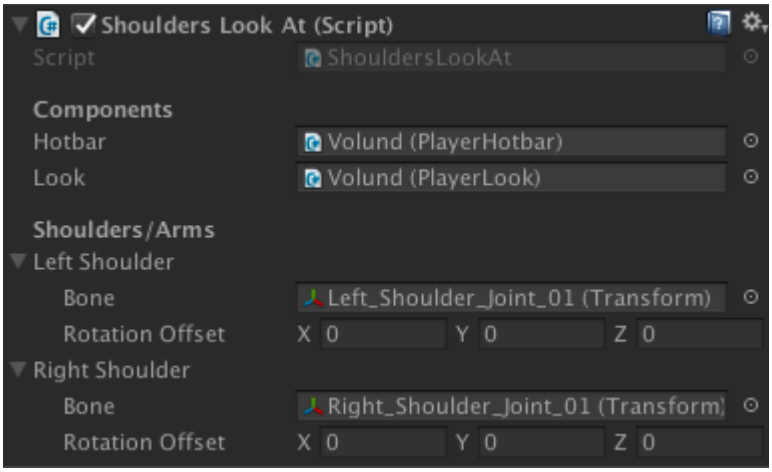
a. 


b. The **WeaponMount** can be used to adjust the weapon position in the player's hand. If your character holds the weapon in the wrong direction / position later, then adjust the weapon mount position & rotation. In our case, the following works perfectly:

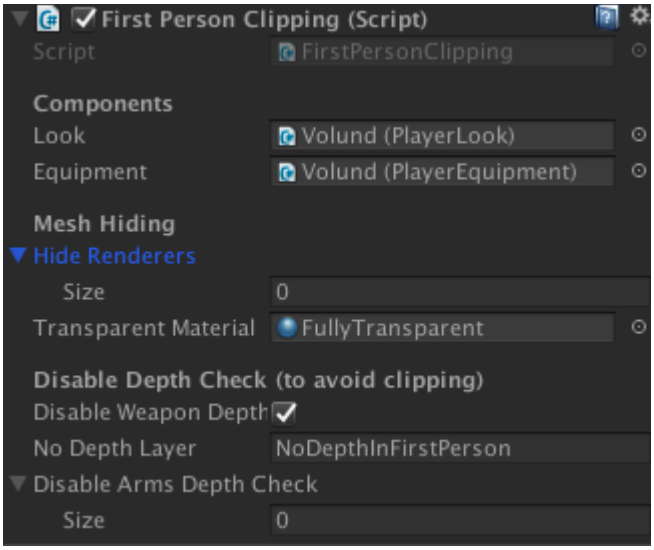
c. 


14. Select the **ShouldersLookAt** component and drag the model's shoulder bones into the **bone** slots. *(If the arms behave strangely while holding a weapon later, then try the arms bone instead of the shoulder bone)*:

a. 


b. 


15. Select the **FirstPersonClipping** component and clear the **Hide Renderers** and **Disable Arms Depth Check** arrays by setting their **sizes to 0** for now. *(You can later drag in parts that should be hidden/shown in first person only)*

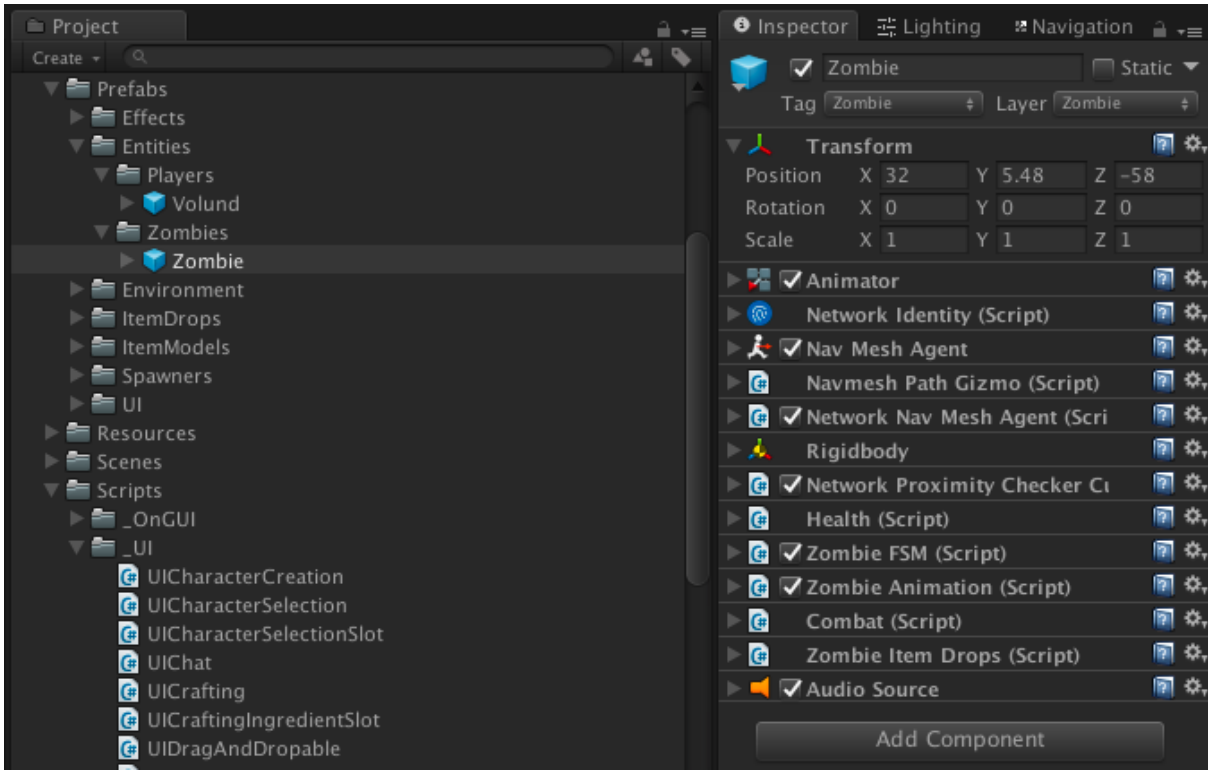
a. 


16. Select the root GameObject *(Volund)* again. Take a look at the Inspector and assign the now missing fields to the parts of the Model
    a. PlayerMovement **Feet Audio** to the hips AudioSource
    b. PlayerMovement **Collider** to the hips CapsuleCollider
    c. PlayerLook **Capsule** to the hips CapsuleCollider
    d. PlayerLook **First Person Parent** to the head bone *(ideally create an empty GameObject in the Head and move it to the center between both eyes, then use this one. Otherwise first person view might be too far behind)*
    e. **Important:** *If you are getting errors when pressing Play then check all components for missing fields, and assign where needed.*
17. Afterwards rename it to **Robot**, drag it into the Prefabs folder to save it as a different character, and drag it into the **NetworkManager**'s spawnable prefabs.
18. **Remove** the player GameObject from the Scene, otherwise strange errors will appear if a networked player prefab still hangs out in the scene when we start.
19. Press **Play**, create a new character of class **Robot** and try it out:
    a. 

**IMPORTANT:** the Space Robot model was created by a professional 3D artist, and everything worked perfectly. Some poorly created models will have strange bone structures or weird shoulder rotations, which often cause the arms to end up in weird positions while holding a weapon. If this happens, the easiest solution is to ask your artist to fix the rig. Send your artist the space robot for comparison. Another option is to modify the ShouldersLookAt component's rotation offsets until it fits, but that's a very tedious process.

## Zombies

The Zombie(s) can be found in Prefabs/Entities/Zombies:




You can modify it to your needs, just like the Player prefab, so please check out the part on modifying players.

You can also take a look at this video that was created by a community member.

There is no complicated spawning system for zombies. Simply drag them into the scenes and position them wherever you want them to live.

### Zombie Movement

Zombies have a simple AI and they need to be able to navigate around obstacles, for example to follow a player into a building. This is very difficult to do, but thanks to Unity's Navigation feature, all we have to do is set the Zombie's NavMeshAgent.destination property.

So in other words, Zombies are NavMeshAgents on Unity's Navmesh. Make sure to rebake the Navmesh whenever you modify the game world.

## Zombie Behaviour

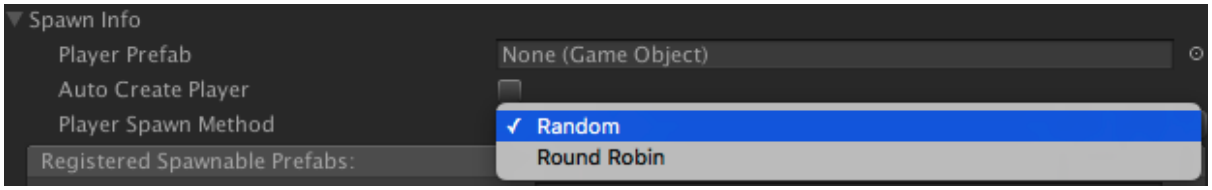
Zombies are driven by the ZombieFSM component, which stands for Zombie Finite State Machine. The behaviour is 100% code, so you'll have to modify the ZombieFSM script if you want to change it.

*Note: there's no easier way to do it with UNET at the moment, and it does work really well too.*

## Spawn Position

If you want to modify your Player's spawn position, simply move the Respawn point in the Scene / Hierarchy:


If you want multiple respawn points, simply duplicate one and move it somewhere else. UNET has different ways to select the Respawn point, you can modify that in the NetworkManager:



## Modifying the Environment & Terrain

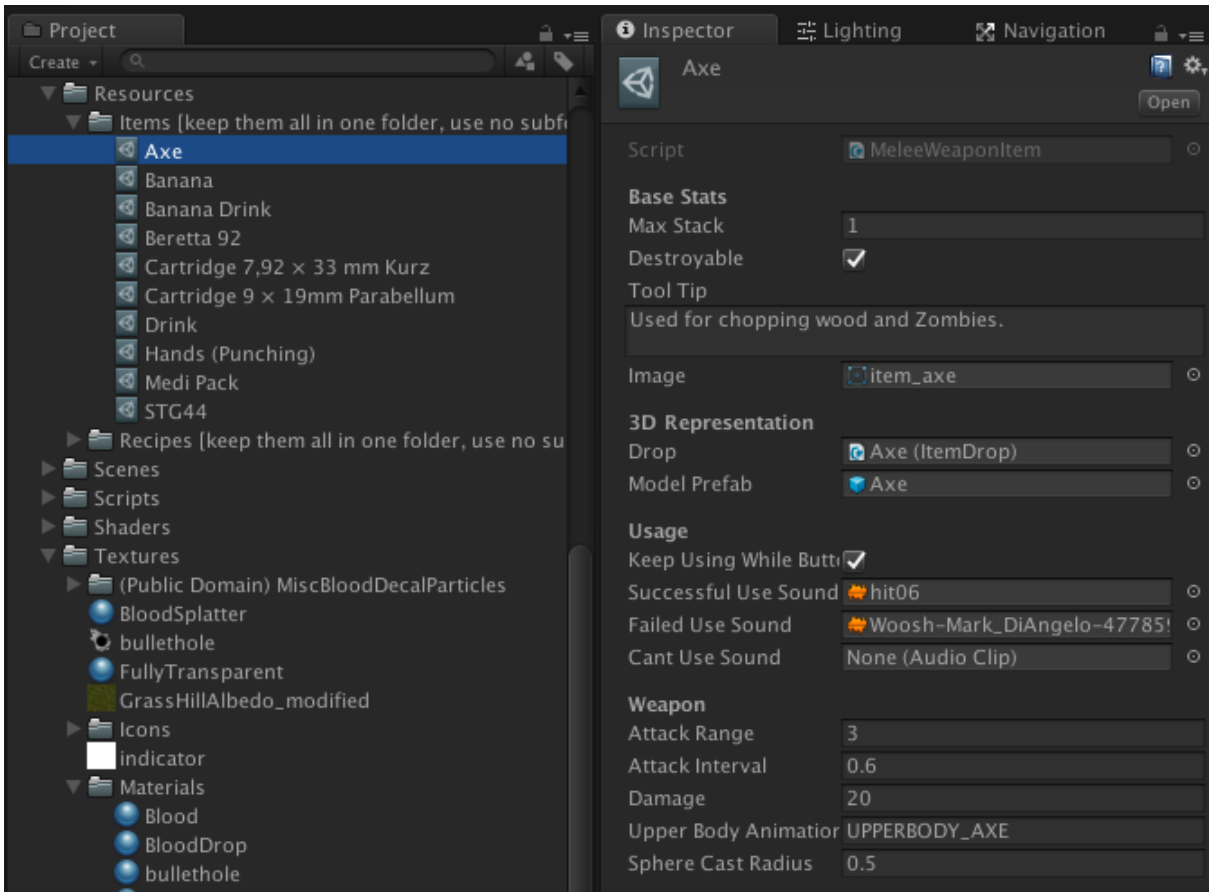
uSurvival uses a small and simple scene to display all of the features. You can of course add any 3D models or environment assets that you like, as well as Terrains. Simply make sure that:

- The GameObjects have decent colliders. MeshColliders are recommended, so that bullet holes show are displayed at the correct hit position, not on a rough estimation (e.g. when using a BoxCollider for a car)
- The GameObjects should be marked as Static
- And go to Window->Navigation and rebake the Navmesh each time you modify the environment. The Zombies need the Navmesh to move.

## Items

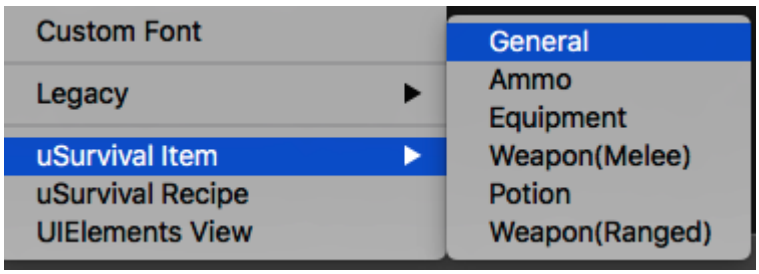
### Scriptable Items

We already added the most basic items types, so you can simply duplicate existing items to make similar ones:

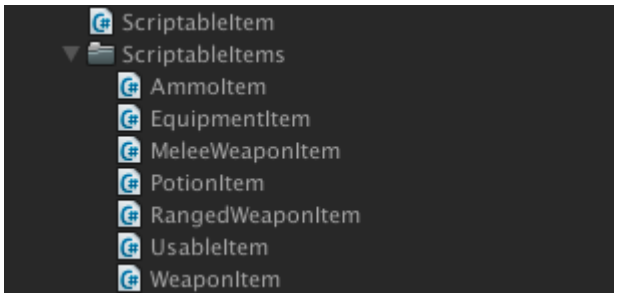



- You can duplicate and modify the banana to make an apple or a health potion.
- You can duplicate and modify the Beretta pistol to make a Revolver or a Rifle.
- You can duplicate and modify the axe to make a baseball bat.
- Etc.

You can also create new Items by right clicking in the Project area to select Create->uSurvival Item:




As you can see, there are already different item usage mechanisms. Food like a banana will be consumed to increase health/hydration/nutrition. Other items like the Beretta do none of that, but shoot bullets instead. Every Survival game will need all kinds of different item usage mechanisms, which is why we implemented **Scriptable Items**.




Please take look at the **Scripts/ScriptableItem.cs** file and the **Scripts/ScriptableItems** folder with the currently implemented Scriptable Item types. In most cases, you will want to inherit from UsableItem.cs. All you have to do is overwrite the **.Use()** function (and a few others depending on your needs) to add any item mechanism that you want. You could have an item that kills every Zombie on the server in .Use(), you could have a tent that is built into the game world, the possibilities are endless.

To create a new Scriptable Item type, simply create a new Script, inherit from ScriptableItem (or UsableItem if it's supposed to be usable) and then add your logic / properties as needed. Make sure to add a menu entry like this:

```
using UnityEngine;

[CreateAssetMenu(menuName="uSurvival Item/Test")]
public class TestItem : ScriptableItem
{
}
```

So that you can create an item of that type via right click.

## The Item Struct

We just talked about **ScriptableItem.cs**, which is the true 'Item' class in uSurvival. But there is also **Item.cs** - *what the hell?*

uSurvival uses UNET's SyncListStructs for the inventory and the equipment. Those SyncListStructs only work with structs, so we can't put ScriptableItem types in there. And that's a good thing, here is why:

- ScriptableItem has huge amounts of item data like the name, icons, reload times, etc.
- Syncing all that over the Network would require large amounts of bandwidth
- Clients already know the ScriptableItems anyway - there's no point in syncing them again. All they need to know is which Scriptable Item they need to refer too.
- The Item.cs struct is just that. It contains the name (a hash to be exact) to refer to the Scriptable Item. So all we need are a couple of bytes and the clients know which ScriptableItem is in which inventory slot, etc.

Note that all 'dynamic' Item properties like current Ammo are also in Item.cs - since two Items of the same ScriptableItem type might as well have different amounts of ammo in their magazine.

## ItemSlot

There is also the ItemSlot struct, which is just Item + amount. No magic here.

The Inventory and Equipment SyncListStructs work with ItemSlots.

A slot contains a valid .item if the .amount > 0. If .amount == 0 then the .item is invalid and should not be accessed.

## Item Drops


*Another Item class - what the hell?*

uSurvival needs Item Drops in the game world, like a Banana that lies on a kitchen table in an old farm house. We can't just drag ScriptableItems or Item/ItemSlot structs into the game world, they have no 3D representation whatsoever.

Hence why we also need an ItemDrop component. There is no magic here either. Those are just 3D models with an ItemDrop component which refers to the ScriptableItem that is rewarded to the Player after picking it up.

By the way, ItemDrops are NetworkIdentities with NetworkProximityCheckers so that everyone on the server sees them when close enough.

## Item Models


*Oh come on.. enough with the Item classes now!*

Okay, last one. If the player uses an Axe, we should see an Axe model in the player's hands. That's why we need Item Models. They are really just 3D models this time, nothing complicated.

*Note: we can't reuse ItemDrops here. They are 3D models too, but ItemDrops are NetworkIdentities with ProximityCheckers, and ItemModels are literally just that, models. UNET doesn't allow NetworkIdentities to be children of other NetworkIdentities, hence why we need Item Models too.*

## Combining them all

Let's say you want to add an apple to your game. Here is the step by step guide:

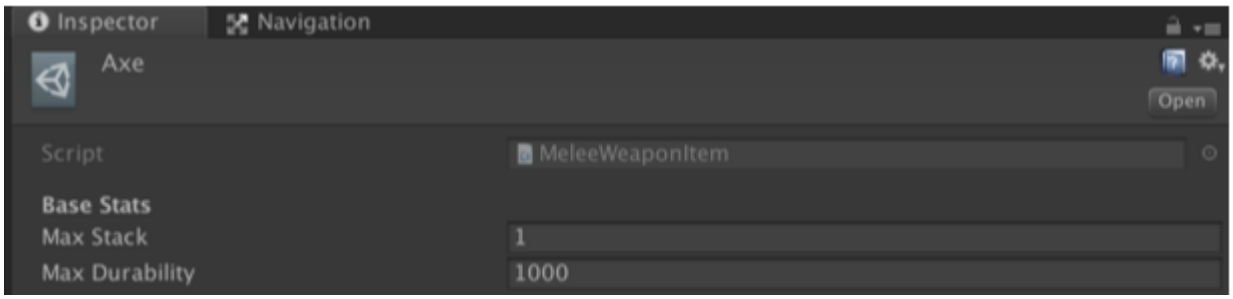
1. Duplicate the Banana ScriptableItem, rename it to Apple and give it an apple icon
2. Duplicate the Banana ItemDrop, rename it to Apple, drag it into the scene, replace the Banana 3D Model with an Apple model, drag it back into the folder to create a new Apple ItemDrop prefab.
3. Select your Apple ScriptableItem and drag the Apple ItemDrop into the ScriptableItem's itemDrop field, so it knows which drop to spawn into the game world when the player tosses it out of his inventory, etc.
4. Drag the new Apple ItemDrop into the NetworkManager's spawnable prefabs list, so that UNET knows how to spawn it.
5. Now add it to the game world by Duplicating one of the ItemDropSpawners in the Hierarchy, then assign the ItemDrop field to your Apple ItemDrop.

Now press Play, walk to your Apple, pick it up and find it in your inventory.

For tools / guns that the player can hold in his hand, you will also have to create the ItemModel (as usual, duplicate an existing one and modify it), and then assign the Scriptable Item's modelPrefab field too.

## Durability

Each ScriptableItem has a durability option:




An item's durability is decreased:

- When getting attacked by the amount of damage that the player received
- When using a ranged weapon (always)
- When using a melee weapon (if something was hit)

You can modify each scriptable item's OnUse function to also reduce durability on use if you want to.

ScriptableItem's max. Durability can also be set to 0 to completely ignore durability for that item.

The item's color will also change if the durability is either low or if the item is completely broken:

## Build System (Structures)

uSurvival comes with a StructureItem type that can be used to build things. The palette in the Demo Scene can be used to play around with the build system:


Structures snap to a grid, kind of like Minecraft where blocks can only be placed next to other blocks (and not one-third of a way to it).

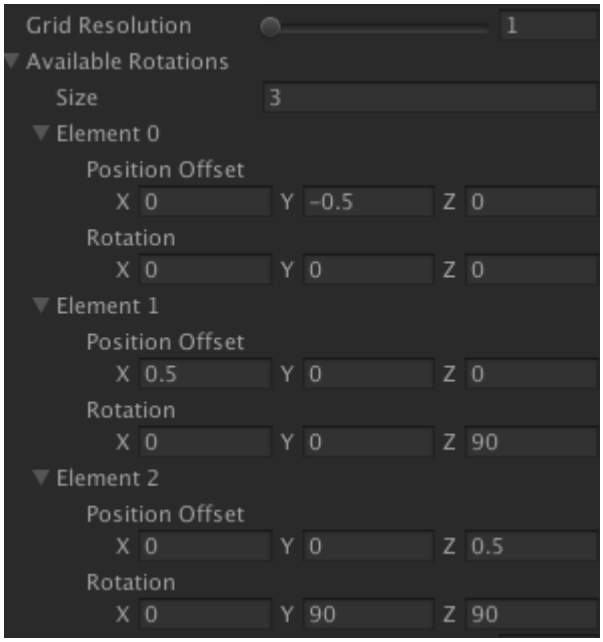
## Grid Resolutions

By default, the palette uses a Grid Resolution of 2, so it can be placed at 0, 0.5, 1, 1.5, etc. In other words, the grid is 'half a palette':

The grid can also be modified. For example, Grid Resolution = 1 means that they can only be placed at 0, 1, 2, 3, etc.:

## Available Rotations and Position Offsets

Different grid resolutions may need position offsets in the AvailableRotations property. For example, here are the offsets and rotations for a resolution of 1:




Play around with those values while building palettes to see what they do!.

*Note: you can cycle through the available rotations with the R key ingame.*

## Equipment

There are two types of equipment: static and animated (aka skinned mesh) equipment.

- An Axe is static equipment, it's simply moved into the player's hand and the attack animation will cause it to swing around.
- Pants are animated equipment, they have to follow the player's animation while running, etc.

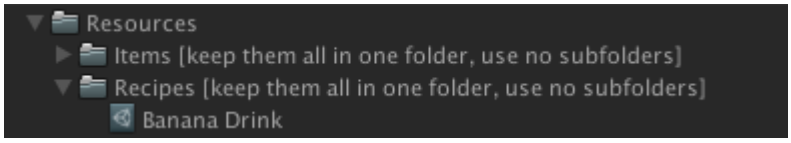
We don't have to worry about static equipment at all, simply create whatever model that you like and uSurvival will show it in the player's right hand.

Animated equipment (skinned meshes) are moved into the Player object, and then their attached Animator follows the main animator's state. Artists will have to export the skinned meshes rigged and animated in order for that to work.

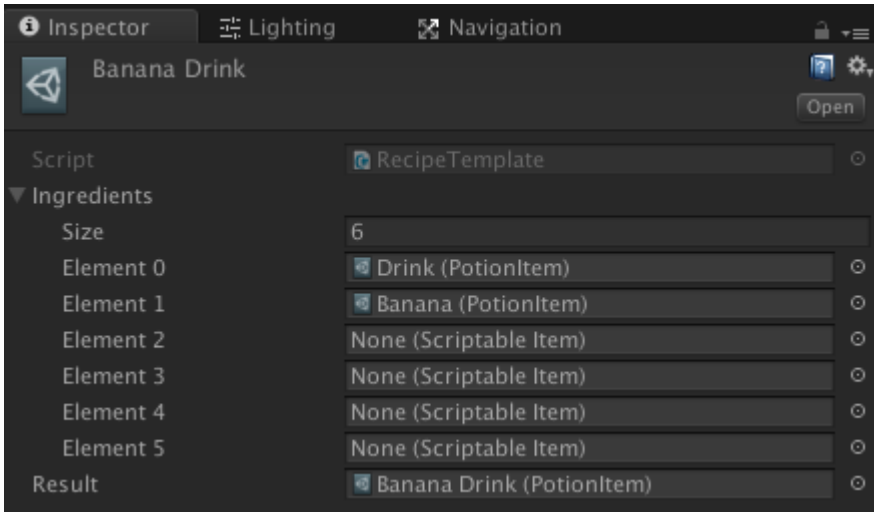
uSurvival already supports skinned mesh equipment, the code was copied from my other uMMORPG asset. uSurvival skinned mesh examples will follow soon.

## Crafting

Crafting recipes can be found in the Resources/Recipes folder:




Recipes are very simple. They have a list with ingredients and a result item:



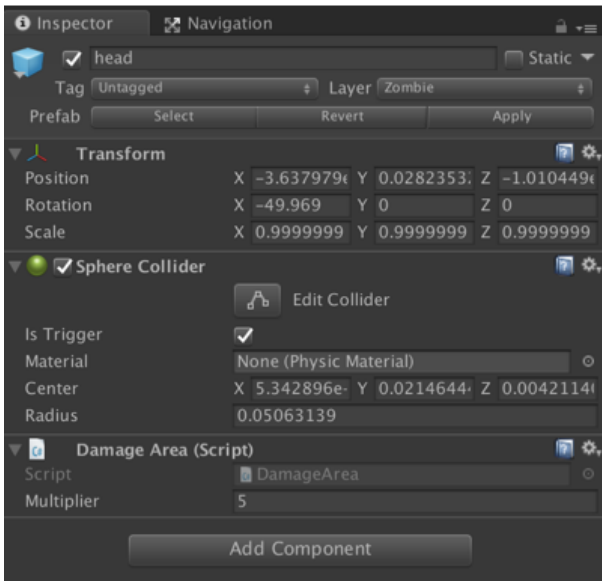

You can craft them ingame by dragging the ingredients into the crafting slots, afterwards the item that can be crafted will appear:

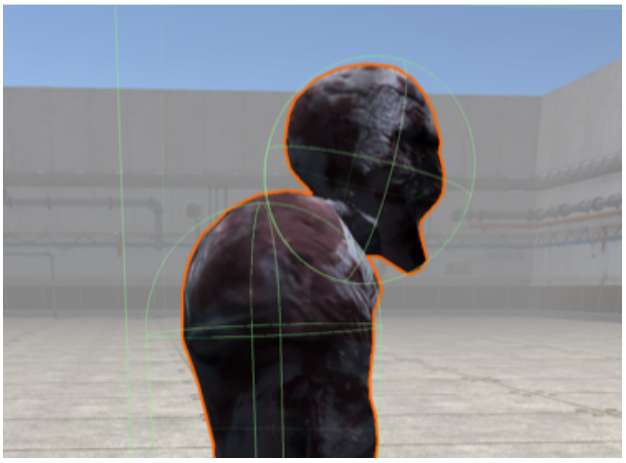

This system also allows for real recipe items. For example, the banana drink could also require a secret scroll that players need to find in order to craft it. The scroll can simply be added as one of the ingredients.

## Damage Areas

Players and Zombies can have different Damage Areas, so that hits on the head (or other body parts) cause more damage than hits on the body.

To add a Damage Area, simply add a (Trigger)Collider to the Head/Arm/etc. and then also add a DamageArea component to it:



## Interaction System

uSurvival comes with a built in networked interaction system:


Here is how to add another interactable object, like a campfire that can be lit:

- Drag the object into the Scene
- Add a collider so that the Interaction system detects it with Raycasting
- Create a Campfire script that inherits from Interactable and add it to the GameObject
- Set the component's **.text** to 'Light fire'
- Implement the **public override void OnInteract(GameObject player)** function and do your fire lighting logic there
- Press Play, walk up to the object, see the interaction hint and then press F to interact with it

## Scenes

UNET can only handle one scene at a time, and there isn't really any need to have multiple scenes in a Survival game.

You should simply modify the existing scene to your needs.

## Energy

uSurvival has one abstract Energy type which can be inherited from to by all kinds of energies like Health, Nutrition and Hydration.

Energies can recover once per second. They can recover positively and negatively.

Energies can underflow and overflow into other Energies. For example, if Nutrition recovers by -1 per second (aka you get hungry), then it will reach 0 eventually. Once that happened, it can continue to reduce (for example) health by -1 once per second until Nutrition was filled up again (until the player ate something).

In other words, a player can die after not eating for a while, etc.

You can also add more energies like Oxygen. Simply inherit from Energy, add the component to the Player and then modify UIHud to display it on the screen.

# The Database

## About SQLITE

uSurvival uses SQLITE for the database. SQLITE is like MySQL, but all stored in a single file. There's no need for a database server or any setup at all, uSurvival automatically creates the Database.sqlite file when the server is started.

SQLITE and MySQL are very similar, so you could modify the Database.cs script to work with MYSQL if needed.

Note that SQLITE is more than capable though. Read the SQLITE Wikipedia entry to understand why.

## How to view and modify the Database

The database can be found in the project folder, it has the name Database.sqlite.

The database structure is very simple, it contains a few tables for characters and accounts. They can be modified with just about any SQLite browser and we listed several good options in the comments of the Database script (e.g. SqliteBrowser).

Characters can be moved to a different account by simply modifying their 'account' property in the database.

A character can be deleted by setting the 'deleted' property to 1 and can be restored by setting it to 0 again. This is very useful in case someone accidentally deleted their character.

## The User Interface (UI)

uSurvival uses Unity's new UI system. Please read through the UI manual first:

https://docs.unity3d.com/Manual/UISystem.html

Modifying the UI is very easy. Just modify it in the Scene in 2D view.

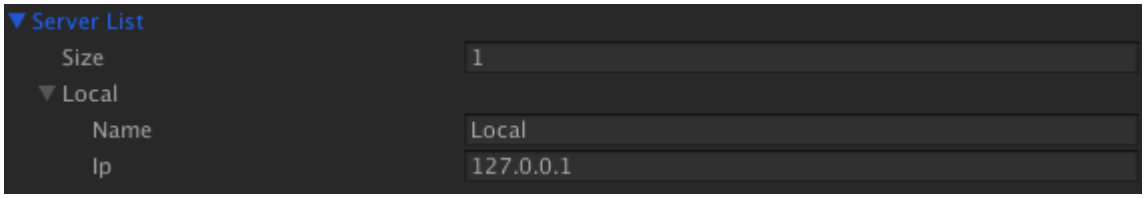
Most of the UI elements have UI components attached to them. There is no magic here, they usually just find the local player and display his stats in a UI element.

Feel free to modify all the UI to your needs.

## Server Hosting

### The Server List

uSurvival's NetworkManager has a Server List:




The default entry is for a local server on your own computer, so you can test multiplayer easily. If you want to host uSurvival on the internet, you can add another entry here with some name and the server's IP. Players can then select it in the Login screen:


### Building the Server Binary

UNET puts the server and the client into one project by default, so any build can run as client or server as necessary. There is no special build process needed.

It's obviously a bad idea to host the server on a mobile device or in WebGL of course, it should be a standalone platform like Windows/Mac/Linux with some decent hardware.

The recommended server platform is Linux. Unity can create a headless build of your game there, so that no rendering happens at all. This is great for performance.

## Setting up a Server Machine

Linux is the recommended Server system. If you have no idea how to get started hosting a UNET game on a Linux system or where and which one to even rent, then please go through my UNET Server hosting tutorial for a step by step guide and then continue reading here.

If you already know your way around the Terminal, then use the following commands:

Upload Headless build to home directory:

```
scp /path/to/headless.zip root@1.2.3.4:~/headless.zip
```

Login via ssh:

```
ssh root@1.2.3.4
```

Install 32 bit support (just in case), sqlite, unzip:

```
sudo dpkg --add-architecture i386

apt-get update

sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386

sudo apt-get install libsqlite3-0:i386

sudo apt-get install unzip
```

Unzip the headless build:

```
unzip headless.zip
```

Run the server with Log messages shown in the terminal:

```
./uSurvival.x86_64 -logfile /dev/stdout
```

## Hiding Server code from the Client

The whole point of UNET was to have all the server and client source code in one project. This seems counter-intuitive at first, but that's the part that saves us years of work. Where we previously needed 50.000 lines of code or more, we only need 5000 lines now because the server and the client share 90% of it.

The 10% of the code that are only for the server are mostly commands. Reverse engineering the client could make this code visible to interested eyes, and some developers are concerned about that.

The first thing to keep in mind is that this does not matter as long as the server validates all the input correctly. Seeing the source code of a command doesn't make hacking that much easier, since for most commands it's pretty obvious what the code would do anyway. For example, the Linux operating system is very secure, even though it's code is fully open source.

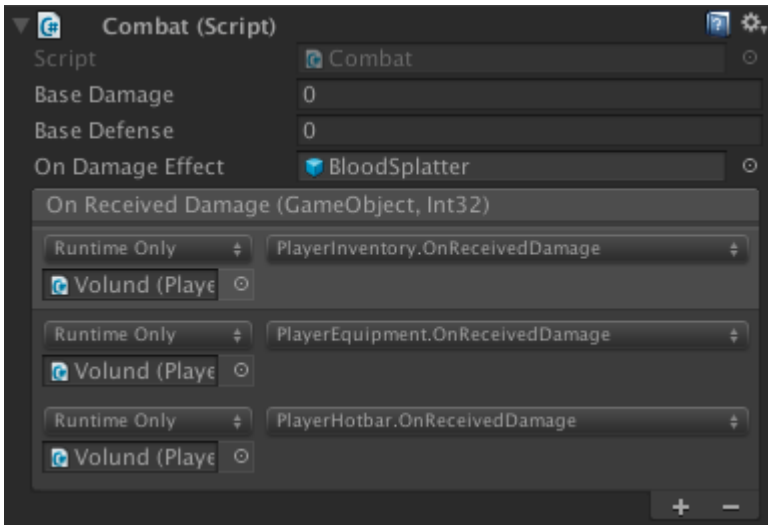
Furthermore it's usually a good idea for game developers to spend all of their efforts on the gameplay to make the best game possible, without worrying about DRM or code obfuscation too much. That being said, if you still want to hide some of the server's code from clients, you could wrap your [Command]s like this:

```
1 [Command]
2 void CmdTeleport(Vector3 position)
3 {
4     #if SERVER
5     ... your teleport code here
6     #end
7 }
```

And then #define SERVER in your code before building your server.

## Script Events

uSurvival's components offer several events that can be hooked into, for example:




You can use those events to add custom functionality without touching the core code. Simply press the **+** Button, drag in the prefab and select which public function to call in your component (e.g. for OnReceivedDamage, show another damage effect).

You can add your own events easily by using Unity's UnityEvent type:

```
[Header("Events")]
public UnityEvent onEmpty;
```

And then calling .Invoke() for it.

## Updating

There are a lot more features and improvements planned for uSurvival. If you want to update your local version to the latest version, you should keep a few things in mind:

- Always make backups before updating. In fact, you should make backups at least daily so that you can go back through your changes if things go wrong.
- uSurvival comes with 100% of the source code. Updating would be effortless if we would just ship a DLL file that you can't modify, but we decided against that. You get all the code so you can modify it if needed, but always remember: with great power comes great responsibility. If you modify core code and it stops working or it's not compatible with an update,, then you have to fix it yourself.
- You should not update forever. At some point in your development it's smart to stick with one version and only manually apply bug fixes afterwards. For example, Valve didn't continuously update Counter Strike 1.6 - at some point they only applied fixes while starting to work on Counter Strike Source with their newest engine.
- A local Git repository is a great tool to keep track of your own code changes and of uSurvival code changes. Maybe Git Branches are a good idea for you.

## Networking Tests (CCU Benchmarks)

We did several community tests to proof that uSurvival can handle large amounts of players. Keep reading to learn about the CCU (concurrent users) goal, the server setup and the results.

### The Goal

Games like Fortnite and DayZ managed to make millions of dollars with a 100 CCU limit, so 100 seems like a reasonable goal.

Networking games use areas of interest so that not all 100 players send their state to all 100 other players, as this would result in 100 x 100 = 10,000 broadcasts per state change, which is an insane amount of packets being sent around, DayZ for sure can't handle that, and we aren't sure about Fortnite either. Instead, people are usually surrounded by less than 5 players, often times they are even completely alone, resulting in way less broadcasts per state change.

## Test Results

We did a **worst case** test because we really want to see how much the server can handle if it comes down to it.

In other words, multiply our CCU result with 2x-5x and you can estimate the **real case** CCU.

**Date:** August 27, 2018

**Players (CCU):** 122

**Result:** **Success!** 0 errors. 0 loss. 0 disconnects. 0 hickups.

**Video:** https://youtu.be/gKdi-Fk290Q

**Screenshots:** *(right click and open images in new Tab for details)*

121 Players ingame:

## The Server Machine

We rented the PX92 dedicated server from Hetzner for 105€/month with the following specs:

| CPU | RAM | HARD DRIVE | OPERATING SYSTEM |
|---|---|---|---|
| Xeon W-2145 8 Cores x 3.7 GHz | 128 GB DDR3 | 240 GB SSD, 6GB/s | Ubuntu 16.04 LTS x86_64 |

Many survival games run on $400+/month servers. We will test on one of those next time.

## The Software Stack

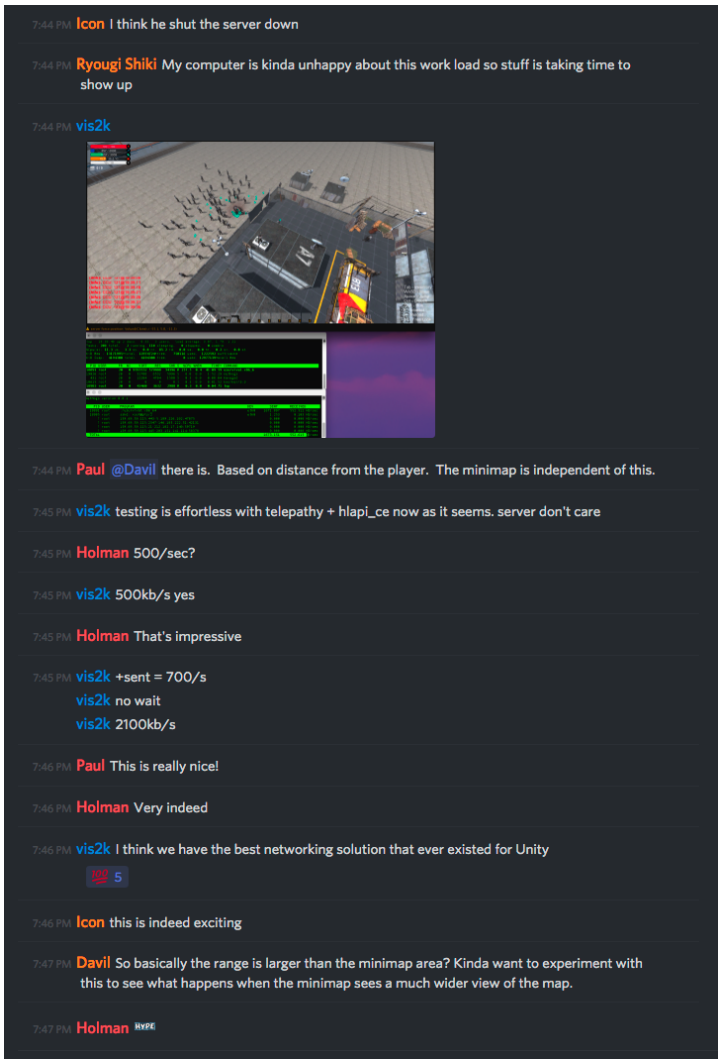
| uSurvival | Unity Version | High Level Networking | Low Level Networking |
|---|---|---|---|
| V1.18 | 2017.4.7f1 LTS | HLAPI CE | Telepathy |

## Witnesses

We always test CCU with our community, so that there are witnesses for the test results.

If you don't believe the test results, feel free to come by our Discord server and ask any of the participants:

`vis2k, KarlGG, Ryougi Shiki, Isaac, Davil, Holman, Cioa, Icon, Radioactive Bullfrog, Faiyth, ...`



## Websocket Test

We also ran a Websocket Test a while ago, just to be sure.

| Date | uSurvival | Unity | HLAPI | Server | Client | Protocol | CCU Goal | Participants |
|---|---|---|---|---|---|---|---|---|
| 2018-07-08 | V1.11 | 2017.4.6 | HLAPI CE | Headless Linux | WebGL | WebSockets | 25 | Ronith<br>UberWigget<br>vis2k<br>etc. |

**Limitations:** NetworkTransport doesn't provide debug/statistics with WebSockets enabled, so we couldn't measure bandwidth / pending packets, etc.

**Result:** **Test succeeded.** 25 CCU ran perfectly smooth.

**Proof:** *(right click and open images in new Tab for details)*


**Notes:** we contacted the NetworkTransport developer to hopefully get Websocket debugging in the future. All the -1 values in the screenshot above were not available in

WebSockets. Will redo test when available and when we have more processing power in the community.