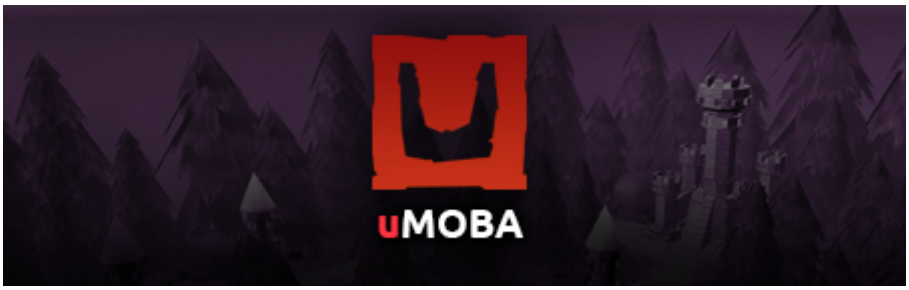
# Unity uMOBA Documentation




## Introduction

**uMOBA** is a simple and easy to understand **M**ultiplayer **O**nline **B**attle **A**rena Demo project, similar to Dota and League of Legends. It was developed with Unity's UNET Networking system.

What we provide is a learning project that focuses on the core MOBA features like Entities, Combat, Items & Skills while avoiding the overhead around it.

This project is as simple as it gets when it comes to MOBA development because **the Server and the Client are ONE**, there is no more separation between them. Unity takes care of it all!

uMOBA currently contains only about **2500 lines** of clean, elegant and easy to understand source code. We designed it in a way that allows you to just run it and start hacking around in it immediately.

**Download**

You can Download uMOBA on the Unity Asset Store!
*Note: in order to respect Unity's Asset Store agreement, uMOBA is **not** part of the noobtuts Premium files right now.*

Feel free to take a look at our uMOBA WebGL Demo first!

## Features

uMOBA contains all of the MOBA genre's core features:

- Fully Unity + UNET based, no other tools needed
- Same Scripts for Client & Server
- Uses Unity's new UI system
- Players, Monsters, NPCs, Towers, Bases
- Movement via Navigation
- Animations via Mecanim
- Beautiful 3D Character and Environment Models
- 3 Lanes
- All, Whisper, Team, Info Chat
- Minimap

- Levels
- Skills & Buffs
- Items & Gold
- NPC Trading
- PvP
- Death and Respawning
- Only 3500 lines of carefully crafted C# Code
- Secure: all logic is simulated on the Server
- MOBA Camera with Zoom
- Lobby & Matchmaking
- Network Time Synchronization
- Platform Independent: works where UNET works
- Mobile Support
- Free of Deadlocks & Race Conditions

## Quick start guide

1. Install and open Unity 5.4 or newer.
2. Import the Project from Unity's Asset Store and open the Lobby scene file.
3. Press **Play** in the Editor, select "Play and Host", press "Join".

### General usage advice

If you are a developer then you can either use this project to build your own customized MOBA on top of it, you can use it as a reference to learn the best way to implement specific MOBA features or you can even take components out of it and add them to your own game.

We used lots of comments throughout the code, so if you want to learn more, simply take a look at the implementation of the function that you are interested in.

## About the Network Game Lobby

The lobby scene was entirely developed by Unity can be found on the Asset Store.

We added a few modifications to allow hero selection. We suggest not updating the lobby asset yourself. If you notice a new lobby update, simply let us know and we will take care of updating it in uMOBA.

## How-Tos

### How to add a Monster Type

The best way to add a new Monster type is to drag an existing Monster into the scene, modify it as needed and then drag it into the Prefabs folder to create a new one. Afterwards you can select the monster spawner objects in the Hierarchy and then drag the monster prefab of your choice into it.

### How to add an Npc Type

The best way to add a new Npc is to duplicate an existing Npc, modify it as needed and then drag it into the Prefabs folder to save it permanently.

### How to add a Player Type

Just like Monsters and Npcs, adding a different player type isn't hard because all we have to do is duplicate the existing one. We can duplicate a Unity Prefab like this: drag the Prefab into the Hierarchy, rename it, drag it back into the Project Area to create Prefab, then remove both of them from the Hierarchy again. Afterwards we can modify the parts in

the new Player Prefab that we want to modify and then select the LobbyManager in the Hierarchy and drag the new player Prefab into **Game Player Prefab** property.

### How to add a Skill

Adding a new Skill is very easy. At first we right click in the Project Area and select **Create->Scriptable Object** and press the **SkillTemplate** button. This creates a new ScriptableObject that we can now rename to something like "Strong Attack". Afterwards we can select it in the Project Area and the modify the skill's properties in the Inspector. It's usually a good idea to find a similar existing skill in the ScriptableObjects folder and then copy the category, cooldown, cast range etc.

Afterwards we select the player prefab and then drag our new skill into the **Skill Templates** list to make sure that the player can learn it.

### How to add an Item

Adding a new Item is very easy. At first we right click in the Project Area and select **Create->Scriptable Object** and press the **ItemTemplate** button. This creates a new ScriptableObject that we can now rename to something like "Strong Potion". Afterwards we can select it in the Project Area and the modify the items's properties in the Inspector. It's usually a good idea to find a similar existing item in the ScriptableObjects folder and then copy the category, max stack, prices and other properties.

Afterwards we have to make sure that the item can be found in the game world somehow. There are several options, for example:

- We could add it to an Npc's **Sale Items** list
- We could add it to a Player's **Default Items** list

### How to change the Start Position

The NetworkManager will always search for a GameObject with a **NetworkStartPosition** component attached to it and then use it as the start position.

uMOBA has a **PlayerSpawn** GameObject in the Hierarchy, which can be moved around in order to modify the character's start position.

### How to add Environment Models

Environment models like rocks, trees and buildings can simply be dragged into the Scene. They don't need a Collider, but they should be made **Static** so that the Navigation system recognizes them.

Afterwards we can select **Window->Navigation** and press the **Bake** button in order to refresh the navigation mesh. This makes sure that the player can't walk through those new environment objects.

### How to connect to the Server over the Local Network

If you want to connect to the game server from another computer in your local network, then all you have to do is select the LobbyManager in the Hierarchy and modify the Network Info -> Network Address property to the IP address of the server.

*Note: you also have to configure both computers so that they can talk to each other without being blocked by firewalls.*

### How to update your own modified MOBA to the latest uMOBA

uMOBA's code is likely to change a bit every now and then, so upgrading your modified MOBA to the latest uMOBA version could always break something in your project. We recommend to pick the

latest uMOBA version and then develop your own MOBA with it without updating to the latest uMOBA version all the time.

If however uMOBA has a new feature or bugfix that you desperately need, then we recommend the following steps:

- Make a backup of your current project in any case
- Create a new Unity project and download the latest uMOBA version
- Then either:
  - Find the bugfix/feature that you want and manually copy it into your own MOBA
  - Or add your own MOBA's content to the new Project again

Of course, you can always try to just update your current project and hope for the best / fix occurring errors. Just be warned that this might be stressful, because uMOBA is a source code project. Conventional software can easily be made downwards compatible because the user never modifies the source code directly. But if you modify code that we change later on, then there can always be complications.

*Note: we always try to fix all known bugs before releasing a new uMOBA version, so you really won't have to update to the latest version all the time and can work with one version for a long time instead.*

## Documentation

We will now give a basic overview about uMOBA's architecture and design decisions and then explain each Script file in detail. Note that the documentation is parsed from the project's script files, so you might as well jump right into the project and learn even more.

### Technology Choices

We will begin by explaining our technology choices:

- **Networking:** When it comes to networking, we have several different choices like UNET, TCP/IP or external solutions. UNET is a really good choice for networking. Dealing with TCP Clients & Servers means lots of manual serialization, lots of packets and opcodes and dealing with networking issues. UNET takes care of all these problems. UNET is built directly into Unity, so it's really the most simple choice too, in comparison to external solutions. When it comes to UNET, we can either use SyncVars/SyncLists or use manual Serialization via OnSerialize and OnDeserialize. Using SyncVars/SyncLists saves us a lot of work when synchronizing things like item lists, skills or gold. Doing that manually with OnSerialize and OnDeserialize would require dirty bits and lots of serialization that we don't really want to worry about. We should also keep in mind that if we were to add something to our item type, we would always have to remember to serialize it as well. Luckily for us, Unity takes care of that automatically when using SyncLists. The only downside of SyncLists is the fact that we can only use simple types like int and float, or structs. We can choose between different Qos Channels in the NetworkManager. We use *Reliable Fragmented* for uMOBA because packets should be reliable (we really do need the client to receive them). Synchronizing bigger structures like a SyncList of Items will often exceed the maximum packet size, hence why we need a *fragmented* channel type that can split big packets into smaller fragments.

- **NavMeshAgents vs. real Physics:** we don't really use any *real* physics in uMOBA. Instead we completely rely on the NavMeshAgent for movement. A lot of online games have problems with gravity, falling through the level or walking through thin walls. Using only a NavMeshAgent without any physics is the solution to all of those problems because the agent can only walk on the

NavMesh. There is absolutely no way to fall through the map or to walk through walls. On a side note, this also means that we only have to synchronize the agent's destination once, instead of synchronizing a player's position every few milliseconds, hence we save a LOT of computations and bandwidth.

- **OnGUI vs. UI:** Unity's old OnGUI system works great. The new UI system is not that bad either. The new UI system is more complex, but it's more mobile friendly and has easier drag and drop support. When using the new UI, we always create a new script like UISkillbar and attach it to the UI component in the Hierarchy. This is a better architecture than one big UI script, or handling UI in the player class, etc.

- **Simplicity vs. Performance:** Our #1 goal when designing uMOBA was simplicity. We prefer simple and clean code over highly optimized and overly complex code. A 100.000 lines of code MOBA with all the latest cutting edge performance improvements is nice if you have enough people to work on that huge amount of code. But for indie developers a 5000 lines that can be maintained by a single developers is far more valuable.

- **Scripting Language:** Unity currently supports CSharp, JavaScript and Boo. Boo is pretty much C# with a shorter syntax, but the language has no support for SyncVar hooks, hence it's not an option. CSharp is well documented and used by most of the Unity developers, hence the choice over JavaScript.

**123Skill.cs**

The Skill struct only contains the dynamic skill properties and a name, so that the static properties can be read from the scriptable object. The benefits are low bandwidth and easy editing in the Inspector.

Skills have to be structs in order to work with SyncLists.

We implemented the cooldowns in a non-traditional way. Instead of counting and increasing the elapsed time since the last cast, we simply set the 'end' Time variable to Time.time + cooldown after casting each time. This way we don't need an extra Update method that increases the elapsed time for each skill all the time.

Note: the file can't be named "Skill.cs" because of the following UNET bug: http://forum.unity3d.com/threads/bug-synclist struct-only-works-with-some-file-names.384582/

**AggroArea.cs**

Catches the Aggro Sphere's OnTrigger functions and forwards them to the Entity. Make sure that the aggro area's layer is IgnoreRaycast, so that clicking on the area won't cause the monster to be selected.

Note that a player's collider might be on the pelvis for animation reasons, so we need to use GetComponentInParent to find the Player script.

**Barrack.cs**

The Barrack entity type.

**Base.cs**

The Base entity type, used for buildings that can be destroyed.

**CameraScrolling.cs**

Moves the camera around when the player's cursor is at the edge of the screen

**CameraZooming.cs**

Zooms in with the mouse wheel.

**DefaultVelocity.cs**

Sets the Rigidbody's velocity in Start().

**DestroyAfter.cs**

Destroys the GameObject afer a certain time.

**Entity.cs**

The Entity class is rather simple. It contains a few basic entity properties like health, mana and level *(which are not public)* and then offers several public functions to read and modify them.

Entities also have a *target* Entity that can't be synchronized with a SyncVar. Instead we created a EntityTargetSync component that takes care of that for us.

Entities use a deterministic finite state machine to handle IDLE/MOVING/DEAD/ CASTING etc. states and events. Using a deterministic FSM means that we react to every single event that can happen in every state (as opposed to just taking care of the ones that we care about right now). This means a bit more code, but it also means that we avoid all kinds of weird situations like 'the monster doesn't react to a dead target when casting' etc. The next state is always set with the return value of the UpdateServer function. It can never be set outside of it, to make sure that all events are truly handled in the state machine and not outside of it. Otherwise we may be tempted to set a state in CmdBeingTrading etc., but would likely forget of special things to do depending on the current state.

Each entity needs two colliders. First of all, the proximity checks don't work if there is no collider on that same GameObject, hence why all Entities have a very small trigger BoxCollider on them. They also need a real trigger that always matches their position, so that Raycast selection works. The real trigger is always attached to the pelvis in the bone structure, so that it automatically follows the animation. Otherwise we wouldn't be able to select dead entities because their death animation often throws them far behind.

Entities also need a kinematic Rigidbody so that OnTrigger functions can be called. Note that there is currently a Unity bug that slows down the agent when having lots of FPS(300+) if the Rigidbody's Interpolate option is enabled. So for now it's important to disable Interpolation - which is a good idea in general to increase performance.

**EntityTargetSync.cs**

[SyncVar] GameObject doesn't work, [SyncVar] NetworkIdentity works but can't be set to null without UNET bugs, so this class is used to serialize an Entity's target. We can't use Serialization in classes that already use SyncVars, hence why we need an extra class.

We always serialize the entity's GameObject and then use GetComponent, because we can't directly serialize the Entity type.

**Extensions.cs**

This class adds functions to built-in types.

**FaceCamera.cs**

Useful for Text Meshes / Canvases that should face the camera.

In some cases there seems to be a Unity bug where the text meshes end up in weird positions if it's not positioned at (0,0,0). In that case simply put it into an empty GameObject and use that empty GameObject for positioning.

**FogOfWar.cs**

The Fog of War plane in 3D should be slightly above the ground, so that trees etc. aren't darkened, only the ground. Entities will never stand above it, because no entities should be visible where there is fog of war anyway.

Note: it finds the local player, which means that it's a client sided effect and we don't have to worry about performance issues on the server.

Note: we use a Unlit/TransparentBlur shader to blur the cutout circles.

**Item.cs**

The Item struct only contains the dynamic item properties and a name, so that the static properties can be read from the scriptable object.

Items have to be structs in order to work with SyncLists.

The player inventory actually needs Item slots that can sometimes be empty and sometimes contain an Item. The obvious way to do this would be a InventorySlot class that can store an Item, but SyncLists only work with structs - so the Item struct needs an option to be *empty* to act like a slot. The simple solution to it is the *valid* property in the Item struct. If valid is false then this Item is to be considered empty.

*Note: the alternative is to have a list of Slots that can contain Items and to serialize them manually in OnSerialize and OnDeserialize, but that would be a whole lot of work and the workaround with the valid property is much simpler.*

Items can be compared with their name property, two items are the same type if their names are equal.

**ItemTemplate.cs**

Saves the item info in a ScriptableObject that can be used ingame by referencing it from a MonoBehaviour. It only stores an item's static data.

We also add each one to a dictionary automatically, so that all of them can be found by name without having to put them all in a database. Note that we have to put them all into the Resources folder and use Resources.LoadAll to load them. This is important because some items may not be referenced by any entity ingame (e.g. when a special event item isn't dropped anymore after the event). But all items should still be loadable from the database, even if they are not referenced by anyone.

An Item can be created by right clicking the Resources folder and selecting Create -> uMOBA Item. Existing items can be found in the Resources folder.

**LobbyHookMOBA.cs**

This component transfers data from the lobby player to the gameobject player.

**Monster.cs**

The Monster class has a few different features that all aim to make monsters behave as realistically as possible.

- **States:** first of all, the monster has several different states like IDLE, ATTACKING, MOVING and DEATH. The monster will randomly move around in a certain movement radius and try to attack any players in its aggro range. *Note: monsters use NavMeshAgents to move on the NavMesh.*

- **Aggro:** To save computations, we let Unity take care of finding players in the aggro range by simply adding a AggroArea *(see AggroArea.cs)* sphere to the monster's children in the Hierarchy. We then use the OnTrigger functions to find players that are in the aggro area. The monster will always move to the nearest aggro player and then attack it as long as the player is in the follow radius. If the player happens to walk out of the follow radius then the monster will continue to walk to its goal.

- **Respawning:** The monsters have a *respawn* property that can be set to true in order to make the monster respawn after it died. We developed the respawn system with simplicity in mind, there are no extra spawner objects needed. As soon as a monster dies, it will make itself invisible for a while and then go back to the starting position to respawn. This feature allows the developer to quickly drag monster Prefabs into the scene and place them anywhere, without worrying about spawners and spawn areas.

- **Loot:** Dead monsters can also generate loot, based on the *lootGold*, which generates gold between a minimum and a maximum amount.

**MonsterSpawner.cs**

Used to spawn monsters repeatedly. The 'monsterGoals' will be passed to the monster after spawning it, so that the monster knows where to move (which lane to walk along).

**MouseoverOutline.cs**

Draws an outline around the entity while the mouse hovers over it. The outline strength can be set in the entity's material. Note: requires a shader with "_OutlineColor" parameter. Note: we use the outline color alpha channel for visibility, which is easier than saving the default strenghts and settings strengths to 0.

**NavmeshPathGizmo.cs**

Draws the agent's path as Gizmo.

**NetworkMessages.cs**

Contains all the network messages that we need.

**NetworkName.cs**

Synchronizing an entity's name is crucial for components that need the proper name in the Start function (e.g. to load the skillbar by name).

Simply using OnSerialize and OnDeserialize is the easiest way to do it. Using a SyncVar would require Start, Hooks etc.

**NetworkNavMeshAgent.cs**

UNET's current NetworkTransform is really laggy, so we make it smooth by simply synchronizing the agent's destination. We could also

lerp between the transform positions, but this is much easier and saves lots of bandwidth.

Using a NavMeshAgent also has the benefit that no rotation has to be synced while moving.

Notes:
- Teleportations have to be detected and synchronized properly
- Caching the agent won't work because serialization sometimes happens before awake/start
- We also need the stopping distance, otherwise entities move too far.

**NetworkProximityCheckerTeam.cs**

A simple proximity check around ourselves isn't enough in games where we have multiple units in our team (MOBA, RTS etc.). There we want to see everything around us and everything around each of our team members.

We also modify the NetworkProximityChecker source from BitBucket to support colliders on child objects by searching the NetworkIdentity in parents.

Note: requires at least Unity 5.3.5, otherwise there is IL2CPP bug #786499. Note: visRange means 'enemies in range we should use the same visrange for everything

**NetworkTime.cs**

Clients need to know the server time for cooldown calculations etc. Synchronizing the server time every second or so wouldn't be very precise, so we calculate an offset that can be added to the client's time in order to calculate the server time.

The component should be attached to a NetworkTime GameObject that is always in the scene and that has no duplicates.

**Npc.cs**

The Npc class is rather simple. It contains state Update functions that do nothing at the moment, because Npcs are supposed to stand around all day.

Npcs first show the welcome text and then have options for item trading and quests.

**Player.cs**

All player logic was put into this class. We could also split it into several smaller components, but this would result in many GetComponent calls and a more complex syntax.

The default Player class takes care of the basic player logic like the state machine and some properties like damage and defense.

The Player class stores the maximum experience for each level in a simple array. So the maximum experience for level 1 can be found in expMax[0] and the maximum experience for level 2 can be found in expMax[1] and so on. The player's health and mana are also level dependent in most MMORPGs, hence why there are hpMax and mpMax arrays too. We can find out a players's max health in level 1 by using hpMax[0] and so on.

The class also takes care of selection handling, which detects 3D world clicks and then targets/navigates somewhere/interacts with someone.

Animations are not handled by the NetworkAnimator because it's still very buggy and because it can't really react to movement stops fast enough, which results in moonwalking. Not synchronizing animations over the network will also save us bandwidth.

**PlayerChat.cs**

We implemented a chat system that works directly with UNET. The chat supports different channels that can be used to communicate with other players:

- **Team Chat:** by default, all messages that don't start with a / are addressed to the team.
- **Whisper Chat:** a player can write a private message to another player by using the **/ name message** format.
- **All Chat:** we implemented all chat support with the **/all message** command.
- **Info Chat:** the info chat can be used by the server to notify all players about important news. The clients won't be able to write any info messages.

*Note: the channel names, colors and commands can be edited in the Inspector by selecting the Player prefab and taking a look at the PlayerChat component.*

A player can also click on a chat message in order to reply to it.

**PlayerDndHandling.cs**

Takes care of Drag and Drop events for the player. Works with UI and OnGUI. Simply do: FindObjectOfType().OnDragAndDrop(...);

**PlayerSpawn.cs**

Used to set a spawnpoint's team, so that the correct players can be spawned here.

**Projectile.cs**

This class is for bullets, arrows, fireballs and so on.

**SkillTemplate.cs**

Saves the skill info in a ScriptableObject that can be used ingame by referencing it from a MonoBehaviour. It only stores an skill's static data.

We also add each one to a dictionary automatically, so that all of them can be found by name without having to put them all in a database. Note that we have to put them all into the Resources folder and use Resources.LoadAll to load them. This is important because some skills may not be referenced by any entity ingame (e.g. after a special event). But all skills should still be loadable from the database, even if they are not referenced by anyone.

Skills can have different stats for each skill level. This is what the 'levels' list is for. If you only need one level, then only add one entry to it in the Inspector.

A Skill can be created by right clicking the Resources folder and selecting Create -> uMOBA Skill. Existing skills can be found in the Resources folder.

**StartServerIfHeadless.cs**

Automatically starts a dedicated server if running in headless mode. (because we can't click the button in headless mode)

**TextMeshCopyText.cs**

Copy text from one text mesh to another (for shadows etc.)

**TextMeshFadeAlpha.cs**

Fade the text mesh color's alpha value over time.

**TextMeshHealthBar.cs**

HealthBar via TextMesh of "_". This is a workaround for the following Unity bug: https://issuetracker.unity3d.com/issues/destroy-gameobject-time-causes-unity-to-crash-in-sharedgfxbuffer-unshare-sharedgfxbuffer-star

**Tower.cs**

The Tower entity type. Automatically attacks entities from the opposite team.

**Utils.cs**

This class contains some helper functions.

---