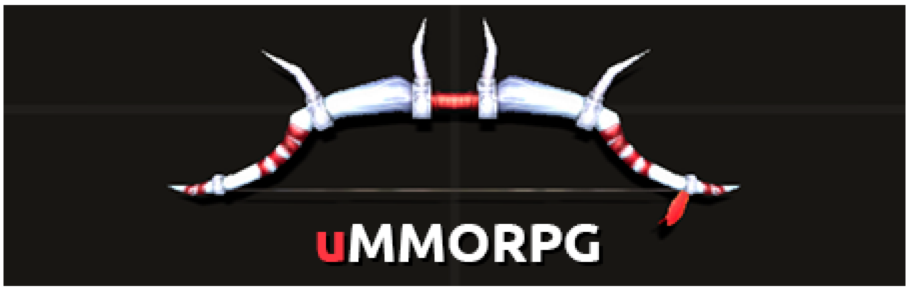
Unity uMMORPG Documentation



Introduction

uMMORPG is a simple and easy to understand **M**assive **M**ultiplayer **O**nline **R**ole **P**laying **G**ame Demo project developed with Unity's UNET Networking system.

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

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

uMMORPG currently contains only about **3500 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 uMMORPG on the Unity Asset Store!
*Note: in order to respect Unity's Asset Store agreement, uMMORPG is **not** part of the noobtuts Premium files right now.*

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

Quick start guide

1. Install and open Unity 5.4.1p2 or newer.
2. Import the Project from Unity's Asset Store and open the World scene file.
3. Build & Run it for your operating system via **File->Build Settings**
   Note: you can also press Ctrl+B to immediately build and run it afterwards.
4. Select "Server & Play" in the Editor to start the Server and play on it (with the account entered in the Login Mask)
   Note: a Dedicated Server lets you start a Server without playing on it at the same time..

5. Enter a different account (any one is accepted right now) and press "Login" in the build to see the first two players in your MMORPG.

Architecture Overview

uMMORPG was designed for you to modify. Here is a quick overview for the code and architecture:

- The **NetworkManagerMMO** lives in the Hierarchy. It's used for login, character selection/creation and to start/stop the server.
- The Canvas holds all **UI** elements. Each element has a script that pulls in the local player's state for health, mana, etc.
- The **Main Camera** follows the local player.
- Players, Monsters, Npcs all inherit from **Entity**. Entity holds common properties like health, mana, damage, defense.
    - **NPCs** don't do much except for standing around and offering quests and items.
    - **Monsters** use a finite state machine for IDLE/MOVE/CASTING/DEAD etc.
    - **Players** use a finite state machine for IDLE/MOVE/CASTING/DEAD etc too. There is also a localPlayer check. Local players react to input and send commands to the server.
    - There are **Prefabs** for all players/monsters/npcs. Use the prefabs to modify them.

- Item**Template**/Quest**Template**/Skill**Template** are what's usually stored in a database. Thanks to Unity ScriptableObjects we can edit them in the Inspector though. Templates can be found and modified in the Resources folder.
- **Item**/**Quest**/**Skill** structs are the actual instances that live in the player/monster/npc. They pull their static properties from the templates. They can have dynamic properties like cooldown, amount, etc.
- There is also **NetworkTime** which has to be in the Hierarchy for time synchronization.

General usage advice

If you are a developer then you can either use this project to build your own customized MMORPG on top of it, you can use it as a reference to learn the best way to implement specific MMORPG 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.

A Beginner's Exercise

Here is a simple exercise to get your feet wet without writing any code: try adding a new 'Universal Potion' item that restores health and mana. Give it a nice description, adjust all the properties and add an icon too. When you are done, give players a way to obtain it in the game world, e.g. by adding it to a monster's drops.

Note: the How-To section on items could be useful.

How-Tos

How to add a Monster Type

The best way to add a new Monster type is to duplicate an existing Monster, rename it, position it anywhere in the Scene and then modify the Monster component's properties in the Inspector to change the damage, the level and so on. It's also a good idea to create a Prefab of the new Monster and save it in the Prefabs/Entities folder.

How to add an Npc Type

The best way to add a new Npc is to duplicate an existing Npc, rename it, position it anywhere in the Scene and then modify the Npc component's properties in the Inspector in order to change the items that are for sale and the quests. It's also a good idea to create a Prefab of the new Npc and save it in the Prefabs/Entities folder.

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 NetworkManager in the Hierarchy and drag the new player Prefab into the Spawnable Objects list.

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 a Monster's **Drop Chances** list
- We could add it to an Npc's **Sale Items** list
- We could add it to a Player's **Default Items** list
- We could add it to a Player's **Default Equipment** 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.

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

How to change the ground or use a different Terrain

We can use just about any mesh or terrain as the ground element. In order to allow entities to move on it, we simply have to make it **Static**, enable the collider and then select **Window**->**Navigation** and press the **Bake** button in order to refresh the navigation mesh.

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 change a Player/Monster/Npc Model

In order to change a Player model, we first have to drag the player prefab into the Hierarchy so that we can see all its child objects properly. Each player GameObject has two child objects: a **NameOverlayPosition** and a 3D model. We can remove the previous 3D model and then drag our modified 3D model into it.
*Note: some 3D modeling tools cause weird rotations, so sometimes we have to drag our 3D model into an empty **RotationFix** object to adjust it.*

Afterwards we can select the player GameObject again and modify the Animator's **Controller** and **Avatar** properties to our new model. The controller is the animation state machine which can be found in the Project Area. The Avatar can be found in the 3D model's children in the Project Area. The Controller should have the same animation states and triggers like default controller that we used for our prefab.
Note: you can duplicate the controller file with your file explorer and then simply change each state's animations in Unity.

Each player also needs a Collider. While we could add it to the main GameObject, it's usually a better idea to attach it to some kind of bone that moves around with the Animation *(like the **Pelvis**)*, so that a Collider always follows the animation properly. Imagine a standing monster and a dead monster that lays on the ground. If we would attach the Collider to the main GameObject, then it would be in the default standing position at all times. If we attach it to a bone that moves around with the Animation, then the Collider will stand while the monsters is standing and it will lay on the floor when the monster is laying on the floor.
So in other words: make sure to find the pelvis bone and then attach a Capsule Collider to it.

There is one last adjustment to be made. If a player equips a weapon, it should be put into his hands. All of this is already implemented, all we have to do is find the weapon position *(e.g. the left hand)* and then add a **PlayerEquipmentLocation** component with a **EquipmentWeapon** category to it. Afterwards we find the other hand and add another **PlayerEquipmentLocation** component with a **EquipmentShield** category to it.
Note: remember that we did that for the existing player prefabs already, so just take a closer look at them if in doubt.

Other entity models for Monsters and Npcs can be changed in the same way, just without the need for any EquipmentSlots.

How to use another Database System

The Database.cs class is the the only place that has to be modified in order to use another database system like SQLITE or MYSQL.

We decided against using MYSQL for our database, simply because people shouldn't have to setup a whole MYSQL database just to run our Unity MMORPG.

The database currently uses XML files for several reasons, mainly because they are incredibly easy to work with and very fast to read and write. We also tried SQLITE a while ago, but it turned out that it's just far more complicated to deal with *(transactions, locks, table initialization, SQL injection, DLL files for Unity, ...)* and actually much slower than XML. Saving 1000 players took about 30 seconds with SQLITE and only 2 seconds with XML.

In the end, SQLITE and XML are both just files on our hard drive, so we might as well pick the easier solution.

Note: Item, Skill and Quest templates don't really need to be stored in a Database. Right now they are just ScriptableObjects that can be referenced in the game world, which makes development very easy.

How to modify Database Characters

uMMORPG uses the XML document format for its database. The database can be found in the Project's **Database** folder above the Assets folder or next to the executable in builds.

The database structure is very simple, it contains folders for accounts and each folder contains XML files for the account's characters. The files can be modified with any text editor.

Characters can be moved to a different account by simply moving them out of one folder and into another folder.

A character can be deleted by simply deleting the XML file.

How to add an Account Database

In a real MMORPG, we would most likely have a **C**ontent **M**anagement **System** that manages the Forum, News, Accounts and Item Shop. We could then modify the NetworkManagerCustom's IsValidAccount function to verify the login credentials. The function already contains some example code for HTTP-GET and MYSQL.

How to use another Scene

First of all, we have to understand that the game server can only handle one Scene right now, so our whole game world should be in that Scene. If you want to replace the current Scene, you can either just build on top of it or duplicate it and then modify what you want to modify. Unity doesn't have a Duplicate option for Scenes, but we can open the project folder with a file manager, duplicate the scene file and then rename it.

Note: having multiple Scenes at the same time and allowing players to teleport to them makes sense too. It's very likely that UNET will get a feature like that sooner or later, which would be the best solution. Right now we can't use UNET to connect from one UNET server to another UNET server, which makes transferring players impossible without some really weird workarounds, hence why we don't want to implement that feature just yet.

How to add another attribute like Dexterity

uMMORPG already has strength and intelligence attributes which can be upgraded after each level up. To add more attributes, simply take a look at the Player.cs script, find the Attributes section, duplicate one of the current atributes rename it.

You can then modify the Player's Health, Mana, Damage, Defense properties to include your attribute in the formula.

You can show your new attribute in the UI by first adding elements to the CharacterInfo panel in the Canvas and then updating them with the UIRefresh script where attributes can be found in the UpdateCharacterInfo function.

You will notice that the UI script uses Commands when the player asks the server to increase an attribute. The final step is to add one of those commands to the Player script. This is very easy again, since you can simply copy one of the existing attribute commands and modify it to your attribute.

How to make a 2D MMORPG with uMMORPG

uMMORPG is mostly networking code, it doesn't really matter if your game is 2D or 3D. For example, you could easily make your camera ortographic and make it look down on the player to have a 2D game already *(which still uses 3D models)*.

If you want to make a real 2D MMORPG with sprites instead of 3D models, then you will to keep a few things in mind:

- You should be comfortable with Unity's 2D features.
- You will have to replace all 3D models with Sprites and proper 2D Colliders.
- If you want 2D top-down movement with navigation, then you need a 2D navigation system like our Navigation2D asset. You also have to modify our NetworkNavMeshAgent component to make it work with NavMeshAgent2D then - which should be very easy.
- If you want 2D WSAD or sidescroller movement, then you will have to implement that yourself and also make it synchronize with the server.
- The networking code uses Vector3 for positions, which means that an unecessary '0' is sent around all the time. You don't really have to worry about that, unless you run into bandwidth issues some day.

Other than that, items, quests, skills etc. are all the same.

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 NetworkManager 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 host a Server on the Internet

If you want to run the game server on the Internet, then please read our UNET Server Hosting Tutorial. It recommends a hoster and explains the whole process step-by-step.

How to update your own modified MMORPG to the latest uMMORPG

uMMORPG's code is likely to change a bit every now and then, so upgrading your modified MMORPG to the latest uMMORPG version could always break something in your project. We recommend to pick the latest uMMORPG version and then develop your own MMORPG with it without updating to the latest uMMORPG version all the time.

If however uMMORPG 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 uMMORPG version
- Then either:
    - Find the bugfix/feature that you want and manually copy it into your own MMORPG
    - Or add your own MMORPG'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 uMMORPG 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 uMMORPG 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.

How to keep track of uMMORPG changes in detail

If you want to know every single line of code that was changed between versions, there are several ways to do that:

- Look at the file modified dates to see which ones were changed lately
- Use a any text comparison tool
- Use Git or similar version control tools

If you don't know how to use git, here is how you can see version changes with just a few mouse clicks:

- Create a new Unity project, download uMMORPG from the Asset Store
- Put a .gitignore file into that your project folder to ignore some files that we don't need to track:

```
Temp/
Library/
ExportedObj/
obj/
*.svd
*.userprefs
/*.csproj
*.pidb
*.suo
/*.sln
*.user
*.unityproj
*.booproj
.DS_Store
.DS_Store?
._*
.Spotlight-V100
.Trashes
ehthumbs.db
Thumbs.db
```

- Download and open SmartGit
    - Select Repository -> Add or Create
    - Select your project folder, press OK
    - Press 'Initialize' in the next window
    - Now you see the list of files that are new. Select all of them and press Commit and then click Commit in the next window, use the current uMMORPG version like 'V1.1' as the commit message

- Now if a new uMMORPG update is released:
    - Update to the new uMMORPG version for that project

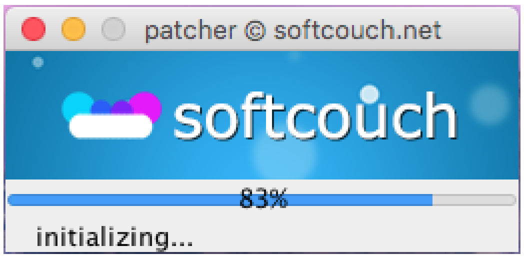
  - Open SmartGit, select your repository on the left
  - Now you see all the changed files in the middle
  - You can click each file to see a comparison between the old and the new content
  - Once you are done, select all files and Commit with 'V1.2', so that this version is now 'default'. Next time you will see all the changed files again.

Note: please don't put uMMORPG into a public repository on the internet. Git works perfectly fine on your local machine.

Development Info

A list of planned features, bugs and requests can be found in our Official uMMORPG Thread on the Unity forums.

Patcher



We developed a game patcher that works for all Unity games. It's a Java program and can't be offered in the Unity Asset Store, but we offer it on our other website here: softcouch.net Patcher.

Script Documentation

We will now give a basic overview about uMMORPG'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 uMMORPG 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 our MMO. Instead we completely rely on the NavMeshAgent for movement. A lot of MMORPGs 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.

- **Simplicity vs. Performance:** Our #1 goal when designing uMMORPG was simplicity. We prefer simple and clean code over highly optimized and overly complex code. A 100.000 lines of code MMORPG that can handle 5000 players at once is nice if you have enough people to work on that huge amount of code. But for indie developers a 5000 lines of code Project that only supports 1000 players at once is much more valuable.

- **OnGUI vs. UI:** Unity's old OnGUI system was great, we used it for uMMORPG in the beginning. People started requesting the new UI, so we changed over to that one later. 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.

- **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.

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 select the entity.

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

BuggyQuest.cs

The Quest struct only contains the dynamic quest properties and a name, so that the static properties can be read from the scriptable object. The benefits are low bandwidth and easy Player database saving (saves always refer to the scriptable quest, so we can change that any time).

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

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

BuggySkill.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 Player database saving (saves always refer to the scriptable skill, so we can change that any time).

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-syncliststruct-only-works-with-some-file-names.384582/

CameraMMO.cs

We developed a simple but useful MMORPG style camera. The player can zoom in and out with the mouse wheel and rotate the camera around the hero by holding down the right mouse button.

Note: we turned off the linecast obstacle detection because it would require colliders on all environment models, which means additional physics complexity (which is not needed due to navmesh movement) and additional components on many gameobjects. Even if performance is not a problem, there is still the weird case where if a tent would have a collider, the inside would still be part of the navmesh, but it's not clickable because of to the collider. Clicking on top of that collider would move the agent into the tent though, which is not very good. Not worrying about all those things and having a fast server is a better tradeoff.

ConsoleGUI.cs

People should be able to see and report errors to the developer very easily.

Unity's Developer Console only works in development builds and it only shows errors. This class provides a console that works in all builds and also shows log and warnings in development builds.

Note: we don't include the stack trace, because that can also be grabbed from the log files if needed.

Note: there is no 'hide' button because we DO want people to see those errors and report them back to us.

CopyPosition.cs

This component copies a Transform's position to automatically follow it, which is useful for the camera.

Database.cs

Saves Character Data in XML files. The design is very simple, we store chars in a "Database/Account/Character" file. This way we can get all characters for a certain account very easily without parsing ALL the characters.

benchmarks: saving 10 chars: 0.03s saving 100 chars: 0.45s saving 1000 chars: 1.7s

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 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.

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.

ItemDropChance.cs

Defines the drop chance of an item for monster loot generation.

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 anymore. So we have to use Resources.Load. (before we added them to the dict in OnEnable, but that's only called for those that are referenced in the game. All others will be ignored be Unity.)

A Item can be created by right clicking the Resources folder and selecting Create -> Scriptable Object -> ItemTemplate. Existing items can be found in the Resources folder.

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 walk back to the start position quickly.

- **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 *lootItems* list. Each monster has a list of items with their dropchance, so that loot will always be generated randomly. Monsters can also randomly generate loot gold between a minimum and a maximum amount.

NavmeshPathGizmo.cs

Draws the agent's path as Gizmo.

NetworkManagerMMO.cs

We use a custom NetworkManager that also takes care of login, character selection, character creation and more.

We don't use the playerPrefab, instead all available player classes should be dragged into the spawnable objects property.

NetworkManagerMMOUI.cs

Updates the UI to the current NetworkManager state.

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.

NetworkProximityCheckerCustom.cs

The default NetworkProximityChecker requires a collider on the same object, but in some cases we want to put the collider onto a child object (e.g. for animations).

We 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.

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:

- **Local Chat:** by default, all messages that don't start with a / are addressed to the local chat. If one player writes a local message, then all players around him *(all observers)* will be able to see the message.
- **Whisper Chat:** a player can write a private message to another player by using the **/ name message** format.

- **Guild Chat:** we implemented guild chat support with the **/g message** command. Please note that the guild feature itself is still in development, so the message will not be read by anyone just yet.
- **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.

PlayerEquipmentLocation.cs

Used to find out where an equipped item should be shown in the 3D world. This component can be attached to the shoes, hands, shoulders or head of the player in the Hierarchy. The *acceptedCategory* defines the item category that is accepted in this slot.

Note: modify the equipment location's transform to mirror it if necessary.

PlayerNameColor.cs

Colors the name overlay in case of offender/murderer status.

Projectile.cs

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

QuestTemplate.cs

Saves the quest info in a ScriptableObject that can be used ingame by referencing it from a MonoBehaviour. It only stores an quest'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 quests may not be referenced by any entity ingame (e.g. after a special event). But all quests should still be loadable from the database, even if they are not referenced by anyone anymore. So we have to use Resources.Load. (before we added them to the dict in OnEnable, but that's only called for those that are referenced in the game. All others will be ignored be Unity.)

A Quest can be created by right clicking the Resources folder and selecting Create -> Scriptable Object -> QuestTemplate. Existing quests can be found in the Resources folder.

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 anymore. So we have to use Resources.Load.

(before we added them to the dict in OnEnable, but that's only called for those that are referenced in the game. All others will be ignored be Unity.)

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 -> Scriptable Object -> SkillTemplate. Existing skills can be found in the Resources folder.

UIDragAndDropable.cs

Drag and Drop support for UI elements. Drag and Drop actions will be sent to the GameObject with the 'handlerTag' tag.

UIKeepInScreen.cs

This component can be attached to moveable windows, so that they are only moveable within the Screen boundaries.

UIRefresh.cs

All Player UI logic in one place. We can't attach this component to the player prefab, because prefabs can't refer to Scene objects. So we could either use GameObject.Find to find the UI elements dynamically, or we can have this component attached to the canvas and find the local Player from it.

UIShowToolTip.cs

Instantiates a tooltip while the cursor is over this UI element.

UITextCopyName.cs

UIWindow.cs

Adds window like behaviour to UI panels, so that they can be moved and closed by the user.

Utils.cs

This class contains some helper functions.
