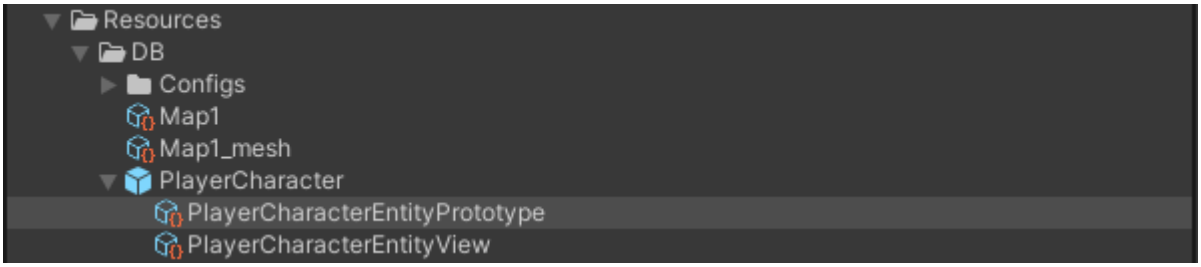
Quantum 104 - Player Spawning

Overview

With the player character entity created the next steps are spawning a player character for each player that joins the game and linking up the input of the player to the entity.

Player Character Prefab

Currently, the character is a scene object. Quantum can spawn entities at runtime similar to how prefabs can be spawned at runtime in a single player Unity game. Create a `PlayerCharacter` prefab by dragging the `PlayerCharacter` GameObject from the scene into the `Resources/DB` folder. After doing so, delete the `PlayerCharacter` from the scene.



Note how a `EntityPrototype` and a `EntityView` file have been created under the prefab. These are used by Quantum to spawn the entity and link it up to the Unity view.

IMPORTANT: All Quantum Prefabs need to be inside the `Resources/DB` folder or a subfolder of it. Prefabs outside won't have an `EntityPrototype` file created.

photon

Player Link Component

Quantum has a concept of a player. Each client can have one or multiple players. However, Quantum does not have a built-in concept of a player object/avatar. Each player that is connected to the game is given a unique ID. This ID is called a `PlayerRef`. To link an entity to a specific player we will create a PlayerLink component that contains the `PlayerRef` of the owner.

Create a `PlayerLink.qtn` file in the `Platformer` folder of the `quantum-code` project and add the following code to it:

C#

```
component PlayerLink
{
    player_ref Player;
}
```

Player Data

To dynamically spawn a character we need to let the gameplay code know what entity to create. Quantum has a player data concept. Player data allows each player to pass information into the simulation on connection. This can be information such as what character a player is playing or what skin they are using or anything else.

The player data can be found in the `RuntimePlayer.User` file. Open the `RuntimePlayer.User` file in the `quantum_code` project and replace its content with:

C#

```
using Photon.Deterministic;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
{
    partial class RuntimePlayer
    {
        public AssetRefEntityPrototype CharacterPrototype;

        partial void SerializeUserData(BitStream stream)
        {
            stream.Serialize(ref CharacterPrototype);
        }
    }
}
```

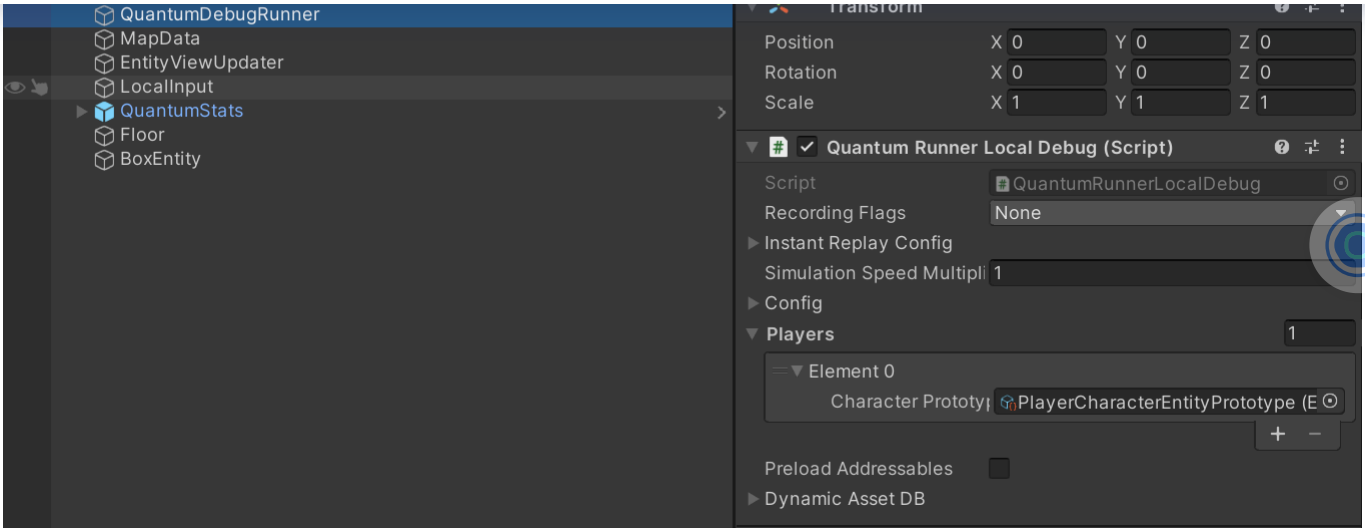
Press build (`ctrl + shift + b`).

This code adds a `AssetRefEntityPrototype` to the player data which is the Quantum equivalent to a prefab. This will later be used to spawn the player character entity.

IMPORTANT: All data added to the `RuntimePlayer` class must be serialized in the `SerializeUserData` function. The `BitStream` provides serialization for all basic types.

When entering play mode the Quantum simulation automatically runs. This is driven by the `QuantumLocalRunnerDebug` component on the `QuantumRunnerDebug` GameObject in the `Game` scene in Unity. This component is used to debug run a single player version of the `Game` locally for development purposes.

The `QuantumLocalRunnerDebug` allows to simulate any numbers of local players. In the `QuantumLocalRunnerDebug` under `Players` add a new entry to the list. The entry contains a `CharacterPrototype` field which is the field that was added to the player data before. Drag and drop the `PlayerCharacterEntityPrototype` file that can be found under the `PlayerCharacter` prefab into the field.



Now that the prefab is linked up to the player data all that is left is to write code to spawn the entity when a player joins.

Spawning Player Objects

Create a new `PlayerSpawnsystem.cs` class. Add the following code:

C#

```
namespace Quantum.Game
{
    unsafe class PlayerSpawnSystem : SystemSignalsOnly, ISignalOn
    {
        public void OnPlayerDataSet(Frame frame, PlayerRef player
        {
            var data = frame.GetPlayerData(player);

            // resolve the reference to the avatar prototype.
            var prototype = frame.FindAsset<EntityPrototype>(data

            // Create a new entity for the player based on the pr
            var entity = frame.Create(prototype);

            // Create a PlayerLink component. Initialize it with
            var playerLink = new PlayerLink()
```

```
                };
                frame.Add(entity, playerLink);

                // Offset the instantiated object in the world, based
                if (frame.Unsafe.TryGetPointer<Transform3D>(entity, o
                {
                    transform->Position.X = (int)player;
                }
            }
        }
    }
```

This code creates the character entity when a player joins and links it up to the player by adding a PlayerLink component to it.

Signals are similar to events in C#. They are used by Quantum systems to communicate with each other. Quantum comes with a lot of existing signals such as the `ISignalOnPlayerDataSet` which gets called after a player has joined the session and shared their player data.

SystemSignalsOnly is a special type of system that doesn't do anything on its own. It allows for the implementation of a system that just listens to signals.

Add the PlayerSpawnSystem to the list of systems in the `SystemSetup.cs` after the `MovementSystem`.

Update MovementSystem

Until now the MovementSystem always moved using inputs from the 0 player using the following code:

C#

```
var input = *f.GetPlayerInput(0);
```

C#

```
Input input = default;
if(f.Unsafe.TryGetPointer(filter.Entity, out PlayerLink* playerLi
{
    input = *f.GetPlayerInput(playerLink->Player);
}
```

Note that the filter has not been adjusted so the system will still filter for entities with a CharacterController but no PlayerLink component. In this case it will use the `default` value for the input. This results in no movement besides gravity being applied.

Getting a component using `TryGet` in Quantum is very fast O(n) because Quantum uses a sparse set ECS.

Press build (`ctrl + shift + b`) and switch to Unity and enter play mode. A player character will be spawned in addition to the character that is already in the scene and it reacts to keyboard inputs.

Back to top

photon

Gaming Circle

Industries Circle

Support

Gaming Circle
Industries Circle
Circle Discord
Circle Stack Overflow

Resources

Dashboard
Samples
SDK Downloads
Cloud Status

Connect

Public Discord
YouTube
Facebook
Twitter
Blog
Contact Us

Languages

English
日本語
한국어
简体中文
繁体中文