Configuration Files

Introduction

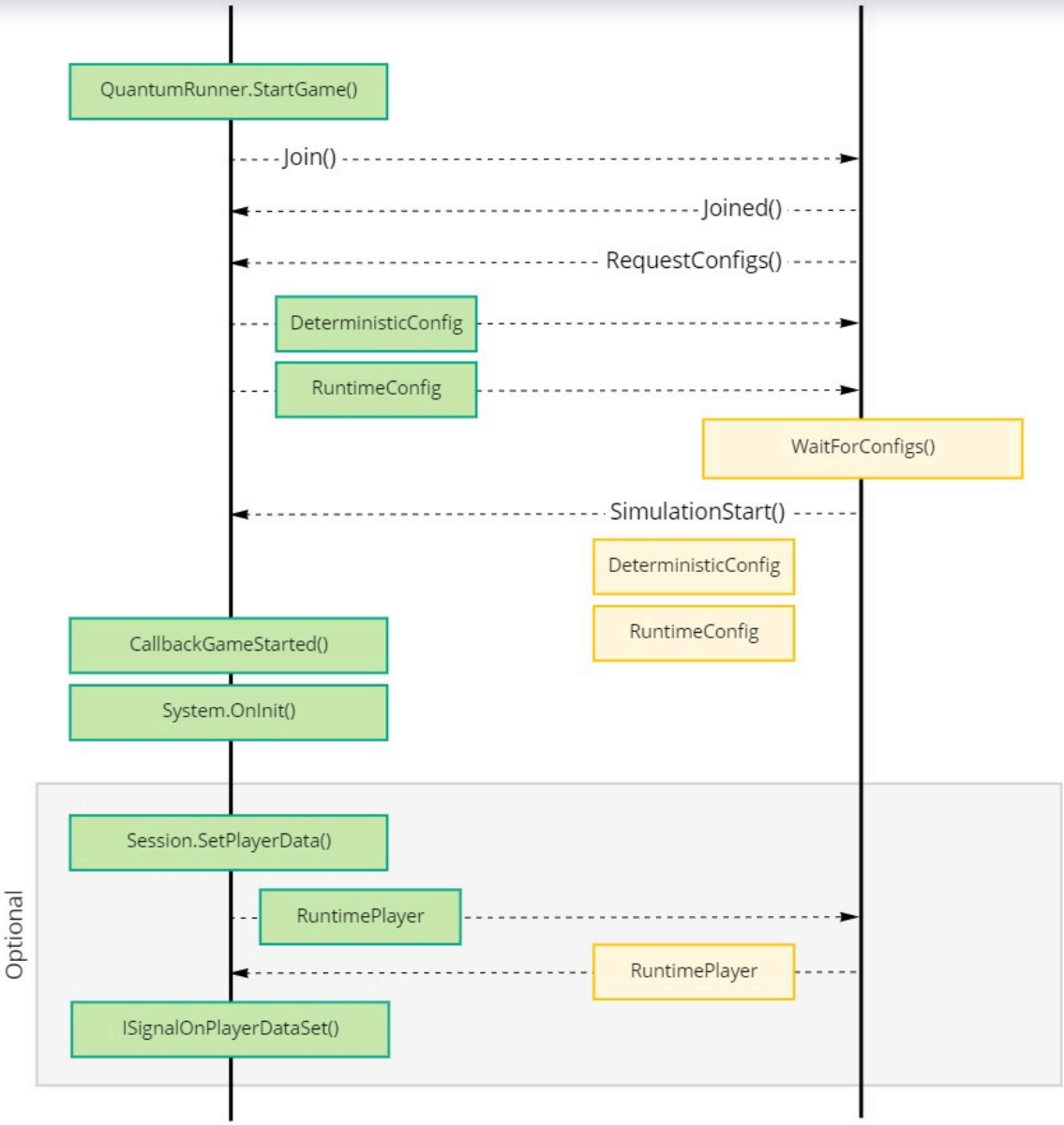
There are a few Quantum config files that have specific roles and purposes.

These config files are placed in different folders in the Unity project. Finding them quickly is made easy with the shortcuts (unity) editor window found in "Menu/Quantum/Show Shortcuts".

Most of default config instances reside as Scriptable Objects inside the "Resources" folder at the root level of the Unity project Assets, and will end up in your app build from there (see DeterministicSessionConfigAsset.Instance for example) while others (RuntimeConfig, RuntimePlayer) can be assembled during run-time.

Quantum Start Sequence

Which config is used by whom and send when is shown in the diagram below.



Config Sequence Diagram

Config Files

PhotonServerSettings

`Assets/Resources/PhotonServerSettings.asset`

Quantum, from version 2.0, uses Photon Realtime to connect and communicate to the Photon Cloud. This config describes where the client connects to (cloud + region, local ip, ..).

Also a valid AppId (referring to an active Quantum plugin) is set here.

Only one instance of this config file is allowed. The loading is tightly integrated into the PhotonNetwork class. See `PhotonNetwork.PhotonServerSettings`.

Photon Server Settings

DeterministicConfig

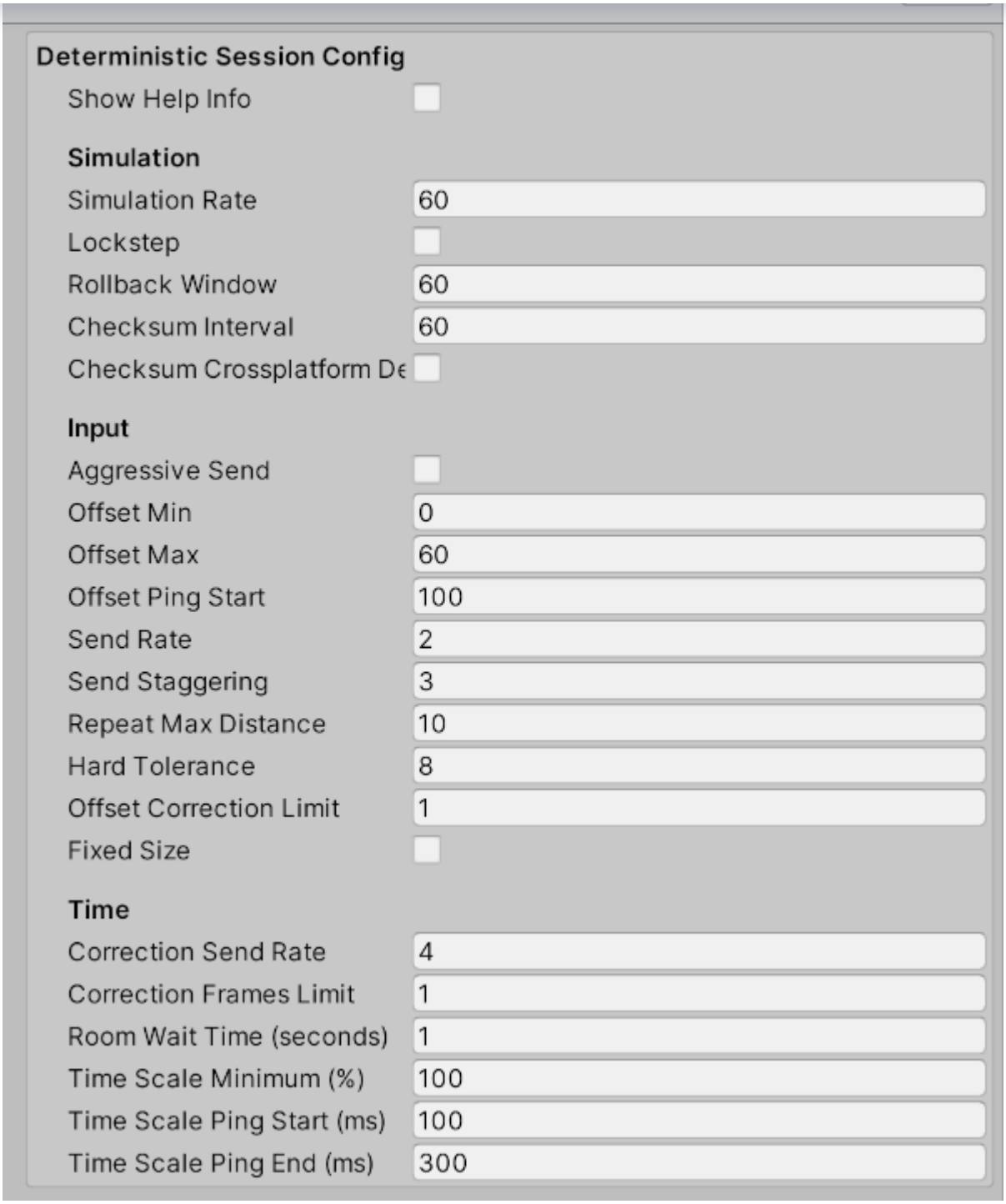
`Assets/Resouces/DeterministicConfig.asset`

details of each parameter.

The default way only allows one instance of this asset but as long as it is passed into `QuantumRunner.StartParameters` it does not matter how the file is retrieved.

This config file will be synchronized between all clients of one session. Although each player starts their own simulation locally with their own version of the DeterministicConfig, the server will distribute the config file instance of the first player who joined the plugin.

The data on this config is included in the checksum generation.



Deterministic Config

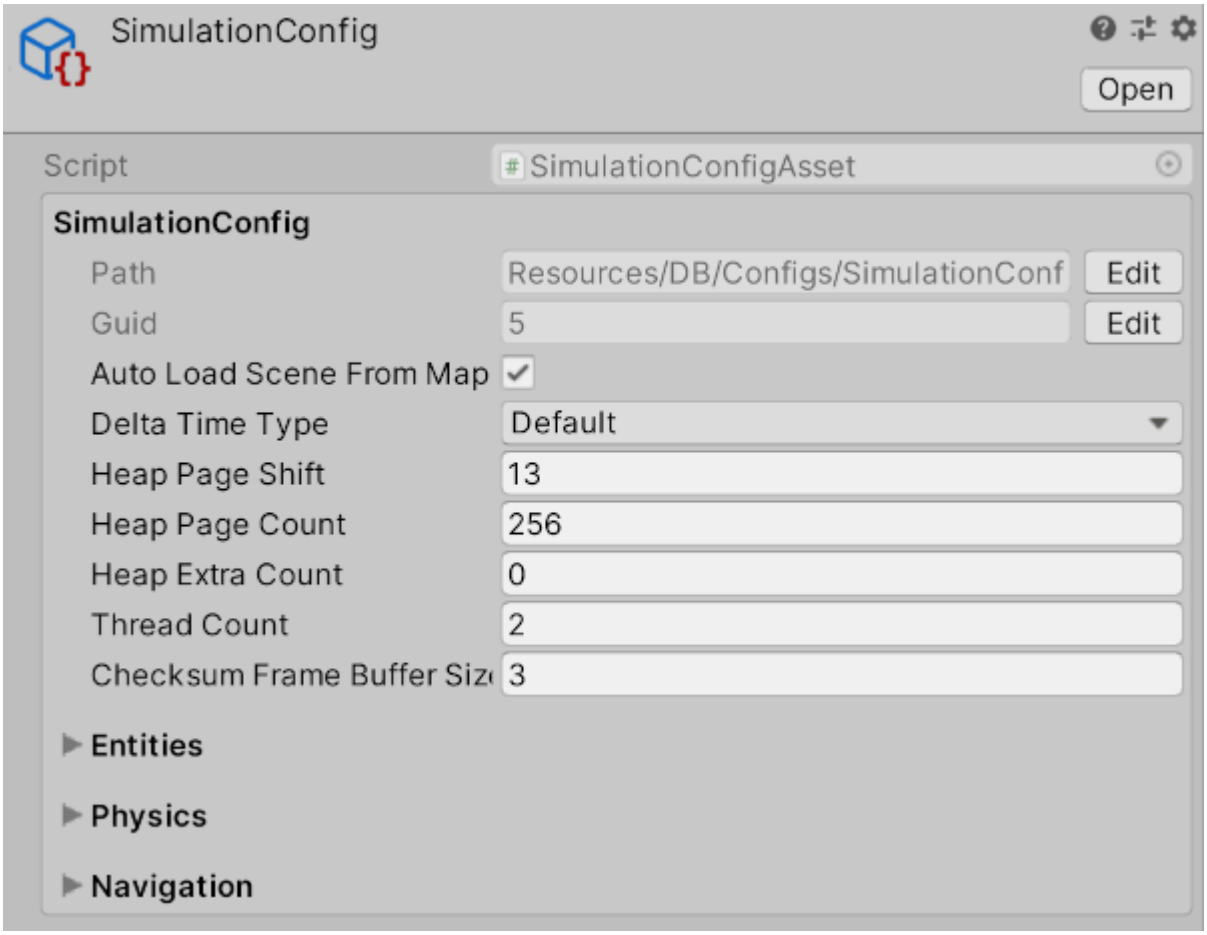
SimulationConfig

`Assets/Resources/DB/Configs/SimulationConfig.asset`

This config file holds parameters used in the ECS layer and inside core systems like physics and navigation. See the related system sections in the manual for more details of each value.

Add the config asset GUID to the RuntimeConfig to select which SimulationConfig should be used.

Developers can "extend" (create a partial class) the `quantum_code/quantum.state/Core/SimulationConfig.cs` class and add more data to it.



Simulation Config

Delta Time Type

You can customize how the QuantumRunner will accumulate elapsed time to update the Quantum simulation (see the QuantumRunner.DeltaTime property).

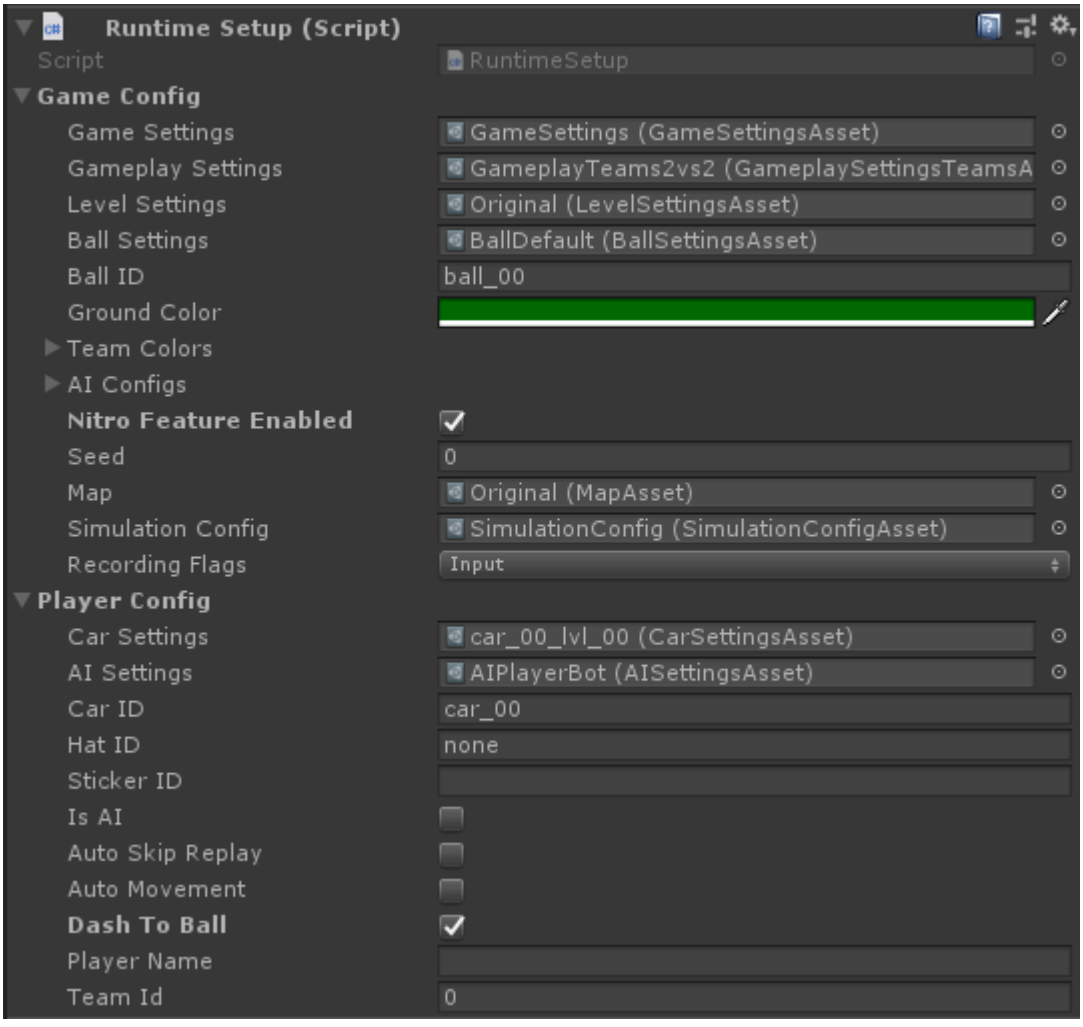
- The `Default` setting will use an internal stopwatch and is the recommended setting for production.
- `EngineDeltaTime` will use, for example Unity.deltaTime, to track when to trigger simulation updates. This is very handy when debugging the project using break points, because upon resuming the simulation it will not fast-forward, but continue from the exact time the simulation was paused. Alas, this setting can cause **issues with time synchronization when initializing online matches**: the time tracking can be inaccurate under load (e.g. level loading) and result in a lot of large extra time syncs request and cancelled inputs for a client when starting an online game.

In contrast to the SimulationConfig, which has only static configuration data, the RuntimeConfig holds information that can be **different from game to game**. By default is defines for example what map to load and the random start seed. It is assembled from scratch each time starting a game.

Developers can add custom data to `quantum_code/quantum.state/RuntimeConfig.User.cs` (don't forget to fill out the serialization methods).

Like the DeterministicConfig this "file" is distributed to every other client after the first player connected and joined the Quantum plugin.

A convenient way of using this config is by creating a MonoBehaviour that stores an instance of RuntimeConfig (and RuntimePlayer) with default values and asset links (GUIDs) for example pointing to other asset files containing specific balancing data. When the player is inside a game lobby parts of the Runtime configs can be overwritten with his custom load-out before connecting and starting the game. See `QuantumRunnerLocalDebug.cs` or the sample below:



Runtime Setup

```
using Quantum;
using UnityEngine;

public sealed class RuntimeSetup : MonoBehaviour
{
  public static RuntimeSetup  Instance { get; private set; }

  public RuntimeConfig GameConfig { get { return _gameConfig;   }
  public RuntimePlayer PlayerConfig { get { return _playerConfig;

  [SerializeField] private RuntimeConfig _gameConfig;
  [SerializeField] private RuntimePlayer _playerConfig;

  private void Awake() {
    Instance = this;
  }
}
```

RuntimePlayer

Similar to the RuntimeConfig the RuntimePlayer describes dynamic properties for one player (quantum_code/quantum.state/RuntimePlayer.User.cs).

The data for a player behaves differently to the other configs, because it is send by each player individually after the actual game has been started. See the *Player* document in the manual for more information.

Using DSL Generated Code With RuntimePlayer and RuntimeConfig Serialization

RuntimeConfig and RuntimePlayer require to write manual serialization code. When using DSL generated structs of component prototypes the serialization code can be simplified.

Caveat: Never use objects that are actually pointers that require a frame to be resolved (e.g. Quantum collections).

```
struct Foo43 {
  int Integer;
  array<Byte>[8] Bytes;
  asset_ref<Map> MapAssetReference;
  Bar43 Bar43;
}

struct Bar43 {
  FPVector3 Vector3;
}

component Component43 {
  int Integer;
  OtherComponent43 OtherComponent;
}

component OtherComponent43 {
  int Integer;
  FP FP;
}
```

The `partial RuntimePlayer.User` implementation looks like this.

C#

```
partial class RuntimePlayer {
  // A) Use a DSL generated struct on RuntimePlayer
  public Foo43 Foo;

  // B) Piggyback on a component prototype to set data
  public Component43_Prototype Component43 = new Component43_Prot

  partial void SerializeUserData(BitStream stream) {
    // A) Because the struct is memory alined we can pin the memo
    unsafe {
    fixed (Foo43* p = &Foo) {
```

```
    }

    // B) Initialized the references in the field declaration wit
    stream.Serialize(ref Component43.Integer);
    stream.Serialize(ref Component43.OtherComponent.Integer);
    stream.Serialize(ref Component43.OtherComponent.FP);
  }
}
```

Send the `RuntimePlayer` from the client:

C#

```
var runtimePlayer = new Quantum.RuntimePlayer {
  Component43 = new Quantum.Prototypes.Component43_Prototype {
    Integer = 1,
    OtherComponent = new Quantum.Prototypes.OtherComponent43_Prot
  Foo = new Foo43 {
    Bar43 = new Bar43 { Vector3 = FPVector3.One },
    Integer = 4,
   MapAssetReference = new AssetRefMap() { Id = 66 }
  }
};

unsafe {
  runtimePlayer.Foo.Bytes[0] = 7;
  runtimePlayer.Foo.Bytes[1] = 6;
}

game.SendPlayerData(lp, runtimePlayer);
```

Back to top

photon

Products

Fusion
Quantum
Realtime
Chat
Voice
PUN

Memberships

Gaming Circle
Industries Circle

Support

Gaming Circle
Industries Circle
Circle Discord
Circle Stack Overflow

Connect

Public Discord
YouTube
Facebook
Twitter
Blog
Contact Us

Documentation

Fusion
Quantum
Realtime
Chat
Voice
PUN
Bolt
Server
VR | AR | MR

Resources

Dashboard
Samples
SDK Downloads
Cloud Status

Languages

English
日本語
한국어
简体中文
繁体中文