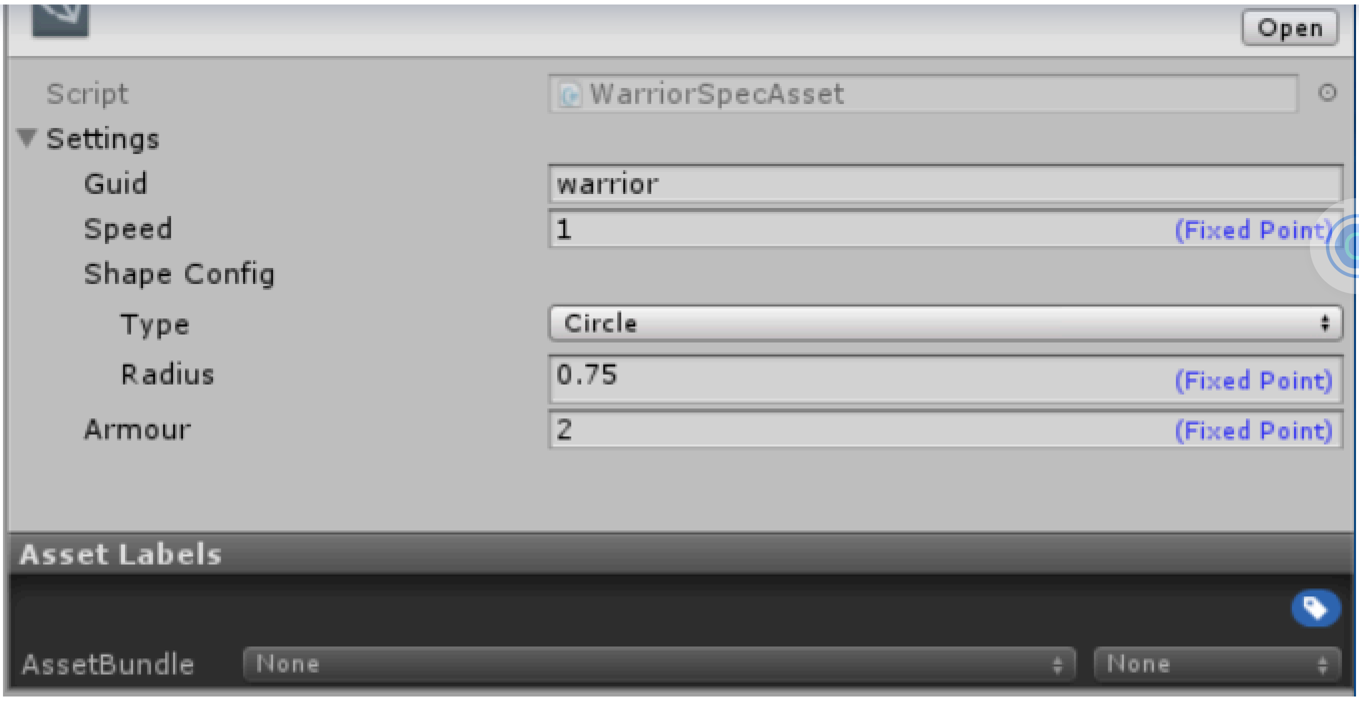
This page is a work in progress and could be pending updates.

Assets in Unity

Overview

Quantum generates a `ScriptableObject`-based wrapper partial class for each each asset type available in Unity. The base class for such wrappers is `AssetBase`. The main class managing `AssetBase` instances is called `UnityDB`.


Script: WarriorSpecAsset

Settings

Guid: warrior

Speed: 1 (Fixed Point)

Shape Config

Type: Circle

Radius: 0.75 (Fixed Point)

Armour: 2 (Fixed Point)

Editing properties of a data asset from Unity.

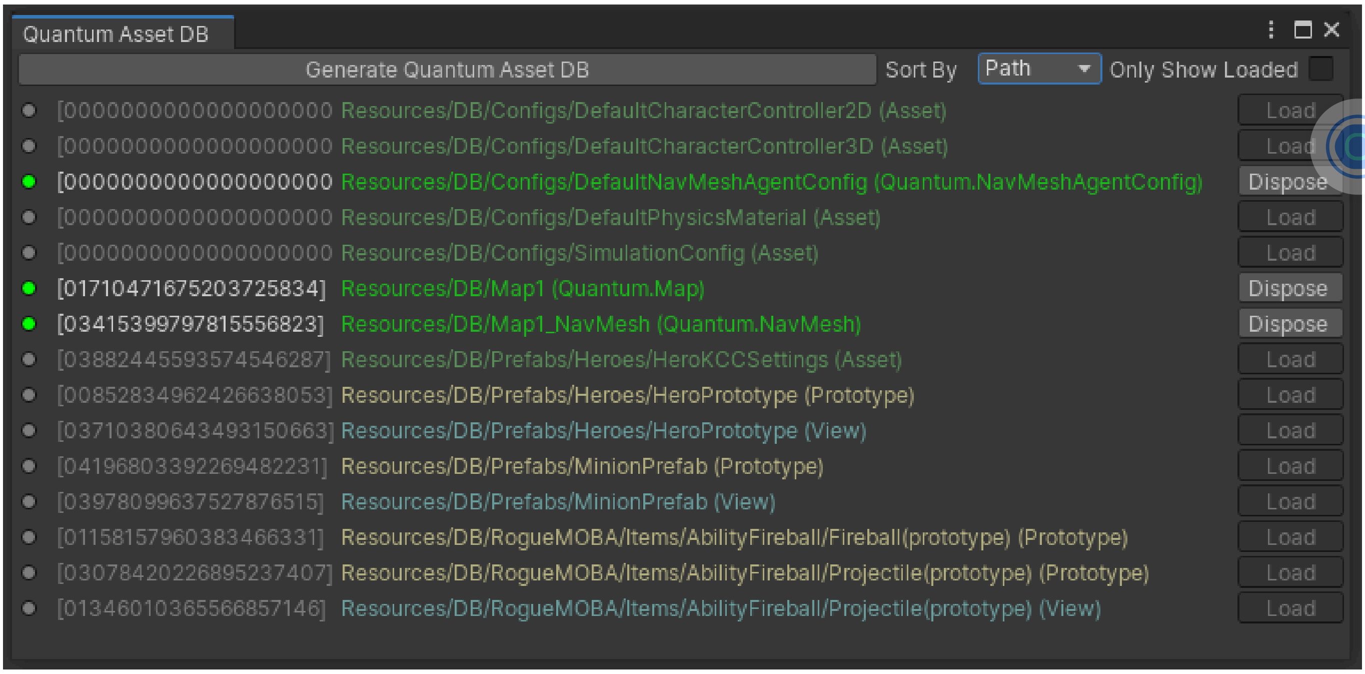
The Quantum SDK will generate the unique GUIDs for each asset.

`AssetGuids` of all the available assets have to be known to the simulation when it starts.

The process for this to happen is:

- In the Editor:
  1. All `AssetBase` assets collected from the locations defined in `QuantumEditorSettings.AssetSearchPaths` (by default `Assets/Resources/DB`).
  2. Each `AssetBase` has a generated entry containing the `AssetGuid` and the information needed to load the `AssetBase` at runtime.
  3. Entries are saved into the `AssetResourceContainer` asset at defined in the `QuantumEditorSettings.AssetResourcePath` (by default `Assets/Resources/AssetResources.asset`).
- At runtime:
  1. The first time any of `UnityDB` members is used `AssetResourceContainer` is loaded using `Resources.Load` (based on `QuantumEditorSettings.AssetResourcePath`).
  2. The list of entries is used to initialize the simulation's `IResourceManager` along with the information needed to load each asset dynamically.

menu items.



●	[0000000000000000000	Resources/DB/Configs/DefaultCharacterController2D (Asset)	Load
●	[0000000000000000000	Resources/DB/Configs/DefaultCharacterController3D (Asset)	Load
●	[0000000000000000000	Resources/DB/Configs/DefaultNavMeshAgentConfig (Quantum.NavMeshAgentConfig)	Dispose
●	[0000000000000000000	Resources/DB/Configs/DefaultPhysicsMaterial (Asset)	Load
●	[0000000000000000000	Resources/DB/Configs/SimulationConfig (Asset)	Load
●	[01710471675203725834]	Resources/DB/Map1 (Quantum.Map)	Dispose
●	[03415399797815556823]	Resources/DB/Map1_NavMesh (Quantum.NavMesh)	Dispose
●	[03882445593574546287]	Resources/DB/Prefabs/Heroes/HeroKCCSettings (Asset)	Load
●	[00852834962426638053]	Resources/DB/Prefabs/Heroes/HeroPrototype (Prototype)	Load
●	[03710380643493150663]	Resources/DB/Prefabs/Heroes/HeroPrototype (View)	Load
●	[04196803392269482231]	Resources/DB/Prefabs/MinionPrefab (Prototype)	Load
●	[03978099637527876515]	Resources/DB/Prefabs/MinionPrefab (View)	Load
●	[01158157960383466331]	Resources/DB/RogueMOBA/Items/AbilityFireball/Fireball(prototype) (Prototype)	Load
●	[03078420226895237407]	Resources/DB/RogueMOBA/Items/AbilityFireball/Projectile(prototype) (Prototype)	Load
●	[01346010365566857146]	Resources/DB/RogueMOBA/Items/AbilityFireball/Projectile(prototype) (View)	Load

Finding Quantum Assets in Unity scripts

Every concrete asset class created by the user gets a corresponding class generated on the Unity side to enable their instantiation as actual Unity Scriptable Objects.

Just to exemplify: a Quantum asset named `CharacterData` gets, in Unity, a class named `CharacterDataAsset`, where the word `Asset` is always the suffix added. The Unity class always contains a Property named `AssetObject` which can be cast to the Quantum class in order to access the simulation specific fields.

Use the `UnityDB` class in order to find assets in the Unity side. Here is a complete snippet of a Quantum asset declaration, and how to access it's fields on Unity:

In the Quantum side:

C#

```
// in any .qtn file:
asset CharacterData;
```

```
public unsafe partial class CharacterData
{
    public FP MaximumHealth;
}
```

In the Unity side:

C#

```
var characterDataAsset = UnityDB.FindAsset<CharacterDataAsset>(my
var characterData = characterDataAsset.Settings;
FP maximumHealth = characterData.MaximumHealth;
```

Finding Assets In the Inspector

It's important to note that when attempting to load a Quantum asset from an editor script, `FindAssetForInspector` should be used instead. *Usage:*

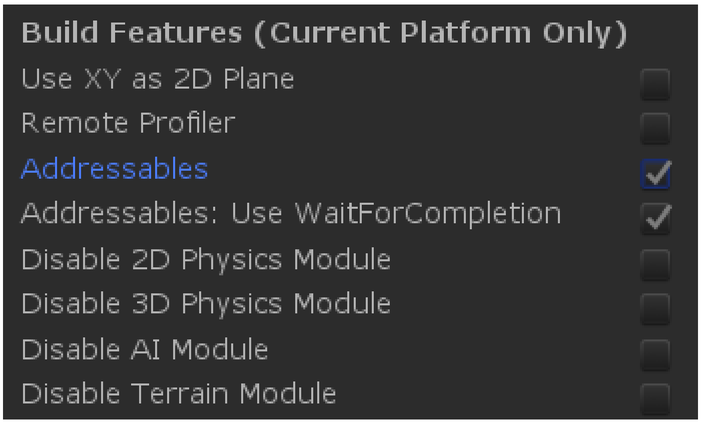
C#

```
public override void OnInspectorGUI()
{
    base.OnInspectorGUI();

      var characterDataAsset = UnityDB.FindAssetAssetForInspector
    var characterData = characterDataAsset.Settings;
    FP maximumHealth = characterData.MaximumHealth;

    // do something

    EditorUtility.SetDirty(characterDataAsset);
}
```

Resources, Addressables and Asset Bundles

Quantum never forms hard-references to `AssetBase` assets. This enables the use of any dynamic content delivery. The following methods of loading assets are supported out of the box:

- Resources
- Addressables (needs to be explicitly enabled)
- Asset Bundles (as a proof-of-concept, due to Asset Bundles demanding highly custom, per-project approach)



Build Features (Current Platform Only)

Use XY as 2D Plane □
Remote Profiler □
Addressables ☑
Addressables: Use WaitForCompletion ☑
Disable 2D Physics Module □
Disable 3D Physics Module □
Disable AI Module □
Disable Terrain Module □

Only needed in Unity versions prior to 2019. Enabling Addressables in QuantumEditorSettings. Alternatively, define `QUANTUM_ADDRESSABLES` and `QUANTUM_ADDRESSABLES_WAIT_FOR_COMPLETION` (for Addressables 1.17 or newer).

There are not any extra steps needed for `AssetBase` to be loadable dynamically using any of the methods above. The details on how to load each asset are stored in `AssetResourceContainer`. This information is accessed when a simulation calls `Frame.FindAsset` or when `UnityDB.FindAsset` is called and leads to an appropriate method of loading being used.

- If an asset is in a `Resource` folder, it will be loaded using the `Resources` API.
- If an asset has an address (explicit or implicit), it will be loaded using the `Addressables` API.
- If an asset belongs to an Asset Bundle (explicitly or implicitly), there will be an attempt to load it using the `AssetBundle` API.

To make the list of the assets (`AssetResourceContainer`) dynamic itself some extra code is needed; pleasr refer to the Updating Quantum Assets At Runtime section for more information.

User scripts can avoid hard references by using `AssetRef` types (e.g. `AssetRefSimulationConfig`) instead of `AssetBase` references (e.g. `SimulationConfig`) to

C#

```
public class TestScript : MonoBehaviour {
  // hard reference
  public SimulationConfigAsset HardRef;
  // soft reference
  public AssetRefSimulationConfig SoftRef;

  void Start() {
    // depending on the target asset's settings, this call may re
    // any of the supported loading methods being used
    SimulationConfigAsset config = UnityDB.FindAsset<SimulationCo
  }
}
```

Drag-And-Dropping Assets In Unity

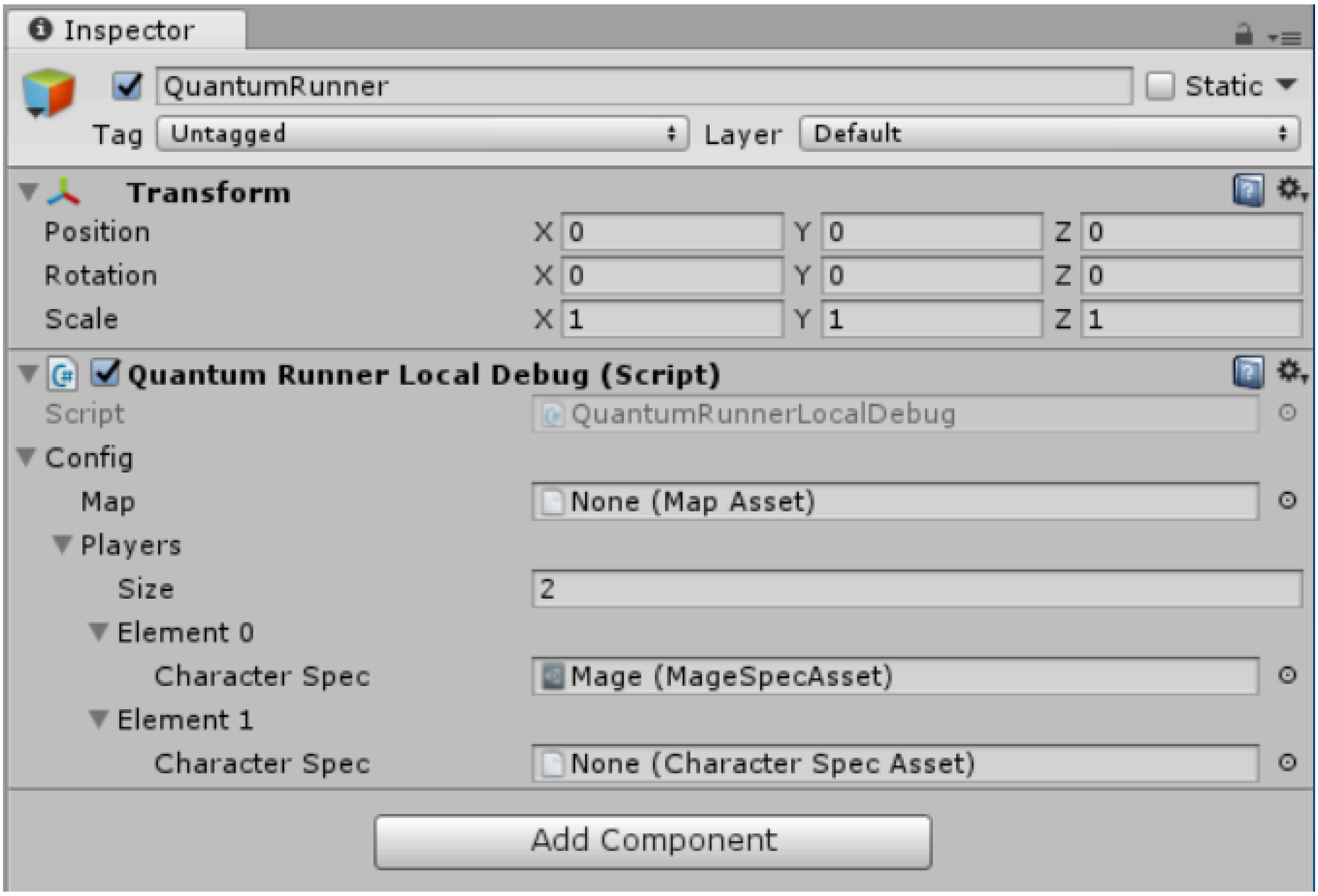
Adding asset instances and searching them through the *Frame* class from inside simulation Systems can only go so far. At convenient solution arises from the ability to have asset instances point to database references and being able to drag-and-drop these references inside Unity Editor.

One common use is to extend the pre-build `RuntimePlayer` class to include an `AssetRef` to a particular `CharacterSpec` asset chosen by a player. The generated and type-safe `asset_ref` type is used for linking references between assets or other configuration objects.

C#

```
// this is added to the RuntimePlayer.User.cs file
namespace Quantum {
  partial class RuntimePlayer {
    public AssetRefCharacterSpec CharacterSpec;

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

This snippet will generate an `asset_ref` which only accepts a link to an asset of type `CharacterSpec`. This field will show up in the Unity inspector and can be populated by drag-and-dropping an asset into the slot.



Inspector

☑ QuantumRunner □ Static

Tag: Untagged Layer: Default

Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

☑ Quantum Runner Local Debug (Script)

Script: QuantumRunnerLocalDebug

Config
- Map: None (Map Asset)
- Players
  - Size: 2
  - Element 0
    - Character Spec: Mage (MageSpecAsset)
  - Element 1
    - Character Spec: None (Character Spec Asset)

Asset ref properties are shown as type-safe slots for Quantum scriptable objects.

Map Asset Baking Pipeline

Another entry point for generating custom data in Quantum is the map baking pipeline. The `MapAsset` is the `AssetBase`-based wrapper for `Map` asset.

The `Map` asset is required by a Quantum simulation and contains basic information such as Navmeshes and static colliders; additional custom data can be saved as part of the asset placed in its custom asset slot - this can be an instance of any custom data asset. The custom asset can be used to store any static data meant to be used during initialization or at runtime. A typical example would be an array of spawn point data such as position, spawned type, etc.

component needs to be present on a `GameObject` in the scene. Once `MapData.Asset` points to a valid `MapAsset`, the baking process can take place. By default, Quantum bakes navmeshes, static colldiers and scene prototypes automatically as a scene is saved or when entering play mode; this behaviour can be changed in `QuantumEditorSettings`.

To assign a custom piece of code to be called every time the a bake happens, create a class inheriting from the abstract `MapDataBakerCallback` class.

C#

```csharp
public abstract class MapDataBakerCallback {
  public abstract void OnBake(MapData data);
  public abstract void OnBeforeBake(MapData data);
  public virtual void OnBakeNavMesh(MapData data) { }
  public virtual void OnBeforeBakeNavMesh(MapData data) { }
}
```

Then override the mandatory `OnBake(MapData data)` and `OnBakeBefore(MapData data)` methods.

C#

```csharp
public class MyCustomDataBaker: MapDataBakerCallback {
  public void OnBake(MapData data) {
    // any custom code to live-load data from scene to be baked i
    // generated custom asset can then be assigned to data.Asset.
  }
  public void OnBeforeBake(MapData data) {

  }
}
```

Preloading Addressable Assets

v1.16 or older

In the Addressables version prior to 1.17, there were no means to load Addressable assets synchronously other than preloading before the simulation started or using Unity's `SyncAddressables` sample.

v1.17 or newer

`WaitForCompletion` was addded in Addressables 1.17 which added the ability to load assets synchronously. To enable it for Quantum, define `QUANTUM_ADDRESSABLES_USE_WAIT_FOR_COMPLETION` or use the toggles in the `QuantumEditorSettings` asset's `Build Features` section.

Although synchronous loading is possible, there are situations in which preloading assets might still be preferable; the `QuantumRunnerLocalDebug.cs` script demonstrates how to achieve this.

Baking AssetBase Load Information

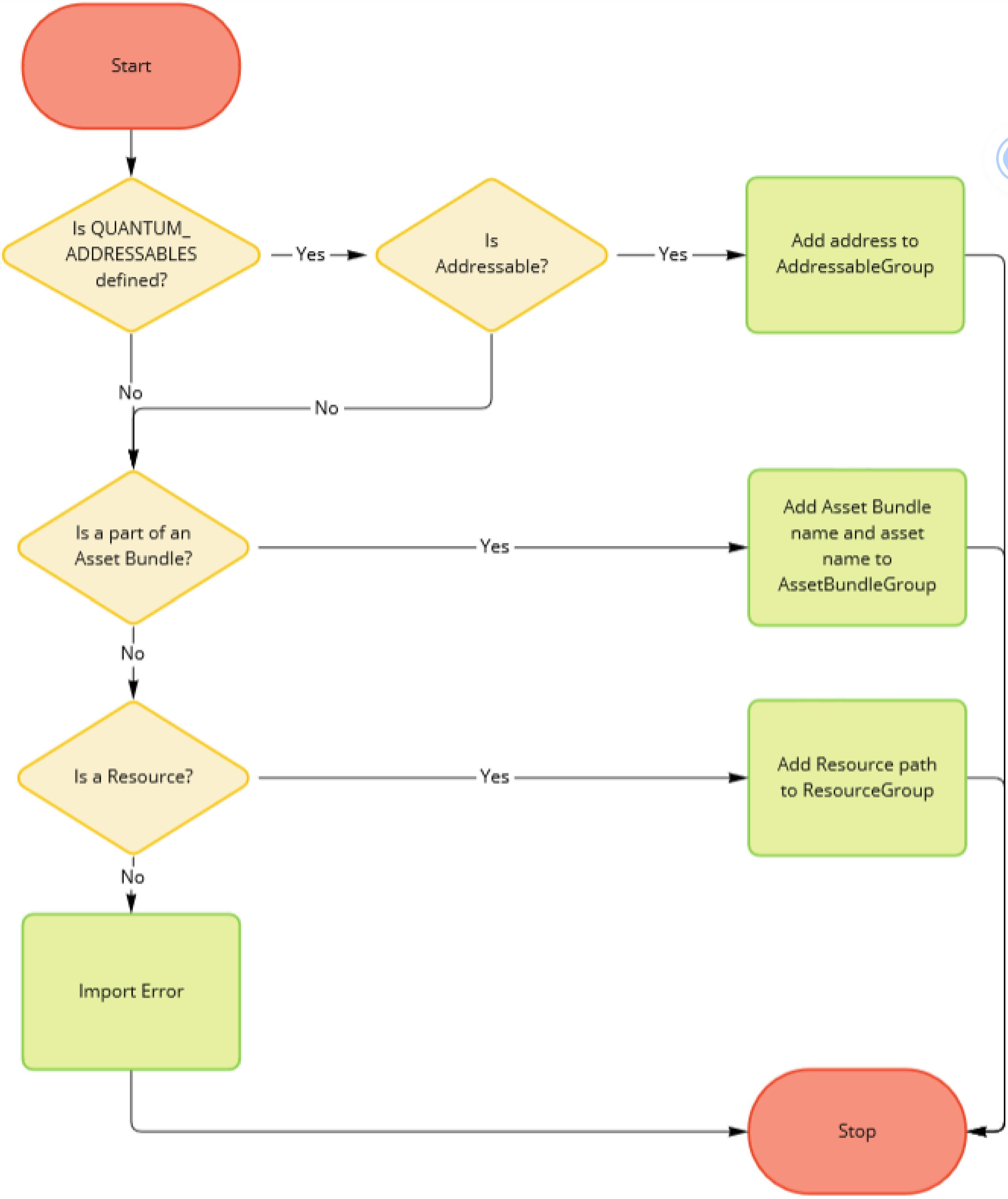
`AssetResourceContainer` is a `ScriptableObject` containing information on how to load each `AssetBase` and maps them to `AssetGuids`.

> **N.B.:** \`AssetBase\` is *not* a Quantum asset itself.

Every time the menu option `Quantum > Generate Asset Resources` is used or an asset in one of `QuantumEditorSettings.AssetSearchPaths` is imported, the `AssetResourceContainer` is recreated in full at the location specified by the `QuantumEditorSettings.AssetResourcePath`.

During the creation of the `AssetResourceContainer`, each `AssetBase` located in any `QuantumEditorSettings.AssetSearchPaths` is assigned to a group. By default, three groups exist:

- `ResourcesGroup`;
- `AssetBundlesGroup`; and,
- `AddressablesGroup`.


Start
Is QUANTUM_ ADDRESSABLES defined?
Yes
Is Addressable?
Yes
Add address to AddressableGroup
No
No
Is a part of an Asset Bundle?
Yes
Add Asset Bundle name and asset name to AssetBundleGroup
No
Is a Resource?
Yes
Add Resource path to ResourceGroup
No
Import Error
Stop

The flow of assigning an asset a group.

An asset is considered Addressable **if**:

- it has an address assigned;
- any of its parent folders is Addressable; or,
- it is nested in another Addressable asset.

To disable baking `AssetBase` when assets are imported, untick `QuantumEditorSettings.UseAssetBasePostprocessor`.

Updating Quantum Assets in Build

It is possible for an external CMS to provide data assets; this is particularly useful for providing balancing updates to an already released game without making create a new build to which players would have to update.

This approach allows balancing sheets containing information about data-driven aspects such as character specs, maps, NPC specs, etc... to be updated independently from the game build itself. In this case, game clients would always try to connect to the CMS service, check for whether there is an update and (if necessary) upgrade their game data to the most recent version before starting or joining online matches.

Updating Existing Assets

The use of Addressables or Asset Bundles is recommended as these are supported out of the box. Any `AssetBase` that is an Addressable or part of an Asset Bundle will get loaded at runtime using the appropriate methods.

To avoid unpredictable lag spikes resulting from downloading assets during the game simulation, consider downloading and preloading your assets as discussed here: *Preloading Addressable Assets*.

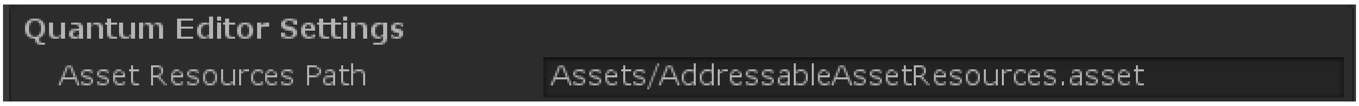
Adding New Assets

The `AssetResourceContainer` generated in the editor will contain the list of all the assets present at its creation. If a project's dynamic content includes adding new Quantum assets during without creating a new build, a way to update the list needs to be implemented.

The recommended approach to achieve this is with an extension of the partial `UnityDB.LoadAssetResourceContainerUser` method. When the first simulation starts or any `UnityDB` method is called, Quantum will make an attempt to load the `AssetResourceContainer`. By default it is assumed the `AssetResourcesContainer` is a Resource located at the

Example Implementation

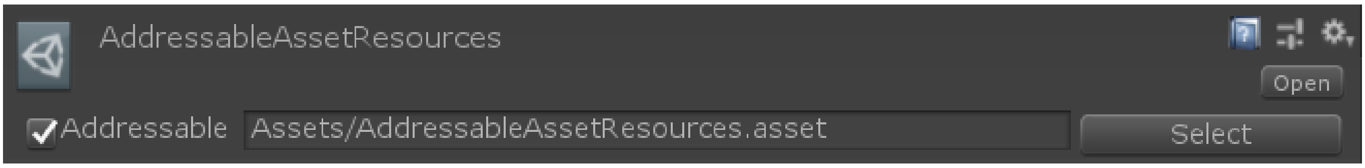
First, the `AssetResourceContainer` needs to be moved out of the `Resources` folder. This is done by setting the `QuantumEditorSettings.AssetResourcePath` :



Quantum Editor Settings

Asset Resources Path: Assets/AddressableAssetResources.asset

Second, the new `AssetResourceContainer` needs to be made into an Addressable:



AddressableAssetResources

☑ Addressable: Assets/AddressableAssetResources.asset

Finally, the snippet implementing the partial method:

C#

```
partial class UnityDB {
  static partial void LoadAssetResourceContainerUser(ref AssetRes
    var path = QuantumEditorSettings.Instance.AssetResourcesPath;

#if UNITY_EDITOR
    if (!UnityEditor.EditorApplication.isPlaying) {
      container = UnityEditor.AssetDatabase.LoadAssetAtPath<Asset
      Debug.Assert(container != null);
      return;
    }
#endif

    var op = Addressables.LoadAssetAsync<AssetResourceContainer>(
    container = op.WaitForCompletion();
    Debug.Assert(container != null);
  }
}
```

management of the `AsyncOperationHandle` returned by `Addressables.LoadAssetAsync` may need to be added.

Adding New Assets With DynamicAssetDB

If new assets can be created in a deterministic way, the `DynamicAssetDB` can be used as discussed here: *Dynamic Assets*.

Improving editor performance when generating multiple assets

Sometimes it might be useful to generate many Quantum assets via code in edit time. During that, the Quantum assets importer will post process every asset instance, which might slow down the overall time taken for the assets to be generate.

In order to speed up this process, it is possible to disable and enable the assets importer at will, by controlling Unity's asset editing pipeline, in order to only perform the post processing step when all the assets are generated.

To do this, use `AssetDatabase.StartAssetEditing` and `AssetDatabase.StopAssetEditing` .

Back to top

photon

Memberships

Gaming Circle
Industries Circle

Server
VR | AR | MR

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

Terms Regulatory Privacy Policy Privacy Code of Conduct Cookie Settings