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

Assets in Simulation

Data Asset Classes

Quantum assets are normal C# classes that will act as immutable data containers during runtime. A few rules define how these assets must be designed, implemented and used in Quantum.

Here is a minimal definition of an asset class (for a character spec) with some simple deterministic properties:

C#

```
namespace Quantum {
  partial class CharacterSpec {
    public FP Speed;
    public FP MaxHealth;
  }
}
```

Creating and loading instances of asset classes into the database (editing from Unity) will be covered later in this chapter.

Using and Linking Assets

To tell Quantum this is an asset class (adding internal meta-data by making it inherit the basic *AssetObject* class and preparing the database to contain instances of it):

C#

```
// this goes into a DSL file
asset CharacterSpec;
```

Asset instances are immutable objects that must be carried as references. Because normal C# object references are not allowed to be included into our memory aligned ECS structs, the ***asset_ref*** special type must be used inside the DSL to declare properties inside the game state (from entities, components or any other transient data structure):

C#

```
component CharacterData {
    // reference to an immutable instance of CharacterSpec (from
    asset_ref<CharacterSpec> Spec;
    // other component data
}
```

To assign an asset reference when creating a Character entity, one option is to obtain the instance directly from the frame asset database and set it to the property:

C#

```
// using the SLOW string path option (fast data driven asset refs
cd->Spec = frame.FindAsset<CharacterSpec>("path-to-spec");
```

The basic use of assets is to read data in runtime and apply it to any computation inside systems. The following example uses the *Speed* value from the assigned *CharacterSpec* to compute the corresponding character velocity (physics engine):

C#

```
// consider cd a CharacterData*, and body a PhysicsBody2D* (from
var spec = frame.FindAsset<CharacterSpec>(cd->Spec.Id);
body->Velocity = FPVector2.Right * spec.Speed;
```

A Note On Determinism

Notice that the above code only **reads** the *Speed* property to compute the desired velocity for the character during runtime, but its value (speed) is never changed.

It is completely safe and valid to switch a game state asset reference in runtime from inside an Update (as asset_ref is a rollback-able type which hence can be part of the game state).

However, changing the **values** of properties of a data asset is NOT DETERMINISTIC (as the internal data on assets is not considered part of the game state, so it is never rolled back).

The following snippet shows examples of what is safe (switching refs) and not safe (changing internal data) during runtime:

C#

```
// cd is a CharacterData*

// this is VALID and SAFE, as the CharacterSpec asset ref is part
cd->Spec = frame.FindAsset<CharacterSpec>("anotherCharacterSpec-p
// (DO NOT do this) changing a value directly in the asset object
spec.Speed = 10;
```

AssetObjectConfig Attribute

You can fine-tune the asset linking script generation with the `AssetObjectConfig` attribute.

C#

```
[AssetObjectConfig(GenerateLinkingScripts = false)]
partial class CharacterSpec {
    // ...
}
```

- **GenerateLinkingScripts** (default=true) - prevent the generation of any scripts that make the asset editable in Unity.
- **GenerateAssetCreateMenu** (default=true) - prevent the generation of the Unity `CreateAssetMenu` attribute for this asset.
- **GenerateAssetResetMethod** (default=true) - prevent the generation of the Unity Scriptable Object `Reset()` method (where a Guid is automatically generated when the asset is created).
- **CustomCreateAssetMenuName** (default=null) - overwrite the `CreateAssetMenu` name. If set to `null` an menu path is generated automatically using the inheritance graph.
- **CustomCreateAssetMenuOrder** (default=-1) - overwrite the `CreateAssetMenu` order. If set to -1 an alphabetical order is used.

Overwrite Asset Script Location And Disable AOT File generation

Overwrite the default asset script destination folder and disable AOT file generation.

See the tool section: Quantum Codegen Unity

Asset Inheritance

The basic step for inheritance is to create an abstract base asset class (we'll continue with our *CharacterSpec* example):

C#

```
namespace Quantum {
  partial abstract class CharacterSpec {
    public FP Speed;
    public FP MaxHealth;
  }
}
```

Concrete sub-classes of *CharacterSpec* may add custom data properties of their own, and must be marked as *Serializable* types:

C#

```
namespace Quantum {
  [Serializable]
  public partial class MageSpec : CharacterSpec {
    public FP HealthRegenerationFactor;
  }

  [Serializable]
  public partial class WarriorSpec : CharacterSpec {
    public FP Armour;
  }
}
```

In the DSL

Once you have declared an asset in the DSL, you can use the `asset_ref<T>` type in the DSL to hold references to the base class and any of its subclasses.

```
component CharacterData {
  // Accepts asset references to CharacterSpec base class and its
  asset_ref<CharacterSpec> ClassDefinition;
  FP CooldownTimer;
}
```

Should you want to keep a reference specific to a sub-class, the derived asset needs to be declared in the DSL first using `asset import` :

C#

```
asset CharacterSpec;
asset import MageSpec;
```

If the derived asset has already been declared in the DSL, simply use `asset_ref<T>` as for the base class. For instance, to use the `MageSpec` directly in the DSL instead of `CharacterSpec` , we would have to write the following:

C#

```
component MageData {
  // Only accepts asset references to MageSpec class.
  asset_ref<MageSpec> ClassDefinition;
  FP CooldownTimer;
}
```

Data-Driven Polymorphism

Having gameplay logic to direct evaluate (in if or switch statements) the concrete *CharacterSpec* class would be very bad design, so asset inheritance makes more sense when coupled with polymorphic methods.

- Operate on transient game state data: that means logic methods in data assets must receive transient data as parameters (either entity pointers or the Frame object itself);
- Only read, never modify data on the assets themselves: assets must still be treated as *immutable* read-only instances;

The following example adds a virtual method to the base class, and a custom implementation to one of the subclasses (notice we use the *Health* field defined for the *Character* entity more to the top of this document):

C#

```
namespace Quantum {
  partial unsafe abstract class CharacterSpec {
    public FP Speed;
    public FP MaxHealth;
    public virtual void Update(Frame f, EntityRef e, CharacterDat
      if (cd->Health < 0)
        f.Destroy(e);
    }
  }

  [Serializable]
  public partial unsafe class MageSpec : CharacterSpec {
    public FP HealthRegenerationFactor;
    // reads data from own instance and uses it to update transie
    public override void Update(Frame f, EntityRef e, CharacterDa
      cd->Health += HealthRegenerationFactor * f.DeltaTime;
      base.Update(f, e, cd);
    }
  }
}
```

To use this flexible method implementation independently of the concrete asset assigned to each *CharacterData*, this could be executed from any System:

C#

```
var spec = frame.FindAsset<CharacterSpec>(cd->Spec.Id);
// Updating Health using data-driven polymorphism (behavior depen
spec.Update(frame, entity, cd);
```

Using DSL Generated Structs In Assets

`Structs` defined in the DSL can also be used on assets. The DSL struct must be annotated with the `[Serializable]` attribute, otherwise the data is not inspectable in Unity.

```
[Serializable]
struct Foo {
  int Bar;
}

asset FooUser;
```

Using the DSL `struct` in a Quantum asset.

C#

```
namespace Quantum {
  public partial class FooUser {
    public Foo F;
  }
}
```

If a struct is not `[Serializable]` -friendly (e.g. because it is an union or contains a Quantum collection), prototype can be used instead:

C#

```
namespace Quantum {
  public partial class FooUser {
    public Foo_Prototype F;
  }
}
```

The prototype can be materialized into the simulation struct when needed:

C#

```
Foo f = new Foo();
fooUser.F.Materialize(frame, ref f, default);
```

Dynamic Assets

Assets can be created at runtime, by the simulation. This feature is called *DynamicAssetDB*.

C#

```
var assetGuid = frame.AddAsset(new MageSpec() {
  Speed = 1,
  MaxHealth = 100,
  HealthRegenerationFactor = 1
});
```

Such asset can be loaded and disposed of just like any other asset:

C#

```
MageSpec asset = frame.FindAsset<MageSpec>(assetGuid);
```

Dynamic assets are not synced between peers. Instead, the code that creates new assets needs to be deterministic and ensure that each peer will generate an asset using the same values.

The only exception to the rule above is when there is a late-join - the new client will receive a snapshot of the *DynamicAssetDB* along with the latest frame data. Unlike serialization of the frame, serialization and deserialization of dynamic assets is delegated outside of the simulation, to `IAssetSerializer` interface. When run in Unity, `QuantumUnityJsonSerializer` is used by default: it is able to serialize/deserialize any Unity-serializable type.

Initializing DynamicAssetDB

Simulation can be initialized with preexisting dynamic assets. Similar to adding assets during the simulation, these need to be deterministic across clients.

First, an instance of `DynamicAssetDB` needs to be created and filled with assets:

C#

```
var initialAssets = new DynamicAssetDB();
initialAssets.AddAsset(new MageSpec() {
    HealthRegenerationFactor = 10
});
initialAssets.AddAsset(new WarriorSpec() {
    Armour = 100
});
...
```

Second, `QuantumGame.StartParameters.InitialDynamicAssets` needs to be used to pass the instance to a new simulation. In Unity, since it is the `QuantumRunner` behaviour that manages a `QuantumGame`, `QuantumRunner.StartParamters.InitialDynamicAssets` is used instead.

Built in Assets

Quantum also comes shipped with several built-in assets, such as:

management setup, heap configuration, thread count and Physics/Navigation settings;

- **DeterministicConfig** - specifies details on the game session, such as it's simulation rate, the checksum interval and lots of configuration regarding Input related to both the client and the server;
- **QuantumEditorSettings** - has the definition for many editor-only details, like the folder the DB should be based in, the color of the Gizmos and auto build options for automatically baking maps, nav meshes, etc;
- **BinaryDataAsset** - an asset that allows the user to reference arbitrary binary information (in the form of a `byte[]`). For example, by default the Physics and the Navigation engines uses binary data assets to store information like the static triangles data. This asset also has built in utilities to compress and decompress the data using gzip.
- **CharacterController3DConfig** - config asset for the built in 3D KCC.
- **CharacterController2DConfig** - config asset for the built in 2D KCC.
- **PhysicsMaterial** - defines a `Physics Material` for Quantum's 3D physics engine.
- **PolygonCollider** - defines a `Polygon Collider` for Quantum's 2D physics engine.
- **NavMeshAsset** - defines a `NavMesh` used by Quantum's navigation system.
- **NavMeshAgentConfig** - defines a `NavMesh Agent Config` for Quantum's navigation system.
- **MapAsset** - stores many static per-scene information such as Physics settings, colliders, NavMesh settings, links, regions and also the Scene Entity Prototypes on that Map. Every Map is correlated with a single Unity scene.

Back to top

photon

We Make Multiplayer Simple

Products

Fusion
Quantum
Realtime
Chat
Voice
PUN

Documentation

Fusion
Quantum
Realtime
Chat
Voice
PUN
Bolt

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

Terms Regulatory Privacy Policy Privacy Code of Conduct Cookie Settings