photon

Search

FUSION

QUANTUM

API Reference

Getting Started

Quantum 100

Game Samples

Technical Samples

AddOns

Manual

- Quantum Ecs
  - DSL (game state)
  - Systems (game logic)
  - Components
  - Events & Callbacks
  - Fixed Point
- Animation
- Assets
- Cheat Protection
- Commands
- Configuration Files
- Custom Server Plugin
- Entity Prototypes
- Entity View
- Frames

Input
Materialization
Map Baking
Multi-Client Runner
Navigation
Physics
Player
Prediction Culling
Profiling
Quantum Project
Replays
RNG Session
WebGL

Concepts And Patterns

Consoles

Gaming Circle

Reference

REALTIME

CHAT

VOICE

SERVER

PUN

BOLT

Systems (game logic)

- Introduction
- Core Systems
- Basic System
- System Setup
  - Activating And Deactivating Systems
  - Special System Types
  - System Groups
- Entity Lifecycle API
  - The EntityRef Type
  - Filters
  - Pre-Built Assets And Config Classes
  - Assets Database
- Signals
  - Generated And Pre-Built Signals
- Triggering Events
- Extra Frame API Items
- Optimization By Scheduling

Introduction

Systems are the entry points for all gameplay logic in Quantum.

They are implemented as normal C# classes, although a few restrictions apply for a System to be compliant with the predict/rollback model:

- Have to be stateless (gameplay data - an instance of the Frame class - will be passed as a parameter by Quantum's simulator to every system Update);
- Implement and/or use only deterministic libraries and algorithms (we provide libraries for fixed point math, vector math, physics, random number generation, path finding, etc);
- Reside in the Quantum namespace;

There are three base system classes one can inherit from:

- `SystemMainThread` : for simple gameplay implementation (init and update callbacks + signals).
- `SystemSignalsOnly` : update-less system just to implement signals (reduces overhead by not scheduling a task for it).
- `SystemBase` : advanced uses only, for scheduling parallel jobs into the task graph (not covered in this basic manual).

photon

Core Systems

By default the Quantum SDK includes all *Core* systems in the `SystemSetup`.

- `Core.CullingSystem2D()`: Culls entities with a `Transform2D` component in predicted frames.
- `Core.CullingSystem3D()`: Culls entities with a `Transform3D` component in predicted frames.
- `Core.PhysicsSystem2D()`: Runs physics on all entities with a `Transform2D` AND a `PhysicsCollider2D` component.
- `Core.PhysicsSystem3D()`: Runs physics on all entities with a `Transform3D` AND a `PhysicsCollider3D` component.
- `Core.NavigationSystem()`: Used for all NavMesh related components.
- `Core.EntityPrototypeSystem()`: Creates, Materializes and Initializes `EntityPrototypes`.
- `Core.PlayerConnectedSystem()`: Used to trigger the `ISignalOnPlayerConnected` and `ISignalOnPlayerDisconnected` signals.
- `Core.DebugCommand.CreateSystem()`: Used by the state inspector to send data to instantiate / remove / modify entities on the fly (*Only available in the Editor!*).

All systems are included by default for the user's convenience. Core systems can be selectively added / removed based on the game's required functionalities; e.g. only keep the `PhysicsSystem2D` or `PhysicsSystem3D` based on whether the game is 2D or 3D.

Back To Top

Basic System

A most basic System in Quantum is a C# class that inherits from `SystemMainThread`. The skeleton implementation requires at least the Update callback to be defined:

```
namespace Quantum
{
  public unsafe class MySystem : SystemMainThread
  {
    public override void Update(Frame f)
    {
    }
  }
}
```

These are the callbacks that can be overridden in a System class:

- `OnInit(Frame f)`: called only once, when the gameplay is being initialized (good place to set up game control data, etc);
- `Update(Frame f)`: used to advance the game state (game loop entry point);
- `OnDisabled(Frame f)` and `OnEnabled(Frame f)`: called when a system is disabled/enabled by another system;

Notice that all available callbacks include the same parameter (an instance of Frame). The Frame class is the container for all the transient and static game state data, including entities, physics, navigation and others like immutable asset

guarantees determinism if all (mutable) game state data is fully contained in the Frame instance.

It is valid to create read-only constants or private methods (that should receive all need data as parameters).

The following code snippet shows some basic examples of valid and not valid (violating the stateless requirement) in a System:

```
namespace Quantum
{

  public unsafe class MySystem : SystemMainThread
  {
    // this is ok
    private const int _readOnlyData = 10;
    // this is NOT ok (this data will not be rolled back, so it would lead to instant
    private int _transientData = 10;

    public override void Update(Frame f)
    {
        // ok to use a constant to compute something here
        var temporaryData = _readOnlyData + 5;

        // NOT ok to modify transient data that lives outside of the Frame object:
        _transientData = 5;
    }
  }

}
```

Back To Top

System Setup

Concrete System classes must be injected into Quantum's simulator during gameplay initialization. This is done in the `SystemSetup.cs` file :

```
namespace Quantum
{
  public static class SystemSetup
  {
    public static SystemBase[] CreateSystems(RuntimeConfig gameConfig, SimulationConfi
    {
      return new SystemBase[]
      {
        // pre-defined core systems
        new Core.PhysicsSystem(),
        new Core.NavMeshAgentSystem(),
        new Core.EntityPrototypeSystem(),

        // user systems go here
        new MySystem(),
```

```
}
```

Notice that Quantum includes a few pre-built Systems (entry point for the physics engine updates, navmesh and entity prototype instantiations).

To guarantee determinism, the order in which Systems are inserted will be the order in which all callbacks will be executed by the simulator on all clients. So, to control the sequence in which your updates occur, just insert your custom systems in the desired order.

Back To Top

Activating And Deactivating Systems

All injected systems are active by default, but it is possible to control their status in runtime by calling these generic functions from any place in the simulation (they are available in the Frame object):

```
public override void OnInit(Frame f)
{
  // deactivates MySystem, so no updates (or signals) are called in it
  f.SystemDisable<MySystem>();
  // (re)activates MySystem
  f.SystemEnable<MySystem>();
  // possible to query if a System is currently enabled
  var enabled = f.SystemIsEnabled<MySystem>();
}
```

Any System can deactivate (and re-activate) another System, so one common pattern is to have a main controller system that manages the active/inactive lifecycle of more specialized Systems using a simple state machine (one example is to have an in-game lobby first, with a countdown to gameplay, then normal gameplay, and finally a score state).

To make a system start disabled by default override this property:

```
public override bool StartEnabled => false;
```

Back To Top

Special System Types

Although you are likely to use the default `SystemMainThread` type for most of your systems, Quantum offers several alternative options for specialized systems.

System	Description
SystemMainThread	Most common system type. Implements a regular Update() with all the usual features.

SystemSignalsOnly	Does not have an Update() function. It is meant for systems that focus solely on implementing and receiving signals from other systems. By avoiding the Update loop, it helps you save some overhead
SystemMainThreadFilter	This type of system uses a FilterStruct of type T to filter a set of entities based on it, loop through them and calls a method. The Any and Without virtual functions can be overriden for more advanced filtering. If you need a more complex option, we advise you to inherit from SystemMainThread and iterate through the filter yourself (See the Components page for more information on FilterStructs and Filters).

Back To Top

System Groups

Systems can be setup and processed as a group.

The first step involves creating a class inheriting from `SystemMainThreadGroup` .

```
namespace Quantum
{
  public class MySystemGroup : SystemMainThreadGroup
  {
    public MySystemGroup(string update, params SystemMainThread[] children) : base(upd
    {
    }
  }
}
```

The `MySystemGroup` system can now be used in `SystemSetup.cs` to group systems together. System groups can be used in mixed and matched with regular systems.

```
namespace Quantum {
  public static class SystemSetup {
    public static SystemBase[] CreateSystems(RuntimeConfig gameConfig, SimulationConfi
      return new SystemBase[] {

        new MyRegularSystem(),

        new MySystemGroup("Gameplay Systems", new MyMovementSystem(), new MyOrbitScanS
      };
    }
  }
}
```

they each need their own implementation to allow enabling / disabling multiple system groups independently.

Back To Top

Entity Lifecycle API

This section uses the direct API methods for entity creation and composition. Please refer to the chapter on entity prototypes for the the data-driven approach.

To create a new entity instance, just use this (method returns an EntityRef):

```
var e = frame.Create();
```

Entities do not have pre-defined components any more, to add a Transform3D and a PhysicsCollider3D to this entity, just type:

```
var t = Transform3D.Create();
frame.Set(e, t);

var c =  PhysicsCollider3D.Create(f, Shape3D.CreateSphere(1));
frame.Set(e, c);
```

These two methods are also useful:

```
// destroys the entity, including any component that was added to it.
frame.Destroy(e);

// checks if an EntityRef is still valid (good for when you store it as a reference in
if (frame.Exists(e)) {
  // safe to do stuff, Get/Set components, etc
}
```

Also possible to check dynamically if an entity contains a certain component type, and get a pointer to the component data directly from frame:

```
if (frame.Has<Transform3D>(e)) {
    var t = frame.Unsafe.GetPointer<Transform3D>(e);
}
```

With ComponentSet, you can do a single check if an entity has multiple components:

```
var components = ComponentSet.Create<CharacterController3D, PhysicsBody3D>();
if (frame.Has(e, components)) {
  // do something
}
```

Removing components dynamically is as easy as:

Back To Top

The EntityRef Type

Quantum's rollback model maintains a variable sized frame buffer; in other words several copies of the game state data (defined from the DSL) are kept in memory blocks at separate locations. This means any pointer to either an entity, component or struct is only valid within a single Frame object (updates, etc).

Entity refs are safe-to-keep references to entities (temporarily replacing pointers) which work across frames, as long as the entity in question still exists. Entity refs contain the following data internally:

- Entity index: entity slot, from the DSL-defined maximum number for the specific type;
- Entity version number: used to render old entity refs obsolete when an entity instance is destroyed and the slot can be reused for a new one.

Back To Top

Filters

Quantum v2 does not have ***entity types***. In the sparse-set ECS memory model, entities are indexes to a collection of components; the ***EntityRef*** type holds some additional information such as versioning. These collections are kept in dynamically allocated sparse sets. Therefore, instead of iterating over a collection of entities, filters are used to create a set of components the system will work on.

```
public unsafe class MySystem : SystemMainThread
{
    public override void Update(Frame f)
    {
        var filtered = frame.Filter<Transform3D, PhysicsBody3D>();

        while (filtered.Next(out var e, out var t, out var b)) {
          t.Position += FPVector3.Forward * frame.DeltaTime;
          frame.Set(e, t);
        }
    }
}
```

For a comprehensive view on how filters are used, please refer to the ***Components*** page.

Back To Top

Pre-Built Assets And Config Classes

Quantum contains a few pre-built data assets that are always passed into Systems through the Frame object.

These are the most important pre-built asset objects (from Quantum's Asset DB):

- `Map` and `NavMesh` : data about the playable area, static physics colliders, navigation meshes, etc... . Custom player data can be added from a data asset slot (will be covered in the data assets chapter);
- `SimulationConfig` : general configuration data for physics engine, navmesh system, etc.

```
// Map is the container for several static data, such as navmeshes, etc
Map map = f.Map;
var navmesh = map.NavMeshes["MyNavmesh"];
```

Back To Top

Assets Database

All Quantum data assets are available inside Systems through the dynamic asset DataBase API. The following snippets (DSL then C# code from a System) shows how to acquire a data asset from the database and assign it to an asset_ref slot into a Character. First you declare the asset in a qtn file, and create a component that can hold it:

```
asset CharacterSpec;

component CharacterData
{
    asset_ref<CharacterSpec> Spec;
    // other data
}
```

Once the asset and component holding a reference to it are declared, you can set the reference in a system like so:

```
// C# code from inside a System

// grabing the data asset from the database, using the unique GUID (long) or path (str
var spec = frame.FindAsset<CharacterData>("path-to-spec");
// assigning the asset reference assuming you have a pointer to CharacterData componen
data->Spec = spec;
```

Data assets are explained in more detail in their own chapter (including options on how to populate it either through Unity scriptable objects - default; custom serializers or procedurally generated content).

Back To Top

Signals

As explained in the previous chapter, signals are function signatures used to generate a publisher/subscriber API for inter-systems communication.

The following example in a DSL file (from the previous chapter):

```
signal OnDamage(FP damage, entity_ref entity);
```

Would lead to this trigger signal being generated on the Frame class (f variable), which can be called from "publisher" Systems:

A "subscriber" System would implement the generated "ISignalOnDamage" interface, which would look like this:

```
namespace Quantum
{
  class CallbacksSystem : SystemSignalsOnly, ISignalOnDamage
  {
    public void OnDamage(Frame f, FP damage, EntityRef entity)
    {
      // this will be called everytime any other system calls the OnDamage signal
    }

  }
}
```

Notice signals always include the Frame object as the first parameter, as this is normally needed to do anything useful to the game state.

Back To Top

Generated And Pre-Built Signals

Besides explicit signals defined directly in the DSL, Quantum also includes some pre-built ("raw" physics collision callbacks, for example) and generated ones based on the entity definitions (entity-type-specific create/destroy callbacks).

The collision callback signals will be covered in the specific chapter about the physics engine, so here's a brief description of other pre-built signals:

- `ISignalOnPlayerDataSet` : called when a game client sends an instance of RuntimePlayer to server (and the data is confirmed/attached to one tick).
- `ISignalOnAdd<T>` , `ISignalOnRemove<T>` : called when a component type T is added/removed to/from an entity.

Back To Top

Triggering Events

Similar to what happens to signals, the entry point for triggering events is the Frame object, and each (concrete) event will result in a specific generated function (with the event data as the parameters).

```
// taking this DSL event definition as a basis
event TriggerSound
{
    FPVector2 Position;
    FP Volume;
}
```

```
// any System can trigger the generated events (FP._0_5 means fixed point value for 0.
f.Events.TriggerSound(FPVector2.Zero, FP._0_5);
```

Important to reinforce that events MUST NOT be used to implement gameplay itself (as the callbacks on the Unity side are not deterministic). Events are just a one-way fine-grained API to communicate the rendering engine of detailed game state updates, so the visuals, sound and any UI-related object can be updated on Unity.

Back To Top

Extra Frame API Items

The Frame class also contains entry points for several other deterministic parts of the API that need to be treated as transient data (so rolled back when needed). The following snippet shows the most important ones:

```
// RNG is a pointer.
// Next gives a random FP between 0 and 1.
// There are also bound options for both FP and int
f.RNG->Next();

// any property defined in the global {} scope in the DSL files is accessed through th
var d = f.Global->DeltaTime;

// input from a player is referenced by its index (i is a pointer to the DSL defined I
var i = f.GetPlayerInput(0);
```

Back To Top

Optimization By Scheduling

To optimize systems identified as performance hotspots a simple modulo-based entity scheduling can help. Using this only a subset of entities are updated while iterating through them each tick.

```
public override void Update(Frame frame) {
  foreach (var (entity, c) in f.GetComponentIterator<Component>()) {
    const int schedulePeriod = 5;
    if (frame.Number % entity.Index == frame.Number % schedulePeriod) {
      // it is time to update this entity
    }
}
```

Choosing a `schedulePeriod` of `5` will make the entity only be updated every 5th tick. Choosing `2` would mean every other tick.

This way the total number of updates is significantly reduced. To avoid updating all entities in **one** tick adding `entity.Index` will make the load be spread over multiple frames.

Deferring the entity update like this has requirements on the user code:

photon

- Using `EntityFilter` may add to the laziness because new information is processed sooner or later for different entities.

The Quantum Navigation system has this feature build-in.

To Document Top