photon

Search

photon

SERVER

PUN

BOLT

VR | AR | MR

Languages English , 日本語 , 한국어 , 繁体中文

FIND HELP ON DISCORD

QUANTUM | v2 switch to V1 switch to V3

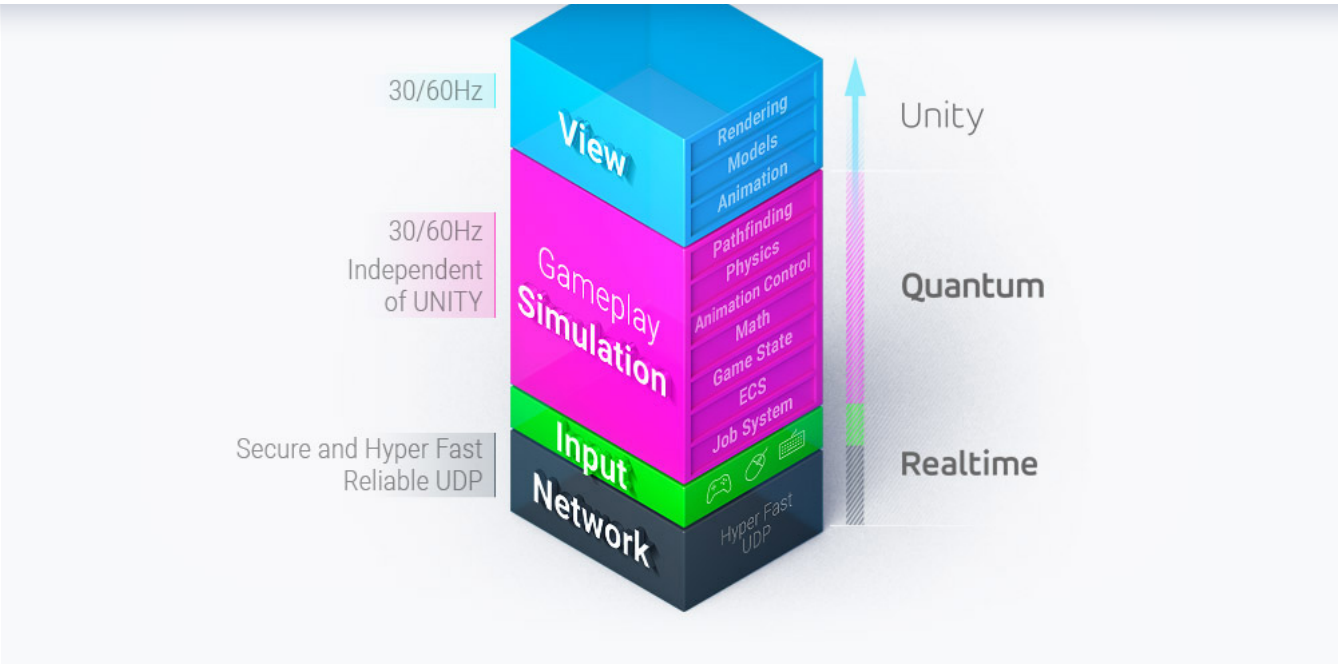
Quantum Intro

Overview

Photon Quantum is a high-performance deterministic ECS (Entity Component System) framework for online multiplayer games made with Unity.

It is based on the predict/rollback approach which is ideal for latency-sensitive online games such as action RPGs, sports games, fighting games, FPS and more.

Quantum also helps the developer to write clean code, fully decoupling simulation logic (Quantum ECS) from view/presentation (Unity), while also taking care of the network implementations specifics (internal predict/rollback + transport layer + game agnostic server logic):



Quantum engine-architecture overview. Gameplay code fully decoupled from presentation logic.

Quantum implements a state-of-the-art tech stack composed of the following pieces:

- Server-managed predict/rollback simulation core.
- Sparse-set ECS memory model and API.
- Complete set of stateless deterministic libraries (math, 2D and 3D physics, navigation, etc.).
- Rich Unity editor integration and tooling.

All built on top of mature and industry-proven existing Photon products and infrastructure (photon realtime transport layer, photon server plugin to host server logic, etc.);

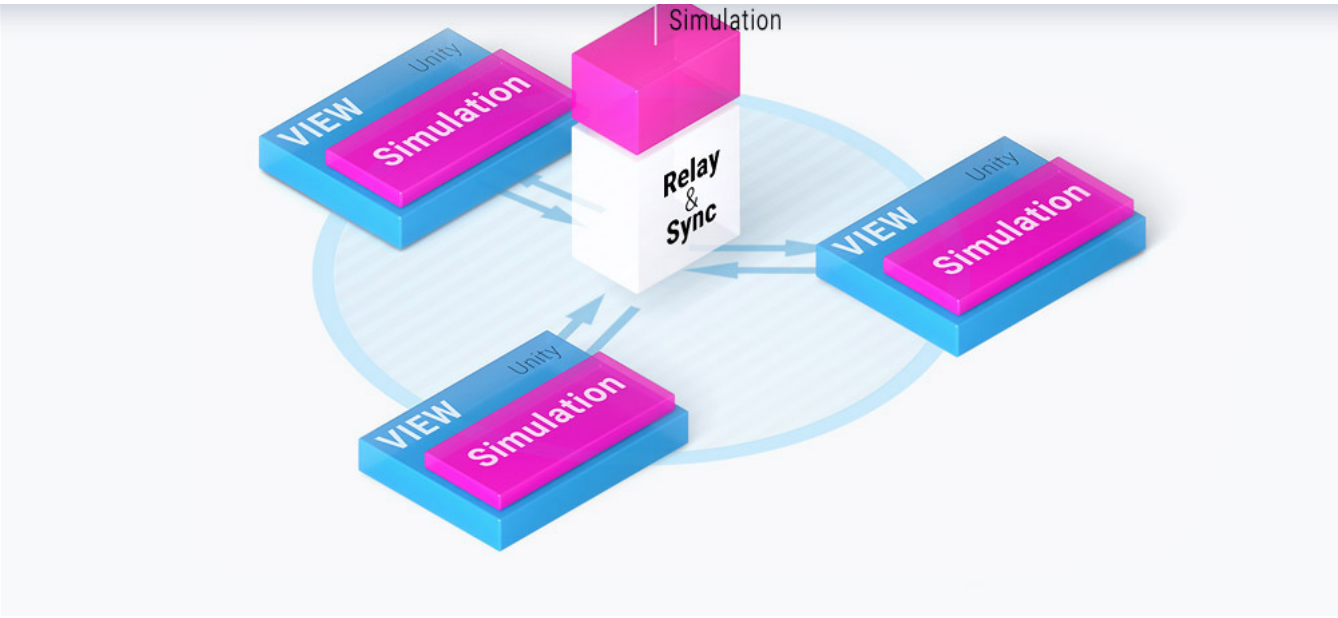
Back To Top

Determinism Without Lockstep

In deterministic systems, game clients only exchange player input with the simulation running locally on all clients. In the past, this has used a lockstep approach, in which game clients would wait for the input from all other players before updating each tick/frame of the simulation.

In Quantum, however, game clients are free to advance the simulation locally using input prediction, and an advanced rollback system takes care of restoring game state and re-simulating any mispredictions.

Because Quantum also relies on a game-agnostic authoritative server component (photon server plugin) to manage input latency and clock synchronization, clients never need to wait for the slowest one to rollback/confirm the simulation forward:



In Quantum, deterministic input exchange is managed via game-agnostic server logic. This prevents game clients with bad network from interfering with the experience of players on good networks.

These are the basic building blocks of a Quantum game:

- Quantum Server Plugin: manages input timing and delivery between game clients, acts as clock synchronization source. Can be extended to integrate with customer-hosted back-end systems (matchmaking, player services, etc.).
- Game Client Simulator: communicates with Quantum server, runs local simulation, performing all input prediction and rollbacks.
- Custom Gameplay Code: developed by the customer as an isolated pure C# simulation (decoupled from Unity) using the Quantum ECS. Besides providing a framework for how to organize high-performance code, Quantum's API offers a great range of pre-built components (data) and systems (logic) that can be reused in any game such as deterministic 3D vector math, 2D and 3D physics engines, navmesh pathfinder, etc.

Back To Top

Old School Coding

Starting from the assumption that all simulation code must be high-performance out of the box, Quantum internal systems are all designed with that in mind from the ground up.

The key to Quantum's high performance is the use of pointer-based C# code combined with its sparse-set ECS memory model (all based memory aligned data-structs and a custom heap allocator - no garbage collection at runtime from simulation code).

The goal is to leave most of the CPU budget for view/rendering code (Unity), including here the re-simulations induced by input mispredictions, inherent to the predict/rollback approach:

Quantum is designed to make your simulation code run as fast as possible, leaving most of the CPU budget for rendering updates.

photon

Back To Top

Code Generation

In Quantum, all gameplay data (game state) is kept either in the sparse-set ECS data structures (entities and components) or in our custom heap-allocator (dynamic collections and custom data), always as blittable memory-aligned C# structs.

To define all data structures that go into that, the developer uses a custom DSL (domain specific language) that lets him concentrate on the game concepts instead of performance-oriented restrictions:

```
// components define reusable game state data groups

component Resources
{
  Int32 Mana;
  FP Health;
}

// structs, c-style unions, enums, flags, etc, can be defined directly from the DSL as
struct CustomData
{
  FP Resources;
  Boolean Active;
}
```

The code-snippet above would generate the corresponding types (with explicit memory alignment), serialization code, and a all boiler plate control logic for special types (like components).

The auto-generated API lets you both query and modify the game state with comprehensive functions to iterate, modify, create or destroy entities (based on composition):

```
var es = frame.Filter<Transform3D, Resources>();
// Next fills in copies of each of the components + the EntityRef
while (es.NextUnsafe(out var entity, out var transform, out var resources)) {
  transform->Position += FPVector3.Forward * frame.DeltaTime;
}
```

Back To Top

Stateless Systems

While Quantum's DSL covers game state data definition with concepts such as entities, components and auxiliary structures (structs, enums, unions, bitsets, collections, etc.), there needs to be a way to organize the custom game logic that will update this game state.

You write custom logic by implementing Systems, which are stateless pieces of logic that will be executed every tick update by Quantum's client simulation loop:

```
  public override void Update(Frame f)
  {
    // your game logic here (f is a reference for the generated game state container).
  }
}
```

The Systems API game loop call order, signals for system intercommunication (both custom and pre-built, such as the physics engine collision callbacks), events and several other extension hooks.

Back To Top

Events

While the simulation is implemented in pure C#, without referencing Unity's API directly, there are two important features to let gameplay code communicate with the rendering engine: events and the asset linking system.

Events are a way for the game code to inform the rendering engine that important things happened during the simulation. One good example is when something results in damage to a character.

Using the state from the previous section as a basis, imagine that damage reduces the health value from the resources component of a character entity. From the Unity rendering scripts, the only noticeable data will be the new health value itself, without any way to know what caused the damage, and also what was the previous health value, etc.

Event definition in a file DSL:

```
event Damage
{
    entity_ref Character;
    FP Amount;
}
```

Gameplay code raises events as a simple API call (generated):

```
public void ApplyDamage(Frame f, EntityRef c, FP amount)
{
  // missing here, the logic to apply damage to the character itself

  // this sends an event to the "view" (Unity)
  f.Events.Damage(amount, c);
}
```

Quantum's event processor will handle all generated events after the tick update is done, taking care of events that require server-confirmed input, event repetitions, and also cancelation/confirmation when simulation rollbacks do occur.

Events raised from the simulation code can then be consumed in runtime from callbacks created in Unity scripts:

```
public void OnDamage(DamageEvent dmg)
{
```

Back To Top

Asset Linking

Unity is known for its flexible editor and smooth asset pipeline. The Asset Linking system allows game and level designers to create and edit data-driven simulation objects from the Unity Editor, which are then fed into the simulation. This is essential to prototype and to add final balancing touches to the gameplay.

From the C# simulation project, the developer creates a data-driven class exposing the desired attributes:

```
public partial class CharacterClass
{
  public Int32 MaxMana;
  public FP MaxHealth;
}
```

Then from Unity, level designers can create as many instances of this asset as needed, each one being automatically assigned with a Unique GUID:

Example of the Asset Linking system: data-driven character-class asset containers being created/modified directly from the Unity Editor.

Then, programmers can use data from these assets directly from inside the simulation:

```
var data = frame.FindAsset<CharacterClass>("character_class_id");
var mana = data.MaxMana;
```

It's also possible to reference these assets directly in components from the state definition DSL:

```
component CharacterAbilities
{
    asset_ref<CharacterClass> CharacterData;
}
```

Back To Top

Deterministic Library

In Quantum, the simulation needs to compute the same results on all clients, given they use the same input values. This means it has to be deterministic, which implies neither using any float or doubles variables, nor anything from the Unity API such as their vectors, physics engine, etc.

To help game developers to implement this style of gameplay, Quantum comes bundled with a flexible and extensible deterministic library, both with a set of classes/struct/functions and also some components that can be used directly

- Deterministic math library: FP (Fixed Point) type (Q48.16) to replace floats/doubles, FPVector2, FPVector3, FPMatrix, FPQuaternion, RNGSession, FPBounds2, and all extra math utils including safe casts, and parsers from native types. The math library is implemented with performance as a primary goal, so we make intense use of inlining, lookup tables and fast operators whenever possible.
- 2D and 3D Physics Engines: high performance stateless 2D/3D physics engines with support for static and dynamic objects, callbacks, joints, etc.
- NavMesh/PathFinder/Agents: includes both an exporter from an existing Unity navmesh or an editor to manipulate the mesh directly. Also includes industry standard HRVO collision avoidance, funneled paths, and many more features.

Back To Top

Where To Go Next

To get started with Quantum we strongly recommend beginning with the Quantum 100 series. This tutorial teaches you all the necessary basics to get started with Quantum. It is available in text and video format.

To Document Top

photon

Circle Discord
Circle Stack Overflow
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