photon

Search

This document is about: **QUANTUM 2** SWITCH TO

Input

Introduction

Input is a crucial component of Quantum's core architecture. In a deterministic networking library, the output of the system is fixed and predetermined given a certain input. This means that as long as the input is the same across all clients in the network, the output will also be the same.

Defining in DSL

Input can be defined in any DSL file. For example, an input struct where you have a movement direction and a singluar jump button would look something like this:

C#

```
input
{
    button Jump;
    FPVector3 Direction;
}
```

The server is responsible for batching and sending down input confirmations for full tick-sets (all player's input). For this reason, this struct should be kept to a minimal size as much as possible.

photon

Commands

Deterministic Commands are another input path for Quantum, and can have arbitrary data and size, which make them ideal for special types of inputs, like "buy this item", "teleport somewhere", etc.

Polling in Unity

To send input to the Quantum simulation, you must poll for it inside of unity. To do this, we subscribe to the `PollInput` callback inside of a MonoBehaviour in our gameplay scene.

C#

```
private void OnEnable()
{
  QuantumCallback.Subscribe(this, (CallbackPollInput callback) =>
}
```

Then, in the callback, we read from out input source and populate our struct.

C#

```
public void PollInput(CallbackPollInput callback)
 {
  Quantum.Input i = new Quantum.Input();

  var direction = new Vector3();
  direction.x = UnityEngine.Input.GetAxisRaw("Horizontal");
  direction.y = UnityEngine.Input.GetAxisRaw("Vertical");

  i.Jump = UnityEngine.Input.GetKey(KeyCode.Space);

  // convert to fixed point.
  i.Direction = direction.ToFPVector3();
```

NOTE: The float to fixed point conversion here is deterministic because it is done before it is shared with the simulation.

Optimization

It is genrally best practice to make sure that your `Input` definition consumes as little bandwidth as possible in order to maximize the amount of players your game can handle. Below are a few ways to optimize it.

Buttons

Instead of using booleans or similar data types to represent key presses, the `Button` type is used inside the Input DSL definition. This is because it only uses one bit per instance, so it is favorable to use where possible. Although they only use one bit over the network, locally they will contain a bit more game state. This is because the single bit is only representative of whether or not the button was pressed during the current frame, the rest of the information is computed locally. Buttons are defined as follows:

C#

```
input
{
    button Jump;
}
```

Encoded Direction

In a typical setting, movement is often represented using a direction vector, often defined in a `DSL` file as such:

```
input
{
    FPVector2 Direction;
}
```

However, `FPVector2` is comprised of two 'FP', which takes up 16 bytes of data, which can be a lot of data sent, especially with many clients in the same room. One such way of optimizing it, is by extending the `Input` struct and encoding the directional vector into a `Byte` instead of sending the full vector every single time. One such implemetation is as follows:

First, we define our input like normal, but instead of including an `FPVector2` for direction, we replace it with a `Byte` where we will store the encoded version.

C#

```
input
{
    Byte EncodedDirection;
}
```

Next, we extend the input struct the same way a component is extended (see: Adding Functionality):

C#

```
namespace Quantum
{
    partial struct Input
    {
        public FPVector2 Direction
        {
            get
            {
                if (EncodedDirection == default)
                    return default;
                return FPVector2.Rotate(FPVector2.Up, angle * FP.
            }
            set
            {
                if (value == default)
                {
                    EncodedDirection = default;
                        return;
                }

                var angle = FPVector2.RadiansSigned(FPVector2.Up,

                angle = (((angle + 360) % 360) / 2) + 1;

                EncodedDirection = (Byte) (angle.AsInt);
            }
        }
    }
}
```

This implementation allows for the same usage as before, but it only takes up a singular byte instead of 16 bytes. It does this by utilzing a `Direction` property, which encodes and decodes the value from `EncodedDirection` automatically. No newline at end of file

Back to top

PUN

PUN
Bolt
Server
VR | AR | MR

Memberships

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