photon

Search

This document is about: **QUANTUM 2**

SWITCH TO

Commands

Introduction

Quantum Commands are an input data paths to Quantum standard APIs. They are similar to Quantum Inputs but **are not** required to be sent every tick.

Quantum Commands are fully reliable. The server will ***always*** accept them and confirm it, regardless of the time at which they are sent. This comes with a trade-off; namely local clients, including the one who sent the command, cannot predict the tick in which the command is received by the simulation. While visual prediction can be shown if needed, the simulation will only receive the Command ***after*** the server has confirmed it as part of a tick.

Commands are implemented as regular C# classes that inherit from `Photon.Deterministic.DeterministicCommand`. They can contain any serializable data.

C#

```
using Photon.Deterministic;
namespace Quantum
{
    public class CommandSpawnEnemy : DeterministicCommand
    {
        public long enemyPrototypeGUID;

        public override void Serialize(BitStream stream)
        {
            stream.Serialize(ref enemyPrototypeGUID);
        }
```

```
            {
                var enemyPrototype = f.FindAsset<EntityPrototype>(ene
                enemyPrototype.Container.CreateEntity(f);
            }
        }
    }
```

Command System Setup in the Simulation

In order for commands to be sent by Quantum, a system needs to be created for them and be made aware of the available commands.

- In Quantum 2.1 and onwards, Commands need to be registered in `CommandSetup.User.cs` .
- In Quantum 2.0, Commands need to be registered in `CommandSetup.cs` .

DeterministicCommandSetup

Commands need to be added to the command factories found in `CommandSetup.User.cs` to be available at runtime.

N.B.: `CommandSetup.Legacy.cs` is not directly used in this setup; however, it needs to be kept in 2.1 for compatibility reasons.

C#

```
// CommandSetup.User.cs

namespace Quantum {
  public static partial class DeterministicCommandSetup {
    static partial void AddCommandFactoriesUser(ICollection<IDete
      // user commands go here
      // new instances will be created when a FooCommand is recei
      factories.Add(new FooCommand());

      // BazCommand instances will be acquired from/disposed back
  }
}

----------

// CommandSetup.Legacy.cs

using Photon.Deterministic;
namespace Quantum {
  public static class CommandSetup {
    public static DeterministicCommand[] CreateCommands(RuntimeCo
      return new null;
    }
  }
}
```

CommandSetup (Only for 2.0!)

In Quantum 2.0 Commands need to be registered in `CommandSetup.cs` to be available at runtime.

N.B.: This system is obsolete in newer versions of Quantum; refer to `DeterministicCommandSetup` for more information.

C#

```
  using Photon.Deterministic;
  namespace Quantum {
    public static class CommandSetup {
      public static DeterministicCommand[] CreateCommands(RuntimeCo
        return new DeterministicCommand[] {

          // user commands go here
          new CommandSpawnEnemy(),
        };
      }
    }
  }
```

Sending Commands From The View

Commands can be send from anywhere inside Unity.

C#

```
using Quantum;
using UnityEngine;

public class UISpawnEnemy : MonoBehaviour
{
    [SerializeField] private EntityPrototypeAsset enemyPrototype
    private PlayerRef _playerRef;

    public void Initialize(PlayerRef playerRef) {
        _playerRef = playerRef;
    }

    public void SpawnEnemy() {
        CommandSpawnEnemy command = new CommandSpawnEnemy()
        {
            enemyPrototypeGUID = enemyPrototype.Settings.Guid.Val
        };
        QuantumRunner.Default.Game.SendCommand(command);
    }
}
```

Sending CompoundCommands From The View

N.B.: This is only available from Quantum 2.1 onwards.

To send multiple commands at once, simply create a `CompoundCommand` and add each individual `DeterministicCommand` to it before sending it.

```
var compound = new Quantum.Core.CompoundCommand();
compound.Commands.Add(new FooCommand());
compound.Commands.Add(new BazCommand());

QuantumRunner.Default.Game.SendCommand(compound);
```

Overloads

`SendCommand()` has two overloads.

C#

```
void SendCommand(DeterministicCommand command);
void SendCommand(Int32 player, DeterministicCommand command);
```

Specify the player index (PlayerRef) if multiple players are controlled from the same machine. Games with only one local player can ignore the player index field.

Polling Commands From The Simulation

To receive and handle Commands inside the simulation poll the frame for a specific player:

C#

```
using Photon.Deterministic;
namespace Quantum
{
    public class PlayerCommandsSystem : SystemMainThread
    {
        public override void Update(Frame f)
        {
                var command = f.GetPlayerCommand(i) as CommandSp
                command?.Execute(f);
            }
        }
    }
}
```

Like any other sytem, the system handling the command polling and consumption needs to be included in `SystemSetup.cs`

C#

```
namespace Quantum {
  public static class SystemSetup {
    public static SystemBase[] CreateSystems(RuntimeConfig gameCo
      return new SystemBase[] {
        // pre-defined core systems
        [...]

        // user systems go here
        new PlayerCommandsSystem(),

      };
    }
  }
}
```

Note

The API does neither enforce, nor implement, a specific callback mechanism or design pattern for Commands. It is up to the developer to chose how to consume, interpret and execute Commands; for example by encoding them into signals, using a Chain of Responsibility, or implementing the command execution as a method in them.

Examples for Collections

C#

```csharp
using System.Collections.Generic;
using Photon.Deterministic;

namespace Quantum
{
    public class ExampleCommand : DeterministicCommand
    {
        public List<EntityRef> Entities = new List<EntityRef>();

        public override void Serialize(BitStream stream)
        {
            var count = Entities.Count;
            stream.Serialize(ref count);
            if (stream.Writing)
            {
                foreach (var e in Entities)
                {
                    var copy = e;
                    stream.Serialize(ref copy.Index);
                    stream.Serialize(ref copy.Version);
                }
            }
            else
            {
                for (int i = 0; i < count; i++)
                {
                    EntityRef readEntity = default;
                    stream.Serialize(ref readEntity.Index);
                    stream.Serialize(ref readEntity.Version);
                    Entities.Add(readEntity);
                }
            }
        }
    }
}
```

photon

C#

```
using Photon.Deterministic;

namespace Quantum
{
    public class ExampleCommand : DeterministicCommand
    {
        public EntityRef[] Entities;

        public override void Serialize(BitStream stream)
        {
            stream.SerializeArrayLength(ref Entities);
            for (int i = 0; i < Cars.Length; i++)
            {
                EntityRef e = Entities[i];
                stream.Serialize(ref e.Index);
                stream.Serialize(ref e.Version);
                Entities[i] = e;
            }
        }
    }
}
```

Compound Command Example

Only one command can be attached to an input stream per tick. Even though a client can send multiple Deterministic Commands per tick, the commands will not reach the simulation at the same tick, rather they will arrive separately on consecutive ticks. To go around this limitation, you can pack multiple Deterministic Commands into a single `CompoundCommand` .

Quantum 2.1 already offers this class. And for previous versions it can be added like this:

C#

photon

```
    public static DeterministicCommandSerializer CommandSerializer;
    public List<DeterministicCommand> Commands = new List<Determini

    public override void Serialize(BitStream stream) {
      if (CommandSerializer == null) {
        CommandSerializer = new DeterministicCommandSerializer();
        CommandSerializer.RegisterPrototypes(CommandSetup.CreateCom
      }

      var count = Commands.Count;
      stream.Serialize(ref count);

      if (stream.Reading) {
        Commands.Clear();
      }

      for (var i = 0; i < count; i++) {
        if (stream.Reading) {
          CommandSerializer.ReadNext(stream, out var cmd);
          Commands.Add(cmd);
        } else {
          CommandSerializer.PackNext(stream, Commands[i]);
        }
      }
    }
  }
```

To then dispatch the compounded commands:

C#

```
  public override void Update(Frame f) {
    for (var i = 0; i < f.PlayerCount; i++) {
        var compoundCommand = f.GetPlayerCommand(i) as CompoundComm
        if (compoundCommand != null) {
          foreach (var cmd in compoundCommand.Commands) {
          }
        }
```

Back to top

We Make Multiplayer Simple

Products

Fusion
Quantum
Realtime
Chat
Voice
PUN

Memberships

Gaming Circle
Industries Circle

Documentation

Fusion
Quantum
Realtime
Chat
Voice
PUN
Bolt
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

Languages

English
日本語
한국어
简体中文

photon