Materialization

Introduction

The process of creating an entity or component instance from a `Component Prototype` or `Entity Prototype` is called **Materialization** .

The materialization of scene prototypes baked into the map asset follow the same rules and execution flow as the materialization of code created instances using the `Frame.Create` API.

Prototype vs Instance

The component instances and entity instances are part of the game state; in other words they can be manipulated at runtime. Components declared in the DSL are used to generate their corresponding `Component Prototypes` . The code generated prototypes follow the naming convention `MyComponent_Prototype` .

`Component Prototypes` and `Entity Prototypes` are both **assets**; this means they are not part of the game state, immutable at runtime and have to be identical for all clients at all time. Each `Component Prototype` has a `ComponentPrototypeRef` which can be used to find it the corresponding asset using the `Frame.FindPrototype<MyComponentName_Prototype>(MyComponentPrototypeRef)` .

Component Prototypes

component or exclude read-only data from the frame to keep the game state slim.

Code generated `Component Prototypes` are partial classes which can be easily extended:

1. Create a C# file called `MyComponentName_Prototype.cs`;
2. Place the body of the script into the `Quantum.Prototypes` namespace;
3. (*Optional*) Add `using Quantum.Inspector;` to have access to the inspector attributes presented in the `Attributes` section of the `Manual \ ECS` page.

It is then possible to add extra data to the `Component Prototype` asset and implement the partial `MaterializeUser()` method to add custom materialization logic.

Example

The following example presents the materialization of the `Vehicle` component as found in the **Arcade Racing Template**.

The `Vehicle` component holds mainly dynamic values computed at runtime. Since these cannot be initialized, the component definition in the DSL uses the `ExcludeFromPrototype` attribute on those parameters to exclude them from the `Vehicle_Prototype` asset designers can manipulate in the Unity editor. The `Nitro` parameter is only part that can be edited to allow designers to decide with how much nitro a specific `Vehicle` is initialized.

C#

```
component Vehicle
{
    [ExcludeFromPrototype]
    ComponentPrototypeRef Prototype;

    [ExcludeFromPrototype]
    Byte Flags;
    [ExcludeFromPrototype]
    FP Speed;
    [ExcludeFromPrototype]
    FP ForwardSpeed;
    [ExcludeFromPrototype]
    FPVector3 EngineForce;
```

```
        [ExcludeFromPrototype]
        FPVector3 AvgNormal;

        [ExcludeFromPrototype]
        array<Wheel>[4] Wheels;

        FP Nitro;
    }
```

The `Vehicle_Prototype` asset is extended to provide designers with customizable read-only parameters. The `Vehicle_Prototype` asset can thus hold shared values for all instances of a specific vehicle entity prototype "type". The `Prototype` parameter in the `Vehicle` component is of type `ComponentPrototypeRef` which is the component specific equivalent to `AssetRef`. To populate it, the partial `MaterializeUser()` method is used to assign the reference of the `Vehicle_Prototype`.

C#

```
    using Photon.Deterministic;
    using Quantum.Inspector;
    using System;

    namespace Quantum.Prototypes
    {
    public unsafe partial class Vehicle_Prototype
    {
        // PUBLIC METHODS

        [Header("Engine")]
        public FP EngineForwardForce = 130;
        public FP EngineBackwardForce = 120;
        public FPVector3 EngineForcePosition;
        public FP ApproximateMaxSpeed = 20;

        [Header("Hand Brake")]
        public FP HandBrakeStrength = 10;
        public FP HandBrakeTractionMultiplier = 1;
        public FP RollingResistance = FP._0_10 * 6;
        public FP DownForceFactor = 0;
        public FP TractionGripMultiplier = 10;
        public FP AirTractionDecreaseSpeed = FP._0_50;

        [Header("Axles")]
        public AxleSetup FrontAxle = new AxleSetup();
        public AxleSetup RearAxle = new AxleSetup();

        [Header("Nitro")]
        public FP MaxNitro = 100;
        public FP NitroForceMultiplier = 2;

        // PARTIAL METHODS
        partial void MaterializeUser(Frame frame, ref Vehicle result,
        {
            result.Prototype = context.ComponentPrototypeRef;
        }

        [Serializable]
        public class AxleSetup
        {
            public FPVector3 PositionOffset;
            public FP Width = 1;
            public FP SpringForce = 120;
            public FP DampingForce = 175;
            public FP SuspensionLength = FP._0_10 * 6;
            public FP SuspensionOffset = -FP._0_25;
        }
    }
    }
```

The parameters in the `Vehicle_Prototype` hold values necessary to compute the dynamic values found in the component instance which impact the behaviour of the entity to which the `Vehicle` component is attached. For example, when a player picks up additional `Nitro`, the value held in the `Vehicle` component is clamped to the `MaxNitro` value found in the `Vehicle_Prototype`. This enforces the limits under penality of desynchronization and keeps the game state slim.

C#

```
{
    public unsafe partial struct Vehicle
    {
        public void AddNitro(Frame frame, EntityRef entity, FP am
        {
            var prototype = frame.FindPrototype<Vehicle_Prototype
            Nitro = FPMath.Clamp(Nitro + amount, 0, prototype.Max
        }
    }
}
```

Materialization Order

Every `Entity Prototype` , including the scene prototypes, the materialization executes the following steps in order:

1. An empty entity is created.
2. For each `Component Prototype` contained in the `Entity Prototype` :
    1. the component instance is created on the stack;
    2. the `Component Prototype` is materialized into the component instance;
    3. (*Optional*) `MaterializeUser()` is called; and,
    4. the component is added to the entity which triggers the `ISignalOnComponentAdded<MyComponent>` signal.
3. `ISignalOnEntityPrototypeMaterialized` is invoked for each materialized entity.
    - Load Map / Scene: the signal is invoked for all entity & `Entity Prototype` pair after all scene prototypes have been materialized.
    - Created with `Frame.Create()` : the signal is invoked immediately after the prototype has been materialized.

The `Component Prototype` materialization step materializes default components in a predetermined order.

C#

```
Transform2D
Transform3D
```

```
PhysicsBody2D
PhysicsCollider3D
PhysicsBody3D
PhysicsJoints2D
PhysicsJoints3D
PhysicsCallbacks2D
PhysicsCallbacks3D
CharacterController2D
CharacterController3D
NavMeshPathfinder
NavMeshSteeringAgent
NavMeshAvoidanceAgent
NavMeshAvoidanceObstacle
View
MapEntityLink
```

Once all default components have been materialized, the user defined components are materialized in alphabetically order.

C#

```
MyComponentAA
MyComponentBB
MyComponentCC
...
```

Back to top

We Make Multiplayer Simple

Products

Documentation

photon

Realtime
Chat
Voice
PUN

Realtime
Chat
Voice
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