photon

Search

FUSION

QUANTUM

photon

DSL (game state)

Introduction

Quantum requires components and other runtime game state data types to be declared with its own DSL (domain-specific-language).

photon

compiler will merge them accordingly).

The goal of the DSL is to abstract away from the developer the complex memory alignment requirements imposed by Quantum's ECS sparse set memory model, required to support the deterministic predict/rollback approach to simulation.

This code generation approach also eliminates the need to write "boiler-plate" code for type serialization (used for snapshots, game saves, killcam replays), checksumming and other functions, like printing/dumping frame data for debugging purposes.

The Quantum DSL Integration section shows how to add `qtn` files to the workflow.

Back To Top

Components

Components are special structs that can be attached to entities, and used for filtering them (iterating only a subset of the active entities based on its attached components). This is a basic example definition of a component:

```
component Action
{
    FP Cooldown;
    FP Power;
}
```

These will be turned into regular C# structs. Labelling them as components (like above) will generate the appropriate code structure (marker interface, id property, etc).

Aside from custom components, Quantum comes with several pre-built ones:

- Transform2D/Transform3D: position and rotation using Fixed Point (FP) values;
- PhysicsCollider, PhysicsBody, PhysicsCallbacks, PhysicsJoints (2D/3D): used by Quantum's stateless physics engines;
- PathFinderAgent, SteeringAgent, AvoidanceAgent, AvoidanceObstacle: navmesh-based path finding and movement.

Back To Top

Structs

Structs can be defined in both the DSL and C#.

DSL Defined

The Quantum DSL also allows the definition of regular structs (just like components, memory alignment, and helper functions will be taken care of):

```
struct ResourceItem
{
    FP Value;
    FP MaxValue;
```

The fields will be ordered in alphabetical order when the struct is generated. If you need / want to have them appear in a specific order, you will have to define the structs in C# (see section below).

This would let you use the "Resources" struct as a type in all other parts of the DSL, for example using it inside a component definition:

```
component Resources
{
    ResourceItem Health;
    ResourceItem Strength;
    ResourceItem Mana;
}
```

The generated struct is partial and can be extended in C# if so desired.

Back To Top

CSharp Defined

You can define structs in C# as well; however, in this case you will have to manually define:

- The memory layout of the struct (using LayoutKind.Explicit)
- Add a const int `SIZE` to the struct containing its size.
- Implement the `Serialize` function.

```
[StructLayout(LayoutKind.Explicit)]
public struct Foo {
  public const int SIZE = 12; // the size in bytes of all members in bytes.

  [FieldOffset(0)]
  public int A;

  [FieldOffset(4)]
  public int B;

  [FieldOffset(8)]
  public int C;

  public static unsafe void Serialize(void* ptr, FrameSerializer serializer)
  {
      var foo = (Foo*)ptr;
      serializer.Stream.Serialize(&foo->A);
      serializer.Stream.Serialize(&foo->B);
      serializer.Stream.Serialize(&foo->C);
  }
}
```

When using C# defined structs in the DSL (e.g. inside components), you will have to manually import the struct definition.

N.B.: The *import* does not support constants in the size; you will have to specify the exact numerical value each time.

Back To Top

Components Vs. Structs

An important question is why and when should components be used instead of regular structs (components, in the end, are also structs).

Components contain generated meta-data that turns them into a special type with the following features:

- Can be attached directly to entities;
- Used to filter entities when traversing the game state (next chapter will dive into the filter API);

Components can accessed, used or passed as parameters as either pointers or as value types, just like any other struct.

Back To Top

Dynamic Collections

Quantum's custom allocator exposes blittable collections as part of the rollback-able game state. Collections only support support blittable types (i.e. primitive and DSL-defined types).

To manage collection, the Frame API offers 3 methods for each:

- `Frame.AllocateXXX` : To allocate space for the collection on the heap.
- `Frame.FreeXXX` : To free/deallocate the collection's memory.
- `Frame.ResolveXXX` : To access the collection by resolving the pointer it.

Note: After freeing a collection, it **HAS TO** be nullified by setting it to `default` . This is required for serialization of the game state to work properly. Omitting the nullification will result in indeterministic behavior and desynchronisation. As alternative to freeing a collection and nullifying its Ptrs manually, it possible to use the `FreeOnComponentRemoved` attribute on the field in question.

Back To Top

Important Notes

- Several components can reference the same collection instance.
- Dynamic collections are stored as references inside components and structs. They therefore **have to** to be *allocated* when initializing them, and more importantly, *freed* when they are not needed any more. If the collection is part of a component, two options are available:
  - implement the reactive callbacks `ISignalOnAdd<T>` and `ISignalOnRemove<T>` and allocate/free the collections there. (For more information on these specific signals, see the Components page in the ECS section of the Manual); or,
  - use the `[AllocateOnComponentAdded]` and `[FreeOnComponentRemoved]` attributes to let Quantum handle the allocation and deallocation when the component is added and removed respectively.

- Attempting to free a collection more than once will throw an error and puts the heap in an invalid state internally.

Back To Top

Lists

Dynamic lists can be defined in the DSL using `list<T> MyList` .

```
component Targets {
  list<EntityRef> Enemies;
}
```

The basic API methods for dealing with these Lists are:

- `Frame.AllocateList<T>()`
- `Frame.FreeList(QListPtr<T> ptr)`
- `Frame.ResolveList(QListPtr<T> ptr)`

Once resolved, a list can be iterated over or manipulated with all the expected API methods of a list such as Add, Remove, Contains, IndexOf, RemoveAt, [], etc... .

To use the list in the component of type *Targets* defined in the code snippet above, you could create the following system:

```
namespace Quantum
{
  public unsafe class HandleTargets : SystemMainThread, ISignalOnComponentAdded<Target
    {
        public override void Update(Frame f)
        {
      foreach (var (entity, component) in f.GetComponentIterator<Targets>()) {
        // To use a list, you must first resolve its pointer via the frame
        var list = f.ResolveList(component.Enemies);

        // Do stuff
      }
        }

        public void OnAdded(Frame f, EntityRef entity, Targets* component)
    {
            // allocating a new List (returns the blittable reference type - QListPtr)
        component->Enemies = f.AllocateList<EntityRef>();
    }

        public void OnRemoved(Frame f, EntityRef entity, Targets* component)
    {
            // A component HAS TO de-allocate all collection it owns from the frame da
            // receives the list QListPtr reference.
      f.FreeList(component->Enemies);

            // All dynamic collections a component points to HAVE TO be nullified in a
```

```
    }
    }
}
```

Back To Top

Dictionaries

Dictionaries can be declared in the DSL like so `dictionary<key, value> MyDictionary` .

```
component Hazard{
    dictionary<EntityRef, Int32> DamageDealt;
}
```

The basic API methods for dealing with these dictionaries are:

- `Frame.AllocateDictionary<K,V>()`
- `Frame.FreeDictionary(QDictionaryPtr<K,V> ptr)`
- `Frame.ResolveDictionary(QDictionaryPtr<K,V> ptr)`

Just like with any other dynamic collection it is mandatory to allocate it before using it, as well as de-allocate it from the frame data and nullified it once the dictionary is no longer used. See the example provided in the section about lists here above.

Back To Top

HashSet

HashSets can be declared in the DSL like so `hash_set<T> MyHashSet` .

```
component Nodes{
    hash_set<FP> ProcessedNodes;
}
```

The basic API methods for dealing with these dictionaries are:

- `Frame.AllocateHashSet(QHashSetPtr<T> ptr, int capacity = 8)`
- `Frame.FreeHashSet(QHashSetPtr<T> ptr)`
- `Frame.ResolveHashSet(QHashSetPtr<T> ptr)`

Just like with any other dynamic collection it is mandatory to allocate it before using it, as well as de-allocate it from the frame data and nullified it once the hash set is no longer used. See the example provided in the section about lists here above.

Back To Top

overlapping some mutually exclusive data types/values into a union:

```
struct DataA
{
    FPVector2 Something;
    FP Anything;
}

struct DataB
{
    FPVector3 SomethingElse;
    Int32 AnythingElse;
}

union Data
{
    DataA A;
    DataB B;
}
```

The generated type **Data** will also include a differentiator property (to tell which union-type has been populated). "Touching" any of the union sub-types will set this property to the appropriate value.

Bitsets can be used to declared fixed-size memory blocks for any desired purpose (for example fog-of-war, grid-like structures for pixel perfect game mechanics, etc.):

```
struct FOWData
{
    bitset[256] Map;
}
```

Back To Top

Input

In Quantum, the runtime input exchanged between clients is also declared in the DSL. This example defines a simple movement vector and a Fire button as input for a game:

```
input
{
    FPVector2 Movement;
    button Fire;
}
```

The input struct is polled every tick and sent to the server (when playing online).

For more information about input, such as best practices and recommended approaches to optimization, refer to this page: Input

photon

Signals

Signals are function signatures used as a decoupled inter-system communication API (a form of publisher/subscriber API). This would define a simple signal (notice the special type **entity_ref** - these will be listed at the end of this chapter):

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

This would generate the following interface (that can be implemented by any System):

```
public interface ISignalOnDamage
{
    public void OnDamage(Frame f, FP damage, EntityRef entity);
}
```

Signals are the only concept which allows the direct declaration of a pointer in Quantum's DSL, so passing data by reference can be used to modify the original data directly in their concrete implementations:

```
signal OnBeforeDamage(FP damage, Resources* resources);
```

Notice this allows the passing of a component pointer (instead of the entity reference type).

Back To Top

Events

Events are a fine-grained solution to communicate what happens inside the simulation to the rendering engine / view (they should never be used to modify/update part of the game state). Use the "event" keyword to define its name and data:

Find detailed information about events in the Game Events Manual.

Define an event using the Quantum DSL

```
event MyEvent{
    int Foo;
}
```

Trigger the event from the simulation

```
f.Events.MyEvent(2022);
```

And subscribe and consume the event in Unity

```
QuantumEvent.Subscribe(listener: this, handler: (MyEvent e) => Debug.Log($"MyEvent {e.
```

photon

Globals

It is possible to define globally accessible variables in the DSL. Globals can be declared in any .qtn file by using the `global` scope.

```
global {
    // Any type that is valid in the DSL can also be used.
    FP MyGlobalValue;
}
```

Like all things DSL-defined, global variables are part of the state and are fully compatible with the predict-rollback system.

Variables declared in the global scope are made available through the Frame API. They can be accessed (read/write) from any place that has access to the frame - see the ***Systems*** document in the ECS section.

N.B.: An alternative to global variables are the Singleton Components; for more information please refer to the *Components* page in the ECS section of the manual.

Back To Top

Special Types

Quantum has a few special types that are used to either abstract complex concepts (entity reference, player indexes, etc.), or to protect against common mistakes with unmanaged code, or both. The following special types are available to be used inside other data types (including in components, also in events, signals, etc.):

- `player_ref` : represents a runtime player index (also cast to and from Int32). When defined in a component, can be used to store which player controls the associated entity (combined with Quantum's player-index-based input).
- `entity_ref` : because each frame/tick data in quantum resides on a separate memory region/block (Quantum keeps a a few copies to support rollbacks), pointers cannot be cached in-between frames (nor in the game state neither in Unity scripts). An entity ref abstracts an entity's index and version properties (protecting the developer from accidentally accessing deprecated data over destroyed or reused entity slots with old refs).
- `asset_ref<AssetType>` : rollback-able reference to a data asset instance from the Quantum asset database (please refer to the data assets chapter).
- `list<T>` , `dictionary<K,T>` : dynamic collection references (stored in Quantum's frame heap). Only supports blittable types (primitives + DSL-defined types).
- `array<Type>[size]` : fixed sized "arrays" to represent data collections. A normal C# array would be a heap-allocated object reference (it has properties, etc.), which violates Quantum's memory requirements, so the special array type generates a pointer based simple API to keep rollback-able data collections inside the game state;

Back To Top

A Note On Assets

Assets are a special feature of Quantum that let the developer define data-driven containers (normal classes, with inheritance, polymorphic methods, etc.) that end up as immutable instances inside an indexed database. The "asset"

```
asset CharacterData; // the CharacterData class is partially defined in a normal C# fi
```

The following struct show some valid examples of the types above (sometimes referencing previously defined types):

```
struct SpecialData
{
    player_ref Player;
    entity_ref Character;
    entity_ref AnotherEntity;
    asset_ref<CharacterData> CharacterData;
    array<FP>[10] TenNumbers;
}
```

Back To Top

Available Types

When working in the DSL, you can use a variety of types. Some are pre-imported by the parsers, while others need to be manually imported.

By Default

Quantum's DSL parser has a list of pre-imported cross-platform deterministic types that can be used in the game state definition:

- Boolean / bool - internally gets wrapped in QBoolean which works identically (get/set, compare, etc...)
- Byte
- SByte
- UInt16 / Int16
- UInt32 / Int32
- UInt64 / Int64
- FP
- FPVector2
- FPVector3
- FPMatrix
- FPQuaternion
- PlayerRef / player_ref in the DSL
- EntityRef / entity_ref in the DSL
- LayerMask
- NullableFP / FP? in the DSL
- NullableFPVector2 / FPVector2? in the DSL
- NullableFPVector3 / FPVector3? in the DSL
- QString is for UTF-16 (aka Unicode in .NET)
- QStringUtf8 is always UTF-8
- Hit
- Hit3D
- Shape2D

Note on QStrings: `N` represents the total size of the string in bytes minus 2 bytes used for bookkeeping. In other words `QString<64>` will use 64 bytes for a string with a max byte length of 62 bytes, i.e. up to 31 UTF-16 characters.

Back To Top

Manual Import

If you need a type that is not listed in the previous section, you will have to import it manually when using it in QTN files.

Namespaces / Types Outside Of Quantum

To import types defined in other namespaces, you can use the following syntax:

```
import MyInterface;
or
import MyNameSpace.Utils;
```

For an enum the syntax is as follows:

```
import enum MyEnum(underlying_type);

// This syntax is identical for Quantum specific enums
import enum Shape3DType(byte);
```

Back To Top

Built-In Quantum Type And Custom Type

When importing a Quantum built-in type or a custom type, the struct size is predefined in their C# declaration. It is therefore important to add some safety measures.

```
namespace Quantum {
  [StructLayout(LayoutKind.Explicit)]
  public struct Foo {
    public const int SIZE = sizeof(Int32) * 2;
    [FieldOffset(0)]
    public Int32 A;
    [FieldOffset(sizeof(Int32))]
    public Int32 B;
  }
}
```

```
#define FOO_SIZE 8 // Define a constant value with the known size of the struct
import struct Foo(8);
```

To ensure the expected size of the struct is equal to the actual size, it is recommended to add an `Assert` as shown below in one of your systems.

```
    {
        Assert.Check(Constants.FOO_SIZE == Foo.SIZE);
    }
}
```

If the size of the built-in struct changes during an upgrade, this `Assert` will throw and allow you to update the values in the DSL.

Back To Top

Attributes

Quantum supports several attributes to present parameters in the Inspector.

The attributes are contained within the `Quantum.Inspector` namespace.

Attribute	Parameters	Description
DrawIf	`string` fieldName `long` value `CompareOperator` compare `HideType` hide	Displays the property only if the condition evaluates to true. *fieldName* = the name of the property to evaluate. *value* = the value used for comparison. *compare* = the comparison operation to be performed `Equal`, `NotEqual`, `Less`, `LessOrEqual`, `GreaterOrEqual` or `Greater`. *hide* = the field's behavior when the expression evaluates to `False`:`Hide` or `ReadOnly`. For more information on compare and hide, see below.
Header	string header	Adds a header above the property. *header* = the header text to display.
HideInInspector	-	Serializes the field and hides the following property in the Unity inspector.
Layer	-	Can only be applied to type **int**. Will call `EditorGUI.LayerField` on the field.
Optional	`string` enabledPropertyPath	Allows to turn the display of a property on/off. *enabledPropertyPath* = the path to the `bool` used to evaluate the toggle.
Space	-	Adds a space above the property

photon

Tooltip	`string` tooltip	Displays a tool tip when hovering over the property. *tooltip* = the tip to display.
ArrayLength (since 2.1) **FixedArray** (in 2.0) ONLY FOR CSharp	`int` length -------- `int` minLength `int` maxLength	Using *length* allows to define the size of a an array. ------ Using *minLength* and *maxLength* allows to define a range for the size in the Inspector. The final size can then be set in the Inspector. (*minLength* and *maxLength* are inclusive)
ExcludeFromPrototype	-	Can be applied to both a component and component fields. ------ - Field: Excludes field from a the prototype generated for the component. - Component: No prototype will be generated for this component.
PreserveInPrototype	-	Added to a type marks it as usable in prototypes and prevents prototype class from being emit. Added to a field only affects a specific field. Useful for simple `[Serializable]` structs as it avoids having to use `_Prototype` types on Unity side.
AllocateOnComponentAdded		Can be applied to dynamic collections. This will allocate memory for the collection if it has not already been allocated when the component holding the collection is added to an entity.
FreeOnComponentRemoved		Can be applied to dynamic collections and `Ptrs`. This will deallocate the associated memory and nullify the `Ptr` held in the field when the component is removed. ------ IMPORTANT: Do **NOT** use this attribute in combination with cross-referenced collections as it only nullifies the `Ptr` held in that particular field and the others will be pointing to invalid memory.

The *Attributes* can be used in both C# and qtn files unless otherwise specified; however, there are some syntactic differences.

Back To Top

Use In CSharp

In C# files, attributes can be used and concatenated like any other attribute.

```
// Multiple single attributes
[Header("Example Array")][Tooltip("min = 1\nmax = 20")] public FP[] TestArray = new FP
```

Back To Top

Use In Qtn

In qtn files, the usage of single attributes remains the same as in C#.

```
[Header("Example Array")] array<FP>[20] TestArray;
```

When combining multiple attributes, they **have** to be concatenated.

```
[Header("Example Array"), Tooltip("min = 1\nmax = 20")] array<FP>[20] TestArray;
```

Back To Top

Compiler Options

The following compiler options are currently available to be used inside Quantum's DSL files (more will be added in the future):

```
// pre defining max number of players (default is 6, absolute max is 64)
#pragma max_players 16

// numeric constants (useable inside the DSL by MY_NUMBER and useable in code by Const
#define MY_NUMBER 10

// overriding the base class name for the generated constants (default is "Constants")
#pragma constants_class_name MyFancyConstants
```

Back To Top

Custom FP Constants

You can also define custom `FP` constants inside Quantum's DSL files. ex:

```
// in a DSL file
#define Pi 3.14
```

Then, Quantum codegen will generate the corresponding constant in the `FP` struct:

```
// 3.14
FP constant = Constants.Pi;
```

It will also generate the corresponding raw value as well:

photon

To Document Top

photon

We Make Multiplayer Simple

Products

Fusion
Quantum
Realtime
Chat
Voice
PUN

Memberships

Gaming Circle
Industries Circle

Support

Gaming Circle
Industries Circle
Circle Discord
Circle Stack Overflow

Connect

Public Discord
YouTube
Facebook

Documentation

Fusion
Quantum
Realtime
Chat
Voice
PUN
Bolt
Server
VR | AR | MR

Resources

Dashboard
Samples
SDK Downloads
Cloud Status

Languages

English
日本語
한국어

photon

Contact Us