photon

Search

FUSION

QUANTUM

photon

- Input
- Materialization
- Map Baking
- Multi-Client Runner
- Navigation
- Physics
- Player
- Prediction Culling
- Profiling
- Quantum Project
- Replays
- RNG Session
- WebGL

Concepts And Patterns

Consoles

Gaming Circle

Reference

REALTIME

CHAT

VOICE

SERVER

PUN

BOLT

photon

Components

Introduction

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).

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

Component

This is a basic example definition of a component in the DSL:

```
    FP Cooldown;
    FP Power;
}
```

Labeling them as components (like above), instead of structs, will generate the appropriate code structure (marker interface, id property, etc). Once compiled, these will also be available in the Unity Editor for use with the Entity Prototype. In the editor, custom components are named *Entity Component ComponentName*.

The API to work on components is presented via the *Frame* class. You have the option of working on copies on the components, or on them components via pointers. To distinguish between the access type, the API for working on copies is accessible directly via `Frame` and the API for accessing pointers is available under `Frame.Unsafe` - as the latter modifies the memory.

The most basic functions you will require to add, get and set components are the functions of the same name.

`Add<T>` is used to add a component to an entity. Each entity can only carry one copy of a certain component. To aid you in debugging, `Add<T>` returns an *AddResult* enum.

```
public enum AddResult {
    EntityDoesNotExist     = 0, // The EntityRef passed in is invalid.
    ComponentAlreadyExists = 1, // The Entity in question already has this component a
    ComponentAdded         = 2  // The component was successfully added to the entity.
}
```

Once an entity has a component, you can retrieve it with `Get<T>`. This will return a copy of the component value. Since you are working on a copy, you will need to save the modified values on the component using `Set<T>`. Similarly to the *Add* method, it returns a *SetResult* which can be used to verify the operation's result or react to it.

```
public enum SetResult {
    EntityDoesNotExist = 0, // The EntityRef passed in is invalid.
    ComponentUpdated   = 1, // The component values were successfully updated.
    ComponentAdded     = 2  // The Entity did not have a component of this type yet, s
}
```

For example if you were to set the starting value of a health component, you would do the following:

```
private void SetHealth(Frame f, EntityRef entity, FP value){
    var health = f.Get<Health>(entity);
    health.Value = value;
    f.Set(entity, health);
}
```

This table recaps the methods already presented and the others offered to you to manipulate components and their values are:

Method	Return	Additional Info
Add<T>(EntityRef entityRef)	AddResult enum, see above.	allows an invalid entity ref.

photon

Get<T>(EntityRef entityRef)	T a copy of T with the current values.	does NOT allow an invalid entity ref. Throws an exception if the component T is not present on the entity.
Set<T>(EntityRef entityRef)	SetResult enum, see above.	allows an invalid entity ref.
Has<T>(EntityRef entityRef)	bool true = entity exists and the component is attached false = entity does not exist, or component is not attached.	allows invalid entity ref, and component to not exist.
TryGet<T>(EntityRef entityRef, out T value)	bool true = entity exists and component is attached to it. false = entity does not exist, or component not attached to it.	allows an invalid entity ref.
TryGetComponentSet(EntityRef entityRef, out ComponentSet componentSet)	bool true = entity exists and all components of the components are attached false = entity does not exist, or one or more components of the set are not attached.	allows an invalid entity ref.
Remove<T>(EntityRef entityRef)	No return value. Will remove component if the entity exists and carries the component. Otherwise does nothing.	allows an invalid entity ref.

To facilitate working on components directly and avoid the -small- overhead from using Get/Set, `Frame.Unsafe` offers unsafe versions of Get and TryGet (see table below).

Method	Return	Additional Info
GetPointer<T>(EntityRef entityRef)	T*	does NOT allow invalid entity ref. Throws an exception if the component T is not present on the entity.
TryGetPointer<T>(EntityRef entityRef out T* value)	bool true = entity exists and component is attached to it.	allows an invalid entity ref.

	component not attached to it.	

Back To Top

Singleton Component

A ***Singleton Component*** is a special type of component of which only one can exist at any given time. There can ever only be one instance of a specific T singleton component, on ***any*** entity in the entire game state - this is enforced deep in the core of the ECS data buffers. This is strictly enforced by Quantum.

A custom ***Singleton Component*** can be defined in the DSL using `singleton component` .

```
singleton component MySingleton{
    FP Foo;
}
```

Singletons inherit an interface called `IComponentSingleton` which itself inherits from `IComponent` . It can therefore do all the common things you would expect from regular components:

- It can be attached to any entity.
- It can be managed with all the regular safe & unsafe methods (e.g. Get, Set, TryGetPointer, etc...).
- It can be put on entity prototypes via the Unity Editor, or instantiated in code on an entity.

In addition to the regular component related methods, there are several special methods dedicated to singletons. Just like for regular components, the methods are separated in ***Safe*** and ***Unsafe*** based on whether they return a value type or a pointer.

Method	Return	Additional Info
API - Frame		
SetSingleton<T> (T component, EntityRef optionalAddTarget = default)	void	Sets a singleton IF the singleton does not exist. --- EntityRef (optional), specifies which entity to add it to. IF none is given, a new entity will be created to add the singleton to.
GetSingleton<T>()	T	Throws exception if singleton does not exist. No entity ref is needed, it will find that automatically.

photon

TryGetSingleton<T>(out T component)	bool true = singleton exists false = singleton does NOT exist	Does NOT throw an exception if singleton does not exist. No entity ref is needed, it will find that automatically.
GetOrAddSingleton<T>(EntityRef optionalAddTarget = default)	T	Gets a singleton and returns it. IF the singleton does not exist, it will be created like in SetSingleton. --- EntityRef (optional), specifies which entity to add it to if it has to be created. A new entity will be created to add the singleton to if no EntityRef is passed in.
GetSingletonEntityRef<T>()	EntityRef	Returns the entity which currently holds the singleton. Throws if the singleton does not exist.
TryGetSingletonEntityRef<T>(out EntityRef entityRef)	bool true = singleton exists. false = singleton does NOT exist.	Get the entity which currently holds the singleton.Does NOT throw if the single does not exist.
API - Frame.Unsafe		
Unsafe.GetPointerSingleton<T>()	T*	Gets a singleton pointer. Throws exception if it does not exist.
TryGetPointerSingleton<T>(out T* component)	bool true = singleton exists. false = singleton does NOT exist.	Gets a singleton pointer.
GetOrAddSingletonPointer<T>(EntityRef optionalAddTarget = default)	T*	Gets or Adds a singleton and returns it. IF the singleton does not exist, it will be created. --- EntityRef (optional), specifies which entity to add it to if it has to be created. A new entity will be created to add the singleton to if no EntityRef is passed in.

photon

Adding Functionality

Since components are special structs, you can extend them with custom methods by writing a *partial* struct definition in a C# file. For example, if we could extend our Action component from before as follows:

```
namespace Quantum
{
    public partial struct Action
    {
        public void UpdateCooldown(FP deltaTime){
            Cooldown -= deltaTime;
        }
    }
}
```

Back To Top

Reactive Callbacks

There are two component specific reactive callbacks:

- `ISignalOnAdd<T>` : called when a component type T is added to an entity.
- `ISignalOnRemove<T>` : called when a component type T is removed from an entity.

These are particularly useful in case you need to manipulate part of the component when it is added/removed - for instance allocate and deallocate a list in a custom component.

To receive these signals, simply implement them in a system.

Back To Top

Components Iterators

If you were to require a single component only, *ComponentIterator* (safe) and *ComponentBlockIterator* (unsafe) are best suited.

```
foreach (var pair in frame.GetComponentIterator<Transform3D>())
{
    var component = pair.Component;
    component.Position += FPVector3.Forward * frame.DeltaTime;
    frame.Set(pair.Entity, component);
}
```

Component block iterators give you the fastest possible access via pointers.

```
// This syntax returns an EntityComponentPointerPair struct
// which holds the EntityRef of the entity and the requested Component of type T.
foreach (var pair in frame.Unsafe.GetComponentBlockIterator<Transform3D>())
```

```
// Alternatively, it is possible to use the following syntax to deconstruct the struct
// and get direct access to the EntityRef and the component
foreach (var (entityRef, transform) in frame.Unsafe.GetComponentBlockIterator<Transfor
{
    transform->Position += FPVector3.Forward * frame.DeltaTime;
}
```

Back To Top

Filters

Filters are a convenient way to filter entities based on a set of components, as well as grabbing only the necessary components required by the system. Filters can be used for both Safe (Get/Set) and Unsafe (pointer) code.

Generic

To create a filter simply use the **Filter()** API provided by the frame.

```
var filtered = frame.Filter<Transform3D, PhysicsBody3D>();
```

The generic filter can contain up to 8 components. If you need to more specific by creating *without* and *any* **ComponentSet** filters.

```
var without = ComponentSet.Create<CharacterController3D>();
var any = ComponentSet.Create<NavMeshPathFinder, NavMeshSteeringAgent>();
var filtered = frame.Filter<Transform3D, PhysicsBody3D>(without, any);
```

A *ComponentSet* can hold up to 8 components. The *ComponentSet* passed as the *without* parameter will exclude all entities carrying at least one of the components specified in the set. The *any* set ensures entities have at least one or more of the specified components; if an entity has none of the components specified, it will be excluded by the filter.

Iterating through the filter is as simple as using a while loop with `filter.Next()`. This will fill in all copies of the components, and the `EntityRef` of the entity they are attached to.

```
while (filtered.Next(out var e, out var t, out var b)) {
  t.Position += FPVector3.Forward * frame.DeltaTime;
  frame.Set(e, t);
}
```

N.B.: You are iterating through and working on **copies** of the components. So you need to set the new data back on their respective entity.

The generic filter also offers the possibility to work with component pointers.

```
while (filtered.UnsafeNext(out var e, out var t, out var b)) {
  t->Position += FPVector3.Forward * frame.DeltaTime;
```

In this instance you are modifying the components' data directly.

Back To Top

FilterStruct

In addition to regular filters, you may use the *FilterStruct* approach. For this you need to first define a struct with **public** properties for each component type you would like to receive.

```
struct PlayerFilter
{
    public EntityRef Entity;
    public CharacterController3D* KCC;
    public Health* Health;
    public FP AccumulatedDamage;
}
```

Just like a *ComponentSet*, a *FilterStruct* can filter up to 8 different component pointers.

N.B.: A struct used as a *FilterStruct* is **required** to have an *EntityRef* field!

The **component type** members in a *FilterStruct* **HAVE TO BE** pointers; only those will be filled by the filter. In addition to component pointers, you can also define other variables, however, these will be ignored by the filter and are left to you to manage.

```
var players = f.Unsafe.FilterStruct<PlayerFilter>();
var playerStruct = default(PlayerFilter);

while (players.Next(&playerStruct))
{
    // Do stuff
}
```

`Frame.Unsafe.FilterStruct<T>()` has an overload utilizing the optional ComponentSets *any* and *without* to further specify the filter.

Back To Top

Note On Count

A filter does not know in advance how many entities it will touch and iterate over. This is due to the way filters work in *Sparse-Set* ECS:

1. the filter finds which among the components provided to it has the least entities associated with it (smaller set to check for intersection); and then,
2. it goes through the set and discards any entity that does not have the other queried components.

Knowing the exact number in advance would require traversing the filter once; as this is an (O(n) operation, it would not be efficient.

photon

Components Getter

Should you want to get a specific set of components from a ***known*** entity, use a filter struct in combination with the `Frame.Unsafe.ComponentGetter` . **N.B.:** This is only available in an unsafe context!

```
public unsafe class MySpecificEntitySystem : SystemMainThread

    struct MyFilter {
        public EntityRef      Entity; // Mandatory member!
        public Transform2D*   Transform2D;
        public PhysicsBody2D* Body;
    }

    public override void Update(Frame f) {
        MyFilter result = default;

        if (f.Unsafe.ComponentGetter<MyFilter>().TryGet(f, f.Global->MyEntity, &result)
            // Do Stuff
        }
    }
```

If this operation has to performed often, you can cache the look-up struct in the system as shown below (100% safe).

```
public unsafe class MySpecificEntitySystem : SystemMainThread

    struct MyFilter {
        public EntityRef      Entity; // Mandatory member!
        public Transform2D*   Transform2D;
        public PhysicsBody2D* Body;
    }

    ComponentGetter<MyFilter> _myFilterGetter;

    public override void OnInit(Frame f) {
      _myFilterGetter = f.Unsafe.ComponentGetter<MyFilter>();
    }

    public override void Update(Frame f) {
      MyFilter result = default;

      if (_myFilterGetter.TryGet(f, f.Global->MyEntity, &result)) {
        // Do Stuff
      }
    }
```

Back To Top

Previously we introduced the components and tools available in Quantum to filter them; in this section, we will present some strategies that utilize these. **N.B.:** The *best* approach will depend on your own game and its systems. We recommend taking the strategies below as a jumping off point to create a fitting one to your unique situation.

Note: All terminology used below has been created in-house to encapsulate otherwise wordy concepts.

Back To Top

Micro-component

Although many entities may be using the same component types, few entities use the same component composition. One way to further specialize their composition is by the use of **micro-components** . **Micro-components** are highly specialized components with data for a specific system or behaviour. Their uniqueness will allow you to create filters that can quickly identify the entities carrying it.

Back To Top

Flag-component

One common way to identify entities is by adding a **flag-component** to them. In ECS the concept of *flags* does not exist per-se, nor does Quantum support *entity types*; so what exactly are **flag-components** ? They are components holding little to no data and created for the exclusive purpose of identifying entities.

For instance, in a team based game you could have:

1. a "Team" component with an enum for TeamA and TeamB; or
2. a "TeamA" and "TeamB" component.

Option 1. is helpful when the main purpose is polling the data from the View, while option 2. will enable you to benefit from the filtering performance in the relevant simulation systems.

Note: Sometimes a flag-component are also referred to as tag-component because tagging and flagging entities is used interchangeably.

Back To Top

Count

The amount of a components T currently existing in the simulation can be retrieved using `Frame.ComponentCount<T>()` . When used in conjunction with flag components it enables a quick count of, for instance, a certain type of units.

Back To Top

Add / Remove

In case you only need to *temporarily* attach a flag-component or micro-component to an entity, they remain a suitable options as both the `Add` and `Remove` operations are O(1).

Back To Top

Global Lists

limited.

If you wanted to highlight all players with less than 50% health, you could hold a global list and do the following:

- Have a system at the beginning of the simulation that add/removes entity_refs to the list;
- Use that same list in all subsequent systems.

N.B.: If you only need to identified these types of conditions sporadically, we would advise to dynamically calculated it when needed rather than keeping global lists.

To Document Top

photon

Facebook
Twitter
Blog
Contact Us

한국어
简体中文
繁体中文