Deferred Operations (Deferreds)

It’s no secret and if fact by design, that you can’t execute methods that change a Subject’s structure when using the Solid semantics. That essentially means you can only change the state of the individual Traits themselves, but not add nor remove the Traits themselves. The main advantage of the Solid enchaining is that it provides for the concurrent Operating and it would certainly be great to have more flexibility while evaluating the multi-threaded processing.

So there come handy the deferred operations (or *Deferreds* for short). Like the naming implies, those are not executed immediately but are instead delayed for a later, more suitable occasion. Please note, that the Deferreds API is available for C++ only, since the whole Solid semantics is also C++-based and it’s not possible to Enchain into a Solid Chain in Blueprints.

Setting Traits

Say, we are implementing a real-time strategy game. A user can select multiple units and assign them orders (tasks). Send them marching to a destination point, for example. We could defer the Trait setting operation while Operating concurrently on the selected units, using the corresponding API method. Check out this exemplary snippet:

```
FVector Destination = GetUserClickedPoint(); // Retrieve the currently user-
clicked point on the map.
auto SolidChain = Mechanism->EnchainSolid(TFilter<FUnit, FSelected>()); //
Enchain all of the selected units.
SolidChain->OperateConcurrently([Destination](FSolidSubjectHandle Unit) //
Process the selected units in a parallel fashion.
{
    Unit.SetTraitDeferred(FMoveToPointOrder{Destination}); // Defer-assign a
Trait that's telling the unit to move to the needed point.
});
```

Removing Traits

Removing Traits can also be deferred in a quite similar fashion. Here’s an API usage example which removes a “buffed” status from the units, when it’s expiring:

```
SolidChain->OperateConcurrently([DeltaSeconds](FSolidSubjectHandle Unit,
FBuffed& Buffed) // Process the selected units in a parallel fashion.
{
    Buffed.Timeout -= DeltaSeconds; // Decrease the initially allocated
timeout using the current Tick's delta time period.
    if (Buffed.Timeout <= 0.0f)
```

```
    {
        Unit.RemoveTraitDeferred<FBuffed>(); // Defer-remove the Trait when
it's no longer viable.
    }
});
```

There is also a possibility to remove all the traits altogether. In a deferred fashion of course. This is just a matter of calling the corresponding method as in here:

```
Unit.RemoveAllTraitsDeferred();
```

Spawning Subjects

Not only traits can be added or removed in a deferred fashion but the whole Subjects can be spawned and despawned this way. So, if you have multiple units spawning a projectile when they’re charged, you could do it like so:

```
SolidChain->OperateConcurrently([Mechanism,
DeltaSeconds](FSolidSubjectHandle Unit, FCharging& Charging, FDamageDealer&
DamageDealer)
{
    Charging.Timeout -= DeltaSeconds;
    if (Charging.Timeout <= 0.0f)
    {
        Mechanism->SpawnDeferred(FProjectile{DamageDealer.Power}); // Spawn
a new subject with an FProjectile trait.
    }
});
```

Despawning Subjects

The process of destroying a Subject is quite analogous. Kill all units once their health is zero or below. Just do something like:

```
SolidChain->OperateConcurrently([](FSolidSubjectHandle Unit, FHealth&
Health)
{
    if (Health.Level <= 0.0f)
    {
        Unit.DespawnDeferred();
    }
});
```

Applying

Until when? This is quite a logical question when dealing with something that is deferred by design. And the default answer is “when the time is right”. That essentially means that the behavior is automatic by default, i.e. when the corresponding Mechanism’s Chain gets disposed and the current active state is non-Solid.

The default automatic behavior minimizes the effort and guarantees that the Deferreds get applied accordingly, but maybe you would like to have more control on when and where the application is happening. This is exactly why the concept of *Deferreds Applicators* was introduced.

Deferreds Applicators are created explicitly, by calling the UMechanism::CreateDeferredsApplicator method as in:

```
{ // Start of the explicit scope.
    auto Applicator = Mechanism->CreateDeferredsApplicator();
    Mechanism->EnchainSolid(...)->OperateConcurrently([](){
        // Your first mechanic producing deferred operations.
    });
    // The Deferreds won't be applied at this point.
    Mechanism->EnchainSolid(...)->OperateConcurrently([](){
        // Your second mechanic producing deferred operations.
    });
    // Now the Deferreds get actually applied.
} // End of the explicit scope.
```

Note that the Applicator is actually introduced within its own explicit scope (the curly brace’d region). That is in fact done on purpose since the Applicator will apply the pending changes right when it is destroyed, which is guaranteed by `Applicator` being a local (automatic) variable and the `{}` scope.

From:
http://turbanov.ru/wiki/ - **Turbopedia**

Permanent link:
http://turbanov.ru/wiki/en/toolworks/docs/apparatus/deferred?rev=1654454455

Last update: **2022/06/05 18:40**