Apparatus Architecture Overview

Apparatus is a complex tool. It’s more of a framework with its own ecosystem then some simple plugin. In order to use it effectively and consciously you have to understand how it actually works. We’re not talking about the very specifics of the implementation however but the main top-level architectural concepts. We will begin our acquaintance with a top level “singletonian” entity called *Machine*.

Machine

Machine is the main system of Apparatus. It’s a manager for all the things global and thereby is a global singleton itself. It’s actually a UObject but its lifespan is defined by its internal state, not the standard garbage collecting procedures. If the Machine has some Mechanicals defined within it or there are still some Subjects in the “scene” it will be retained and remain available. Only when it’s no longer needed and becomes meaningless it will be enqueued for a disposal.

Within the Machine in particular and inside Apparatus as a whole, two relating “worlds” exist. These are two levels of ECS data processing with their own unique features and optimizations.

The API documentation page for the `UMachine` class is of course available for your reference.

Mechanisms

Mechanisms to Apparatus are what Worlds to Unreal Engine. They provide a somewhat global state, a “scenery” context.

Mechanisms are bound tight to their Worlds. If there are some Mechanics or Subjectives with a World, a Mechanism is created automatically, but you can also create it manually like you would any other `UObject` (`NewObject<UMechanism>(...)`).

Please, see the corresponding API reference for more details.

Low-Level Traits

Let’s start with the lower layer first. The *Traits* subsystem was actually developed later in time, but it’s now at the core of the framework and provides the needed functionality for the upper layer to work properly.

ECS was once developed with performance in mind. Packing and storing the data linearly in memory, what could be simpler? While it’s actually not that easy to implement this for dynamically structured entities and requires some sophisticated bookkeeping, the whole notion is correct. The hardware layer of CPUs and RAM is really tailored towards this memory organization. Modern-day CPUs have some large cache capacities which are utilized more efficiently when used with the data pieces stored next to each other.

Unreal Engine's own memory model doesn't guarantee this level of linearity and using custom allocators is rather quirky or not viable at all. That's why we created the Traits subsystem.

Traits are primarily based on Structs. Those are managed exclusively by Apparatus and are stored in special buffers called *Chunks*, right how it's supposed to be stored - one by one, sequentially, no gaps.

Traits are in turn assembled into collections (or this wouldn't be an ECS after all). Those collections are called *Subjects*. Subjects are referenced through some specially designed *Handles*, not pointers. They are absolutely GC-independent and are disposed explicitly.

The whole design maximizes the performance of the Mechanics running on the Subjects, but it actually has some limitations comparing to the higher-level *Details*.

High-Level Details

Unlike Traits, Details are not Structs. They are high-order "Unrealean" types of things - Objects (or UObjects to be more specific). This makes them really versatile in terms of utilizing existing Unreal Engine's functionality. Not only that but they also support hierarchical filtering and even multi-detail iterating (which is quite useful when dealing with multiple details of the same type on a single Subject).

Details are always stored in their respective *Subjectives* - this is a special type of container that is directly associated with an Unreal Actor or User Widget. Subjectives are not iterated directly however but through a special caching storage called *Belt*. It's a sparse type of storage and is used as an optimization mainly, containing only references to original details. You can assign custom Belts manually on a per-Subjective basis and they will expand appropriately when needed.

Please note, that all of the Subjectives are actually Subjects internally. They all have a Subject Handle in them. This essentially means that you can add traits to them. You can interchangeably utilize the both worlds together when/if needed. It's all up to you.

Enchaining

One of the main technical goals of Apparatus is to effectively process some very large amounts of Subjects and Subjectives (our own term for ECS' Entities) under a specific filter. With that in mind a special concept of *enchaining* was developed.

Enchaining is a process of collecting all of the currently available belts and chunks matching a certain filter and embodying them in a special type of array called *Chain*. Chains are managed by the Machine singleton and you won't be creating them manually, even when using the C++ workflow.

Instead you have to use global (static) Enchain methods, passing them a filter of the desired selection. You can either enchain Belts or Chunks. While they are being enchained and the Chain actually exists the respective Belts or Chunks also remain *locked*.

Locking

During iterating the chains and their respective chunks and belts we have to guarantee a certain level of immutability for them. We don't want to process the same subject twice, for example, as it may be tossed around Chunks while being structurally modified inside the currently ongoing iterating. The Belt/Chunk locking functionality was designed specifically for that purpose.

When the Chunk (or Belt) becomes enchained, its internal locks counter is incremented, making it somewhat frozen to the evaluating Mechanic (the one that is going be working on that Chain). You can worry-free use the Chain *Cursor* and let Apparatus handle all the immutability issues for you.

Filtering

Filtering is an essential part of the proper ECS implementation. It lets you select specific Subjects and Subjectives to work apon. Using the word "select" in this context is not by chance as the term could be very familiar to a database programmer. Technically it's quite the same. You define a "WHERE" clause with a set of conditions to meet. These can be both inclusive (positive) and exclusive (negative).

Apparatus uses all sorts of different optimization schemes and caches to make the filtering process as fast as possible. You shouldn't worry too much about that.

API documentation for filters.

Iterating

Once you have your Subjects spawned and set. Belts or Chunks enchained you're ready to iterate on them to deliver the necessary logic of the game or application. This is done through a very common concept of *Iterators* and *Cursors*.

Both Belt and Chunk have their own types of Iterators, but you would rarely use them directly. Instead you'll almost always use the Chain Cursors. They are essentially Iterators with a naming chosen to eliminate some possible ambiguity. For now you should only use the default (implicit) Cursor as the threading is still a planned feature and you would rarely need to iterate a Belt (or a Chunk) with multiple different Cursors.

The API documentation for Begin and Advance methods is provided accordingly.

Afterword

This overview is of course just an overview of what Apparatus really is but we hope it helps you to grasp the main idea of the toolset. We will continue to elaborate on the specifics of the implementation in some separate articles of the Turbopedia. Stay tuned.

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

Permanent link:
http://turbanov.ru/wiki/en/toolworks/docs/apparatus/architecture?rev=1630501885

Last update: **2021/09/01 16:11**