StoryObject - work sample

This document shall explain the structure and thoughts behind this work sample.

Intention

Building features for a Story like for example playing a cutscene is easy as long as you only need to play the cutscene and maybe fade the screen but it becomes difficult once you have to do more operations on the player or operations that interact with other gameplay elements for example the time/weather, AI's (that need to be despawned or resetted) or the entire level itself (changing the world to reflect what's happened in the Story).
This is what the StoryObject system is made for.

The StoryObject system is thought of as a base for a scalable AA-size narrative. It uses a composition approach to achieve certain features instead of working with big monolithic classes.
Any StoryObject can have any number of Clients. Single functionalities like playing a sequence, fading the screen or loading resources are splitted into such clients. These are then getting executed by the StoryObject.
This allows the developer to quickly assemble actors that meet the requirements or iterate on existing actors.

The surrounding project

To demonstrate the functionality of the system, I created some structures and features outside the StoryObject system.
I'm using Unreal's Third-Person-demo as base project. A custom GameInstance class (*MyCustomGameInstance*) is serving as a linker for all important objects.

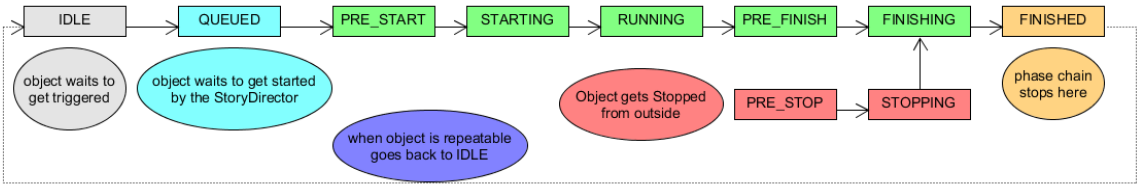
To enable screen fades a custom Blueprint HUD class (*BP_ExampleHUD*) creates a widget (*BP_WDG_FadeScreen*) on BeginPlay that can be called to either fade the screen to black or to transparent. This widget inherits from a c++ anchor class (*WDG_FadeScreen*) to enable communication with native code.

The System

As already mentioned every **StoryObject** can have **Clients**. Clients can either be **ActorComponents**, **SceneComponents** or other **Actors**.
All of them need to implement the **IStoryObjectClient** interface to be registered by the StoryObject. Client Actors need to be attached to it.
Since I wanted the developer to be able to implement clients for all above-mentioned types I chose the interface over a narrow base class (since this would also narrow clients to one of these types).

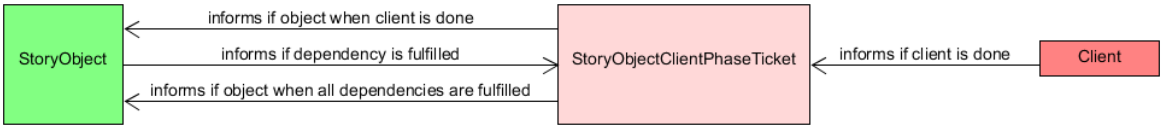
A StoryObject can be activated and stopped. This can be done via code or via **StoryObjectTrigger**.

When activated it goes through different phases (defined by the ***EStoryObjectPhase*** enum) in the following order:



In the **QUEUED** phase the StoryObject waits on the StoryDirector to be actually started.
The **StoryDirector** is a singleton-like class that is responsible for managing and orchestrating the execution, execution order and info and status tracking of StoryObjects. When the StoryDirector gives permission the object actually starts and continues with the phase order.
I chose the singleton-like pattern here just because it's easy to implement and work with.

This raises the question how these phases are actually executed.
When a phase starts the StoryObject will call the **ExecutePhase()** method from the IStoryObjectClient interface which returns a **StoryObjectPhaseTicket**.
This ticket functions as a bridge between the Client and the Object. Its main purpose is to keep track of the status of the client.
I choose to encapsulate this functionality into a separate object to keep the StoryObject code clean and easy to understand.
All fetched tickets are saved in the Object which then binds itself to the ticket. The ticket then informs the object if it's client is done and when all its dependencies are fulfilled.



A client can have dependencies for a certain phase. It will only be executed when all dependencies are fulfilled. This is useful when two tasks are dependent on each other in the same phase.
An example for this would be the object did some sublevel load/unloading (via a for this task created client) in the FINISHING phase and then fades in in the FINISHED phase. It can happen that things like foliage is plopping into the scene while fading in because unreal does not take displaying foliage as part of the loading process. In this case it would be beneficial to let the screen fading client wait one second via some kind of timer client before fading in.

A Client can also run an operation over multiple phases. This is important for heavy operations like loading and unloading sublevels or loading giant resources.

There are three client base classes (**StoryObjectComponent** for ActorComponents, **StoryObjectSceneComponent** for SceneComponents and **StoryObjectClientActor** for Actors) that can be used as base for specific clients. It is not necessary to use them since core

functionality is done via the interface but it eases working with the system significantly. They provide code which utilizes a State-Pattern like structure with which you can assign Methods to be called in specific phases. This is done with the **DECLARE_PHASE_IMPLEMENTATION** macro. When the specific client is done with its operation it needs to call **NotifyTaskIsDone()**.
Since all three base classes need have the same code and can not be settled higher in the inheritance hierarchy I put the code into the **GENERATE_CLIENT_DECLARATION_BODY** macro (for the header file) and the **GENERATE_CLIENT_IMPLEMENTATION_BODY** macro (for the implementation file).

Potential of this system

In the Example Project I created a simple proof-of-concept TestObject that can play a sequence and perform screen fades.
This limited application shows that the core system works but would of course be too simple for real application use.

The system also allows it to create StoryObjects that have ‘spatial’ characteristics. When **IsSpatialObject** is ticked the object will be stopped if the is not triggering it anymore.
An example for this would be an Object that plays a certain effect when triggered that is used to guide the player and stops playing when the player is leaving the trigger (the area).

As long as the implementation of the clients allows it there can be StoryObjects of any complexity without the need to duplicate code or reimplement certain structures. The composition of these StoryObject classes can easily be done on Editor/Blueprint level and therefore allow faster iteration due to no programmers needing to be involved unless the clients need extended functionality.