What Is Awperative?

Awperative is a passion fueled **Game Development Platform** in **C#**; created by me, **Avery Norris**. With the goal of low level, customizable and event driven game design. It is completely **free** and **open source** and modifying it is actually encouraged!

Awperative could also be considered a Game Library but the definition is rather loose. I prefer to narcissistically use the term **Game Framework**. Due to the alternate nature of Awperative compared to most Game Development Tools.

Principles

Instead of offering specific "Technology" or "Tools" Awperative's goal is to provide a platform and system rather than the content itself. My initial efforts have gone into purging as much **"Development Bias"** from Awperative as possible.

Unity has a very good example of this bias. If you have worked in Unity **2D** you are almost certainly familiar with the fact that all *Vectors* for 2D Game Objects, are actually **3D** despite the 2D environment. This is becuase Unity was originally built as a 3D Game Engine. After initial success they decided to make a 2D version. And when they did, rather than completely modify Unity internals, They sanely chose to bandaid this original 3D environment by pretending the Z axis did not exist.

In other words, what I call "Developmental Bias" is when Development Tools implement ideas with specific assumptions or panderings to a certain type of Game; It is not detrimental to development. However it can certainly muddy the waters.

Modern Game engines have hundreds if not thousands of small bias' plaguing their **Renderers**, **Physics**, **Lighting**, etc. While Awperative does not claim to be a 100% bias free system. I believe it is quite close, and certainly more effective than any Game Engine by many times.

Design

So then how does one purge bias from a **System**? There's two schools of thought.

1. Make a system that works in any circumstance imaginable.
2. Allow people to modify the framework such that they can make it do anything.

If it's not immediately clear, one of these goals is very unrealistic. To explain why let's imagine we are building a different Game Engine, and we would like to provide developers with a Player Movement system, the system should work with any player provided transformation struct (Which should generally hold things such as position, rotation etc.).

Idea 1

Following our previous rule of bias elimination, we should make it work in as many environments as possible; For the sake of this example we will pretend the only factor that can change how transform works in the world. For instance 2d or 3d, Quaternions or Euler Angles, etc.

Making a player controller that could work from 0-∞ dimensions is nothing short of computer magic. And even if we did find a way to make that, we are still restricted to one type. Because our goal is to be able to use structs provided by developers, Lest we modify the source code, our only option is to implement an interface or abstract class so that we can actually use the transformation struct.

Even if we get past all of these problems, making a system that works for every design philosophy is still impossible. You can only build systems with so much foresight. We can't make a system that uses Quaternions AND Euler Angles without manually coding them both in. Our availablility and customization relies solely on what our foresight predicted.

We have no way of knowing if someone will ever make another Quaternion type of Angle. And if someone happens to, and wants to use it in the game, our library cannot magically support it unfortunately.

Idea 2

That leaves us with one other option, which is the idea of modification. In Awperatives case I like to call this "Modulation". Since Awperative is already nearly unbiased you *rarely* need to remove specific old features.

Think of it as modding a video game. The end goal of Awperative is to provide a very special kind of asset store; where even Transformation profiles are something you get or make outside of Awperative.

Imagine creating a new project and being prompted with a list of modules you can import. Say "Johns Transformation Matrix", and then "Jasons Collision System", which uses Johns Matrix as a dependency. I'd love to eventually see a module based philosophy like this become mainstream, even if not within Awperative.

Purpose

So if most of the Game-Related features you would expect from a Game Development Tool is something meant to be built on it's own, what is Awperative ***actually***?

Well I'm glad you asked. The purpose of this documentation is to be discussing what I have been calling "Awperative", but I also like the call it the Awperative **"Kernel"**. The Kernel's unique strength comes from two core design principles

- **Generalization**
- **Reduction and Modulation**

Generalization

Starting with Generalization; I've worked very hard to reduce redundancy and specification in some scenarios. A good example of Generalizaiton is **Doors** and **Windows**. To us humans they are different things, but in General, they are both just passages that open.

Awperative takes a doors and windows approach with multiple core systems; Most times we can generalize this scenario by having both doors and windows inherit an interface called something along the lines of "ThingsThatOpen". The same can be done with abstract classes or sometimes even better, we can compress both systems into one class, for instance: bowls and cups can be combined into one joined **receptical** class.

Reduction And Modulation

While filled with much lamer examples, I would say reduction is much more important to what makes Awperative special. Reduction is the affirmentioned process of simplifying what Awperative handles, when you purge Development Bias, that is Reduction, since it is reducing what is in the way.

Modulation is almost the opposite of reduction, the process of turning some specific element or system into a deployable feature. For instance, if you make a sick Json Loading System, a good Game Framework should make it easy to share it for other projects.

A good module should always work unless nefarious modules do not play nice with one another.

Namespace Awperative

Classes

Awperative

Initiating class of Awperative. Call Start() to start the kernel.

Base

Base class of Awperative. Carries events from MonoGame into scenes and hooks.

Body

BodyComponent

BodyCreateEvent

BodyDestroyEvent

Component

The lowest level scripting class in Awperative. Components are scene level and provide access to all scene level methods, can be applied to any docker and inherited Sadly component does not have excessive access to specific types. Anything that inherits Component is built to work in any DockerEntity, which leads to generic Assumptions. If you want to make a body specific or scene specific component both classes are available.

ComponentCreateEvent

ComponentDestroyEvent

Debug

DockerEntity

Base class for all Awperative entities, manages components as a requirement because that is the job of all entities.

Scene

SceneComponent

SceneCreateEvent

SceneDestroyEvent

Transform

TransformModifyEvent

Interfaces

AwperativeHook

Awperative hooks are the source of entry for scripts using Awperative. Create a hook and send into Start() to be recognized by the engine.