

Reference Documentation

Last Updated 2022-02-10

Contents

[Overview 6](#_Toc95337824)

[Introduction 6](#_Toc95337825)

[Project Goals 6](#_Toc95337826)

[Project Philosophy 6](#_Toc95337827)

[Features 6](#_Toc95337828)

[Version History 7](#_Toc95337829)

[Project Organization 10](#_Toc95337830)

[Games that use CommonCore 11](#_Toc95337831)

[CommonCore Anatomy 13](#_Toc95337832)

[CommonCore Project Structure 13](#_Toc95337833)

[CommonCore Startup Sequence 19](#_Toc95337834)

[CommonCore Scene Flow 21](#_Toc95337835)

[CommonCore Scene 23](#_Toc95337836)

[Tags, Layers, and Sorting 25](#_Toc95337837)

[Modules and Assemblies 27](#_Toc95337838)

[State Objects 29](#_Toc95337839)

[More on save/load 30](#_Toc95337840)

[CommonCore Core 32](#_Toc95337841)

[Core Params 32](#_Toc95337842)

[Core Utilities 35](#_Toc95337843)

[Resource Management 37](#_Toc95337844)

[Addons 41](#_Toc95337845)

[CommonCore Modules 43](#_Toc95337846)

[Async 43](#_Toc95337847)

[Audio 43](#_Toc95337848)

[Config 44](#_Toc95337849)

[Console 44](#_Toc95337850)

[DebugLog 44](#_Toc95337851)

[GameData 44](#_Toc95337852)

[Input 44](#_Toc95337853)

[LockPause 45](#_Toc95337854)

[QDMS 45](#_Toc95337855)

[Scripting 46](#_Toc95337856)

[State 46](#_Toc95337857)

[StringSub 47](#_Toc95337858)

[UI 47](#_Toc95337859)

[Util 47](#_Toc95337860)

[Video 48](#_Toc95337861)

[Addon Support 48](#_Toc95337862)

[Basic Console 48](#_Toc95337863)

[Basic Humanoid 48](#_Toc95337864)

[Campaign 48](#_Toc95337865)

[Explicit KBM Input 48](#_Toc95337866)

[Test Module 49](#_Toc95337867)

[Unity Post Processing V2 Integration 49](#_Toc95337868)

[World 49](#_Toc95337869)

[Achievements 49](#_Toc95337870)

[Bigscreen 49](#_Toc95337871)

[CD Audio 50](#_Toc95337872)

[Immersive Monologue 50](#_Toc95337873)

[SickDev Console Integration 50](#_Toc95337874)

[Speed Hacks 50](#_Toc95337875)

[Unsplash 50](#_Toc95337876)

[WindowTitle 50](#_Toc95337877)

[XSMP 50](#_Toc95337878)

[CommonCore Audio Module 51](#_Toc95337879)

[CommonCore Input Module 52](#_Toc95337880)

[Available input mappers 52](#_Toc95337881)

[Configuring available controls 52](#_Toc95337882)

[The MappedInput API 52](#_Toc95337883)

[Creating an input mapper 53](#_Toc95337884)

[CommonCore Campaign Module 54](#_Toc95337885)

[CommonCore World Module 56](#_Toc95337886)

[WorldUtils 56](#_Toc95337887)

[WorldConsoleCommands 56](#_Toc95337888)

[CommonCore Entities 56](#_Toc95337889)

[Entity Prefab Prefixes 56](#_Toc95337890)

[Placeholders 57](#_Toc95337891)

[CommonCore Effects 57](#_Toc95337892)

[Bullets, Hitboxes, etc 57](#_Toc95337893)

[Players and Actors 58](#_Toc95337894)

[Player Spawn Selection 58](#_Toc95337895)

[Saving and loading 59](#_Toc95337896)

[Action Specials 60](#_Toc95337897)

[PlayerFlags 61](#_Toc95337898)

[Cartographer and maps 61](#_Toc95337899)

[Light reporting 62](#_Toc95337900)

[CommonCore RPGGame 63](#_Toc95337901)

[Dialogue 63](#_Toc95337902)

[Inventory 63](#_Toc95337903)

[Scripts 63](#_Toc95337904)

[Weapons 64](#_Toc95337905)

[Player Object 64](#_Toc95337906)

[Player State 64](#_Toc95337907)

[Actors 64](#_Toc95337908)

[Delayed Events 65](#_Toc95337909)

[How to do things in CommonCore 66](#_Toc95337910)

[Setting up a Scene 66](#_Toc95337911)

[Applying Camera configuration 66](#_Toc95337912)

[Choosing a SceneController 66](#_Toc95337913)

[Changing Scenes 66](#_Toc95337914)

[Using Scripts 67](#_Toc95337915)

[Using Scripting Hooks 67](#_Toc95337916)

[Using the Console 68](#_Toc95337917)

[Writing to the console 68](#_Toc95337918)

[Creating console commands 68](#_Toc95337919)

[Sending and receiving messages 68](#_Toc95337920)

[Using QdmsMessageInterface 68](#_Toc95337921)

[Using string substitution 68](#_Toc95337922)

[Implementing a StringSubber 69](#_Toc95337923)

[Playing Audio and Music 69](#_Toc95337924)

[Fading in and out 69](#_Toc95337925)

[Creating a Module 69](#_Toc95337926)

[Playing Videos 70](#_Toc95337927)

[Building UIs 70](#_Toc95337928)

[Using Themes 70](#_Toc95337929)

[Creating Migrations 70](#_Toc95337930)

[Miscellanea 71](#_Toc95337931)

[Useful things to know 71](#_Toc95337932)

[Useful Console Commands 72](#_Toc95337933)

[Platform Compatibility 73](#_Toc95337934)

[Data Paths 75](#_Toc95337935)

[Dialogue Format 77](#_Toc95337936)

[The Intent behind Intents 78](#_Toc95337937)

[Built-in Script Hooks 79](#_Toc95337938)

[Named Hooks 80](#_Toc95337939)

[Default String Substitution Lists 81](#_Toc95337940)

# Overview

## Introduction

CommonCore (formerly ARES) is a complete Role-Playing Game library for Unity... or will be someday. The intent is to provide a base that allows easy development of everything from quick adventures to epic open-world sagas, as well as being flexible enough to be adapted for mechanically similar genres such as open-world sandbox, shooters, and more.

CommonCore handles or will handle standard RPG mechanics, game state with saving and loading, the player object, NPCs, dialogue, input, UI, configuration and more. It is (or will be) a complete solution that can be loaded as a template followed immediately by building the actual game. For cases where you don't need all the functionality, it is divided into a separate Core, some modules, and RPGGame so you don't have to use it all (more modularity is planned).

//TODO rewrite this (someday)

## Project Goals

At one point, CommonCore was supposed to be a comprehensive RPG library for Unity and would have made it to the Asset Store, but I realized after a while that there was no way I had the time to pull off something like that.

So I downgraded it to an internal project, in support of Ascension III and my other ~~janky shit~~ game projects. It is publicly available and freely licensed, but I will not provide support or good documentation.

Fundamentally, I’m trying to get this to a point where I don’t have to worry about the nuts and bolts and I can just *build a game*. This is an ill-defined goal, and probably an ever-changing one.

I doubt CommonCore will ever be “finished” because there are always new features I want to add and new things I want to try.

## Project Philosophy

* Ease of use is absolutely critical. Assume whoever is making the game is incredibly lazy and incompetent
* Performance should be good enough, but doesn’t need to be better than that. We’re trying for 60FPS on a midrange gaming PC. This isn’t a good library for mobile games.
* Alert the user to errors, but handle and recover where possible. Don’t fail catastrophically but don’t fail silently either.
* Have extensibility where it is practical to do so. Ideally, you shouldn’t have to modify the core at all, ever.

## Features

//TODO at some point

**Current Features**

**Planned Features (near-term)**

**Planned Features (long-term)**

## Version History

**1.0 *Arroyo\****

* Public Test Release 1 (2018-07-26)
  + Inventory, save/load, NPCs, player working. Quest log and leveling systems partially implemented.
* Public Test Release 2 (2018-12-13)
  + Many changes/improvements, new bigger “test island” map with demo quest.
  + Last version of Arroyo series

\* A cut-down version was briefly developed under the codename *Aradesh*. The idea of a separate “basic” version was abandoned in favour of breaking up CommonCore into Core, modules, and RPGGame within the same repository.

**2.x *Balmora***

* Public Test Release 3 (2019-02-06)
  + Core separated from RPGGame.
* 2.0.0 Preview 2 (2019-05-26)
  + Ascension III demo mostly separated from RPG library, BasicConsole and Basic Humanoid added, project reorganized somewhat. First release with new naming convention.
  + Last version before repo was separated
* 2.0.0 Preview 3 (2019-08-27)
  + Combat and NPC changes, as well as core fixes and changes. New character controller?
  + Now separated into public CommonCore and private Californium (Ascension III) repositories.
* 2.0.0 Preview 4 (2019-09-23)
  + Almost all Ascension III assets removed from public repository, repository cleaned up and migrated off Git LFS.
* 2.0.0 Preview 5 (2019-12-25)
  + Async and Scripting moved into core, improved config options, core and utility cleanup/changes, skill checks in dialogue and new hit puffs with pufftype.
* 2.0.0 Preview 6 (2020-01-28)
  + Overhauled weapons with better viewmodels and RPG integration, less buggy projectiles, difficulty and gameplay options, dialogue system revised UI and feature parity with Katana.
* 2.0.0 Preview 7 (2020-02-09)
  + Upgraded character controller (fall damage and sounds), changes to gameplay options, actors moved to their own layer.
* 2.0.0 Preview 8 (2020-02-26)
  + Bugfixes and convenience features, autosave and quicksave, skippable wait and screenfade functions, campaign var type flexibility.
* 2.0.0 Preview 9 (2020-03-28)
  + Facing sprites (2.5D shooter style), experimental IL2CPP and UWP support, remappable keyboard/mouse input, resolution setting, config windows reworked.
* 2.0.0 Preview 10 (2020-05-07)
  + Quality-of-life changes, editor-only/player-only config, convenience methods, entity tag and dialogue system changes, fixes for longstanding bugs (including the jump-slide).
* 2.0.0 Preview 11 (2020-06-11)
  + New resource manager (WIP), longstanding Unix path bug breaking saves fixed, named script hooks added, sliding and swinging doors and keys added.
* 2.0.0 Preview 12 (2020-07-23)
  + Async startup, campaign state move into module, world time moved into core, targeting and damage handling in World cleaned up
* 2.0.0 Preview 12a (2020-07-24)
  + Minor fixes
* 2.0.0 Preview 14
  + UI theming/styling support, Halo-style shields
* 2.0.0 Preview 15
  + More actor and inventory APIs, PlayerController fixes, minor save rework, utilities
* 2.0.0 Preview 16
  + Basic addon support including loading assemblies and resources. Minor fixes, mostly to actors.
* 2.0.0 Release Candidate 1
  + Hit flags and other changes to damage handling. Tests and bug fixes.
* 2.0.1
  + Bugfixes, slightly improved debug logging

**3.x *Citadel***

* 3.0.0 Preview 1
  + Dialogue system ImageFrame, scripting and visual options, improved damage and pain handling in Actors, UIDs for inventory, mutable faction state, visual fixes for ultrawide
* 3.0.0 Preview 2
  + Updated to Unity 2019, new loading indicator (finally), theming and resource import bugfixes, improved aid item handling, dialogue keyboard support, saves version info
* 3.0.0 Preview 3
  + Additions to dialogue, many bug fixes, game-specific input maps, some fixes for ultrawide, enabled incremental GC by default, infrastructure for pending changes in options panel, timescale, WIP dialogue trace, lighting for sprite weapons
* 3.0.0 Preview 4
  + Args field in ScriptExecutionContext, more references in dialogue object, Terminated flag in CCBase, items now equipped by ID, revamped player spawn logic, major fixes to messaging and resource manager, WIP fake physics for actors
* 3.0.0 Preview 5
  + Item scripts, startup metadata in PersistState, fixes/tweaks to dialogue, factions, actors
* 3.0.0 Preview 6
  + additions to weapons, theming for scrollbars, follower/enemy difficulty handling on actors, bright flags for facing and billboard sprites, improvements to projectiles, convenience VideoModule, GameData into core
* 3.0.0 Preview 7
  + Additions and fixes to weapons, inventory, and actors, RpgInventoryModified message, Bright flags for sprite weapons, fatal error handling, resource manifest (experimental), actor refactor moving more functionality into now-abstract ActorAttackComponentBase
* 3.0.0 Preview 8
  + Many bugfixes and minor tweaks, CharacterModel messages now specify *which* CharacterModel, generalized slideshow, quality of life additions.
* 3.0.0
  + Minor bugfixes
* 3.1.0
  + Minor bugfixes, a few small experimental features

**4.x *Downwarren***

* 4.0.0 Preview 1
  + Minor bugfixes, updated libraries, convenience features mostly in async and audio
* 4.0.0 Preview 2
  + Migrations, save metadata, get mapping API, generic UI modals, extensible microscripts/conditionals, brightness control, plus minor bugfixes and tweaks

## Project Organization

There are two repositories that constantly participate in the CommonCore development process. One is the [public commoncore repository](https://github.com/XCVG/commoncore/) you probably found this on. The other is the private repository where Ascension III lives.

~~Most of the development work is done on the Ascension III repository, and is periodically pushed up to the public repository.~~ With the abandonment of Ascension III, most work is done on the CommonCore repository itself. Sometimes changes are pushed up from other game projects based on CommonCore as well.

None of this is done with any sort of automation. The repositories are technically unrelated and all merging is done manually.

At one point, CommonCore Core and the RPGGame part were considered separate projects, but this hasn’t been the case in a long time.

There is a [separate repository for extra modules](https://github.com/XCVG/commoncore-modules), and at some point I will probably put up an “experimental” repository for ~~broken~~ crazier stuff as well.

## Games that use CommonCore

*strikethrough indicates abandoned projects*

**~~KILLERS~~** – XCVG Systems/EiNR – 2018-2019 – Aradesh/Arroyo early 1.x (?) Core

[**STARFURY**](https://www.xcvgsystems.com/starfury/) – XCVG Systems – February 2019 – Public Test Release 3 (2.0.0 Preview 1) Core

[**Beach Defend 2000**](https://www.xcvgsystems.com/beach-defend-2000/) - XCVG Systems – August 2019 - 2.0.0 Preview 3 Core

**“whistler” (unreleased)** – XCVG Systems – February 2020 – 2.0.0 Preview 7 Full

[**bang ouch**](https://www.xcvgsystems.com/static/bcgj2020/) – XCVG Systems – February 2020 – 2.0.0 Preview 7 Full

**~~“nuremberg"~~** – XCVG Systems – March 2020 – 2.0.0 Preview 8 Full (or 1ca4404?)

**~~“kitee”~~** – XCVG Systems – 2020 TBD – 2.0.0 Preview TBD

[**Toilet Paper Panic**](https://xcvg.itch.io/toilet-paper-panic) – XCVG Systems – April 2020 – 18f00d1 Full

[**Heavy Metal Slug**](https://xcvg.itch.io/hmslug) – XCVG Systems – April 2020 - 5ad60c4 Core

[**Shattered – Why Not Me**](https://xcvg.itch.io/shattered) – XCVG Systems – April 2020 - 3cf1b94 Full

[**RiftBreak (partially)**](https://xcvg.itch.io/riftbreak) – XCVG Systems – June 2020 – 2.0.0 Preview 11

[**In The Middle Of The Night**](https://xcvg.itch.io/in-the-middle-of-the-night) – XCVG Systems – Aug-Sept 2020 – 2.0.0 Preview 14

[**Bang Ouch Infinite**](https://xcvg.itch.io/bang-ouch-infinite) – XCVG Systems – Holiday 2020 – 2.0.0

**~~“Reality”~~** – XCVG Systems – 2021 – 3.x

[**A Dream Of Valhalla**](https://xcvg.itch.io/valhalla) – XCVG Systems – March 2021 – 3.0.0 Preview

[**~~Ascension III~~**](https://www.xcvgsystems.com/ascension-iii-2/) – XCVG Systems – TBD – TBD

[**Safety and Security at the Liberty Macvonden Building**](https://xcvg.itch.io/itc1141a) – XCVG Systems – June 2021 – 3.0.0 Preview

**“My Confession” (unreleased)** – XCVG Systems – Summer 2021 – 3.0.0 Preview

[**At The Break Of Dawn**](https://xcvg.itch.io/thc) – XCVG Systems – September 2021 – 3.0.0 Preview

[**Glowstick Deliverance Infinite**](https://www.xcvgsystems.com/static/bcgj2021/) – XCVG Systems – September 2021 – 3.0.0 Preview

[**The Crystal Tower**](https://xcvg.itch.io/takagi) – XCVG Systems – Holiday 2021 – 3.0.0

**~~“Moonbathers”~~** – XCVG Systems – Winter 2022 – 4.0.0 Preview

**Shattered 2: The Promise** – XCVG Systems – 2022 – 4.x

**“Aurora”** – XCVG Systems – 2022? – TBD

**“Skylake”** – XCVG Systems – 2022? – TBD

**“Apocalypse & Chill”** – XCVG Systems – 2022 – TBD

**“Vomit Comet”** – XCVG Systems – 2023 – TBD

# CommonCore Anatomy

//TODO explain what is a module

## CommonCore Project Structure

At the top level, a CommonCore project is organized like this:



For the most part, this is convention rather than strict requirement.

* **CommonCore**
  + Contains CommonCore core files
* **CommonCoreGame**
  + Contains RPGGame or other game module files
* **CommonCoreModules**
  + Contains modules, each in their own folder
* **Editor**
  + Contains editor scripts that are not part of core or modules
* **Migrations**
  + Contains migrations, including those for core and modules
* **Objects**
  + Contains models, materials, etc for specific objects
* **Plugins**
  + Unity magic folder. Contains libraries and third-party asset plugins
* **ProCore**
  + Config for ProBuilder.
* **Resources**
  + Unity magic folder. Contains runtime loaded resources
* **Scenes**
  + Your game’s scenes and scene-specific resources
* **Shared**
  + Graphics, audio, textures, etc shared between scenes
* **StreamingAssets**
  + Unity magic folder. Contains streaming assets.
* **ThirdParty**
  + Third-party assets
* **~~UI~~**
  + UI-related graphics, audio, and scripts. Removed in latest versions, use Shared/Graphics and Resources/UI instead

**CommonCore**

Contains CommonCore Core files

* **Core**
  + Code files compiled into CommonCore.Core.dll
* **CoreShared**
  + Code files without asmdef (compiled into Assembly-CSharp.dll)
* **Editor**
  + Editor script files
* **Fonts**
  + Core UI fonts
* **Graphics**
  + Core UI graphics
* **Materials**
  + Core/debug materials
* **Resources**
  + Core resources. Mostly UI. Note the extra Core folder, see the section on resource management on why this is.
* **Scenes**
  + Core/infrastructure scenes including scene-specific scripts and assets.

**CommonCoreGame**

Contains files for the game module, RPGGame unless you’ve made your own

**CommonCoreModules**

Most modules can simply be copied into the CommonCoreModules, but some need extra steps like modifying CoreParams or adding scenes to the build.

**Objects**

I like to divide the Objects folder into a few subfolders:

* OpenSource
  + FOSS assets that can be safely and freely shared
* ThirdParty
  + Assets from the Asset Store etc
* ThirdPartyCustom
  + Customized textures, prefabs, etc used with assets in ThirdParty
* TestObjects
  + Assets used as placeholders and/or for testing

The repository is set up with some of these folders but you don’t have to use them if you don’t want them. The main reason I set up things this way is so I could .gitignore out assets that weren’t licensed appropriately for inclusion in a public source tree.

**Plugins**

At the time of writing, the following freely-licensed libraries are bundled with CommonCore:

* Json.NET (Newtonsoft.Json) 12.0.3
* Json.NET for Unity AOT 12.0.3
* System.Collections.Immutable 4.6.26515
* WaveLoader 1.0.0.0

Depending on the platform, either vanilla Json.NET or Json.NET for Unity AOT will be used.

~~I thought I’d used the version of System.Collections.Immutable from .NET Core or the one from NuGet, but this one seems to be an old .NET Framework version (?!). It may have been recycled from .NET Framework to .NET Core, as the license information indicates it is from or for .NET Core.~~ It’s [System.Collections.Immutable 1.5.0](https://www.nuget.org/packages/System.Collections.Immutable/1.5.0), the last version that doesn’t require System.Memory.

[WaveLoader](https://github.com/XCVG/WaveLoader) is my own creation.

Some assets also dump their contents in this folder, sometimes correctly and sometimes not (I’m looking at you, DevConsole 2).

**Resources**

The Resources folder contains run-time resources for the game.

* Data
  + Dialogue
  + Items
  + Monologue
  + Quests
  + RPGDefs
  + Strings
* Dialogue
  + bg
  + char
* DynamicMusic
* DynamicSound
* DynamicTexture
* Effects
* Entities
* UI
  + Icons
  + Maps
  + Portraits
  + Sound
* User
* Voice
* WeaponViewModels

Wow, that’s a lot to unpack! Fortunately, most things are named sanely (though there are exceptions)

The **Data** folder contains game data, mostly as JSON files. Dialogue contains dialogue files, Items contains item definition files, Monologue contains monologue (NPC ambient dialogue, more or less) files, Quests contains quest definition files. Strings contains string substitution lists. RPGDefs contains a few miscellaneous definitions including faction definitions, initial container state, and initial player state (rpg\_quests.json is a relic of a bygone era).

The **Dialogue** folder contains graphics for the dialogue system, organized as they were in [katana](https://github.com/XCVG/katana).

The **DynamicMusic**, **DynamicSound**, **UI/Sound**, and **Voice** folders contain audio files that can be accessed by the Audio system.

**DynamicTexture** is used for dynamically usable textures like DynamicMusic and DynamicSound, though this is much less widely supported at the time of writing (basically just TextureAssignScript).

**Entities** contains prefabs of CommonCore Entities and **Effects** contains prefabs of Effects. These are described in greater detail in the World section of this document.

The **UI** folder contains UI prefabs and dynamically loadable resources. Icons is meant for inventory item icons and such; things that will be looked up at runtime. Maps is for the world map system (see the section on the Cartographer), Portraits are shown on the status screen.

The **User** folder is for user/game-specific files like startmessage/MOTD. That’s literally the only thing it’s used for currently.

Finally, the **WeaponViewModels** folder contains viewmodels for weapons, described in more detail in the RPGGame section of this document.

The virtual hierarchy accessed via CoreUtils.LoadResource\* and ResourceManager is a bit more complex and that is explained later in this document.

Yes, I know Unity doesn’t recommend the Resources folder anymore. I’m moving toward enabling support for AssetBundles for some purposes, but I’ll give up my Resources folder when they come out with a replacement that actually works and isn’t a gigantic pain in the ass to use.

Do not create folders called *Addons*, *Modules*, or *Streaming* here as those are treated specially by the resource manager.

**Scenes**

By default, the Scenes folder is broken up into three subfolders:

* Meta
  + Menus and other “infrastructure” type scenes
* Other
  + Tests and miscellaneous scenes that aren’t really part of the game
* World
  + Scenes that are part of the game world

The actual code doesn’t care about these folders, it’s purely for your own organization.

At some point I will probably add a “Cutscene” folder or similar. Right now it’s up to you if you want to consider these Meta, Other, or add another folder. I think I added another folder for both Whistler and Lucidity.

I’m also a big fan of using a folder beside the scene, named the same as the scene, to contain scene-specific assets and scripts. If you generate lighting a suitable folder is created automatically.

**Shared**

The Shared folder is intended for assets and scripts that are shared across scenes and/or objects and do not need to be loaded dynamically.

* Fonts
* Graphics
* Materials
* Models
* Scripts
* Sounds
* Textures

These are all pretty self explanatory. I’ve been throwing terrain layers in Materials but they should probably have their own folder. I’ve also been using Sounds very little, because you can’t use these sounds from the Audio system directly.

I used to just throw this stuff in the root but a subfolder seems nicer.

**StreamingAssets**

StreamingAssets is a Unity special folder. Things put in StreamingAssets will be copied to a folder in the game’s data at build time. They are not loaded by default.

CommonCore will load certain things from the StreamingAssets folder. The elocal and expand folders are mounted just like addons (we’ll get to those shortly) except that elocal is mounted at Streaming/ instead of Addons/<package name>/

Addons can also be placed in StreamingAssets/Addons/

Note that addon support must be enabled in CoreParams and the game must be built for a platform that supports addons to load resources from elocal and expand. To load addons from the Addons folder they must be added to the load order in ConfigState (or the config file), addons enabled in config and keep in mind addons in other locations may override them.

**UI**

The UI folder contained assets and scripts for this game’s UI. They have all been moved to other locations, though legacy projects may have things in the UI folder.

* Fonts
  + Moved to /Shared/Fonts
* Graphics
  + Moved to /Shared/Graphics
* Scripts
  + Moved to /Shared/Scripts
* Sounds
  + Moved to /Shared/Sounds or /Resources/UI/Sound

**Other Remarks**

The ThirdParty folder was originally created to .gitignore out certain assets from public repositories, mostly for licensing reasons

If you are using Standard Assets, they should be in the Standard Assets folder. I *think* this is done by default but I’m not sure. Not that it matters all that much anyway, I’m just pedantic

You don’t have to use ProBuilder and I’m pretty sure the ProCore folder was included in the repository by accident.

## CommonCore Startup Sequence

CommonCore is initialized immediately when the application is started, after Unity initialization but before the first scene is loaded. This is done in CCBase.OnApplicationStart, which is decorated with a RuntimeInitializeOnLoadMethodAttribute. There are a few other hooks mostly for coordinating asynchronous startup.

CCBase performs the following, in order:

* Set Initializing to *true*
* Initialize CoreParams
  + Set initial values and load overrides
* Scan for all “relevant” Types and make BaseGameTypes available
  + “Relevant” types are defined as ones that are from CommonCore, modules, or game/user code, as opposed to system or Unity code. These are the ones scanned via reflection by Core and various modules.
* Initialize MigrationsManager
  + Loads Migrations into MigrationsManager, making it ready to migrate state objects in later load steps
* Set up a MonoBehaviour hook
  + This is a small MonoBehaviour on a DontDestroyOnLoad’ed GameObject that propagates certain event functions to Core.
* Hook the OnApplicationQuit event, sceneLoaded event, and sceneUnloaded event
  + These are propagated to modules via event function calls
* Create save, data, and debug folders if they don’t exist
* Initialize Explicit modules
  + Only Explicit modules specified to be loaded in CoreParams will be initialized!
  + All modules derive from CCModule and are discovered via reflection.
* Initialize the ResourceManager
  + This will only happen on platforms that support it, otherwise the legacy resource manager will be used instead
* Print the system data string to console
  + Not important to the startup process but useful for debugging
* Initialize Early modules
* Initialize undecorated modules
* Initialize Late modules
* Setup ModulesByType
  + This is for fast lookup in GetModule<T>, but as a backup that function also searches the list of modules if it is called (for example in another module’s init) before all modules are initialized
* Execute OnAllModulesLoaded on all loaded modules
  + OnAllModulesLoaded script hook is also run here and it is the first script hook guaranteed to be run
* Load resources from StreamingAssets, if enabled
  + OnAddonLoaded will be executed on all loaded modules after loading resources
* Load addons, if enabled
  + OnAddonLoaded will be executed on all loaded modules for each addon, after it is loaded
* Execute OnAllAddonsLoaded on all loaded modules
  + OnAllAddonsLoaded script hook is also run here
  + This step will always occur even if addon loading is not enabled
* Run garbage collection
* Set Initialized to *true* and Initializing to *false*

There are three variations on how startup is timed, which can be configured (separately for editor and player) in CoreParams:

* SynchronousEarly
  + All initialization occurs in a single frame, before scene load
  + Formerly (before 2.0.0 Preview 12) how CommonCore always started up
  + Causes a significant stutter, and the initial loading screen is only seen briefly in builds
  + Does not support addons or StreamingAssets loading
* Synchronous
  + All initialization occurs in a single frame, after scene load
  + Still causes a significant stutter but the initial loading screen can be seen
  + Does not support addons or StreamingAssets loading
* Asynchronous
  + Initialization occurs over several frames, after scene load
  + Is slower than synchronous loading but allows the loading scene to actually run
  + Supports addons and StreamingAssets loading, if enabled

The InitScene (see the next section) has some scripting in it which checks Initializing, Initialized, and LoadSceneAfterInit. It waits for CommonCore to finish initializing and then goes to the next scene, usually MainMenuScene. It will *always* wait until initialization is complete no matter what startup type is used. Unless you are doing something very out of the ordinary this is why your game should start with InitScene.

There is some hackery to allow testing from an open scene in the editor. With SynchronousEarly, everything works fine since CommonCore is fully initialized before any scene scripts can run. With the other modes, things will break as scene scripts try to access modules and resources that have not been initialized. So in Synchronous mode, we load everything and then reload the scene on the next frame. In Asynchronous mode, we go to the InitScene, wait for asynchronous loading to complete, and then reload the scene we started in.

Modules are loaded one at a time, even in asynchronous mode. Modules deriving from CCAsyncModule have their Load or LoadAsync methods called immediately after they are created. Neither of these are strong guarantees.

Most of the actual magic happens in modules, some of which are considered integral to Core and others of which are more separated.

## CommonCore Scene Flow

A basic CommonCore project has a few “built-in” or “system” scenes in addition to your game scenes.



The standard way to customize these scenes is to duplicate the scene into the Scenes folder, making sure it has the same name, and make edits to that. Remove the old scene from the build and add yours, as long as the name is the same it should work perfectly.

For example, Ascension III at the time of writing looks like this:



Note especially GameOverScene.

The CommonCore included scenes provide a “skeleton” for the game. For the most part these are fairly self-explanatory: InitScene handles initialization, MainMenuScene is the main menu, LoadingScene is the loading screen, and GameOverScene is a game over screen.



InitScene should always be the first scene in your build. It is a loading screen displayed when CommonCore is loading. Once CommonCore has loaded a script in InitScene automatically loads the main menu. If you customize this scene, do not attempt to use CommonCore services in your scripts, as nothing is guaranteed to be initialized until after startup has completed and the scene has transitioned to MainMenuScene.

Nearly all scene transitions go through LoadingScene. While there is an option to SkipLoadingScene, it actually only hides the visuals of the loading screen, it doesn’t skip LoadingScene. The LoadingSceneController handles cleaning up and/or loading state when starting a new game, loading a game, or ending a game, calling intents and script hooks, and eventually looking nice.

The one significant exception to this is when going to the Game Over screen via SharedUtils.ShowGameOver. This simply loads the GameOverScene, setting the current scene as MetaState.NextScene so we can return to it later if configured to do so. Once the “end game” option is chosen we end the game properly, going through LoadingScene.

MetaState contains the information necessary to make the scene transition. Normally, you would not interact with this directly, instead convenience methods are provided in SharedUtils and WorldUtils.

Additional actions are performed by the SceneController on scene exit, if present. This includes saving state to GameState (*committing* the scene, described later) if configured to do so. It may also include performing a full autosave.

## CommonCore Scene

There is also a common structure to CommonCore scenes. At a minimum, they should have a valid WorldRoot object with a valid SceneController and all world/in-game objects below it. A valid UIRoot object



If these are not at (0,0,0) undefined behaviour may result. Other than that and the SceneController requirement for WorldRoot, the only thing that matters is the names.

A SceneController in the CommonCore sense derives from BaseSceneController and provides lifecycle handling and save/load for the scene as well as calling script hooks. It also provides a LocalStore key/value store that is committed and restored with scene state. Subclasses, such as the WorldSceneController provided with RPGGame, can customize things and provide more functionality.

In the example we have used BlankSceneController which does not provide any extra functionality and is intended for simple scenes where only the basics are needed.



**Autosave On Enter** and **Autosave On Exit** are self-explanatory. Note that these will not override globally configured save settings. **Auto Restore** and **Auto Commit** configure whether scene state will be saved and restored to GameState or not. See the sections on save/load for more info. **Autoinit Ui** and **Autoinit Hud** control whether the in-game menu and HUD are loaded on start, respectively. By default, the UI is loaded from UI/IGUI\_Menu and UI/DefaultEventSystem and the HUD is UI/DefaultHUD. These use the resource loading system and will be affected by redirects and overrides done there. Additionally, subclasses of SceneController can change the default HUD object, and it can always be overridden with the **Hud Override** field (note that it still expects it in the UI/ folder). Finally, if **Autoinit State** is set the scene controller will check if GameState is initialized on start and if not will initialize it, also calling the OnGameStart hook. This is intended for editor use; this situation should never happen in normal gameplay.

At runtime, we get a few more objects created dynamically. Some of these differ based on how the scene and project are configured.



You can see the InGameMenu, EventSystem, and HUD objects created here. The EphemeralRoot object is used for modals and other UI objects attached to the in-game menu, and its contents are purged whenever the in-game menu is closed. It’s a bit of a hack and in retrospect I wish I’d gone with a full stack-based UI.

The objects under DontDestroyOnLoad are created by the framework. CCMonoBehaviourHook propagates certain events that can only be received by MonoBehaviour objects through the CommonCore systems. FPSCounter is self-explanatory and is created by the DebugLog module. AsyncCoroutineRunner is part of the Async module. BasicConsole is our command console. TimeHook is used to call delayed events. AudioPlayer and its children handle audio playback and are part of the Audio module.

## Tags, Layers, and Sorting

Tags are set up like this



The EphemeralRoot tag is used to find the EphemeralRoot object. The CCObject tag was once going to be used to tag CommonCore Entities, but now isn’t used for anything. Both of these are considered deprecated.

The built-in *Player* and *MainCamera* tags are also used for player and camera lookup though they are not the only criteria.

The poorly named CommonCore Entity Tags are described in the World section. These have nothing to do with Unity Tags, and probably should have been called Flags or something.

Layers are set up like this

Table

Description automatically generated A picture containing text

Description automatically generated

Layers 8-12 are used to set up specific collisions between world objects. For the most part, world objects including terrain should remain on the Default layer. The BlcokActors layer is for objects that block actors while letting other things through. Bullets and bullet-like objects should be on the Bullet layer, which makes them not collide with other bullets and BlockActors objects. NonShootableEffect is intended for things that need to collide with the world but little else, like shell casings.

Layer 22, PostProcessVolume, is used solely to define volumes for postprocessing. If you aren’t using PostProcessingStackV2 or aren’t using volumes, you can ignore this.

Layer 23, ViewModel, is used for the player’s weapon view model. It is rendered by a separate camera and collides with nothing.

Layer 24, LightReporter, is only used for probed light reporting used for sprite weapons.

Generally, you should render all layers except 22, 23, and 24.

Note that as of 2.0.0 Preview 15, Actor and Bullet no longer have collision with each other. Add a hitbox on the ActorHitbox layer or enable raycasting on the bullet.

Sorting Layers are not used, however, specific values are used for “order in layer” to sort the UI correctly.

World HUD: 0-999

Fader (below HUD): 1

Default: 100

Dialogue: 200

Container: 300

Fader (above HUD): 999

Ingame Menu: 1000-1999 (modals above 1499)

Main: 1080

Level up modal: 1500

Input config: 1520

Other modals: 1600

Save indicator: 1701

Debug: 5000-5999

Basic console: 5700

FPS counter: 5800

## Modules and Assemblies

Code (and, for that matter, assets and resources) in a CommonCore project can be divided into four sort of levels:

* Core
  + Contained in CommonCore folder
  + May not depend on anything except Unity and shared libraries
  + Resources in Resources/Core, always loaded first
* Modules
  + Contained in CommonCoreModules subfolder
  + Depends on Core, may depend on each other or on Game package
  + Resources in Resources/Modules/<modulename>
* Game module
  + Contained in CommonCoreGame folder
  + Depends on Core, may depend on modules
  + Resources in Resources/Game, loaded after Core but before loose resources
* Your game
  + Not contained in a specific folder
  + Depends on everything, cannot be depended on by Core, Modules, or Game

Early on, the intent was to eliminate dependencies between modules and dependencies between modules and the game package. It was realized quickly that this was rather optimistic, and for a long time CommonCore was a mess of interdependencies. These have been cleaned up and while there are still many dependencies, they now conform to a specific set of rules:

* Core must not be dependent on any modules.
* All modules are dependent on Core. Modules may depend on Game OR may be depended on by Game, but not both. A module may be dependent on another module so long as no dependency loops direct or indirect are created.
* Game may depend on Core and may also depend on modules that do not depend on Game and do not depend on modules that depend on Game.

I *think* this can be generalized to “the dependencies must form a directed acyclic graph” but my grasp on graph theory is limited.

Ironically, a lot of functionality ended up being moved into Core when we cleaned things up.



This is the diagram for Ascension III with some optional modules and actually it’s not so bad. Note that it’s missing a few modules that were added after this diagram was drawn.

For more details on how these interact with resource management, see the *Resource Management* subsection within the next section.

Currently, the only Game module that exists is RPGGame, though there was some experimental work with SideScrollerGame done for Heavy Metal Slug (can be found in that project’s repository). Only one Game module may be used at once. It is not expected that there will ever be a different game module created, as RPGGame fulfills most of my needs, but it’s theoretically possible.

Modules do not strictly correspond to assemblies. The Core module is mostly contained in CommonCore.Core.dll (CommonCore/Core folder), but some of it is in Assembly-CSharp.dll (CommonCore/CoreShared folder, no assembly definition file). Modules have been given their own assemblies where possible but there are exceptions, notably the large World module.

Each module should, however, have its own namespace.

Yes, I know Unity’s advice is to either use Assembly Definition Files entirely or not at all. This is what happens when you don’t know about those when you start the project. I suspect a lot of real-world projects mix the two as well.

Although some modules provide Instance properties, the preferred way to get a module reference is now CCBase.GetModule<T>. Some modules, mostly really old ones, also use static methods for functionality, which is now discouraged.

## State Objects

Three state objects are provided for storage of game state, each with well-defined lifetimes:

* **PersistState** exists until it is explicitly cleared. It is automatically loaded on startup and automatically saved on exit. Note that it is still best practice to save after making changes. By default it provides an untyped key/value store and IsFirstRun status.
* **MetaState** exists for the duration of program execution, and is not saved or restored on exit or launch. This is used for storing temporary state between scene transitions, including crucial things like *which scene to load next*. Mostly this is used internally. It is partially soft-reset when a game is loaded, started, or ended. It offers specifically scoped stores as dictionaries as well.
* **GameState** exists for the duration of a game session, ie between start or load and game end. It is saved and loaded in its entirety when the game is saved or loaded by the user or via script. A fresh GameState is created for a new game.

They are all singletons\* accessible via *.Instance*, which is bad but oh well.

\*Technically, each has a well defined lifetime and can be null. PersistState and MetaState are made available very early in the startup process, while GameState is created when a game is started or loaded from a save. You can check if GameState exists via GameState.Exists, and this is semantically equivalent to checking if the player is currently in a game. For more details on GameState and how saving and loading works, refer to the sections on save/load.

These are deliberately not located in separate assemblies, and are defined as partial classes to make it easy to add your own variables. This is done when using the full framework; very few properties are defined in the Core GameStateBase.cs, and a lot more are defined in the World module GameState.cs, Campaign module GameState.cs, and RPGGame GameState.cs.

You can use Newtonsoft.Json annotations to control serialization. Additionally, there are a few other attributes you can use.

**GameState**

* **Init**
  + If attached to a method, the method will be run when GameState is first created/reset. Priority can be specified, methods with a higher priority number will be run first (0 is default).
* **AfterLoad**
  + If attached to a method, the method will be run when GameState is loaded from file. Priority can be specified, methods with a higher priority number will be run first (0 is default).

**MetaState**

* **Clear**
  + If attached to a property, the property will be reset when MetaState is soft-reset\*. Can be combined with System.ComponentModel.DefaultValueAttribute.

\*MetaState is soft-reset via MetaState.Clear() when a game is started, loaded, or ended.

There is a similar class with somewhat different handling used for storing config information, called **ConfigState**. ConfigState is saved and loaded by the Config module and is not designed to be extended. Instead, you can use the CustomConfigFlags and CustomConfigVars properties to store your own config, using their related convenience APIs if you prefer. Core configuration properties are part of the class itself.

**Migrations** are used for ConfigState and PersistState to handle changes between versions. A single migration is provided for each: ConfigStateUnifiedMigration and PersistStateUnifiedMigration respectively. Normally, you will not need to edit these. They are set up to handle changes between CommonCore versions. But if you need to handle model changes between your versions you can handle them in these classes. When a migration happens, a backup of the old version will be saved under <CoreParams.PersistentDataPath>/migrationbackup if CoreParams.UseMigrationBackups is set or CoreParams.IsDebug is true (ie in editor or development build). Migrations are not provided for GameState but can be added.

### More on save/load

Functionality for saving and loading GameState to file is split between SaveUtils and SharedUtils. SaveUtils contains helper methods meant to be used in the process of saving and loading as well as convenience methods for performing quick- and autosaves.

Quicksave and autosave functionality is wired up by default and set up to display an indicator on screen. You probably won’t have to touch this, other than to maybe manually call DoAutoSave from scripts if you want to have that at explicit points in your game.

The main functions for saving and loading are located in SharedUtils. They are well commented, but you will probably never have to touch them.

The ability to save/load can be controlled by a few properties in CoreParams and GameState:

* CoreParams.AllowSaveLoad
  + Enables/disables the save/load functionality entirely
* CoreParams.AllowManualSave
  + Enables/disables manual save including quicksave, does not affect autosave
* GameState.SaveLocked
  + Enables/disables saving this GameState
* GameState.ManualSaveLocked
  + Enables/disables manually saving this GameState including quicksave, does not affect autosave

As of 4.0.0 Preview 2, *save metadata* is included in the save file. This is used to display a thumbnail and some information in the load game panel, but if it does not exist, this case is gracefully handled and the save can still be loaded. It is generated just before save, and gets a thumbnail by calling the script Save.GetSaveThumbnail if it exists (it will not save a thumbnail if it does not).

If you absolutely need to get the loaded metadata, you can use GameState.AdditionalData.Metadata, but isn’t really defined behaviour and shouldn’t be considered a stable API. The fact that metadata is in the save file itself at all is considered an implementation detail and may change.

The exact sequence of events when saving is call the BeforeSaveSerialize script hook, serialize the GameState to an intermediate JObject, insert the metadata, call the BeforeSaveWrite script hook, then save the json file.

Before loading a save, CommonCore will attempt to run migrations if they exist. At the time of writing, no stock migrations are provided. Like with other state objects, a backup will be saved if CoreParams.UseMigrationBackups is set or CoreParams.IsDebug is true. CommonCore will then call the AfterSaveRead script hook, deserialize the save data into GameState.Instance, then call AfterSaveDeserialize.

GameState includes a “campaign identifier” uniquely identifying the game instance from start to end across reloads (this can be disabled with CoreParams.UseCampaignIdentifier), as well as start date/time, game and company name, current version, and current/highest/lowest difficulty.

# CommonCore Core

## Core Params

CoreParams contains two groups of properties which probably should have been kept separate:

* Basic CommonCore configuration settings
* Convenient and thread-safe versions of Unity built-in properties, mostly paths and version information.

Generally you should not edit anything that lacks a default and is noted as being automatically set.

Notable configuration settings include:

* **VersionCode** and **VersionName**
  + These are for the CommonCore library and probably shouldn’t be changed by you unless you’re forking the library itself.
* **GameVersionName**
  + A nice name for the specific version of your game. I don’t know why this exists.
* **ExplicitModules**
  + A list of modules that is loaded explicitly on startup. Mentioned earlier in this document.
* **PreferredCommandConsole**
  + The preferred command console implementation to use. If it cannot be found, BasicConsole will always be tried before falling back to the null implementation.
* **DefaultResourceManager**
  + Which resource manager to use (Legacy or New). Can be set up to use the results from one but test both. Default is now to use the new resource manager exclusively.
* **LoadAddons**
  + Whether to enable the addons system (which also handles loading streaming resources at startup).
* **EditorStartupPolicy**
  + Whether to use the SynchronousEarly, Synchronous, or Asynchronous startup sequence when running in the Editor. Refer to the startup sequence section for details
* **PlayerStartupPolicy**
  + Which startup sequence to use when running in a standalone player rather than the editor.
* **PersistentDataPathWindows**
  + Which folder to use as a PersistentDataPath on Windows. Note that this only affects CoreParams.PersistentDataPath, not Application.PersistentDataPath, and Unity will still write things to the folder it thinks is the right path.
* **CorrectWindowsLocalDataPath**
  + If set, will use AppData/Local/\* instead of AppData/LocalLow/\* for LocalDataPath. This only affects CoreParams.LocalDataPath and Unity will still write things to the other folder.
* **UseGlobalScreenshotFolder**
  + If set, will use the system Pictures folder for screenshots (in a Screenshots subfolder). Otherwise, screenshots will be saved in a subfolder in PersistentDataPath.
* **SetSafeResolutionOnExit** and **SafeResolution**
  + Used for resolution handling. If configured, CommonCore will set the game’s resolution to SafeResolution and mode to Windowed on exit. This is saved by Unity and will be used on the next startup. The intent is to force the game to start in a safe resolution. Note that by default, CommonCore’s config system will always try to set the user-configured resolution on startup.
* **InitialScene**
  + The scene to enter when starting a new game.
* **UseCampaignIdentifier**
  + If set, will add a (theoretically) unique identifier to GameState when a new game is started. This can be useful but could create privacy concerns.
* **UseCampaignStartDate**
  + If set, will add the creation date and time to GameState when a game is started. This can be useful but could create privacy concerns.
* **AllowSaveLoad**
  + Enables/disables save and load globally. More intended for games where save/load isn’t implement than games where the player is not allowed to save. Note that it is *always* possible to save and load through the console.
* **AllowManualSave**
  + Enabled/disables manual saving (including saving from the menu and quicksaves) globally. Autosaves, explicit scripted saves, and console saves are still possible. This one is intended for games where the player is not allowed to save.

Notable properties include:

* DataPath
* GameFolderPath
* PersistentDataPath
* SaveBasePath
* SavePath
* FinalSavePath
* LocalDataPath
* DebugPath
* StreamingAssetsPath
* ScreenshotsPath
* CommandLineArgs

CoreParams is located in CommonCore.Core.dll and is accessible to all assemblies that reference it, which is probably all of the assemblies in a CommonCore project.

CoreParams can be overridden on startup from a coreparams.json file located in the game’s root directory. Outside of *maybe* some rare debugging scenarios, you should never do this. In theory values could also be overwritten by reflection at runtime, but this also is something that probably shouldn’t be done.

**Interim Module Params** are also handled by CoreParams. It is a read-only dictionary of key-value pairs that can be retrieved via GetParamsForModule. All entries starting with the module name and a delimiter (typically .) will be aggregated into a dictionary and returned. No type conversion will be done. As is implied by the name, this is a crude interim system and the plan is to (eventually) come up with something better to provide parameters to modules.

## Core Utilities

*The following utilities are defined in CommonCore.Core.dll*

**CoreUtils**

Contains LoadResource\* APIs (see below), Json convenience methods, fast methods for getting WorldRoot and UIRoot, GetSceneList and Quit convenience methods.

**CollectionUtils**

Contains some extension methods for various collections, including but not limited to: Dictionary GetOrDefault, GetKey(s)ForValue, Array/List element swap, Enum->Dictionary setup, Array/List shuffle, IReadOnlyList IndexOf.

**MathUtils**

Contains some math functions. It will have more eventually.

(We had a similar class in a project I worked on professionally, I thought I’d need one here too, but that project had a lot more math than my janky games…)

**SceneUtils**

Contains some utility methods for manipulating Unity things in Unity scenes, including a few extension methods on Transform.

**StringUtils**

Contains (or will contain) some utility methods for manipulating strings.

**TypeUtils**

Contains utilities for type conversion, coercion, introspection, and general fuckery. Of special note is the Ref() extension method which allows you to use the ?. and ?? operators with Unity’s fake null, IsNullOrEmpty for JToken, the IsNumericType and IsIntegerType functions, and of course the weirdness of CoerceValue (aggressively tries to convert a value to a target type) and StringToNumericAuto (converts strings to numeric types unless it can’t). And of course AddValuesDynamic which is like 3 lines in Mono+.NET 4.x but ten times longer if dynamic isn’t available.

Actually this whole class is pretty entertaining. Probably not what you want to ever hear from your framework author but oh well.

**VectorUtils**

Contains some vector math functions. It will have more eventually.

(Another one where we had something similar at work and I figured oh, I need something like that, and didn’t. I think Firefighter VR had something similar also, but it had more fuckery in general.)

**Types defined in Core**

This is not a complete list

* CommandAttribute (used for defining console commands)
* Envelope structs:
  + RangeEnvelope (min/max/gain/decay)
  + IntervalEnvelope (min/max)
  + RandomEnvelope (average/spread)
  + PulseEnvelope (intensity/time/violence)
* StringCase enum
* Message classes (used for the messaging system)
  + QdmsMessage (abstract base class)
  + QdmsFlagMessage
  + QdmsKeyValueMessage
  + HudPushMessage
  + HudClearMessage
  + SubtitleMessage
* LazyLooseDictionary

**Convenience methods in Core**

~~SkippableWait.WaitForSeconds(Realtime)~~ [OBSOLETE]

Waits for a specified period of time but allows the player to skip

~~SkippableWait.Delay(Scaled/Realtime)~~ [OBSOLETE]

Similar to WaitForSeconds but async Task instead of Coroutine/IEnumerator

Use the WaitForSecondsEx class instead.

Modal.Push\*Modal(Async)

Pushes a modal popup window to the screen. Available in callback and async variants. Requires CommonCore.UI to be included

TextAnimation.TypeOn

Does a “type on” effect with some text. Never used, probably untested

Subtitle.Show  
Subtitle.Clear

Convenience methods for throwing subtitles onto the screen

*The following are defined in Assembly-csharp.dll which limits their visibility to other assemblies.*

**SharedUtils**

Contains methods for starting, loading, and ending game, as well as showing Game Over screen, changing scene and saving the game. It’s highly recommended to use these instead of manipulating things directly to accomplish these tasks.

Also contains GetSceneController for some reason.

**SaveUtils**

Contains methods for saving, loading, and handling savegames, including one-line solutions for autosave and quicksave. Again, it’s recommended to use these rather than rolling your own.

It’s hiding out in the CoreShared/State folder

**Convenience methods in CoreShared**

ScreenFader.Fade(To/From)  
ScreenFader.Crossfade  
ScreenFader.ClearFade

Use these to fade the screen in and out, and clear the screen fade. Described more later in this document

MusicFader.Fade(To/In/Out)  
MusicFader.ClearFade  
MusicFader.ClearAllFades

Use these to fade music in and out. Hacky; these operate on channel volume.

## Resource Management

CommonCore has a concept of Resources which is related to (and based on) the Unity Resources system, but with some differences. Like Unity’s Resources, resources are accessed by path and type, and it has a simple synchronous API. However, paths are virtual and resources may have multiple variants.

The path a resource is accessed by may not be the actual path in which it resides (if there is one- see below). The top-level folders *Core* and *Game* are treated specially. The contents of these folders are treated as if they were located at the root, and are loaded at a lower preference to other folders.



In this example, we make a call to CoreUtils.LoadResource<T>, looking for a TextAsset at the path Data/Strings/mystrings (note that like Unity Resources we do not specify an extension). In this case, there is an appropriate resource located at Data/Strings/mystrings, but if there was not, we would try Game/Data/Strings, then Core/Data/Strings, and only return a null result if *none* of these paths worked.

It is also possible to get all the resource variants:



Note that the returned resources are sorted by priority ascending, with the highest priority resource at the *end* of the collection.

Similar APIs exist for operating at a folder level, (such as Data/Strings/). The LoadResourcesVariants API, which is now preferred, returns [resource][variant] while the old LoadDataResources API returns in the form [priority][resource] and will probably break horribly with the new ResourceManager (see below).

The Core/ and Game/ folders are used for “base” resources included with CommonCore and the game module, as you may have surmised. As mentioned, they are loaded at a lower priority level. Addons and streaming resources are loaded at a higher than normal priority level.

In fact, the following top-level folders should all be treated with some level of care:

* Core
  + **Do not use for general purposes**. Contains resources that are part of CommonCore Core.
  + **Retrieve with Resources**.**Load if necessary.** Generally you would never need to do this.
  + **Runtime overrides are not possible.**
* Game
  + **Do not use for general purposes. DO use if you’re creating a new game module**. Contains resources that are part of game module.
  + **Retrieve with Resources**.**Load if necessary.** Generally you would never need to do this.
  + **Runtime overrides are not possible.**
* Modules
  + **Do not use for general purposes. DO use if you’re creating a new module**. Contains resources that are part of module, each in its own folder.
  + **Retrieve normally**.
  + **Runtime overrides are possible but not recommended**.
* Addons
  + **Do not use.** Addon elocal resources will be mounted in folders here.
  + **Retrieve normally**.
  + **Runtime overrides are possible but not recommended**.
* Streaming
  + **Do not use.** StreamingAssets elocal will be mounted here.
  + **Retrieve normally**.
  + **Runtime overrides are TBD, but likely not recommended**.

Additionally, **all paths starting with the underscore character ‘\_’ should be avoided** as they are reserved for future use.

Note that there are multiple physical Resources folders in the project repository. Unity mashes these all together. The Resources folder in the filesystem, Unity Resources folder, and CommonCore Resources are all related but different concepts, it’s confusing I know.

The new ResourceManager supports additional resource variants, resources backed by runtime assets or files instead of Unity Resources, and path redirection. For the most part, this is to support addon functionality. As of 2.0.0 Preview 16 it is now enabled by default on most platforms.

Always use CoreUtils.LoadResource and variants unless you have a *very* good reason not to\*. It’s slightly slower and more obtuse than Resources.Load, and the current amount of redirection isn’t very exciting, but when mod support and runtime resource redirection are added you won’t have to change your calls for things to work.

\*For instance, in early module init when ResourceManager has not yet been initialized.

If you need more functionality than LoadResource can provide, your next stop should be the ResourceManager itself, accessible through CCBase.ResourceManager. This provides access to the ResourceLoader and ResourceManifest as well as methods for adding resources and retrieving/manipulating underlying resource handles. Be careful when working with handles directly!

The new ResourceManager is not available on some platforms (notably WebGL). If you are working with these platforms, you will need to check for its existence before attempting to utilize it.

Unity’s built-in resource system does not allow recursively enumerating folders. As a workaround, the Resource Manifest system is provided. This is a list of resources created at build time and added as a StreamingAsset (it is simulated with editor-only APIs in the editor). You can retrieve this through ResourceManager.ResourceManifest, and then enumerate folders in a folder. Note that this will only get folders that existed at build time. ResourceManager provides GetSubfolders and GetSubfoldersRecursive methods that leverage the resource manifest and also include folders loaded at run time.

Like the ResourceManager, ResourceManifest may not be provided on all platforms. There are settings in CoreParams to enable/disable resource manifest functionality and to enforce it as required or not. If you don’t want to require it but want to optionally use it you will need to check for its existence yourself before using it.

## Addons

As of 2.0.0 Preview 16, CommonCore now has support for loading addons. In order for addon support to work, the following conditions must be met:

* Running on a platform that supports addons. Anything IL2CPP is out, anything that doesn’t support async is out. Basically just standalone desktop.
* Startup policy set to Asynchronous
* LoadAddons set to true in CoreParams

If all this is set up, addon support will be enabled and resources will be loaded from StreamingAssets automatically. However, to actually load addons, LoadAddons must be enabled in the game’s config file and the package names added to AddonsToLoad. The order of the package names in AddonsToLoad defines the load order.

Addons are loaded from the following locations, from lowest to highest priority in the event of a conflict:

* <StreamingAssets>/Addons
* <Game Folder>/Addons
* <Local Data Path>/Addons
* <Persistent/Roaming Data Path>/Addons (beside the config file)

By convention, addons should be in a folder matching their package name but the addon loader doesn’t actually care about this, only the Name field in the manifest file. It does however need to be in its own folder.

A typical addon looks something like this:

 

At an absolute minimum, the addon must contain a valid manifest.json file, meaning at minimum a Name field as that is how the package name is defined. It won’t do anything useful, though.

When an addon in the load order is located, the addon loader first attempts to read its manifest. Take a look at the AddonManifest model class for some possible options. If MainAssembly is defined, it will then load that assembly from the *managed* subfolder. It will then look for a class deriving from AddonBase, create an instance, and call LoadAddonAsync on it. If there is no main assembly or it does not contain an AddonBase derived class, the default AddonBase will be used. (multiple AddonBase derived classes will result in undefined behaviour)

The default AddonBase will load the rest of the assemblies in the managed folder, then load resources, then call OnAddonLoaded, passing it lists of newly loaded assemblies and resources for modules to handle. By creating a custom AddonBase you can override this behaviour. You can reuse some or all of the default behaviour by calling back to AddonManager via the passed in AddonLoadContext.

There are two resource folders: elocal and expand. They differ in how they are “mounted” in the resource manager. The contents of the elocal folder are mounted at Addons/<package name>/ while the contents of expand are mounted at the resource root, possibly overriding other resources.

Loose resources are loaded through ResourceLoader, which provides a limited set of default importers for images, audio, and text assets. This can be extended by registering instances of IResourceImporter to the ResourceLoader. This is currently done for Sprite and FacingSpriteAsset assets, both of which are defined in .jasset files that can be recognized and loaded by their respective IResourceImporters. A .jasset file, by convention, is an asset defined in json that can be used by an importer, take a look at the examples.

Note that files with names starting with ‘.’ (like UNIX-style hidden files) and files with the extension .meta will be ignored by the default loader as of 3.0.0.

All of this is very primitive and limited at this point but will be improved in future releases.

AssetBundles are loaded slightly differently. First, AssetBundles must have the extension .assetbundle as this is how they are identified. Second, the contents of each AssetBundle are mounted in a directory at their path. For instance, the contents of Dialogue/char.assetbundle would be mounted at Dialogue/char/ . This means that you can have an elocal.assetbundle and expand.assetbundle. The full paths of the resources in the AssetBundle are ignored by default; this can be enabled in the addon manifest as an experimental option.

Scene asset bundles work fine. Put them in elocal. They can be enumerated through AddonManager.EnumerateAddonScenePaths.

If you need to access the AddonBase later, you can do so via AddonManager.GetAddonBase (AddonManager is accessible through CCBase). The default AddonBase isn’t very interesting, its only public property is the elocal resource mount path.

Loading from StreamingAssets is a very similar codepath to loading an addon. However, it does not attempt to locate or read a manifest, does not attempt to load AddonBase (though it will load loose assemblies from managed/ as of 3.0.0 Preview 6), and will happen even if loading addons is disabled in the game’s config (though it must still be enabled in CoreParams). The expand folder/assetbundle is mounted the same way as an addon, while elocal is mounted at the Streaming/ path. Resources are loaded at a higher priority than the base but lower than true addons.

# CommonCore Modules

Modules come in three types:

* built-in, which are tightly coupled with and usually considered part of Core
* bundled, which are separate but usually necessary and may be depended upon by Game
* optional, which are provided in separate repository and are usually unnecessary or experimental

This section is really an overview of modules. Later sections describe some of the modules and how to use them in further detail.

## Async

*Built-in module*

Provides convenience methods and classes that allow awaiting Coroutines and yielding on Tasks. Probably a bit buggy. Also includes a few utility/convenience methods for working with Tasks, including RunWithExceptionHandling which should be used instead of async void.

Usage is:  
 await <coroutine>().AsTask();

yield return new WaitForTask(<task>, <throwexceptions>);

## Audio

*Built-in module*

Includes functionality for playing sounds and music.

Sounds can be retrieved with the AudioModule.GetSound convenience method, or played through the PlaySound(ex) methods of AudioPlayer. AudioPlayer is a singleton and AudioPlayer.Instance should be used to access it (not the greatest design in retrospect). The intent was to make it very easy to play sounds with a minimum of setup.

Music is handled separately, also through AudioPlayer. There are five slots, and playing music in a higher slot overrides music in lower slots until it is stopped/removed. The slots in order from lowest priority are: Ambient, Event, User, Cinematic, and Override. The User music slot is treated specially (notably if a user music component is enabled, it is considered “playing” even if there is no track) and is for use with user music components, a horrendously complicated thing intended for ingame radio stations and the ingame music player.

The module is set up to search for sounds in Resources/DynamicSound, Resources/Voice, and Resources/DynamicMusic It goes through CoreUtils and ResourceManager so any overrides or redirection done there will be used.

Also includes a MusicSetterScript which can be added to a scene to set music on start.

## Config

*Built-in module*

Includes ConfigState, handles the lifecycle of ConfigState, and handles applying configuration settings. Also handles adding additional panels to the config panel with RegisterConfigPanel/UnregisterConfigPanel.

A ConfigChangeMessage is sent whenever config is applied.

## Console

*Built-in module*

Provides abstraction for developer/command console. Commands are designated with the [Command] attribute, and command console implementations implement IConsole. The preferred console can be set in CoreParams but the Console module will fall back to an available implementation if that is not found. BasicConsole is the fallback if present, otherwise a null implementation will be used.

There’s some more information on using the Console elsewhere in this document.

## DebugLog

*Built-in module*

Catch-all submodule for debug utilities. Includes a dirt-simple FPS counter, some debug utilities (mostly for writing out data), screenshot functionality, and alternate logging functions (CDebug).

At one point CDebug.Log(ex) was preferred over Debug.Log but in practice it’s less than useful. Both the console and any future logging implementation will capture Debug.Log output

## GameData

*Built-in module*

Provides a relatively easy way to get arbitrary game data that can be overridden. Create a model class and use CoreUtils.GetGameData<T> to get an instance of it. It will look in /Data/GameData for JSON data corresponding to the name of the class (not a qualified name), and if that is not found return a new instance instead. These are always cached; once a data object is loaded or created it will be returned until explicitly cleared.

The design and implementation of this is still somewhat WIP, though stable for the current version. It also reloads *all* cached data every time an addon is loaded, which is not great.

## Input

*Built-in module*

Provides abstracted input support. The API is basically identical to UnityEngine.Input. It also provides a set of default control names.

All input is routed through InputMappers which can be swapped in and out. InputMappers are ephemeral and may be created and destroyed a lot, and at random times. If you have heavy initialization and/or need to store state, create a Module for your actual logic and use the InputMapper as a thin wrapper over it.

OpenMenu is hidden in CoreParams by default and probably shouldn’t be remappable, but you can hook something to it if you want.

By convention, you should avoid mapping anything to the Esc or tilde/backtick keys as these are used by the menu and console respectively.

There’s an InputModule for EventSystem that uses MappedInput but I’m not sure if it’s technically part of this module. It is included with Core.

//TODO input will have its own section eventually

## LockPause

*Built-in module*

Handles pausing the game and locking input for menus, cutscenes, etc. Capturing the mouse is also handled through this module.

The API is implemented as a set of static methods (it’s one of the oldest systems) and is fairly simple. Pausing and locking controls are handled separately, and there are “levels” of pausing or locking. Input locks can be All, GameOnly, or MoveOnly while pause locks can be All, AllowMenu or AllowCutscene. Yes, the semantics of those are slightly obtuse; “GameOnly” means “only game input is locked out” while “AllowMenu” means “game is paused but menu is allowed”.

Conceptually, you “take out a lock”, specifying a lock level and a unique token. At some point you then “release the lock” either explicitly or when the token ceases to exist. The PauseLock module does not use WeakReferences internally but does offer specific handling for UnityEngine.Object/Unity-fake-null and WeakReference/WeakReference<T> so those ~~will~~ should work as expected. The currently active lock level is the most restrictive of all the locks that are currently taken out.

The main APIs are LockControls/UnlockControls and PauseGame/UnpauseGame.

There are also, of course, methods for querying the lock states, which are needed to actually implement the functionality (input does not pass through this module, though it does pause the game by setting TimeScale to 0).

## QDMS

*Built-in module*

The built-in broadcast messaging system. Includes a message bus, message receivers and a few message types. All messages must inherit from QdmsMessage.

The messaging system is used for many things, including notifying of config changes, updating the HUD, and handling subtitles.

“QDMS” stands for **Q**uick and **D**irty **M**essaging **S**ystem. It was originally hacked together in one afternoon and isn’t great, but has been somewhat improved over time.

See also the section on using messages.

## Scripting

*Built-in module*

Handles calling arbitrary methods- “scripts” in this context. These can be called by name or attached to “hooks” and executed at specific points in program execution.

Scripts are declared with the [CCScript] attribute. Visibility doesn’t matter, but static methods are preferred; there is some handling for instances and non-static methods but it is somewhat hacky. The attribute provides options for overriding the name and class name used for lookup. Hooked scripts must have both the [CCScript] and [CCScriptHook] attributes. By default, these cannot be called explicitly through the scripting system, but this can be enabled with the AllowExplicitCalls option on the CCScriptHook attribute.

If a script takes a ScriptExecutionContext object as its first argument, this will be passed in. Note that because of the quirks of type coercion, this can sometimes happen if the argument type isn’t exactly ScriptExecutionContext (say, if it’s Object). If you don’t want this, there is an option on the CCScript attribute to disable passing ScriptExecutionContext.

For this reason overloads also are not supported.

ScriptExecutionContext specifies where a script was called from and will someday specify more.

There are some APIs for adding new Scripts at runtime but these are mostly WIP/experimental and don’t really work. The one that takes a delegate in particular isn’t implemented at all, and won’t be until significant reworking is done.

The APIs for calling scripts are just called Call\*, and are implemented as static methods in ScriptingModule. These will automatically create a ScriptExecutionContext and will automatically coerce argument types to fit the target script if possible. These do throw exceptions on failure, usually ScriptNotFoundException or ScriptExecutionFailedException. The latter exception will generally contain a more specific inner exception.

Be very careful about calling CallHooked, as most script hooks have well defined execution times and are already called by built-in code. CallNamedHooked is designed for custom hooks in user code. Neither CallHooked nor CallNamedHooked throw exceptions; they log exceptions from each failed script and continue.

//will likely move or duplicate some of this to the “using scripts” section

## State

*Built-in module*

See the documentation on state objects. The status of State as a separate built-in module/submodule is largely a legacy thing from when CommonCore was a sea of separate modules, before a lot of functionality was merged into Core.

Recently (2.0.0 Preview 12) handling of world time has been moved into the State module. It is responsible for updating the world time in GameState and calling the OnWorldTimeUpdate script hook.

Note that unlike all other built-in modules, State is contained entirely inside the CoreShared folder and gets built into Assembly-CSharp.dll.

## StringSub

*Built-in module*

Handles string substitution; replacing bits of strings with runtime values or lookups from lists. This is used in the dialogue system and many parts of the UI.

The string substitution system supports both a direct “lookup name from list” replacement and a more complex macro/format based replacement which can do more complex things multiple times in a string (note that it is NOT recursive, because I do somewhat value my sanity).

The general format of a macro is: <general pattern:option:option…>. For example <l:listname:stringname> does a lookup.

Some substitution, including the lookups, is handled in StringSubModule, while other functionality is delegated to IStringSubber implementations. These are scanned for and instances created on startup via reflection. An IStringSubber specifies which general patterns it can match and provides a method to replace those. For example StateStringSubber provides a bunch of patterns and implementation for substituting RPG stats etc into text.

The lookup tables/lists are loaded from Data/Strings and are in JSON format. This is done through CoreUtils/ResourceManager and uses LoadResourcesVariants so all redirection and overrides and such are handled.

The public APIs are Sub.Replace, Sub.Exists, and Sub.Macro. The former two work with the substitution lists, while the latter works with the macro system. While the replacement simply returns the original string if the target cannot be found, the macro system returns more verbose and ugly errors.

See the documentation on string substitution //TODO documentation on string substitution

## UI

*Built-in module*

The UI module is less a cohesive module and more a catch-all for all the panel controllers and miscellaneous scripts that implement the default menus, in-game menu, and HUD. It is spread across CommonCore.Core.dll (Core folder) and Assembly-CSharp.dll (CoreShared folder).

Of particular interest (perhaps) are the Modal panels accessible through the Modal class, which are general-purpose popups used through the UI.

The UI module also handles UI theming.

Functionality is also provided for adding additional panels to the ingame menu, I’m not sure if this is actually tested though.

## Util

*Built-in module*

Includes some useful types, utilities for type conversion and math, as well as convenience methods for subtitles, music fading, screen fading, and skippable timers/skippable wait.

TODO document this in more detail, here or elsewhere

The status of Util as a separate built-in module/submodule is largely a legacy thing from when CommonCore was a sea of separate modules, before a lot of functionality was merged into Core.

//TODO async test modules?

## Video

*Built-in module*

Handles path resolution for full motion video, allows addons to specify video paths. In other words, if videos are retrieved through this module there is some flexibility for location and addons will be able to override them. Provides utilities to assist in video playback.

## Addon Support

*Bundled module*

Provides some convenience methods, proxy classes, and other resources for addons to use. Does some really weird redirection so that it can be imported into another project to access things that are actually in Assembly-CSharp, which can’t be imported as a plugin. It’s still highly experimental and will hopefully get better over time.

## Basic Console

*Bundled module*

A basic command console implementation. Buggy and feature-poor but functional. I hacked it together in a week as something that’s good enough and can be included out of the box.

If this module is included and no other implementation is available, it will be used.

## Basic Humanoid

*Bundled module*

Provides a basic humanoid model and set of animations. This is indeed a Unity-compatible humanoid which means the animations can and should be retargeted. I made this myself and it is not very good.

## Campaign

*Bundled module*

Provides a CampaignModel in GameState that allows storing arbitrary variables, flags, and quest stages. Was recently (2.0.0 Preview 12) moved out of RPGGame into its own module. Originally part of an effort to move the dialogue system out of RPGGame, work on that has stalled.

## Explicit KBM Input

*Bundled module*

Provides a remappable keyboard/mouse input mapper that wraps UnityEngine.Input. It’s not great but it does work.

Default input map is loaded from /Modules/ExplicitKBMInput/DefaultControls, but this can be overridden by adding an input map at Resources/Data/ExplicitKBMInput/DefaultControls. The easiest way to set up a control map is to set it up in-game and pull the map out of config.json.

## Test Module

*Bundled module*

Provides test scripts and test console commands. You can safely remove this in production projects, it’s solely for debugging.

## Unity Post Processing V2 Integration

*Bundled module*

Provides integration with the Unity Post Processing (V2) stack. You will need to add the package, and then import this module. Provides a tackon; see also the section on setting up cameras.

## World

*Bundled module*

A large catch-all module that is big enough to warrant its own section. World is kind of hard to describe. The best way of putting it would be “the basics of a game’s world” versus core which is “the basics of a game’s infrastructure”.

Conceptually, World contains everything necessary for a game with a 3D world, and most of the basic concepts (ie base classes, interfaces, etc) defining things one would generally have in such a world. It contains logic and components for save/load, hitboxes/bullets, the concepts of Actors and Player (though not implementations themselves), scene controllers and plenty of convenience/utility methods.

Crucially, CommonCore Entities are defined here.

The ObjectActions (Action Special) system and FacingSprites (doom-style sprites) are also contained in World.

Note that RPGGame is dependent on World.

## Achievements

*Optional module – experimental*

Provides utility scripts and UI for in-game achievements. Achievements are stored in PersistState. Currently there is no UI or audio for granting achievements, this is coming later at some point.

## Bigscreen

*Optional module – experimental*

Provides limited controller-friendly main and ingame menus. You will need to add the Bigscreen scenes to the build after bringing in this module.

## CD Audio

*Optional module - deprecated*

Provides playback of CD audio in an incredibly hacky manner. Only used for Beach Defend 2000. Does not integrate with the audio system. Now deprecated.

## Immersive Monologue

*Optional module – experimental*

Provides fancy ingame messages that can be customized through the inspector but leverage the dialogue system. The design and implementation of this is still WIP.

## SickDev Console Integration

*Optional module*

Integration for [DevConsole 2](https://assetstore.unity.com/packages/tools/gui/devconsole-2-16833) by Cobo Antonio. You will need to import the asset, import this module, and set SickDevConsoleImplementation as your preferred console implementation in CoreParams. I think you need to enable DontDestroyOnLoad and disable auto instantiate in DevConsole options.

## Speed Hacks

*Optional module – experimental*

UI and integration components for “speed hacks” in options that can improve performance on low-end systems by doing dramatic stuff like disabling all dynamic lights.

## Unsplash

*Optional module - deprecated*

Displays an “unsplash” screen after exiting the game. Only used for Beach Defend 2000. Works by invoking another executable which is (or at least should be) included with the module. Now deprecated, though it may be undeprecated someday.

## WindowTitle

*Optional module*

This module can change the window title to something other than the game name. By default it changes this on startup, but this can be changed and the title changed in WindowTitleModule. Currently it only supports Windows Standalone. There’s an attempt at UWP support but it doesn’t work, and I don’t have the first clue on how to do it on other platforms.

It will also use the string “WindowTitle” from the list “IGUI” (using the string substitution system)

## XSMP

*Optional module - experimental*

Provides user music using **X**CVG **S**ystems **M**edia **P**rovider, allowing players to play their own music ingame. Requires the [XSMP backend](https://github.com/XCVG/XSMP), very experimental. It does *work* as a proof of concept, but the UI is shit and it’s buggy. Someday I’ll make this work properly.

# CommonCore Audio Module

//TODO probably won’t get its own section yet, but it will someday

# CommonCore Input Module

The CommonCore Input module (aka MappedInput or CommonCore.Input) abstracts control inputs, allowing controls to be provided by different mapping backends and remapped on the fly (if the mapper supports it). Using the CommonCore input system is as simple as using the MappedInput class instead of UnityEngine.Input, and the API is almost identical.

The MappedInputModule class is not a CommonCore Module, but a StandaloneInputModule that routes mapped input to the UI system in conjunction with MappedInputComponent. The actual CommonCore Module class is InputModule.

## Available input mappers

A few input mappers are provided out of the box:

**Null Input Mapper (NullInputMapper)**

Built in. Placeholder. Always returns 0 or false. Not really useful.

**Unity Input Mapper (UnityInputMapper)**

Built in. Basic input mapper thunking through to UnityEngine.Input APIs, using controls set up in the editor. Not remappable at runtime.

Note that all input mappers currently rely on the Unity Input Manager system being enabled and the Mouse X and Mouse Y axes being set to something sane. This will change someday but not today.

**Explicit KBM Input (ExplicitKBMInputMapper)**

Included as a module. Basic input mapper using built-in Unity APIs to query by saved keycodes, allowing remapping. Loads default input map from Data/ExplicitKBMInput/DefaultControls, and saves configuration to ExplicitKBMInputMap in ConfigState CustomConfigVars\*.

## Configuring available controls

The input module has a concept of available controls. The exact interpretation of these is up to the input mapper, but in general you can expect a valid result to be returned for these and you can expect mappers that allow remapping to present these as remappable to the user. There is also a distinction between axes and buttons; axes are valid for GetAxis, buttons are valid for GetButton, and mappers may present these differently for remapping.

Available controls include all the controls defined in the DefaultControls class, plus those defined by AdditionalAxes and AdditionalButtons in CoreParams. You can inform mappers to hide controls from configuration with HideControlMappings, but they will still be available to query the state of.

Configuring default *mappings* varies from mapper to mapper. For the usual setup with Explicit KBM Input, I usually configure the input map in game and then copy it out of the config file into its own JSON file to set up defaults.

## The MappedInput API

The MappedInput class is set up with GetAxis, GetAxisRaw, GetButton, GetButtonDown, and GetButtonUp methods that work just like UnityEngine.Input. If you’re just throwing together a game

You can use GetDescriptorForAxis and GetDescriptorForButton to get a MappingDescriptor. MappingDescriptor provides a human-friendly description (ie name) of the keys/buttons/etc a control is mapped to, ie for displaying hints on screen.

Note that the MappedInput class thunks through to the InputMapper; you are not accessing the InputMapper directly. If you need to get the actual mapper for whatever reason, use MappedInput.GetCurrentMapper().

## Creating an input mapper

To create an input mapper, derive from the InputMapper class and implement the abstract methods. The input module will automatically pick up your mapper and provide it as an option.

Your Configure() method should create an ephemeral (ie attached to EphemeralRoot) modal that allows the user to set mappings.

You can implement mapping descriptors however you like, but a list of strings and the string substitution system might be a good way to start.

The input mapper object should be considered ephemeral, and you should not do a lot of setup/teardown work in it. Instead, create a Module, and put your processing there.

# CommonCore Campaign Module

The Campaign module defines a “campaign”: flags, quests, and vars for describing the current state of the game. It adds one property to GameState: CampaignState, which is a CampaignModel.

CampaignModel stores Flags, Vars, and Quests, but does not allow the backing stores to be accessed directly. The three types of state information, along with their intents and some information about their access APIs, is listed below.

**Flags**

Flags are boolean; any given flag either exists or does not. This is checked with HasFlag, while a flag can be added with AddFlag, removed with ClearFlag, or the state explicitly set with SetFlag. Methods exist also for retrieving all flags as a string or collection.

**Vars**

Vars store arbitrary data as a key/value store. The existence of a var can be checked with HasVar, while a var can be retrieved with GetVar<T>. The value of the var, if it exists, will be coerced to type T (this is done with TypeUtils.CoerceValue plus some quick but equivalent logic meant to speed things up), otherwise, the default value for T will be returned. SetVar<T> will store whatever is passed in directly, while SetVarEx<T> will attempt to coerce to the type of the existing value if one exists. There are also methods for retrieving all vars as a string or collection (no type coercion is performed).

**Quests**

Quests is effectively a key/value store for integers, plus some (optional) special logic. The terminology used for the value of a quest is *quest stage*. GetQuestStage retrieves the current stage of the quest, while IsQuestStarted, IsQuestActive, and IsQuestFinished return its relative state. A quest is considered started if its stage is not 0, active if its stage is greater than 0, and finished if its stage is less than zero. If a quest does not exist (no key in the backing store, no state ever set) this is considered equivalent to stage being 0. StartQuest and EndQuest send out messages (RpgQuestStarted and RpgQuestEnded) respectively in addition to modifying quest stage. There are also methods for retrieving all quests as either a string or collection.

Quest Definitions are part of RpgGame and are not handled here.

**Other remarks**

Note that Flags, Vars, and Quests are all case-sensitive.

Like the rest of GameState, the CampaignState is persisted across save/load.

The Campaign module also contains console commands for manipulating campaign state (Campaign.\*).

One of the reason for the somewhat odd design (particularly the loose typing of Vars) is because this state stuff is inherited from Katana, which was written in TypeScript/Javascript and mostly conformed to the conventions of code in that language.

# CommonCore World Module

The intent of the World module is to roll up everything needed to underpin a game world with a player and things the player can interact with, but without making assumptions about the specific type of game being created. It includes base classes for Entites and Effects, as well as logic for spawning, save/load, and more.

## WorldUtils

WorldUtils contains many utilities for interacting with entities, effects, players, and actors. This includes but is not limited to methods to determine if something is a player, actor, or effect, and methods to spawn entities and effects.

## WorldConsoleCommands

WorldConsoleCommands contains many console commands for manipulating objects, entities and PlayerFlags.

Note that *Prid* DOES NOT work as it does in Besthesda games. The rough equivalent is *Pick*, which selects an object by TID (object name). *Prid* selects by FID (Form ID) which is the name of the prefab the entity is spawned from. *Prbt* picks entities with a given CommonCore Tag (distinct from Unity tags, see below). The object currently selected is not displayed but you can use GetInfo to print some information on the selected object if one is selected and *Dref* to deselect.

## CommonCore Entities

Entities are game objects that contain a BaseController (or derived) component. Any active object that contains significant amounts of logic or needs to have its state persisted should probably be an Entity. Examples include monsters, the player object, item pickups, and destroyable things.

Entities should be uniquely named if they are pre-placed in a scene; this is necessary for persistence and a few other things to work properly. In some cases you can get away with non-uniquely named entities.

Entities have their own tags, henceforth referred to as CommonCore Tags. An entity can have any number of CommonCore Tags, and this system is completely orthogonal to the Unity tag system.

Entities are not automatically persisted; a RestorableComponent needs to be put on the entity and the entity controller needs to have support for persistence. The entity controllers included with World and RPGGame all support persistence. Note that for persistence to *fully* work, all components on the entity with state must support persistence as well.

Entities placed in Resources/Entities (as prefabs) can be spawned with WorldUtils.SpawnEntity. Most entities should be a prefab in Resources/Entities. **Make sure you set the Editor Form ID to the name of the prefab!**

You can implement IReceiveEntityEvents on a component you make that will be added to an entity in order to make it receive event function calls from the entity controller. You can also implement IHaveRestorableExtraData to save and restore data (see the section on save/load below).

### Entity Prefab Prefixes

These are used in the test project and not a strict convention (except for *spec\_player* which is handled specially, and maybe *spec\_item* as well).

* junk\_ - junk prop items
* npc\_ - npc/monters (Actors)
* phys\_ - physics props
* spec\_ - special purpose objects
  + spec\_player - the player object
  + spec\_item - generic dropped-item worldmodel
* static\_ - static props
* test\_ - test objects
* wm\_ - item world models

Many of these types are only applicable when RPGGame is being used.

## Placeholders

EntityPlaceholder allows “placing” an entity in the world without a direct reference to its prefab. At runtime, it will be spawned by name.

Objects with EntityPlaceholder and anything else that implements IPlaceholderComponent will be ignored by WorldUtils.FindDeepChildIgnorePlaceholders, which is used for restoring entities.

If CoreParams.AlwaysPreactivateEntityPlaceholders is true, entity placeholders will be activated early to spawn entities by WorldSceneController. If it is not set to true, entity placeholders will only be activated early if AutoRestore is enabled. If not activated early, entities are placed when EntityPlaceholder.Start is called by Unity.

## CommonCore Effects

Effects are game objects that contain an EffectScript (or derived) component. They are meant to be used for projectiles, visual, and aural effects; anything of that sort that does not contain significant logic or need to be persisted.

Effects may be preserved (ie not unloaded) across scenes; there is an option in EffectScript for this. This is distinct from scene-local persistence as entities may have. There is no provision for persisting effects in save games or across scene loads.

Effects placed in Resources/Effects (as prefabs) can be spawned with WorldUtils.SpawnEffect.

## Bullets, Hitboxes, etc

The World module lays the groundwork for entities that can hit, be hit, and take damage.

The ITakeDamage interface represents an entity that can take damage, and defines methods for querying health and applying damage. The IAmTargetable interface represents an entity that can be targeted (for example, a monster), though there is no handling in World itself for entities that target other entities (RPGGame has ActorController and friends).

DestroyableThingController is an example of an entity controller that implements both ITakeDamage (it can take damage) and IAmTargetable (it can be targeted).

Hitbox components implement IHitboxComponent. In most cases you should just use HitboxComponent. These define some modifiers that can be picked up by a bullet or other hit, but also implement logic to take damage when an IDamageOnHit implementation hits them.

IDamageOnHit is implemented by BulletScript, which provides a great degree of flexibility for bullets. It provides functionality for fake gravity and raycasting for hits (which works much better than trying to detect collisions for very fast bullets).

Note that in both cases (real hit and raycast hit) the IDamageOnHit implementation (BulletScript) is responsible for determining hit location, hit material etc, spawning the hit puff, and calling TakeDamage on the ITakeDamage entity controller.

## Players and Actors

The World module contains the *concept* of an Actor and the *concept* of a Player, but does not include implementations of either.

An Actor is defined as:

* a CommonCore Entity
* with a CommonCore Tag of Actor

Generally, NPCs and monsters will be Actors.

A Player is defined as

* a game object with the tag Player or named Player

The Player object is handled specially; it will be created by the scene controller if it does not exist, and its object state is not scene-local and is stored separately. By default the player prefab is *spec\_player*. The Player object is nominally an entity (and some things *will break* if it is not).

### Player Spawn Selection

Selection of the player spawn point is handled by WorldUtils.FindDefaultPlayerSpawn and WorldUtils.FindPlayerSpawn.

Spawn points are matched by name; there is provision to select a specific spawn point in DoorInteractableComponent and probably some RPG scripts\*, these create a PlayerSpawnIntent which is then picked up by WorldSceneController when it goes to spawn the player. It is also possible to specify a position and rotation instead of a spawn point. Spawn points are located by TID (object name).

If a player spawn point is specified, the logic in FindPlayerSpawn will be used. This selects from active objects with a matching name containing a PlayerSpawnPoint script, then looks for active objects with a matching name that do not contain a PlayerSpawnPoint script. If neither can be found, it will fall back to default behaviour and print a warning to the console.

Default behaviour if no spawn point is specified uses logic in FindDefaultPlayerSpawn. This will use, in order from highest to lowest precedence:

* Active object called “DefaultPlayerSpawn” with PlayerSpawnPoint script
* Active object called “DefaultPlayerSpawn” without PlayerSpawnPoint script
* Any active object with a PlayerSpawnPoint script

If a spawn point cannot be found, the player will be spawned at world origin.

## Saving and loading

Game state data is stored in the GameState object (GameState.Instance) in CommonCore. When a game is saved, this is serialized, and when a game is loaded, it is deserialized.

You can extend GameState as described in the State Objects section. Serialization and deserialization follows Json.NET rules:

* public properties are serialized and deserialized
* public fields are serialized and deserialized
* properties with a private getter are not serialized
* properties with a private setter are not serialized
* private fields are not serialized or deserialized
* anything with a [JsonProperty] attribute is serialized and deserialized even if it would normally not be
* anything with a [JsonIgnore] attribute is not serialized or deserialized even if it would be

Additional steps are taken to persist scene state. Scene state is *committed* to and *restored* from GameState.



The GameState properties involved in this process are LocalObjectState, MotileObjectState, and PlayerWorldState. These are defined in CommonCoreModules/World/State/GameState.cs. LocalDataState may also be involved.

Provided AutoCommit is enabled on the WorldSceneController, entity state will be saved to GameState. Local data state (LocalState on the SceneController) will also be saved to LocalDataState\*.

\*LocalStore/LocalDataState is handled by BaseSceneController and will be committed and restored even if the specialized WorldSceneController is not used. For any of the entity saving stuff or player spawn voodoo to work, WorldSceneController must be used.

These are scene-local and will be stored by scene name (so make sure your scenes are uniquely named). In other words, local data state for one scene will not be loaded for a different scene.

Provided AutoRestore is enabled on the WorldSceneController and previously saved state exists, entity state will be restored on scene load. If an entity does not yet exist in the scene it will be spawned. Local data state (LocalStore on the SceneController) will also be restored from GameState.

In order for an entity to be committed and restored, it must have a RestorableComponent attached. For nearly all applications, LocalRestorableComponent is the correct one to use. If you are building your own RestorableComponent, BlankRestorableComponent can be derived from, but it is not used in World or RPGGame. The rest are all either abstract base classes or specialized.

The RestorableComponent has two methods: Save and Load. These are called by the WorldSceneController when committing and restoring, respectively.

These work with RestorableData objects, an abstract type (RestorableComponent probably should have been generic, but oh well). If you are using LocalRestorableComponent it will be DynamicRestorableData, which includes entity, transform, and dynamics data.

Components on an entity can commit and restore their own data by implementing IHaveRestorableExtraData. The CommitExtraData and RestoreExtraData methods on the component will be called by BaseController.CommitEntityData and BaseController.RestoreEntityData respectively\*, and will have the data stored in an ExtraData dictionary in the RestorableData. If you implement IHaveRestorableExtraData, modify the dictionary passed in rather than attempting to replace it, and be careful not to clobber other data with your key names. A good convention would be <object name>\_<component type>.

\*Some specialized controllers in RPGGame override these to add their own extra data. It’s dirty but I do it so you can too.

There is some special handling for “motile objects”, ie entities that can move between scenes. MotileObjectState and MotileRestorableComponent exist for this purpose, and Intents were designed with an eye toward it as well. Unfortunately, there is no complete implementation at this time, so you’ll have to do the legwork yourself if you want motile objects to work.

Player state is stored in PlayerWorldState. The player object is restored to DefaultPlayerSpawn unless an intent exists with specific information (see Player Spawn Selection above). The player object has a PlayerRestorableComponent.

## Action Specials

The Action Special system is the oldest part of CommonCore, dating back to (if I recall correctly) 2016, before CommonCore even existed. It was originally created for my (unreleased) Christmas game, and as the name implies is inspired by the Action Special system in Doom and derivatives. It was revised again for Firefighter VR+Touch, and then reworked and added into CommonCore shortly after its inception. In early 2020 it was moved into World; previously it was a separate module but highly interdependent. It still remains in a separate namespace (CommonCore.ObjectActions).

The Action Special system consists of *Triggers* and *Actions*.

Triggers inherit from ActionTrigger and have an ActionSpecialEvent field (Special) which references the Execute method it will trigger. Triggers can be activated by things like player input, an actor entering a trigger, or simply on start.

Specials inherit from ActionSpecial and have an Execute method which takes an ActionInvokerData struct. Specials perform some action, such as opening a door, changing the scene, or playing a sound.

You must handle triggered/repeatable and AllowInvokeWhenDisabled in your inheriting class if you are making new triggers and specials.

Note that a trigger can technically invoke any MonoBehaviour with a public method that takes an ActionInvokerData, as it uses a UnityEvent<ActionInvokerData>. However, there are a few other components (in RPGGame mostly) that require a real ActionSpecial. You can bridge this gap with UnityEventSpecial or ActionSpecialSplitter.

ActionInvokerData contains a few fields, some of which may or may not be populated:

* **Activator** is the entity that was ultimately responsible for the event being triggered. Usually this is the player object, or null.
* **Caller** is the class that was
* **Position, Rotation,** and **Velocity** are only populated in cases where it makes semantic sense to do so, such as BulletScript putting in the position of the hit or OnTriggerEnterTrigger the transform of the crossing entity.
* **ExtraData** can be used for custom or future use. Be careful as the backing dictionary is not copied when ActionInvokerData is!

Doors probably should have been entities, but aren’t.

Some action specials and triggers support persisting their activation state. This is done with LocalDataStore. By default the name of the object the component is attached to is used as the key for persisting; for this reason it should be uniquely named. This may be overridden explicitly (variously called Save Tag or Persist Key).

## PlayerFlags

PlayerFlags can be used to modify aspects of the player’s behaviour, such as freezing them in place or disabling their weapons. Despite the name, it’s largely inspired by ZDoom’s [PlayerProperty](https://zdoom.org/wiki/SetPlayerProperty) system.

The current player flags are retrieved through GameState.PlayerFlags, which is a PlayerFlagsCollection that both contains player flags\* and checks any registered IPlayerFlagsSource. In RPGGame, both DialogueController and PlayerRpgState register an IPlayerFlagsSource, and in World, WorldSceneController registers an IPlayerFlagsSource.

\*You can add player flags to GameState.PlayerFlags directly and this is pretty common in my games. These are persisted when the game is saved and loaded.

The WorldSceneController IPlayerFlagsSource is called TempPlayerFlags and can be used to store player flags temporarily in a scene-local way. They are not persisted in any way; temporary means *temporary*.

SetPlayerFlagsSource is an implementation you can use for your own flags source that is backed by a HashSet.

PlayerFlags are not case-sensitive. No PlayerFlags are defined in World, but some are defined in RPGGame (in RpgMiscTypes.cs).

## Cartographer and maps

The Cartographer component is half of the logic that drives the pause menu maps. It can be attached to the WorldRoot to define world bounds and center and specify a graphic for the map, as well as the positions of map markers that can appear on the map.

Handling the map and markers on the UI is done in MapPanelController in RPGGame. GameState.MapMarkers, which contains the reveal state of the map markers, is also in RPGGame.

## Light reporting

Player light reporting is an experimental feature added in 3.0.0 Preview 3 and improved in subsequent previews. Depending on a configuration setting, it will either calculate based on ambient values or use a camera-and-object setup to “probe” light at the player’s location. The ambient level can be overridden by attaching a ReportedLightOverride to WorldRoot, and light reporting can be forced on CoreParams. This is currently used in RPGGame for lighting sprite weapon viewmodels.

PlayerLightReportingScript contains most of the logic.

This feature is still considered experimental. It is also quite expensive and is only fully enabled (probe mode) at the Ultra quality level.

# CommonCore RPGGame

//this is probably going to require a week on its own

## Dialogue

## Inventory

### Scripts

Scripts can be executed when inventory items are manipulated. These are specified in the *Scripts* property of an inventory item, which is a JSON object. All are optional, as is the *Scripts* node itself. You can only have one of each type of script.

OnAdd

caller: inventory model

args: item model, item instance

return: void

OnRemove

caller: inventory model

args: item model, item instance

return: void

OnEquip (available on weapon and armor items only)

caller: character model

args: item model, item instance

return: void

OnUnequip (available on weapon and armor items only)

caller: character model

args: item model, item instance

return: void

OnApply (aid items only)

caller: character model

args: item model, item instance

return: AidItemScriptResult

OnQuantityChange (stackable items only)

caller: inventory model

args: item model, item instance, old quantity, new quantity

return: void

OnFire (weapon items only)

caller: weapon component

args: item instance, item model

return: void

OnReload (ranged weapon items only)

caller: weapon component

args: item instance, item model

return: void

## Weapons

## Player Object

## Player State

## Actors

## Delayed Events

# How to do things in CommonCore

Basically short application notes on how to accomplish various things in CommonCore.

## Setting up a Scene

## Applying Camera configuration

//TODO describe tackons

## Choosing a SceneController

For projects using the full RPG library, or at least the World module, use:

* **WorldSceneController** for scenes in the game world where there will be a player object and other entities.
* **BlankSceneController** for scenes not in the game world that will not have a player object or other entities (ie cutscenes).

For projects that are not using the full library, you should probably derive your own SceneController from BaseSceneController (make sure to set its script execution order!) though you may make use of BlankSceneController as you see fit as well.

## Changing Scenes

There are a few ways to change scenes in CommonCore:

* SceneManager.LoadScene (from Unity)
  + Changes scene directly. Will call some, but not all, script hooks, and will not go through LoadingScene. Use this only when transitioning scenes outside of a running game (ie going to other menus from main menu)
* SharedUtils.StartGame
  + Changes scene to CoreParams.InitialScene by default using MetaState and LoadingScene with handling for a new game that creates a new GameState. Will call all appropriate script hooks. Use this when starting a new game.
* SharedUtils.ShowGameOver
  + Changes scene to CoreParams.GameOverScene directly. Will call OnGameOver script hook. Will put current scene in MetaState.NextScene. Use this when the player hits a failure condition to show the game over screen.
* SharedUtils.EndGame
  + Changes scene to CoreParams.MainMenuScene by default using MetaState and LoadingScene with handling for ending a game that cleans up GameState. Will call all appropriate script hooks. Use this when leaving a game session.
* SharedUtils.ChangeScene
  + Changes scene using MetaState and LoadingScene. Will call all appropriate script hooks. Use this for changing scenes inside a game (ie most cases).
* SharedUtils.LoadGame
  + Changes scene using MetaState and LoadingScene, loading game data from a save file. Will call all appropriate script hooks. This is used by the save/load system and you probably won’t have to use this directly.
* WorldUtils.ChangeScene (in World module)
  + Changes scene like SharedUtils.ChangeScene, but allows specifying a spawn point either by name or position/rotation. Creates a PlayerSpawnIntent for this. Use this for changing scenes inside a game and specifying a spawn point.
* RpgWorldUtils.ChangeScene (in RPGGame)
  + Changes scene like WorldUtils.ChangeScene, but allows specifying a prop override for the loading screen. Creates a LoadingScreenPropIntent for this. Use this for changing scenes inside a game, specifying a spawn point and loading screen prop. Probably deprecated.

Each of these is appropriate for some situations, and inappropriate for others. The one you’ll use the most is probably SharedUtils.ChangeScene.

## Using Scripts

Scripts, in CommonCore parlance, are small chunks of code that can be called by name. This is handled through the Scripting module. Scripts are used to run arbitrary behaviour in the dialogue system, with inventory items, and more. Scripts, and hooked scripts especially (see below), are also a key part of the CommonCore extensibility story.

To mark a method as a script, add the CCScriptAttribute. Visibility of the method does not matter. By default, the script will be callable as ClassName.MethodName via ScriptingModule.Call. Scripts may return a value, which can be retrieved by using CallForResult instead of Call.

The name and class name can be overridden with the CCScriptAttribute.

Generally, scripts should be static methods, although there is limited provision for resolving a singleton instance or specifying an instance via CallOn/CallOnForResult.

A ScriptExecutionContext struct is passed as the first argument of a script if the type matches and NeverPassExecutionContext is not set. Arguments are passed in order and will be truncated if too many are passed in. Arguments are coerced before passing via TypeUtils.CoerceValue. All arguments passed to the script are available in uncoerced, untruncated form via ScriptExecutionContext.Args.

## Using Scripting Hooks

Script hooks are one of the most powerful and useful features of CommonCore. At various points, the scripting system will call all scripts that are marked to be called at that point. This is done with the CCScriptHookAttribute. Hooked scripts must be marked with both CCScriptAttribute and CCScriptHookAttribute.

See TestModule.ScriptingTests for some examples.

Named Hooks allow the same thing to happen but at points not defined by the base framework. For example, the dialogue system in RPGGame calls a named hook after presenting a dialogue frame. The behaviour when both a hook and named hook is specified for the same script is undefined. Named hooks are orthogonal to normal hooks; ScriptHook.AfterAddonLoaded is not the same as the named hook “AfterAddonLoaded”. To call a named hook, use CallNamedHooked.

## Using the Console

### Writing to the console

Use ConsoleModule.WriteLine to write only to the console, or any of the Debug.Log\* methods to also log to player log.

### Creating console commands

Mark methods that you want to be callable as commands with CommandAttribute (ie [Command]). The method does not need to be public but it must be static. The CommandAttribute has parameters that can alter the name of the command; by default it is just the method name. The alias property can be used to give it an alternate name, className to give it an alternate class name, and useClassName to make it use the class name (overridden or real).

CommandAttribute is in the base CommonCore namespace. Be careful not to use the wrong CommandAttribute such as the one from DevConsole if you are using that asset.

## Sending and receiving messages

Messages are received by IQdmsMessageReceiver implementations registered with QdmsMessageBus. You can implement the interface yourself, or use the premade QdmsMessageInterface if you prefer composition. QdmsMessageComponent is a third option that is a MonoBehaviour Component and uses UnityEvents.

Sending messages is very easy, it can be done through QdmsMessageBus.Instance.PushBroadcast or via a QdmsMessageInterface.

### Using QdmsMessageInterface

The slightly confusingly named QdmsMessageInterface class provides a generic interface to the message interface. It is meant to be attached to a UnityEngine.Object, set as its “Attachment” and it will automatically declare itself as invalid (allowing it to be disconnected from the bus and destroyed) when the attachment ceases to exist. It maintains a message queue which can be manually accessed, and/or can call a method when a message is received by attaching a listener with SubscribeReceiver.

## Using string substitution

## Implementing a StringSubber

## Playing Audio and Music

## Fading in and out

Use ScreenFader.\* and MusicFader.\*

## Creating a Module

Modules are best described as chunks of functionality that extend CommonCore and contribute a part of your project. They are located in their own folders in CommonCoreModules\* and generally contain code but may also contain assets.

\*Modules should be self-contained, even if this means creating Resources, Plugins, or Editor folders as subfolders of your module.

Generally, modules have a class deriving from CCModule at their heart, which is automatically created by CCBase and has many overridable lifecycle functions. This is certainly the case for any code module, although strictly speaking you may not need it depending on what you are doing. The CCModule provides the entry point for your module. It should have a no-argument constructor which will be called to initialize the module, and a Dispose() method which will be called to clean it up. The other virtual methods can be optionally overridden to run code at other points.

Attributes can be added to control module load order:

* **Explicit** means that the module will be loaded very early, if and only if it is specified in CoreParams to be loaded. Otherwise it will not be loaded at all.
* **Early** modules are loaded before undecorated modules.
* **Late** modules are loaded after undecorated modules.

Note that combining any two of these attributes will result in undefined behaviour!

Modules can also by async, since at least 2.0.0 final. Async modules derive from CCAsyncModule. CanLoadSynchronously must be overridden to indicate if the module can be loaded synchronously or can *only* be loaded asynchronously, and LoadAsync must always be overridden. Load can be optionally overridden, and will be called if CanLoadSynchronously is true and CommonCore is not using async startup (say, due to platform limitations). If an async module cannot load synchronously and CommonCore is not starting up asynchronously, an error will result. Load or LoadAsync is called immediately after the module is constructed.

## Playing Videos

You can use the Video Module (CommonCore.Video) to player videos in game. This streamlines the process a bit and allows addons to override and insert their own videos.

The VideoModule class itself (retrievable via CCBase.GetModule) provides methods to check if a video exists and get the path to the video. You can then feed this into the utility methods in VideoUtils to set up a player and begin playback.

Your videos must be placed under StreamingAssets/Video and must have an extension of .mp4 or .webm, with a Unity compatible codec (h.264 is compatible for most platforms). Each video must be uniquely named, case-insensitive.

The video module has not been tested for mobile platforms, UWP, or WebGL. It should work for WebGL, will probably work for UWP and Android, and might not work for iOS.

## Building UIs

//panelcontroller, etc

//probably do this when we implement UI themes

//explain DefaultEventSystem here

## Using Themes

//what is a theme, creating a theme, panel and menu handling and ApplyThemeScript

## Working With Migrations

**Migrations** are pieces of code that are used to alter data during loading to account for changes in the underlying model between versions. Specifically in CommonCore, migrations are used when restoring state objects from JSON files when they have changed between versions.

Even more specifically, a migration is a class that derives from the Migration class. You should never derive from the base Migration class directly, only Migration<T>. The type parameter T is the model class, and it must be a class that implements the IMigratable interface. All this means is that it has a LastMigratedVersion property. Although the interface specifies a Version property and it must implement this in some way, it can also be a VersionInfo or a string that can be parsed as a Version in the actual serialized data.

Migrations are used internally for handling ConfigState and PersistState model changes, and you can add your own migrations for GameState, but you can also make any of your own data model classes that need to be saved/loaded IMigratable and use migrations for them, too.

MigrationsManager is the “hub” of the migration system. It is not a CommonCore module but a specialized manager class that is loaded very early in the startup process. It can be accessed via MigrationsManager.Instance. Its main API is MigrateToLatest, which takes a JObject and returns a JObject that has been upgraded to the latest model (note that it may or may not modify the original object, and the original object may or may not be the one returned).

By convention, migrations are placed in the Migrations folder, but any class that derives from Migration will be considered by the migration system. This includes classes in addon assemblies.

The CommonCore migration system is very simplistic compared to other systems you may be used to. It is unidirectional- it can only migrate up, not down- and this is a fundamental limitation. The selection logic for migrations is rudimentary; it is described a few paragraphs down. It works with JObject (from Newtonsoft.Json.Linq) Currently it is only possible to migrate from an unknown version to the latest version, but the ability to migrate to a specific version may be added in the future.

The version of an IMigratable and compared by the migration system always refers to the application version (ie CoreParams.GameVersion), not framework or engine version or an arbitrary model version. More options may be provided in the future.

There are no convenience methods or declarative API provided to Migrations. Migrations must implement everything themselves, in the Migrate method. *This includes updating LastMigratedVersion*, which must be done for migrations to work properly. Note that while the method must return a JObject, it is allowable to modify the input JObject and return it. All work should be done when Migrate is called; long setup steps should not be performed in the constructor and state should never be stored in the Migration class, as their lifetime is not defined. The other properties declare the input version range of the Migration.

Migrations are chosen based on their range of allowed input versions defined by MinInputVersion, MaxInputVersion, and MigrateMaxVersion. There is no precedence or order to migrations other than this; if two possible migrations can be run, there is no guarantee which will run first. MinInputVersion is inclusive; if the version equals MinInputVersion, this check will pass. MaxInputVersion is normally exclusive; if the version equals MaxInputVersion, this check will not pass. However, if MigrateMaxVersion is true, then it will be treated as inclusive and the same check will pass. Both the min version and max version checks must pass for a migration to run. If either MinInputVersion or MaxInputVersion is null, the lower or upper version respectively will be considered unbounded and the check will always pass.

The same process will be run again, *with previous migration(s) excluded*, until no compatible migrations are left.

There are two exceptions that may be thrown: MigrationFailedException and MigrationIncompleteException. A MigrationFailedException is thrown if an error occurs while migrating (ie the Migrate method of a Migration threw), and contains that exception as its InnerException. A MigrationIncompleteException is thrown if MigrateToLatest is called with allowIncompleteMigration false and the sequence of migrations failed to bring the model up to the current version.

It is possible to set up migrations in a degenerate way that will result in dead-end migrations (though I don’t have a mathematical proof for this). There are two recommended strategies for setting up your migrations that will keep problems from cropping up.

**Single Unified Migration**: One migration that runs for all input versions, either when version is lower than current version or even when it equals the current version. The check for what changes need to be made (either by version or by the change itself) are done in the Migrate method. Recommended for simple cases where model changes are small and infrequent. This is the strategy used for ConfigState and PersistState.

**Step By Step Migration**: Multiple migrations, one for each version (or each version with model changes). Each migration produces a result version equal to the next migration’s minimum input version, and each migration has a maximum input version less than or equal to its output version, such that once a given migration has migrated an object it will not run again. The graph of the migrations will be entirely linear. Recommended for more complex migrations, such as if you need to change your GameState model between versions.

# Miscellanea

## Useful things to know

* F12 is the default screenshot key
* You can preconfigure a lot of stuff in ConfigState, including the screenshot key, resolution mode, max framerate, and whether the FPS counter is on or not
* Check the default quality settings! They’re probably fucked
* You can do a LOT through the console, including stuff I forgot about
* Config subpanels on the config screen are added dynamically via ConfigModule.RegisterConfigPanel. This is done for the custom graphics settings panel as well as the gameplay settings panel.
* Use AsyncUtils.RunWithExceptionHandling instead of async void methods, because Unity will swallow exceptions from the latter.
* You must call StartMusic() after SetMusic() if you want it to actually play. No, I don’t know why I designed it this way either. A new API, PlayMusic(), is available that does both
* Make sure your weapon viewmodel is *actually* on the ViewModel layer.
* Use DefaultEventSystem prefab where you can, it’s slightly weird mostly to handle MappedInput.
* Need to know if you’re in game? GameState.Exists will tell you!
* You can actually make the ingame menu appear and disappear programmatically now (for the longest time this wasn’t a thing)
* Self-testing is behind a config flag: RunTests (add this to CustomConfigFlags in config.json)
* There’s some functionality to teleport players back into the world if they go out of bounds, but I forget where it’s actually implemented

## Useful Console Commands

//TODO talk about how pick/prid/etc works

## Platform Compatibility

In summary:

|  |  |  |
| --- | --- | --- |
| **Platform** | **Compatibility** | **Notes** |
| Windows Mono | Full | Reference/development platform |
| Windows IL2CPP | Partial | No mods or dynamic code. |
| macOS Mono | Full\* | Last macOS release worked, but no longer tested on macOS |
| macOS IL2CPP | Untested | No mods or dynamic code. |
| Linux Mono | Full\* | Last Linux release worked, rarely tested on Linux |
| Linux IL2CPP | Untested | No mods or dynamic code. |
| Android Mono | Untested | Should work well, RAM limit and performance may be a problem |
| Android IL2CPP | Untested | No mods or dynamic code. Probably performance limitations |
| iOS IL2CPP | Untested | No mods or dynamic code. Possible path and other issues. |
| UWP .NET | Not Supported | Deprecated by Unity in 2018.x, removed in 2019.x |
| UWP IL2CPP | Partial | No mods or dynamic code. Some path/permissions weirdness. |
| WebGL IL2CPP | Partial | No mods or dynamic code, async doesn’t work. Slow. |
|  |  |  |

(but keep reading)

CommonCore targets the standalone desktop platform, the .NET 4.x runtime, Mono scripting backend and Unity 2020 LTS at the time of writing (up from Unity 2018 LTS in 2.x and Unity 2019 LTS in 3.x).

Some experimental work has been done on getting CommonCore working on other platforms. KILLERS used a fork of CommonCore Core (probably PTR1 or PTR2) before it was abandoned, and it was running successfully on the WebGL platform using the IL2CPP scripting backend. Much later (March 2020, 2.0.0 Preview 7), Bang Ouch was ported to UWP which required resolving issues with IL2CPP and UWP.

The main change needed to make things work on IL2CPP was to use an AOT compatible version of Json.NET, such as [this one](https://github.com/jilleJr/Newtonsoft.Json-for-Unity). After adding a very conservative link.xml (CommonCore.Core.dll and Assembly-CSharp.dll not stripped) everything worked pretty much as expected.

Of course, mod support is not possible because assemblies can’t be loaded, and there is some use of dynamic code in CommonCore now which has (mostly?) been substituted behind defines. There may also be undiscovered edge cases, probably involving generics or reflection, that will break.

It still didn’t work on UWP because of minor issues with paths and audio. The game attempted to create a screenshot folder in a location it did not have permissions to access and locked up there. This is actually an oversight- failing to create the screenshot folder should not stop the game from loading. The other issue was that audio output defaulted to the Raw channel layout which is not supported on UWP.

CommonCore’s default menu systems also require a mouse or touchscreen to use- they don’t work properly with navigation and don’t work with a controller. This was hacked around to get Bang Ouch playable on Xbox with substitute limited-functionality menus.

Different input mappers will probably be required for different platforms. The existing and planned input mappers will work for standalone platforms, but maybe not others. UWP on Xbox needs a gamepad input mapper that uses the UWP API, mobile platforms need a virtual gamepad input mapper, and HTML5 might require a different mapper too.

This was also hacked around for the Bang Ouch Xbox version by just modifying the default input assignments a bit. It’s still using the UnityInputMapper.

At the time of writing, mainline CommonCore has experimental support for IL2CPP and UWP, limited controller-friendly menus through the experimental Bigscreen module, and no platform-specific controller support.

## Data Paths

CommonCore provides some flexibility in where data is saved, and this differs from platform to platform as well.

For the most part, things will be saved in one of a few paths, which may or may not resolve to the same physical folder

* Application.PersistentDataPath
* CoreParams.PersistentDataPath
* CoreParams.LocalDataPath
* CoreParams.ScreenshotsPath

**Application.PersistentDataPath**

Unity saves some cache and log files in this folder. Intrinsic to Unity and cannot be changed.

On Windows, it resolves to *%UserProfile%\AppData\LocalLow\<Game Publisher>\<Game Name>*

On Linux, it resolves to *$XDG\_CONFIG\_HOME/unity3d/<Game Publisher>/<Game Name>* where *$XDG\_CONFIG\_HOME* is usually *~/.config*

On macOS, it resolves to *~/Library/Application Support/<Game Publisher>/<Game Name>*

On UWP (experimental) it resolves to *%UserProfile%\AppData\Local\Packages\<package name>\LocalState*

**CoreParams.PersistentDataPath**

On Windows, this is controlled by the value of CoreParams.PersistentDataPathWindows:

* UnityDefault: Pass through to Application.PersistentDataPath
* Corrected: *%UserProfile%\AppData\Local\<Game Publisher>\<Game Name>*
* Roaming: *%UserProfile%\AppData\Roaming\<Game Publisher>\<Game Name>*
* Documents: *%UserProfile%\Documents\<Game Publisher>\<Game Name>*
* MyGames: *%UserProfile%\Documents\My Games\<Game Publisher>\<Game Name>*
* SavedGames: *%UserProfile%\Saved Games\<Game Publisher>\<Game Name>*

Note the paths are based on known folders and may vary based on locale.

Why? Because there’s no clear guidance on where save games *should* go, so I’m basically just saying, here, you decide. The default in the repository and the one I use for my projects is *Roaming*

On other platforms (including UWP), it always resolves to Application.PersistentDataPath

**CoreParams.LocalDataPath**

On Windows, this is controlled by the value of CorrectWindowsLocalDataPath. If set to *true*, *%UserProfile%\AppData\Local\<Game Publisher>\<Game Name>\local* will be used. If set to *false*, *%UserProfile%\AppData\LocalLow\<Game Publisher>\<Game Name>\local* will be used.

On other platforms, it resolves to <*Application.PersistentDataPath>/local*

The use of a subfolder is to reduce the possibility of conflicts if CoreParams.PersistentDataPath would other resolve to the exact same path.

**CoreParams.ScreenshotsPath**

This is controlled by the value of CoreParams.UseGlobalScreenshotFolder. If set to *false*, the path will resolve to the *<CoreParams.PersistentDataPath>/screenshot* folder. Otherwise, the folder is platform-dependent, except on UWP.

On Windows, it will resolve to *%UserProfile%\Pictures\Screenshots*

On Linux, it will resolve to *~/Pictures/Screenshots* or *~/Screenshots* depending on your distro and configuration.

On macOS, it will resolve to *~/Pictures/Screenshots*

On UWP, it will always resolve to *%UserProfile%\AppData\Local\Packages\<package name>\LocalState\screenshot* regardless of the value of CoreParams.UseGlobalScreenshotFolder. This will not be the final behavior.

## Dialogue Format

The CommonCore dialogue format is mostly compatible with Project Katana’s dialogue files to the point where the same ones can be used across both engines, although there are some features that are only implemented in one or the other. CommonCore 2.0.0 Preview 6 brought it to near feature parity, with previous versions missing much more. CommonCore 3.0.0 Preview 1 finally implemented timed frame advance and ImageFrame, with only VideoFrame and some visual options not yet supported. Additionally, CommonCore 3.0.0 has been adding features not present in Katana at all.

Some crude documentation hacked together during development should be provided with this.

## The Intent behind Intents

## Built-in Script Hooks

AfterModulesLoaded

AfterAddonsLoaded

BeforeApplicationExit

OnGameStart

OnGameEnd

OnGameLoad

OnSceneTransition

OnSceneLoad

AfterSceneLoad

OnSceneUnload

OnPlayerSpawn

OnGameOver

AfterMainMenuCreate

AfterIGUIMenuCreate

OnIGUIMenuOpen

OnFrameUpdate

OnWorldTimeUpdate

AfterAddonLoaded

OnLoadingSceneOpen

OnConfigPanelOpen

OnConfigPanelRendered

OnEntitySpawn

OnEffectSpawn

BeforeSaveSerialize

BeforeSaveWrite

AfterSaveRead

AfterSaveDeserialize

### Named Hooks

**Provided by RPG/Dialogue**

* DialogueOnOpen
* DialogueOnPresent
* DialogueOnClose

## Default String Substitution Lists

**Defined in Core/Data/Strings/BaseStrings.json (may be redefined elsewhere)**

IGUI General GUI strings

IGUI\_SAVE Save/load related GUI strings

CFG Config-related GUI strings

CFG\_MAPPERS Nice names of input mappers

CFG\_MAPPINGS Nice names of input mappings (buttons/axes)

CFG\_KEYNAME Nice names for individual keys

TEST Test strings, currently unused

**Defined in Data/Strings/ExplicitKBMInput.json (provided with Explicit KBM Input)**

EXPLICITKBMINPUT Generic module-related strings

EXPLICITKBMINPUT\_MOUSEAXIS Names of the mouse axes

**Defined in Game/Data/Strings/UIStrings.json (provided with RPGGame)**

IGUI\_RPG RPG-related GUI strings

IGUI\_DIALOGUE Strings for the dialogue system

RPG\_AV Names for actor values (skills/stats/etc)

RPG\_AV\_DESCRIPTION Descriptions for actor values

RPG\_MESSAGE RPG-related messages to be displayed

**Defined in Game/Data/Strings/PronounStrings.json (provided with RPGGame)**

ALIAS\_NOGENDER It/its pronoun strings

ALIAS\_FEMALE She/her pronoun strings

ALIAS\_MALE He/him pronoun strings

ALIAS\_NEUTRAL They/them pronoun strings