CommonCore

Reference Documentation

Last Updated 2020-11-02

Contents

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
- Have extensibility where it is practical to do so

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.

3.x *Citadel*

Project Organization

There are two repositories that constantly participate in the CommonCore development process. One is the public commoncore repository 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. 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, and at some point I will probably put up an "experimental" repository for ~~broken~~ crazier stuff as well.

Games that use CommonCore

All of mine since 2019, basically

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

STARFURY – XCVG Systems – February 2019 – Public Test Release 3 (2.0.0 Preview 1) Core

Beach Defend 2000 - XCVG Systems – August 2019 - 2.0.0 Preview 3 Core

~~**"whistler"**~~ – XCVG Systems – February 2020 – 2.0.0 Preview 7 Full

bang ouch – 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 – XCVG Systems – April 2020 – 18f00d1 Full

Heavy Metal Slug – XCVG Systems – April 2020 - 5ad60c4 Core

Shattered – Why Not Me – XCVG Systems – April 2020 - 3cf1b94 Full

RiftBreak (partially) – XCVG Systems – June 2020 – 2.0.0 Preview 11

In The Middle Of The Night – XCVG Systems – Aug-Sept 2020 – 2.0.0 Preview 14

"vail" – XCVG Systems – Holiday 2020 – 2.0.0 Final

"Mother Earth" – XCVG Systems – 2021 – 3.x

"Reality" – XCVG Systems – 2021 – 3.x

Ascension III – XCVG Systems – TBD – 3.x

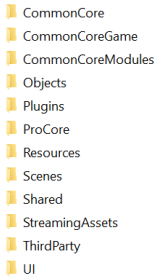
strikethrough indicates abandoned or unreleased projects

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
- **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

CommonCore

Contains CommonCore Core files

//TODO explain the subfolder organization at some point, but it's not critical enough to do now

CommonCoreGame

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

//TODO explain the subfolder organization at some point, but it's not critical enough to do now

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, the last version that doesn't require System.Memory.

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
- User
- Video
- 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.

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

DynamicTexture is currently unused but will be used for dynamically usable textures like DynamicMusic and DynamicSound. **Video** is in a similar boat.

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 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.

- 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 contains assets and scripts for this game's UI

- Fonts
- Graphics
- Scripts
- Sounds

These subfolders should be self-explanatory.

This folder is kind of in limbo right now, not officially deprecated but not used so much anymore. My intent is to move everything into Resources and Shared- things have been slowly migrating out of UI for a while now- but I haven't got around to it yet.

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.
- 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 is very much a WIP
- 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, still the default for editor
    - 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.

☑ CommonCore/Scenes/InitScene	0
☑ CommonCore/Scenes/MainMenuScene	1
☑ CommonCore/Scenes/LoadingScene	2
☑ CommonCore/Scenes/GameOverScene	3

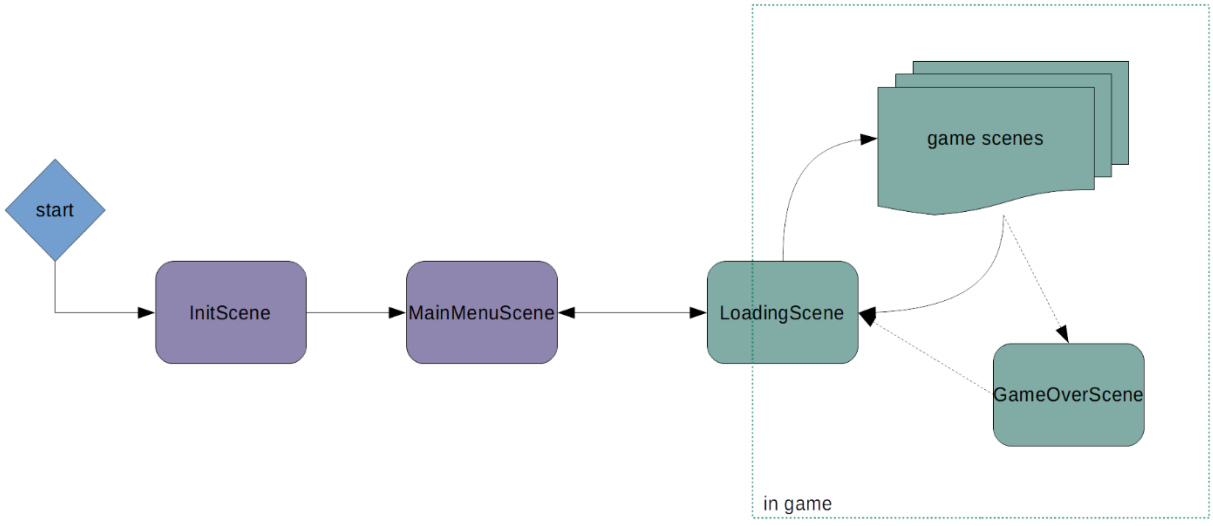
The standard way to customize these scenes is to duplicate the scene into the Scenes folder, making sure it has the same name, and make edits to that. Remove the old scene from the build and add yours, as long as the name is the same it should work perfectly.

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

☑ CommonCore/Scenes/InitScene	0
☑ CommonCore/Scenes/MainMenuScene	1
☑ CommonCore/Scenes/LoadingScene	2
☐ CommonCore/Scenes/GameOverScene	
☑ Scenes/Meta/GameOverScene	3
☑ CommonCore/Scenes/ExampleScene	4
☑ Scenes/Other/TestScene	5
☑ CommonCoreGame/Scenes/DialogueScene	6
☑ Scenes/Other/SoundTestScene	7
☑ Scenes/Meta/DemoWinScene	8
☑ Scenes/World/World_Ext_TestIsland	9
☑ Scenes/World/World_Ext_MichaelCity	10
☑ Scenes/World/World_Int_TestIsland_House1	11
☑ Scenes/World/World_Int_TestIsland_MechanicShack	12
☑ Scenes/World/World_Int_TestIsland_House2	13
☑ Scenes/World/World_Ext_Frangis_Arena	14
☑ CommonCoreModules/Bigscreen/Scenes/BigscreenMainMenuScene	15

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 supposed to be a loading screen displayed when CommonCore is loading but this is only half-implemented at the time of writing. Once CommonCore has loaded a script in InitScene automatically loads the main menu.

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, and I think it's been broken for like a year. 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.

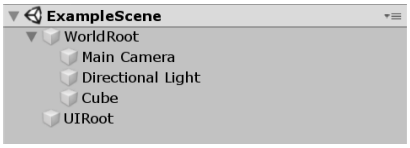
The one significant exception to this is when going to the Game Over screen via SharedUtils.ShowGameOver. This simply loads the GameOverScene, setting the current scene as MetaState.NextScene so we can return to it later if configured to do so. Once the "end game" option is chosen we end the game properly,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

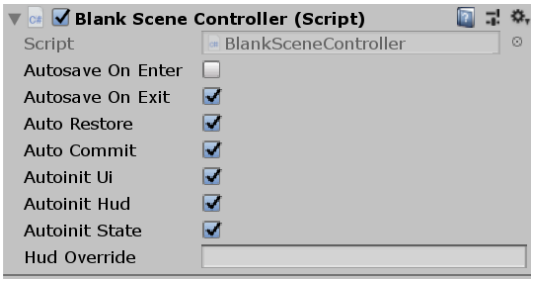
There is also a common structure to CommonCore scenes. At a minimum, they should have a valid WorldRoot object with a valid SceneController and all world/in-game objects below it. A valid UIRoot object



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.

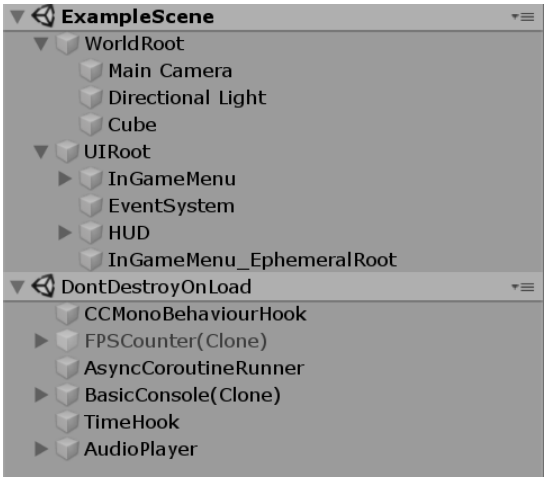
A SceneController in the CommonCore sense derives from BaseSceneController and provides lifecycle handling and save/load for the scene as well as calling script hooks. It also provides a LocalStore key/value store that is committed and restored with scene state. Subclasses, such as the WorldSceneController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

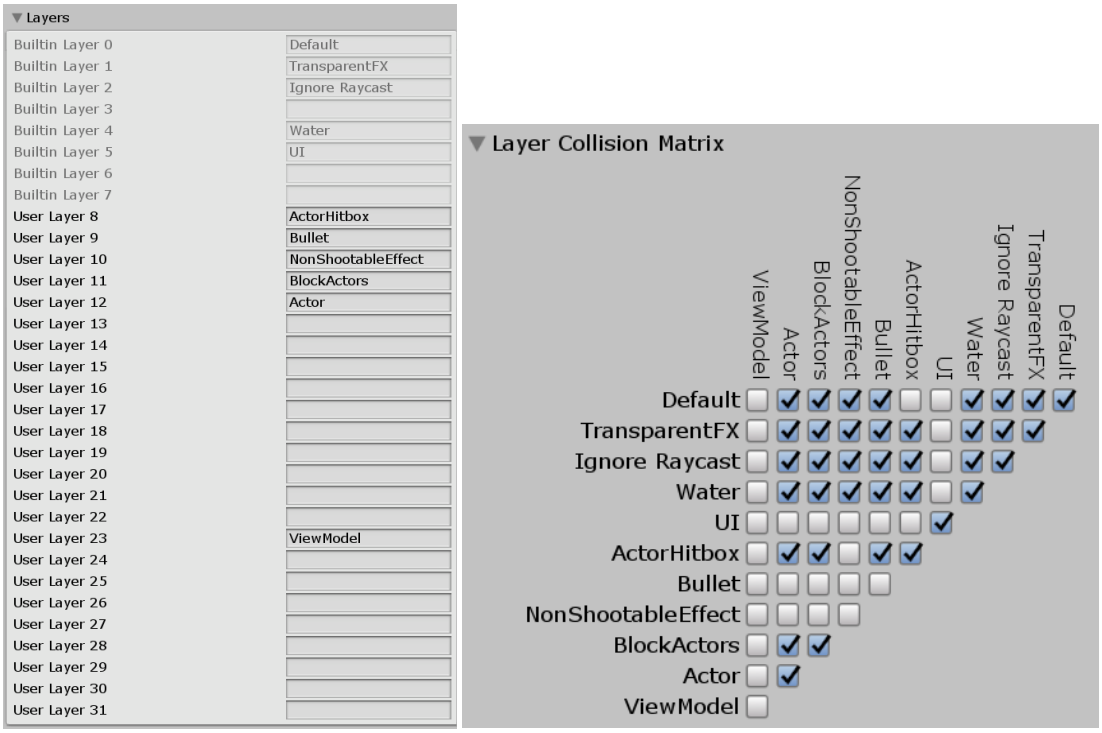
Tags are set up like this

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 basically isn't used for anything.

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



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

This collision matrix will likely be revised soon.

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.

On a side note, does anyone else think it's fucking weird that layers are used for both physics and rendering despite them having little to do with each other?

Sorting Layers are not used, however, magic 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:

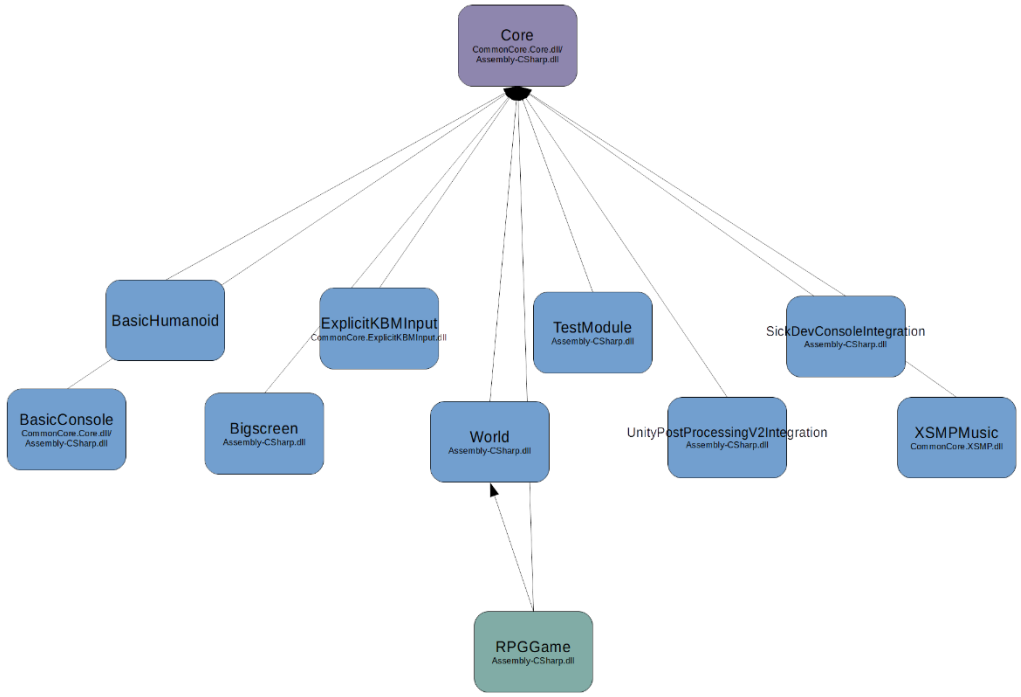
- Core
    - Contained in CommonCore folder
    - May not depend on anything except Unity and shared libraries
    - Resources in Resources/Core, always loaded first
- Modules
    - Contained in CommonCoreModules subfolder
    - Depends on Core, may depend on each other or on Game package
    - Resources 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.

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.
- **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.

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.

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.

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.

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.

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 (WIP, don't use this yet)

Convenience methods in Core

SkippableWait.WaitForSeconds(Realtime)

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

SkippableWait.Delay(Scaled/Realtime)

Similar to WaitForSeconds but async Task instead of Coroutine/IEnumerator

Modal.Push*Modal(Async)

Pushes a modal popup window to the screen. Available in callback and async variants

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

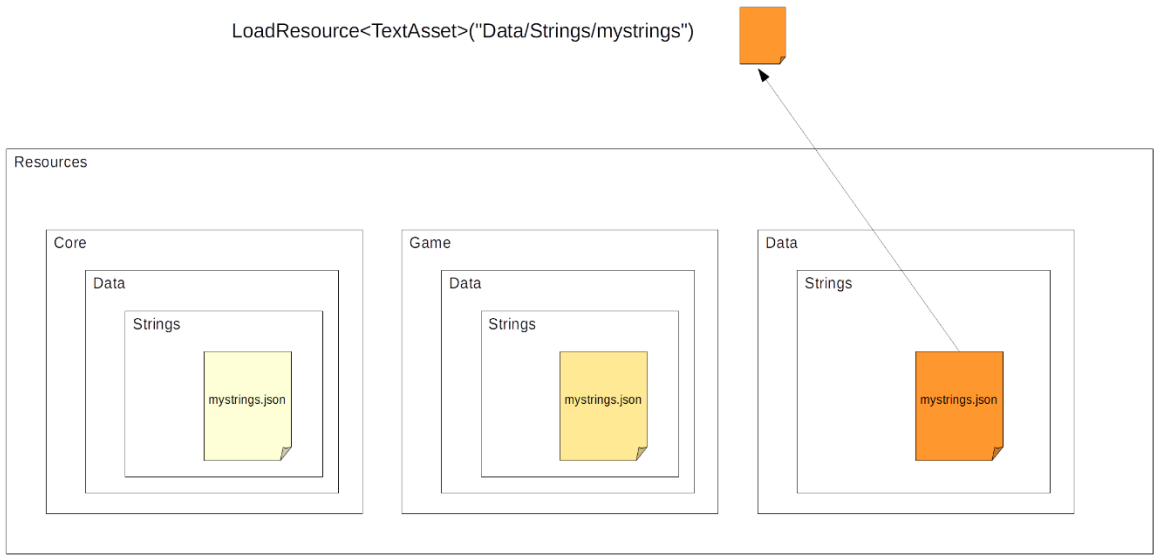
MusicFader.Fade(To/In/Out)
MusicFader.ClearFade
MusicFader.ClearAllFades

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

Resource Management

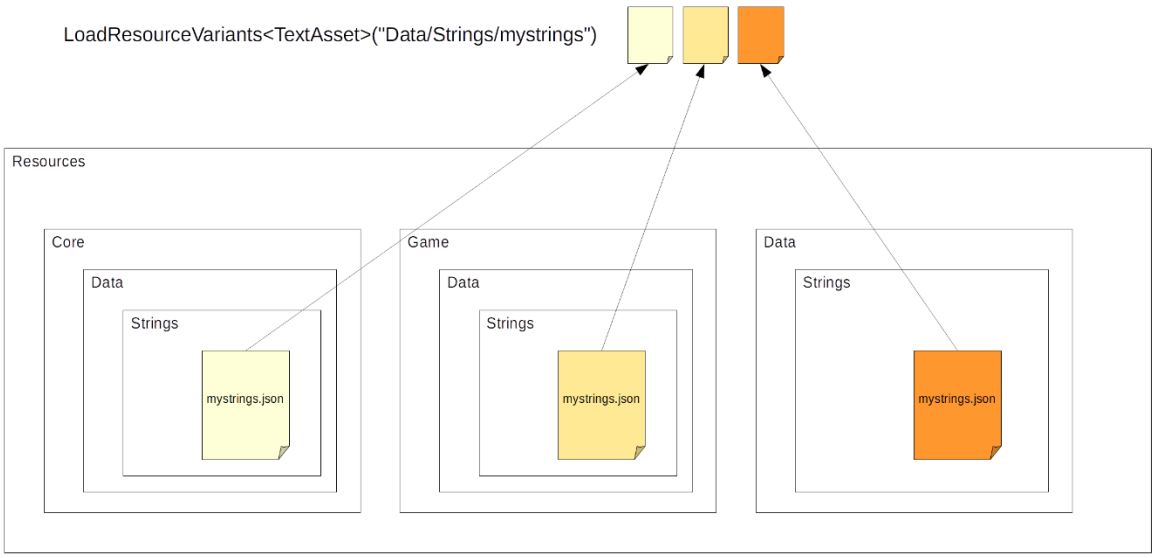
CommonCore has a concept of Resources which is related to (and based on) the Unity Resources system, but with some differences. Like Unity's Resources, resources are accessed by path and type, and it has a simple synchronous API. However, paths are virtual and resources may have multiple variants.

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 will be possible (when implemented), but not recommended**.
- Addons
    - **Do not use.** Addon elocal resources will be mounted in folders here.
    - **Retrieve normally**.
    - **Runtime overrides will be possible (when implemented), but not recommended**.
- Streaming
    - **Do not use.** StreamingAssets 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.

Work is currently underway on a new ResourceManager which will support additional resource variants, resources backed by runtime assets or files instead of Unity Resources, and path redirection. This is to support addon functionality in the future rather than for core functionality. As of Preview 16 it is now enabled by default though still kind of experimental.

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.

TODO more on how the ResourceManager actually works, probably in an appendix

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

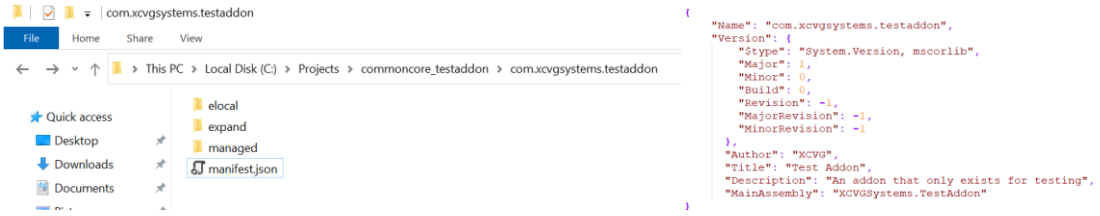
If all this is set up, addon support will be enabled and resources will be loaded from StreamingAssets automatically. However, to actually load addons, LoadAddons must be enabled in the game's config file and the package names added to AddonsToLoad. The order of the package names in AddonsToLoad defines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

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 any assemblies, 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

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 should we give Input its own section?

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 (yes, there are seriously three ways of doing things…) 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 or AllowMenu.

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 so those will 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 will eventually handle UI themes if we ever get that implemented.

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?

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. Is somewhat WIP and is part of an effort to move the dialogue system out of RPGGame.

Explicit KBM Input

Bundled module

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

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.

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 - experimental

Provides playback of CD audio in an incredibly hacky manner. Only used for Beach Defend 2000.

Does not integrate with the audio system. I may implement this properly some day, or not because nobody has a CD drive anymore anyway.

SickDev Console Integration

Optional module

Integration for DevConsole 2 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.

I really do recommend this third-party console if you can spare the cash or already have it, it's a lot better than BasicConsole.

Unsplash

Optional module

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.

WindowTitle

Optional module – experimental

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 can also be configured to 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, 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 World Module

Overview

CommonCore Entities

Bullets, Hitboxes, etc

//TODO damage handling, ITakeDamage and IAmTargetable (?)

World utilities

The concept of a Player

Saving and loading

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.

Cartographer and maps (?)

CommonCore RPGGame

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

Dialogue

Inventory

Saving and Loading

Weapons

Delayed Events

How to do things in CommonCore

//we will likely not get to this section this week

Fading in and out

Using Scripting Hooks

Script hooks are one of the most powerful and useful features of CommonCore.

Using the Console

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.

//TODO explain MessageInterface

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

Using string substitution

Implementing a StringSubber

Playing Audio and Music

Fading in and out

Using Config

Using GameState

Setting up a Scene

Applying Camera configuration

//TODO describe tackons

Creating a Module

//TODO describe structure and organization

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.

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!

Creating an InputMapper

Building UIs

//panelcontroller, etc

//probably do this when we implement UI themes

Using Themes

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

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.
- 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)

Useful Console Commands

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	Untested	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, broken async Task. Slow.

(but keep reading)

CommonCore targets the standalone desktop platform, the .NET 4.x runtime, Mono scripting backend and Unity 2018 LTS at the time of writing. This is what Ascension III is targeting, and covers most of the projects I want to do.

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 is to use an AOT compatible version of Json.NET, such as this one. 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 will need to be substituted. 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 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. A new InputModule for the EventSystem that actually uses MappedInput also needs to be written at some point.

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 shitty UnityInputMapper.

At the time of writing, mainline CommonCore has experimental support for IL2CPP and UWP, limited controller-friendly menus through the 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>*

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 version missing much more.

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

//TODO write some better documentation

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

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