**Impulse Framework**

This barebones framework is designed to help you expedite the creation of your next idea in Unity. It includes a number of highly customizable prebuilt systems such as a scene manager, mobile-optimized main menu, cameras, and more.

**Note about licensing**: Everything in this framework is licensed under [LGPL](http://www.gnu.org/licenses/lgpl.html) and can be used for almost any purpose (including commercial). The single exception is the *Impulse* logo, located in “Assets/Sprites/Not\_Available\_For\_Commercial\_Use”. The *Impulse* logo is proprietary and may *not* be used for any purposes, commercial or non-commercial.

**General Setup**

If you choose not to use prebuilt managers that come with the framework, feel free to skip this section. However, if you do not already have a solution in mind for switching scenes, setting up the main menu, or handling music, it is highly recommended that you configure the general setup.

In the Build Settings, set the Splash scene to 0 and Menu to 1. Unity preloads everything in each scene, with the exception of the first scene (scene 0). For optimal performance, you should keep your splash scene as lightweight as possible and try not to add too many objects.

**Set a Custom Splash Image**

Most games have a splash image or studio logo shown before the game begins. This framework can be set up to display a custom splash image before loading the main menu.

A video demonstration of setting the splash image, setting up scene loading, and customizing the menu can be found here: <https://www.youtube.com/watch?v=btNqHCoRwB8&list=PLLXw4Fw6qNw5WVLPn1hhJNEcwXjxt3b9j&index=1>

1. Locate the Resources/Prefabs/Scene/SplashFadeIn object in the project files (not in the hierarchy!). Select the ImageToFade child object.
2. Set the Source Image of the Image component to whatever splash image you want to display.

**Set Scene Loading Parameters (fade in/out, interpolation, duration)**

The framework provides a scene manager to handle transitioning between scenes, such as fading in/out and transition delay.

Refer to the video in the section above for a video walkthrough of the scene manager.

1. Locate the Resources/Prefabs/Scene/SceneManager object.
2. In the SceneManager component, you can specify the Duration of fade in/out as well as the Interpolation of the fade. If you do not want to fade in/out scenes, set the duration to 0.

**Customize the Main Menu**

The framework provides a customizable main menu that is contained within a single scene in order to remain mobile-optimized.

Refer to the video in the section above for a video walkthrough of the main menu.

1. Open \_Scenes/Menu.unity
2. Open the MenuSystem object. You’ll notice a main menu and options menu are already set up for you, but are inactive.
3. Create a new child object under MenuSystem and attach the MenuScreen script to it.
4. Add your new menu elements to this new child object.
5. Set your new child object as inactive once you are finished with it.

To switch menus using UGUI OnClick(), call the MenuManager.ChangeMenuAndFade() or MenuManager.ChangeMenu() function.

To run one of the examples in the GameExamples folder, replace Menu and the sample Level01 in the build settings with the specific menu and game scene from the example game’s \_Scenes folder.

**BaseGameController (State Machine)**

BaseGameController.cs is a state machine for managing gameplay (not menus or non-gameplay sections). States inherit from a base State.cs class and are managed as objects that can be transitioned in and out of using ChangeState().

**Usage**

Since your specific game will use its own GameController deriving from BaseGameController, it’s best to create an ExtendedState for your game that inherits from State. You can put properties specific to your game in this ExtendedState. For example, here is a snippet of ExtendedState from the IsometricShooter3D example project:

namespace IsometricShooter3D

{

public class ExtendedState : State

{

// Ref to our game controller.

protected GameController controller;

void Awake()

{

controller = GetComponent<GameController>();

}

}

}

Since the IsometricShooter3D example project uses a custom game controller inheriting from BaseGameController, we grab a reference to it in our ExtendedState (all states in the example project inherit from this state).

**Transitions**

You can transition to another state by calling ChangeState<INSERT STATE NAME HERE>()

It is important to note you cannot change state within the Enter() or Exit() methods of an existing state since you cannot change state during the middle of a state transition. In some cases it is necessary to use a coroutine to change state to allow an Enter() or Exit() method to finish. This is especially true in states where gameplay setup is done:

public override void Enter()

{

base.Enter();

StartCoroutine(Init());

}

IEnumerator Init()

{

// Initial setup here.

controller.ModifyEnemyCount(0);

yield return null;

// Change to next state here.

ChangeState<GameplayState>();

}

We put the transition after the yield statement to ensure setup completes before transitioning to the next state.

For a deeper look at how the game controller in the framework is used, check out the IsometricShooter3D project.

**Subscribing To and Posting Notifications (Event Messaging System)**

The event messaging system can be found in “Assets/Scripts/Utility/Notifications”. However, you do not need to put any scripts from that folder onto your gameobjects to use the notification system.

POSTING A NOTIFICATION

Begin by creating a const string to represent the notification you will post. In this example, we are creating an OnClick() function for when the user clicks on a cell in a menu. The script is called ProgramCellSelect.cs and this is our const string:

// The notification sent when this tile is selected.

public const string ProgramCellSelectNotification = "ProgramCell.SelectNotification";

The function we want to call for the OnClick() event is simply:

public void OnProgramCellSelectClick()

{

this.PostNotification(ProgramCellSelectNotification);

}

The PostNotification() function is all you need to do to call to post a notification.

SUBSCRIBING TO A NOTIFICATION

To make something happen when a notification is posted, you need to subscribe to that notification. Subscribers can be other game objects or states within a state machine waiting for an event to happen.

In our example, we want to get which menu cell was clicked. The menu is only shown during a ShowMenu state, so we begin by adding subscriptions to the notifications that enable while the state is active and disable when the state is not:

void OnEnable()

{

this.AddObserver(OnInventoryCellSelectNotification, ProgramCellSelect.ProgramCellSelectNotification);

}

void OnDisable()

{

this.RemoveObserver(OnInventoryCellSelectNotification, ProgramCellSelect.ProgramCellSelectNotification);

}

The first argument is the function we call when we receive the notification (see below) and the second is the notification we are listening for.

Now that we are subscribed to the notification, we can write the code to act upon that notification:

void OnInventoryCellSelectNotification(object sender, object args)

{

// Get selected cell.

ProgramCellSelect selectedProgramCell = sender as ProgramCellSelect;

}

Note that your notification function needs to have ‘sender’ and ‘args’ arguments. ‘sender’ is the object posting the notification. We use this to determine which menu cell was clicked by the user. ‘args’ is additional data that might be included with the notification.

**Utility**

**Playing Music and Managing Playlists**

The music manager and music playlist system allow for easy playback and organization of background music within scenes.

For a video demonstration of the music manager and music playlists: <https://www.youtube.com/watch?v=jQGTqGalGVw&index=2&list=PLLXw4Fw6qNw5WVLPn1hhJNEcwXjxt3b9j>

1. Drag the MusicManager prefab from Assets/Prefabs/Music/MusicManager into your splash scene, or whichever scene is the first one in your build settings. The MusicManager is persistent from scene to scene, so you do not need to instantiate it in each scene.
2. In each scene where you want music to be played, create a new empty game object and attach the MusicPlaylist.cs script. This script can be found in Assets/Scripts/Music/MusicPlaylist.cs. I recommend naming the game object ‘MusicPlaylist’. Then, just populate the Music List array in the game object with song files. Leave ‘Activate On Awake’ to true if you want the playlist to begin playing as soon as the scene is loaded.

**Object Pooling**

The built-in pooling system creates a separate pool for each prefab. This pool is contained at the prefab level so pools survive scene transitions and editor reloads.

Each pooled object should inherit from the *PooledObject* class.

To get a reference to a pooled object:

var obj = prefabReference.GetPooledInstance<InsertTypeHere>();

You can then set the object’s transform position and other properties so it shows up in the game world. If you want to have multiple objects be spawned at the start of a scene or some other time, it’s recommended that you make a spawner object specifically for that purpose.

**Cameras**

**Top-Down Camera**

This camera is best suited for 2D games.

For a video demonstration of the top-down camera: <https://www.youtube.com/watch?v=DLTyrbMxytA&list=PLLXw4Fw6qNw5WVLPn1hhJNEcwXjxt3b9j&index=3>

1. Locate the script in Assets/Scripts/Camera/TopDownFollow\_Camera.cs.
2. Attach this script to a camera object in your scene.
3. Drag a Transform into the Follow Target parameter. This is the object the camera will try to follow.
4. Set the Target Offset and Move Speed parameters to your liking. Target Offset is x,y,z distance from the follow target (the camera position offset relative to the follow target object). Move speed is how fast the camera moves when the object moves.

**Third Person Camera**

This camera is based on the camera used in many popular MMORPG games and automatically zooms in when the follow target is obstructed by an object.

For a video demonstration of the third person camera: <https://www.youtube.com/watch?v=DDdnLPPZXLg&index=4&list=PLLXw4Fw6qNw5WVLPn1hhJNEcwXjxt3b9j>

1. Locate the script in Assets/Scripts/Camera/Third\_Person\_Camera.cs
2. Attach this script to a camera in your scene.
3. Create an empty game object and rename it to ‘LookAt’. This is the object the camera will focus on and follow.
4. Make the LookAt object a child object of the gameobject you want to follow.
5. Assign the LookAt object in the Target Look Transform parameter of the Third Person Camera component on the camera.
6. To add mouse controls such as zoom-in with the mouse scrollwheel, attach the Third\_Person\_Mouse\_Input.cs script to the camera. This script is located in the Assets/Scripts/Camera folder.

**User Interface**

The framework includes an InterfaceManager that works similar to the Menu System. You can find the Interface Manager prefab in the “Assets/Prefabs/UI” folder.

The Interface Manager works with game objects containing an Interface Screen component. Each screen can be switched to by calling InterfaceManager.ChangeScreen(InterfaceScreen screen).

The Interface Manager can also load scenes by calling InterfaceManager.LoadScene(string sceneName) or InterfaceManager.LoadSceneFadeIn(string sceneName). This allows you to create a Pause menu with options such as a ‘Return to Main Menu’ button.

**AI**

Most AI scripts in the framework are based around a Faction component that specifies what faction a gameobject belongs to. For a gameobject to be used with the AI scripts, it must have the Faction.cs script attached along with a faction specified (factions can be neutral in addition to friendly or hostile).

Faction.cs is located in the Assets/Scripts/AI folder.

**Waypoints**

The waypoint system provides an easy way of generating a connected path of points, with the option to ensure it is a closed loop. The waypoints system does not use the Faction system, it simply creates waypoints that any object can follow.

1. Locate the WaypointPathManager.cs script in Assets/Scripts/Utility/Waypoints
2. Create an empty game object in your scene and attach WaypointPathManager.cs
3. Create any number of empty game objects and place them throughout your scene. Make them a child object of the transform with the WaypointPathManager component. These empty game objects are the ‘waypoints’ in the path.

The WaypointPathManager loops through each child object in its transform and generates a path through them.

Public methods for using waypoints (located in WaypointPathManager.cs):

* **public int FindNearestWaypoint (Vector3 fromPos, float maxRange)** – Returns the integer index of the nearest waypoint from the supplied position and within the supplied maximum range.
* **public int FindNearestWaypoint (Vector3 fromPos, Transform exceptThis, float maxRange)** – The same as the above, except a waypoint transform can be passed in to ensure the nearest waypoint is not the waypoint an object is currently at.
* **public int GetNextWaypoint (int index, bool reverse)** – Gets the next waypoint in the path based on the supplied index. If *reverse* is true, then it assumes the path is going backward (e.g. point 0 is next after point 1).
* **public Transform GetWaypoint (int index)** – Returns the transform of the waypoint at the given index.
* **public int GetTotal ()** – Returns the total number of waypoints in the path (number of child objects under the WaypointPathManager object).
* **public bool ReachedEndOfPath (int index)** – Returns true if the waypoint at the given index is the last waypoint in the path or the first waypoint in the path. This is useful for switching the waypoint traversal of an object if you want it to go back and forth from one end of the path to the other.

**Viewcones**

These are procedurally generated cones that can be used to give a gameobject the ability to ‘see’ other gameobjects.

*TODO: Change viewcones from using tags to factions instead.*

**Sphere Detector**

The sphere detector projects an invisible sphere around an object. The idea is other objects within this sphere are ‘detected’ by the object, similar to radar. This system does not actually involve AI behavior, but can be useful in setting one up.

For a video demonstration of the sphere detector: <https://www.youtube.com/watch?v=1ZLkDv9OUNc&list=PLLXw4Fw6qNw5WVLPn1hhJNEcwXjxt3b9j&index=7>

1. Locate the script in Assets/Scripts/AI/Detector.cs
2. Attach the Detector script to a gameobject.
3. In the Detector component, assign allied and enemy factions. Objects belonging to a faction that is not assigned will show up under the ‘Detected Neutral’ array during runtime.

The Detector component includes useful methods for fetching data during runtime:

*TODO: Insert methods here.*

**Game Examples**

There are a number of example games provided with the framework as a showcase of how it is used and to provide a groundwork for building your own project of the same genre.

**Space Shooter 2D**

The 2D space shooter example provides a skirmish scene where the player can maneuver and shoot a ship against enemy ships. The example game uses the model-view-controller (MVC) pattern for spaceships, detailed as follows:

* Each ship has a model class to control its variables and move its transform, a controller class to take user input and feed them to the model and view, and a view class to manage the animations of the ship.
* Each ship’s model, controller, and view class derive from a generic model, controller, and view class. These are prefixed with the word Generic, for example the generic model is called GenericShipModel.
* There are two generic controllers:
  + GenericShipPlayerController – Takes input from the keyboard to move the ship, and input from the mouse to rotate and fire the ship.
  + GenericShipAIController – Makes the ship move and attack ships of an enemy faction.

**GenericShipModel**

This takes up to two Fire Projectile components. The Fire Projectile component should be on the same object as your ship model. It is used to shoot projectiles from the ship.

Some variables in GenericShipModel are used only if there is an AI controller:

* protected float chaseRotation – How quickly the ship rotates toward the target during chasing state.
* protected float strafeMaxAcceleration – How quickly the ship strafes the target during chasing state.

It assumes several scripts are on the object:

* Faction.cs – This assigns the ship’s faction and is used primarily with AI.
* SphereDetector.cs – Used primarily with AI to detect which ships are in close range. Read more about it in the [Sphere Detector](#SphereDetector) section.
* Sphere Collider – Required by the Sphere Detector.
* Rigidbody – Required by the Sphere Detector.

**GenericShipAIController**

The AI controller makes the ship patrol automatically if no WaypointPathManager is assigned. It uses the Faction component to determine if an enemy ship is within range, and if so, then it begins chasing and attacking that enemy.

Variables shown in inspector:

* **private float behaviorChangeRate** – How often the AI updates. The smaller the number, the bigger the performance impact per AI ship.
* **private WaypointPathManager waypointManager** – If this is assigned, the AI will follow along preset waypoints instead of automatically patrolling.
* **private float forwardPatrolDistance** – If no Waypoint Path Manager is assigned, the AI will patrol automatically. It does so by setting a random forward value between 0 and the forward patrol distance, moving forward by that amount, then turning left a random angle between 0 and 90, then creating a new random forward point to move to. This repeats endlessly until an enemy is within detection range.
* **private float chaseLockDuration** – When the AI controller detects an enemy through the sphere detector component, it locks onto that enemy and chases it for this duration, in seconds, before trying to acquire a new target based on which enemy is closest. This prevents constant switching between enemy targets if the closest enemy variable constantly switches.
* **private bool strafeAroundTarget** – If an enemy is within range and locked as an attack target, setting this to true will make the AI ship strafe around the attack target instead of trying to move to the target’s position. It is highly recommended to set this to true.
* **private float strafeDistanceFromTarget** – If strafeAroundTarget is true, this is the minimum distance the AI ship will get to the enemy target when strafing. Note that position calculation is dependent on the behaviorChangeRate, so collisions with the attack target can still occur.
* **private float distanceFromTargetVariance** – How much variation in distance the AI ship will have when strafing the target.
* **private float varianceInterval** – How often the distanceFromTargetVariance is calculated.

**GenericShipView**

The View is used to animate the ship. The View assumes an Animator is attached to the object. Animations should be assigned using Mechanim and called as needed in the specific View component for that ship.

The View assumes certain parameters are set, such as Forward, Turn Left, Turn Right, and Reverse. It assumes there are animations for forward/reverse movement and left/right rotation specified in Mechanim. The game scene provided does not contain these, so warnings will be logged in the console when you run it.

**Fire Projectile**

Used to fire a single projectile prefab from any number of spawn positions specified as ‘gun mounts’. By default, the GenericShipPlayerController listens for mouse button presses to fire projectiles. This component also takes a Projectiles Parent parameter that lets you set a parent transform for all projectiles.

**Isometric Shooter 3D**

This is a fixed-camera top-down third person shooter where waves of enemies spawn on random tiles on the square tile map (or sometimes on the same tile as the player if he or she is ‘camping’).

The game is managed by a GameController object within the IsometricShooter3D namespace. The GameController enables the InitState in Start() to set up initial values like the tracked number of enemies and to generate the map using the SquareTileMapGenerator. Once setup is complete, the game moves to the GameplayState where the player is repositioned to (0, 1, 0), the gameplay UI is shown by calling the Interface Manager, and the first wave of enemies is spawned using the Spawner.

The Player object and Enemy objects have their own scripts that follow the model-view-controller pattern. Both the PlayerModel and EnemyModel components inherit from CharacterModel, a base class that contains variables such as health and methods to manage those variables. Each object is controlled by a Controller component, with the PlayerController listening for both mouse and keyboard inputs while the EnemyController is AI-driven.