Extending EditorXR

This documentation is aimed at software engineers and technical artists.

Some notes first...

- Our goal is that you won't need to make any changes directly to the system. All of your creations will come from making use of interfaces. So, If you are going to create a Tool/Workspace, please place it outside of the EditorVR directory.
- Take a look at the full list of interfaces under EditorVR/Scripts/Interfaces. Most are documented well. The use of interfaces here is somewhat non-standard and is inspired by Dependency Injection (DI) / Inversion of Control (IoC) / Decorator and Adapter design patterns.
- In some cases you will only need to implement a property setter with a private getter in order to receive a method from the system.
- Post any questions you have on the forum or create issues on GitHub.

Getting Started

- Read the User Guide to get set-up and familiar with EditorXR
- Watch our under-the-hood session from Unite LA 2016.
- It may be helpful to familiarize yourself with the Input Prototype, since EditorXR makes use of it and Action Maps
- If you want to create workspaces with lists, check out the List View Framework
- You will be creating a standalone package -- you do not need to bundle EXR
- Try not to modify EXR itself, since we will be releasing updates to the platform. Talk to us, so we can figure out how to properly expose things for your needs. You're welcome to fork the project on GitHub if you would like to communicate via code.

Plan your UX

- Decide whether to create a Tool, Workspace, or combination of both
    - **Tools** have access to controllers. They can consume button presses and thumbstick motion. Tools also have a reference to a Ray Origin, which is a transform that tracks controller movement.
    - **Workspaces** are the VR equivalent of windows. By default, all workspaces can be moved and resized by grabbing different parts of the 3D frame with the selection cone. Additionally, all workspaces come with a close button and are added to the Main Menu.
- Is your tool an *Exclusive Mode* tool? This means that you will be overriding all of the default tools, and will be responsible for handling selection, transformation, and locomotion in a single tool.
- EXR also contains modules for processes like highlighting, spatial hash, pixel raycasting, and more. You may want to create a Module, but there is no way to do so currently without modifying EXR, so this is not officially supported. Talk to us on the forum if you feel there is a need for a new module.

Learn the System

- Proxies are our way of abstracting physical hardware (controllers, etc) from the concept of a tracked object with buttons. They're specifically ambiguous, so that future support can be added to non-controller based input.
- We use Unity's new Input System in a very specific way. Read more about this below, in the description of IProcessInput. Be sure to make use of the Device Assignments Window (Window -> Players) to see what action maps are currently in the Player Handle Map and which are active.
- UI takes priority over all tools, but is only active when the ray hovers over a UI element. In this way, you can always get back to the main menu and select tools, create workspaces, interact with the manipulator, and use the Radial menu. You can scroll certain UI's with the thumbpad or joystick on the hand whose ray is hovering over that UI.
- We use Ray Origins as an index throughout the system to specify which hand (or representation of a hand in the case of the MiniWorld) is doing what. This is why you will find that many of our interface methods take a rayOrigin as an argument. It allows us to keep track of each hand separately, and thus let users do different things with each hand.
- EditorXR introduced a runInEditMode property to the MonoBehaviour class, which specifies that its Awake/Start/Update methods will be called while EditorXR is running. This is different from the existing ExecuteInEditMode attribute, though classes with the attribute will also run when EditorXR is running. We use static utility methods

(U.Object.Instantiate, U.Object.AddComponent, etc.) to set this flag on components of EditorXR that have MonoBehaviours that need to run. But for instantiating scene objects, we use regular Object.Instantiate and similar methods. Be sure to consider whether you are instantiating a part of your tool that needs to run its behaviours or a part of the scene that you are editing, which should remain dormant.

Creating a Tool

First create a Monobehaviour that implements the (empty) ITool interface. In order for EditorXR to treat your new class as a tool in the system, Implementing the ITool interface is required.

```
public class TransformTool : MonoBehaviour, ITool
```

Next, start writing your Awake/Start/Update code as you normally would in a game or runtime tool. Bear in mind that if you want to use input, you can technically access input states in Update, but you should use IProcessInput instead. Our CreatePrimitiveTool is a great starting point for what we imagine most tools will need. It uses a Standard Action Map (IStandardActionMap), a Custom Menu (IInstantiateMenuUI), a Ray Origin (IUsesRayOrigin), as well as a few other interfaces.

Tool Interfaces

In order to gain access to additional EditorXR features, start implementing interfaces such as:

- IUsesRayOrigin
    - Any tool that needs to know the position or direction of the controller will need a reference to the Ray Origin transform for that controller.
- IUsesRaycastResults
    - Provides getFirstGameObject method, which returns the first GameObject hit by the ray from a given Ray Origin, or null if the ray isn't pointing at anything.
- IRayLocking
    - Allows your tool to lock the current show/hide state of the DefaultProxyRay.
- ICustomRay : IRayLocking
    - Allows your tool to provide a custom ray which overrides (and hides) the default ray and cone attached to all controllers. See BlinkLocomotionTool as an example. Use in conjunction with IRayLocking to prevent the ray from being shown by other systems.
- IUsesViewerPivot
    - Provides access to the Viewer Pivot, which is the parent transform of the camera, used to locomote the player or position objects relative to the "middle of the room."
- IUsesViewerBody

    - Provides access to EditorXR's IsOverShoulder function, used to detect whether an object's bounds are inside a special region behind the user's head used for a delete/dismiss gesture.
- IProcessInput
    - Due to the way that EditorXR defines precedence for inputs and tools, we need to explicitly control the order in which tools and other systems get to interact with the input system. Rather than relying on Script Execution Order to make sure Update functions happen in proper order, EditorXR has its own ProcessInput method that calls ProcessInput (provided by IProcessInput) on different parts of the system, explicitly within its own Update function.
    - The only argument to the ProcessInput method is a delegate called consumeControl, which will mark a particular InputControl object as locked until it returns to its default state. This will allow your tool to "grab" a trigger-press, for example, so that no other tools or systems with a lower order of precedence will react to the same input. Thus you can, for example, instantiate an object without selecting the one behind it.
- IStandardActionMap : IProcessInput
    - On top of providing a ProcessInput function to your tool, IStandardActionMap gives access to an ActionMapInput created from the StandardActionMap, which contains only the primary trigger. ActionMapInputs relay input events via the new Unity Input Prototype.
- ICustomActionMap : IProcessInput
    - On top of providing a ProcessInput function to your tool, ICustomActionMap allows you to specify your own action map, which you can assign directly to a serialized field on your script. EditorXR will create an ActionMapInput from this ActionMap, and set it back on your tool at startup.
- ITrackedObjectActionMap
    - If for some reason you want the raw tracking values for the tool's input-device/controller, ITrackingObjectActionMap will provide access to the ActionMapInput.
- IUsesProxyType
    - In the rare case where you will need to differentiate between, say Vive and Touch input, you can get the Type of the proxy that this tool's RayOrigin is attached to. Thus, your tool can compare against ViveProxy or TouchProxy and act accordingly.
- ICreateWorkspace
    - If you would like to spawn workspaces via a tool, have your tool implement ICreateWorkspace. Just give it a class that extends Workspace as an argument, and EditorXR will spawn that type of workspace. For example, createWorkspace(typeof(ProfilerWorkspace)).
- IConnectInterfaces
    - If your tool creates objects which implement EditorXR interfaces, and needs EditorXR to provide methods and features to those objects, implement

IConnectInterfaces and call connectInterfaces on your sub-objects to make the necessary connections.

- IExclusiveMode
    - Implementing this empty interface will mark your tool as an ExclusiveMode tool. Exclusive mode tools disable all other tools while they are active.
- IUsesSpatialHash
    - EditorXR uses a spatial hash to detect when the controller/device selection cones have intersected with objects. GameObjects in your scene may not have a collider component; this system allows for detecting collision without colliders. As such, we need to explicitly add and remove any renderers from the spatial hash if we want to be able to manipulate them with the cones.
- IDeleteSceneObject
    - As a requirement of the spatial hash system, we have defined an explicit interface for deleting scene objects, which will also remove them from the spatial hash. This helps avoid bogging down the system with numerous potentially null objects lingering in the spatial hash
- IGrabObject
    - If you want to grab or “pick up” an object, you will need to inform other systems that you are holding that object, and potentially check ahead of time to verify that you are allowed to grab it.
- IGetPreviewOrigin
    - Sometimes it is helpful to display a preview of a held object, or other non-menu objects attached to the controller. IPreviewOrigin provides access to a specific transform child of the controller (assigned in ProxyHelper) for these purposes.
- IPlaceObject
    - EditorXR has some built-in logic for placing objects intelligently, so that they do not surround you if they are very large. Use IPlaceObject to take advantage of this.
- IGameObjectLocking
    - Allows a tool to lock or unlock a GameObject, in order to prevent selection while locked.
- IDirectSelection
    - Gives your tool access to the current Direct Selection which will contain entries if a (controller) selection cone is currently intersecting with a scene object
- IDropReceiver
    - Allows an object to accept drag-and-drop objects
- IDroppable
    - Designates an object as something which can be dropped on objects implementing IDropReceiver.
- IInstantiateMenuUI
    - Used to instantiate/create custom menus for tools, in the manner required by EditorXR for proper input-handling.
- ISelectionChanged

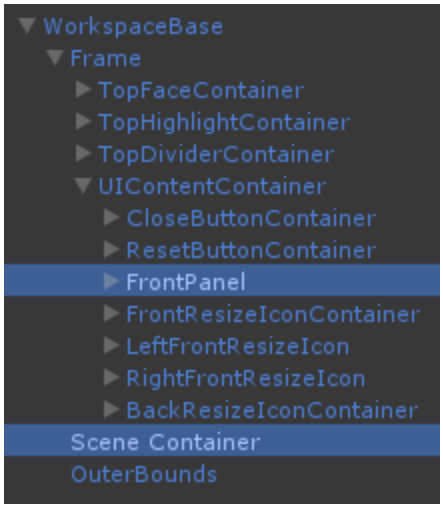
    - Provides access to EditorXR's OnSelectionChanged callback, which is called explicitly at start in order to inform tools (and other systems) of objects that were selected when EditorXR was started
- ISetHighlight
    - Provides access to the Highlight Module's SetHighlight function, which will use the Highlight Module to draw a faint blue mesh on top of a given renderer

Creating a Workspace

Workspaces are a little more complicated than tools. They are built upon a class that extends Workspace; and also include prefabs that contain your workspace UI elements. These prefabs will be parented under the workspace frame. We recommend starting by pulling the WorkspaceBase into the scene, as it provides the Canvas you will be using for UI, and has non-standard pixel per unit settings.

Feel free to copy an existing workspace as a model for your own (Console / Project / Inspector / MiniWorld / Hierarchy). The Hierarchy is a good example of a simple workspace with nothing but a list, and the MiniWorld is a good example of a more advanced workspace with scene objects and complex interaction & visual behaviours.

You can add objects to workspaces in one of two places: the SceneContainer (Top of workspace) and the FrontPanel.



While you are mocking up your workspace UI elements, be sure to add your objects as children of these transforms. The content prefab and front face UI are optional, of course, and could be created entirely in code. However, it's usually helpful to design your UI ahead of time. You can define a minimum size, and custom starting size for your workspaces. Resizing of workspaces can be enable/disabled. If your workspace support resizing, you need to update the size/aspect

of your rectangles in an OnBoundsChanged override in your Workspace class. A simple example implementation of OnBoundsChanged can be seen in the HierarchyWorkspace.

The Workspace class also provides virtual OnCloseClicked and OnDestroy methods so that derived Workspaces can do things on close such as clean-up.