Extending editors

Changelist

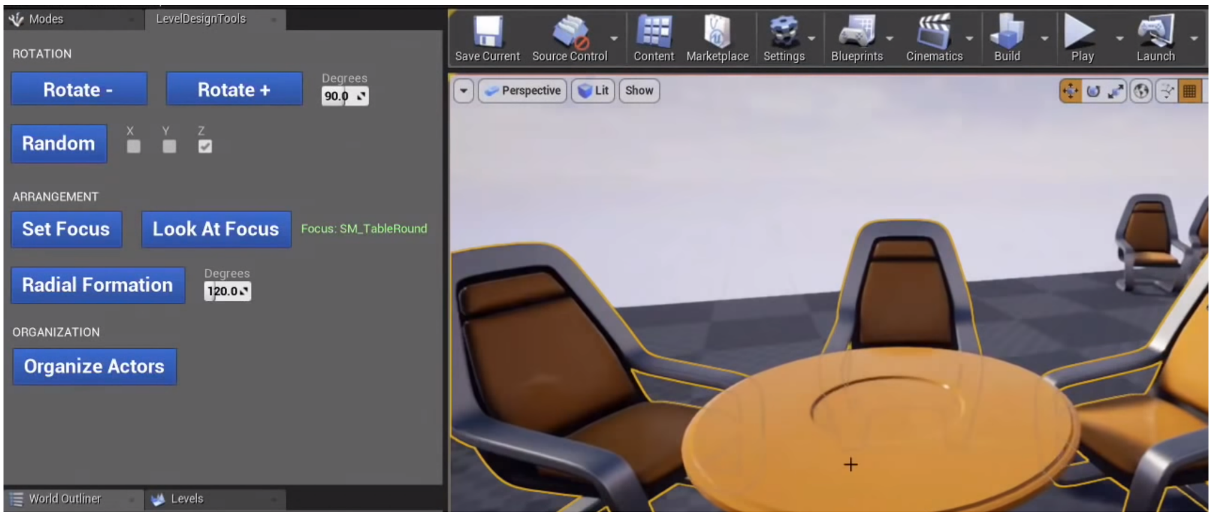
Date	Name	Topics
17-09-2020	Jens Petter	Research into how to extend the Unity and Godot editor.
19-09-2020	Jens Petter	Finishing research into how to extend the editor in Unreal.
20-09-2020	Jens Petter	Finished the base of this document by adding my research into Amazon Lumberyard, CryEngine and adding my research takeaways / final thoughts.
21-09-2020	Tim Rademaker	Small changes to sentence structure, phrasing and grammar

Engines

Unreal

Editor Utility Widgets

Unreal works with widgets. This is how both in game and in editor UI is made. One can create an editor utility widget using the option Editor Utilities -> Editor Widget. This widget behaves the same as how a normal in game UI widget is constructed. It has the same UI drag and drop functionality and also the same functionality such as creating on click events for a button just like one can do with a “normal” in-game UI widget in Unreal.

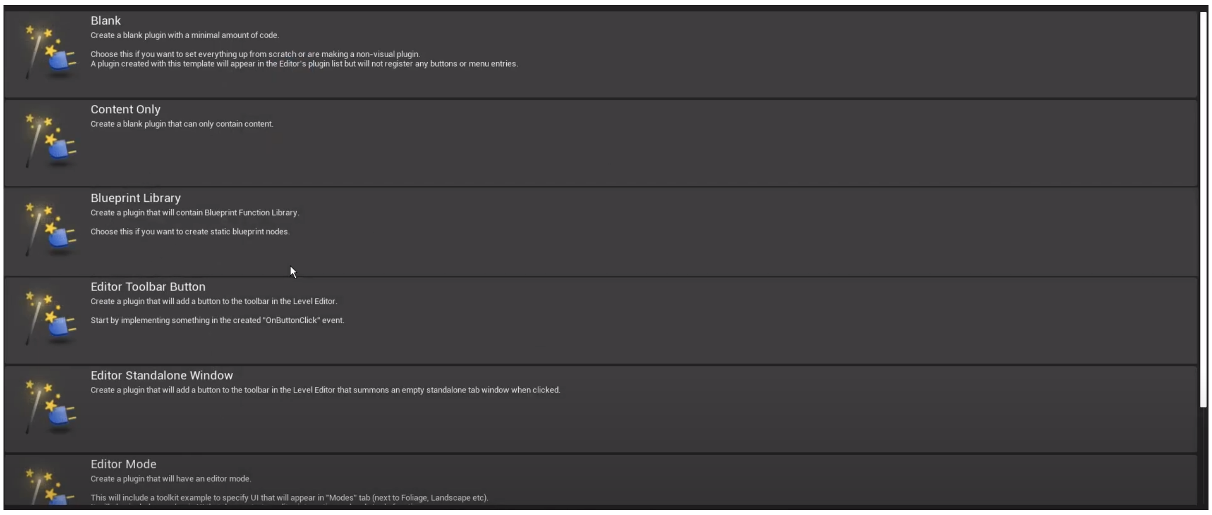


A screenshot is found above from an Unreal Engine livestream where they showed how to use editor utility widgets for this first time. Here they made a small level editor tool. Link to the source of this image here.

Plugins

Unreal has the option to enable / disable plugins. There are built-in plugins but users can also create a plugin themselves. This is done in C++. A plugin is more powerful than an editor utility widget. One reason for this is that a plugin is a package that can be plugged into any project and is more extendable than a one-file editor utility widget.

Unreal already made life a bit easy for a programmer who wants to make a plugin themselves. There is a list of plugin templates developers can choose from. Plugins can be seen as extending the editor, however there are also plugins that extend the UI of the editor. I think this feature is very powerful in Unreal and I look forward to seeing what can be made with this feature.



Blank
Create a blank plugin with a minimal amount of code.

Choose this if you want to set everything up from scratch or are making a non-visual plugin.
A plugin created with this template will appear in the Editor's plugin list but will not register any buttons or menu entries.

Content Only
Create a blank plugin that can only contain content.

Blueprint Library
Create a plugin that will contain Blueprint Function Library.

Choose this if you want to create static blueprint nodes.

Editor Toolbar Button
Create a plugin that will add a button to the toolbar in the Level Editor.

Start by implementing something in the created "OnButtonClick" event.

Editor Standalone Window
Create a plugin that will add a button to the toolbar in the Level Editor that summons an empty standalone tab window when clicked.

Editor Mode
Create a plugin that will have an editor mode.

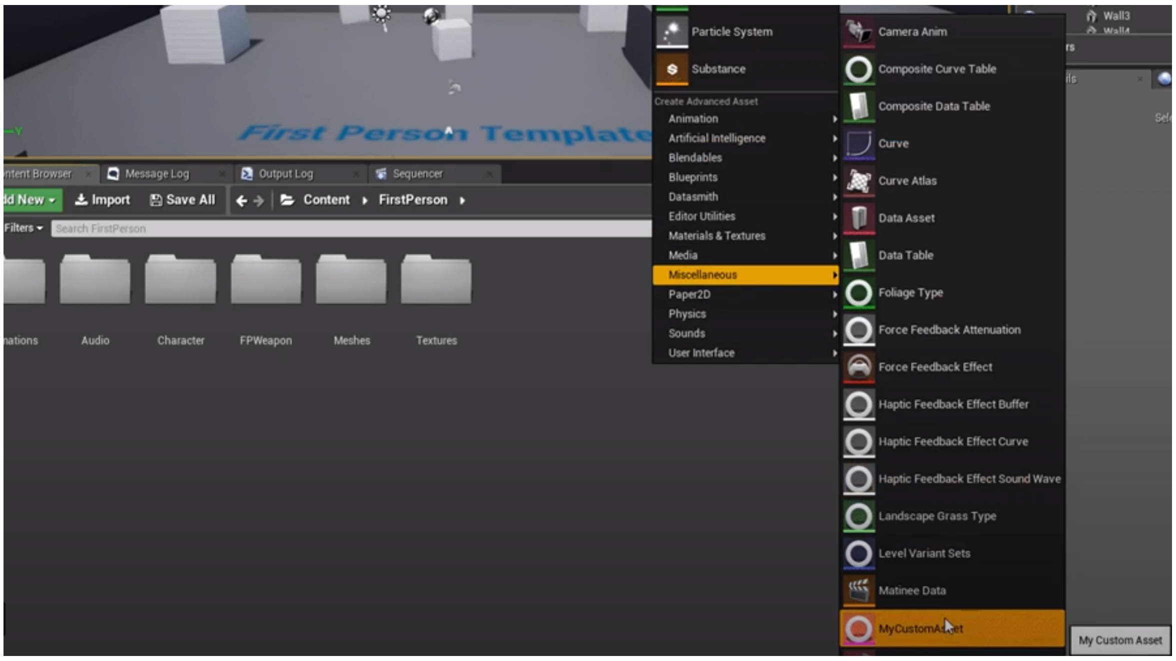
This will include a toolkit example to specify UI that will appear in "Modes" tab (next to Foliage, Landscape etc).

A screenshot showing a few of the example templates one can choose as a start for his / her plugin when he / she wants to make a plugin. Link to the source of this image here.

Custom Assets

Developers have the possibility to create custom assets in Unreal. This requires some C++ code but one will have a struct-like blueprint that behaves as its own asset when this code is constructed. Users can create their own custom asset through the editor like one would create any other asset. Maybe the most important feature about creating your own asset is that, in code, developers can specify what needs to happen to this asset at what time. Developers can, for example, decide that certain things need to happen when the asset is created where asset creation is a callback given to the user for when the user wants to create a custom asset.

This option might not be directly seen as a method to extend the editor but this method does allow the user of the Unreal engine to create new types of assets which I think is very powerful. Because of this I see this method as a method to extend the editor.



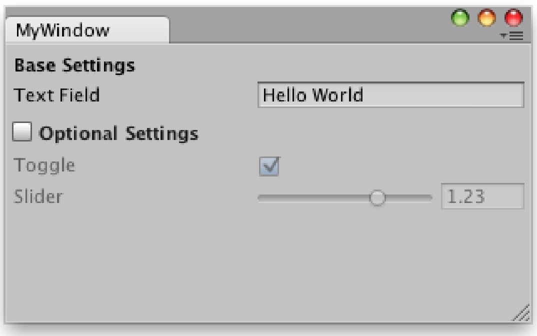
A screenshot is found above where the person on this computer shows that his / her custom asset is visible in the asset creation dropdown window. Link to the source of this image here.

Unity

There are several ways to extend the editor in Unity. One either writes a custom editor window, a custom inspector window or uses UIElements to extend the editor. This is all done in C#.

Editor window

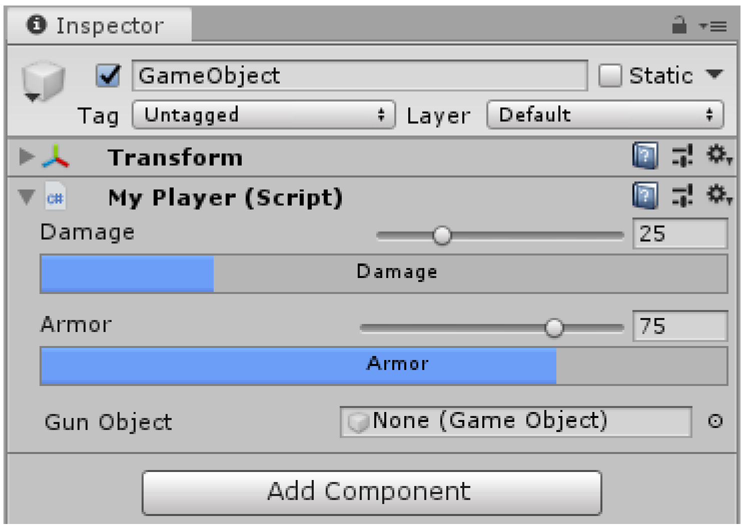
An editor window is simply a new window inside the editor. One can then (in C#) decide what this window is going to look like. This window can be made so it is accessible through the application window bar.


MyWindow
Base Settings
Text Field: Hello World
Optional Settings
Toggle: ✓
Slider: 1.23

An example of a custom editor window. Link to this image on the Unity documentation [here](#).

Inspector window

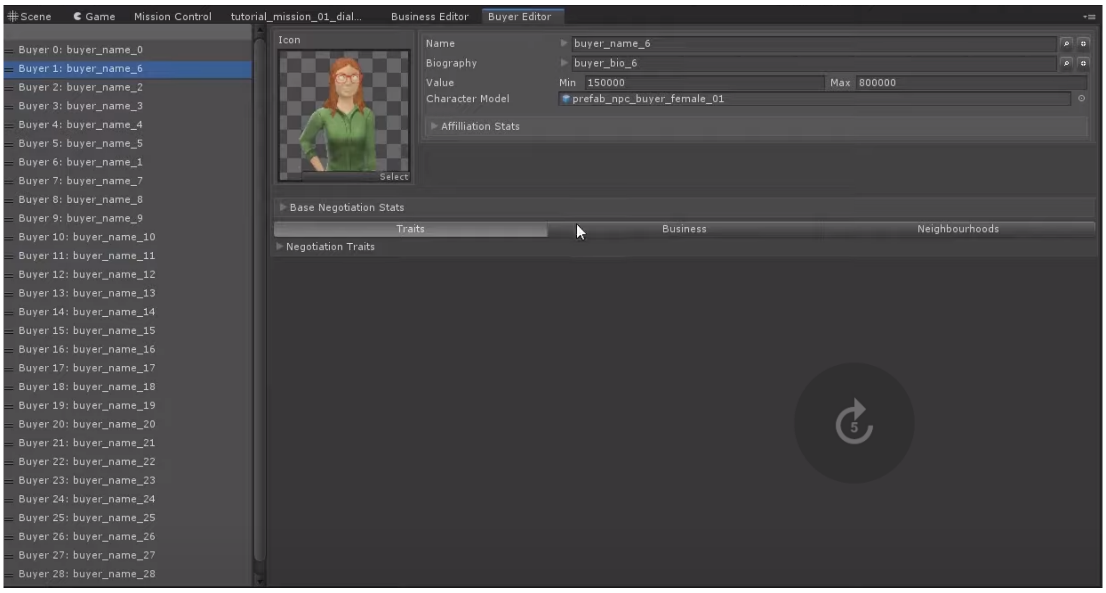
An inspector window is the window users can see whenever selecting an object in Unity where then in the inspector the information of that object (components etc) shows up. A custom inspector can manipulate the data of this inspector window where the user can make sure that something can happen programming-wise, for example based on a value change in the inspector. Other elements like custom bars or buttons can be made in a custom editor window. The beauty of this is that with custom code one can easily access and tweak component values.


Inspector
GameObject
Static
Tag: Untagged
Layer: Default
Transform
My Player (Script)
Damage: 25
Damage
Armor: 75
Armor
Gun Object: None (Game Object)
Add Component

An example of a custom inspector window where in this case custom bars are made. Link to image on Google [here](#).

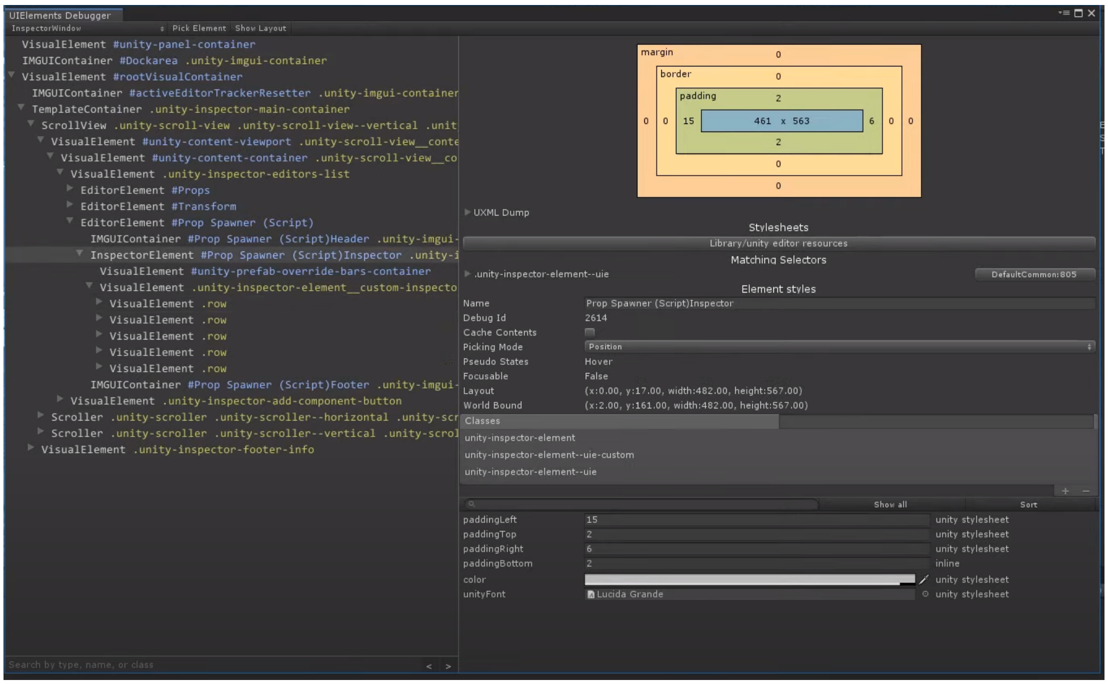
UIElements

UIElements is a new feature of how to extend the editor in Unity that has been added to Unity in 2019. One can extend the editor even further with the use of UIElements instead of using a custom editor or inspector window. A construction of a UIElement needs 3 files rather than 1 file when creating a custom editor or inspector window. An editor script, styles script and template script are needed to create a UIElement window. The editor script is the script one also needs for a custom editor or inspector window, the template file is a .uxml file with all objects that exist in the UIElement and the styles file is a .uss where connections to this .uxml objects and .uss file are made. Where in the .uss file the objects from the .uxml file can be graphically modified. One can see this .uss file as a .css file.



```
Buyer 0: buyer_name_0
Buyer 1: buyer_name_6
Buyer 2: buyer_name_2
Buyer 3: buyer_name_3
Buyer 4: buyer_name_4
Buyer 5: buyer_name_5
Buyer 6: buyer_name_1
Buyer 7: buyer_name_7
Buyer 8: buyer_name_8
Buyer 9: buyer_name_9
Buyer 10: buyer_name_10
Buyer 11: buyer_name_11
Buyer 12: buyer_name_12
Buyer 13: buyer_name_13
Buyer 14: buyer_name_14
Buyer 15: buyer_name_15
Buyer 16: buyer_name_16
Buyer 17: buyer_name_17
Buyer 18: buyer_name_18
Buyer 19: buyer_name_19
Buyer 20: buyer_name_20
Buyer 21: buyer_name_21
Buyer 22: buyer_name_22
Buyer 23: buyer_name_23
Buyer 24: buyer_name_24
Buyer 25: buyer_name_25
Buyer 26: buyer_name_26
Buyer 27: buyer_name_27
Buyer 28: buyer_name_28

Name             buyer_name_6
Biography        buyer_bio_6
Value            Min 150000        Max 800000
Character Model  prefab_npc_buyer_female_01
```

A screenshot of a tool that was created using UIElements. Link to the source of this image here.



```
VisualElement #unity-panel-container
 IMGUIContainer #Dockarea .unity-imgui-container
 VisualElement #rootVisualContainer
  IMGUIContainer #activeEditorTrackerResetter .unity-imgui-container
  TemplateContainer .unity-inspector-main-container
   ScrollView .unity-scroll-view .unity-scroll-view--vertical .unit
    VisualElement #unity-content-viewport .unity-scroll-view__conte
     VisualElement #unity-content-container .unity-scroll-view__co
      VisualElement .unity-inspector-editors-list
       EditorElement #Props
       EditorElement #Transform
       EditorElement #Prop Spawner (Script)
        IMGUIContainer #Prop Spawner (Script)Header .unity-imgui-
        InspectorElement #Prop Spawner (Script)Inspector .unity-i
         VisualElement #unity-prefab-override-bars-container
         VisualElement .unity-inspector-element__custom-inspecto
          VisualElement .row
          VisualElement .row
          VisualElement .row
          VisualElement .row
          VisualElement .row
        IMGUIContainer #Prop Spawner (Script)Footer .unity-imgui-
      VisualElement .unity-inspector-add-component-button
    Scroller .unity-scroller .unity-scroller--horizontal .unity-scr
    Scroller .unity-scroller .unity-scroller--vertical .unity-scrol
   VisualElement .unity-inspector-footer-info

margin 0 / border 0 / padding 2
0  0  15  461 x 563  6  0  0
2 / 0 / 0

.unity-inspector-element--uie          DefaultCommon:805

Name            Prop Spawner (Script)Inspector
Debug Id        2614
Cache Contents
Picking Mode    Position
Pseudo States   Hover
Focusable       False
Layout          (x:0.00, y:17.00, width:482.00, height:567.00)
World Bound     (x:2.00, y:161.00, width:482.00, height:567.00)

unity-inspector-element
unity-inspector-element--uie-custom
unity-inspector-element--uie

paddingLeft     15                     unity stylesheet
paddingTop      2                      unity stylesheet
paddingRight    6                      unity stylesheet
paddingBottom   2                      inline
color                                  unity stylesheet
unityFont       Lucida Grande          unity stylesheet
```

A screenshot of the UIElement debugger. Link to the source of this image here.

Plugins

Unity has the possibility to create native plug-ins and managed plug-ins. Users can add functionality to the editor and to game code with plugins.

Unity has extensive support for native plug-ins, which are libraries of native code written in C, C++, Objective-C, etc. Plug-ins allow game code (written in C#) to call functions from these libraries. This feature allows Unity to integrate with middleware libraries or existing C/C++ game code. In general, plug-ins in Unity are built with native code compilers on the target platform. Since plug-in functions use a C-based call interface, one must avoid name mangling issues when using C++ or Objective-C.

Managed plug-ins are managed .NET assemblies that one creates with tools like Visual Studio. They contain only .NET code which means that they can’t access any features that .NET libraries do not support. However, the standard .NET tools that Unity uses to compile scripts can access the managed code. Therefore, there isn’t a lot of difference in usage between managed plug-in code and Unity script code, except that plug-ins are compiled outside of Unity and so the source might not be available.

CryEngine

One can see below that during my research I found 2 options on how one can extend the editor in CryEngine. Documentation on this is very limited. I could find nothing other than the CryEngine documentation on how to extend the editor in this engine which means that close to no one has tried extending the editor and put their finding of that online. I tried to do this myself in CryEngine, but Visual Studio 2019 (the version I use) is not supported. The most up to date version of Visual Studio that is supported is Visual Studio 2015 which I think is a shame.

Editor window

One can create an editor window in CryEngine. This is done in C++. Documentation on how to do this is however very limited. The documentation (link in the reference(s) page of this document) shows which class one should inherit from to make an editor window and what an editor window can be used for, but that is sadly the extent of the documentation for that.

Plugins

The CryEngine plugin system exposes support for creating plugins in both C++ and C# and allows users to simply drop a plugin into their project and to see the benefits instantaneously. The Plugin system, according to the documentation, is still in beta. It also concerns me that the plugin system documentation of CryEngine is also very limited.

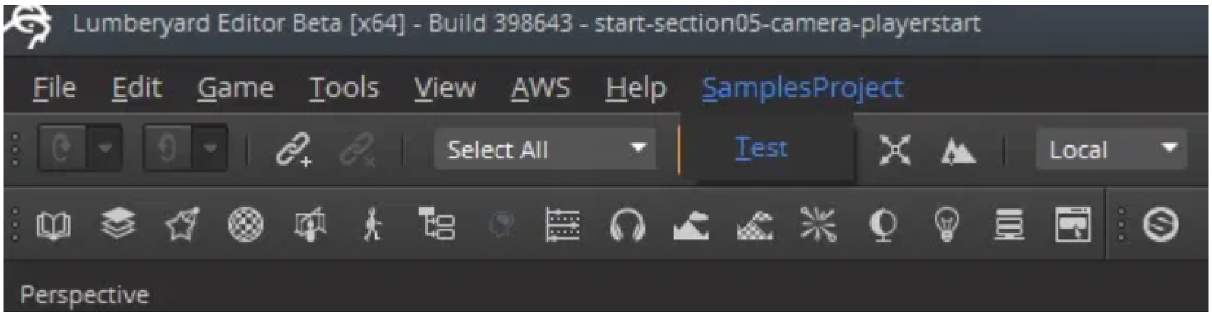
Amazon Lumberyard

The documentation of Amazon Lumberyard doesn't mention directly how to extend the editor anywhere. However there are ways to do this. Amazon Lumberyard does give the user access to its source code which can be pretty powerful for a tools programmer to extend the engine with.

Project menu

One can easily add menu items that perform different functionality by simply adding that custom code to the existing code of the editor. The fact that one simply adds code to the existing code of Amazon Lumberyard in order to make their custom functionality work concerns me a bit since because of this there is no clear separation between existing engine code and modified engine code, although I have to say that Unreal has this as well where one has access to the source code which sometimes can come in handy.

Amazon Lumberyard is c++ based which is why adding a project menu is also done in c++.



Lumberyard Editor Beta [x64] - Build 398643 - start-section05-camera-playerstart
File
Edit
Game
Tools
View
AWS
Help
SamplesProject
Select All
Test
Local
Perspective

A screenshot of a tutorial I followed to add a custom button to the project menu toolbar. Link to the source of this image here.

Gems

The modular gems system is a management infrastructure for sharing code and art assets between Lumberyard game projects. The modular gems system consists of gem packages that you can access and manage with the Project Configurator or Lmbr.exe. You can use gems to add functionality such as code, art, scripts, supporting files, and references to other gems to your game project. Gems seem to be the one main feature of Amazon Lumberyard to extend the editor with since it claims that basically anything can be achieved with the gems system.

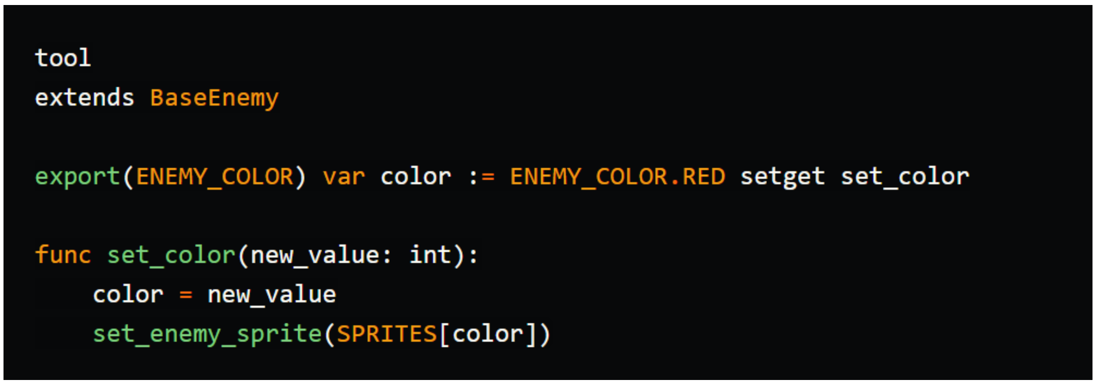
The gems system can also be seen as modules. It can easily be taken in and out of any project and code for gems is written in C++.

Godot

Godot also has a few different ways of letting the user extend Godot. 2 of the 3 methods that I will talk about below here are written in GDScript. GDScript is a high-level, dynamically typed programming language used to create content. It uses a syntax similar to Python.

GDScript Tool Mode

This method is the simplest way of extending the Godot editor. This method can be applied to any script. A simple ‘tool’ keyword at the top of the script is needed to make sure the script does not only run in game mode but also in editor mode. One has to attach this script to an object in the scene in order to make such a script run in the Godot engine. Certain callback functions from the engine (setting color, getting value etc), basic Godot script functions (OnReady etc), Coroutines, signals and more can be used in a script that is in ‘tool’ mode.

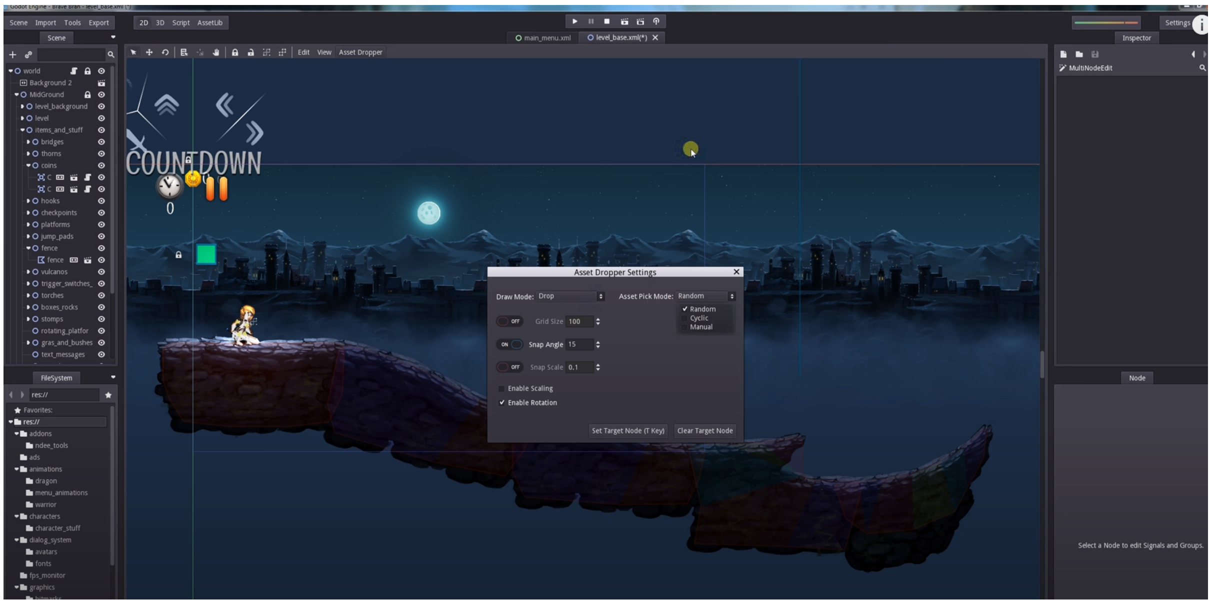


```
tool
extends BaseEnemy

export(ENEMY_COLOR) var color := ENEMY_COLOR.RED setget set_color

func set_color(new_value: int):
    color = new_value
    set_enemy_sprite(SPRITES[color])
```

An example of how a ‘tool’ mode script can look like in GDScript. In this script whenever the enemies color property is changed the sprite is updated to match the new color. Link to the source of this image here.

Plugins

Plugins are much more powerful than the GDScript Tool Mode method. Plugins are executed when the editor starts and the plugin is enabled. They are not part of Godot scenes, instead they run independently inside the editor. Often plugins are used for making custom UI windows inside the engine but also things like an importer / exporter for a specific file type can be achieved by using plugins.



Asset Dropper Settings

Draw Mode: Drop

Asset Pick Mode: Random

✓ Random
Cyclic
Manual

OFF Grid Size 100

ON Snap Angle 15

OFF Snap Scale 0.1

Enable Scaling

✓ Enable Rotation

Set Target Node (T Key) Clear Target Node

Screenshot of a custom asset tool that someone made in Godot in order for him to create levels faster. Link to the source of this image here.

C++ Modules

Custom C++ modules can be imported in the Godot engine. This feature is extremely powerful since a lot can be achieved with C++ modules. One can also access engine or object related data, even extending this data. C++ runs faster than GDScript as well.

Research Takeaways / Final thoughts

Game engines

- Godot, Unity and Unreal are the 3 engines that have the most possibilities to extend their engine. Unity definitely wins out of these 3 engines, since it can do everything Unreal (except for custom assets) and Godot can but also can have a custom CSS editor for their tools which is very cool and powerful I think.
- The creation of custom assets is I think one of the most important features of Unreal when it comes to extending the editor.
- The fact that the same widget system in Unreal for making in-game UI and custom editor windows is very neat and nice I think.

More

- CryEngine doesn’t support Visual Studio 2017 and 2019.
- Amazon Lumberyard is based on CryEngine.
- The documentation of Amazon Lumberyard is very limited.

Reference(s)

Unreal

- Creating Editor Utility Widgets | Inside Unreal (link)
- CREATING AN EDITOR MODULE IN UNREAL ENGINE 4 (link)
- WRITING A CUSTOM ASSET EDITOR FOR UNREAL ENGINE 4 - PART 2 (link)
- SAVE TIME PROGRAMMING - Unreal Engine 4 Plugins (link)
- Customizing the editor's toolbar buttons menu via custom plugin (link)
- Unreal Engine C++ Tutorial: Plugins (link)

Unity

- How to make an EDITOR WINDOW in Unity (link)
- How to make a CUSTOM INSPECTOR in Unity (link)
- Editor Windows (link)
- Custom Editors (link)
- Customize the Unity Editor with UIElements! (link)
- UIElements Developer Guide (link)
- Easy Editor Windows in Unity with Serialized Properties (link)
- Native plug-ins (link)
- Managed plug-ins (link)

CryEngine

- Creating a new editor window (link)
- Sandbox C++ plugins (link)
- Sandbox Python Plugins (link)
- CEditorWidget base class on GitHub (link)

Amazon Lumberyard

- Adding A Project Menu To The Lumberyard Editor (link)
- Add modular features and assets with Gems (link)
- How to Create & Enable Gems in Amazon Lumberyard | Lumberyard Tutorial 2019.19 (link)
- Gems Available in Lumberyard (link)

Godot

- Godot Engine: Extendable Editor (link)
- Tool mode in GDscript (link)
- Making plugins (link)
- Godot 3 Tutorial | Custom Modules with C++ (Inspired from Godot Docs) (link)
- Custom modules in C++ (link)