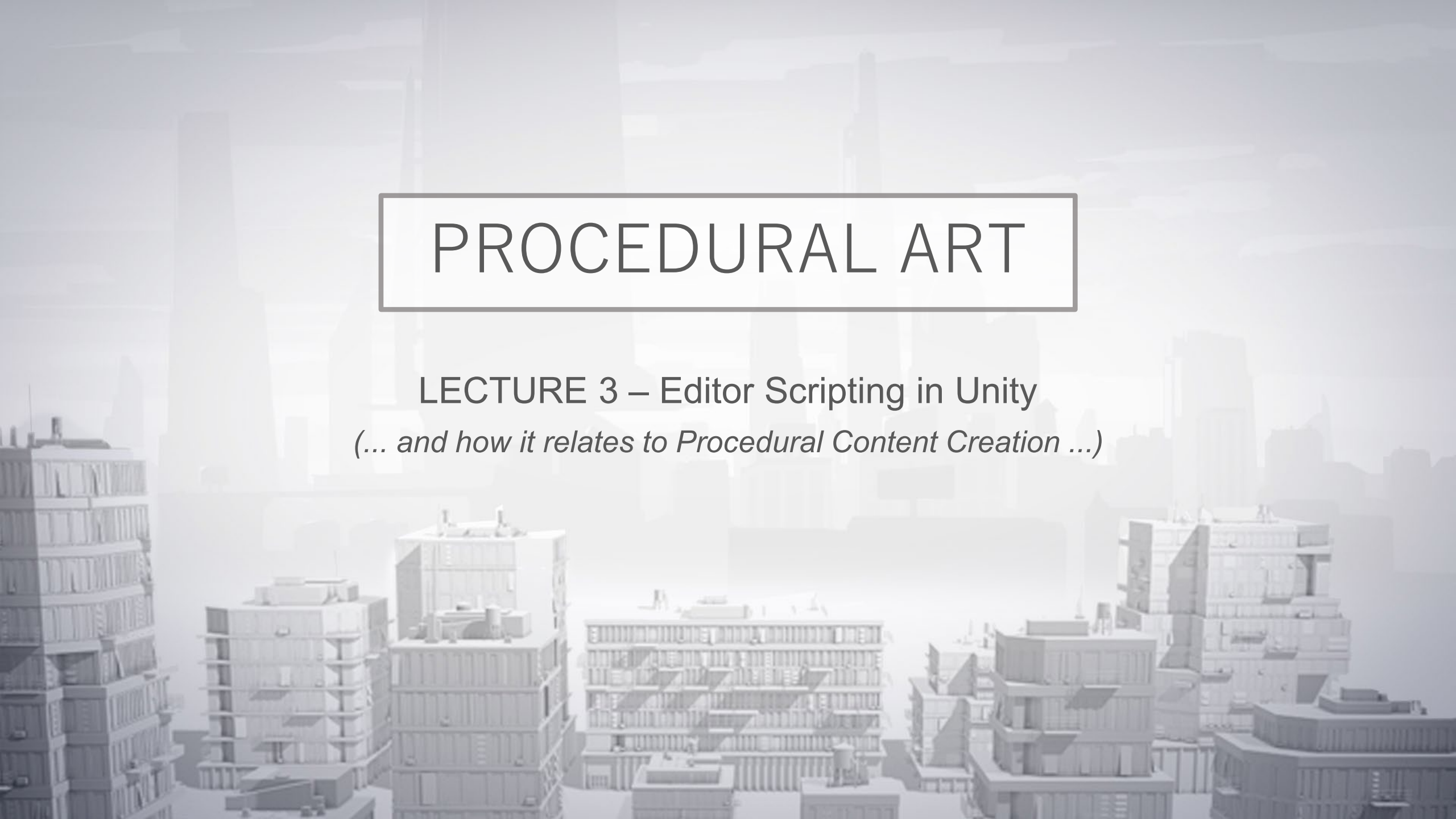


PROCEDURAL ART

LECTURE 3 – Editor Scripting in Unity

(... and how it relates to Procedural Content Creation ...)

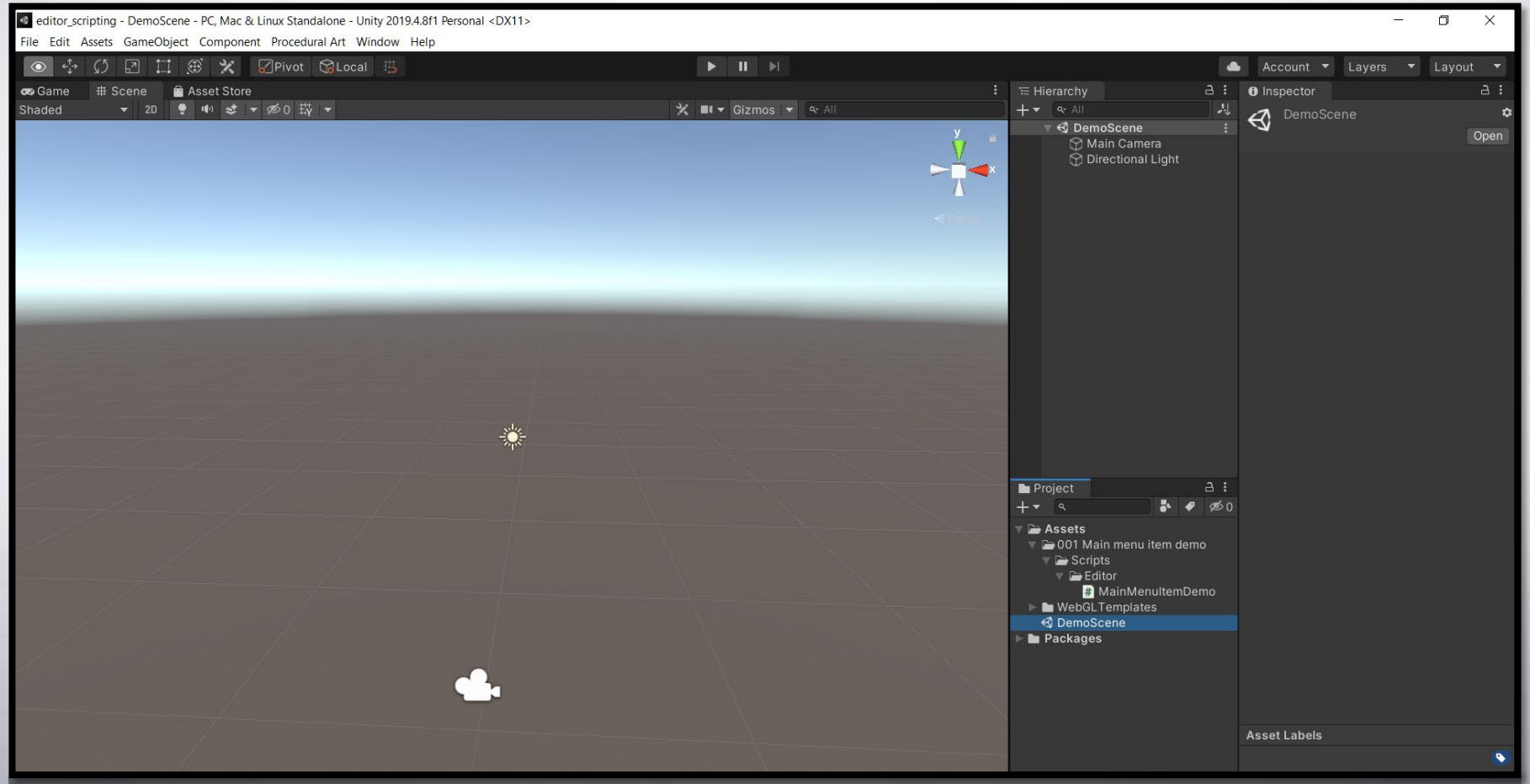
Lecture Overview

- What is Editor Scripting?
  - In general
  - Examples & inspiration (most of which will be on Blackboard later)
  - Different categories of editor scripting
- Editor Scripting in the context of the procedural art course
  - For what can/would/should you use it?
  - Do you really need it?
- Getting started with Editor Scripting yourself:
  - Introduction
  - Cookbook
  - References
- TL;DR; after the lecture:
  - Check 'Getting starting part 4' for triggering code at Editor time
  - Check the Stacking tool, buildingplot & polygon tool
  - This is not a tutorial, but an overview of the most important possibilities of editor scripting and a reference to be used when and if necessary.
  - Editor Scripting is not a requirement to pass the course, pick and choose what you can use

What is Editor Scripting?

Editor Scripting in general

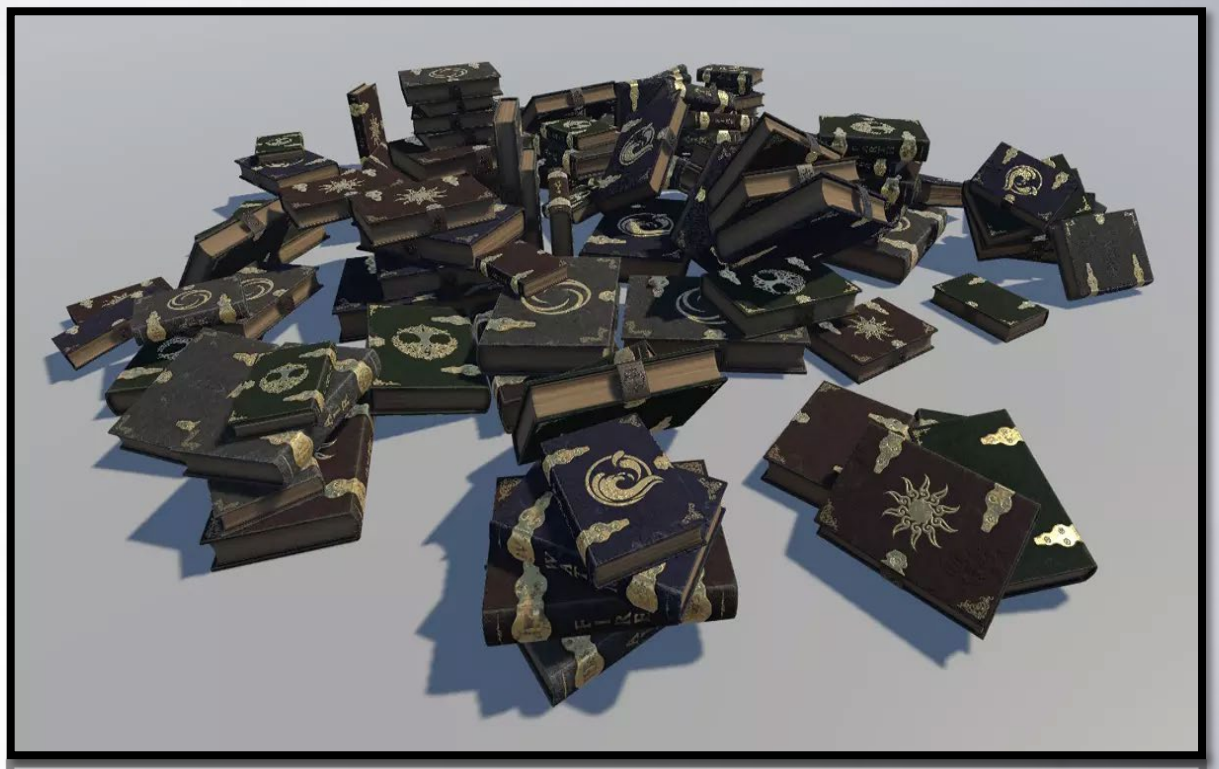
(Almost) everything you see and can do at Editor time is implemented or can be replicated through editor scripting:



Editor Scripting examples

Stacking tools

- Simple goal: easily create stacks of objects



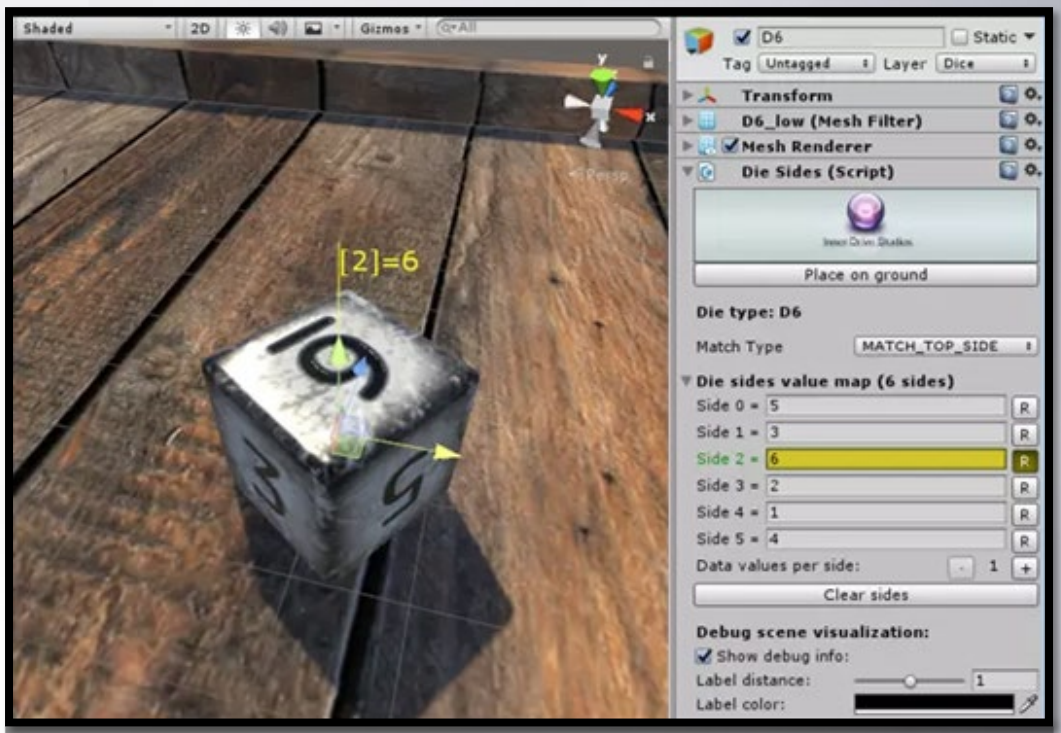
- Demonstrates:
  - Custom inspector
  - Scene interaction (could be better)
  - That it is not editor time **or** runtime, but both
  - Lack of undo ;)
  - Variety/variability
- Procedural part:
  - Stacking objects automatically
  - Finding mesh bounds
- Where can you find it?
  - Full version available on blackboard (including the books ☺)
  - https://assetstore.unity.com/packages/tools/utilities/stacking-tools-112062 (free, without books)

Dice Creator Pack

- Goal: be the best dice pack on the asset store ;)

- Demonstrates:
  - Custom inspectors (generating side normals)
  - Scene interaction (showing side normals)
  - Processing all assets in batch
    (Generating material sets)
  - Custom windows
    (Circle layout, Side editors)
  - Custom utilities (Screenshot generator)

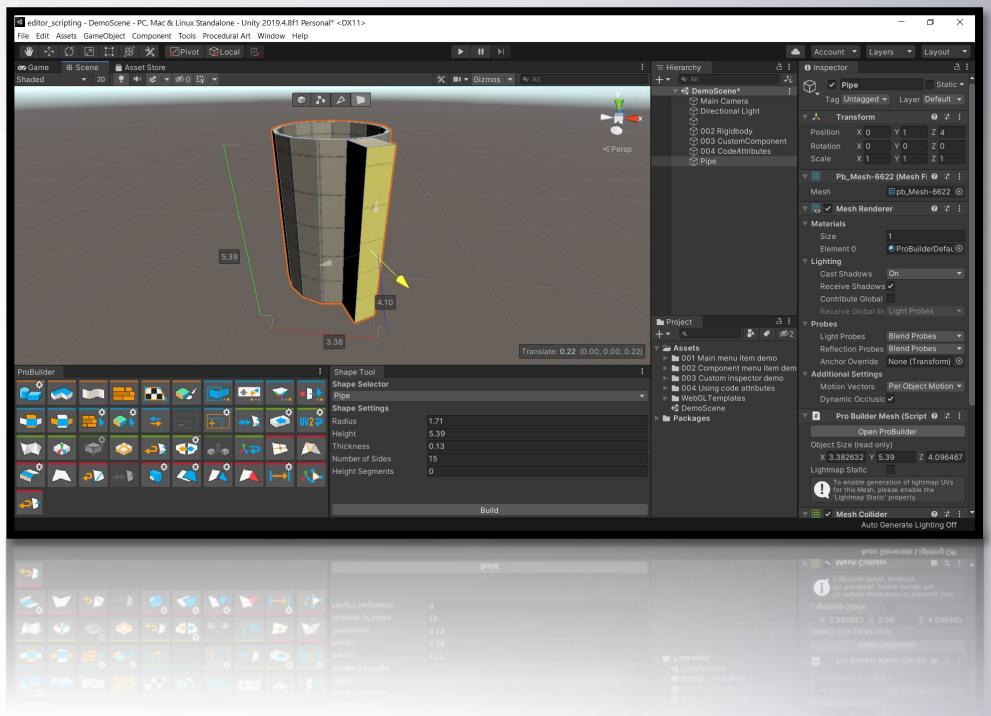
- Procedural part:
  - Finding die sides, aligning with ground, layouting, taking screenshots: all things you could do manually



- Where can you find it?
  - Lite version with ***all*** scripts available on blackboard
  - https://assetstore.unity.com/packages/3d/props/dice-creator-pack-128640 (paid)

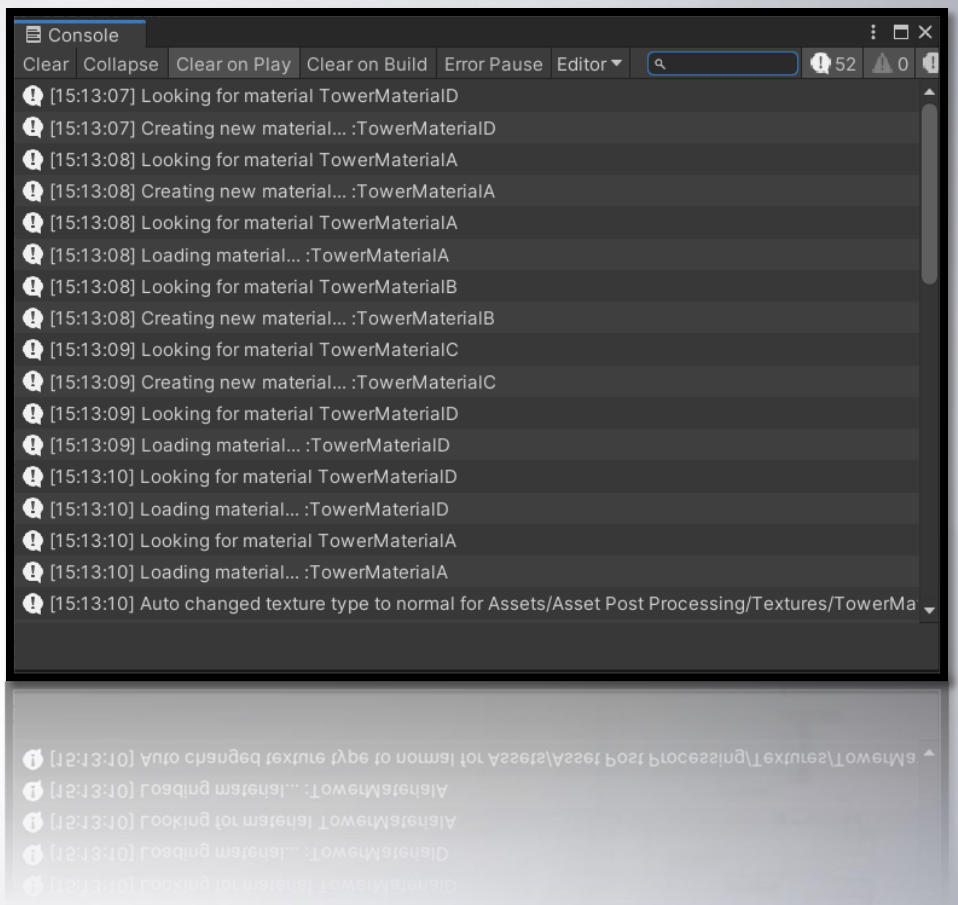
ProBuilder

- Goal: Enabling 3D modelling in Unity:
  - Excellent for prototyping, not for high poly sculpting ;)
- Demonstrates:
  - That it is possible to seamlesly extend the whole interface with new capabilities that weren't there before
- Procedural part:
  - Parametrized shape generation and texturing
- Installation:
  - Just use the example project from blackboard
  - Or create an empty project and then:
    - Open the package manager
    - Select Advanced / Show preview packages
    - Install ProBuilder & ProGrids



Asset Post Processing

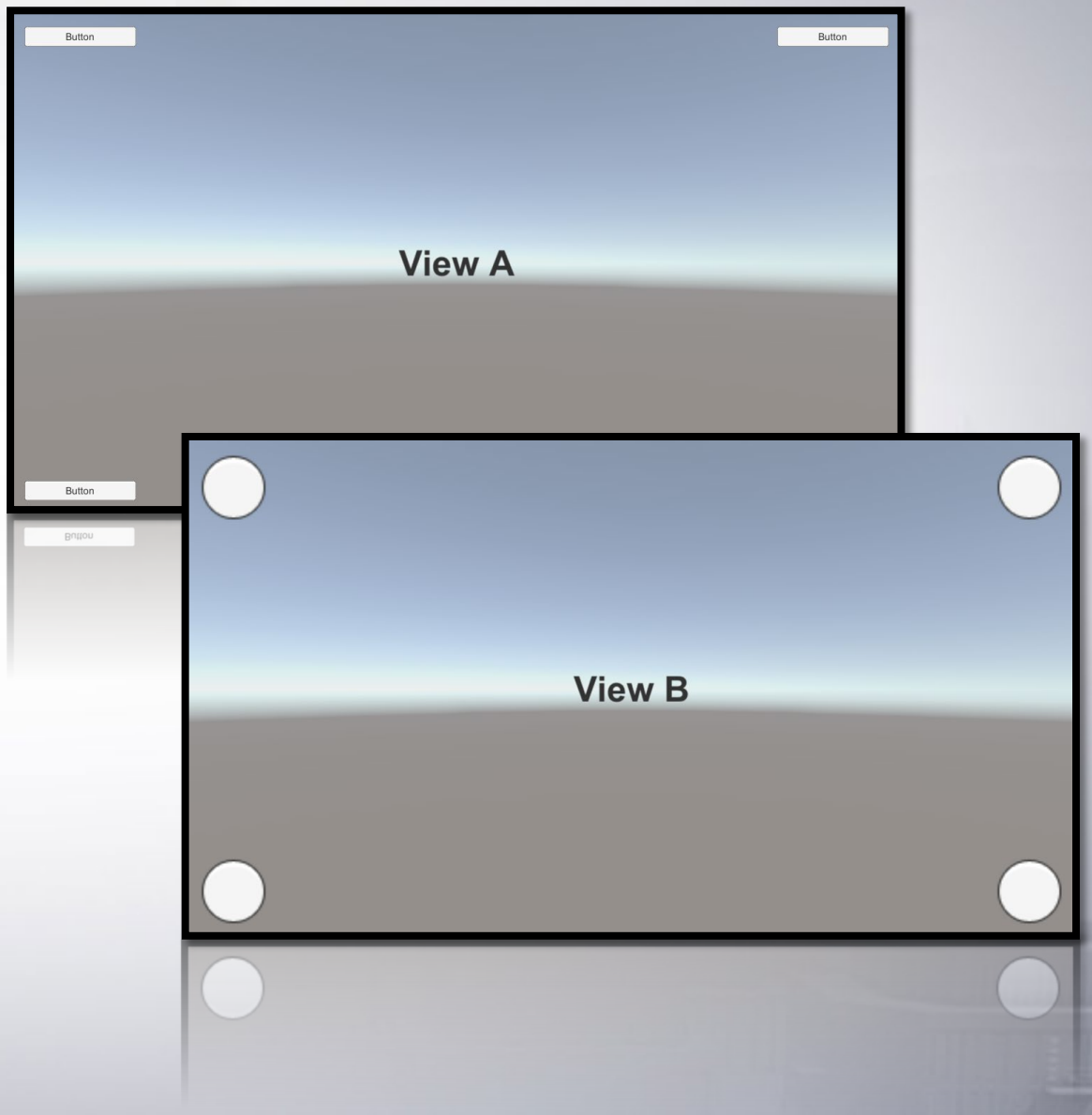
- Use case:
  - "Let's make a cool game!" Ok yeah !
  - Artist creates 100 PBR materials:
    - Albedo, Normal, Height, Metallic, Occlusion textures exported from Substance Painter for each material
  - With (500 textures) do {
    Create material
    Drag albedo texture in material's albedo slot
    Set normal's texture type to normal
    Drag normal texture in material's normal slot
    Drag height texture in material's height slot
    Drag metallic texture in material's metallic slot
    Drag occlusion texture in material's occlusion slot
    }
  - Cool down wrists with ice pack



  - OOOOOORRRRR.... implement an ***AssetPostProcessor***

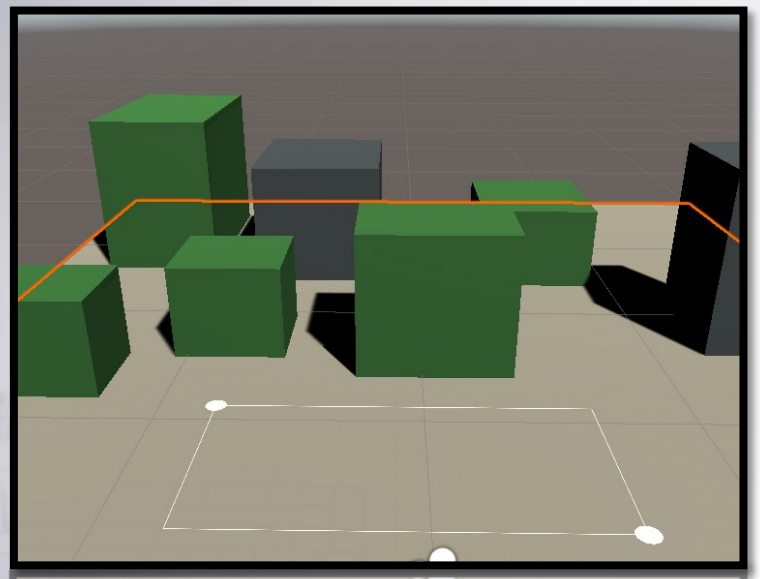
View switching

- Use case:
  - multiple screens in your game
  - while editing you switch a lot between them
  - constantly requires you to select and disable one screen, and enable the other
- Goal: automatically disable the all other screens when selecting another one.
- Demonstrates:
  - very minor issues that can still save a lot of time
  - that editor scripts don't *have* to be big
- See **ViewSwitcher**

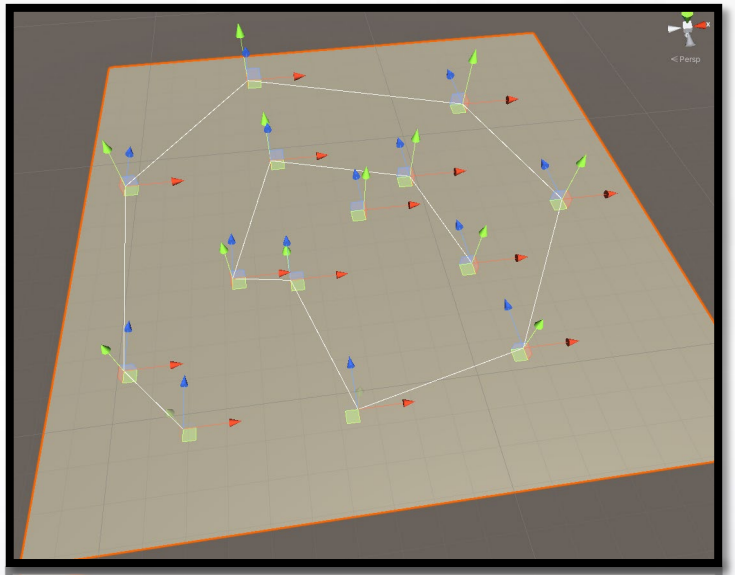


Additional examples

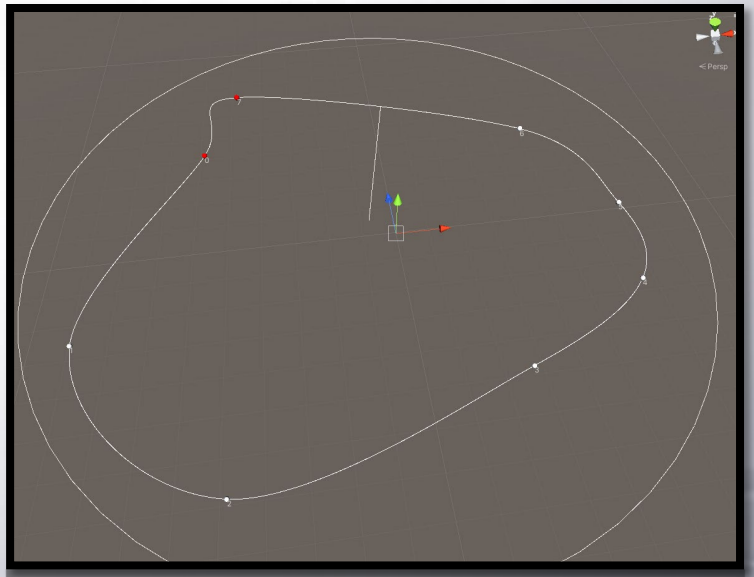
Drag n Draw Building plot generator



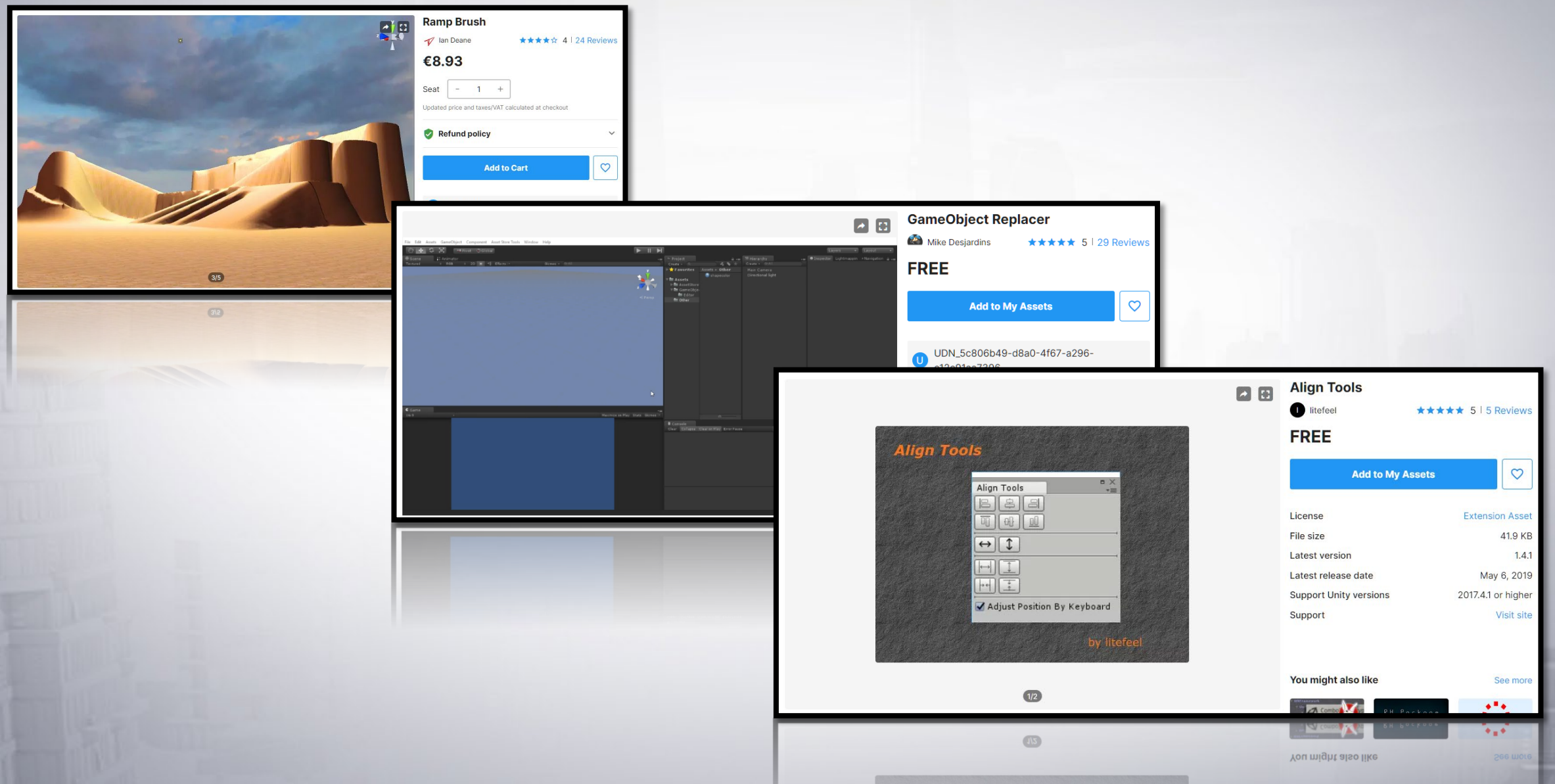
Polygon editor (Paul)



Spline editor (learn.unity.com)



Etcetera etcetera ...



Summing up: Editor Scripting is AweSoMe \^^\/oO\/^^/

- As seen, Editor Scripting has a lot of different capabilities/application areas:
  - Trigger things in your code that were already there, making your runtime adventures editable
  - Build simple utilities for things you could already do manually, but now much faster
    (eg Texture Importing, but also aligning objects, replacing prefabs, stacking objects, processing your whole scene, etc, etc)
  - Build complicated new features into Unity that weren't there before
    (eg ProBuilder, ProGrid and a lot of other packages/assets)

- All these different types of capabilities can be divided in a couple of different categories of Editor Scripting:
  - Creating (context) menu items
  - Creating custom inspectors / propertydrawers
  - Creating Scene/Editor extensions
  - Implementing Asset processing
  - ... and much more ...

Editor Scripting ...

... in the context of procedural art

How does this relate to Procedural Art?

- You can generate content at run time, but Editor Scripting makes it possible to do the same at Editor time
- Editor time content can be saved, optimized and processed in different ways
  (eg think of light baking, occlusion baking, navmesh baking, etc)
- It is basically another tool in your toolbox, to do things that you couldn't do before or do things that you already did before but now easier.

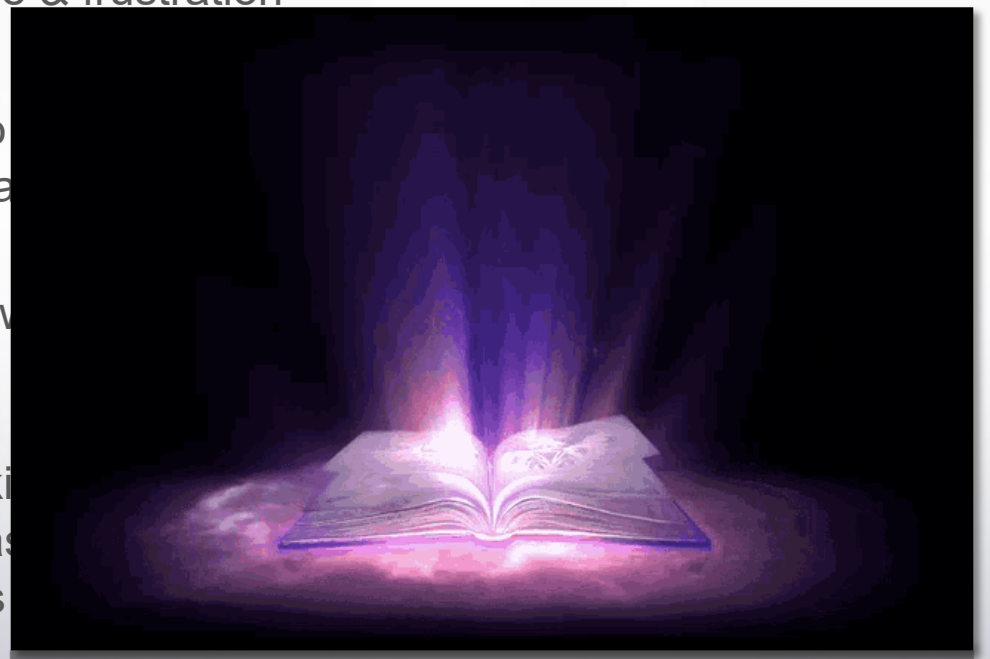
- Do we reaaally need it?
  - Well, not always, buuut...
    - as one tutorial put it: "Editor scripting is like a wheel barrow"
    - It is not absolutely required, but it sure is handy

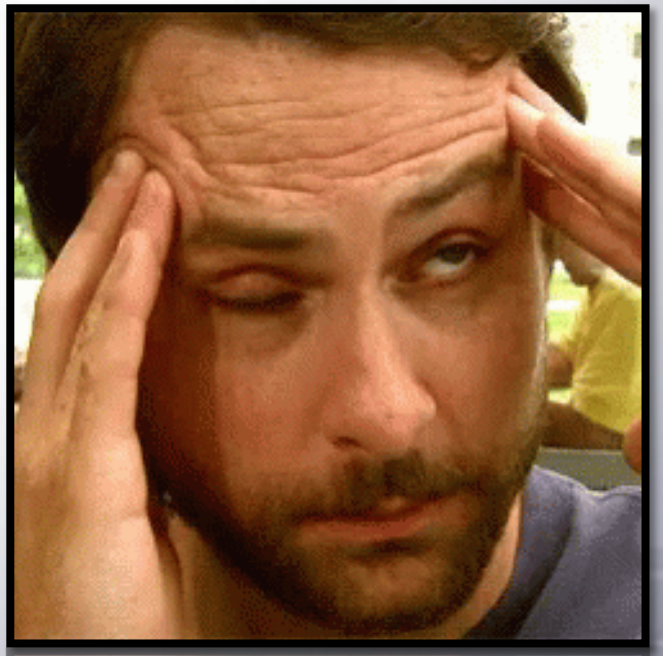
However, Editor Scripting comes at a 'price'

- Documentation is not always easy to find
- Layout can be a pain
- Unity keeps changing (target, targets, SerializedObjects, IMGUI, UIElements, to name a few)
- You often find new / better ways by accident or through Google instead of through the official Unity documentation

- In other words the *price* is: time & frustration

- Which means that you have to
  - Is the automation of the ta
- That depends for example on w
  - saves a lot of time
  - is going to be used often
  - will teach you valuable ski
  - can be sold (e.g. on the as
  - comes with other benefits





Luckily

- You probably only need Editor Scripting for limited purposes:
  1. Triggering code that would otherwise execute at runtime (e.g. placing objects)
  2. Processing your scene for optimization

- Setup for rest of the lecture:
  - Some general info on Editor Scripting (how do we start)
  - Cookbook approach: if you want this, you should do that ... (including Unity download)
  - Bunch of references at the end of this lecture

Editor Scripting Getting Started Part 1

- What makes a script an Editor script?
- What are Editor folders (good for) ?
- Your first Editor script

What makes a script an Editor script?

- An Editor script is *any* script that **extends** or **uses** *any* class from the **Editor namespace**

- A namespace is a collection of classes that can be imported through a *using* statement at the top of your code. It is a method to organize your classes. In Unity most of these namespace are defined in their corresponding dll's.

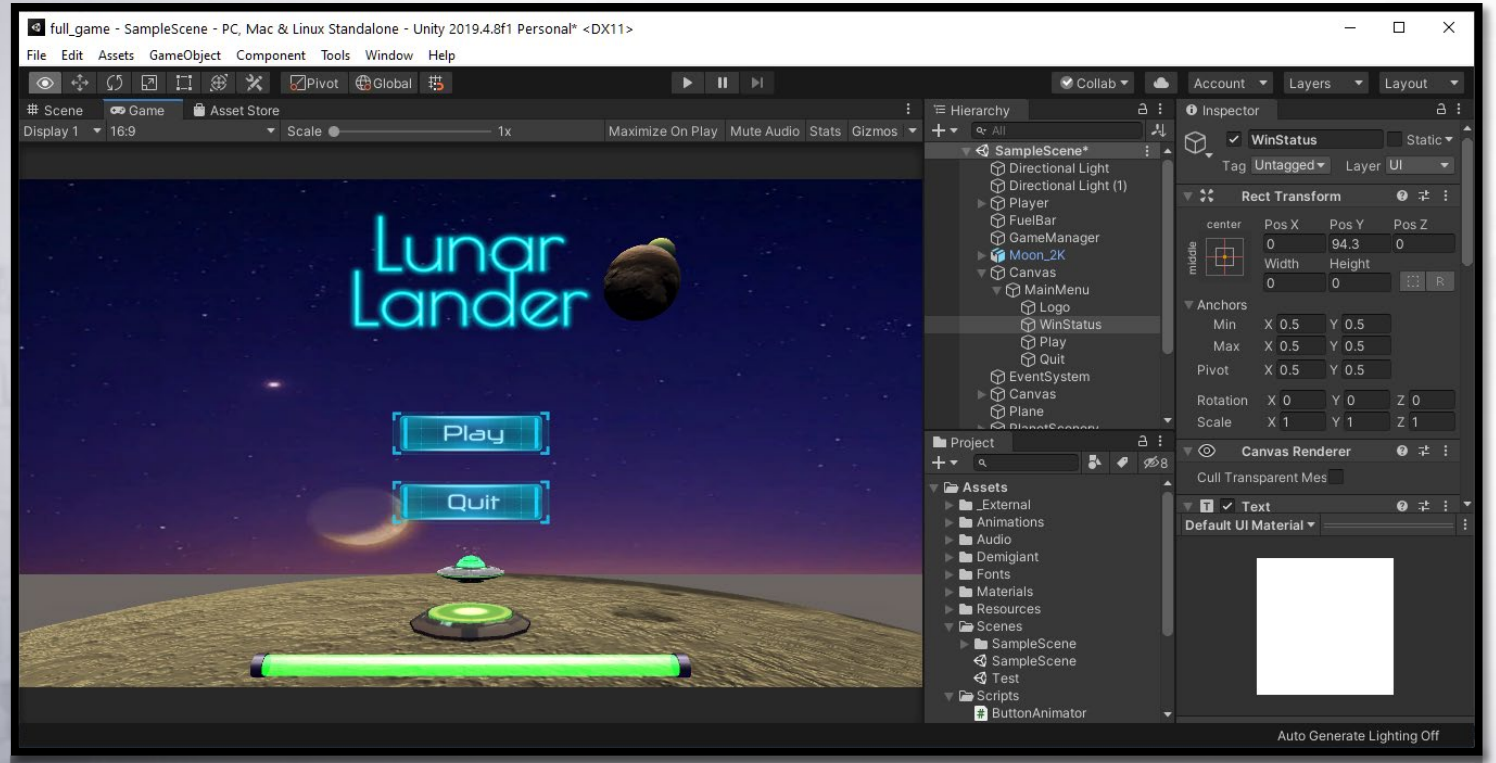
  For example:
  - using UnityEngine; ➔ UnityEngine.dll
  - using UnityEngine.UI; ➔ UnityEngineUI.dll
  - using UnityEditor; ➔ UnityEditor.dll

  (* check the data/managed folder of any of your builds)

Available Unity namespaces in the editor vs a build

In editor:

UnityEngine + UnityEditor



In build:

UnityEngine + ~~UnityEditor~~

But this has some implications for your project structure...

- *All* Editor scripts by definition use the UnityEditor namespace
- The UnityEditor namespace with all its classes is not available in your build
- Scripts using this namespace *cannot run in build and therefore must be prevented from being included in your build*

- **Unity's solution:**
  - **All** editor scripts **must** be in a folder called '**Editor**'
    (You can have as **many** Editor folders in your projects as you want in **any** location)
  - Unity automatically excludes *all* files in *all* Editor folders *from your build*

Recipe: Your first Editor Script / Menu Item

```
[MenuItem("My Custom Menu/Show a dialog _F1")]
private static void TheNameOfThisMethodDoesNotMatterButItMustBeStatic()
{
	EditorUtility.DisplayDialog (
		"Main menu item demo", "It works :)", "Awesome, thanks."
	);
}
```

- Things to try:
  - Put script in Editor folder, test and build
  - Put script outside of Editor folder, test and build → that is why you build early and often

As a side note: some other menu item possibilities

- Checkout Cookbook - MenuItems
  - Recipe 1: Different ways/locations where you can add menu items
  - Recipe 2: Menu item validation
  - Recipe 3: Menu item order
  - Recipe 4: Creating custom GameObjects using menu items
  - Recipe 5: Context menu items for (custom) components and custom component values
  - Recipe 6: Processing game objects in your scene (this demonstrates a lot of interesting LINQ)

- https://docs.unity3d.com/ScriptReference/MenuItem.html

- Used in for example:
  - The texture processor demo
  - To open additional windows with more functionality

Editor Scripting Getting Started Part 2

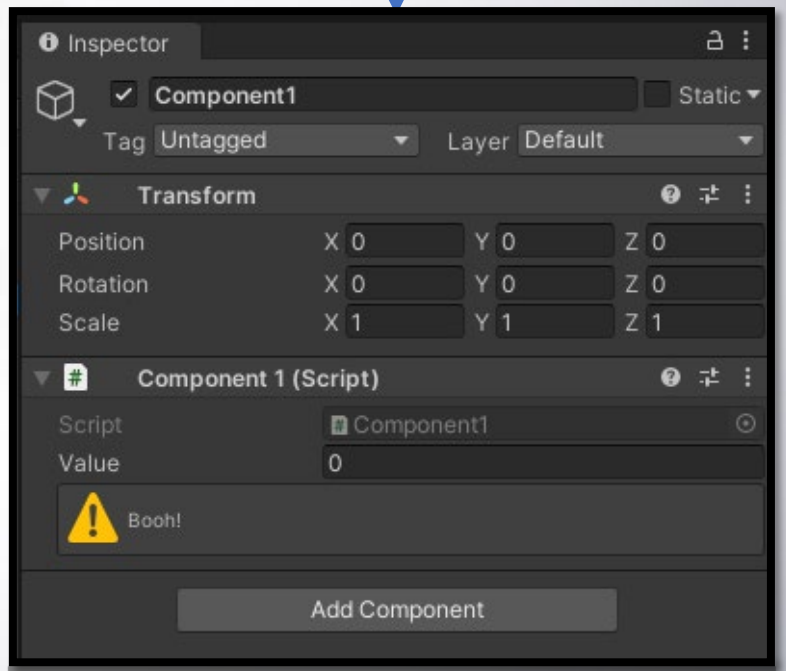
- Creating Custom Inspectors
- A look at some standard Editor options
- IMGUI vs Canvas vs UIElements
- GUI vs GUILayout vs EditorGUI vs EditorGUILayout

(all example code for this section can be found in Cookbook – CustomInspectors)

Custom inspectors

- Use case:
  - we have a custom Component1 ...
  - .. and we would like to change how this default inspector behaves

```
using UnityEngine;

public class Component1 : MonoBehaviour
{
        public int value = 0;
}
```

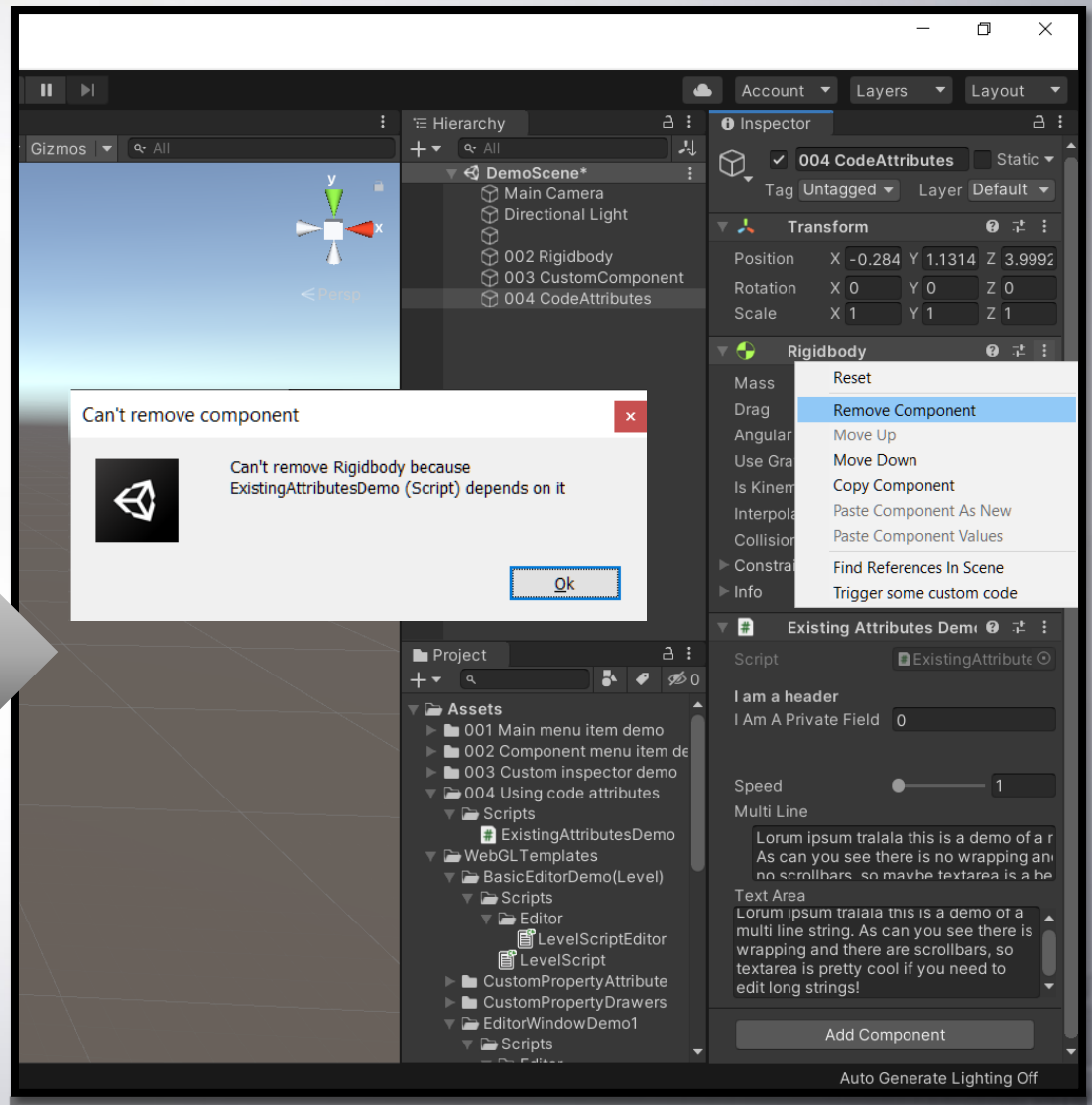


First off, let's make sure we do not reinvent the wheel ...

... because not all features require us to code custom editors and you can already do a lot with some standard Unity attributes:

P.s. [DisallowMultipleComponent] is also cool

```
using UnityEngine;

[DefaultExecutionOrder(-1)]
[RequireComponent(typeof(Rigidbody))]
public class ExistingAttributesDemo : MonoBehaviour
{
    [Header("I am a header")]

    [SerializeField]                    private float iAmAPrivateField;
    [HideInInspector]                   public float iAmAPublicField;

    [Space(30)]

    [Tooltip("Booh")] [Range(1, 100)]   public float speed;
    [Multiline(3)]                      public string multiLine;
    [TextArea(2, 5)]                    public string textArea;
}
```



... and some more standard features ...

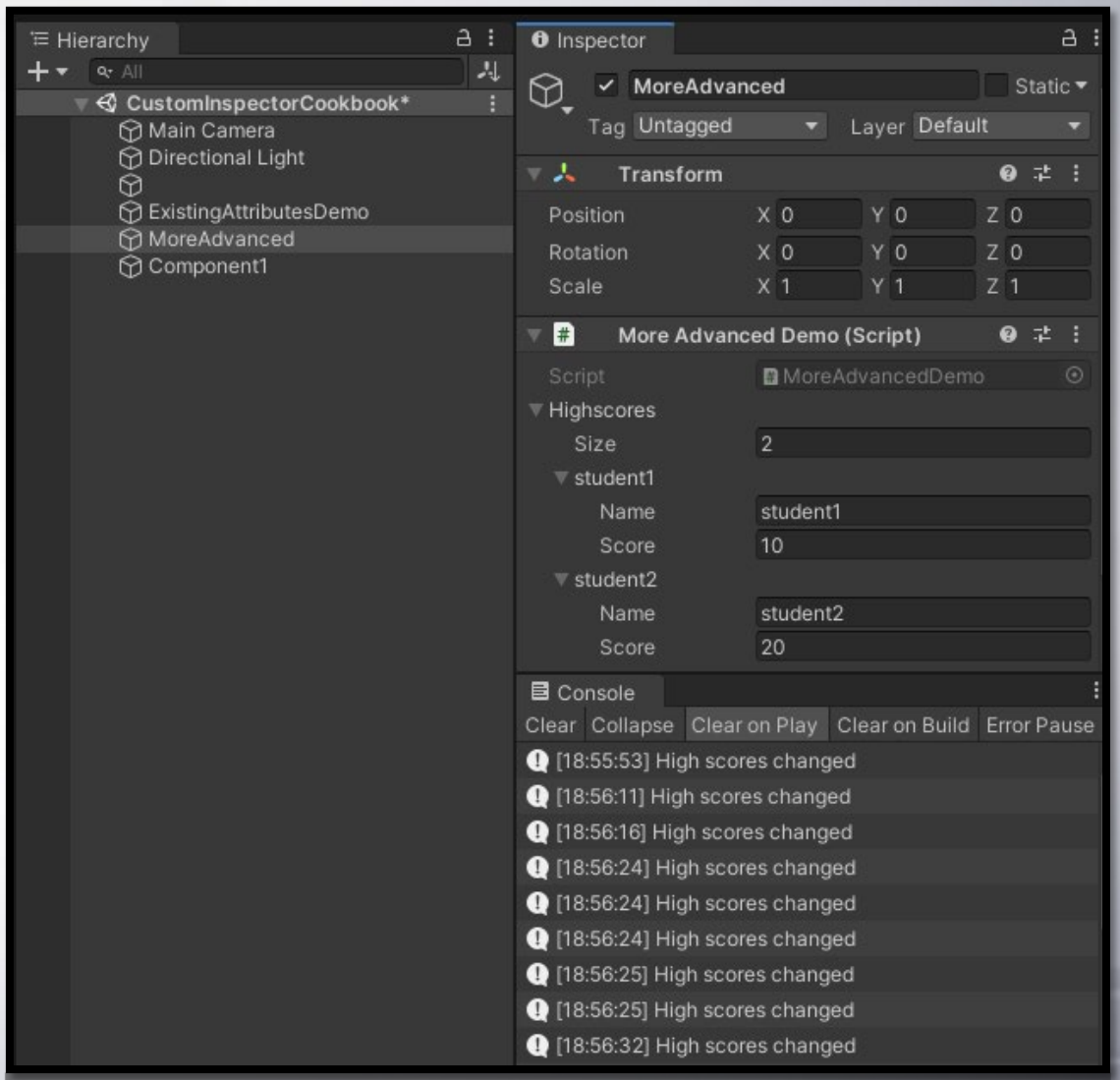
```
using UnityEngine;

[System.Serializable]
public class HighScore
{
    public string name;
    public int score;
}

public class MoreAdvancedDemo : MonoBehaviour
{
    public HighScore[] highscores;

    private void Reset()
    {
        highscores = null;
    }

    private void OnValidate()
    {
        Debug.Log("High scores changed");
    }
}
```

Custom types require a Serializable tag to show up in the inspector.

If the first class member is a string, it will be used as the element's description.

Reset is called when you choose the 'Reset' option from the component's context menu.

OnValidate is called when you change anything at all in the Inspector.

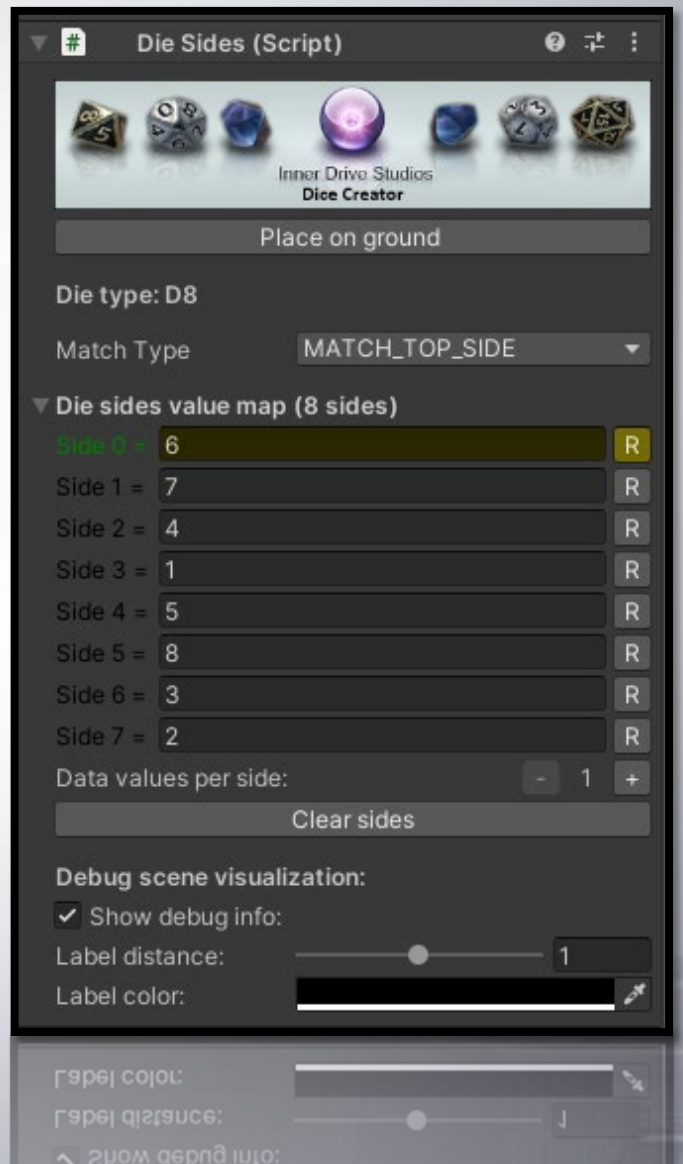


But what if this is not enough?

- For example, we might like to:
  - show an image
  - ***add some buttons to trigger some code***
  - add extra functionality like highlighting
  - build in custom validation
  - etc

- This requires inspector customization through:
  - a custom editor,
  - custom property drawers
  - or custom attributes

- Let's focus on custom *editors* for now



The basic structure of a custom Editor script:

```
using UnityEditor;

[CustomEditor(typeof(Component1))]
public class Editor1 : Editor
{
        public override void OnInspectorGUI()
        {
                base.OnInspectorGUI();
                //(same as DrawDefaultInspector();)
                EditorGUILayout.HelpBox("Booh!", MessageType.Warning);
        }
}
```

UnityEditor namespace required for Editor class (and other stuff ...)

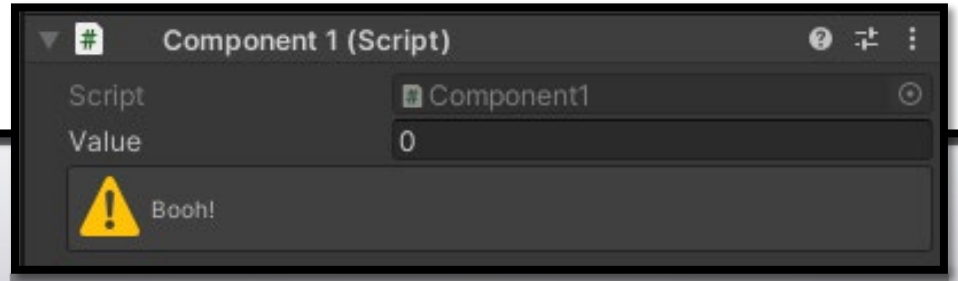
Which component this is a custom Editor for is specified through the CustomEditor class attribute.

The class name of your editor does not matter as long as your class extends the Editor class.

Override this method to implement your own UI.

Call this method to reuse the default UI.

One of the many methods you can call to add your own stuff.



How do we add different/new UI elements?

- There are three UI systems in unity, in order of appearance, make sure you look for the right documentation:
  - **IMGUI** (Immediate Mode Graphical User Interface):
    - implemented through the GUI, GUILayout, EditorGUI & EditorGUILayout classes
    - the oldest, useable directly from code in MonoBehaviour.OnGUI & Editor.OnInspectorGUI
    - works in game (GUI[Layout]) and editor (GUI[Layout] & EditorGUI[Layout])
  - **Canvas** (also called uGUI, found in UnityEngine.UI namespace):
    - in game UI development through drag/drop & code (game only)
  - **UIElements** (found in UnityEngine.UIElements namespace):
    - new retained GUI system with stylesheets, xml definitions etc

IMGUI (Immediate Mode Graphical User Interface):

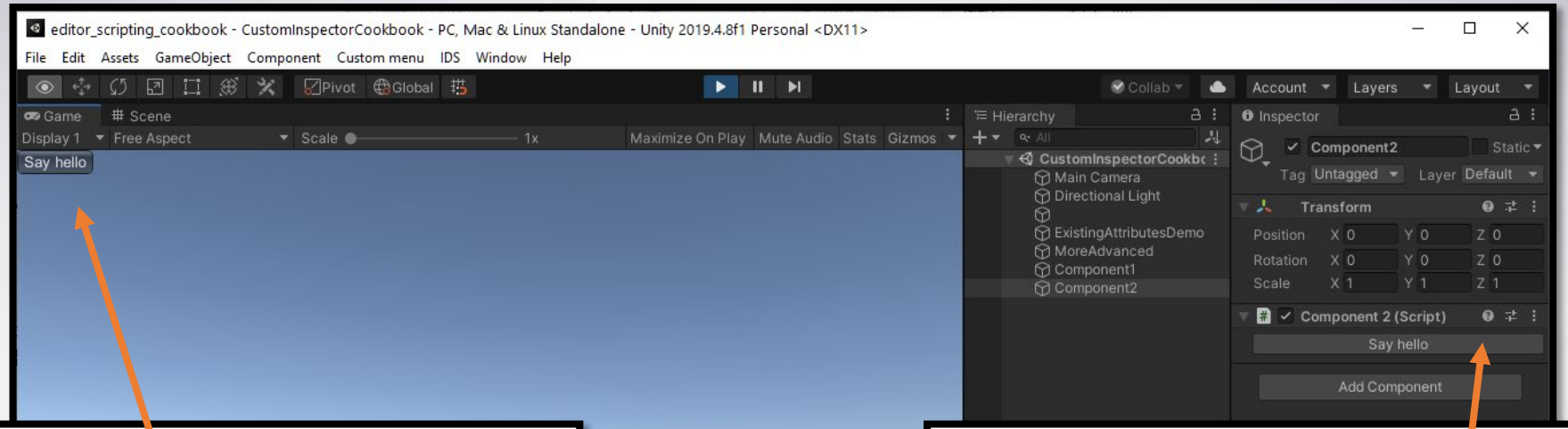
- The oldest UI, useable directly from code in for example OnGUI, OnInspectorGUI
- Works in game (GUI & GUILayout) and editor (GUI, GUILayout, EditorGUI, EditorGUILayout classes)

	Game & Editor (OnGUI)	Editor only (OnInspectorGUI)
Manual layout	GUI	EditorGUI
Auto layout	GUILayout	EditorGUILayout

- In most situations, you use either ***EditorGUILayout*** or ***GUILayout***, tinker, search, tinker some more.

- If needed, check the API documentation for these classes:
  - https://docs.unity3d.com/ScriptReference/GUI.html
  - https://docs.unity3d.com/ScriptReference/GUILayout.html
  - https://docs.unity3d.com/ScriptReference/EditorGUI.html
  - https://docs.unity3d.com/ScriptReference/EditorGUILayout.html

Adding a Button using IMGUI




```
using UnityEngine;

public class Component2 : MonoBehaviour
{
    private void OnGUI()
    {
        if (GUILayout.Button("Say hello"))
        {
            Debug.Log("Hello!");
        }
    }
}
```

```
using UnityEditor;
using UnityEngine;

[CustomEditor(typeof(Component2))]
public class Editor2 : Editor
{
    public override void OnInspectorGUI()
    {
        if (GUILayout.Button("Say hello"))
        {
            Debug.Log("Hello!");
        }
    }
}
```

Editor Scripting Getting Started Part 3

Accessing a component instance from your Editor script
using *target* and *serializedObject*

(all example code for this section can be found in Cookbook – CustomInspectors)

Accessing a component instance using the target property

```
using UnityEngine;

public class Component3 : MonoBehaviour
{
    public int value;
}
```

```
using UnityEditor;
using UnityEngine;

[CustomEditor(typeof(Component3))]
public class Editor3 : Editor
{
    public override void OnInspectorGUI()
    {
        Component3 myComponent = target as Component3;

        GUILayout.BeginHorizontal();

            GUILayout.Label("Value:");

            if(GUILayout.Button("-")) myComponent.value--;
            myComponent.value = EditorGUILayout.IntField(myComponent.value);
            if(GUILayout.Button("+")) myComponent.value++;

        GUILayout.EndHorizontal();
    }
}
```

Target property is available if every editor script and refers to an instance of the component you are editing.

Using target is easy and quick but has some caveats:

- changes are not saved
- there is no undo
- doesn't support multi object editing
- doesn't handle prefab overrides

There are ways to migitate this using:

- EditorGUI.BeginChangeCheck
- Editor-targets
- EditorSceneManager.MarkSceneDirty

BUT, there is a better way...

Accessing component values using serialized properties

- Serialized properties have a number of things already built in:
  - Undo
  - "Dirty-ing" the Scene
  - Multi object editing
  - Handling prefab overrides

- More info:
  - SerializedObject

- In other words:
  - more functionality out of the box at the cost of added complexity and more 'behind-the-scenes' stuff

```
using UnityEditor;

[CustomEditor(typeof(Component4)), CanEditMultipleObjects]
public class Editor4 : Editor
{
    public override void OnInspectorGUI()
    {
        serializedObject.Update();

        EditorGUILayout.PropertyField(
            serializedObject.FindProperty("value")
        );

        serializedObject.ApplyModifiedProperties();
    }
}
```

This expects a public property 'value' in your component.

```
}
    }
```

Summing up:

- In the beginning there was: target (and also targets)
- Both didn't 'suffice' so Unity added: new SerializedObject(target) and FindProperty
- new SerializedObject (target) was used so much that we now have: serializedObject (which is the same)

- Given the choice:
  - if you need to do something very ***simple*** without undo blahblahblah ***using target is fine***
  - if you need something more ***complicated***, ***consider using serializedObject***

- Downsides?
  - serializedObjects use string based reflection which makes spelling errors real easy
  - serializedObjects can do more, but are in themselves already more complicated to handle

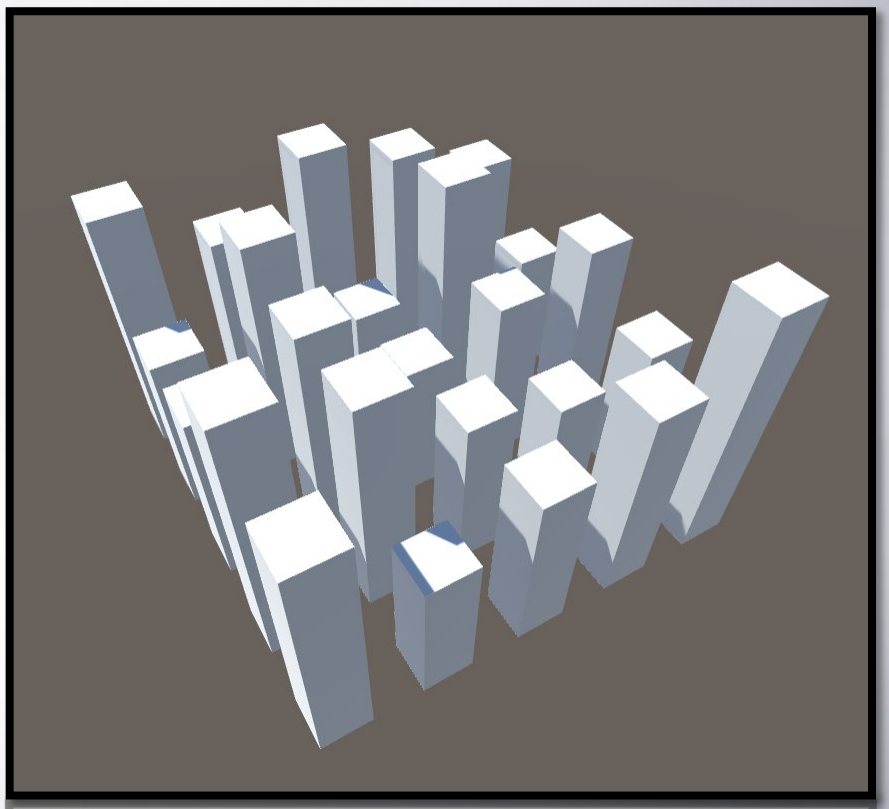
Editor Scripting Getting Started Part 4

Creating / triggering stuff from code at Editor time

(all example code for this section can be found in Cookbook – CustomInspectors)

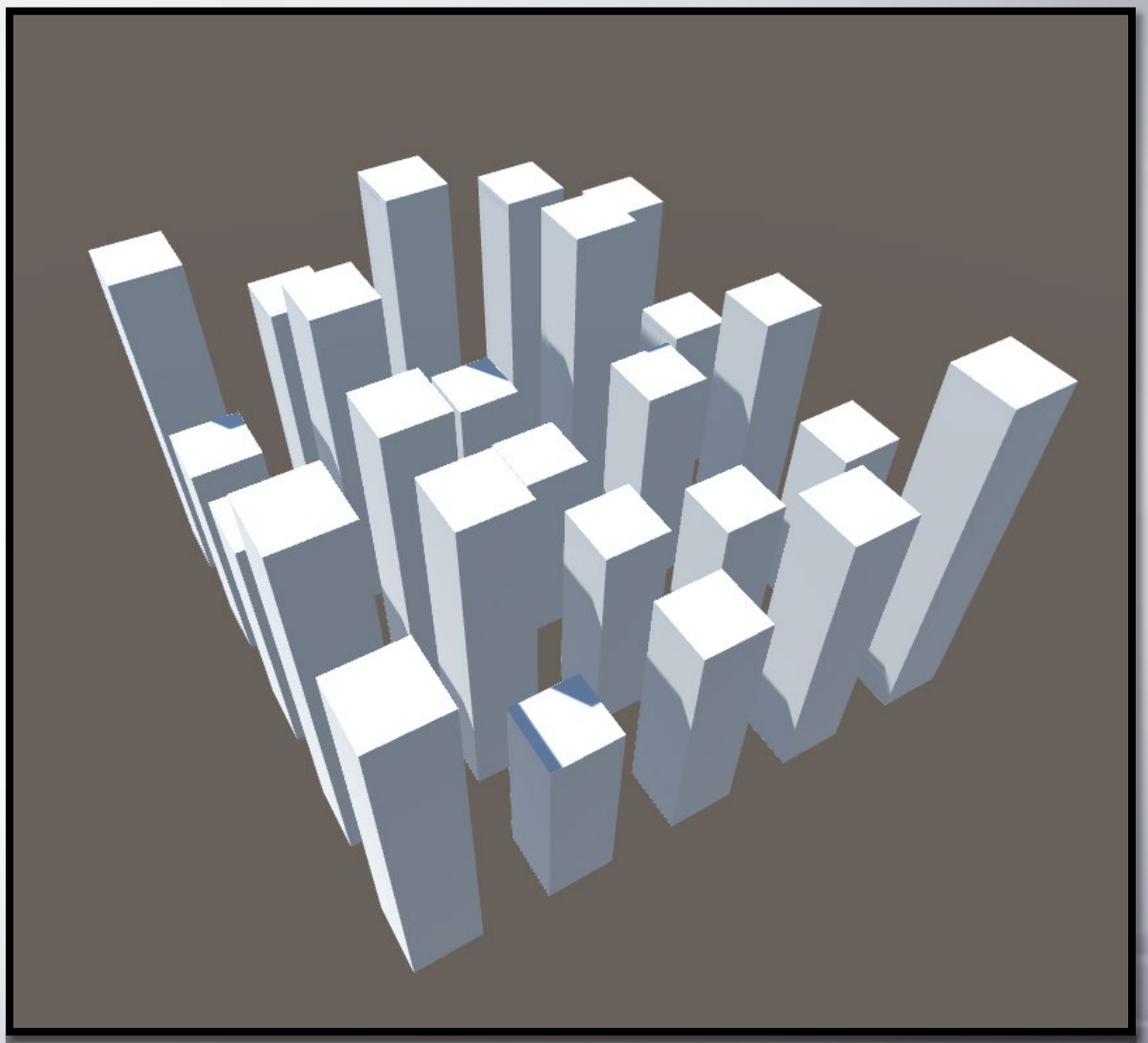
Creating stuff from code at Editor time

- Use case:
  - you have created a gameobject that already generates content in Awake
  - you want to trigger the creation of that content at Editor time, so you can process/tweak the results
- What are your (quickest) options?
  - copy and paste runtime content to the editor
  - use a [ContextMenu] tag in your component
  - use a custom component editor
- Check out ***RandomCubeSpawner1***



Things to note:

- Use of standard attributes (Range)
- Use of OnValidate()
- Editor & Runtime generation
- Use of ***,true*** in [CustomEditor] tag
- At least two issues:
  - Scene is not marked as changed
  - There is no undo/redo functionality



Changes and Undo

- Scene isn't being flagged as changed (aka dirty)
- Undo isn't working

- Solution:
  - `EditorSceneManager.MarkSceneDirty(EditorSceneManager.GetActiveScene());`
  - `Undo.RegisterCreatedObjectUndo(t.gameObject, "random cubes");`
    `(this last call automatically marks the scene as dirty)`

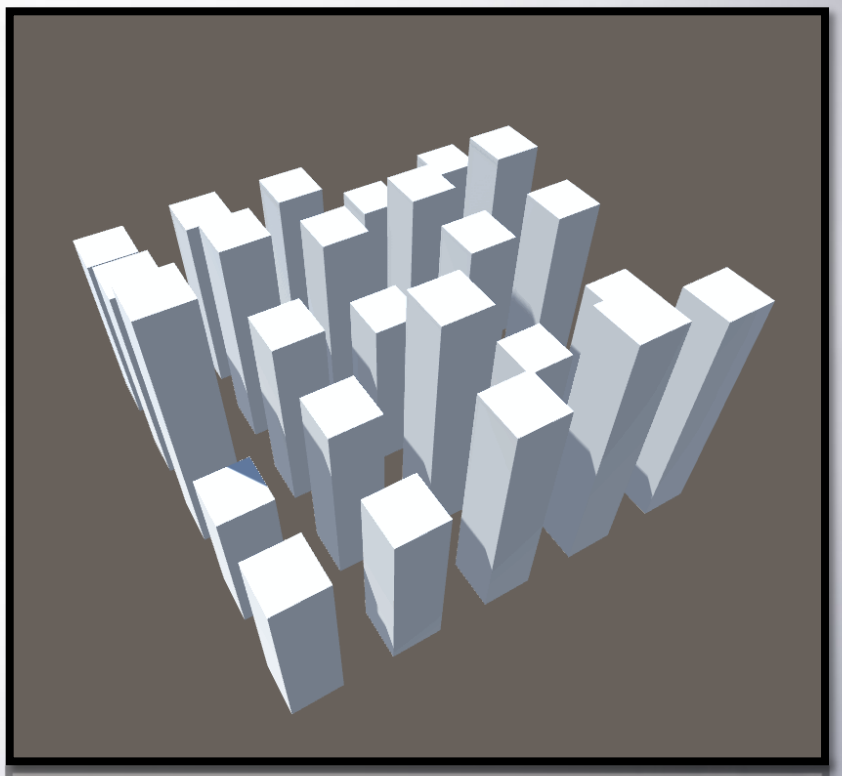
- However: these are *Editor* calls, which cannot be included in the build:
  - so either move all generation code to the Editor class (losing runtime generation)
  - or (easier) use compilation directives such as #if UNITY_EDITOR
    - Checkout ***RandomCubeSpawner2***
    - Try to create a build with and without these directives!

Regenerate immediately (RandomCubeSpawner3)

- If you want to be able to regenerate your scene immediately, you need to remove all generated objects first

- Easiest approach:
  - make sure all generated objects are children of your 'spawning' object
  - remove all child objects in a ***backwards*** loop using Destroy / DestroyImmediate based on whether the application is *playing* or not
  - Optionally include some more Undo functionality
  - Optionally include a 'spawnParent' → only remove 1

- Keep in mind that you don't **have** to add all these options!
- You can also choose for your algorithms to ***only work*** at either runtime OR editor time and WITHOUT Undo.



Purple pain, purple paihaaaain

- Unity only includes assets you are actually using in your scene at build time:
  - using editor time generation → everything is fine
  - using runtime generation → everything is purple

- Solutions:
  - generate everything at editor time ☺
  - include elements in your scene that reference the required resources (eg default cube)
  - use prefabs that reference your materials
  - use ScriptableObjects that ref your materials
  - add the required shader to always be included in the graphic settings

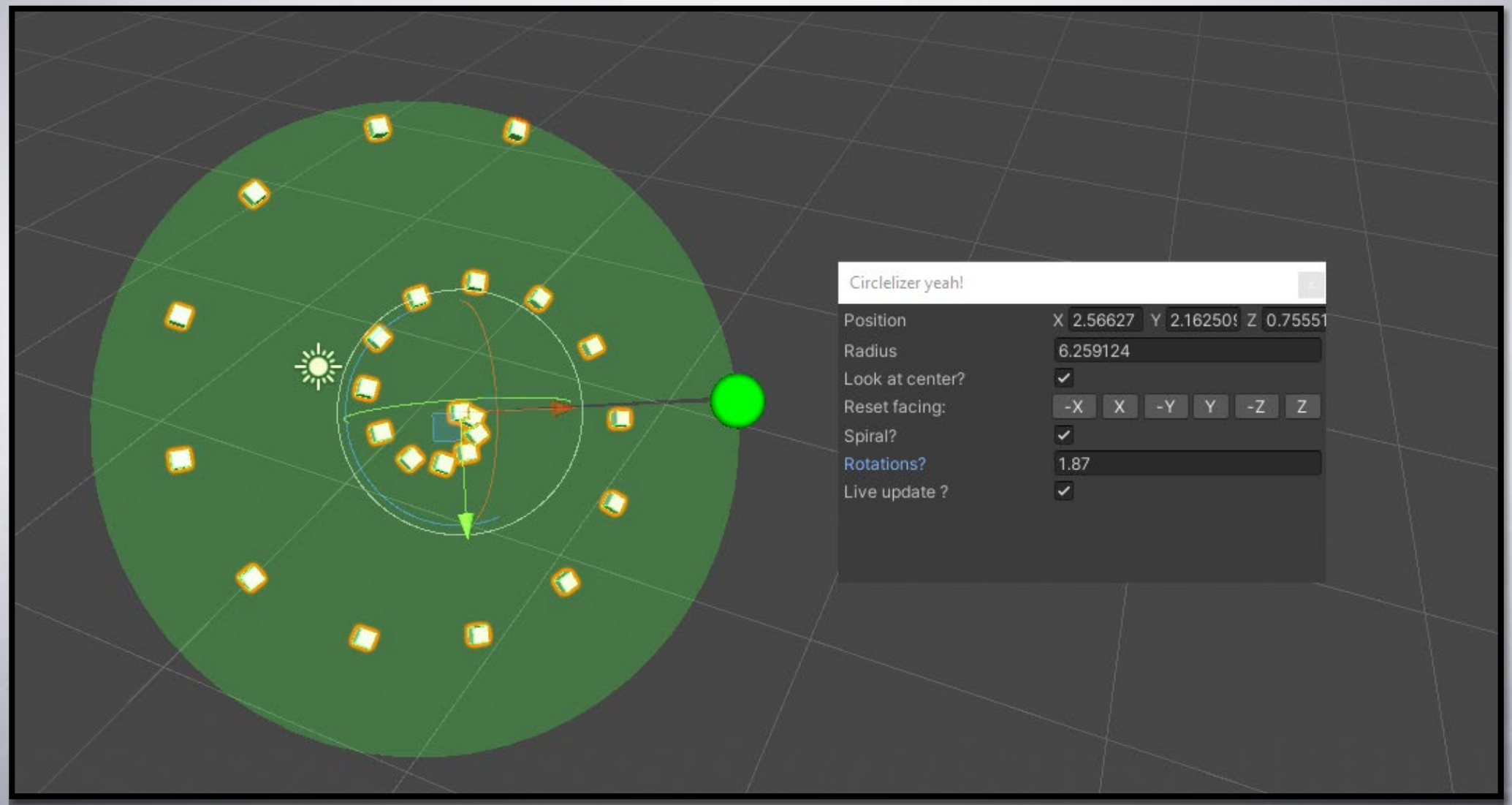
- Moral of the story: it doesn't matter which solution you choose as long as the content you are using is referenced somewhere, so it will be included in your build.

Editor Scripting Getting Started Part 5

Implementing custom windows

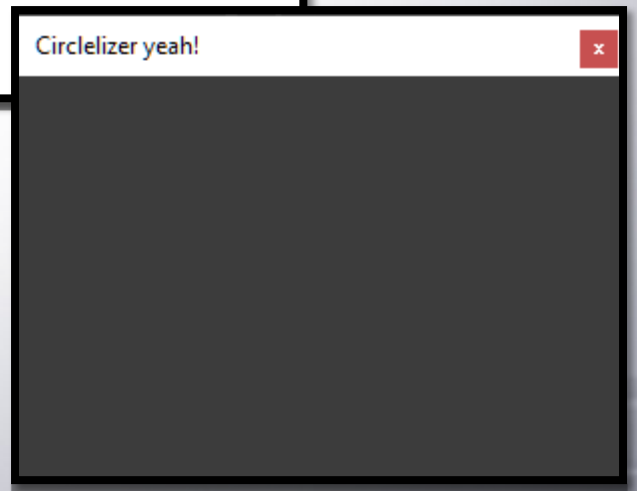
(all example code for this section can be found in Cookbook – CustomWindows)

Let's create ... a Circlelizer!! (totally made up words ftw!)



Step 1 – A window

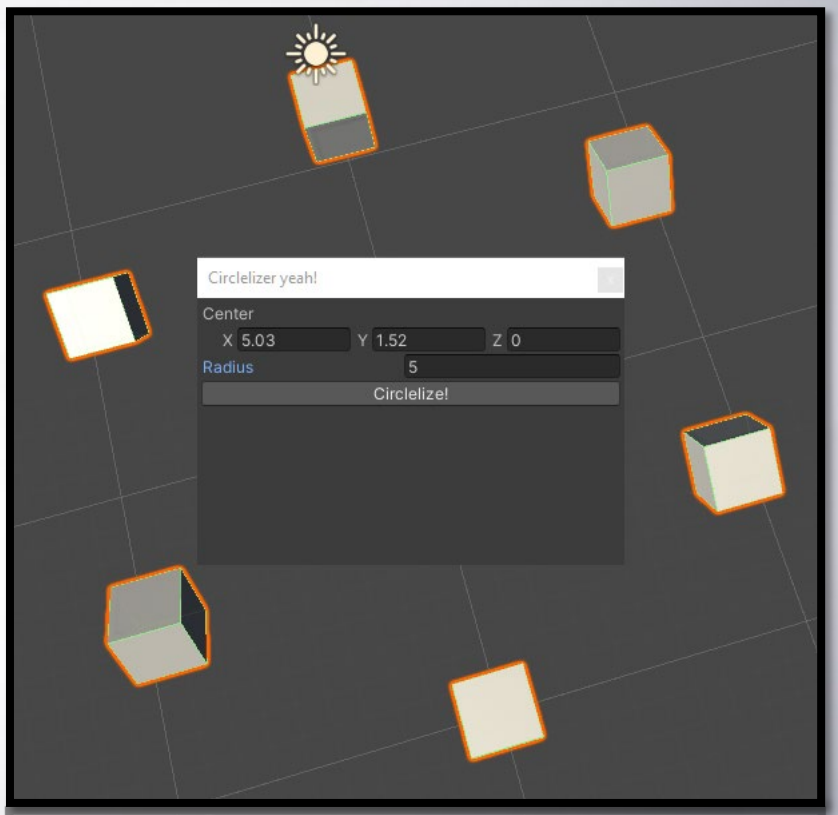
```
using UnityEditor;
using UnityEngine;

public class Circlelizer1 : EditorWindow
{
    [MenuItem("Custom menu/Cirlelizer yeah!")]
    private static void init()
    {
        Circlelizer1 window = GetWindow<Circlelizer1>("Circlelizer yeah!");
        window.minSize = window.maxSize = new Vector2(300, 200);
        window.Show();
    }
}
```



Step 2 – Circlelizer MVP (Minimum Viable Product)

```
private Vector3 _centerPoint = Vector3.zero;
private float _radius = 5;

private void OnGUI()
{
     _centerPoint = EditorGUILayout.Vector3Field("Center", _centerPoint);
     _radius = EditorGUILayout.FloatField("Radius", _radius);
     if (GUILayout.Button("Circlelize!")) updateLayout();
}

private void updateLayout()
{
     int count = Selection.gameObjects.Length;
     float angleStep = 2 * Mathf.PI / count;
     Quaternion facing = Quaternion.Euler(new Vector3(90,0,0));
     Vector3 right = facing * Vector3.right * _radius;
     Vector3 up = facing * Vector3.up * _radius;

     for (int i = 0; i < count; i++)
     {
          float angle = i * angleStep;
          Selection.gameObjects[i].transform.position =
          _centerPoint + Mathf.Cos(angle) * right + Mathf.Sin(angle) * up;
     }
}
```

Note the OnGUI instead of OnInspectorGUI



Evaluation – some possible improvements

- Position changing is cumbersome, filling in numbers, yuck who does that anyway?
- Cannot change the facing of the circle, since it is hardcoded at the moment
- No live updates and live updates roooooooooooooock!
- Selection might include assets instead of scene objects → no idea what would actually happen
- No convenience buttons for common orientations and resetting the position
- Settings are not saved → and remembering all those values is reaaaaally haaaaard

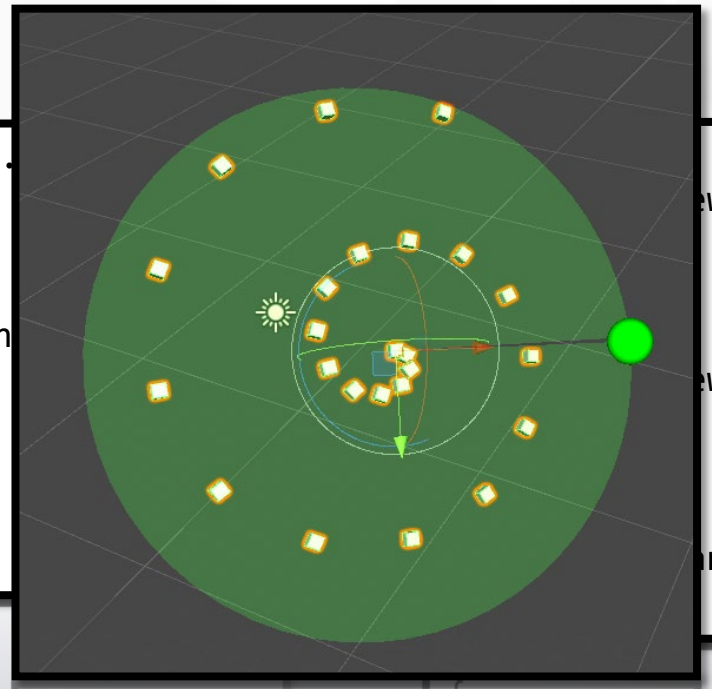
So how do we go ... from ← to →

User friendly position and orientation changes

- Entering a position or orientation (vector3 / quaternion) is not really user friendly
- Instead we can use the Handles class:
  - Handles is an editor class, meant to be used in OnSceneGUI:
  - https://docs.unity3d.com/ScriptReference/Handles.html
  - ... unfortunately EditorWindow does not have an OnSceneGUI ;)

Editor

```
private Vector3 somePosition = Vector3.

public void OnSceneGUI() {

     somePosition = Handles.PositionHan
          somePosition,
          Quaternion.identity
     );

}
```



EditorWindow

```
 OnEnable(){
w.duringSceneGui += onSceneGUI;


 OnDisable() {
w.duringSceneGui -= onSceneGUI;


 onSceneGUI(SceneView sceneView) {
ndles....
```

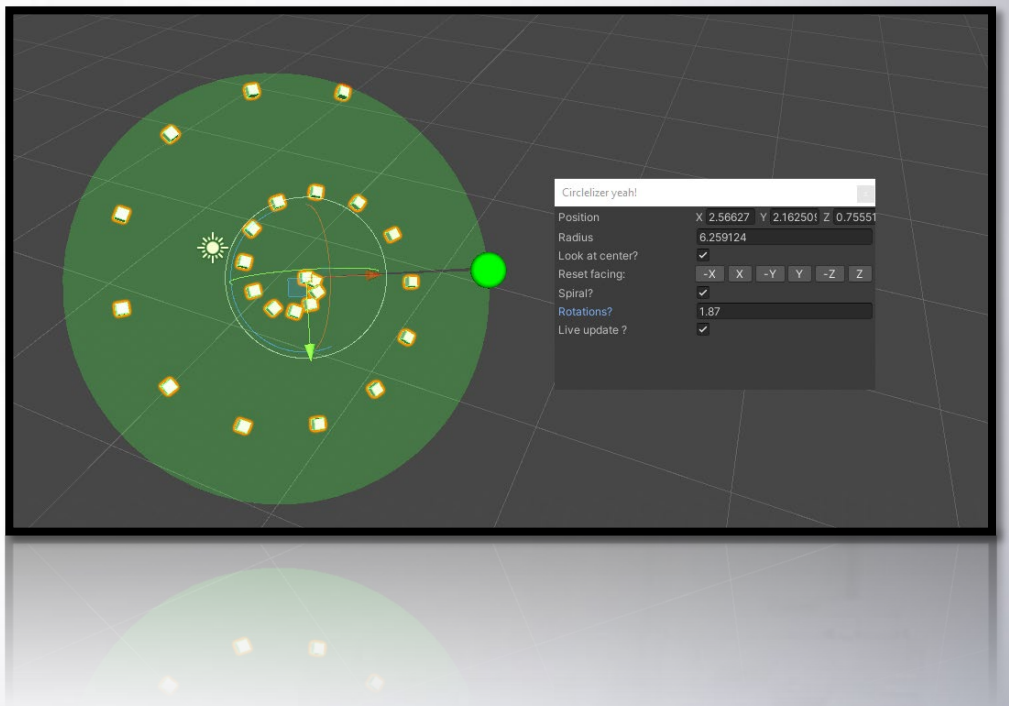
How about invalid selections?

- Selection.gameObjects can also contain prefabs from your Project Window → we don't want to change those
- How can we filter them out?
  - Check GameObject.scene.***IsValid***()
  - scene.IsValid() will only return true for GameObjects that are actually in the scene

Sample handles demo (Circlelizer3)

- The end result after some major tinkering:
  - Better UI including some more auto layout-ing
  - Handles in the sceneview such as:
    - RotationHandle
    - PositionHandle
    - Arcs
    - Sliders
  - Live updates etc
  - Some more math to create some interesting layouts (hint hint...)

- Interesting event handlers used:
  - OnEnable, OnDisable, OnSelectionChange, OnGUI, OnSceneGUI



Why are our settings not saved?

- By default Unity automatically (de)serializes (aka: difficult word for loads/saves) all public fields and any private fields marked with [SerializeField] from your ***Monobehaviour & ScriptableObject*** instances (more on this later)
- Any ***other*** fields that you add outside of these classes, e.g. public/private variables in your custom editor(window) are not saved.
- Long story short: you must save/load all data outside of your component/scriptable asset *yourself*, but how?

Saving / loading values in the Editor manually

- PlayerPrefs:
  - not such a good idea, is for player settings per project
  - (re)stores float, int, string
- EditorPrefs:
  - better, doesn't pollute playerprefs and is cross project, stored on disc
  - (re)stores float, int, string && bool (see what I did there?;))
- SessionState:
  - almost same as EditorPrefs, but not saved between sessions
  - (re)stores float, int, string, bool, int[], vector3
- JsonUtility:
  - Converts any serializable object to a string and vice versa, no storage by default
  - Also very good for SaveGames since you can serialize a whole SaveGame object easily
- ScriptableObject:
  - Can store any unity type such as prefabs, great tool to simplify your project and designer friendly.
  - Little bit more cumbersome to get to work in an EditorWindow, but each value is automatically saved.

JSON (JavaScript Object Notation)

```
{
    "potion_name" : "magic potion",
    "value" : 100,
    "effect_name" : "healing",
    "description" : "heal your wound"
}
```

Saving settings: so which one do we pick?

- Simple values only ? → Use EditorPrefs/SessionState optionally in combination with JSONUtility
- Complex values, e.g. prefabs and other references → Use a ScriptableObject

- EditorPrefs/JSONUtility example → CustomWindows/Circlelizer4
- ScriptableObject example → CustomWindows/PrefabSpawner

Conclusions and references

Summing up:

- Editor scripting is cool, but documentation is far from clear, small simple things can take a lot of research (*cough* google *cough*)

- If you apply editor scripting in your project:
  - Keep it simple
  - If solution A works but yeah solution B would be more awes... NO! STOP IT! SOLUTION A IS FINE!
  - Reuse elements from the examples in the "cookbook“, it should take you a long way!

- If you find out cool new tricks, let us know during lab, we'd love to see 'm!

Some tutorials & references (... but only a small part ...)

- Tutorials:
  - Keep in mind some of the tutorials use outdated/cumbersome approaches here and there
  - https://learn.unity.com/tutorial/editor-scripting
  - https://learn.unity.com/tutorial/introduction-to-editor-scripting
  - https://learn.unity.com/tutorial/scripting-custom-editor-windows-and-inspectors
  - https://learn.unity.com/tutorial/introduction-to-scriptable-objects
  - https://www.patrykgalach.com/2019/09/12/custom-scriptable-wizard/
  - https://medium.com/@prasetion/saving-data-as-json-in-unity-4419042d1334
  - https://www.youtube.com/watch?v=aPXvoWVabPY (Brackeys scriptable objects)

- Unity manual:
  - https://docs.unity3d.com/Manual/ExtendingTheEditor.html
  - https://docs.unity3d.com/ScriptReference/EditorWindow.html
  - https://docs.unity3d.com/Manual/JSONSerialization.html
  - https://docs.unity3d.com/Manual/class-ScriptableObject.html
  - ...and much much more (basically any keyword/methodcall that you see in the code)