VRDC

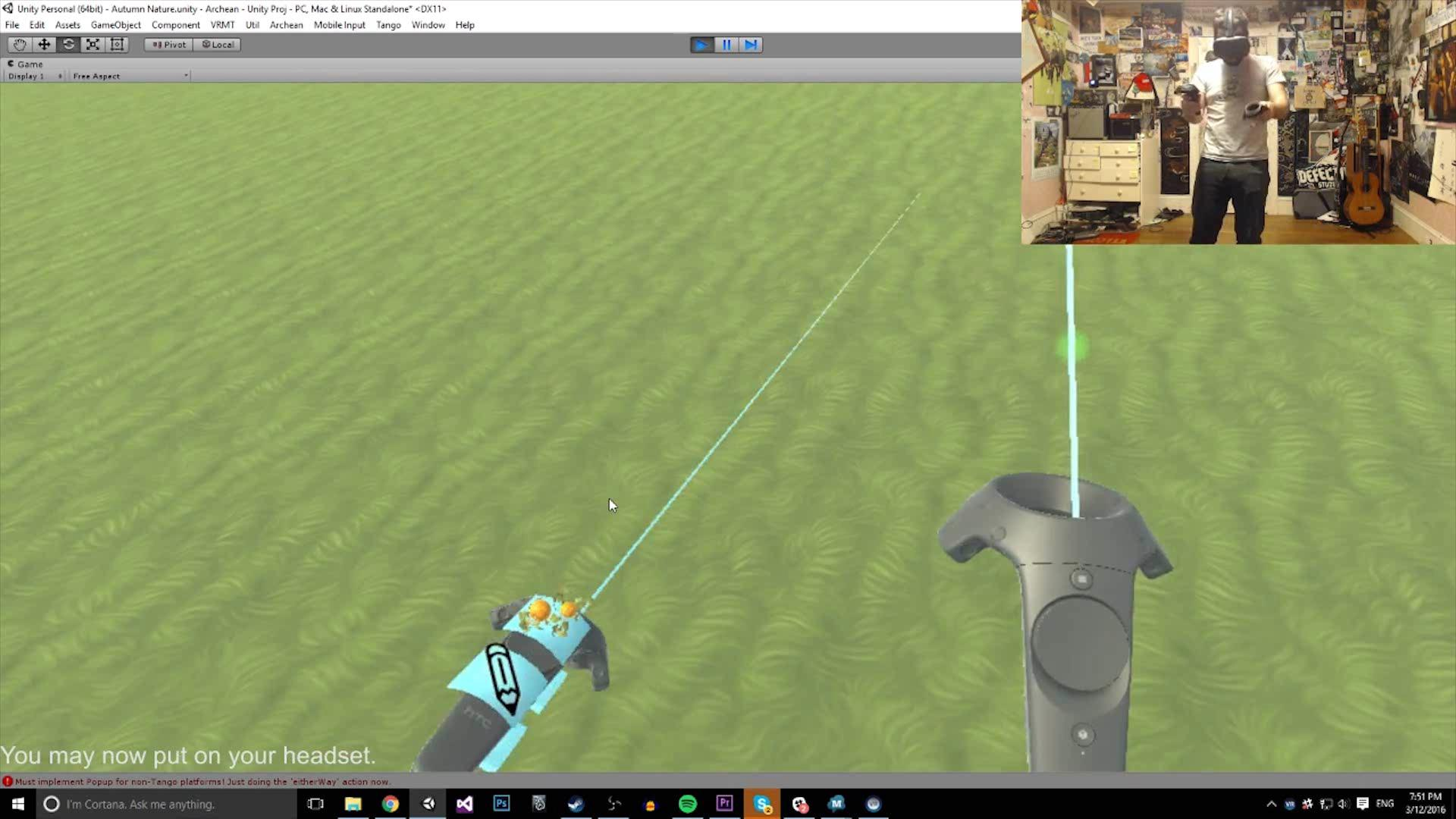
Architecting Archean

Simultaneously Building For Room-Scale and Mobile VR, AR, and All Input Devices

Jono Forbes
Archean

Hello! I’m Jono.

- Many hats, mostly code, design, & 3D
- All the hats on Archean
- Developer & co-founder at Defective Studios
- Boston Unity Group & Boston VR co-organizer


Unity Personal (64bit) - Autumn Nature.unity - Archean - Unity Proj - PC, Mac & Linux Standalone* <DX11>
File Edit Assets GameObject Component VRMT Util Archean Mobile Input Tango Window Help
Pivot Local
Game
Display 1 Free Aspect
You may now put on your headset.
Must implement Popup for non-Tango platforms! Just doing the 'eitherWay' action now.
I'm Cortana. Ask me anything.
ENG
7:51 PM
3/12/2016

Implement All The Things!

- Hardware rundown
- Supporting everything (in Archean)
  - Abstracting input from SDKs
  - Managing SDKs
  - UI framework

VRDC

Hardware Rundown

Headsets

Controllers

Wands

Optical

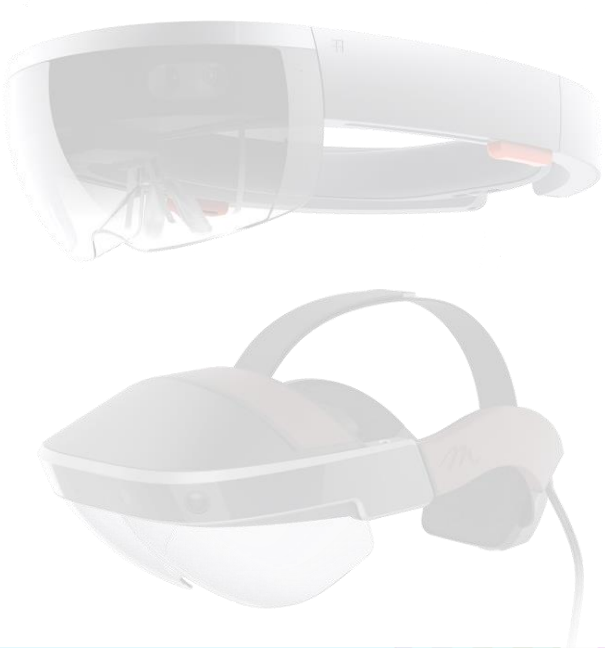
Mobile

Traditional

AR

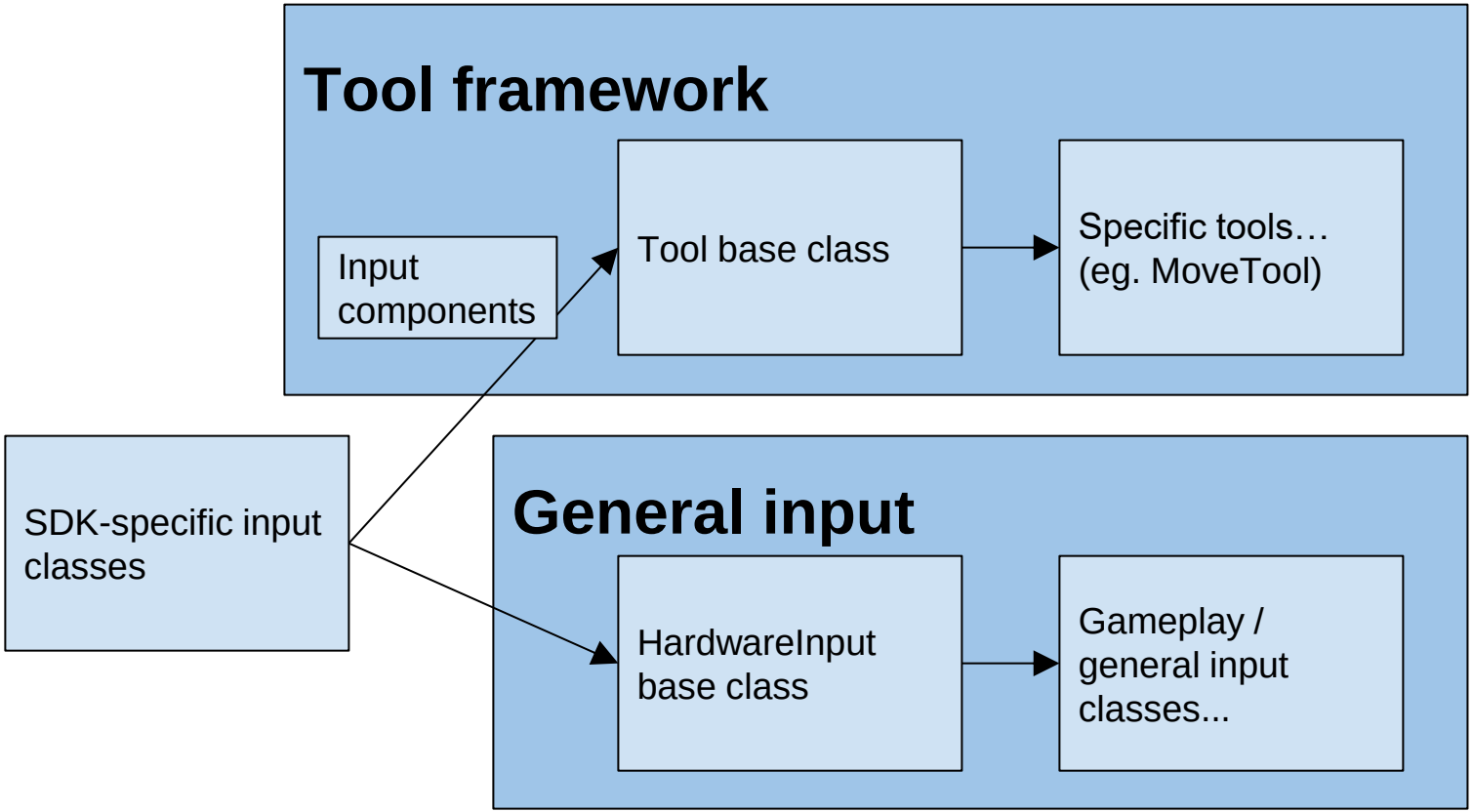
Handsets

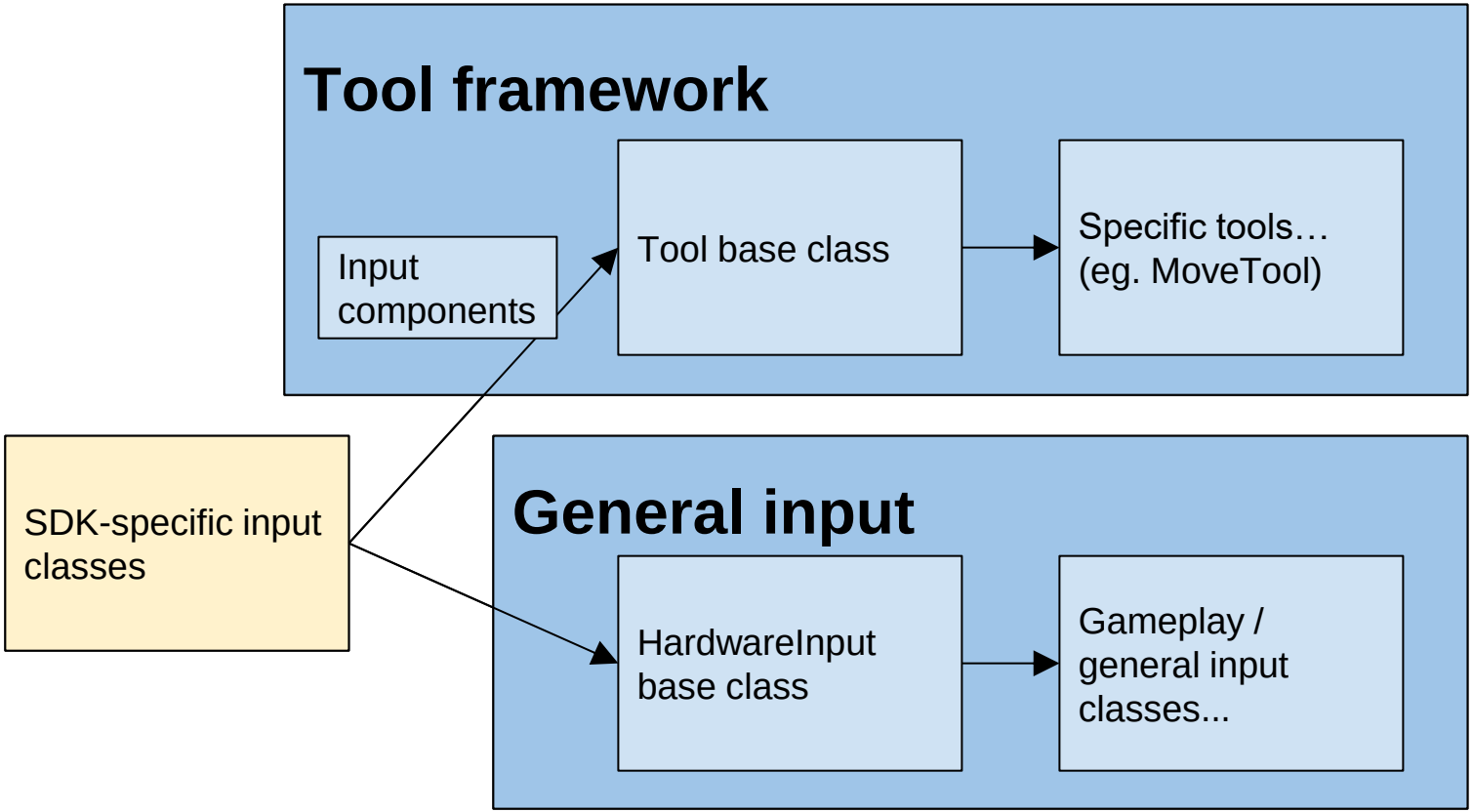
Headsets



VRDC

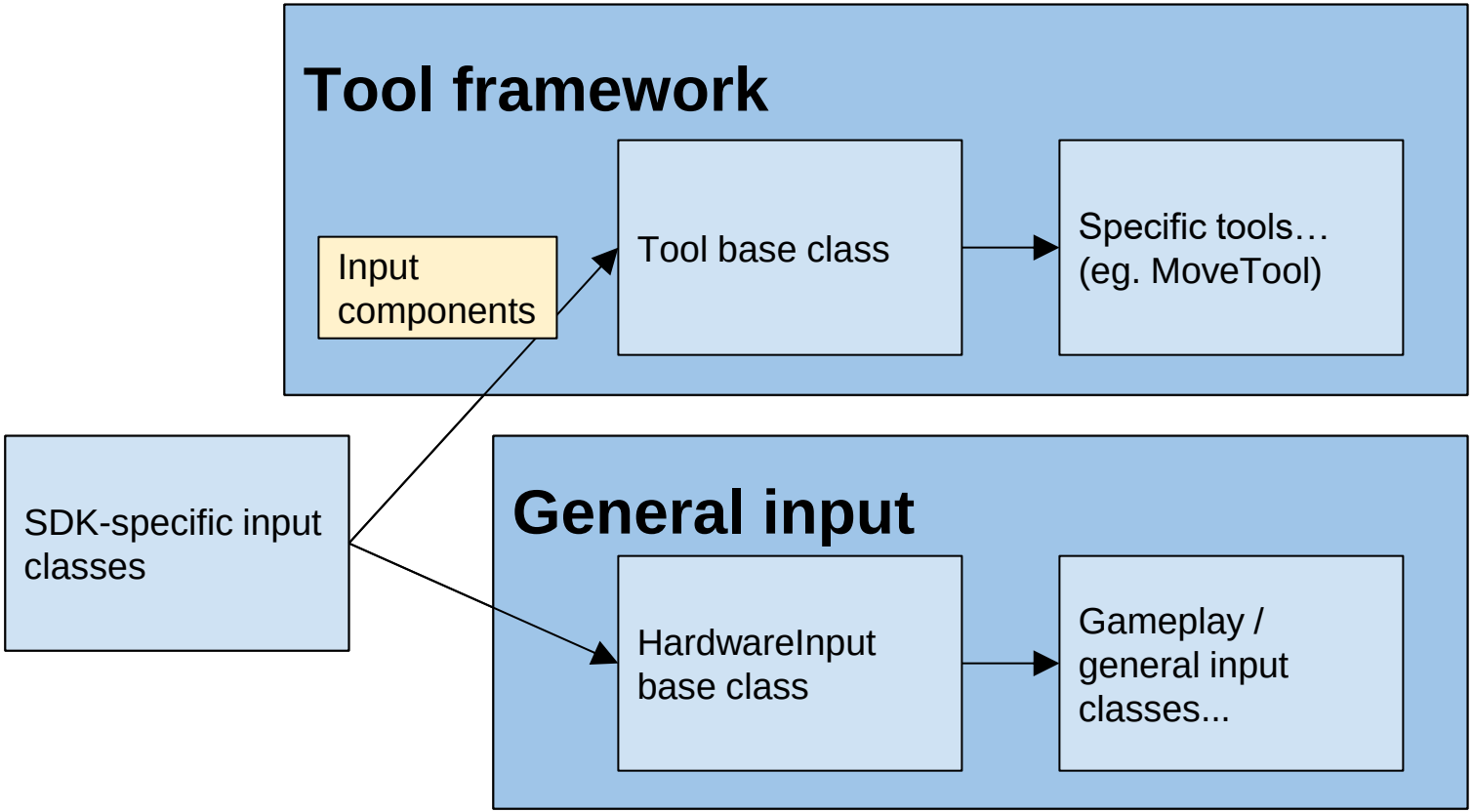
Taming the beast of multiplatform input (in Archean (in Unity/C#))


Tool framework
Input components
Tool base class
Specific tools… (eg. MoveTool)
SDK-specific input classes
General input
HardwareInput base class
Gameplay / general input classes...


Tool framework
Input components
Tool base class
Specific tools… (eg. MoveTool)
SDK-specific input classes
General input
HardwareInput base class
Gameplay / general input classes...

SDK-specific input classes

- One class per piece of hardware / SDK
  - **No** project-specific logic!
  - Listen for device input, call abstract handlers
    - Call Tool down/held/up
    - Call general input down/held/up


Tool framework
Input components
Tool base class
Specific tools… (eg. MoveTool)
SDK-specific input classes
General input
HardwareInput base class
Gameplay / general input classes...

Input Components

SDK-specific components classes contain specific references to hardware features

```
public class ViveControllerComponents : WandComponents {
        public SteamVR_Controller.Device viveController;
}
```

Input Components

Category-specific component classes contain common attributes for hardware types

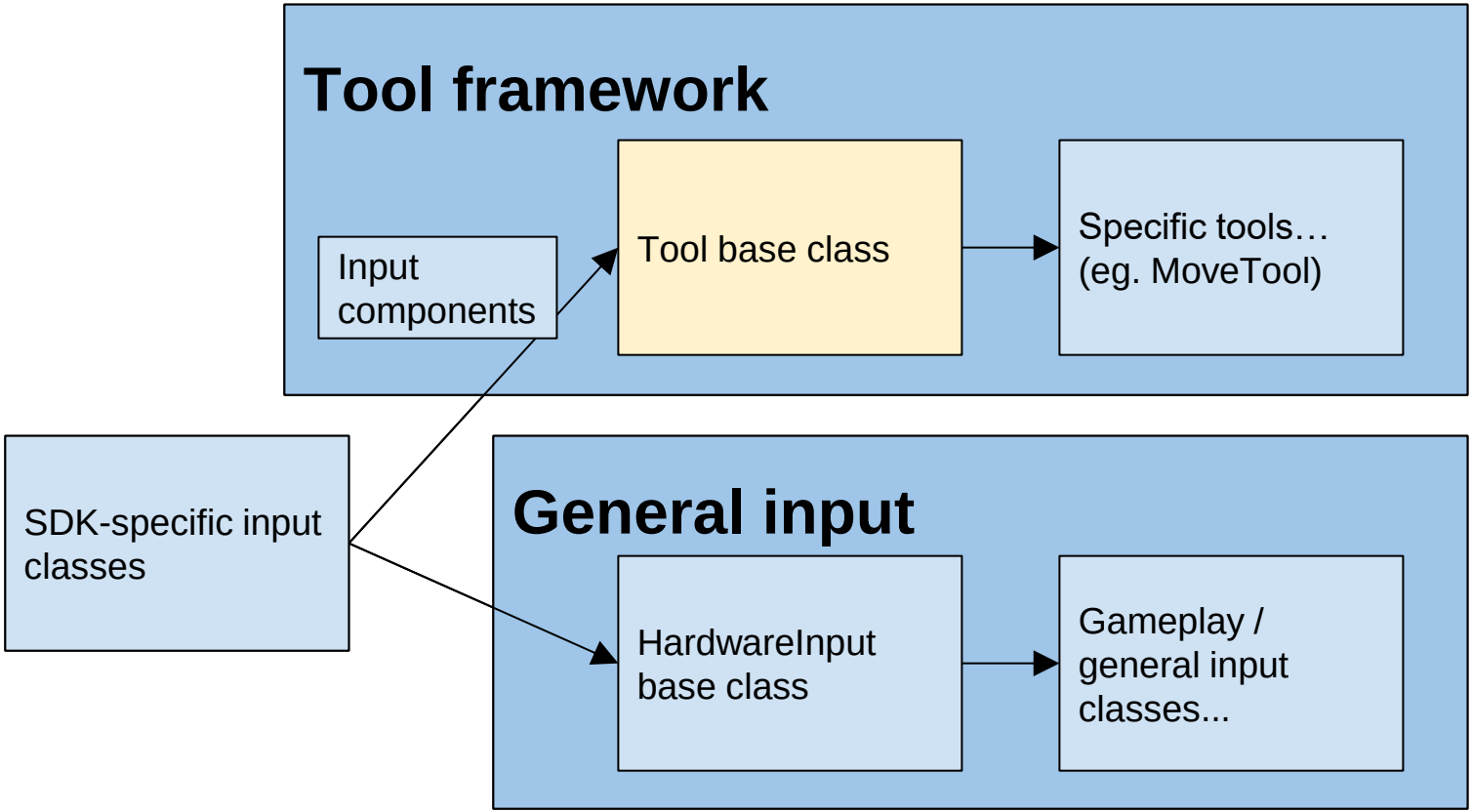
```
public class WandComponents : InputComponents {
    public Transform handTrans;
    public override Vector3 Position { get { return handTrans.position; } }
    public override Vector3 Forward { get { return handTrans.forward; } }
    public override Quaternion Rotation {get{ return handTrans.rotation; }}
}
```

Input Components

InputComponents base class contains most abstract data

```
public class InputComponents {
    public virtual bool Valid { get { return true; } }
    public virtual Vector3 Position { get { return Vector3.zero; } }
    public virtual Vector3 Forward { get { return Vector3.forward; } }
    public virtual Quaternion Rotation { get { return Quaternion.identity; } }
}
```

(Any non-tracked input uses head for position / rotation)


Tool framework
Input components
Tool base class
Specific tools… (eg. MoveTool)
SDK-specific input classes
General input
HardwareInput base class
Gameplay / general input classes...

Tool base class

Down/Held/Up hooks per SDK

```
public virtual void DoToolDown_Sixense(SixenseComponents sxComponents) {
    DoToolDown_Wand(sxComponents); }

public virtual void DoToolDown_Leap(LeapComponents leapComponents) {
    DoToolDown_Optical(leapComponents); }

public virtual void DoToolDown_Tango(TangoComponents tangoComponents) {
    DoToolDown_PointCloud(tangoComponents); }
```

Tool base class

Down/Held/Up hooks per Category

```
public virtual void DoToolDown_Wand(WandComponents wandComponents) {
    DoToolDown_Core(wandComponents); }

public virtual void DoToolDown_Optical(OpticalComponents opticalComps) {
    DoToolDown_Core(opticalComps); }

public virtual void DoToolDown_PointCloud(PCComponents pcComponents) {
    DoToolDown_Core(pcComponents); }
```

Tool base class

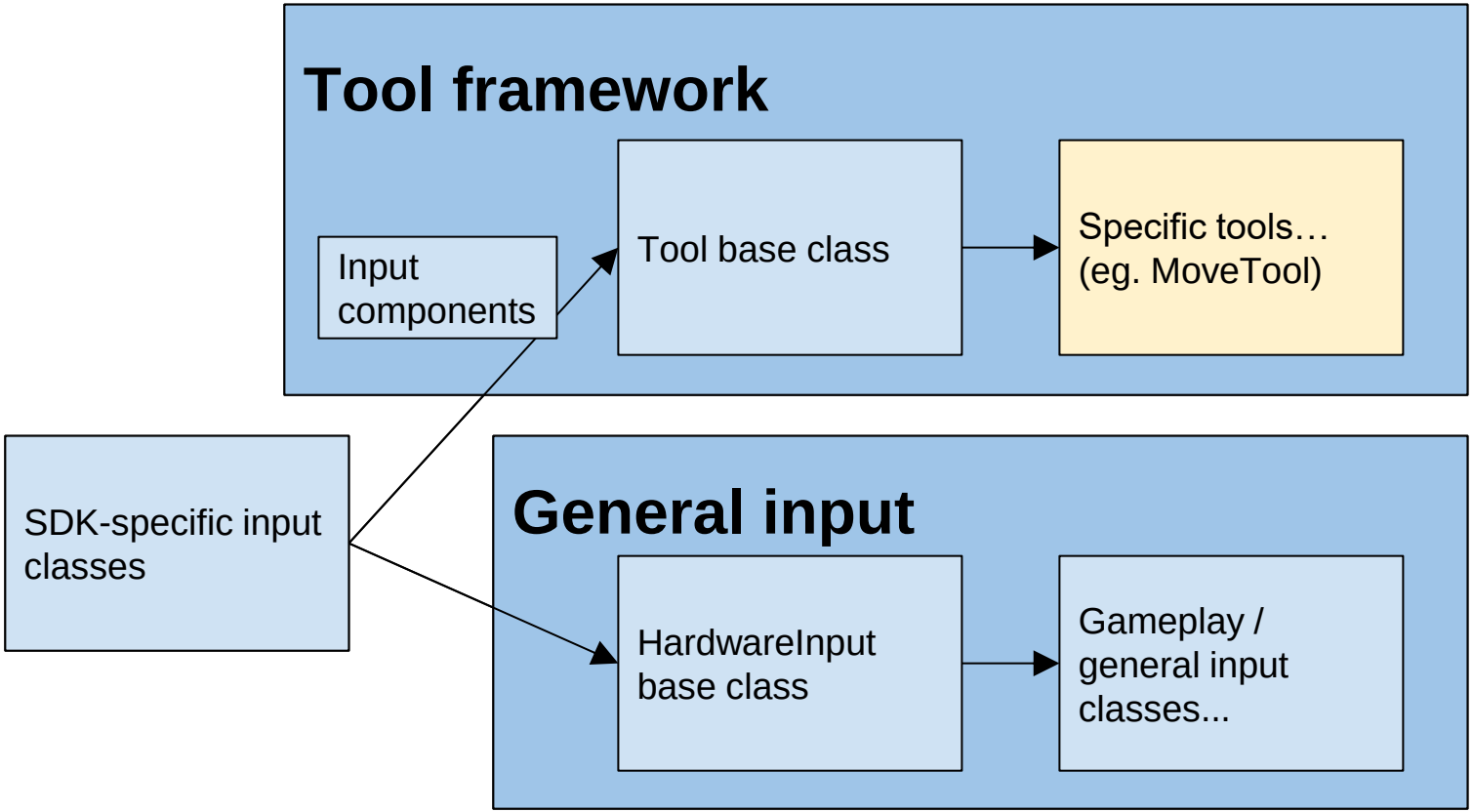
Tool base functions

- `DoToolDown_Core`
  - `DoToolDownAndHit*`
- `DoToolHeld_Core`
- `DoToolUp_Core`
- `DoToolDisplay_Core`

Tool base class

*DoToolDownAndHit:

```
public virtual void DoToolDown_Core(InputComponents comp) {
    if (Physics.Raycast(comp.Position, comp.Forward, out hit, dist, layers))
    {
        DoToolDownAndHit(comp);
    }
}
```


Tool framework
Input components
Tool base class
Specific tools… (eg. MoveTool)
SDK-specific input classes
General input
HardwareInput base class
Gameplay / general input classes...

Specific Tool class

Can override DoToolDown_Core etc for fully platform-agnostic logic

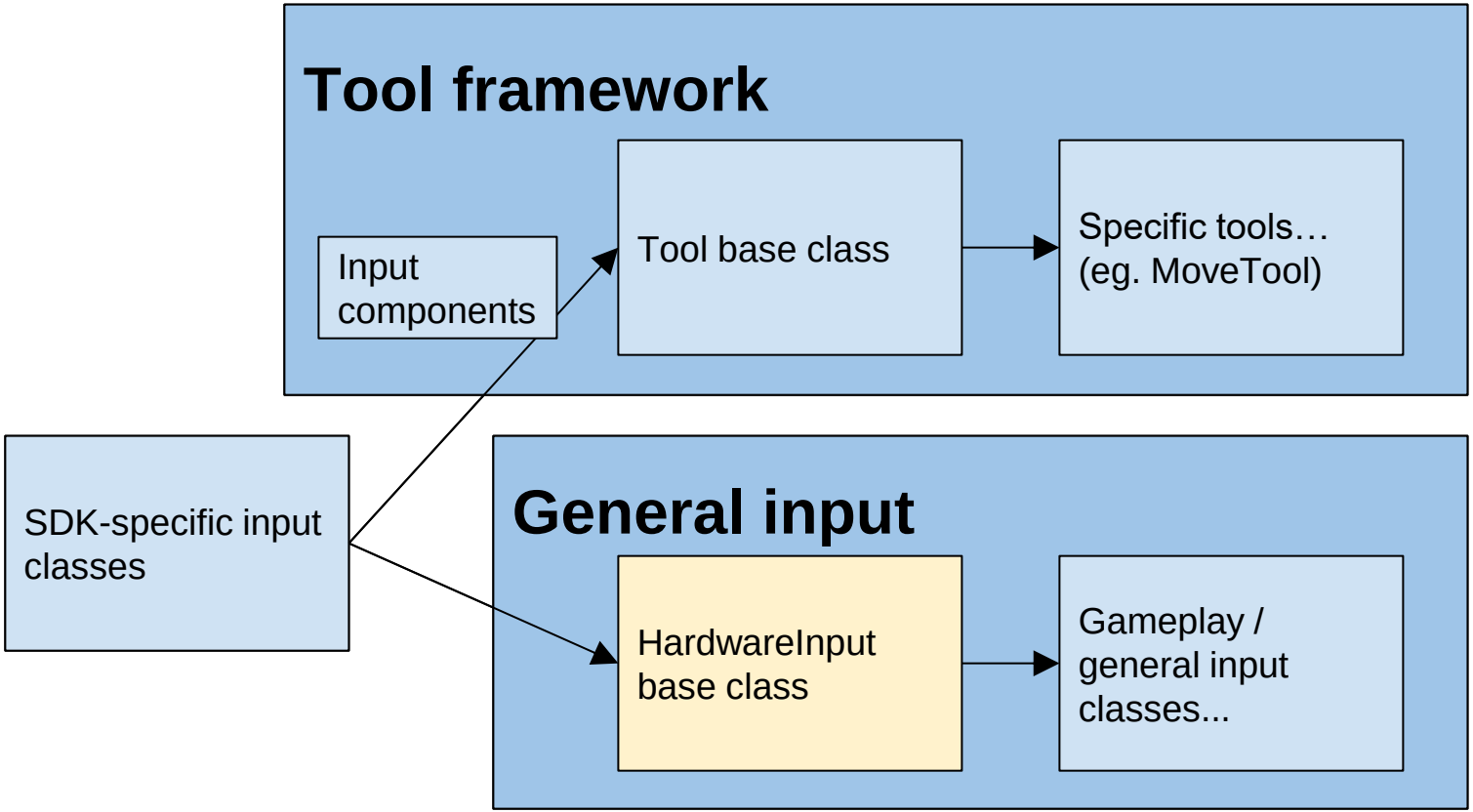
```
public class MoveTool : Tool {
    protected override void DoToolHeldAndHit(InputComps comps) {
        selectedTrans.position = hit.point;
    }
}
```

^ This works on Vive, Cardboard, Tango…!

Specific Tool class

Can override any category- or SDK-specific hook for more customized behavior

```
public class MoveTool : Tool {
    // ...
    protected override void DoToolHeld_Optical(OpticalComps comps) {
        // Move mechanic that's more appropriate for optical control
    }
}
```

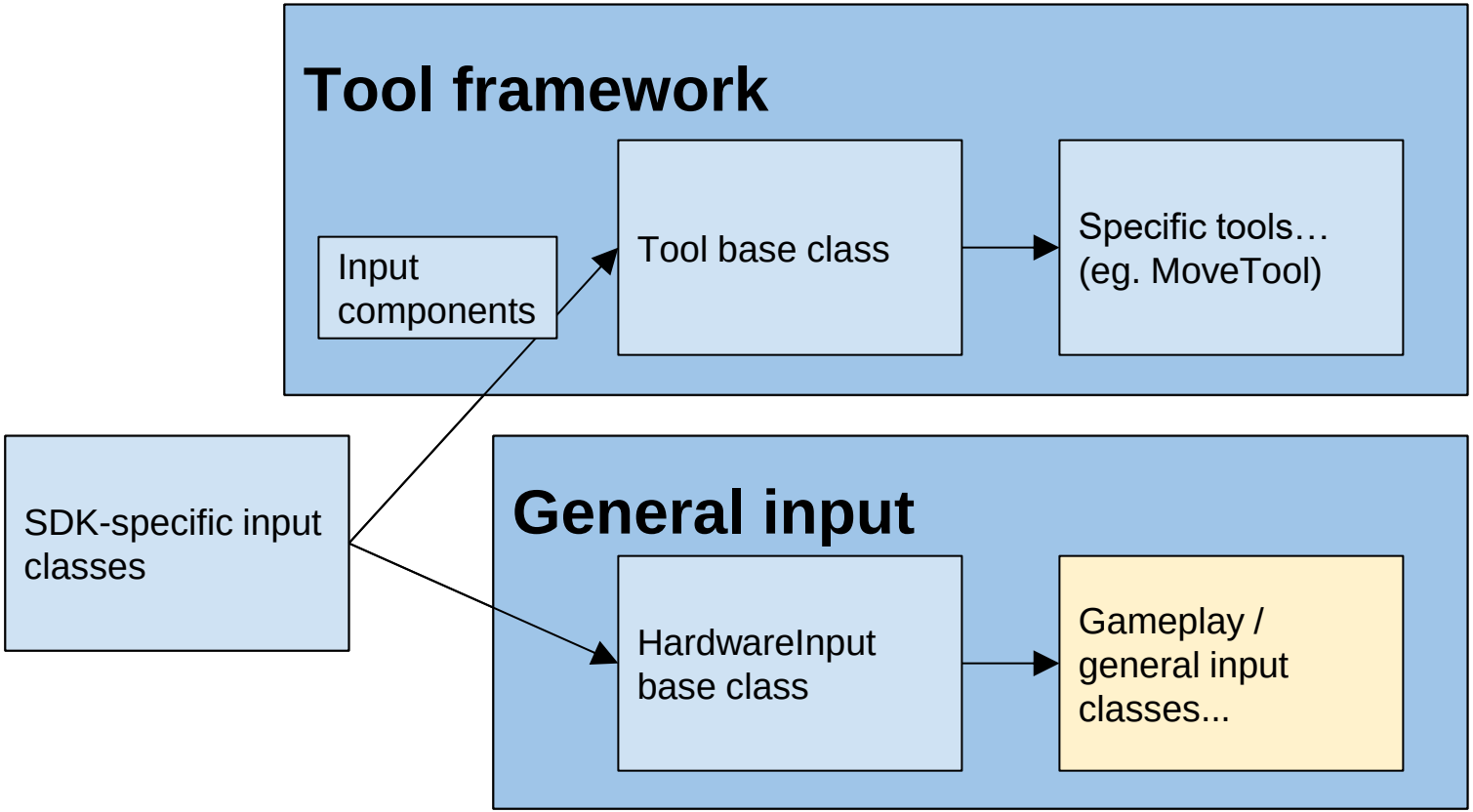

Tool framework
Input components
Tool base class
Specific tools… (eg. MoveTool)
SDK-specific input classes
General input
HardwareInput base class
Gameplay / general input classes...

HardwareInput base class

- Non-Tool abstracted input
- Useful for general gameplay features and one-off interactions

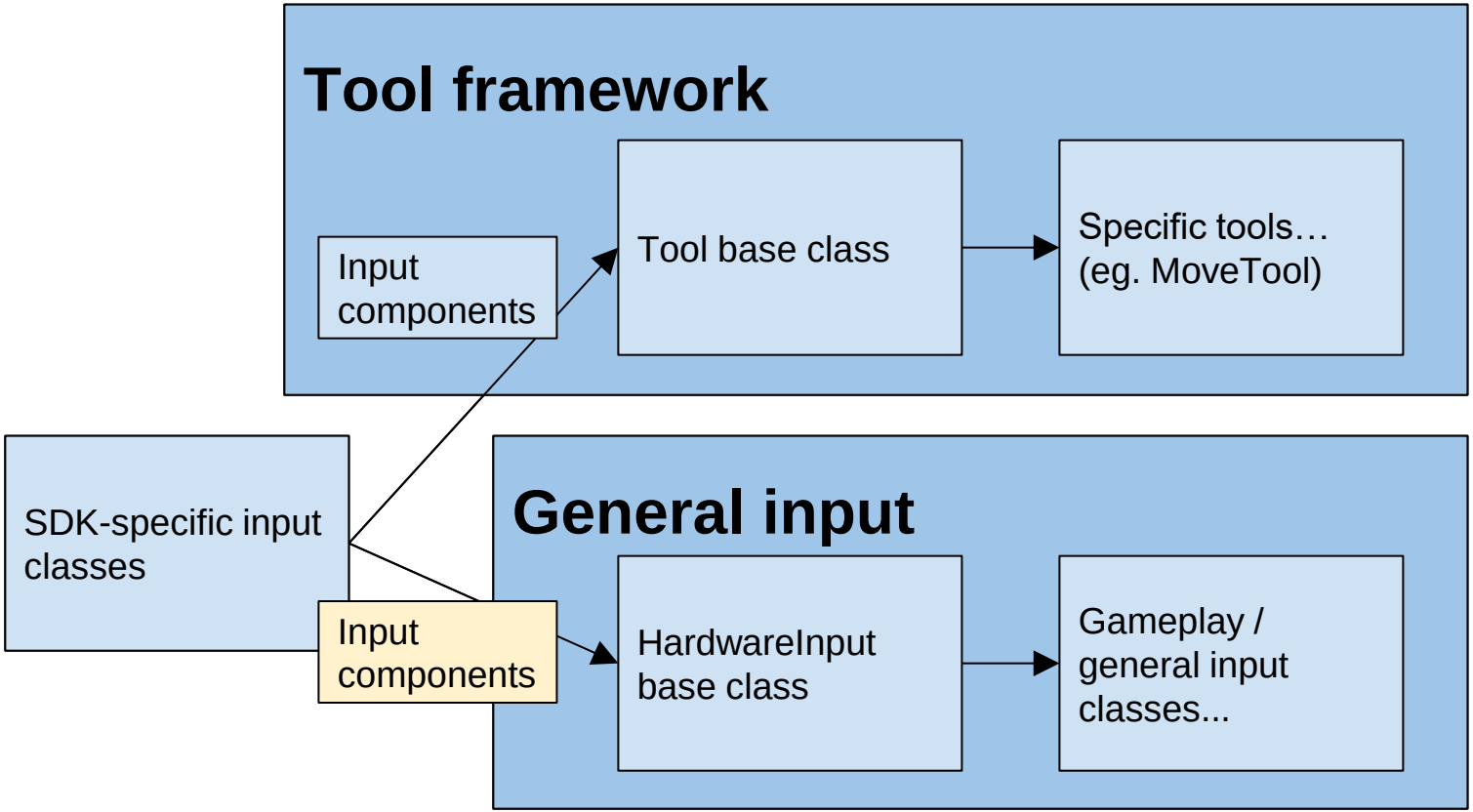
HardwareInput base class

- Three ways to handle it:
  - **bool** HardwareInput.ButtonADown/Held/Up
    - Like Unity's native Input class
  - **event** HardwareInput.OnButtonADown
    - For when you want that observer
  - **void** HardwareInput.HandleButtonADown()
    - Centralized input / gameplay logic


Tool framework
Input components
Tool base class
Specific tools… (eg. MoveTool)
SDK-specific input classes
General input
HardwareInput base class
Gameplay / general input classes...

One-off input

```
public class GameplayController : MonoBehaviour {
    void Update() {
        if(HardwareInput.TriggerDown) {
            WorldConsole.Log("Fire ze missiles!");
        }
    }
    void Awake() {
        HardwareInput.OnButtonADown += HandleButtonA;
    }
    void HandleButtonA() {
        WorldConsole.Log("Boom!"); // Btw: use a “world console”!
    }
}
```


Tool framework
Input components
Tool base class
Specific tools… (eg. MoveTool)
SDK-specific input classes
General input
Input components
HardwareInput base class
Gameplay / general input classes...

General input with components

- Updating those three approaches:
  - HardwareInput.ButtonADown/Held/Up
    - Add HardwareInput.Position, .Forward, etc
  - HardwareInput.OnButtonADown(args)
    - Pass EventArgs containing components
  - HandleButtonADown(components)

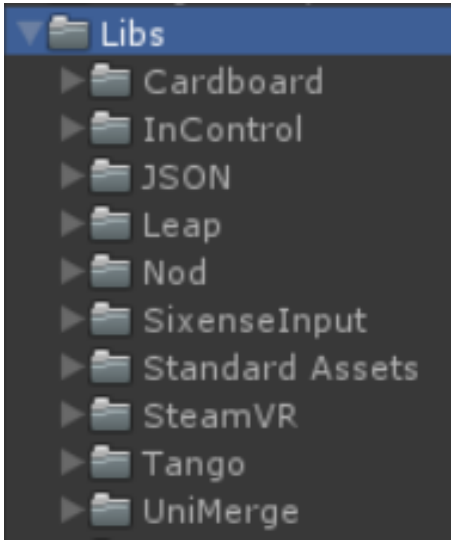
VRDC

Wrangling all those SDKs

Pile of SDKs

- All SDKs exist in the project, in a Libs/ dir

(Not 'Library/'!)



Too many SDKs!

- AndroidManifest & plugin conflicts between SDKs
  - In some cases, you can resolve by merging manifests (for example, Cardboard + Nod)
  - In many cases, you'll just want to move conflicting SDKs in and out of Assets/
    - Can hook into BuildPipeline

Multi-SDK scene setup

- The scene is set up to support all SDKs
  - The Player object has components for ViveInput, GamepadInput, CardboardInput, NodInput, LeapInput, TangoInput...

Multi-SDK scene setup

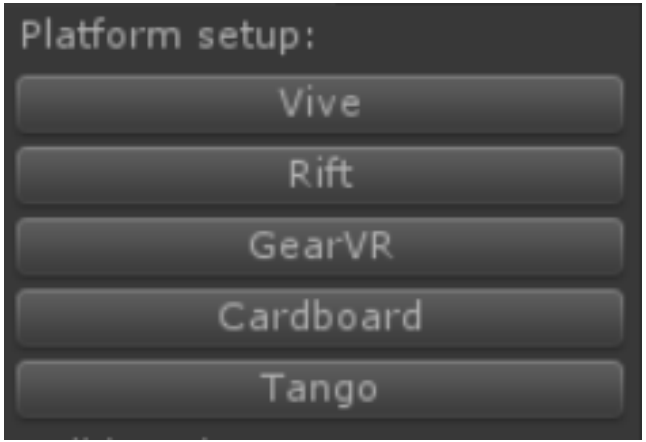
- Pros:
  - No duplicated work between scenes
  - All devices can be enabled simultaneously (eg. Vive + Leap)

Multi-SDK scene setup

- Cons:
  - Letting multiple devices types interact means new design challenges
  - More platforms == more complicated scene
    - You could split the player into more manageable prefabs, and assemble those at runtime or with an editor script

SDK manager editor script

Enable / disable components & objects per-platform, in editor or at build time



SDK manager editor script

```
public void SetupForCardboard() {
    Setup(
        // Build settings
        bundleIdentifier: "io.archean.cardboard",
        vrSupported: false,
        // GameObjects
        cameraMasterActive: true,
        sixenseContainerActive: false,
        // MonoBehaviours
        cardboardInputEnabled: true
    );
}
```

VRDC

Simple custom UI

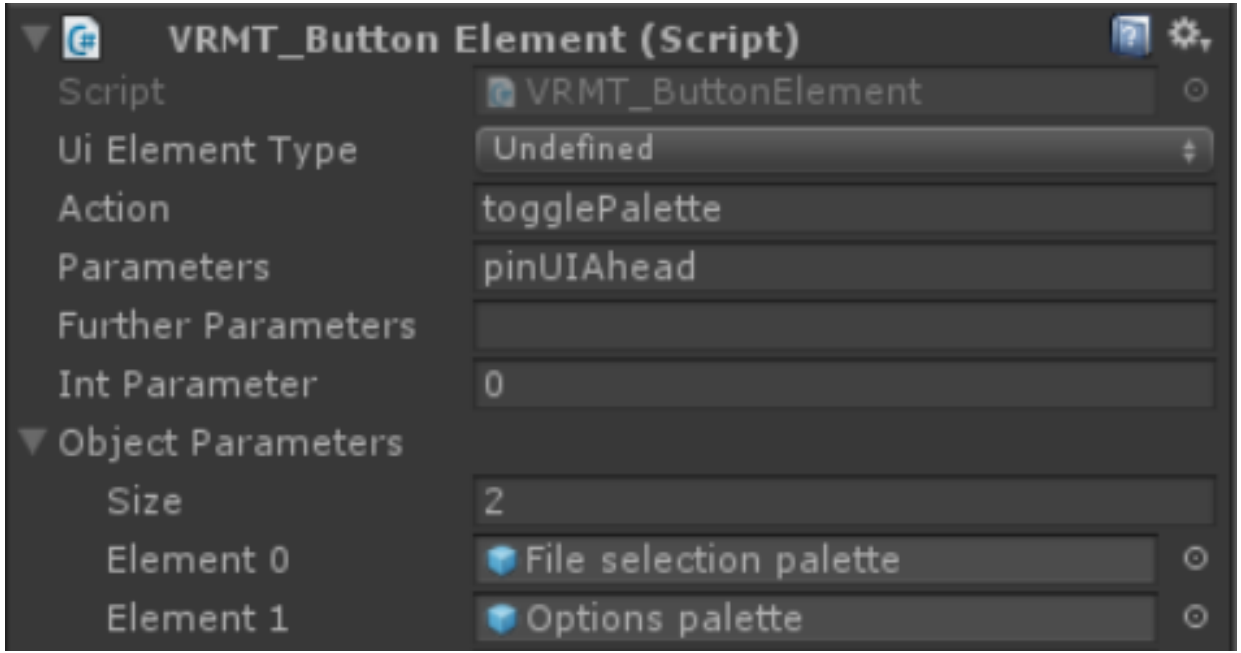
But what about uGUI?

- TL;DR: It was beta when I started Archean
  - 3D objects in uGUI weren't fully supported at the time
  - No complex widgets in Archean yet, but would switch for those rather than re-invent the wheel entirely
- Custom InputModules will let you add new hardware support to uGUI

Block-and-pointer UI

- Point or press
  - <Device>Input raycasts onto UI (eg. Vive)
  - Or simple collision (eg. Leap)

Custom button component



ButtonHandler

There's a giant switch statement in a ButtonHandler class to map all actions

```
switch(button.action) {
case ButtonStrings.Action_TogglePalette:    TogglePalette(button, state); break;
case ButtonStrings.Action_ChangePage:       ChangePage(button, state);  break;
case ButtonStrings.Action_ChangePagination:ChangePagination(button);  break;
case ButtonStrings.Action_SelectProp:       SelectProp(button, state);  break;
case ButtonStrings.Action_SelectTool:       Tool.HandleSelectToolButton(button);
break;
…
```

ButtonStrings

And a file full of const strings for actions

```
public const string Action_TogglePalette    = "togglePalette";
public const string Action_ChangePage       = "changePage";
public const string Action_ChangePagination = "changePagination";
public const string Action_SelectProp       = "propSelect";
public const string Action_SelectTool       = "toolSelect";
....
```

- More advanced & reusable functionality is available through the parameter fields
- Having UI code centralized, and buttons able to pass any data type, is very convenient

VRDC

Looking ahead

So you want to make a multiplatform VR app…

- Are you sure?
- SteamVR & Cardboard coming to Unity's native VR support
- Plan on multiplatform from the get-go

VRDC

Thanks!

Questions?

jono@archean.io
@archeanvr