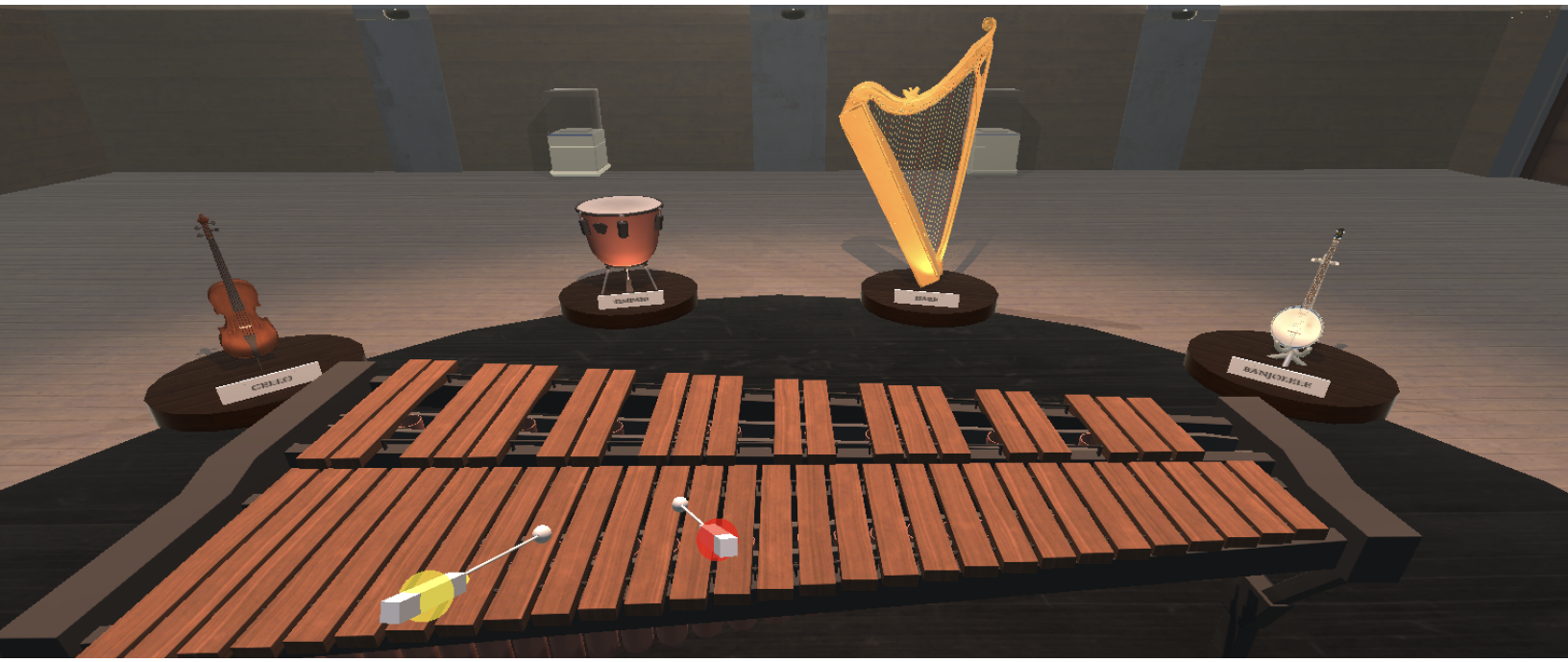



# The SIVE Toolkit Manual

Silvin Willemsen, Helmer Nuijens, Titas Lasickas, and Stefania Serafin

June 2022

# The SIVE Toolkit Manual

Silvin Willemsen, Helmer Nuijens, Titas Lasickas, and Stefania Serafin

June 2022

**Abstract**

This is the manual for the SIVE (Sonic Interaction in Virtual Environments) toolkit. This toolkit allows you to create realistic-sounding musical instruments in VR!

## Contents

# 1 Introduction

Hello, and welcome to the documentation of the SIVE (Sonic Interaction in Virtual Environments) Toolkit! If you are interested in creating realistic-sounding musical instruments in a Virtual Reality (VR) environment, you have come to the right place. The accompanying video to this manual can be found online[1], as well as a demo of what can be achieved with this toolkit[2].

The toolkit, as well as this document, is structured into two main parts:

- Section 2: The ModularVST – which details the sound engine for the instruments created using C++ and the JUCE framework.
- Section 3: Physical Models in VR using Unity – which details the setup of musical instruments in a Virtual Reality environment as well as the inclusion of the ModularVST in Unity.

## 1.1 Software Prerequisites

Before we go into the details of the toolkit, please make sure you have the right software installed.

### JUCE

JUCE is a framework for C++ specifically targeted at audio development. If you have not used JUCE before please follow Appendix A before continuing. In any case, make sure you have the latest version of JUCE[3]. The version used for the repository is *v7.0.0* (subject to change), but newer versions of JUCE should be able to build the project. If any errors occur throughout the building process, download JUCE v7.0.0 instead.

### Unity

Unity is a game engine commonly used to make VR applications. If you do not have Unity Hub, download it[4]. Make sure that you install Unity version *2020.3.27f1* – which is what the example project in this tutorial uses – by clicking on *Installs* and *Install Editor* (see Figure 1). You'll probably have to click on *Archive* and find the correct version through the Unity website.

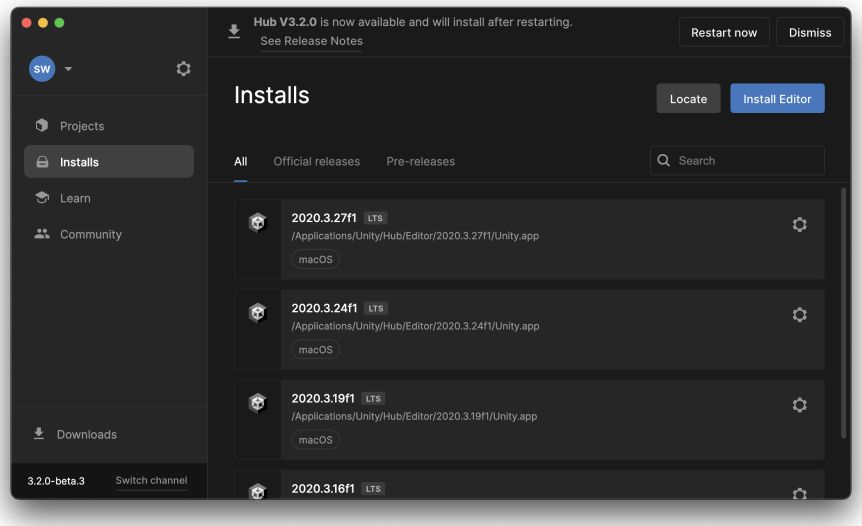



Figure 1: Unity Hub.

If you have the above software installed, you are ready to use the SIVE Toolkit!

[1] SIVE Toolkit Tutorial: https://youtu.be/1Eu0Q3fQyLQ
[2] Demo of the SIVE toolkit: https://youtu.be/qxG7kwYsNZw
[3] JUCE: https://juce.com/get-juce/
[4] Unity Hub: https://unity3d.com/get-unity/download

# 2 The ModularVST

The ModularVST project is a tool for building musical instruments using physical models (based on finite-difference time-domain (FDTD) methods) in a modular fashion. The tool can be used to build the audio engine to be imported into the Unity project. For more details on the inner workings of the application, please refer to [1]. This section is intended as a manual for setting up and using the ModularVST.

Depending on the build settings the ModularVST can be used as a stand-alone application, a Unity plugin or a VST (hence the name) to be used in a digital audio workstation (DAW). Here, the *stand-alone application* will be used to create a new musical instrument, and the *Unity plugin* will be used as the audio engine for the eventual VR application.

## 2.1 Repository Structure and Project Setup

The latest version of the repository can be downloaded from GitHub[5]. The structure of the repository can be found below. Note that only the files that are important for you are highlighted.

```
ModularVST
├── JuceLibraryCode/
├── Presets/
│   ├── <filename>.xml
│   └── ...
├── Source/
│   ├── AppConfig.h
│   └── ..
├── eigen/
├── ModularVST.jucer
└── README.md
```

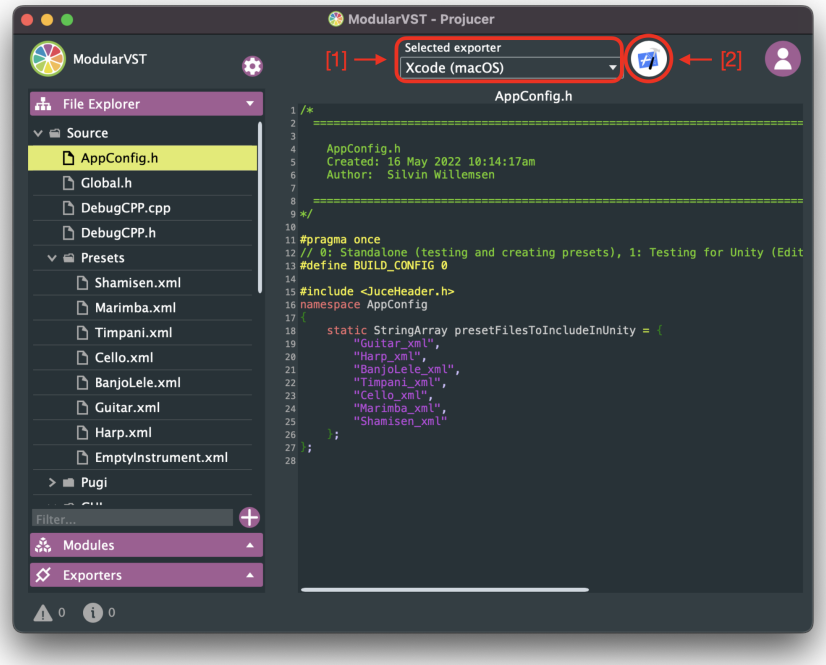



Figure 2: The Projucer, [1] Select the correct exporter, [2] Save and open in IDE.

[5]The ModularVST repository: https://github.com/SilvinWillemsen/ModularVST

Once you cloned the repository to your local drive:

- Open the `ModularVST.jucer` file using the latest version of JUCE (see Section 1.1).
- Select the IDE you use from the dropdown menu ([1] in Figure 2) and click on the *Save and Open in IDE* button ([2] in Figure 2).[6].
- Check that the Build Configuration is set to *Release* to greatly improve performance (see Appendix B.1 for more information).
- Build and run the project, by clicking on the 'play button' (top-left for Xcode, top-center Visual Studio). If you get any errors, please contact Silvin Willemsen.

## 2.2 Functionality of the application (based on [1])

The ModularVST application can be seen in Figure 3. The graphical user interface (GUI) is divided into three main components: the control panel (bottom) containing the buttons to build and configure musical instruments, the excitation panel (right) where one can select the excitation type, and the instrument area, where the instrument can be interacted with.

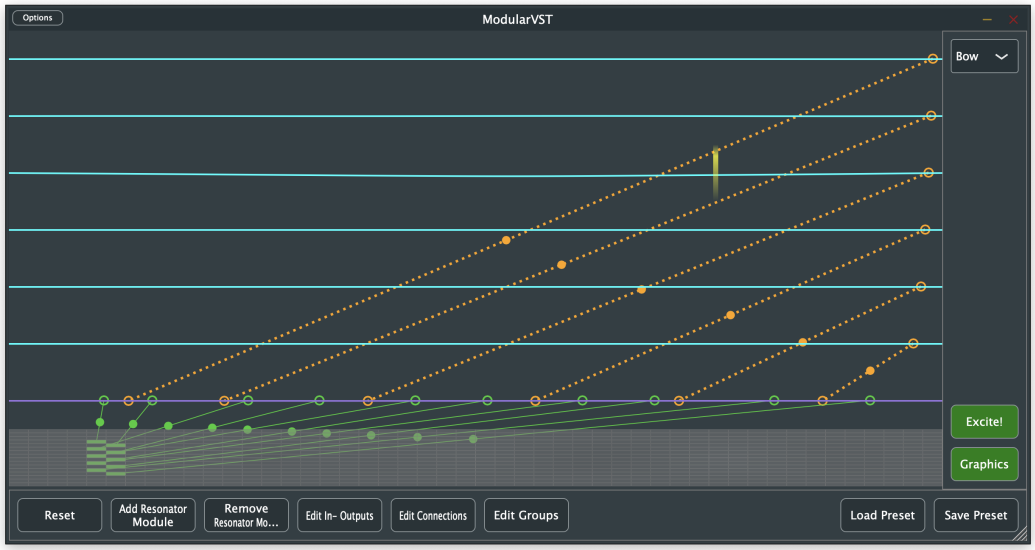



Figure 3: The ModularVST with the guitar preset loaded.

Using the buttons, one can add *resonator components* (such as strings, plates and membranes) to the application and connect them to form either existing or non-existing instruments. These components can then be excited using either a pluck, a hammer or a bow.

The rest of this section will go through the functionality of the application, by going through the control panel button by button. It is important to note that the states of all resonator modules will be set to 0 upon any button press, to prevent audible artefacts. Furthermore, upon clicking some buttons, the control panel will show instructions on the chosen option as well as a *Done* button to return to the normal application state.

### 2.2.1 Reset

The *Reset* button removes all resonator components from the application by loading the `EmptyInstrument` preset.

[6] When using Visual Studio, it is important that the version of the dropdown menu matches the VS version in which the project gets opened.

### 2.2.2 Resonator Modules

The *Add Resonator Module* button opens a separate window for configuring and adding a resonator to the application (see Figure 4). The resonators available to the user are: the stiff string, bar, membrane, thin plate and stiff membrane.

One can toggle between an advanced (all parameters) and non-advanced (a small selection) list of parameters by clicking on the *Advanced* button. For 2D models, an extra parameter called 'maxPoints' is provided such that the number of grid points does not surpass this. This can prevent CPU overloading and keep the application running in real time. The *Add Module* button adds the resonator module to the instrument.

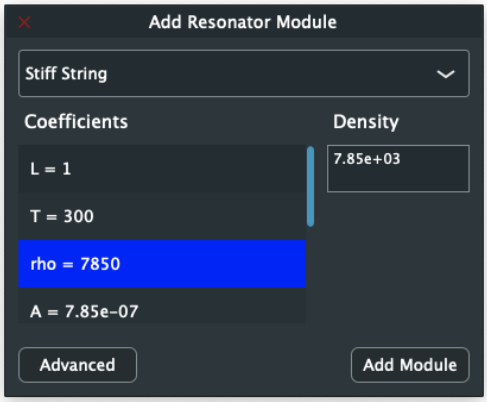



Figure 4: Add Resonator Module window. The resonator type can be selected in the dropdown menu (stiff string, bar, membrane, thin plate, and stiff membrane) and the parameters can be adjusted using the list and the textbox.

Resonators can also be removed from the application by pressing the *Remove Resonator Modules* button. Upon clicking the button, a *Remove* and *Done* button appear. Then, upon clicking on a resonator module, a red overlay is added, indicating that this is the resonator to be removed. Finally, clicking the *Remove* button removes the resonator from the instrument.

### 2.2.3 Outputs

The output locations can be changed by clicking the *Edit Outputs* button. Left, right and stereo channels can be added separately and will be shown in white, red, and yellow respectively. See Figure 5. For 1D systems, outputs will show as downwards pointing arrows, and for 2D systems these will be shown as rectangles around the output grid point.

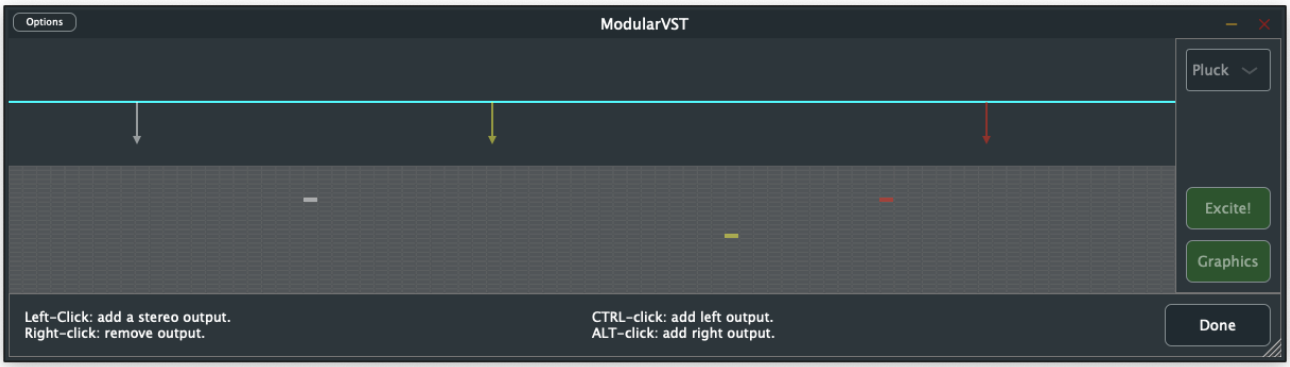



Figure 5: The output locations are shown after clicking the *Edit Outputs* button. Left, right and stereo channels are shown in white, red and yellow respectively.

### 2.2.4 Connections

The *Edit Connections* button allows the user to add connections between various resonators. Through a dropdown menu, the user can select three different connection types – rigid, linear and nonlinear – which are visualised with solid-green, dotted-orange, and dotted-magenta lines respectively (see Figure 6). The mass-ratio between the two connected grid points is visualised with a solid circle along the connection line. The closer the circle is to a resonator, the heavier a grid point of that resonator is when compared to the other. Clicking on a connection-end makes it editable, which is indicated by a yellow 'halo' around the ends of the connection. Overlapping connections are not allowed, and if attempted, the currently active connection will be automatically removed.

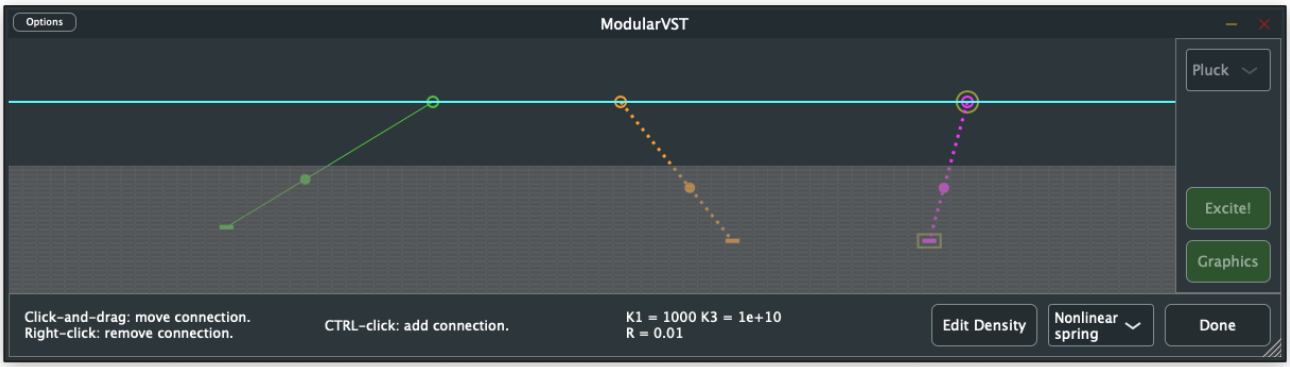



Figure 6: After clicking the *Edit Connections* button, one can edit connections between resonators. Two resonator modules are connected by a rigid (green), linear (orange), and nonlinear (magenta) connection. The nonlinear connection is currently being edited as shown by the yellow 'halo' around the ends of the connection.

Through the edit connections option, a user can change the density of a resonator module by pressing the *Edit Density* button and selecting a resonator module, which is then highlighted in yellow (see Figure 7). A slider allows the density of the resonator to change, which changes the amount that one resonator affects another.

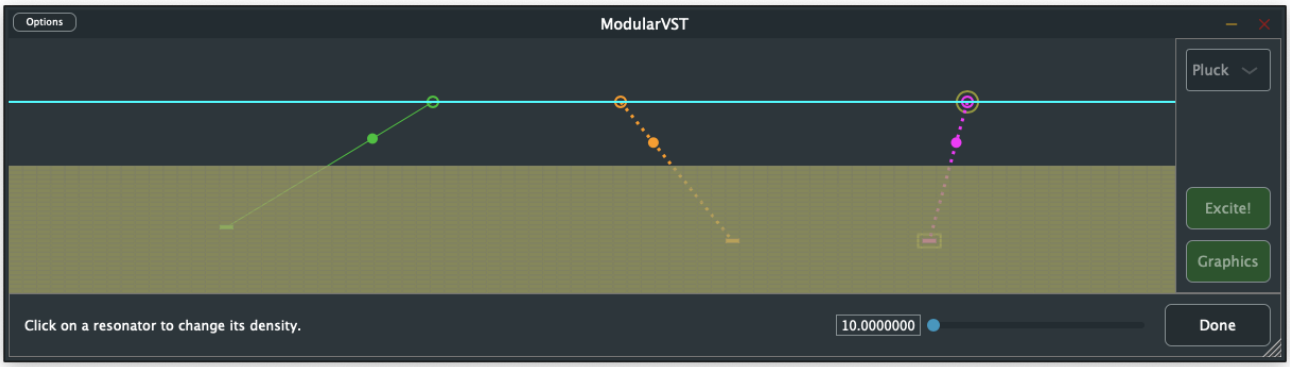



Figure 7: The *Edit Density* button, shown after clicking the *Edit Connections* button allows for the user to change the density of a resonator module using a slider. Notice that the solid circles along the connection locations are closer to the string than in Figure 6 as the density of the plate has been decreased.

### 2.2.5 Groups

1D resonators can be grouped to be excited together using the *Edit Groups* functionality. See Figure 8. A group can be added by clicking the *Add Group* button after which resonators can be selected to belong to the same group. This is indicated by a coloured rectangle around the resonators that belong to a group.

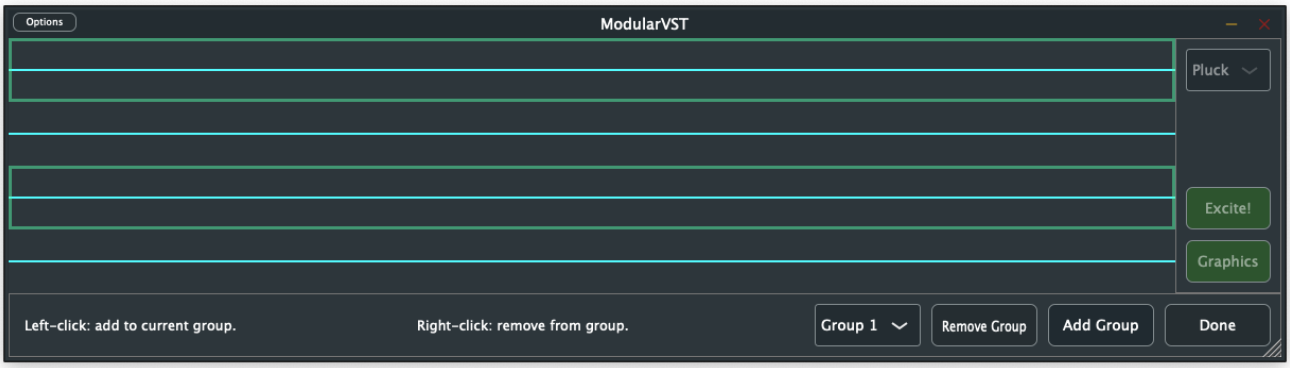



Figure 8: The *Edit Groups* functionality allows for 1D resonators to be grouped, i.e., excited together. Groups are indicated by coloured rectangles around the grouped resonators (in this case the first and third resonator are grouped.

### 2.2.6 Presets

The application allows for presets to be loaded and saved as '.xml' files. These presets are important, as they need to be used to get a working Unity plugin. Clicking on the *Load Preset* button causes a window to pop up. See Figure 9. Here, the user can either choose to load a preset from the hard drive using the *Load preset from file* button, or choose one of the built-in presets from a drop down menu. The latter are added to the application as binaries through the Projucer as shown in Section 2.3 and are as such "part of the application". To load the preset the *Load Preset* button must be clicked.

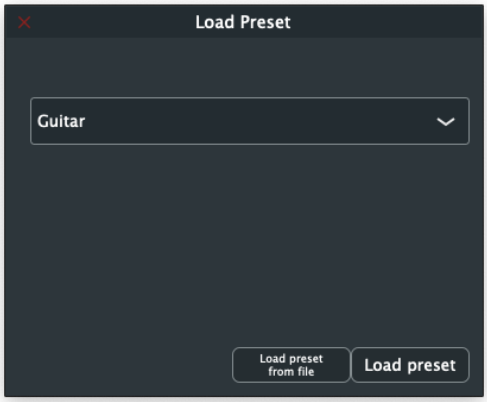



Figure 9: The Load Preset window. The user can either load a preset from the hard drive by pressing the *Load preset from file* button, or choose a built-in preset from the dropdown menu.

Upon clicking the *Save Preset* button, a File Chooser window appears that allows the user to save their preset somewhere on their hard drive. **It is recommended to save all presets in the 'Presets' folder located in the root of the ModularVST repository.**

## 2.3 Adding Presets as Binaries

In order for presets to be contained in the Unity plugin, and thus work in the eventual VR implementation, they need to be included as *binaries*:

1. In the ModularVST, create an instrument and click on the *Save Preset* button.
2. Navigate to the 'Presets' folder located in the root of the ModularVST repository.

3. Open the `ModularVST.jucer` file in the Projucer.
4. Navigate to the Presets folder ([1] in Figure 10).
5. Click on the '+' and select "Add Existing Files..." ([2] and [3] in Figure 10).
6. Find your preset in the File Chooser window.
7. Click on "Save and Open in IDE" (Xcode / VS icon).

Your preset should now be added as a binary to the plugin. To test this, run the ModularVST and click *Load Preset*. The preset should appear in the dropdown menu of the Load Preset window.

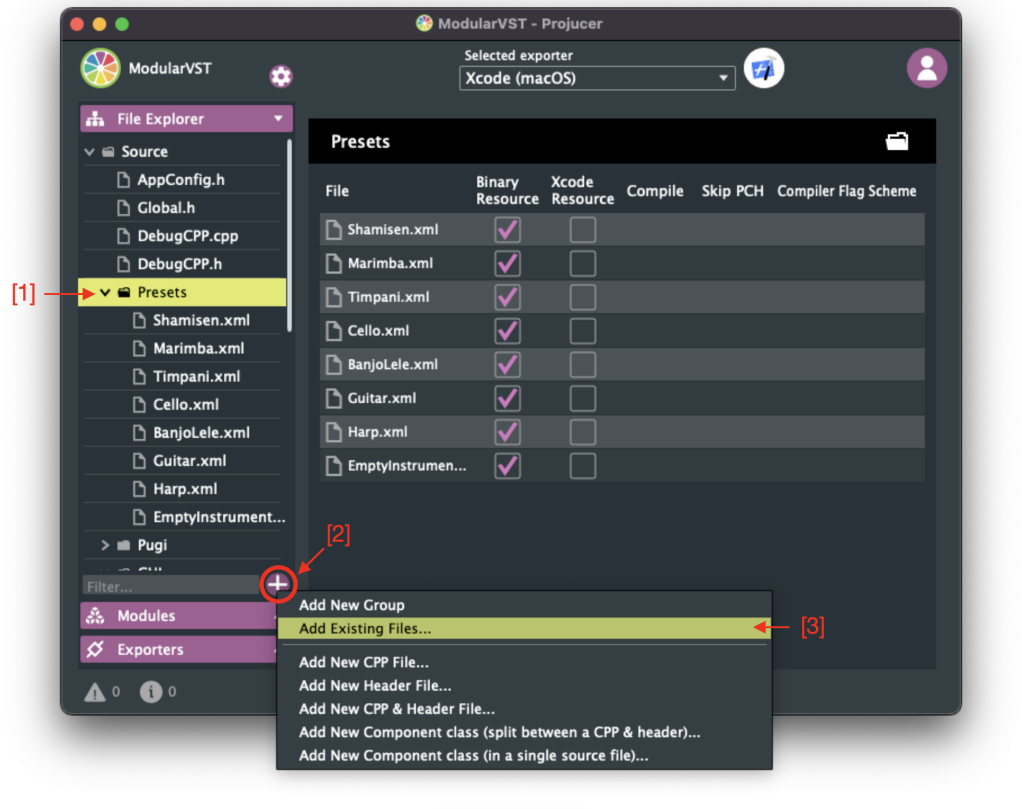



Figure 10: Adding a preset as a binary to the ModularVST. [1] Navigate to the Presets folder. [2] Click on '+'. [3] Click on "Add Existing Files...". Then find your preset in the File Chooser window.

## 2.4 Application Configuration: Standalone vs. Unity

As the ModularVST can be built as a standalone application as well as a Unity plugin, the application must be configured for each case.

### 2.4.1 Build Configuration

One line of code in the `AppConfig.h` file (see Listing 1) sets the value of the `BUILD_CONFIG` macro, and determines how the app will be built. On Mac, the file can be found under (left panel)

$$\textit{ModularVST} \rightarrow \textit{Source} \rightarrow \textit{AppConfig.h}$$

or on Windows (in the Solution Explorer[7])

[7] If the Solution Explorer doesn't show, go to *View* → *Solution Explorer*.

*ModularVST_SharedCode* → *ModularVST* → *Source* → *AppConfig.h*

The `BUILD_CONFIG` macro can be set to the following values:

- 0: *Standalone*. This setting provides the normal GUI where you can build your instrument presets. **This is the setting you will probably use most of the time**.
- 1: *Testing for Unity*. This provides the GUI as well as an extra panel with additional sliders that 'simulate' the sliders used in the *Build for Unity* option. This option should only be used for having a visual feedback when debugging the slider functionality. The functionality of the sliders can be found in the next section in Table 2.
- 2: *Build for Unity*. This option must be used when building the Unity plugin. The resulting build will provide a standard GUI (sliders only) such that it can be used in Unity. Should be used when building the Unity plugin build setting.

```
// Build configuration
// 0: Standalone (testing and creating presets),
// 1: Testing for Unity (Editor and sliders + all presets)
// 2: Build for Unity
#define BUILD_CONFIG 2

#include <JuceHeader.h>
namespace AppConfig
{
    static StringArray presetFilesToIncludeInUnity = {
        "Guitar_xml",
        "Harp_xml"
    };
};
```

Listing 1: The `AppConfig.h` file.

### 2.4.2 Build Target

The appropriate build target must also be selected. In Xcode this is done by selecting

*Product* → *Scheme* → *ModularVST – Standalone Plugin* OR *ModularVST – Unity Plugin*.

In Visual Studio (by default), one can click the run button to build the standalone plugin. To build the Unity plugin, right-click on *ModularVST_UnityPlugin* in the Solution Explorer and select build.

### 2.4.3 Presets to include

When building for Unity, the `presetFilesToIncludeInUnity` variable should be changed to only include presets / instruments you want in your Unity VR application. A single Unity build can thus contain multiple instruments! The project found on the repository contains 2 presets by default, both of which are loaded as binaries. Change the `presetFilesToIncludeInUnity` variable to whatever presets you want to include. The name should be `"<insertBinaryPresetName>_xml"` to be correctly included.

### 2.4.4 Build Configuration: Debug vs. Release

If you are building for Unity, make sure that the *Build Configuration* is set to *Release*. See Appendix B.1 for more information.

## 2.5 Summarised Building Instructions

Building the Standalone Plugin for Instrument Building

1. In the `AppConfig.h` file, set the `BUILD_CONFIG` macro to 0.
2. Select *Product* → *Scheme* → *ModularVST – Standalone Plugin* (Mac) or click the run button (Windows).
3. Build.

Building the Plugin for Unity

1. If the presets you want to include in Unity are not added as binaries, follow Section 2.3.
2. In the `AppConfig.h` file, set the `BUILD_CONFIG` macro to 2.
3. In the `AppConfig.h` file, change the `presetFilesToIncludeInUnity` variable to only contain the presets you want include in your Unity VR application. Names should be of the form `"<insertBinaryPresetName>_xml"`.
4. Make sure to use *Release* (not *Debug*) as the *Build configuration* (see Section B.1).
5. Select *Product* → *Scheme* → *ModularVST – Unity Plugin* (Mac) and build or right-click on the ModularVST Unity Plugin target and select Build (Windows).

Once the plugin is built it will be located here (Mac):

ModularVST/Builds/MacOSX/build/Release/audioplugin_ModularVST.bundle

or here (Windows)

ModularVST\Builds\VisualStudio20XX\x64\Release\Unity Plugin\audioplugin_ModularVST.dll

It can then be included in the Unity project once that is set up (see Section 3.4).

IMPORTANT: It sometimes happens that the plugin file is not overwritten upon building. It is therefore good to always delete the .bundle / .dll file before building for Unity to be sure that the file represents the newest version of the plugin. (You can see this by checking the *Date created* property of the file.)

# 3 Physical Models in VR using Unity

This section will go into the setup of musical instruments in Unity as well as the inclusion of the ModularVST detailed in the previous section.

The SIVE Toolkit Unity project can be found online via the following repository[8]. The folder structure is as follows:

```
SIVEtoolkit
├── SIVEtoolkit/
│   └── ...
├── SIVEtoolkit.unitypackage
└── README.md
```

The SIVEtoolkit folder contains the Unity project and can be opened through Unity Hub using the *Open* button. This project should work out of the box (if Unity version 2020.3.27f1 is used).

## 3.1 Using the SIVEtoolkit package

If you would like to use the SIVE Toolkit in your own Unity project, the following steps show how to set this up from a blank project.

1. Open Unity Hub and create a new 3D project using Unity version 2020.3.27.f1.
2. Locate the folder that just got created, go to the *Packages* folder and open the *manifest.json* file in any texteditor.
3. Copy and paste the following after the first curly bracket: { and before `"dependencies"`:

```
"scopedRegistries": [ { "name": "npmjs", "url": "https://registry.npmjs.org/", "scopes":
["io.extendreality" ] } ],
```

4. Next, add the following dependencies:

```
,
"io.extendreality.tilia.camerarigs.spatialsimulator.unity": "2.0.10",
"io.extendreality.tilia.camerarigs.trackedalias.unity": "2.0.4",
"io.extendreality.tilia.camerarigs.xrpluginframework.unity": "2.0.4",
"io.extendreality.tilia.input.unityinputsystem": "2.0.4",
"io.extendreality.tilia.interactions.interactables.unity": "2.6.0",
"io.extendreality.tilia.interactions.pointerinteractors.unity": "2.1.18",
"com.unity.xr.management": "4.2.1",
"com.unity.xr.oculus": "1.11.2"
```

   Saving the file should trigger unity to download the necessary VRTK (Tilia) and OculusXR packages.

   - If prompted about a "new input system" click *No*. We will change the settings manually in the next step.
   - If prompted about Input Definitions, click on *Add Input Definitions* (see Figure 11).

5. Go to *Edit* → *Project Settings* → *Player and locate Settings for PC, Mac & Linux Standalone* → *Other Settings* → *Active Input Handling*.
6. Set this option to *Both* and restart Unity when it asks you to.
7. Download the SIVEtoolkit unitypackage from the repository.
8. Click *Assets* → *Import Package* → *Custom Package* and locate the *SIVEtoolkit.unitypackage* file. Make sure to select and import all included assets.

[8] https://github.com/SilvinWillemsen/SIVEtoolkit

9. Go to the *Assets* → *Scenes* folder and try out one of the example scenes! The controls for the simulator can be found in Table 1.

10. If you have a VR headset (only non-standalone Oculus headsets such as the Rift are supported as of now) you can switch between the simulator and the headset by going to *Hierarchy* → *SetCameraRig* → *Xr Setup* and choose *Use Oculus* from the dropdown menu.

    - You might need to manually drag the *CameraRigs.UnityXRPluginFramework* GameObject into *CameraRigs.TrackedAlias* → *Tracked Alias Facade* → *Tracked Alias Settings* → *Elements* → *Element 0* (see Figure 12).
    - If you want to move back to using the simulator, do the same action, but with *CameraRigs.SpatialSimulator*.

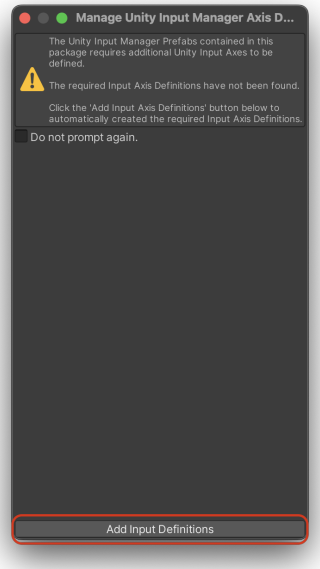



Figure 11: Add Input Definitions.

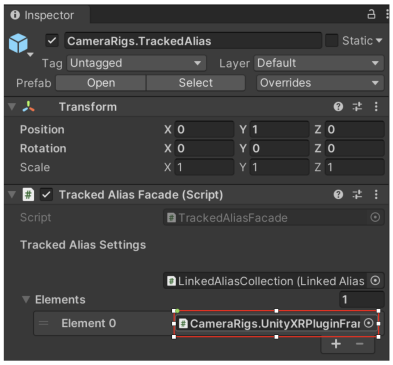


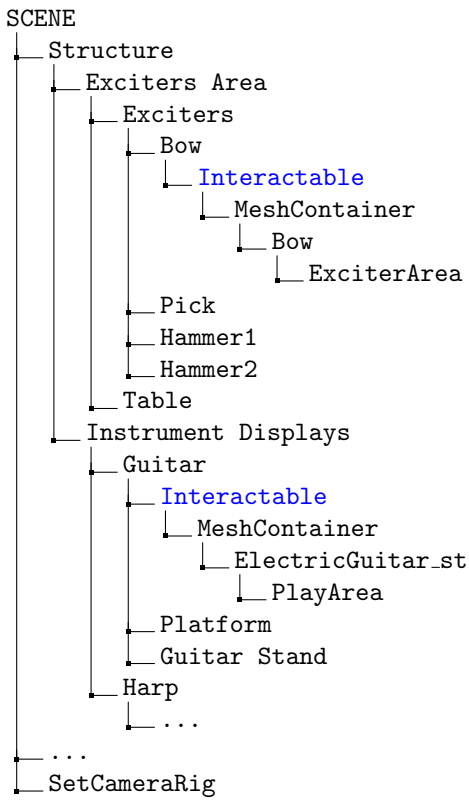

Figure 12: To make the application work with an Oculus headset, manually drag the UnityXR camera rig into this location in *CameraRigs.TrackedAlias*.

| Control | Action |
|---|---|
| `A, S, W, D` | Move |
| Mouse | Rotate |
| `1` | Select player |
| `2` | Select left hand |
| `3` | Select right hand |
| `4` | Reset player position |
| `5` | Reset hand positions |
| Left-click | Left grab |
| Right-click | Right grab |

Table 1: Controls for the VRTK simulator.

## 3.2 Project Hierarchy

After opening one of the example scenes, you can see the project hierarchy (left panel in Figure 13). Below, the most important parts of the hierarchy are highlighted:




The items highlighted in blue – the *Interactables* – can be (as the name suggests) interacted with in the application, either using the simulator or a VR headset. See Section 3.5 for instructions on how to make a Game Object interactable.

## 3.3 Explanation sliders ModularVST

Although the plugin will be controlled via scripts (see Section 3.6), it might be useful to go over the various sliders of the plugin and their functionalities. To find the plugin and its sliders go to:

$$\textit{Window} \rightarrow \textit{Audio} \rightarrow \textit{Audio Mixer}$$

The Audio Mixer panel should show up. Next, click on *Master* ([1] in Figure 13) and finally on the *Plugin* Mixer ([2] in Figure 13). The sliders should show in the Inspector. Table 2 shows the various slider names and their functionalities.

## 3.4 Including the Plugin into the Unity Project

If you have changed the ModularVST plugin by adding presets to it (see Section 2.3), a new version of the plugin needs to be included in the Unity project. For Unity to recognise the plugin (.bundle or .dll), its file name and location are incredibly important. The plugin name has to start with "audioplugin_" in order to be recognised as a plugin by Unity. This is done by default by JUCE, but it is important to know that renaming the file probably causes issues. The plugin location in the Unity project should be (Mac)

Assets/Plugins/audioplugin_ModularVST.bundle

or (Windows)

Assets\Plugins\audioplugin_ModularVST.dll

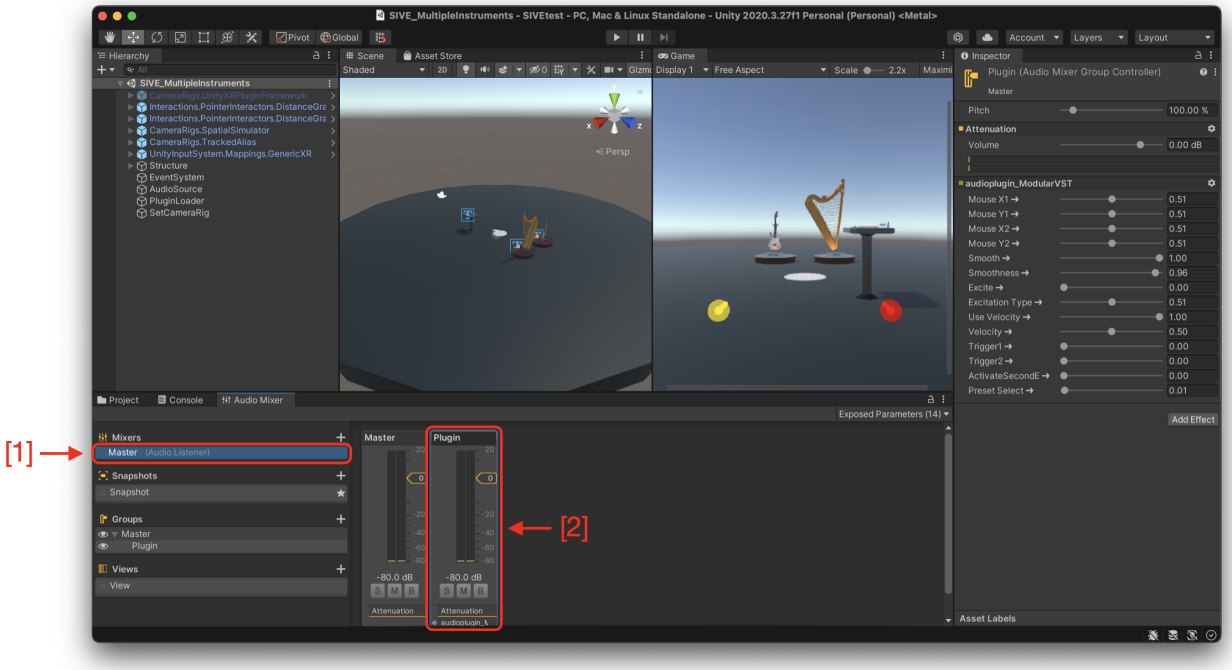



Figure 13: How to find the ModularVST plugin in the Unity Editor.

IMPORTANT: Before adding a new version of the plugin, always remove the plugin and its .meta file from the Assets/Plugins folder. Also, as the plugin is loaded on startup, only open the Unity project after adding the plugin to the Assets/Plugins folder.

Notice that whenever the `presetFilesToIncludeinUnity` variable is changed, a custom error appears in the console at the startup of the project stating

PRESET LIST HAS CHANGED: Make sure you reselect the right preset in the SelectPreset component of each instrument model!

The instrument models in the example project can be found in the hierarchy via

*Structure* → *Instrument Displays* → <*Instrument*> → <*Interactable*> → *MeshContainer* → <*Model*>

Find the *Select Preset* component and change the Instrument Type to the correct instrument.

### Troubleshooting

If the plugin somehow fails to load, check whether you did the following things:

- Is the `BUILD_CONFIG` macro in the `AppConfig.h` file set to 2?
- Is the Build Configuration set to *Release*?
- Did you remove both the plugin (.bundle/.dll) and its .meta file before restarting Unity?
- Did you restart Unity?

| Slider Name | Functionality |
|---|---|
| Mouse X1 | Controls the exciter's x-position (in the ModularVST plugin). |
| Mouse Y1 | Controls the exciter's y-position (in the ModularVST plugin). |
| Mouse X2 | Controls the second exciter's x-position (in the ModularVST plugin). |
| Mouse Y2 | Controls the second exciter's x-position (in the ModularVST plugin). |
| Smooth | Whether or not to use internal smoothing of the x- and y-positions (recommended). |
| Smoothness | Amount of internal smoothing. |
| Excite | Whether to excite or not. Gets triggered when an *Exciter Area* collides with a *Play Area*. |
| Excitation Type | Selects what excitation is currently active: pluck, bow or hammer. Gets changed when an exciter is grabbed. |
| Use Velocity | Whether or not to use the velocity slider. Leave at 1. |
| Velocity | Velocity of the bow / hammer. Gets changed by the game velocity of the bow / hammer. |
| Trigger1 | Gets triggered when the first hammer hits a Play Area. |
| Trigger2 | Gets triggered when the second hammer hits a Play Area. |
| ActivateSecondExciter | Gets activated when the second hammer is grabbed. |
| Preset Select | Determines what preset is active. Gets changed when an instrument is grabbed. |

Table 2: Slider functionality. Green is only controlled by the second hammer.

### 3.5 Adding a New Instrument to the Application

Setting up a new instrument is done using the following steps:

1. Inside *Structure* → *Instrument Displays*, create a new empty GameObject, rename it according to the instrument type, and drag the *Platform* prefab as a child GameObject. (Depending on the instrument you could also add other child objects, like an instrument holder (e.g. guitar stand).)

2. Move the GameObject containing the platform to where you want it in the scene.

3. Navigate to *Platform* → *Sign* → *Text*, and change the text of the *TextMeshPro – Text* component to the instrument name.

4. Import the instrument model (.bfx) as another child of the just created GameObject. Adjust the orientation, size and location so that it is centred on the platform.

5. Change the tag of the instrument model to *Model*, and add the following components:

   - Add a *Mesh Collider* component, and enable Convex. This is needed in order to make the object a VRTK interactable.[9]
   - Add a *SelectPreset* component. This is used to change the plugin preset whenever the instrument is grabbed) and reference the audio mixer that contains the ModularVST plugin from the inspector. Note that after making this instrument a VRTK interactable (see below), you'll have to change the preset from the drop down list so that it matches the instrument (you might have press play in order to update the preset list).
   - Add a *CustomGrabAttachment* component when the instrument is not hand-held (e.g. a harp). This script is used to place the instrument on a custom location, instead of to the player's hand.

6. Create a child GameObject to the instrument GameObject and call it *PlayArea* and give it the tag: *PlayArea*. In order to control the excitation locations within the plugin the following two components are required:

[9]Note that sometimes the Convex does not match the model correctly, in this case you can recreate the shape yourself using one or more *Box Collider*s instead.

- Add a *Box Collider* component and change the location, orientation and scale of the **PlayArea** (not the collider) so that it encompasses the model's excitation space.[10] If the excitation space is not perfectly box-like, this can be changed in the PlayareaInteractionScript (see below). Check the *Is Trigger* checkbox.
- Add a *PlayAreaInteraction* component. This script maps the excitation location (location of the collision of the exciter and the playArea's Box Collider) of the player to a value between 0 and 1 (the accepted range for exposed parameters). Select the mixer with the ModularVST plugin and select the orientation in which the strings are in (might have to be trial and error to figure this out).

7. Select the instrument model's GameObject and to transform it to a VRTK intractable by navigating to *Window* → *Tilia* → *Interactions* → *InteractableCreator* and then click *Convert To Interactable*. This will replace the hierarchy of the model by placing it within a VRTK structure.

8. Give the interactable (with the name *Interactions.Interactable_<instrument>* the tag: *Instrument*.

9. The next step is to define VRTK's grab interactions with the instrument. Select the interactable and locate the Grab Events in the *Interactable Facade* from within the Inspector (see Figure 14). Here, the following events need to be set up:

    - Add one event to the *First Grabbed* list by pressing the '+' button. Drag the instrument model located inside the interactable hierarchy (*Interactable* → *MeshContainer* → *<Model>*), and choose the *SelectInstrument* script and the *InstrumentGrabbed* as the event.

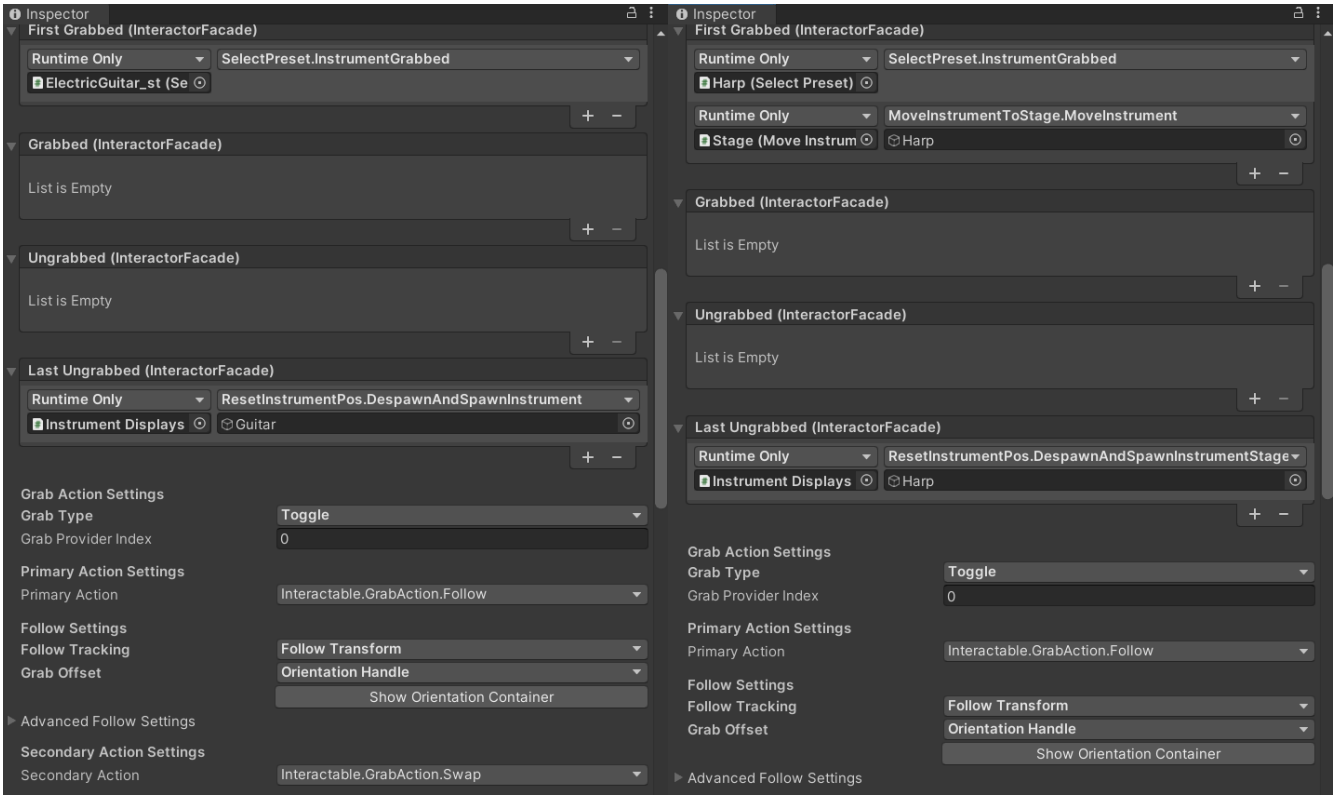



Figure 14: Grab events and settings, with to the left a hand-hand instrument (guitar) and to the right a non-hand-held instrument (harp).

- Add one event to the *Last Ungrabbed* list. Drag the *Instrument Displays* parent GameObject into the event and select the *ResetInstrumentPos* script with the *DespawnAndSpawnInstrument(GameObject)* (or *DespawnAndSpawnInstrumentStage(GameObject)* for non-hand-held instruments) method. Then, drag the instrument parent (parent containing the interactable and platform) into the input GameObject.

[10] Note that the x and y locations on the play area will be mapped to the plugin, so make sure that these have the largest "scale".

- For non hand-held instruments (harp, timpani, Marimba, etc.) add another event to the *First Grabbed* list. Drag the *Stage* GameObject into the event and select the *MoveInstrumentToStage* script with *MoveInstrument(GameObject)* method. Then, drag the instrument parent (parent containing the interactable and platform) into the input GameObject.
- Finally, locate the Grab Action Settings below. Here change *Grab Type* to *Toggle*, the *Primary Action* to *Interactable.GrabAction.Follow*, *Follow Tracking* to *Follow Transform*, *Grab Offset* to *Orientation Handle*, and *Secondary Action* to *Interactable.GrabAction.Swap*. For hand-held instruments, you can move the orientation handle to the neck of the instrument by clicking "Show Orientation Container" and then move the orientation point GameObject to the desired position. To change rotation, navigate to *OrientationHandles* → *OrientationHandleCollection* → *GenericOrientationHandle* and change it here.

10. Finally, (as mentioned above) locate the *Select Preset* component on the instrument model (<Interactable Instrument> → *MeshContainer* → <Model>) and change the preset from the drop down list so that it matches the instrument (you might have press play in order to update the preset list).

### 3.6 Mapping

The last step is to set up the mapping of the instrument play area. Open the *PlayAreaInteraction.cs* script and go down to line 140 (see Figure 15). Here, you'll find a switch statement that maps the location of the exciter on the play area to the *Mouse X* and *Mouse Y* sliders. This will be different for every instrument and will probably take some trial and error to get right.

```
140         // Hard-coded mappings of the play area to the x and y positions //
141         switch (instrumentType)
142         {
143             case "Guitar":
144                 yPos = 0.75f * yPos; // 6/8ths of the plugin are strings, the other 8ths are the bridge and body
145                 float range = 0.1f;
146                 float yPosPre = yPos;
147                 yPos = Global.Limit(Global.Map(yPos, 0, 0.75f, -(1.0f - xPos) * range, 0.75f + (1.0f - xPos) *
                    range), 0, 1);
148                 break;
149
150             case "Harp":
151                 // Map to nonlinear shape of the harp. Top of harp is at at y-coordinate 0, bottom is y-coordinate
                    1
152                 float upperBound = -0.5288f * xPos * xPos + 0.7488f * xPos; // coefficients found using MATLAB
                    cftool
153                 float lowerBound = (1.0f-0.74f*xPos);
154                 yPos = Global.Map(yPos, upperBound, lowerBound, 0.0f, 1.0f);;
155                 outOfBounds1 = (yPos >= 0 && yPos < 1) ? false : true;
156                 audioMixer.SetFloat("excite", outOfBounds1 ? 0.0f : 1.0f);
157                 break;
158
159             //// ADD YOUR OWN CASES HERE FOR THE MAPPINGS OF YOUR CUSTOM INSTRUMENTS ////
160
161             default:
162                 Debug.LogWarning("No custom playarea defined");
163                 break;
164         }
```

Figure 15: Location in the *PlayAreaInteraction.cs* file where the custom mappings of every instrument are defined.

## A Setting up JUCE

If you have not used JUCE before please follow the steps below:

1. Make sure you have an Integrated Development Environment (IDE) that can compile C++.

- **Windows**: make sure you have Visual Studio Community [11]. In the installer select Desktop development with C++ and tick the latest version of the Windows 10 SDK (currently 10.0.19041.0) and the latest MSVC (currently v142).
- **Mac**: make sure you have Xcode (can be found in the App Store).

2. Get JUCE.

   - Download the latest version of JUCE[12], select Education and follow the further instructions.
   - Unzip the downloaded folder and place it somewhere you can find it easily.

3. Set up the Projucer.

   - Find the Projucer in the root folder, open it and go to "Global Paths..." (Windows: under "File", Mac: under "Projucer")
   - Select correct paths for "Path to JUCE" and "JUCE modules", click on Browse (...) and find your JUCE and JUCE/modules folder.

4. Check the if everything works by building your first JUCE plug-in.

   - Under "Plug-in" select "Basic" in the JUCE splash screen (see Figure 16). (If the splash screen is not visible to you choose *File* → *New Project...*).
   - Find a name for your project.
   - Click on "Create Project..." and choose a location to save the project.
   - Click on "Save and Open in IDE" (the Xcode / Visual Studio logo in the top). For Visual Studio users, make sure that the *Selected exporter* matches the version that it opens in (see Figure 2).

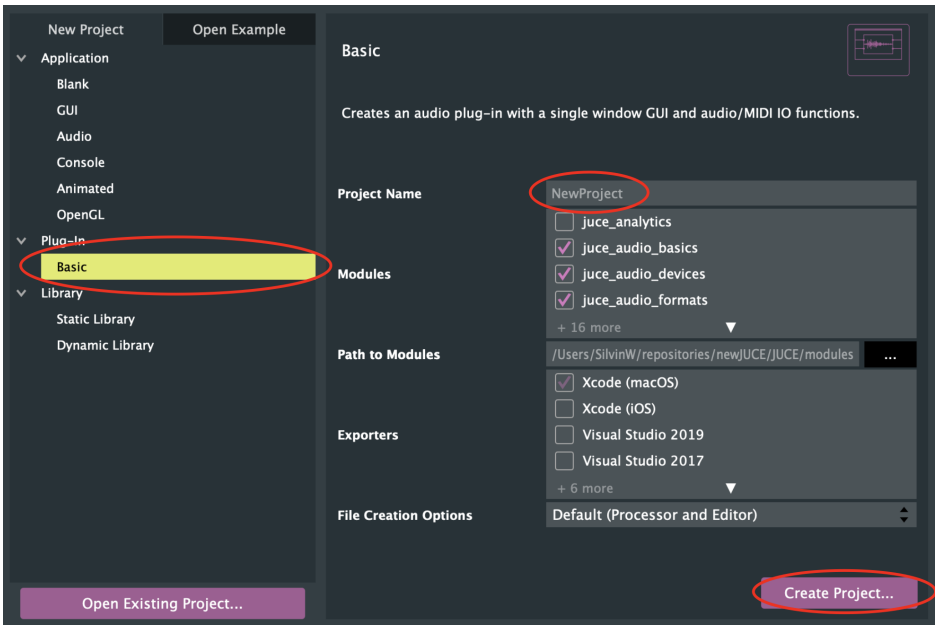



Figure 16: The Projucer (Step 4).

5. Run / Build the project.

   - If you get a pop-up window saying "Hello World!" everything works and you are ready for the tutorial! If not, and you're on Mac, Select *Standalone Plugin* as the scheme from *Product* → *Scheme*, and build again.

[11] Visual Studio: https://www.visualstudio.com/downloads/
[12] https://juce.com/get-juce/

# B Additional Instructions

## B.1 Build Configuration (Debug / Release)

The build configuration, more commonly referred to *Debug* and *Release* modes, determines what optimisation flags the compiler uses when compiling the plugin.

- *Debug* uses the fewest optimisations and is thus builds the application quicker. When using the application, however, auditory dropouts occur much more often due to the lack of optimisations.
- *Release* optimises the code and thus takes a longer time to build the application.

Unless you are editing the source code (not recommended) and want to do quick experimentation, the ***Release* build configuration should always be used**.

In Xcode, the build configuration can be changed by selecting

$$\textit{Product} \rightarrow \textit{Scheme} \rightarrow \textit{Edit scheme...}$$

In Visual Studio, there is a dropdown menu in the top bar, (saying either "Debug" or "Release") by which the build configuration can be selected.

# References

[1] S. Willemsen, T. Lasickas, and S. Serafin, "Modular physical models in a real-time interactive application," in *Proceedings of the 19th International Sound and Music Computing (SMC) Conference*, 2022.