vr-modeling Documentation

Release 1.0.0

Roger Barton

Aug 31, 2020

CONTENTS:

Deform meshes in Virtual Reality with libigl using Unity - A VR editor for libigl

See the Gallery & Showcase **for visual summary.**

CHAPTER

ONE

USER GUIDE

Tip: Most things you can do are indicated by the controller hints, these adapt based on the tool being used. Hover over a UI element to display the tooltip on your left hand.

The rays/lasers are only for UI and teleportation currently. You can grab UI panels and move them around by using the grip when pointing at them.

There is a tool/mode system with currently two modes:

- **Selection** *i.e. Blender edit mode*
  - Edit and grab (multiple) vertex selections
- **Transform** Mesh *i.e. Blender object mode*
  - Move the whole mesh around and select which one can be edited

There are several UI Panels, a generic one and one per mesh. The UI of this mesh is highlighted.

The active mesh is the one currently being modified and will be highlighted if occluded. The active mesh can be set in the top right of the UI of the mesh or in the Transform tool with the brush.

From the UI panel you can see information about the mesh as well as performing more advanced operations. We can enable a deformation from here, currently the libigl biharmonic and as-rigid-as-possible deformations. Deformations can be executed once or continuously every frame.

1.1 Gallery & Showcase

This is a bachelor thesis trying to create a VR editor for libigl with Unity. It is meant to work towards a VR equivalent of the 2D libigl viewer.

Fig. 1.1: **Biharmonic deform** example with **multi-selection** and **two handed transformation**.

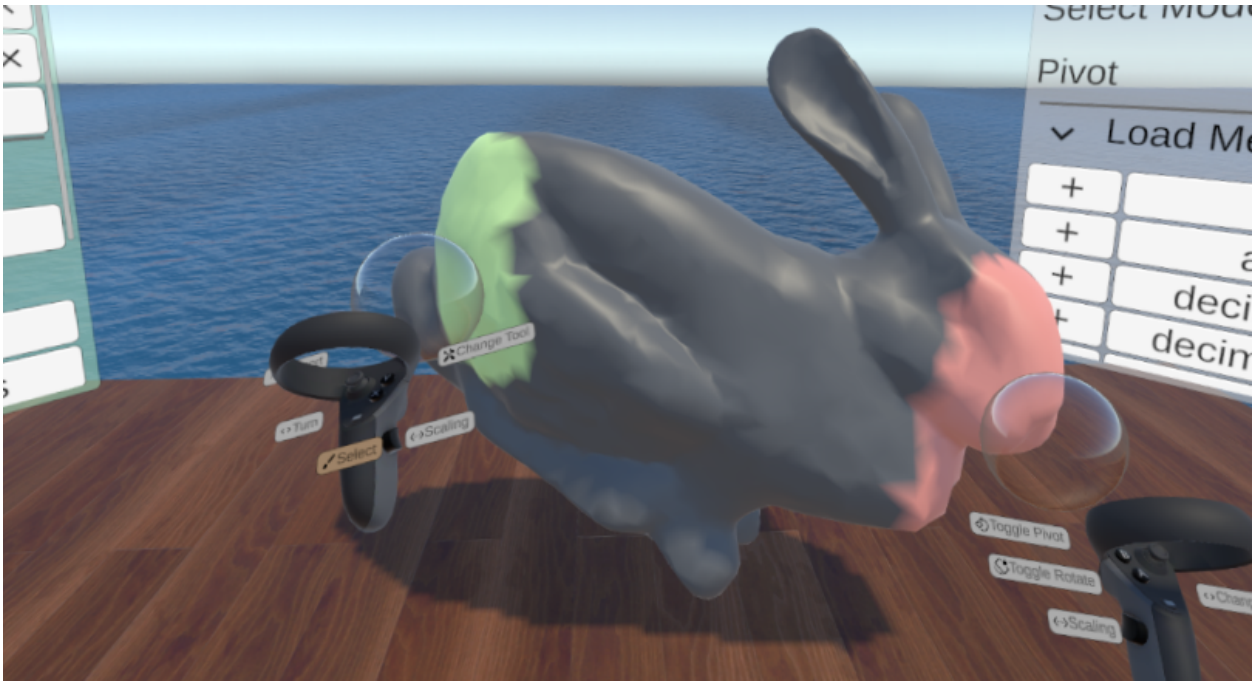

Fig. 1.2: **As-Rigid-As-Possible** deform example. Performs slower than the harmonic, however, causes less distortions and retains more surface details. We can see that the geometry operations are **on a separate thread**.

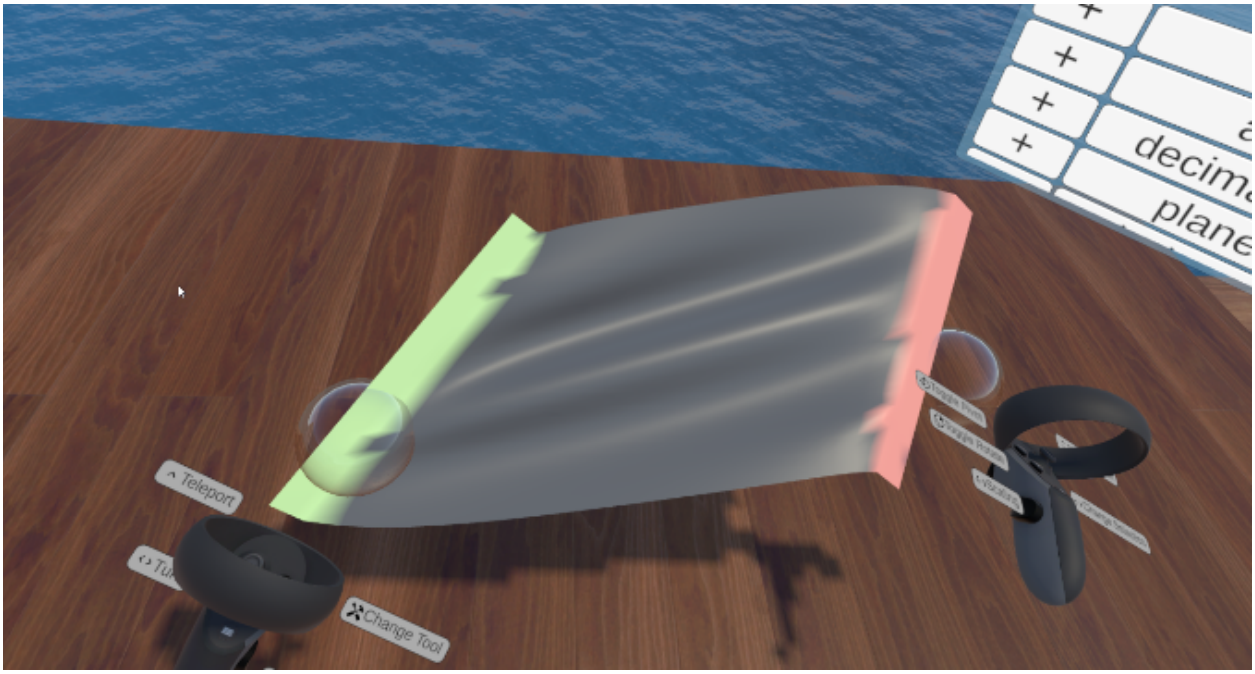

Fig. 1.3: As-Rigid-As-Possible of a **flat sheet**.

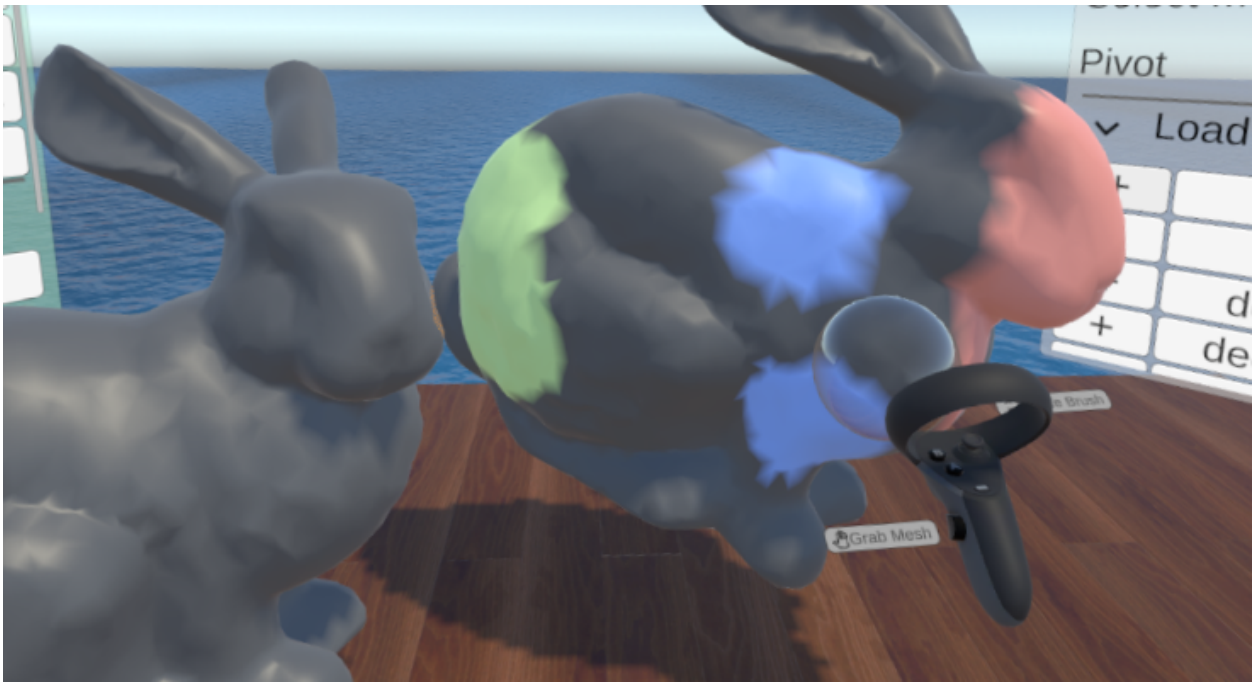

Fig. 1.4: Support for **multiple meshes**. Context sensitive **input hints** can also be seen on the controller.

Fig. 1.5: Biharmonic deform working with varying number of selections as boundaries

Fig. 1.6: As-rigid-as-possible preserves surface details and volume better in comparison to the biharmonic deformation.

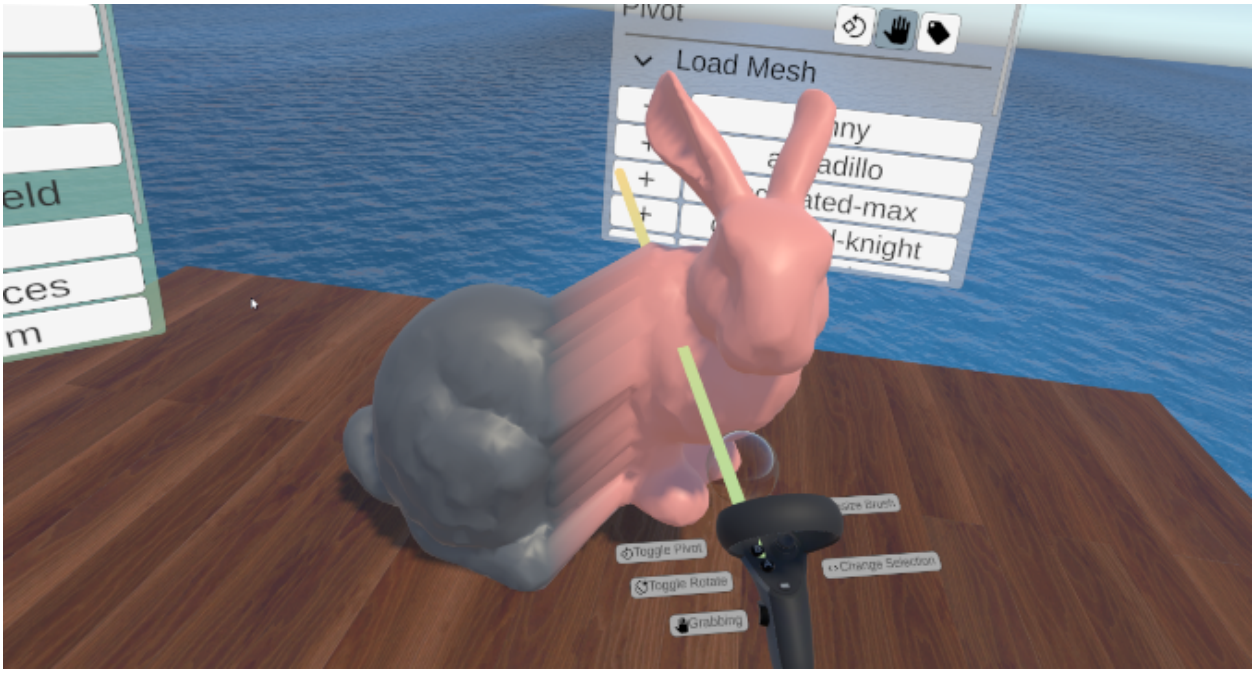

Fig. 1.7: Different **pivot modes** are available - hand, mesh center, selection center. This can be toggled via UI (seen at the top) or via a button.

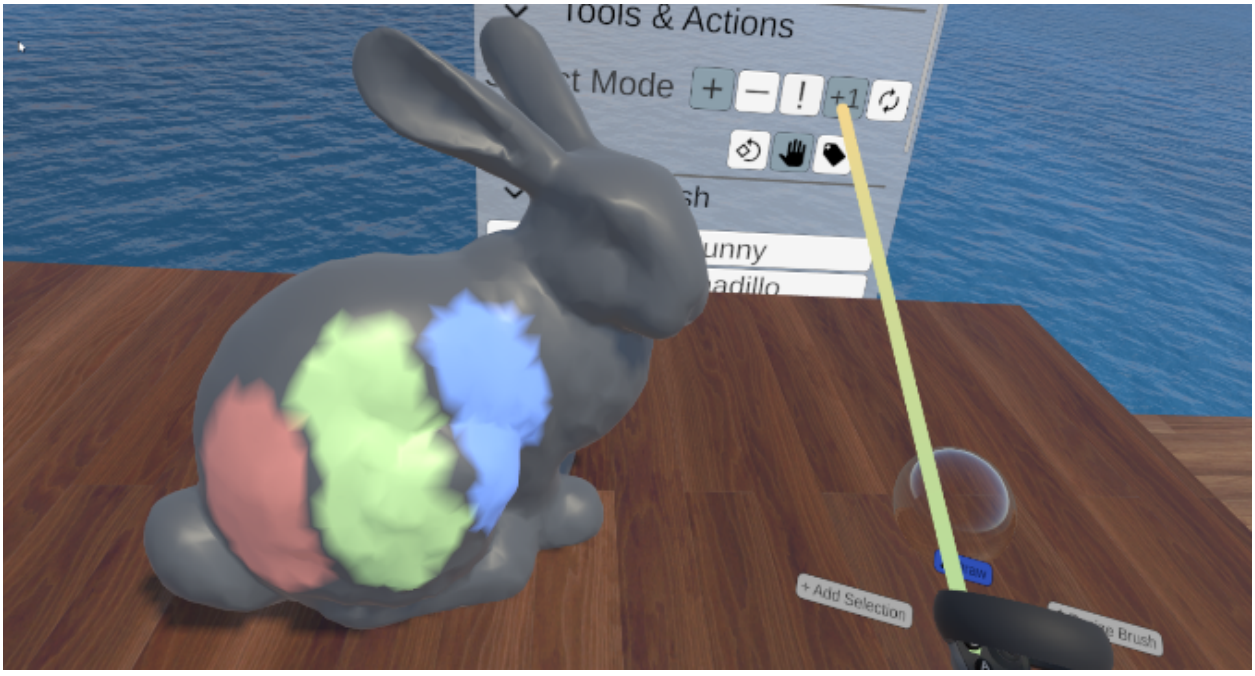

Fig. 1.8: Different modes for **editing the active selection** - add, remove, invert and new/clear selection per stroke.

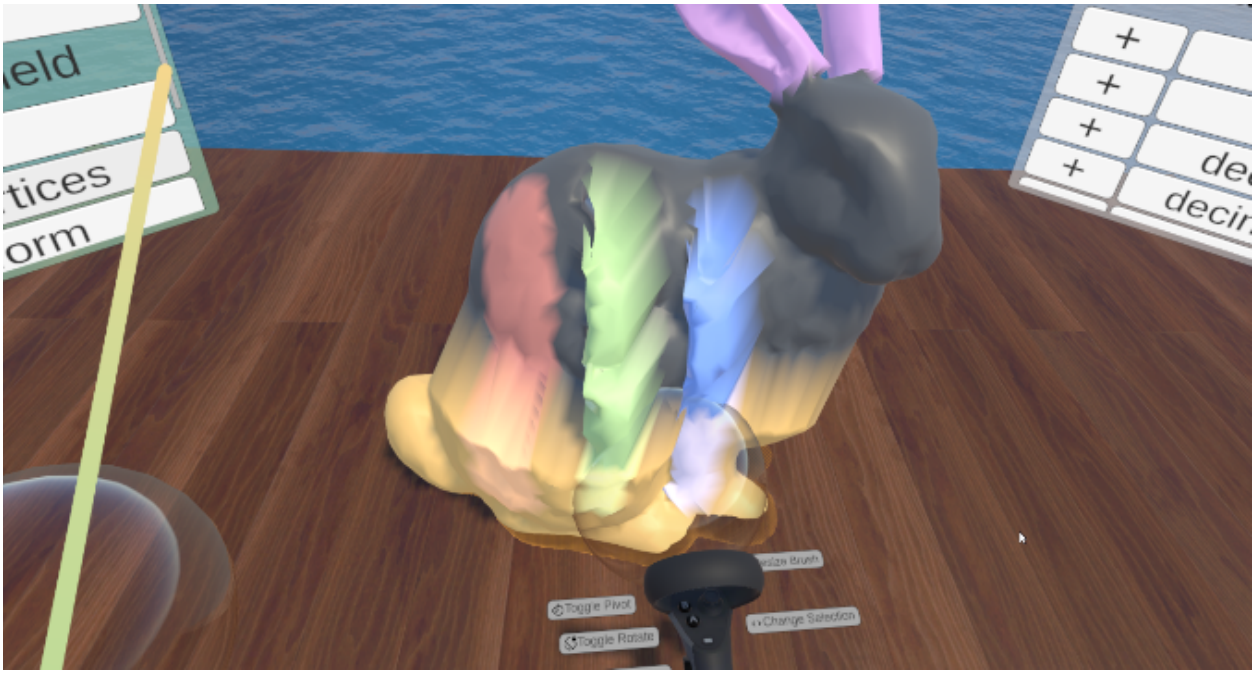

Fig. 1.9: Different ways of selecting **which selection/s should be transformed** based on the bubble brush. When starting a stroke the selection mask inside the brush is used, if empty the active selection mask is used. Only when both hands have the same mask *scaling and two handed rotation* are enabled.

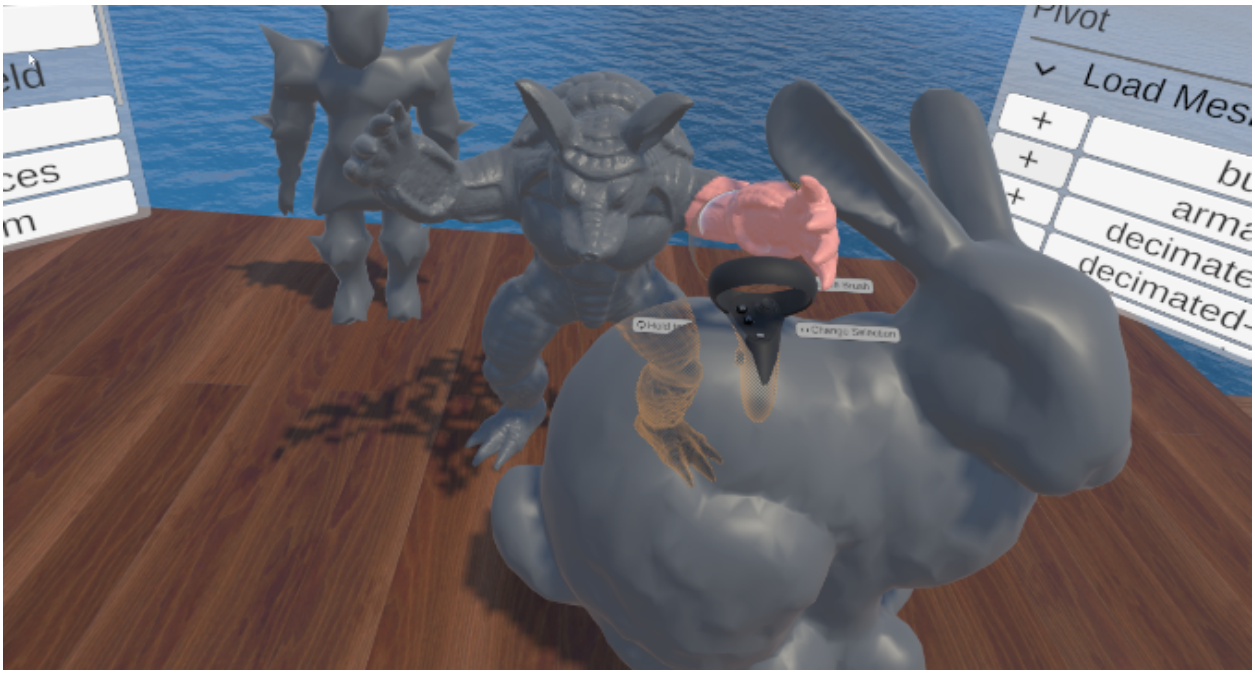

Fig. 1.10: More detailed example of multi-mesh editing. Switching between editing a mesh and transforming the whole mesh (*similar to edit and object mode in Blender*). The holographic shader can also be seen when the active mesh or controllers are occluded.

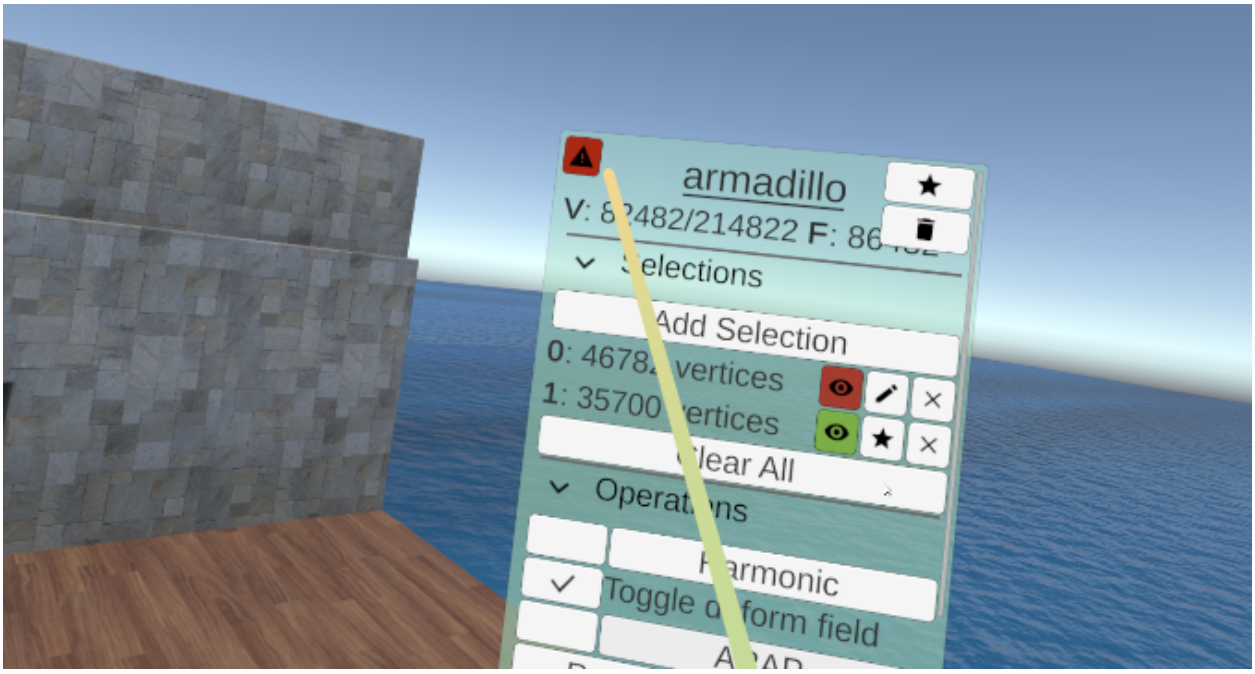

Fig. 1.11: **Performance** is still in real-time for large meshes. If a geometry calculation takes too long the **thread indicator** shows, which is quite useful for debugging.

Fig. 1.12: **Tooltips** are shown on the left hand (green box) when hovering over a UI element. **Input hints** are placed on the controller and adapt based on what you are doing. This makes the app much more user-friendly.

Fig. 1.13: UI panels can be *grabbed*, the laser is only for interaction with the UI and is hidden otherwise.

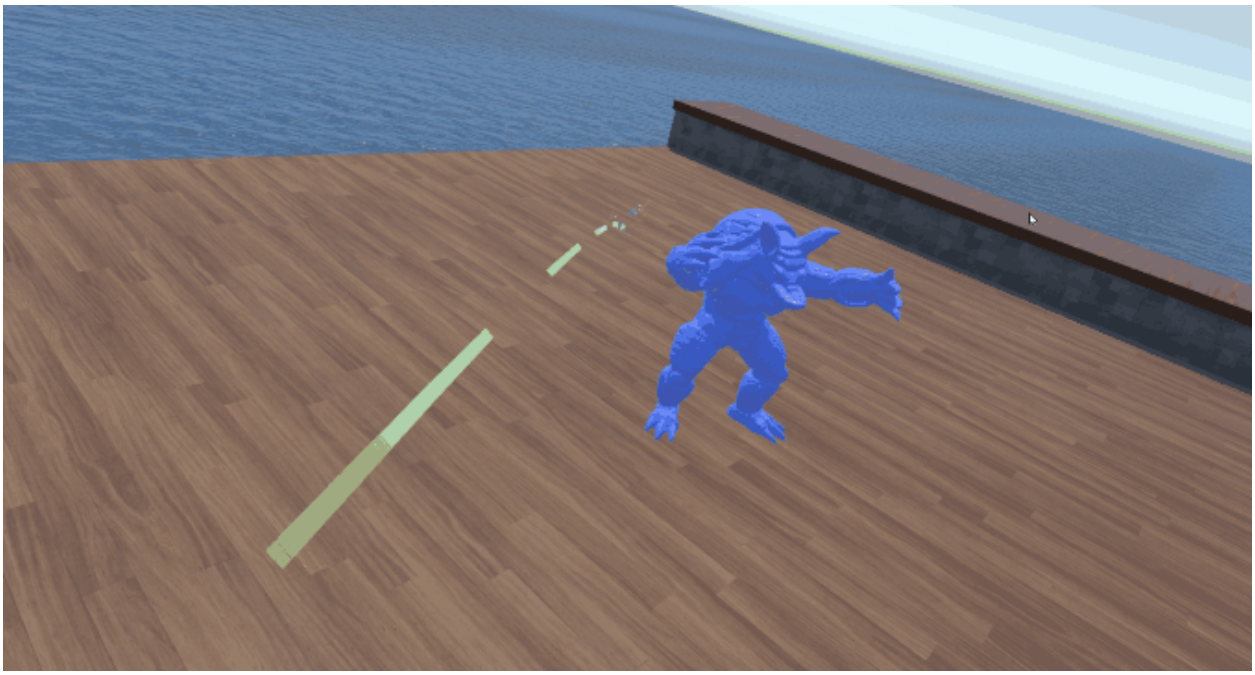

Fig. 1.14: Teleporting and snap turn for **locomotion** in VR

1.1.1 Overview

1.1.2 Deformations

1.1.3 Editing meshes

1.1.4 Misc

1.2 Virtual Reality in Unity

The main areas to consider when making this virtual reality compatible are:

1. **Rendering** - This is the part where Unity helps a lot, setting up all required things. This includes advanced features such as *single pass instanced* rendering, where we gain a lot performance.
2. **Input, Interaction & Locomotion (movement)** - The Unity XR Interaction Toolkit provides this
3. **UI** - We use the standard Unity UI with a *world space canvas*. Canvases are made grabbable so we can easily move them.
4. **Cross-platform** - By using only Unity packages we automatically have that this works for all suitable platforms. A consideration is that some VR devices have limited input (e.g. HTC Vive) and lack some key buttons/axes.

We also need to consider a *different way of interacting* with the world vs a mouse and keyboard.

In particular, we have less degrees of freedom, in terms of the number of buttons, compared to a traditional keyboard and mouse. However, we can easily get absolute positional and rotational input. This indeed does allow for enough flexibility for this kind of application, but *the use of input combinations (or contextual input) is essential*. For example, when drawing a selection by pressing the trigger, the other buttons, should react differently to when not drawing. Currently, this is partially done and reflected in the input hints displayed on the controllers.

1.2.1 Further Reading

- Universal Render Pipeline (URP) v7.1.8, the new render engine
- XR Interaction Toolkit v0.9
- Unity UI, note see Text Mesh Pro (TMP) for text UI

CHAPTER

TWO

DEVELOPER GUIDE

Note: This is aimed at people wanting to view or edit the code. The project is split into two subprojects, a Unity C# project and a C++ library.

2.1 Setup

Required Tools: Unity 2019.3.2f1+, CMake, Visual Studio, 'Desktop development with C++' workload in the Visual Studio installer.

Recommended Tools (Optional): JetBrains CLion (preferably 2020.1+ so you can debug) and Rider for C++ and C# development respectively. This has the benefit that you can debug C# and C++ simultaneously, which is not currently possible with Visual Studio.

2.1.1 After Cloning

Checkout submodules: `git submodule update --init`

Setup the C++ interface to libigl with CMake:

1. Run CMake in the root folder, open the solution in **Visual Studio** and build the target `__libigl-interface`
2. Or setup the CMake project in **CLion** and build, (ensure that the architecture is correct, e.g. `x64` or `amd64`, in the Toolchain settings or you may have errors that the dll was not found).

Open the project in Unity.

1. Open the `Main` scene from the *Project* window.
2. *Reimport* the `Assets/Models/EditableMeshes` folder using the Right-Click menu.
    1. You may also have to reimport the `Main` scene and `Models/Environment Variant` prefab, if you are getting errors.
3. (Optional) Go to the lighting window `Window` > `Rendering` > `Lighting Setting` and press `Generate Lighting` at the bottom.

Press play in Unity and it should work.

2.1.2 Building

CMake Targets

1. `__libigl-interface` - this is the main C++ dll
2. `stubLluiPlugin` - a tiny C++ dll used by the UnityNativeTool (you can leave this alone)
3. `Doxygen` *optional* - builds doxygen html and xml output into `<cmake-build-dir>/docs/doxygen`
4. `Sphinx` *optional* - builds entire documentation (incl. doxygen)
5. `ZERO_CHECK` *Visual Studio only* - re-runs CMake
6. `ALL_BUILD` *Visual Studio only* - builds all targets

Producing an Executable

1. Compile the C++ `__libigl-interface` dll in *release* mode.
2. `Crl` + `Shift` + `B` in Unity to open the build settings.
3. Ensure you are on the platform you want (Windows standalone 64-bit) and set Development mode accordingly, press build.

IL2CPP

This project also works with IL2CPP, which converts C# to C++ upon compile for potential performance gains. These builds are slower.

1. Install the IL2CPP module from the Unity Hub (for this version of Unity).
2. Go to the player settings (either from project settings or from the build window). Find the 'Scripting Backend' and set it to IL2CPP from Mono
3. Build as usual

2.1.3 Generating Documentation

To regenerate this documentation as well as the Doxygen documentation follow these steps. Sphinx is not required for the Doxygen documentation. *Optional*

1. Install Doxygen
2. Install Sphinx, run `pip install -r docs/requirements.txt` from the root directory in a terminal (cmd on windows)
    - You might have to restart for Sphinx to be found
3. Re-run CMake, this will create two new targets, build these like the library
    - **Doxygen**: Creates standard Doxygen html/xml files. View this at `<cmake build folder>/docs/doxygen/index.html`
    - **Sphinx**: Creates the documentation as hosted on ReadTheDocs, using parts from the Doxygen xml output. View locally at `<cmake build folder>/docs/sphinx/index.html` or push to the master branch and then view online.

2.1.4 Project Structure

Important folders:

- `Assets` - Unity related files
- `Scripts` - C# code for things like: UI, Input, Unity mesh interface, Threading, Importing models
  - `Prefabs` - Pre-made components, mostly UI
  - `Models/EditableMeshes` - The meshes that can be modified with libigl
  - `Materials` - Textures, icons and shaders
- `Interface` - C++ project that interfaces with libigl: deformations, modifying meshes via Eigen matrices
  - `source` - the C++ source code which calls libigl
  - `external` - the C++ libraries
- `Packages` - Local Unity packages
- `docs` - non-inline documentation and generation

Generated Folders: `Library` `Temp` by Unity, `obj` by VS

2.2 General Advice

2.2.1 Further Reading

You should make yourself familiar with the Unity engine and some of its packages as well as libigl.

- Valem VR YouTube Tutorials
- XR Interaction Toolkit Documentation
- libigl Tutorial

2.2.2 How to start developing

1. Find a good IDE and learn how to do basic things such as:
   1. Go to definition (!)
   2. Refactor names
   3. Collapse all comments, #regions
   4. Syntax highlighting and intellisense
   5. Display quick documentation, `Crl` + `Q` in JetBrains
2. Setup your debugger so you can set a breakpoint
3. Set up/find out some keyboard shortcuts
4. In C++ use the *`LOG`* macros
5. Have a look at the existing functionality and use it as an example

2.2.3 C# Features

- Use LINQ for manipulating arrays and lists
- Follow some of the advice from the Refactoring Guru
  - Mainly, keep files small and separate independent features.

2.2.4 Use of Bitmasks

Bitmasks are used often in this project for compact/efficient boolean storage. A common example is the selection vector. We have one 32-bit integer for each vertex, equivalent to 32 booleans per vertex. Each bit represents if the vertex is in that selection or not. There are some common operations with bitwise operators you may want to do. Please consider operator precedence.

1. Check if i-th bit is set `(flags & 1 << i) > 0`
2. Set i-th bit `flags |= 1 << i;`
3. Unset i-th bit `flags &= ~(1 << i);`

Note that `1 << i` can also be a mask of several bits or a predefined constant.

2.2.5 VR Hands

- Anything that is related to the controllers always has a left L and right R. This means lots of parameters are essentially duplicated.
- Functions that are independent of these variables are often made and named `*Generic()`
- For booleans: `Left = false`, `Right = true`

2.2.6 Advanced

1. Be aware of the 'Enter Play Mode Settings' in `Project Settings > Editor`
   1. Reload Domain is expensive but ensures that things such as static variables are properly reset when pressing play. This can cause issues not present in a Build, make sure you clean up during OnDestroy.

2.3 Unity Mesh API

This is about how meshes are updated from C++ Eigen matrices to Unity, so that they can be rendered.

Since Unity 2019.3 there have been some updates to the mesh API. However, there are still various limitations and *currently the older API (*`SetVertices`*) is being used* with NativeArray as it is much simpler and thus more reliable. The flowchart below shows the different copies of the mesh data (e.g. vertex position matrix V).

Indeed there are **4 copies of the mesh** currently. Potentially, the 'CPU Unity internal' copy does not exist, but this is unclear. Libigl currently only reliably supports column-major, but Unity requires row-major data. A necessary transpose is required. In this case it is most efficient to have two copies.

Note that updates to the mesh only occur when a *`DirtyFlag`* is set in the *`MeshState.DirtyState`*. *`DirtyFlag`* are propagated and cleared when processed. The native *`ApplyDirty()`* is called by the managed *`ApplyDirty`* in *`UMeshData`*.

It is also important to note that the transposing in *ApplyDirty()* is done on the worker thread. `mesh.SetVertices()` must be called on the main thread (as it is a Unity API). It is called by the managed *ApplyDirtyToMesh* function in *UMeshData*.

Some useful points when working with Unity meshes:

- Call `mesh.MarkDynamic()` on a *readable* mesh to keep a copy of the buffers on the managed CPU memory and make `mesh.vertices` read/writable. This is required in order to be able to modify the vertices at runtime. The mesh must be marked as writable in the import settings in the Inspector.
- Call `mesh.SetVertexBufferParams()` to specify the layout on the GPU, attributes in the same stream are interleaved. Here you can specify the precision as well. See *VertexBufferLayout* in `Native.cs`.
- `mesh.UploadData(false)` to copy the `mesh.vertices` 'CPU Unity internal' managed data to the GPU immediately (else done pre-render), using `true` will delete the CPU copy and the mesh will no longer by dynamic/writable.
- `mesh.GetNativeVertexBufferPtr()` gets the GPU (DirectX/OpenGL) pointer
  - Potentially with this we can apply changes to the GPU copy directly, potentially gaining performance. This has been experimented with in the `source/CustomUploadMesh.cpp`

2.3.1 Further Reading

1. Unity Docs - Mesh
2. Unity Docs - Mesh.SetVertexBufferData
3. Unity Docs - Mesh.SetVertexBufferParams
4. Sample from GraphicsDemos
5. Mesh API Feedback Forum with link to Google Docs
6. NativeArray use cases

2.4 Adding Functionality

2.4.1 Threading PrePostExecute Cycle

Note: The Unity API is not thread safe, only simple methods like Debug.Log can be used in a thread.

In order to keep the virtual reality experience responsive and at high framerates, all the geometry and libigl calls are on a worker thread. Each mesh has its own thread. As the Unity API is not thread safe, it can only be accessed from the main thread. API calls must be made in *PreExecute* and their results copied to the thread via the *MeshInputState*. Because of this we have the cycle shown above.

- *PreExecute* - this is where any preparation is done that needs to be on the main thread. The shared *InputState InputManager.State* is copied. The *MeshInputState Input* is copied to the thread version *_executeInput*.
- *Execute* - depending on the *_executeInput* we call different code, e.g. if *MeshInputState.DoSelectL* is true we modify the selection. This is where most C++ functions are called.
- *PostExecute* - this is where changes are applied to the Unity mesh.

- `Update` - this is the *separate* Unity callback called every frame, in this we update the *`LibiglBehaviour.Input`* state.

Note that only operations done in *`Execute`* do not affect the framerate.

See also Unity Execution Order of Events.

2.4.2 Data Storage

There is a lot of data tied to the mesh and the user input. It is important to know where what is stored and where to add new data.

Note: Please ensure that you understand the difference between a C# *class* and *struct* (reference type vs. value type), particularly that a struct is copied when passed as an argument or with the = operator.

Generally, data falls into one of the following categories:

1. **Input data**, Data that is used to parametrize and decide what is executed (usually from UI/Input). This belongs to the C#. Members are passes as arguments to native functions.
    1. *`InputState`* if shared between meshes, this contains the raw controller input
    2. *`MeshInputState`* if specific to a mesh
2. **Mesh data**
    1. Vertex/Face data required for rendering the mesh, this is the most complicated. It must be part of the *`MeshState`* and shared between C# and C++. There is a lifecycle to this detailed in *Applying Mesh Data To Be Rendered*.
    2. Data that is used only for computations, this belongs to the C++ only *`MeshStateNative`*.
    3. (uncommon) data that must be shared between C++ and C#, such as results of a computation (e.g. selection size). This also belongs to the *`MeshState`*.

Note: Data shared between C#/C++ requires more effort to maintain as everything must be declared twice. For this reason all the input data is C# only.

2.4.3 Customizing UI

This is relatively easy. There are two categories:

1. **Mesh specific UI**, see *`UiMeshDetails`*
2. **Generic UI**, see *`UiManager.InitializeStaticUi`*

In the *`UiManager`* instance there are several prefabs that can be used, e.g. *`buttonPrefab`*. These are built up from the Unity UI (*ugui not the new UIElements*). These often have a custom script in `UI.Components` attached to the root transform for easy modification. There are lots of simple examples for this so just have a look at the code. The normal workflow is:

1. Instantiate a prefab and get the gameObject or custom script (e.g. *`UiSelectionMode`*)
2. Add the gameObject to a group/collapsible/category
3. Initialize the custom script or setup the OnClick callbacks directly

Generic example:

```
var selectionMode = Instantiate(selectionModePrefab, actionsListParent).GetComponent
↪<UiSelectionMode>();
_toolGroup.AddItem(selectionMode.gameObject);
selectionMode.Initialize();
```

Mesh specific example without a custom component, note how we can access the *LibiglBehaviour _behaviour*:

```
var clearAllSelections = Instantiate(UiManager.get.buttonPrefab, _listParent).
↪GetComponent<Button>();
_selectionGroup.AddItem(clearAllSelections.gameObject);

clearAllSelections.GetComponentInChildren<TMP_Text>().text = "Clear All";
clearAllSelections.onClick.AddListener(() => { _behaviour.Input.DoClearSelection =␣
↪uint.MaxValue; });
```

Warning: Be careful not to add an *excessive* amount of UI as raycasting the UI is (surprisingly) one of the most performance intensive operations currently.

To *add your own UI component* create a prefab in the `Assets/Prefabs/UI/Components` folder, then add a reference to it in the *UiManager* so you can access it from C#. It is common to add a MonoBehaviour to the root of the prefab, which you can then use to initialize it. See the scripts in `UI/Components`.

2.4.4 Customizing Input

To make the input adapt to what the user is doing we define the mapping of raw input to actions based on a context or state. How this state is determined is detailed below. It is based on a tree structure. See *ToolTransformMode*, *ToolSelectMode* and *UpdateInputTransform*, *UpdateInputSelect* respectively.

Input Hints

To make it easy for the user to see what each button will do UI is displayed over the controller, called input hints in this project. To update the UI the same state is used. This means if you change what the input does you also need to update the input hints.

However, to make it easy for the developer to specify these hints (icon+text) ScriptableObject s are used: *UiInputHintsDataCollection*, *UiInputHintsData*, *UiInputHintsData*. These allow for data to be entered inside the Editor, not in C#.

This means common hints are inherited based on the same tree that is used to determine the state. The *UiInputHints.Repaint* function will most likely give the best insight to how this works.

2.4.5 Importing Meshes

If you just want to add a new mesh, **add it into** `Assets/Models/EditableMeshes` and follow the warnings in the Unity console when running. The mesh will be checked for its validity. Note for `.off` files you need to have built the C++ library first.

Then add it the *MeshManager* Mesh Prefabs list in the Main scene on the `Editable Meshes` GameObject.

You might need to reimport it from the right-click menu if the C++ library has not been built.

Custom Mesh File Formats (Advanced)

This is about how the meshes are actually imported.

See `Libigl/Editor/`

There are two cases:

1. File extensions that Unity recognizes and already imports. For these we just post-process the imported mesh. See *MeshImportPostprocessor*
2. File extensions unknown to Unity, e.g. `.off` meshes. For these we write an (experimental) ScriptedImporter and import the mesh via libigl. See *OffMeshImporter*

Note that in the end Unity still does the importing in both cases in the Editor. For non-mesh files, e.g. dense matrices, these can be loaded at runtime directly from C++ with libigl. This Unity API is still experimental so there may be some errors in the future.

2.4.6 Applying Mesh Data To Be Rendered

To apply changes made to the vertex position matrix V, or any of the other matrices in the *MeshState*, you need to set the *MeshState::DirtyState* with the appropriate *DirtyFlag*. This tells the system what has changed and the rest will be done automatically. For more control you might want to see `IO.cpp` *ApplyDirty()*.

This is only for data that has to be made available to Unity to render the mesh.

Mesh Data Lifecycle (Advanced)

This details how changes to the mesh are propagated to Unity and its renderer. The example is done with the vertex matrix V, but works also for the other data in *MeshState*.

Note: Unity stores its mesh data in **Row Major**, whereas libigl requires **Column Major**, a necessary conversion by transposing has to be made.

---

1. *(in Execute)* The developer modifes the V matrix and sets it as dirty: `state->DirtyState |= DirtyFlag.VDirty`
2. *(in PostExecute)* *ApplyDirty()* is called to apply the changes from the *MeshState* to the Unity row major copy pointed to in *UMeshDataNative*. Here we also filter out only things that have changed. This is called by `UMeshData.cs`.
3. Once this transposing is done, we pass the data to Unity in *UMeshData.ApplyDirtyToMesh* in C#

2.4.7 Custom Deformation

The above diagram indicates the important parts of implementing a deformation, with the example for the `igl::harmonic()` Biharmonic 'smoothing' deformation.

To add a new deformation there are several things that need to be done. The approach I often use is to start with the complicated C++, then the C# interface and end with the UI/input (roughly in reverse order to the execution):

1. How the deformation is carried out in the C++, see `Deform.cpp`
    1. Be sure to set which data from the mesh you have changed with the `State->DirtyState` variable, see *Applying Mesh Data To Be Rendered*
2. Storing your data in the right place, see `MeshState*.h` and *Data Storage*
3. Making this callable from C#: declare in `Native.h` and redeclare in `Native.cs`
4. Integrating with the Pre/Post/Execute threading loop, see `LibiglBehaviour*.cs` and `MeshInput.cs` as well as *Threading PrePostExecute Cycle*
    1. How this deformation is executed from C#, see `Libigl/LibiglBehaviour.Actions.cs`
    2. Handling of the controller input to determine when/how to execute the deformation, see `LibiglBehaviour.Input.cs`
5. Parametrization in the 2D UI, see examples of UI generation in `UI/UiDetails.cs`

This is indeed quite a long list because of the following complications:

1. C#/C++ interface
2. Threading in Unity, not being able to access any of the Unity API from a worker thread.

2.4.8 Custom Shader

This is quite easy with the new Unity **Shader Graph**. So no HLSL/GLSL is required for most things. Have a look at the default shader `Materials/Shaders/VertexColor` being used on the meshes. Note that to display the shader in Unity you must put the shader into a material and then attach that to the mesh renderer component on the GameObject. See `Materials/EditableMesh.mat` which uses the `VertexColor` shader.

Vertex data, e.g. vertex position or uv coordinates, can be accessed via a node in the graph.

2.5 Documentation Meta

Documentation is generated with Doxygen and then fed to Sphinx via the Breathe extension. A custom C# domain sphinx-csharp is used to be able to render the C# code as well. By using the m2r sphinx extension markdown is converted to reStructuredText `.rst` which is what sphinx expects. This allows us to write markdown with inline rst. rst directives are used to access the Doxygen documentation (see breathe).

Code is documented with xml-doc in C# and javadoc in C++ currently. Using `Crl + Q` in JetBrains IDEs allows displaying the documentation quickly inline when developing.

Note: The markdown files are meant to give an overview, but most documentation should be in the code itself. The recommended cross-platform md/rst editor that I use is Typora, see https://typora.io

---

2.5.1 Modifying the Documentation

- Annotate your code (e.g. functions, vars) with either xml-doc or javadoc
- You can add a new md file but it must be referenced in a `toctree` directive (usually found in `index.*`). CMake must re-run when doing this.
- Ideally have a markdown file per source directory
- Inline diagrams/flowcharts are made with diagrams.net (previously draw.io) and inserted as an iframe
  - These are stored on Google Drive
- Gifs in the gallery are created with ScreenToGif

After making changes fast forward the `read-the-docs` branch to the latest commit on the master branch. This will trigger read the docs to rebuild and update the online documentation.

CHAPTER

THREE

C#/UNITY REFERENCE

This is aimed at people wanting to view or edit the Unity related code.

The `Scripts` folder includes all C# code. However, a lot of functionality (or parametrization) is also inside the scene and prefabs(!)

Generated/unimportant folders:

- `Main`, lighting data
- `Plugins`, dlls
- `Settings`
- `Materials/TextMesh Pro`

External Assets:

- Icons are from material.io - Apache License version 2.0
- Textures from poliigon.com - free section
- Wireframe shader from Chaser324/unity-wireframe on GitHub - MIT License
- VR controller models from Valem YouTube (originally OpenVR/Steam/Valve) Introduction to VR in Unity - PART 2: Input and Hand Presence
- EditableMeshes from IGL, parts from the Stanford 3D Scanning Repository

Note: This part assumes a reasonable understanding of Unity.

3.1 Scene

This will elaborate on the GameObject s in the scene and how we assemble the final application from the C# scripts and prefabs.

Note: Anything that is not inside the scene or referenced/invoked by it will not be in the final build. Most components provide tooltips, hover over an item to see what it does.

Editable Meshes is the parent for all meshes that will be modified by libigl as well as a spawn point to define where new meshes are positioned.

XR Rig contains the cameras and controllers. All components related tracking the head and hands is here. Notably the controllers have a lot of settings that you can tweak related to how the rays are shown and how you can interact with the scene. This is also where the *InputManager* instance is.

UI holds all UI items and has the *UiManager* instance attached.

Environment holds all static meshes, visual items as well as the lighting objects.

Dll Manipulator is and essential object which is only used in the Editor. If you are getting errors related to the using or building the dll, have a look at this object. Note that if the scene is not loaded then this script will not be running. It also runs during edit mode (when outside of play mode).

3.2 Prefabs

Note: Please make sure you are familiar with Unity prefabs and prefab variants, as well as their icons/colors in the hierarchy.

3.2.1 UI

This contains mostly instances of the UI components used when generating the UI panels.

XrCanvas is the 'base' prefab upon which all world space canvases are based on. This contains functionality on how to raycast the UI, how to grab the panels.

MeshDetailsCanvas is an empty details panel for a mesh, before any components have been added via C#.

VerticalScrollList defines the layout of generated elements. Elements are added to the `Content` child.

InputHints is the 'base' prefab for the hints shown on a controller. It is based on the Oculus Touch left hand controller and is used in the VR Controller prefabs.

InputLabel is one label used in the InputHints. Links to one button or axis. It is made so that it fits the content.

3.2.2 UI Generator Components

These are items that can be added to the **VerticalScrollList**, in the **XrCanvas**, from C#. These are all referenced by the instance of the *UiManager* in the scene (found on the UI gameObject) under the `UI Component Prefabs` header.

3.2.3 UI Input Hint Data

These aren't technically prefabs but ScriptableObject s, which is a simple data storage (compared to a database). This is where we store what the input hints will display. It is done in a hierarchical fashion with one instance per state and sub-state. This is used by *UiInputHints.Repaint*. See also *UiInputLabel.SetData*

3.2.4 VR Controllers

This is where you can customize the look of the VR controllers, as well as tweaking the positions of the UI input hints for example.

3.2.5 XR Interaction

These are prefabs related to the interaction with the world.

3.3 C# Scripts Overview

The important overall picture is covered here, for more detailed notes about the implementation see the code itself, `Crl + Q` with Rider or generate the doxygen documentation locally with CMake.

In the `Scripts` folder, the subfolders correspond to the namespaces. Beware that `Editor` folders are treated specially by Unity and will not be included in a build.

- **Libigl** - This is where most of the interesting code is: modifying/updating the geometry, making calls to C++, threading, anything related to the meshes.
  - **Editor** - Scripts related to importing and pre-processing models so that we can modify them with libigl.
- **XrInput** - This is where interface with the controllers is, reading values, setting up controllers when detected, note setting up tracking is done in the scene with the XR Interaction Toolkit components
- **UI** - This is about the 2D user interfaces. How to generate the UI panels, update them and defining the click behaviour.
  - **Components** - This contains lots of smaller scripts to define the behaviour of a generated component. See `Assets/Prefabs/UI`.
  - **Hints** - Behaviour and data related to displaying 2D UI hints over the controllers to show what each button does. This is quite context sensitive for each tool and sub-state. Uses *ScriptableObjects* so that we can enter the data in the Unity Editor.
- **Util** - Some helper scripts
- **Testing** - Scripts here are not used but might be of interest to see how the Unity APIs work (particularly the mesh API).

> **Warning:** The C# code documentation is *'custom'* made. If something is displayed completely incorrectly, please create an issue so it can be fixed. You can always see the **doxygen** documentation locally or **view the code comments** themselves in the IDE.

3.3.1 Libigl Overview

namespace Libigl

class DirtyFlag

Marks which data has changed in *UMeshData* as a bitmask and needs to be applied to the mesh.

This is used to selectively update the Unity mesh and is only for data that Unity requires. Use these constants along with the bitwise operators.

class LibiglBehaviour : **public** IDisposable

This is where the behaviour that interfaces with libigl is. It follows a Pre/Post/Execute threading pattern. This is a `partial` class, meaning it is split between several files.

Input: This handles collecting data from the main thread for the worker thread. It sets up the *MeshInputState* for the Actions. This is where we decide *what* to execute on the worker thread.

Actions: This handles executing things on the worker thread. **Actions**, in this context, are things that can be executed on the worker thread. They are the entry points to the C++ code (mostly). These actions correspond to the Do* variables in the *MeshInputState*

Transform: This handles anything related to transformations of the mesh or selections. This is only used for calculating which transformation to do for the selections. The code for applying the transformation to selections is in *Actions*

See also *Libigl.LibiglMesh* which handles the threading and calls the Pre/Post/Execute callbacks.

class LibiglMesh : **public** MonoBehaviour

This component needs to be attached to any GameObject that you want to modify with libigl.

Any libigl related code is defined in the *LibiglBehaviour* class. This class only handles the threading and connection with the Unity Mesh components.

struct MeshInputState

Struct for storing the current input *for a mesh*. (This is a value type so assigning will copy). Anything that may change as we are executing should be in the InputState, as it is copied in PreExecute. Anything that is the same between all meshes may be put into the InputState. Anything related to what should be executed on the worker thread should be put here (e.g. DoSelect).

Variables that correspond to triggering Actions start with `Do` e.g. `DoSelect`

class MeshManager : **public** MonoBehaviour

Handles loading EditableMeshes and stores references, notably to the *ActiveMesh*.

struct MeshState

The **C++ state for a mesh** in column major. This is linked to Unity and the RowMajor version via *UMeshData*. It stores only pointers to the Eigen data so this can be shared between C++ and C#. The mesh data is allocated in C++ during the Native.InitializeMesh function using *UMeshDataNative*.

The *DirtyState* indicates what has been changed and needs to be applied to the Unity mesh.

As this struct is shared between a managed (C#) and native (C++) context, you must consider Marshalling when adding new variables.

`void*` usually represents an Eigen matrix, but can be anything.

class Native

Contains all C++ function declarations. **C# to C++**

Handles initialization of the DLL and works with the UnityNativeTool for easy reloading/recompilation.

Convention: Pass the *MeshState* as the first argument if the function modifies a mesh.

class NativeCallbacks

Contains all callbacks from the native context. **C++ to C#**

Callbacks should be annotated with the MonoPInvokeCallbackAttribute so that IL2CPP builds will compile properly. Each callback needs to have a corresponding delegate (/type).

struct TransformDelta

Stores information about a single transformation.

class UMeshData : **public** IDisposable
Stores a copy of the Unity Mesh's arrays. This is purely for the interface between the Libigl Mesh *MeshState* and the Unity Mesh / what will be rendered. Important: Uses RowMajor as that is how it is stored by Unity and on the GPU.

struct UMeshDataNative
Stores pointers to the native arrays in *UMeshData* so we can pass this to C++. Pointers are to the first element in the respective NativeArray.

Important: As Native arrays are not managed memory, the underlying array is fixed and will not move due to Garbage Collection. So an instance's pointers will remain valid.

namespace Editor

class MeshImportPostprocessor : **public** AssetPostprocessor
Used for post-processing meshes after importing in the Editor. Only for file formats that Unity recognizes. Simplified version of *OffMeshImporter*

class OffMeshImporter : **public** ScriptedImporter
A custom importer for .off mesh files, which Unity does not recognize or know how to import by default. See tooltips.

3.3.2 XrInput Overview

namespace XrInput

Enums

enum ToolType
Which tool is being used, *InputState.ActiveTool*.

Values:

Transform

Select

enum ToolSelectMode
The sub-state of the Select tool.

Values:

Idle

Selecting

TransformingL

TransformingR

TransformingLr

enum ToolTransformMode
The sub-state of the Transform tool.

Values:

Idle

TransformingL

TransformingR

TransformingLr

enum SelectionMode
How to modify the selection.

Values:

Add

Subtract

Invert

enum PivotMode
How to rotate the mesh/selection.

Values:

Mesh

Hand

Selection

class InputManager : **public** MonoBehaviour
This script handles the controller input and is based on the Unity XR Interaction Toolkit. An important part is that this is where the *InputState* can be accessed and is updated.

struct InputState
This is the 'shared' input state. It is also where you can access the filtered controller input. Stores input that is shared between meshes as well as the raw input that has not been mapped to actions. Raw input has been filtered to prevent conflicts with UI and grabbables.

class XrBrush : **public** MonoBehaviour
Functionality related to the sphere 'bubble' brush. Currently handles resizing the brush, getting the center and finding overlapping bounding boxes via trigger colliders.

3.3.3 UI Overview

namespace UI

class Icons : **public** MonoBehaviour
Stores references to icon sprites. Names are mostly the same as the asset names.

class UiManager : **public** MonoBehaviour
Handles easy creation of operations to be done on a mesh and the user interaction (2D UI, speech, gestures) that comes with it.

class UiMeshDetails : **public** MonoBehaviour
Contains all functionality related to the mesh UI panel. There is one of these per LibiglMesh. Contains most UI generation.

namespace Components

class UiCollapsible : **public** MonoBehaviour
UI Component header that hides items when clicked/toggled.

class UiIconAction : **public** MonoBehaviour
UI Component with two buttons, one icon sized.

class UiPivotMode : **public** MonoBehaviour
Similar to *UiSelectionMode*, effectively an enum field.

class UiProgressIcon : **public** MonoBehaviour
Handles showing the progress icon and animating it based on the PreExecute and PostExecute. This indicates the state of the libigl thread.

class UiSelection : **public** MonoBehaviour
UI for one selection of a mesh (one row)

class UiSelectionMode : **public** MonoBehaviour
Handles the selection mode UI, onclick behaviour. There are several modes from *SelectionMode* as well as the *newSelectionOnDrawBtn* where a new selection is added on each stroke.

class UiToggleAction : **public** MonoBehaviour
A toggle and button UI element. The toggle and button are independent by default

namespace Hints

class UiInputHints : **public** MonoBehaviour
Defines the behaviour of the input hints of one hand. Important functions are: AddTooltip(GameObject, string), *SetData* and *Repaint*.

In short, tooltips can be added so when we hover over a UI element it displays some text. Repaint is called with the collection to set out the default hints for a particular state. This can then be overriden by scripting, e.g. see RepaintTriggerColor where we set the trigger hint color ot the active selection color.

class UiInputHintsData : **public** ScriptableObject
Data for one hand and one state of the ActiveTool. Used by the *UiInputHints*. We have one *UiInputLabelData* per button/axis.

class UiInputHintsDataCollection : **public** ScriptableObject
Stores the hints for all possible states of the ActiveTool and sub-states. These are stored as a hierarchy. *UiInputHints.Repaint* defines how this data is applied. There will be one instance for the left and one for the right hand.

class UiInputLabel : **public** MonoBehaviour
A label for a physical input button/axis to give a hint to what it does.

struct UiInputLabelData
Defines the content for a *UiInputLabel*.

3.4 C# Libigl

3.4.1 MeshManager.cs

class Libigl.MeshManager : **public** MonoBehaviour
Handles loading EditableMeshes and stores references, notably to the *ActiveMesh*.

Public Functions

LibiglMesh **LoadMesh** (GameObject *prefab*, bool *unloadActiveMesh* = *true*, bool *setAsActiveMesh* = *true*, bool *performValidityChecks* = *false*)
Instantiate a mesh that can be used with libigl

Parameters

- `prefab`: Prefab to be created, see *meshPrefabs*
- `unloadActiveMesh`: Delete the active mesh if it exists
- `setAsActiveMesh`: Set this mesh as the active one
- `performValidityChecks`: Check that the prefab can be properly used with libigl.

Meshes from the *meshPrefabs* list are checked during Start and do not need to be checked again.

Return *LibiglMesh* component on the new instance, null if there was an error

void **DestroyMesh** (*LibiglMesh libiglMesh*)
Use this to delete a mesh safely. Handles case when mesh is the active one, *ActiveMesh*

void **RegisterMesh** (*LibiglMesh libiglMesh*)

void **UnregisterMesh** (*LibiglMesh libiglMesh*)

Public Members

GameObject[] **meshPrefabs**

Transform **meshSpawnPoint**

readonly List<*LibiglMesh*> **AllMeshes** = new List<LibiglMesh>()
List of all LibiglMeshes that are instantiated

Material **wireframeMaterial**

Material **wireframeMaterialPrimary**

Material **wireframeMaterialActive**

GameObject **boundingBoxPrefab**

Events

event Action **OnActiveMeshChanged** = delegate { }
Invoked when the *ActiveMesh* is changed. Called after initialization, if the mesh is newly instantiated.

Public Static Functions

bool **CheckPrefabValidity** (GameObject *prefab*)
Checks if a prefab can be loaded and modified with libigl. Does not modify the prefab only logs errors.

Return True if the prefab can be used with libigl

void **SetActiveMesh** (*LibiglMesh libiglMesh*)

Public Static Attributes

MeshManager **get**

LibiglMesh **ActiveMesh**
The mesh currently loaded and being modified

Private Functions

void **Awake** ()

void **Start** ()

Private Static Functions

void **DestroyActiveMesh** ()

Private Static Attributes

int **_defaultLayer**

int **_holographicLayer**

3.4.2 LibiglMesh.cs

class Libigl.LibiglMesh : **public** MonoBehaviour
This component needs to be attached to any GameObject that you want to modify with libigl.

Any libigl related code is defined in the *LibiglBehaviour* class. This class only handles the threading and connection with the Unity Mesh components.

Public Functions

bool **IsJobRunning** () **Return**
True if a job/worker thread is running on the MeshData

bool **IsActiveMesh** () **Return**
True if this is the active mesh set by the *MeshManager*

void **Initialize** ()

void **ResetTransformToSpawn** ()
Move mesh up so it is resting on the spawnPoint, ie min of bounding box is at spawnPoint

Parameters

- `mesh`: Needed for bounding box

void **SetWireframe** (bool *value*)
Toggles the wireframe shader for the mesh (not the bounding box).

void **UpdateBoundingBoxSize** ()
Adjust the bounding box visual to fit the true bounds of the mesh.

void **RepaintBounds** (bool *overrideVisible* = *false*, bool *primary* = *false*)
Shows or hides the wireframe bounding box. Also changes the material accordingly.

Parameters

- `overrideVisible`: Override default visibility set in the InputState
- `primary`: If overriding visibility, should we highlight this as the primary bounding box

Public Members

Transform **BoundingBox**

Properties

Mesh **Mesh** { }
The Unity mesh.

MeshRenderer **MeshRenderer** { }

UMeshData **DataRowMajor** { }
The Unity Mesh data in RowMajor easily accessible as NativeArrays

LibiglBehaviour **Behaviour** { }
The libigl behaviour instance that is executing on this mesh

Private Functions

void **OnActiveMeshChanged** ()

void **Update** ()

void **ExecuteThread** ()
Creates a thread with the *LibiglBehaviour* code

Assert: _workerThread is null (finished and PostExecuteThread has been called)

void **PostExecuteThread** ()
Applies changes and cleans up the threading for re-use once the thread has finished.

Assert: _workerThread is finished executing.

void **OnDestroy** ()

void **Dispose** ()

Private Members

MeshFilter **_meshFilter**

Thread **_workerThread**
Expensive operations executed in *LibiglBehaviour.Execute* are done in this thread

MeshRenderer **_boundingBoxRenderer**

3.4.3 UMeshData.cs

class Libigl.UMeshData : **public** IDisposable
Stores a copy of the Unity Mesh's arrays. This is purely for the interface between the Libigl Mesh *MeshState* and the Unity Mesh / what will be rendered. Important: Uses RowMajor as that is how it is stored by Unity and on the GPU.

Public Functions

UMeshData (*Mesh mesh*)

Parameters

- `mesh`: Unity Mesh to copy from

unsafe void **LinkBehaviourState** (*LibiglBehaviour behaviour*)
Initialize the *UMeshData* with shared data with the *behaviour* .

unsafe void **ApplyDirty** (*MeshState *state*, *MeshInputState inputState*)
Applies changes to the C++ State to this instance. Use this to copy changes from Col to RowMajor.

Can and should be called from a worker thread. Behind the scenes this tranposes and copies the matrices.

The DirtyState is propagated so *ApplyDirtyToMesh* (called on the main thread) will apply the changes.

See Native.ApplyDirty

void **ApplyDirtyToMesh** (*Mesh mesh*)
Apply MeshData changes to the Unity Mesh to see changes when rendered. Uses the *DirtyState* to detect what needs to be applied.

Must be called on the main thread as it accesses the Unity API.

Assert: *IsRowMajor* is true.

UMeshDataNative **GetNative** ()
Note: Changes to the dirtyState are not applied to the MeshData instance (not a reference) and needs to be set manually in a C# context.

Important: The struct itself should be treated as `const` as changes have no effect (it's a copy).

Return A MeshDataNative instance than can be passed to C++ containing all pointers

void **Dispose** ()

Public Members

const bool **IsRowMajor** = true

uint **DirtyState** = DirtyFlag.None

uint **DirtySelections** = 0

uint **DirtySelectionsResized** = 0

NativeArray<Vector3> **V**

NativeArray<Vector3> **N**

NativeArray<Color> **C**

NativeArray<Vector2> **UV**

NativeArray<int> **F**

NativeArray<uint> **S**

readonly int **VSize**

readonly int **FSize**

Private Functions

void **Allocate** (*Mesh mesh*)
Allocated the NativeArrays once VSize and FSize have been set.

void **CopyFrom** (*Mesh mesh*)
Copies all data (e.g. V, F) from Unity mesh into the **already allocated** NativeArrays

Private Members

UMeshDataNative **_native**
Stores pointers to the native arrays, we can pass this to C++

DirtyFlag

class Libigl.DirtyFlag
Marks which data has changed in *UMeshData* as a bitmask and needs to be applied to the mesh.

This is used to selectively update the Unity mesh and is only for data that Unity requires. Use these constants along with the bitwise operators.

Public Members

const uint **None** = 0

const uint **VDirty** = 1

const uint **NDirty** = 2

const uint **CDirty** = 4

const uint **UVDirty** = 8

const uint **FDirty** = 16

const uint **DontComputeNormals** = 32
Don't recaluclate normals when VDirty is set, NDirty overrides this.

const uint **DontComputeBounds** = 64
Don't recalculate bounds when VDirty is set. Bounds are used for occlusion culling.

const uint **DontComputeColorsBySelection** = 128
Don't recompute colors if a visible selection has changed.

const uint **VDirtyExclBoundary** = 256
Use this when the vertex positions have changed, but the boundary conditions are unaffected. VDirty overrides this.

const uint **All** = uint.MaxValue - DontComputeNormals - DontComputeBounds

UMeshDataNative.cs

struct Libigl.UMeshDataNative
Stores pointers to the native arrays in *UMeshData* so we can pass this to C++. Pointers are to the first element in the respective NativeArray.

Important: As Native arrays are not managed memory, the underlying array is fixed and will not move due to Garbage Collection. So an instance's pointers will remain valid.

Public Functions

UMeshDataNative (float **vPtr*, float **nPtr*, float **cPtr*, float **uvPtr*, int **fPtr*, int *vSize*, int *fSize*)

Public Members

readonly float* **VPtr**

readonly float* **NPtr**

readonly float* **CPtr**

readonly float* **UVPtr**

readonly int* **FPtr**

readonly int **VSize**

readonly int **FSize**

3.4.4 LibiglBehaviour.cs

class Libigl.LibiglBehaviour : **public** IDisposable
This is where the behaviour that interfaces with libigl is. It follows a Pre/Post/Execute threading pattern. This is a `partial` class, meaning it is split between several files.

Input: This handles collecting data from the main thread for the worker thread. It sets up the *MeshInputState* for the Actions. This is where we decide *what* to execute on the worker thread.

Actions: This handles executing things on the worker thread. **Actions**, in this context, are things that can be executed on the worker thread. They are the entry points to the C++ code (mostly). These actions correspond to the Do* variables in the *MeshInputState*

Transform: This handles anything related to transformations of the mesh or selections. This is only used for calculating which transformation to do for the selections. The code for applying the transformation to selections is in *Actions*

See also *Libigl.LibiglMesh* which handles the threading and calls the Pre/Post/Execute callbacks.

Public Functions

LibiglBehaviour (*LibiglMesh libiglMesh*)
Create a behaviour for the *Mesh* MonoBehaviour component. Every Mesh has one behaviour.

void **Update** ()
Called every frame, the normal Unity Update. Use this to update UI, input responsively. Do not call any expensive libigl methods here, use Execute instead

Be careful not to modify the shared state if there is a job running *Libigl.LibiglMesh.IsJobRunning*. Consider making a copy of certain data, using *PreExecute* or using atomics/Interlocked. Update is called just before *PreExecute*.

void **PreExecute** ()
Called just before a new thread is started in which *Execute* is called. Use this to update the input state, set flags and access any Unity API from the main thread.

Called on the main thread.

void **Execute** ()
Perform expensive computations here. This is called similarly to Update. Called on a worker thread from which any Unity API function, with a few exceptions such as `Debug.Log`, cannot be called.

There is one worker thread per *Libigl.LibiglMesh*. You should call `_libiglMesh.DataRowMajor.ApplyDirty(_state)` here to apply changes to the RowMajor *UMeshData* outside the main thread.

void **PostExecute** ()
Called after *Execute* to apply changes to the mesh.

Called on the main thread.

Use `Mesh.DataRowMajor.ApplyDirtyToMesh` to apply changes

void **Dispose** ()
This is the destructor. Ensure all C++ owned data is deleted. Calls Native.DisposeMesh

void **SetActiveSelection** (int *value*)
Changes the active selection and triggers *OnActiveSelectionChanged*.

void **SetActiveSelectionIncrement** (int *increment*)
Increments the active selection and safely loops.

Public Members

*MeshState** **State**
A **pointer** to the C++ state. This is allocated and deleted in C++ within Native.InitializeMesh and Native.DisposeMesh.

MeshInputState **Input**
The input state on the main thread. This is copied to the thread input state `State.Input` at the end of PreExecute and is then immediately consumed by *MeshInputState.Consume*.

readonly *LibiglMesh* **Mesh**
Reference to the *Libigl.LibiglMesh* used to apply changes to the *Libigl.LibiglMesh.DataRowMajor* and the Unity UnityEngine.Mesh

Events

event Action **OnActiveSelectionChanged** = delegate { }
Invoked when the active selection of the mesh has changed.

Private Functions

void **ActionTransformSelection** ()
Transforms the selections based on the *TransformDelta*s given in the *MeshInputState* It also decides which selections should be translated, storing this in _currentTranslateMaskL

void **FindBrushSelectionMask** (**ref** uint *maskId*, Vector3 *brushPos*)
Finds which selections are inside the brush and updates the maskId

void **ActionTransformSelectionGeneric** (**ref** *TransformDelta transformDelta*, uint *maskId*)
Does the actual translation, but is independent of the hands (L or R)

void **ActionSelect** ()
Selects what is inside the brushes of the hands.

void **ActionSelectGeneric** (bool *doSelect*, Vector3 *brushPos*, bool *alternateSelectMode*)
Does the actual selection, but is independent of the hands (L or R)

void **ActionHarmonic** ()
Runs the `igl::harmonic` Biharmonic Deformation

void **ActionArap** ()
Runs the `igl::arap` As-Rigid-As-Possible Deformation

void **ActionUi** ()
Applies various actions triggered from the UI or other input

void **UpdateInput** ()
Updates the *Input* every frame, from Update().

void **UpdateInputTransform** ()
Input for the transform tool

void **UpdateInputSelect** ()
Gathering input for the select tool

void **PreExecuteInput** ()
Updates the *Input* just before the worker thread is started. This copies the shared InputManager.State to the *Input*

void **UpdateTransform** ()
Updates the current transformation state from the input, regardless of what the worker thread is doing. Decides when a transformation is started/stopped (changes in the finite state machine FSM). Applies the transformation immediately if we are transforming the mesh (as this is done by Unity). Call this in Update() every frame.

void **ApplyTransformToMesh** ()
Gets and consumes the transformation, applying it to the *LibiglMesh* Transform.

void **ApplyTransformToSelection** ()
Save and consume the transformation, which will later be applied to the selection on the worker thread.

void **PreExecuteTransform** ()

void **ResetTransformStartPositions** ()

void **GetTransformDelta** (bool *consumeInput*, **ref** *TransformDelta transformDelta*, Space *space*, bool *withRotate*, bool *isTwoHanded*, bool *primaryHand*)
Find out the *TransformDelta* that should be done. Independent of what we are transforming, mesh or selection.

Parameters

- `consumeInput`: Should we reset the starting positions/rotations of the hands. This is unrelated to the *MeshInputState.Consume* function which is for the worker thread
- `transformDelta`: Where we should add our transformation to.

void **GetTransformOneHanded** (bool *isRight*, **ref** *TransformDelta transformDelta*, bool *withRotate = true*)
Finds out the transformation when using one hand

void **GetTransformTwoHanded** (**ref** *TransformDelta transformDelta*, bool *withRotate = true*)
Finds out the transformation when using both hands

Private Members

uint **_currentTranslateMaskL**
The selections which are currently being translated by the left hand. This is set by the worker thread.

uint **_currentTranslateMaskR**

MeshInputState **_executeInput**
The input state on the worker thread. When inside an Actions or anywhere on the worker thread you should exclusively access this input state.

readonly *UiMeshDetails* **_uiDetails**
The corresponding UI panel. It is initialized from here.

bool **_firstTransformHand**
The first hand to start a transformation. True = Right

bool **_isTwoHandedTransformation**
Are we using both hands in the transformation

bool **_isTwoHandedTransformationPrev**

bool **_doTransformL**

bool **_doTransformPrevL**

bool **_doTransformR**

bool **_doTransformPrevR**

Vector3 **_transformStartHandPosL**
Where the hand was when the *TransformDelta* was started. Or the hand position at the last time the transformation was consumed.

Vector3 **_transformStartHandPosR**

Quaternion **_transformStartHandRotL**

Quaternion **_transformStartHandRotR**

const float **GrabPressThreshold** = 0.1f
At which point do we consider the button as pressed

TransformDelta

struct Libigl.TransformDelta
Stores information about a single transformation.

Public Functions

void **Add** (*TransformDelta other*)
(experimental) Combines two transformations, does not consider the pivot.

Public Members

Vector3 **Translate**

Quaternion **Rotate**

float **Scale**

PivotMode **Mode**

Vector3 **Pivot**

Public Static Functions

TransformDelta **Identity** ()

3.4.5 MeshInputState.cs

struct Libigl.MeshInputState
Struct for storing the current input *for a mesh*. (This is a value type so assigning will copy). Anything that may change as we are executing should be in the InputState, as it is copied in PreExecute. Anything that is the same between all meshes may be put into the InputState. Anything related to what should be executed on the worker thread should be put here (e.g. DoSelect).

Variables that correspond to triggering Actions start with `Do` e.g. `DoSelect`

Public Functions

void **Consume** ()
Consumes and resets flags raised. Should be called in PreExecute after copying to the State.

Public Members

InputState **Shared**

InputState **SharedPrev**

bool **DoTransformL**

bool **DoTransformR**

bool **DoTransformLPrev**

bool **DoTransformRPrev**

bool **PrimaryTransformHand**

TransformDelta **TransformDeltaJoint**

TransformDelta **TransformDeltaL**

TransformDelta **TransformDeltaR**

int **ActiveSelectionId**

uint **VisibleSelectionMask**

bool **VisibleSelectionMaskChanged**

uint **SCountUi**
For UI, will be copied to the state in PreExecute. This is used when we create a selection in the UI on the main thread.

bool **DoSelectL**

bool **DoSelectLPrev**

bool **DoSelectR**

bool **DoSelectRPrev**

bool **AlternateSelectModeL**
Inverts the selection mode between *SelectionMode.Add* and *SelectionMode.Subtract*

bool **AlternateSelectModeR**

uint **DoClearSelection**
A Mask of the selections that should be cleared

Vector3 **BrushPosL**

Vector3 **BrushPosR**

float **BrushRadiusLocal**

bool **DoHarmonic**
Trigger execution once

bool **DoHarmonicRepeat**
Trigger execution every frame

bool **HarmonicShowDisplacement**

bool **DoArap**

bool **DoArapRepeat**

bool **ResetV**

Public Static Functions

MeshInputState **GetInstance** ()

Return

An instance with the default values

Private Functions

void **ConsumeTransform** ()

Consumes the state for the transformations

3.4.6 MeshState.cs

struct Libigl.MeshState

The **C++ state for a mesh** in column major. This is linked to Unity and the RowMajor version via *UMeshData*. It stores only pointers to the Eigen data so this can be shared between C++ and C#. The mesh data is allocated in C++ during the Native.InitializeMesh function using *UMeshDataNative*.

The *DirtyState* indicates what has been changed and needs to be applied to the Unity mesh.

As this struct is shared between a managed (C#) and native (C++) context, you must consider Marshalling when adding new variables.

`void*` usually represents an Eigen matrix, but can be anything.

Public Members

uint **DirtyState**

Tells us what has changed with the mesh using the *DirtyFlag* constants

uint **DirtySelections**

Tells us which selections have been modified, as a bitmask. Each bit represents one selection.

uint **DirtySelectionsResized**

Less stricter version than *DirtySelections*, where we only consider a selection dirty if the selected vertices size changes, see *SSizes*.

readonly void* **VPtr**

readonly void* **NPtr**

readonly void* **CPtr**

readonly void* **UVPtr**

readonly void* **FPtr**

readonly int **VSize**

readonly int **FSize**

readonly void* **SPtr**

uint **SSize**

Amount of selections enabled

readonly uint* **SSizes**

uint[32], vertices selected per selection

readonly uint **SSizesAll**

Total vertices selected

Private Members

readonly void* **Native**
Native only state

3.4.7 Native.cs

class Libigl.Native
Contains all C++ function declarations. **C# to C++**

Handles initialization of the DLL and works with the UnityNativeTool for easy reloading/recompilation.

Convention: Pass the *MeshState* as the first argument if the function modifies a mesh.

Public Functions

unsafe *MeshState** **InitializeMesh** (*UMeshDataNative data*, string *name*)

unsafe void **DisposeMesh** (*MeshState *data*)

unsafe void **ApplyDirty** (*MeshState *state*, *UMeshDataNative data*, uint *visibleSelectionMask*)

unsafe void **ReadOFF** (string *path*, bool *setCenter*, bool *normalizeScale*, float *scale*, **out** float *VPtr*, **out** int *VSize*, **out** float *NPtr*, **out** int *NSize*, **out** uint *FPtr*, **out** int *FSize*, bool *calculateNormalsIfEmpty*)

unsafe void **TranslateAllVertices** (*MeshState *state*, Vector3 *value*)

unsafe void **TranslateSelection** (*MeshState *state*, Vector3 *value*, uint *maskId*)

unsafe void **TransformSelection** (*MeshState *state*, Vector3 *translation*, float *scale*, Quaternion *rotation*, Vector3 *pivot*, uint *maskId*)

unsafe void **Harmonic** (*MeshState *state*, uint *boundaryMask*, bool *showDeformationField*)

unsafe void **Arap** (*MeshState *state*, uint *boundaryMask*)

unsafe void **ResetV** (*MeshState *state*)

unsafe void **SelectSphere** (*MeshState *state*, Vector3 *position*, float *radius*, int *selectionId*, uint *selectionMode*)

unsafe uint **GetSelectionMaskSphere** (*MeshState *state*, Vector3 *position*, float *radius*)

unsafe Vector3 **GetSelectionCenter** (*MeshState *state*, uint *maskId*)

unsafe void **ClearSelectionMask** (*MeshState *state*, uint *maskId*)

unsafe void **SetColorSingleByMask** (*MeshState *state*, uint *maskId*, int *colorId*)

unsafe void **SetColorByMask** (*MeshState *state*, uint *maskId*)

Public Static Functions

void **Initialize** ()
Initializes the native library and sets up callbacks/delegates for C++ -> C# calls. Note: this may not be called on the main thread. So Unity functions may not be available.

In the editor, this is triggered **each time** the dll has been loaded.

void **Destroy** ()
Clean up native part if required, called **just before** unloading of the dll.

Public Static Attributes

readonly VertexAttributeDescriptor[] **VertexBufferLayout**
Contains the vertex buffer layout (on the GPU) for editable meshes. There will be a copy of the mesh on the CPU which may not have the same layout. This was initially intended to be done for more a efficient applying of mesh data, but it allows for slightly more control over how meshes are stored on the GPU.

Private Functions

void **Initialize** (*NativeCallbacks.StringCallback debugCallback*, *NativeCallbacks.StringCallback debugWarningCallback*, *NativeCallbacks.StringCallback debugErrorCallback*)

Private Members

const string **DllName** = "__libigl-interface"
Name of the dll without the extension. Use this with the DllImportAttribute. This is set such that in the editor we can use the UnityNativeTool and in the build we use the library directly.

In Editor: `libigl-interface`

In Build and Actual Name: `__libigl-interface`

Private Static Functions

static Native ()
In a build, Initialize in the static ctor. This is called once just before the first function is called in this class.

Please read up on static constructors before modifying this.

3.4.8 NativeCallbacks.cs

class Libigl.NativeCallbacks
Contains all callbacks from the native context. **C++ to C#**

Callbacks should be annotated with the MonoPInvokeCallbackAttribute so that IL2CPP builds will compile properly. Each callback needs to have a corresponding delegate (/type).

Public Functions

delegate void **StringCallback** (string *message*)
Based on https://answers.unity.com/questions/30620/how-to-debug-c-dll-code.html The 'function pointer type' passed to C++

Parameters

- `message`: String or char* to be printed

Public Static Functions

void **DebugLog** (string *message*)
The function that we point to in C++

Parameters

- `message`: String or char* to be printed

void **DebugLogWarning** (string *message*)

void **DebugLogError** (string *message*)

3.5 C# Libigl.Editor

3.5.1 MeshImportPostprocessor.cs

class Libigl.Editor.MeshImportPostprocessor : **public** AssetPostprocessor
Used for post-processing meshes after importing in the Editor. Only for file formats that Unity recognizes. Simplified version of *OffMeshImporter*

Private Functions

void **OnPostprocessModel** (GameObject *g*)
Called whenever a model is finished importing by the Unity importer. Custom importers will not call this

Parameters

- `g`

3.5.2 OffMeshImporter.cs

class Libigl.Editor.OffMeshImporter : **public** ScriptedImporter
A custom importer for .off mesh files, which Unity does not recognize or know how to import by default. See tooltips.

Public Functions

override void **OnImportAsset** (AssetImportContext *ctx*)
Called whenever a .off file is imported by Unity Trigger this manually by right-click > Reimport in the project browser

Parameters

- `ctx`: Used to store imported output objects

Public Members

bool **centerToMean** = true

bool **normalizeScale** = true

float **scale** = 1f

bool **optimizeForRendering** = true

Material **material**

Private Functions

Material **GetDefaultMaterial** ()

Private Members

const string **DefaultMaterialName** = "EditableMesh"

const string **DefaultMaterialNameFallbackShader** = "Universal Render Pipeline/Lit"

Private Static Attributes

Material **_defaultMaterial**

3.6 C# XrInput

3.6.1 InputManager.cs

The *InputManager* will detect and create the VR controllers. It handles getting of the current input state, i.e. which buttons are pressed, and saves this into the *InputState*. The *InputManager* and the *InputState* will also handle shared functionality, e.g. which tool is active, what the brush size is.

class XrInput.InputManager : **public** MonoBehaviour
This script handles the controller input and is based on the Unity XR Interaction Toolkit. An important part is that this is where the *InputState* can be accessed and is updated.

Public Functions

void **SetActiveTool** (*ToolType value*)
Sets the active tool and updates the UI

void **RepaintInputHints** (bool *left* = *true*, bool *right* = *true*)
Safely repaint the input hints on the controllers, specify which hands should be repainted.

Public Members

Transform **xrRig**
Get the XR Rig Transform, to determine world positions of controllers.

InputDeviceCharacteristics **handCharL** = InputDeviceCharacteristics.Controller | InputDeviceCharacteristics.Left

InputDevice **HandL**

UiInputHints **HandHintsL**

XrBrush **BrushL**

InputDeviceCharacteristics **handCharR** = InputDeviceCharacteristics.Controller | InputDeviceCharacteristics.Right

InputDevice **HandR**

UiInputHints **HandHintsR**

XrBrush **BrushR**

Events

event Action **OnActiveToolChanged** = delegate { }
Called just after the active tool has changed

Public Static Attributes

InputManager **get**
The singleton instance.

InputState **State**
The current input state shared between all meshes.

InputState **StatePrev**
Input at the last frame (main thread).

Private Functions

void **Awake** ()

void **Start** ()

void **Update** ()

bool **InitializeController** (bool *isRight*, InputDeviceCharacteristics *characteristics*, **out** InputDevice *inputDevice*, GameObject *handPrefab*, XRController *modelParent*, **out** Animator *handAnimator*, **out** *UiInputHints* *inputHints*, **out** *XrBrush* *brush*)
Gets the XR InputDevice and sets the correct model to display. This is where a controller is detected and initialized.

Return True if successful

void **UpdateHandAnimators** ()

void **UpdateRayInteractors** ()
Enables and disables certain rays and UI interaction for performance. Raycasting the UI is one of the most expensive operations currently. See profile EventSystem

void **UpdateSharedState** ()
Updates the *InputState State*. Implementation note: The hands may not be initialized/detected yet.

Private Members

bool **useHands** = false

GameObject **handPrefabL** = default

GameObject **handPrefabR** = default

List<GameObject> **controllerPrefabs** = default

XRController **handRigL** = default

XRRayInteractor **handInteractorL** = default

readonly List<XRBaseInteractable> **_rayHoverTargetsL** = new List<XRBaseInteractable>()

XRController **handRigR** = default

XRRayInteractor **handInteractorR** = default

readonly List<XRBaseInteractable> **_rayHoverTargetsR** = new List<XRBaseInteractable>()

Animator **_handAnimatorL**

Animator **_handAnimatorR**

XRRayInteractor **teleportRayL** = default
The ray interactor for teleporting

bool **_prevAxisClickPressedL**

bool **_prevAxisClickPressedR**

Private Static Attributes

readonly int **TriggerAnimId** = Animator.StringToHash("Trigger")

readonly int **GripAnimId** = Animator.StringToHash("Grip")

3.6.2 InputState.cs

struct XrInput.InputState
This is the 'shared' input state. It is also where you can access the filtered controller input. Stores input that is shared between meshes as well as the raw input that has not been mapped to actions. Raw input has been filtered to prevent conflicts with UI and grabbables.

Public Functions

void **TogglePivotMode** ()
Changes the pivot mode depending on the *ActiveTool*

Public Members

ToolType **ActiveTool**
Which tool are we currently using.

float **GripL**

float **TriggerL**

float **GripR**

float **TriggerR**

Vector3 **HandPosL**

Vector3 **HandPosR**

Quaternion **HandRotL**

Quaternion **HandRotR**

bool **PrimaryBtnL**

bool **SecondaryBtnL**

Vector2 **PrimaryAxisL**

bool **PrimaryBtnR**

bool **SecondaryBtnR**

Vector2 **PrimaryAxisR**

bool **IsTeleporting**
Are we pressing the teleport button? Used to disable other input.

float **BrushRadius**

PivotMode **ActivePivotModeTransform**

PivotMode **ActivePivotModeSelect**

bool **TransformWithRotate**
Is rotation enabled for transformations.

SelectionMode **ActiveSelectionMode**

bool **NewSelectionOnDraw**
Draw into a new selection with each stroke

bool **DiscardSelectionOnDraw**
Clear the selection when starting a stroke

bool **BoundsVisible**
Should we show the bounding boxes of the editable meshes

Properties

PivotMode **ActivePivotMode** { }
The pivot mode for the *ActiveTool*

ToolTransformMode **ToolTransformMode** { }
The sub-state of the Transform tool

ToolSelectMode **ToolSelectMode** { }
The sub-state of the Select tool

Public Static Functions

InputState **GetInstance** ()
An instance with the defaults set

Return

Private Members

ToolTransformMode **_toolTransformMode**

ToolSelectMode **_toolSelectMode**

Enums

enum XrInput.ToolType
Which tool is being used, *InputState.ActiveTool*.

Values:

Transform

Select

enum XrInput.ToolSelectMode
The sub-state of the Select tool.

Values:

Idle

Selecting

TransformingL

TransformingR

TransformingLr

enum XrInput.ToolTransformMode
The sub-state of the Transform tool.

Values:

Idle

TransformingL

TransformingR

TransformingLr

enum XrInput.SelectionMode
How to modify the selection.

Values:

Add

Subtract

Invert

enum XrInput.PivotMode
How to rotate the mesh/selection.

Values:

Mesh

Hand

Selection

3.6.3 XrBrush.cs

class XrInput.XrBrush : **public** MonoBehaviour
Functionality related to the sphere 'bubble' brush. Currently handles resizing the brush, getting the center and finding overlapping bounding boxes via trigger colliders.

Public Functions

void **SetRadius**(float *value*)

void **Initialize**(bool *isRight*)

void **OnActiveToolChanged**()

bool **SetActiveMesh**()
Will set the active mesh as the first hovered, if we are not hovering over the active mesh. Hovering is detected and visualized by the bounding boxes.

Return True if the active mesh has been set

Public Members

Transform **center**

const float **ResizeSpeed** = 0.5f

const float **ResizeDeadZone** = 0.1f

bool **InsideActiveMeshBounds**

Public Static Attributes

Vector2 **RadiusRange** = new Vector2(0.025f, 1f)

Private Functions

void **OnDestroy** ()

void **OnTriggerEnter** (Collider *other*)
Called when the brush bubble enters a trigger collider. Standard Unity callback. We use this to set the hovering status of the individual meshes.

void **MeshLeftTrigger** (*LibiglMesh libiglMesh*)
Called when the mesh leaves the trigger and updates the hovering status of a mesh. Implementation note: split into separate function so when deactivating we leave all triggers.

void **OnTriggerExit** (Collider *other*)

void **RepaintBoundingBoxes** ()
Repaint bounding boxes based on the hovering status. Bounds are hidden if we are hovering over the active mesh. The first hovered mesh is set as the primary one.

void **OnActiveMeshChanged** ()
Called by the event MeshManager.OnActiveMeshChanged

void **OnDisable** ()

Private Members

SphereCollider **_brushCollider**

bool **_isRight**

readonly List<*LibiglMesh*> **_currentLibiglMeshes** = new List<LibiglMesh>()

3.7 C# UI

3.7.1 UiManager.cs

class UI.UiManager : **public** MonoBehaviour
Handles easy creation of operations to be done on a mesh and the user interaction (2D UI, speech, gestures) that comes with it.

Public Functions

UiMeshDetails **CreateDetailsPanel** ()
Creates a new Details panel and initializes it

Return The Vertical Scroll List parent to which items can be added as a child

Public Members

GameObject **meshDetailsCanvasPrefab**

GameObject **headerPrefab**

GameObject **groupPrefab**

GameObject **textPrefab**

GameObject **buttonPrefab**

GameObject **togglePrefab**

GameObject **toggleActionPrefab**

GameObject **iconActionPrefab**

GameObject **selectionPrefab**

GameObject **selectionModePrefab**

GameObject **pivotModePrefab**

Transform **panelSpawnPoint**

Transform **genericUiListParent**

Material **uvGridMaterial**

MeshRenderer[] **toggleUvGridRenderers**

LayerMask **UiLayerMask**

const int **UiLayer** = 5

UiPivotMode **CurrentPivotMode**

Public Static Attributes

UiManager **get**

Private Functions

void **Awake** ()

void **Start** ()

void **InitializeStaticUi** ()
Generates the UI unrelated to a mesh or to manipulate the *active mesh* MeshManager.ActiveMesh

void **CreateActionSpeechUi** (string *uiText*, UnityAction *onClick*, *UiCollapsible* *collapsible* = *null*, string[] *speechKeywords* = *null*)
Generates UI, gesture and speed entry points based on an action

Parameters

- `onClick`: Code to execute when an entry point is triggered
- `collapsible`: The group to add this item under

void **OnDestroy** ()

Private Members

bool **_isShowingUvGrid**

Material[] **_uvGridInitialMaterials**

UiCollapsible **_toolGroup**

UiCollapsible **_meshGroup**

UiCollapsible **_debugGroup**

3.7.2 UiMeshDetails.cs

class UI.UiMeshDetails : **public** MonoBehaviour

Contains all functionality related to the mesh UI panel. There is one of these per LibiglMesh. Contains most UI generation.

Public Functions

void **Initialize** (*LibiglBehaviour behaviour*)

Main function where UI is generated.

Parameters

- `behaviour`: The behaviour we are generating the UI panel for.

void **OnDestroy** ()

void **UpdatePreExecute** ()

Called in PreExecute just before the input is consumed

void **UpdatePostExecute** ()

Update the UI Details panel after executing

bool **AddSelection** () **Return**

true if selection was successfully added, max 32 selections

Public Members

UiProgressIcon **progressIcon**

Button **activeBtn**

Button **deleteBtn**

Private Functions

void **RepaintActiveMesh** ()

Repaint based on which mesh is active. Should be called when MeshManager.OnActiveMeshChanged

Also called when the object is created as this happens just after a mesh was set

void **RepaintActiveSelection** ()

Repaint the active selection. Should be called when LibiglBehaviour.OnActiveSelectionChanged.

void **UpdateVertexCountText** ()

Private Members

LibiglBehaviour **_behaviour**
Access to the behaviour that this UI panel belongs to

Transform **_listParent**
The content of the scroll list view, add new components as a child of this transform.

Image **activeImage** = null

Sprite **editSprite** = null

Sprite **activeSprite** = null

Image **background** = null

Color **activeBackgroundColor** = Color.white

Color **_defaultBackgroundColor**

TMP_Text **_vertexCount**

UiCollapsible **_selectionGroup**

readonly List<*UiSelection*> **_selections** = new List<UiSelection>()

int **_activeSelectionId** = -1
The active selection displayed in the UI

UiCollapsible **_debugGroup**

UiToggleAction **_harmonicToggle**

UiToggleAction **_arapToggle**

3.8 C# UI.Components

3.8.1 UiCollapsible.cs

class UI.Components.UiCollapsible : **public** MonoBehaviour
UI Component header that hides items when clicked/toggled.

Public Functions

void **AddItem** (GameObject *item*, bool *setAsLastSibling = true*)
Add an item to the group, visibility is immediately set

Parameters

- `setAsLastSibling`: Add the item to the end of the group. Set this to false to have an item collapse/hide with this group but leave it in the place/sibling that it is.

void **ToggleVisibility** ()
Called from UI to change the visibility of the group's items

void **SetVisibility** (bool *value*)
Change the visibility of the group's items.

void **Remove** (GameObject *item*)
Remove an item from the collapsible. Will not delete the item.

void **RemoveLast** ()
Remove the last item from the group.

Public Members

TMP_Text **title**

RectTransform **checkmarkIcon**

Private Functions

void **OnEnable** ()

Private Members

readonly List<GameObject> **_items** = new List<GameObject>()

bool **_visible** = true

int **_lastSiblingIndex**

3.8.2 UiIconAction.cs

class UI.Components.UiIconAction : **public** MonoBehaviour
UI Component with two buttons, one icon sized.

Public Members

Button **iconBtn**

Button **actionBtn**

3.8.3 UiToggleAction.cs

class UI.Components.UiToggleAction : **public** MonoBehaviour
A toggle and button UI element. The toggle and button are independent by default

Public Members

Toggle **toggle**

Button **button**

TMP_Text **text**

3.8.4 UiSelection.cs

class UI.Components.UiSelection : **public** MonoBehaviour
UI for one selection of a mesh (one row)

Public Functions

void **Initialize** (int *selectionId*, *UiCollapsible* *uiCollapsible*, *LibiglBehaviour* *behaviour*, IList<*UiSelection*> *selections*)

unsafe void **ToggleVisible** ()

unsafe void **Clear** (bool *forceDelete* = *false*)
Clears the selection and deletes it if it is already empty and the last one

Parameters

- `forceDelete`: Delete this even if the selection is not empty. Still needs to be the last selection

unsafe void **UpdateText** ()

void **ToggleEditSprite** (bool *isActive*)

Public Members

TMP_Text **text**

Button **visibleBtn**

Button **editBtn**

Button **clearBtn**

Private Functions

void **SetAsActive** ()

void **ToggleVisibleSprite** (bool *isVisible*)

Private Members

Image **visibleImage** = null

Sprite **visibleSprite** = null

Sprite **visibleOffSprite** = null

Image **editImage** = null

Sprite **editSprite** = null

Sprite **activeSprite** = null

int **_selectionId**

UiCollapsible **_uiCollapsible**

LibiglBehaviour **_behaviour**

IList<*UiSelection*> **_selections**

3.8.5 UiSelectionMode.cs

class UI.Components.UiSelectionMode : **public** MonoBehaviour

Handles the selection mode UI, onclick behaviour. There are several modes from *SelectionMode* as well as the *newSelectionOnDrawBtn* where a new selection is added on each stroke.

Public Functions

void **Initialize** ()

void **Repaint** ()

Public Members

Image[] **icons**

Color **activeColor**

Button **newSelectionOnDrawBtn**

Button **discardSelectionOnDrawBtn**

Private Functions

void **SetMode** (*SelectionMode* *mode*)

void **ToggleNewSelectionOnDraw** ()

void **RepaintNewSelectionOnDrawBtn** ()

void **ToggleDiscardSelectionOnDraw** ()

void **RepaintDiscardSelectionOnDrawBtn** ()

Private Members

SelectionMode **_mode**

3.8.6 UiPivotMode.cs

class UI.Components.UiPivotMode : **public** MonoBehaviour

Similar to *UiSelectionMode*, effectively an enum field.

Public Functions

void **Initialize** ()

void **Repaint** ()

Public Members

Image[] **icons**

Color **activeColor**

Private Functions

void **OnDestroy** ()

void **SetMode** (*PivotMode* *mode*)

Private Members

PivotMode **_mode**

3.8.7 UiProgressIcon.cs

class UI.Components.UiProgressIcon : **public** MonoBehaviour

Handles showing the progress icon and animating it based on the PreExecute and PostExecute. This indicates the state of the libigl thread.

Public Functions

void **PreExecute** ()

void **PostExecute** ()

Public Members

Gradient **progressGradient**

float **progressDelayTime** = 1f

float **timeoutTime** = 9f

Sprite **progressErrorSprite**

Private Functions

void **Start** ()

IEnumerator **ShowProgressAfterTime** ()

Coroutine that animates progress icon over time. Can be stopped

Private Members

Image **_backgroundImage**

Image **_iconImage**

Sprite **_defaultSprite**

Coroutine **_progressCoroutine**

3.9 C# UI.Hints

3.9.1 UiInputHints.cs

class UI.Hints.UiInputHints : **public** MonoBehaviour

Defines the behaviour of the input hints of one hand. Important functions are: AddTooltip(GameObject, string), *SetData* and *Repaint*.

In short, tooltips can be added so when we hover over a UI element it displays some text. Repaint is called with the collection to set out the default hints for a particular state. This can then be overriden by scripting, e.g. see RepaintTriggerColor where we set the trigger hint color ot the active selection color.

Public Functions

void **Initialize** ()

void **SetData** (*UiInputHintsData* *data*)

Set the data that should be displayed. Note: Consider the overrides and the ordering when calling this multiple times.

void **Repaint** ()

Refreshes the Input hints based on the InputState.ActiveTool and the sub state like ToolTransformMode

Public Members

UiInputLabel **title**

UiInputLabel **help**

UiInputLabel **trigger**

UiInputLabel **grip**

UiInputLabel **primaryBtn**

UiInputLabel **secondaryBtn**

UiInputLabel **primaryAxisX**

UiInputLabel **primaryAxisY**

Public Static Functions

void **AddTooltip** (GameObject *uiElement*, string *msg*)
Display the message when hovering over this UI element. Note: Must be attached to the gameObject with the button/toggle component, otherwise clicking may be blocked

Parameters

- `uiElement`: GameObject with the Button/Toggle UI component
- `msg`: Message to display when hovering

void **AddTooltip** (GameObject *uiElement*, Func<string> *msg*)
An overload where the message can be a lambda.

Private Functions

void **SetTooltip** (string *data*, bool *overrideExisting* = *false*)
Apply text to the help/tooltip. Used by the UI.

void **ClearTooltip** ()

void **RepaintTriggerColor** ()
Sets the trigger hint background color based on the active selection for the active mesh. Implementation note: Example of input hints driven by scripting. We ensure with the events On*Changed that this function is called only when the ActiveMesh changes or the selection of the ActiveMesh.

void **OnActiveMeshChanged** ()

void **OnDestroy** ()

Private Members

UiInputHintsDataCollection **collection** = null

UiInputHintsData **_currentData**

LibiglBehaviour **_activeBehaviour**

const float **DefaultTooltipTimeout** = 60f

3.9.2 UiInputLabel.cs

class UI.Hints.UiInputLabel : **public** MonoBehaviour
A label for a physical input button/axis to give a hint to what it does.

Public Functions

void **Initialize** ()

void **SetData** (*UiInputLabelData* *data*, bool *overwriteData* = *true*)

void **SetText** (string *data*)

void **SetColor** (Color *value*)

void **SetIcon** (Sprite *sprite*)

void **ResetToData** ()
Resets the label to the last written data, set when *SetData* overwriteData = true.

Public Members

Image **background**

Image **icon**

TMP_Text **text**

Private Members

UiInputLabelData **_data**

Color **_defaultColor** = Color.white

3.9.3 Data

UiInputHintsDataCollection.cs

class UI.Hints.UiInputHintsDataCollection : **public** ScriptableObject
Stores the hints for all possible states of the ActiveTool and sub-states. These are stored as a hierarchy. *UiInputHints.Repaint* defines how this data is applied. There will be one instance for the left and one for the right hand.

Public Members

UiInputHintsData **defaultHints**

UiInputHintsData **transform**

UiInputHintsData **transformIdle**

UiInputHintsData **transformTransformingL**

UiInputHintsData **transformTransformingR**

UiInputHintsData **transformTransformingLr**

UiInputHintsData **select**

UiInputHintsData **selectIdle**

UiInputHintsData **selectSelecting**

UiInputHintsData **selectTransformL**

UiInputHintsData **selectTransformR**

UiInputHintsData **selectTransformLr**

UiInputHintsData.cs

class UI.Hints.UiInputHintsData : **public** ScriptableObject

Data for one hand and one state of the ActiveTool. Used by the *UiInputHints*. We have one *UiInputLabelData* per button/axis.

Public Members

UiInputLabelData **title**

UiInputLabelData **help**

UiInputLabelData **trigger**

UiInputLabelData **grip**

UiInputLabelData **primaryBtn**

UiInputLabelData **secondaryBtn**

UiInputLabelData **primaryAxisX**

UiInputLabelData **primaryAxisY**

UiInputLabelData.cs

struct UI.Hints.UiInputLabelData

Defines the content for a *UiInputLabel*.

Public Members

bool **isOverride**

bool **isActive**

Sprite **icon**

string **text**

CHAPTER

FOUR

C++ API REFERENCE

Note: This is aimed at people wanting to view or edit the code.

The C++ code is found in `Interface/source` and is compiled to a .dll (shared library) with CMake. It is *automatically* copied into the `Assets/Plugins` folder so we can use it with Unity. Note that the C++ code is where all the geometry and performance intensive code resides. There is no `main()` function instead `Native.h` includes all the exported functions that can be called from C#.

4.1 C#/C++ Interface

This page is quite techical, in most scenarios you can just look at what already exists!

Note: Native = C++, Managed = C# (think of memory management)

Quick Facts:

- Code written in C++ has to be compiled to a shared library (`.dll` on windows) and placed in the `Assets/Plugins` folder. This is done by CMake when building.
- The native functions must be redeclared in C# as `extern` with the `DllImport` attribute and can then be called normally. All these declarations are in `Libigl/Native.cs`.
- You can use pointers with the `unsafe` keyword.
- Managed data structures can have a different layout than native ones. *'Marshalling'* converts between the two automatically, but can involve expensive copies.
- Classes or structs can be shared between but must be declared in both languages.
- Beware of the garbage collector. Pin *managed* data using `fixed`, or `GcAlloc` with the pinned option, to prevent the garbage collector from moving/deleting whilst the C++ is executing. This is only for classes (not structs/value types).

4.1.1 C++ Building The Library

CMake

- When building the `.dll` is placed in the `Assets/Plugins` folder automatically
- Note that the output directory can be set in the CMake cache with the `UNITY_*` variables
- Set `CMAKE_VERBOSE` for precise message if something goes wrong in CMake
- Currently only Visual Studio Solutions `.sln` have been tested

Rebuilding and Unloading Native Libraries

Unity presents the complication that it *never unloads a dll once loaded*, this prevents write access and rebuilding will fail. A dll is loaded once a function from it is called for the first time.

UnityNativeTool by mcpiroman present a good workaround for this by *'mocking'* native functions and un/loading the dll manually. This is only done in the editor so builds will be unaffected. This method allows us to use the normal P/Invoke attribute `[DllImport("mylib")]` above external native function declarations in C#. So there are no changes to our code(!) This works in edit and play mode and details can be seen in the instance of the `DllManipulatorScript`. However, this means the `Main` scene must be loaded in order to be able to use the dll.

We also get a callback whenever a library is loaded and unloaded (pre/post) allowing us to initialize and clean up the native library nicely. This relied on the mysteriously named `stubLluiPlugin`.

What you need to know:

- The library is **loaded whenever a function is called**, `Alt` + `Shift` + `D` is pressed
- It is **unloaded** when play mode ends, the `DllManipulatorScript` is disabled (`OnDisable`) or when manually unloading via the component inspector or the shortcut `Alt` + `D`
- **When you want to rebuild your library, stop play mode or unload it first in Unity via the shortcut**
- You can use `[DllImport]` as usual
- There are certain limitations to marshalling and similar
- We can get callbacks by using the attributes in `UnityNativeTool/scripts/Attributes.cs`, e.g. when the dll is un/loaded
- Use `[MockNativeDeclarations]` on a class or native function to enable this unloading
- The shortcuts un/load *all* mocked libraries, if there are several

4.1.2 Debugging

C++ or C#: Open the solution in Visual Studio. `Debug > Attach To Process...` and select running `Unity.exe`. Place breakpoints as usual. Ensure that you build before running so that the source code matches the executing code. For C# debuggin in VS also search online...

Simultaneous C#/C++: VS cannot debug both at the same time, two instances do not work. So current solution is to use Jetbrains Rider to debug the C# side and VS (or CLion) for C++.

Tip: The editor/application will crash if there is a segfault in C++, use Visual Studio to debug. Failed assertions will cause a pop up. When this happens you can attach the debugger and then press *Retry* to inspect properly.

4.1.3 Calling Native functions

Do's and Don'ts

Do:

- Check that function **declarations match exactly** by copy-pasting for example
  - Be careful with references
- Label parameters with `in` and `out` to improve performance
- Use `unsafe` to pass pointers along with `UnsafeUtility`
- Use `NativeArray<T>` when possible along with `NativeArrayUnsafeUtility`
- Keep C#/C++ interface calls to a minimum for a simple interface

Don't:

- Pass large non-blittable types, e.g. matrices, use pointers instead
- Have unhandled exceptions. Exceptions should be handled fully in C++ or fully in C#.
- Call a C# delegate/function pointer from C++ without checking if it is valid/null.

Lots of problems can arise if this is not the case.

4.1.4 Global Variables/Persistent Memory in C++

Anything related to a specific mesh **must** be part of the *`MeshState`*. However, global variables can be used to store a state between function calls from C#. Declare these as extern in a header and define them once in a C++. They can be set in the *`Initialize()`*.

Memory allocated with `new` in C++ will persist as usual until it is deleted with `delete`. Notably, the *`MeshState`* is allocated in C++ when *`InitializeMesh()`* is called. C# can access (read/write but not delete) C++ owned memory.

Note: When the dll is unloaded all memory it allocated must be deleted. This can be done in `UnityPluginUnload()` or triggered by a C# destructor, see *LibiglMesh.cs* and *LibiglBehaviour.cs*. Notably, when hot reloading this is also the case.

(advanced) When *hot reloading* (pausing play mode, un/loading the dll) global variables are deleted. Pointers to data allocated with `new` are still valid, but the memory cannot be used as the owner dll has been destroyed (effectively a segmentation fault). You cannot simply keep the same data. As such, all persistent data must be serialized and then deserialized if you want the state to survive a hot reload. This has not yet been implemented but could be done with `igl::serialize`.

4.1.5 Marshalling

Marshalling allows us to pass managed data to a native context. Ensure that you use 'blittable types' as much as possible as these do not involve a copy. Generally:

- blittable types: `int`, `float`, numbers, structs consisting of only these, 1-D arrays of these
- non-blittable types: `string`, `bool`, **n-D arrays**

To pass a struct add the `[StructLayout(Sequential)]` attribute to it in C# and redeclare it in C++ in `InterfaceTypes.h` with the *same variable ordering*. `in` and `out` parameter attributes allow the Marshalling to optimize more. It should match C++ references. For strings use `CharSet = Ansi` in `DllImport`

4.1.6 Calling Managed functions from C++

Note: In certain rare scenarios this may be desirable. Think first if this can be avoided. It is possible via function pointer callbacks.

In C#, a delegate (~function pointer type) must be declared and the function to be called. The function must be annotated with the `[MonoPInvokeCallback(typeof(MyDelegate))]` attribute. It must be a static method. See `Scripts/Libigl/NativeCallbacks.cs` and add your callback there.

In C++, declare a function pointer typedef like the delegate, see *`StringCallback`*. The function pointer must use the `UNITY_INTERFACE_API` to ensure the `__stdcall` C# calling convention is used. Then you declare an instance of the function pointer as extern, see *`DebugLog`*. Finally we must set the pointer when calling *`Initialize()`* and reset to `nullptr` in `UnityPluginUnload()`. The extern variables need to be properly declared in `Native.cpp`. See `source/InterfaceTypes.h` and add your code there.

Warning: Function pointers/callbacks may be invalid or null. Check before invoking them or a crash will occur.

Further reading: Debug.Log example

4.1.7 Further Reading

A good simple introduction to P/Invoke

Unrelated and not what you want:

- C++/CLI (Microsoft) which is not the same as C++
- COM (Microsoft Component Object Model)
- CLR (Microsoft Common Language Runtime)

Related and what you are using/looking for:

- **P/Invoke** used by the `DllImportAttribute` (stands for Platform Invoke)

Links:

- `fixed` keyword
- Mono Interop with Native Libraries, P/Invoke
- Simple OpenCV Example
- x86 Calling Conventions for `__stdcall`

- Unity Macros only the first 20 lines

4.2 C++ API Reference

4.2.1 Native.h

This is the central file with all exported functions that are callable from C#. These are the 'entry points'. As this is a library we do not have a `main()`, however we have the *`Initialize()`* function as a replacement. Unity also triggers the functions `UnityPluginLoad()` and `UnityPluginUnload()` at the start and end. These are called first and last, notably *`Initialize()`* is called after `UnityPluginLoad()`.

The two important functions for the lifecycle of a mesh are *`InitializeMesh()`* and *`DisposeMesh()`*. These are called whenever a *`LibiglMesh`* is instaniated or destroyed in Unity. This is where the C++ owned memory is allocated and deleted.

To make these functions callable from C# we must put the declarations inside an `extern "C"` scope, as well as prepending the `UNITY_INTERFACE_EXPORT` to the declaration. This is because C# and C++ use different default calling conventions, see x86 Calling Conventions specifically `__stdcall`. *This also is different for every platform, but luckily the IUnityInterface header handles this for us if we use this macro.*

Note that the implementations of the defined functions are split across several cpp files, as indicated in the code. This is done so we have one central place where we have all the exported functions that are callable from C#.

Functions

void **`Initialize`**(*StringCallback debugCallback*, *StringCallback debugWarningCallback*, *StringCallback debugErrorCallback*)

Called just before the first function is called from this library. Use this to pass initial data from C#

See C# Native.Initialize()

Parameters

- `debugCallback`: A C# delegate (function pointer) to print to the Unity Debug.Log

MeshState ***`InitializeMesh`**(**`const`** *UMeshDataNative data*, **`const`** char **name*)

Called when a new mesh is loaded. Initialize global variables, do pre-calculations for a mesh

Return A pointer to the C++ state for this mesh

Parameters

- `data`: The Unity MeshData
- `name`: Name of the mesh

void **`DisposeMesh`**(*MeshState* **state*)

Disposes all C++ state tied to a mesh properly

void **`UnityPluginLoad`**(IUnityInterfaces **unityInterfaces*)

Called when the plugin is loaded, this can be after/before *Initialize()*

Declared in IUnityInterface.h

Parameters

- `unityInterfaces`: Unity class for accessing minimal Unity functionality exposed to C++ Plugins

void **`UnityPluginUnload`**()

Called when the plugin is unloaded, clean up here Declared in IUnityInterface.h

void **ApplyDirty**(*MeshState* *state, **const** *UMeshDataNative* data, **const** unsigned int *visibleSelectionMask*)

Propagates changes from the libigl mesh (Eigen matrices) to the Unity NativeArrays so we can apply it to the mesh in C#. This functions also recalculated selection sizes, colors based on the selection. This should be called once after apply all wanted changes to the libigl mesh. This function itself does not modify the Unity mesh, see UMeshData.cs

Parameters

- `data`: Pointers to the Unity Mesh Data, where to apply changes to.
- `visibleSelectionMask`: Selections which are currently visible, will ignore changes to invisible selections.

void **ReadOFF**(**const** char *path, **const** bool *setCenter*, **const** bool *normalizeScale*, **const** float *scale*, void *&*VPtr*, int &*VSize*, void *&*NPtr*, int &*NSize*, void *&*FPtr*, int &*FSize*, bool *calculateNormalsIfEmpty* = false)

Reads an .off file into **row major** Eigen matrices, these can then be mapped by a NativeArray in C#. Matrices are allocated with `new` and must be deleted manually (e.g. by `NativeArray<T>.Dispose()` or converting with `Allocator.Temp`).

Parameters

- `path`: The asset path, absolute or relative to the project root e.g. AssetImportContext.assetPath or "Assets/model.off"
- `setCenter`: Set the center as the mean vertex, see ApplyScale
- `normalizeScale`: Whether to normalize the y-scale to 1
- `scale`: Scale the mesh by this factor *after normalization*
- `[out] VPtr`: Pointer to the first element of the Vertex matrix
- `[out] VSize`: Number of vertices, rows of V
- `[out] NPtr`: Pointer to the first element of the Normals matrix
- `[out] NSize`: Number of normals, usually equal to VSize
- `[out] FPtr`: Pointer to the first element of the Face/Indices matrix, one row is a triangle
- `[out] FSize`: Number of faces, rows of F
- `calculateNormalsIfEmpty`: Calculate per vertex normals, if no normals are present in the .off file

void **TranslateAllVertices**(*MeshState* *state, *Vector3* value)

Debug function to simply translate all vertices by the value.

void **TranslateSelection**(*MeshState* *state, *Vector3* value, unsigned int *maskId* = -1)

Translate vertices in specific selections.

Parameters

- `value`: displacement in local space
- `maskId`: Which selections to transform as a bitmask

void **TransformSelection**(*MeshState* *state, *Vector3* translation, float *scale*, *Quaternion* rotation, *Vector3* pivot, unsigned int *maskId* = -1)

Transform the selected vertices in place. Translate + Rotate + Scale in this order. Rotation is currently only around the origin of the mesh

Parameters

- `maskId`: Which selections to transform

void **`Harmonic`**(*MeshState* **state*, unsigned int *boundaryMask* = -1, bool *showDeformationField* = true)
Run the igl::harmonic biharmonic deformation on the mesh with provided fixed boundary conditions.

Remark From libigl Tutorial 401, https://libigl.github.io/tutorial/#biharmonic-deformation

Parameters

- `boundaryMask`: Which selections to use as the boundary
- `showDeformationField`: Whether to show the deformation field, see libigl tutorial

void **`Arap`**(*MeshState* **state*, unsigned int *boundaryMask* = -1)
Run the igl::arap As-Rigid-As-Possible deformation on the mesh with the provided fixed boundary conditions.

Remark From libigl Tutorial 405, https://libigl.github.io/tutorial/#as-rigid-as-possible

Parameters

- `boundaryMask`: Which selections to use as the boundary

void **`ResetV`**(*MeshState* **state*)
Reset the vertices to their initial position V0 (set when loading the mesh).

void **`SelectSphere`**(*MeshState* **state*, *Vector3* *position*, float *radius*, int *selectionId* = 0, unsigned int *selectionMode* = *SelectionMode*::*Add*)
Modify the selection inside a sphere.

Parameters

- `position`: Center of the sphere in local space
- `radius`: Radius of the sphere in local space
- `selectionId`: Which selection to modify
- `selectionMode`: How to change the selection, use *SelectionMode* constants.

unsigned int **`GetSelectionMaskSphere`**(*MeshState* **state*, *Vector3* *position*, float *radius*)

Return A mask of all selections partially inside the sphere (based on if a vertex is inside).

Parameters

- `position`: Center of the sphere in local space
- `radius`: Radius of the sphere in local space

Vector3 **`GetSelectionCenter`**(*MeshState* **state*, unsigned int *maskId*)

Return The mean vertex of the specified selections

void **`ClearSelectionMask`**(*MeshState* **state*, unsigned int *maskId* = -1)
Resets a particular selections, can clear multiple selections at once.

Parameters

- `maskId`: Which selections to clear as a bitmask

void **`SetColorSingleByMask`**(*MeshState* **state*, unsigned int *maskId* = -1, int *colorId* = 0)
Naive SetColorByMask. Show all selections in the mask in the same color.

Parameters

- `maskId`: Which selections to show in the color
- `colorId`: Which color to use, see *Color::GetColorById*

void **SetColorByMask**(*MeshState* *state, unsigned int *maskId* = -1)
Set the vertex colors based on a bitmask of visible selections.

Parameters

- `maskId`: Which selections to show

Variables

IUnityInterfaces ***s_IUnityInterfaces**
Access to the Unity interfaces, currently not used.

4.2.2 InterfaceTypes.h

This file includes all the types that are shared between C# and C++. These are declared once in both languages and if one if modified the other must also be updated. In C# this corresponds to the classes with the attribute `[StructLayout(LayoutKind.Sequential)]`. The *`MeshState`* is also an interface type but has its own file.

Note: `UNITY_INTERFACE_EXPORT` is a macro provided by Unity in `external/Unity/IUnityInterface.h`, which allows the function to be callable from C# (given it is within an `extern "C"` clause)

---

struct Vector3
#include <InterfaceTypes.h> The Unity *Vector3* with functonality for converting to/from Eigen::Vector3f (float).

Public Functions

Vector3() = default

Vector3(float *x*, float *y*, float *z*)

Vector3(Eigen::Vector3f *value*)

Eigen::Vector3f **AsEigen**() **const**

Eigen::RowVector3f **AsEigenRow**() **const**

Public Members

float **x**

float **y**

float **z**

Public Static Functions

Vector3 **Zero**()

struct Quaternion

#include <InterfaceTypes.h> The Unity *Quaternion* with functonality for converting to/from Eigen::Quaternionf (float).

Beware: Unity and Eigen have different conventions for ordering the values.

Public Functions

Quaternion() = default

Quaternion(float *x*, float *y*, float *z*, float *w*)

Quaternion(Eigen::Quaternionf &*q*)

Eigen::Quaternionf **AsEigen**() **const**

Warning Eigen has a different ordering of the values, handled safely by this function. We cannot simply reinterpret the bits.

Public Members

float **x**

float **y**

float **z**

float **w**

Public Static Functions

Quaternion **Identity**()

struct DirtyFlag

#include <InterfaceTypes.h> Marks which data has changed in *UMeshDataNative* as a bitmask

Public Static Attributes

const unsigned int **None** = 0

const unsigned int **VDirty** = 1

const unsigned int **NDirty** = 2

const unsigned int **CDirty** = 4

const unsigned int **UVDirty** = 8

const unsigned int **FDirty** = 16

const unsigned int **DontComputeNormals** = 32
Don't recaluclate normals when VDirty is set. NDirty overrides this.

const unsigned int **DontComputeBounds** = 64
Don't recalculate bounds when VDirty is set. Bounds are used for occlusion culling.

const unsigned int **DontComputeColorsBySelection** = 128
Don't recompute colors if a visible selection has changed.

const unsigned int **VDirtyExclBoundary** = 256
Use this when the vertex positions have changed, but the boundary conditions are unaffected. VDirty overrides this.

const unsigned int **All** = (unsigned int)-1 - *DontComputeNormals* - *DontComputeBounds* - *DontComputeColorsBySelection*

struct UMeshDataNative
#include <InterfaceTypes.h> Stores all pointers to the MeshData arrays.

Usually this should be as a `const` parameter.

Public Members

float ***VPtr**

float ***NPtr**

float ***CPtr**

float ***UVPtr**

int ***FPtr**

int **VSize**

int **FSize**

struct SelectionMode
#include <InterfaceTypes.h> Constants related to how a select operation modifies the current selection.

Public Static Attributes

const unsigned int **Add** = 0

const unsigned int **Subtract** = 1

const unsigned int **Toggle** = 2

4.2.3 NativeCallbacks.h

Defines

STR(*message*)
Macro to easily concatenate strings using stringstream, use the operator<<

```
STR("My value: " << 123)
```

LOG(*message*)
Macro to easily and safely print to the Unity Debug.Log, disabled in release. Uses STR.

```
LOG("My value: " << 123)
```

LOGWARN(*message*)
Call Unity Debug.LogWarning safely. Uses STR.

LOGERR(*message*)
Call Unity Debug.LogError safely. Uses STR.

Typedefs

typedef void (***StringCallback**)(**const** char *message)
Function pointer to a C# delegate: `void MyFct(string message)`

Note C# delegates are `fixed` by default, so we do not have to worry about these pointers becoming invalid due to the Garbage Collector.

Variables

StringCallback **DebugLog**
Print to the Unity Debug.Log. Check that the function pointer is not null before using

```
if (DebugLog) DebugLog("Hello");
```

This is what the LOG macro does, use that instead.

See Callbacks like this are set in *Initialize* and reset to `nullptr` in *UnityPluginUnload*

StringCallback **DebugLogWarning**

StringCallback **DebugLogError**

4.2.4 Deform.h

This is where the deformations are as well as other functions which manipulate the vertices. This (the .cpp) is a good place to start for how to implement your own deformation.

Functions

bool **UpdateBoundary**(*MeshState* **state*, unsigned int *boundaryMask*)
Recalculates the boundary `state->Native->Boundary` *MeshStateNative.Boundary* if the relevant selections have changed

Return True if boundary has changed

Parameters

- `boundaryMask`: The current mask of selections part of the boundary

bool **UpdateBoundaryConditions**(*MeshState* **state*)
Recalculates the boundary conditions `state->Native->BoundaryConditions` *MeshStateNative.BoundaryConditions* for Harmonic and Arap

Return True if the boundary conditions have changed

4.2.5 MeshState.h

This is the shared state between C++/C# and changes in one **must** be applied to the other. If the two structs do not match *exactly* problems arise with reading/writing to the wrong memory.

struct MeshState

#include <MeshState.h> Stores all data related to a specific mesh. Members are shared between native(C++) and managed(C#). Some members point to native only or managed only and are depicted as void*

Public Functions

MeshState(*UMeshDataNative udata*)

Initialise the shared state from a Unity mesh

Parameters

- `udata`: All data required to create the state

~MeshState()

This is where all C++ allocated memory for a mesh is deleted.

Public Members

unsigned int **DirtyState** = {*DirtyFlag*::*None*}

Tells us what has changed with the mesh using the *DirtyFlag* constants

unsigned int **DirtySelections** = {0}

Tells us which selections have been modified, as a bitmask. Each bit represents one selection.

unsigned int **DirtySelectionsResized** = {0}

Less stricter version than DirtySelections, where we only consider a selection dirty if the selected vertices size changes, see SSizes.

Eigen::MatrixXf ***V**

The vertex matrix in column major with dimensions VSize x 3. Stores position for each vertex, one row represents one vertex.

Eigen::MatrixXf ***N**

The normals matrix in column major with dimensions VSize x 3. Stores the normal for each vertex.

Eigen::MatrixXf ***C**

The rgba color matrix in column major with dimensions VSize x 4. Stores the color for each vertex.

Eigen::MatrixXf ***UV**

The UV0 matrix in column major with dimensions VSize x 2. Stores the 2D uv coordinate for each vertex.

Eigen::MatrixXi ***F**

The Face/Indices matrix in column major with dimensions FSize x 3. Stores the vertex indices for each face/triangle, one row represents one face.

int **VSize** = {0}

Number of vertices, columns in V

int **FSize** = {0}

Number of faces, columns in F

Eigen::VectorXi ***S**

The selection vector, stores the selection state for each vertex. We store one uint per vertex. The selection is represented as a bitmask, with each bit indicating if the vertex is in that selection or not. So there are max 32 selections (32 bits)

unsigned int **SSize** = {1}
Amount of selections that are in use

unsigned int ***SSizes**
uint[32], number of vertices selected per selection. Stored as a pointer so we can easily share this with C#.

unsigned int **SSizesAll** = {0}
Total vertices selected

MeshStateNative ***Native**
Native only state, a void* in C#

4.2.6 MeshStateNative.h

This contains mesh specific data only used in C++, e.g. pre-calculations.

struct MeshStateNative
#include <MeshStateNative.h> Contains all variables that are only used in C++ for a specific mesh. We use one *MeshStateNative* per mesh.

Public Functions

MeshStateNative(Eigen::MatrixXf *V)

~MeshStateNative()

Public Members

Eigen::VectorXi **Boundary** = {Eigen::VectorXi::Zero(0)}
Vertices part of the boundary. Has a variable length.

Note Evaluated in a lazy manner.

Eigen::MatrixXf **BoundaryConditions** = {Eigen::VectorXf::Zero(0)}
Positions of vertices in the Boundary (rows correspond). Has a variable length, the same as Boundary.

Note Evaluated in a lazy manner.

unsigned int **BoundaryMask** = {0}
The selections currently used for the Boundary.

Note Evaluated in a lazy manner.

unsigned int **DirtySelectionsForBoundary** = {0}
Selections that have changed since the last time the Boundary was calculated. Used for lazy recalculation of Boundary.

bool **DirtyBoundaryConditions** = {true}
Whether the boundary conditions have changed since the last time they were calculated. Used for lazy recalculation of BoundaryConditions.

Eigen::MatrixXf ***V0**
Initial V, before deformations. Used for deformations and resetting V

bool **harmonicShowDeformationField** = {false}
The harmonic deformation field value at the last recalculation

igl::ARAPData<float> ***ArapData** = {nullptr}
Pre-computations for Arap

4.2.7 Util.h

Contains various helper functions, classes and constants.

Typedefs

using Color_t = Eigen::RowVector4f
RGBA color

Functions

template<typename **Matrix**, typename **Scalar**>
void **TransposeToMap**(*Matrix* *from, *Scalar* *to)
Transpose an Eigen::Matrix to an Eigen::Map, given by the pointer to the first element Dimensions are inferred from the Matrix

Template Parameters

- `Matrix`: An Eigen Matrix
- `Scalar`: Type on one element

Parameters

- `to`: Pointer to the first element of a matrix or an array

template<typename **Scalar**, typename **Matrix**>
void **TransposeFromMap**(*Scalar* *from, *Matrix* *to)
Transpose an Eigen::Map to an Eigen::Matrix

Template Parameters

- `Scalar`: Type on one element
- `Matrix`: An Eigen Matrix

Parameters

- `from`: Pointer to the first element of a matrix or an array

template<typename **V_T**>
void **CenterToMean**(float *VPtr, int *VSize*)
Set the center/origin of the mesh to be the mean vertex

Template Parameters

- `V_T`: Type of the vertex matrix to support both Col and RowMajor

template<typename **V_T**>
void **ApplyScale**(float *VPtr, int *VSize*, bool *centerToMean* = true, bool *normalize* = true, float *targetScale* = 1.f)
Applies the scale of a mesh to the vertices, i.e. cwise multiply. If `targetScale` is set to zero, the model scale is normalized so it has unit height.

Note y-axis center will be the center of the bounding box for easier positioning

Parameters

- `centerToMean`: If true sets the center to the mean vertex.
- `normalize`: If set to true the absolute y-height of the model will be `targetScale` otherwise only the targetScale factor is applied.

Template Parameters

- `V_T`: Type of the vertex matrix to support both Col and RowMajor

struct Color

#include <*Util.h*> Contains color constants

Public Static Functions

const *Color_t* &**GetColorById**(int *selectionId*)

Gets color based on the selectionId, matched C#

Public Static Attributes

Color_t **White**

Color_t **Gray**

Color_t **Black**

Color_t **Red**

Color_t **Green**

Color_t **Blue**

Color_t **Orange**

Color_t **Purple**

Color_t **GreenLight**

Color_t **BlueLight**

Color_t **Yellow**

4.3 C++ External Libraries

Libigl among other third party libraries are in the `external` folder.

- `libigl`, This is currently a custom fork with some modifications. It also contains Eigen.
  - `arap*` files have been edited to allow for use of `float` instead of `double`.
- `eigen-debug`, Stores natvis files for pretty printing Eigen matrices when debugging.
- `UnityNativeTool`, This is the source code for the small `stubLluiPlugin.dll` library used by the UnityNativeTool. All it does is get the function pointer to the Unity C++ interface class `IUnityInterfaces`. This is required such that we get the callbacks for `UnityPluginLoad()` and `UnityPluginUnload()` when running in the Unity editor. It is compiled via CMake.
- `Unity/PluginAPI`, This has functions related calling Unity related functions from C++. Mostly unused.
- `Unity/RenderAPI`, This includes headers related to sample usage of the render API (e.g. DirectX) from C++. It is currently not used.

4.3.1 UnityNativeTool - Stub UnityInterfaces

A one file library that gets us the pointer to the UnityInterfaces. This library is not mocked.

Functions

void **UnityPluginLoad**(IUnityInterfaces **unityInterfaces*)
Called when the plugin is loaded, this can be after/before *Initialize()*

Declared in IUnityInterface.h

Parameters

- `unityInterfaces`: Unity class for accessing minimal Unity functionality exposed to C++ Plugins

IUnityInterfaces ***GetUnityInterfacesPtr**()
Allow us to retrieve this pointer from C#, so that we can manually call *`UnityPluginLoad()`* for mocked libraries

Variables

IUnityInterfaces ***s_IUnityInterfaces** = NULL

4.3.2 Unity C++ Plugin API

These files contain the (quite limited) Unity C++ API for plugins and is only really intended for special cases. These files are available from any Unity installation. They are located in `<Unity Install Dir>/Editor/Data/PluginAPI` , e.g. `C:\Program Files\Unity\Hub\Editor\2019.3.2f1\Editor\Data\PluginAPI`.

> **Warning:** Comments here mainly reflect *my understanding* of the API! This part is just for reference, but if you really intend on using this you need to look at the source code (which also has a lot more comments).

IUnityInterface.h

The main file to handle the interface between the C++ library and Unity.

Defines

UNITY_INTERFACE_API
Contains appropriate calling convention that C# uses. Use this for C# delegates. https://en.wikipedia.org/wiki/X86_calling_conventions#stdcall

UNITY_INTERFACE_EXPORT
Prepend this to an extern "C" function to make it callable from C#.

UNITY_DECLARE_INTERFACE(*NAME*)

UNITY_REGISTER_INTERFACE_GUID(*HASHH*, *HASHL*, *TYPE*)

UNITY_REGISTER_INTERFACE_GUID_IN_NAMESPACE(*HASHH*, *HASHL*, *TYPE*, *NAMESPACE*)

UNITY_GET_INTERFACE_GUID(*TYPE*)

UNITY_GET_INTERFACE(*INTERFACES*, *TYPE*)

Typedefs

typedef struct UnityInterfaceGUID **UnityInterfaceGUID**

typedef void **IUnityInterface**

typedef struct IUnityInterfaces **IUnityInterfaces**

typedef struct RenderSurfaceBase ***UnityRenderBuffer**

typedef unsigned int **UnityTextureID**

Functions

void **UnityPluginLoad**(IUnityInterfaces **unityInterfaces*)
Called when the plugin is loaded, this can be after/before *Initialize()*

Declared in IUnityInterface.h

Parameters

- `unityInterfaces`: Unity class for accessing minimal Unity functionality exposed to C++ Plugins

void **UnityPluginUnload**()
Called when the plugin is unloaded, clean up here Declared in IUnityInterface.h

struct UnityInterfaceGUID
#include <IUnityInterface.h>

Public Members

unsigned long long **m_GUIDHigh**

unsigned long long **m_GUIDLow**

struct IUnityInterfaces
#include <IUnityInterface.h>

Public Members

IUnityInterface *(***GetInterface**)(UnityInterfaceGUID guid)

void (***RegisterInterface**)(UnityInterfaceGUID guid, IUnityInterface *ptr)

IUnityInterface *(***GetInterfaceSplit**)(unsigned long long guidHigh, unsigned long long guidLow)

void (***RegisterInterfaceSplit**)(unsigned long long guidHigh, unsigned long long guidLow, IUnityInterface *ptr)

IUnityProfilerCallbacks.h

This can be used to create custom profiling 'timestamps'

Typedefs

typedef uint32_t **UnityProfilerMarkerId**

typedef uint16_t **UnityProfilerCategoryId**

typedef uint64_t **UnityProfilerThreadId**

typedef struct UnityProfilerCategoryDesc **UnityProfilerCategoryDesc**

typedef uint16_t **UnityProfilerMarkerFlags**

typedef uint16_t **UnityProfilerMarkerEventType**

typedef struct UnityProfilerMarkerDesc **UnityProfilerMarkerDesc**

typedef uint8_t **UnityProfilerMarkerDataType**

typedef struct UnityProfilerMarkerData **UnityProfilerMarkerData**

typedef uint8_t **UnityProfilerFlowEventType**

typedef struct UnityProfilerThreadDesc **UnityProfilerThreadDesc**

typedef void (*__IUnityProfilerCreateCategoryCallback__)(**const** UnityProfilerCategoryDesc *categoryDesc, void *userData)

typedef void (*__IUnityProfilerCreateMarkerCallback__)(**const** UnityProfilerMarkerDesc *markerDesc, void *userData)

typedef void (*__IUnityProfilerMarkerEventCallback__)(**const** UnityProfilerMarkerDesc *markerDesc, UnityProfilerMarkerEventType eventType, uint16_t eventDataCount, **const** UnityProfilerMarkerData *eventData, void *userData)

typedef void (*__IUnityProfilerFrameCallback__)(void *userData)

typedef void (*__IUnityProfilerThreadCallback__)(**const** UnityProfilerThreadDesc *threadDesc, void *userData)

typedef void (*__IUnityProfilerFlowEventCallback__)(UnityProfilerFlowEventType flowEventType, uint32_t flowId, void *userData)

typedef struct IUnityProfilerCallbacksV2 **IUnityProfilerCallbacksV2**

typedef struct IUnityProfilerCallbacks **IUnityProfilerCallbacks**

Enums

enum UnityProfilerMarkerFlag_

Values:

enumerator kUnityProfilerMarkerFlagDefault = 0

enumerator kUnityProfilerMarkerFlagScriptUser = 1 << 1

enumerator kUnityProfilerMarkerFlagScriptInvoke = 1 << 5

enumerator kUnityProfilerMarkerFlagScriptEnterLeave = 1 << 6

enumerator kUnityProfilerMarkerFlagAvailabilityEditor = 1 << 2

enumerator kUnityProfilerMarkerFlagAvailabilityNonDev = 1 << 3

enumerator kUnityProfilerMarkerFlagWarning = 1 << 4

enumerator kUnityProfilerMarkerFlagVerbosityDebug = 1 << 10

enumerator kUnityProfilerMarkerFlagVerbosityInternal = 1 << 11

enumerator kUnityProfilerMarkerFlagVerbosityAdvanced = 1 << 12

enum UnityProfilerMarkerEventType_

Values:

enumerator kUnityProfilerMarkerEventTypeBegin = 0

enumerator kUnityProfilerMarkerEventTypeEnd = 1

enumerator kUnityProfilerMarkerEventTypeSingle = 2

enum UnityProfilerMarkerDataType_

Values:

enumerator kUnityProfilerMarkerDataTypeNone = 0

enumerator kUnityProfilerMarkerDataTypeInstanceId = 1

enumerator kUnityProfilerMarkerDataTypeInt32 = 2

enumerator kUnityProfilerMarkerDataTypeUInt32 = 3

enumerator kUnityProfilerMarkerDataTypeInt64 = 4

enumerator kUnityProfilerMarkerDataTypeUInt64 = 5

enumerator kUnityProfilerMarkerDataTypeFloat = 6

enumerator kUnityProfilerMarkerDataTypeDouble = 7

enumerator kUnityProfilerMarkerDataTypeString = 8

enumerator kUnityProfilerMarkerDataTypeString16 = 9

enumerator kUnityProfilerMarkerDataTypeBlob8 = 11

enum UnityProfilerFlowEventType_

Values:

enumerator kUnityProfilerFlowEventTypeBegin = 0

enumerator kUnityProfilerFlowEventTypeNext = 1

enumerator kUnityProfilerFlowEventTypeEnd = 2

Variables

const UnityInterfaceGUID **IUnityProfilerCallbacksV2_GUID** = {0x5DEB59E88F2D4571ULL, 0x81E8583069A5E33CU

const UnityInterfaceGUID **IUnityProfilerCallbacks_GUID** = {0x572FDB38CE3C4B1FULL, 0xA6071A9A7C4F52D8UL

struct UnityProfilerCategoryDesc
#include <IUnityProfilerCallbacks.h>

Public Members

UnityProfilerCategoryId **id**

uint16_t **reserved0**

uint32_t **rgbaColor**

const char ***name**

struct UnityProfilerMarkerDesc
#include <IUnityProfilerCallbacks.h>

Public Members

const void ***callback**

UnityProfilerMarkerId **id**

UnityProfilerMarkerFlags **flags**

UnityProfilerCategoryId **categoryId**

const char ***name**

const void ***metaDataDesc**

struct UnityProfilerMarkerData
#include <IUnityProfilerCallbacks.h>

Public Members

UnityProfilerMarkerDataType **type**

uint8_t **reserved0**

uint16_t **reserved1**

uint32_t **size**

const void ***ptr**

struct UnityProfilerThreadDesc
#include <IUnityProfilerCallbacks.h>

Public Members

uint64_t **threadId**

const char ***groupName**

const char ***name**

struct IUnityProfilerCallbacksV2
#include <IUnityProfilerCallbacks.h>

Public Members

int (***RegisterCreateCategoryCallback**)(IUnityProfilerCreateCategoryCallback callback, void *userData)

int (***UnregisterCreateCategoryCallback**)(IUnityProfilerCreateCategoryCallback callback, void *userData)

int (***RegisterCreateMarkerCallback**)(IUnityProfilerCreateMarkerCallback callback, void *userData)

int (***UnregisterCreateMarkerCallback**)(IUnityProfilerCreateMarkerCallback callback, void *userData)

int (***RegisterMarkerEventCallback**)(**const** UnityProfilerMarkerDesc *markerDesc, IUnityProfilerMarkerEventCallback callback, void *userData)

int (***UnregisterMarkerEventCallback**)(**const** UnityProfilerMarkerDesc *markerDesc, IUnityProfilerMarkerEventCallback callback, void *userData)

int (***RegisterFrameCallback**)(IUnityProfilerFrameCallback callback, void *userData)

int (***UnregisterFrameCallback**)(IUnityProfilerFrameCallback callback, void *userData)

int (***RegisterCreateThreadCallback**)(IUnityProfilerThreadCallback callback, void *userData)

int (***UnregisterCreateThreadCallback**)(IUnityProfilerThreadCallback callback, void *userData)

int (***RegisterFlowEventCallback**)(IUnityProfilerFlowEventCallback callback, void *userData)

int (***UnregisterFlowEventCallback**)(IUnityProfilerFlowEventCallback callback, void *userData)

struct IUnityProfilerCallbacks
#include <IUnityProfilerCallbacks.h>

Public Members

int (***RegisterCreateCategoryCallback**)(IUnityProfilerCreateCategoryCallback callback, void *userData)

int (***UnregisterCreateCategoryCallback**)(IUnityProfilerCreateCategoryCallback callback, void *userData)

int (***RegisterCreateMarkerCallback**)(IUnityProfilerCreateMarkerCallback callback, void *userData)

int (***UnregisterCreateMarkerCallback**)(IUnityProfilerCreateMarkerCallback callback, void *userData)

int (***RegisterMarkerEventCallback**) (**const** UnityProfilerMarkerDesc *markerDesc, IUnityProfilerMarkerEventCallback callback, void *userData)

int (***UnregisterMarkerEventCallback**) (**const** UnityProfilerMarkerDesc *markerDesc, IUnityProfilerMarkerEventCallback callback, void *userData)

int (***RegisterFrameCallback**) (IUnityProfilerFrameCallback callback, void *userData)

int (***UnregisterFrameCallback**) (IUnityProfilerFrameCallback callback, void *userData)

int (***RegisterCreateThreadCallback**) (IUnityProfilerThreadCallback callback, void *userData)

int (***UnregisterCreateThreadCallback**) (IUnityProfilerThreadCallback callback, void *userData)

IUnityGraphics.h

Gives access to the graphics API, i.e. DirectX, OpenGL, Vulkan or Metal. These each have their own specialised implementation files. The file `source/sample/CustomUploadMesh.cpp` tries to use the native graphics API to access and update the GPU buffer directly. This is related to the RenderAPI

Typedefs

typedef enum UnityGfxRenderer **UnityGfxRenderer**

typedef enum UnityGfxDeviceEventType **UnityGfxDeviceEventType**

typedef void (***IUnityGraphicsDeviceEventCallback**) (UnityGfxDeviceEventType eventType)

typedef struct IUnityGraphics **IUnityGraphics**

typedef void (***UnityRenderingEvent**) (int eventId)

typedef void (***UnityRenderingEventAndData**) (int eventId, void *data)

Enums

enum UnityGfxRenderer

Values:

enumerator kUnityGfxRendererD3D11 = 2

enumerator kUnityGfxRendererNull = 4

enumerator kUnityGfxRendererOpenGLES20 = 8

enumerator kUnityGfxRendererOpenGLES30 = 11

enumerator kUnityGfxRendererPS4 = 13

enumerator kUnityGfxRendererXboxOne = 14

enumerator kUnityGfxRendererMetal = 16

enumerator kUnityGfxRendererOpenGLCore = 17

enumerator kUnityGfxRendererD3D12 = 18

enumerator kUnityGfxRendererVulkan = 21

enumerator kUnityGfxRendererNvn = 22

enumerator kUnityGfxRendererXboxOneD3D12 = 23

enum UnityGfxDeviceEventType

Values:

enumerator kUnityGfxDeviceEventInitialize = 0

enumerator kUnityGfxDeviceEventShutdown = 1

enumerator kUnityGfxDeviceEventBeforeReset = 2

enumerator kUnityGfxDeviceEventAfterReset = 3

Variables

const UnityInterfaceGUID **IUnityGraphics_GUID** = {0x7CBA0A9CA4DDB544ULL, 0x8C5AD4926EB17B11ULL}

struct IUnityGraphics

#include <IUnityGraphics.h>

Public Members

UnityGfxRenderer (***GetRenderer**) ()

void (***RegisterDeviceEventCallback**) (IUnityGraphicsDeviceEventCallback callback)

void (***UnregisterDeviceEventCallback**) (IUnityGraphicsDeviceEventCallback callback)

int (***ReserveEventIDRange**) (int count)

4.3.3 Unity C++ Render API

This is an example use case by Unity of the IUnityGraphics.h from the PluginAPI, sourced from Github Unity-Technologies/NativeRenderingPlugin.

RenderAPI.h

Functions

RenderAPI ***CreateRenderAPI** (UnityGfxRenderer *apiType*)

class RenderAPI

#include <RenderAPI.h>

Public Functions

~RenderAPI ()

void **ProcessDeviceEvent** (UnityGfxDeviceEventType *type*, IUnityInterfaces **interfaces*) = 0

bool **GetUsesReverseZ** () = 0

void **DrawSimpleTriangles** (**const** float *worldMatrix*[16], int *triangleCount*, **const** void **verticesFloat3Byte4*) = 0

void *__BeginModifyTexture__(void *textureHandle, int textureWidth, int textureHeight, int *outRowPitch) = 0

void __EndModifyTexture__(void *textureHandle, int textureWidth, int textureHeight, int rowPitch, void *dataPtr) = 0

void *__BeginModifyVertexBuffer__(void *bufferHandle, size_t *outBufferSize) = 0

void __EndModifyVertexBuffer__(void *bufferHandle) = 0

CHAPTER

FIVE

REPORT

Note: This report can be read as a pdf from the GitHub release.

5.1 Introduction

Note: This report will focus on the development approach as well as evaluating the project and suggesting future improvements. *For implementation details and how to use this project see the online documentation, where the gallery can also be found* [1.1].

5.1.1 Purpose

The purpose of this thesis is to produce an extensible virtual reality (VR) viewer and editor for use with the libigl library [1.2]. In effect converting the current 2D user interface to the VR setting. Potential use cases include visualizations of 3D models and operations on them, such as those provided by libigl.

VR provides an alternative input and output format in comparison to a conventional 2D screen with a keyboard and mouse. It allows for accurate representation of 3D scenes, notably in terms of depth and scale, due to its stereoscopic rendering.

In terms of input, VR controllers can give precise 3D positional and rotational input for each hand in comparison to 2D positional input from a mouse. This is also superior to 3D mice, which only offer relative 3D positional and rotational input. VR is useful in our scenario as it allows for easier and more intuitive interaction with a mesh.

5.1.2 Related Work

Advances in VR input methods, with 6DOF controllers for each hand, has allowed for more innovation in 3D modeling. Several 3D modeling applications already exist on the Oculus Store such as Google Blocks [1.3] and Facebook's Quill [1.4]. These offer construction of 3D scenes from primitives and brush strokes. Both use a 'palette'-like user interface (UI) on the secondary hand for adding and manipulating meshes. Blocks has more basic functionality, but integrates with a sharing service online. Quill also offers snapping tools and being able to animate scenes.

Oculus Medium [1.5] provides more advanced functionality with sculpting and uses an unconventional method for storing its meshes allowing for fast boolean operations. "Medium defines 3D objects (sculpts) using an implicit surface. The surface is stored as a signed distance field (SDF) in a 3D grid of voxels." [1.6].

Blender [1.7] has recently added a VR scene inspection [1.8] add-on, see also release notes [1.9]. This currently only enables the user to view a scene in VR and does not make use of the controllers. Unreal Engine [1.10] since version

4.17 has developed a VR mode [1.11] for its editor. This is primarily used for level design and previewing scenes. It replicates its 2D editor windows in VR as floating panels allowing a similar level of functionality to the 2D editor. It also makes use of nested pie menus for common actions like snapping. Both Blender and Unreal Engine allow for easy switching between 2D and VR.

In summary, there already exist several VR editors for 3D modeling and animation. However, this is still a developing field with little standardization. User interface is generally interacted with by using raycasting and a secondary hand is often dedicated for quick UI actions. Except for sculpting, complex deformation of meshes in VR has not yet been explored. This is what this thesis focuses on.

5.1.3 System Components

The Oculus Rift S headset was used as the primary target device.

For implementing this the Unity [1.12] game engine was chosen, partly due to experience with the engine. It provides many standard features as well as a cross-platform VR integration. It offers advanced VR features such as single-pass stereo rendering, which provides great performance benefits. Furthermore, it has an easy way of adding functionality via C# scripts. This, however, creates a necessary language interface to C++ such that libigl can be used.

Using the Oculus SDK directly requires too much development overhead and will result in less features. It will also be significantly harder to maintain. Using a game engine which already provides a range of features is the best option given the time available.

For example use cases of libigl two mesh deformations where chosen, a biharmonic deformation [1.13] and an As-Rigid-As-Possible deformation [1.14]. These each require selection of parts of the mesh as well as ways of transforming these, such that we can provide boundary conditions for the libigl algorithms. The deformations have been chosen from the libigl tutorial. The intent is that further libigl functionality, of any kind, can also be added.

Ideally, existing libigl applications would be able to simply switch which viewer is being used, either the current libigl 2D GLFW viewer and the VR viewer. This is not possible with this development approach. This is because libigl will be a library used by the VR editor and not the converse, due to how Unity executables are built. As a result, the interface to the VR viewer cannot be the same as the 2D GLFW viewer.

A workaround to this would be to implement inter-process communication between a libigl executable and a Unity built executable. This is however more involved and outside of the scope of this thesis. It is also unclear whether this approach will yield performant and maintainable results.

5.2 Method

The implementation has following categories:

1. Backend
    1. C#/C++ language interface
    2. Threading of libigl calls
    3. Mesh interface between libigl and Unity
2. Frontend
    1. VR interface
    2. Handling of input and user interface (UI)
    3. High-level actions utilizing libigl
3. Documentation process

5.2.1 Backend

Threading

In order to have a high framerate, the expensive computations done by libigl must be performed on a worker thread. The Unity API, such as getting the transform position, is not thread-safe and thus use from a worker thread is forbidden. This has several implications, with the main one being that all access to the Unity API must be done before starting the thread and the results should be copied. This is what *PreExecute* and the *MeshState* are for.

If the thread wants to make changes to the Unity state, e.g. moving an object, then this must be deferred to the main thread. Here this is done once the thread has finished in *PostExecute*, however, a concurrent queue of actions could also be used.

As we want to execute certain operations every frame and apply their changes, we have a loop of `PreExecute`, `Execute` and `PostExecute`. Where `PreExecute` and `PostExecute` are performed on the main thread. Notably, in `PostExecute` we apply changes to the mesh done by libigl.

Initially, the Unity C# Job System was used. However, this did not provide enough control as only value types could be passed to the thread in order to prevent race conditions. Another model based on a concurrent queue of actions was implemented. The main thread would push an action to the queue to be performed by the worker thread. This was not flexible enough and there was a lack of clear control over the ordering of actions.

C#/C++ Language Interface

To call libigl functions a necessary C#/C++ language interface is required. This adds an extra layer of complexity. We must consider in which language functionality and data resides and what is shared. An important note is that the Unity API is only accessible within C#. Using the libigl python bindings would also require a language interface as Unity does not directly support python.

Functionality

It is important to have a clear distinction of what is done where. All expensive mathematical operations are generally done in C++. The number of interface functions is kept minimal. All input collection and high-level actions are done in C#.

Development of a C++ library inside Unity is particularly challenging as an unhandled exception or runtime error will crash the Unity editor. This can be mitigated be placing assertions, which pause the execution and allow for the attachment of a debugger.

Data

In data we also have a distinction. Shared datatypes are possible but must be declared in both languages, including Unity types such as `Vector3`. By convention, each mesh has a state shared between the languages. This will point also to any other native-only data. By doing this, we can easily handle multiple meshes and in future serialization.

When the pointers are used the C# garbage collector needs to be considered. Within a function this is not a problem as data is `fixed` throughout a native function call. However, if a C# data is passed to C++ and its pointer is used after the function scope, then the memory may have moved and the pointer is no longer valid. This data should be pinned with `GcHandle.Alloc`. To avoid this scenario persistent data is allocated, and thus deleted, in C++, notably the *MeshState*.

Marshalling of managed types are an additional consideration. Passing large non-blittable types to a native function can result in expensive memory operations, in particular 2D C# arrays which have a different layout in each language.

Compiling and CMake

CMake is used to compile the C++ library as well as the documentation in a cross-platform/IDE manner. It helps with finding libraries, but also streamlines the compile process by immediately copying the output `.dll` to the Unity project. The end product is built inside Unity. This also ensures the project remains cross-platform.

Unity Plugin Reloading

Unity presents a complication by never unloading libraries once they are loaded, which happens lazily when they are first used. This means that we cannot recompile the C++ library without restarting Unity, creating a much larger iteration time. In order to counter this, the UnityNativeTool [2.3] open source project is used. This effectively wraps native functions and un/loads the library itself. It is an editor-only tool. A few modifications were made to this in several pull requests, see #14, #15, #18, #19, #20, #21, #28 on GitHub.

Future Work

The C++ interface could be simplified by using a tool such as SWIG [2.4], see also [2.5]. This integrates with CMake and automatically generates the C# declarations as well as having more advanced features such as exception handling between languages. Primarily, it removes redundant code and documentation. However, this simply shifts the development complexity, but it does make the language interface more robust to bugs.

Of course, alternatives to Unity such as Blender or Unreal Engine do not have this interface. In contrast, the difficult parts with the language interface have been done and development should be easier from hereon.

Additionally, implementing serialization of the entire C++ state with the help of `igl::serialize` would be beneficial. This state would only involve all *`MeshState`* instances. It would enable faster testing as well as enabling the use hot-reloading of the C++ library while paused, decreasing the iteration time. Hot-reloading was attempted by simply not deleting the allocated *`MeshState`* memory and retaining the pointers in C#. However as the library is fully unloaded, its entire memory is deallocated and the C# pointers are invalidated.

Mesh Interface

Once we have modified the mesh data used by the renderer, such as the vertex positions, we need to apply these changes. This is the equivalent of `viewer.data().set_vertices(V)` in the 2D viewer. This requires access to the Unity API, so must be done on the main thread. It is done in *`PostExecute`*. A bitmask *`DirtyState`* is used to indicate which parts have been modified and need to be updated. This is done in a coarse-grained fashion. For example, if a single vertex is moved the entire position matrix is updated. This sparse editing of the mesh occurs frequently, for example when an operation is performed on a selection. As a result, this could be a potential area of improvement, which could be fixed by accessing the GPU buffer directly, see *More on Performance*.

An extra complication to this is that Unity uses row-major and libigl expects column-major matrices. Because of this we have two copies of the data, one in column-major and one in row-major. This creates a necessary transpose each time we apply changes. To mitigate performance losses, this is done in C++ on the worker thread. For larger meshes the effect of this transpose on runtime as well as memory performance will be more noticeable. For the meshes tested, this was not an issue with operations on the armadillo mesh [2.6] still being responsive.

Ideally libigl would work equally well in row-major preventing a transpose and reducing the number of copies of the mesh in memory. Although Eigen [2.7] supports row-major well, libigl templates do not always consider this causing compiler errors.

In this part of the development process, the engine source code would have been beneficial.

More on Performance

Unity provides the GPU pointer to the mesh buffer. Thus a way of applying the mesh data directly to the GPU was briefly explored with help of the NativeRenderingPlugin [2.8] example.

Another performance consideration is that vertex attributes are interleaved by default on the GPU in the vertex buffer. This means that updating the position of all vertices results in a non-blittable transfer. This could result in a performance loss. Unity exposes some control over the vertex buffer layout allowing separation of vertex attributes into separate 'streams'. This could be explored further if this process appears to be a performance bottleneck.

5.2.2 Frontend

VR Interface

Oculus provides an Oculus Integration [2.9] on the Unity Asset Store to provide common functionality. However, since Unity 2019.3 there has been a Unity XR Plug-in Framework [2.10] package to simplify the interface with each of the SDKs of the VR platforms. Additionally, there is a Unity XR Interaction Toolkit [2.11] package with provides cross-platform input as well as common VR functionality such as locomotion and interaction with UI. These packages are the preferred option over the Oculus Integration and will ensure that the application can be used on most VR devices.

Locomotion

This involves moving the user in the virtual world. The common approach is to use teleportation with a curved ray to translate the user and snap rotation to turn user. Using a curved ray for indicating where to translate to has the benefit that there is an approximately linear mapping between the angle of the controller and the distance up to a certain point. This makes is easy to precisely indicate a teleportation position far away from the user. Immediate locomotion rather than a smooth interpolation is used to reduce motion sickness.

Input

The Unity XR Interaction Toolkit [2.11] is used for this for getting cross-platform input. It is also used for detecting the controllers.

Contextual Input

Contextual input is when we adapt what the mapping of raw inputs to actions based on the context or state and is important for two reasons. Firstly, a key feature of making input intuitive is by making it adapt to the current context. Secondly, contextual input helps to make maximum use of the limited degrees of freedom provided by the physical input device. When using a keyboard this has never been particularly important for smaller applications.

Context can be inferred from the state of the application or explicitly set by the user, for example when choosing a specific tool. Ideally most of the context is inferred implicitly from what the user is doing. For example, while the user is grabbing a selection we want to provide relevant further input options to that context, such as being able to change the pivot mode or whether rotation is enabled. Having different tools for editing a single mesh and operating with multiple meshes was inspired by the object and edit modes in Blender [1.7]

In this project, the state is inferred using a tree, see documentation [2.12]. This allows for easily determining the context. However, it is unclear how well this will scale once the state space increases. This is an area which could be improved in the future.

User Interface

Overview

Despite this being a VR application, some form of a 2D User Interface (UI) is still necessary. It allows for displaying information as well as providing access to infrequently used actions. Frequently used actions should ideally be mapped directly to contextual controller input. However, if all degrees of freedom in the controller input are already used, then the 2D UI will be the fallback option.

The UI is implemented as a world-space canvas. Using a traditional screen space UI is not an option in VR. This creates the problem of positioning of the UI. The relevant UI needs to be displayed at the right time for an intuitive experience. In this project, the user can grab UI panels and position them explicitly, similar to the Unreal Engine VR Mode. In future, methods of implicitly positioning the UI and displaying relevant parts may work better. Also having the user grab a panel by default when it is created to let them set the initial position will be an improvement, as currently panels are simply arranged in an array-like fashion.

Generation

In order to be able to add new functionality easily, generating UI via C# scripting is done. The goal is to be able to easily add new UI elements and configure them, in particular setting their `OnClick` behavior. Inspired by the 2D libigl UI, we simply have a scrollable vertical layout group, so any child is then automatically formatted.

For this the base UI elements are created in the editor and saved as prefabs. If advanced functionality is required a MonoBehaviour component is added. Once this preparation is done for several UI elements, one can instantiate these via script and access their components to customize them. This method has proven to be effective in terms of easy expansibility.

Performance

After initial performance profiling, a significant amount (>50%) of the frame time was spent raycasting the UI elements. This affected frame rates significantly leading to jitter. To reduce this a straight ray is used for the UI, as curved rays are implemented by using several straight raycasts. Additionally when the UI canvas is not being hovered over by the ray, the UI graphics raycaster is disabled. This works by the assumption that all interactable UI elements are contained inside the canvas, which is not a strict requirement with a Unity world-space canvas.

As the number of UI elements increases in the future, there will most likely need to be further UI performance optimizations. For example, occlusion culling for raycasting not just rendering. It is unclear whether this is done by default. The newer xml/css based Unity UI Toolkit [2.13] will likely solve many of these issues once it becomes a verified package with runtime support.

Tooltips and Input Hints

To make the application more intuitive and user friendly, we need a way of providing the user with relevant help information when required. The intent being that a user learns how to use a feature when they need it, colloquially just-in-time learning. This requires inferring of the context, similarly to the input context. To solve this tooltips are provided to display a short text when hovering over a UI element.

Input hints tell the user what each button/axis does and are displayed over the controller based on the input context, see gallery.

Alternatives

In order to rely less on UI, other input methods are also possible. Speech recognition is an example which was attempted with the KeywordRecognizer [2.14]. This is however still too unreliable and unresponsive, often with a delay of more than one second. However if improved, speech could be used effectively for certain actions.

Controller gestures and pie menus also present potentially fast methods of interacting by making use of the positional input. Using pie menus for numerical input with the joysticks may also be worth exploring in the future.

High Level Actions

Vertex Selection

Vertex selections are used for affecting only parts of the mesh or as an input to a libigl function. A key feature is being able to transform selections with the controllers, as well as being able to transform two selections with each hand independently. This requires that we have multiple selections simultaneously. To solve this efficiently a bitmask is used. Each vertex has an additional 32-bits (represented as an integer), with each bit indicating whether it is in the selection or not. This allows for up to 32 selections, which is reasonable for this use case.

An additional benefit of using bitmasks is that we can provide a mask of selections with one integer. For example, we can choose which selections are visible or will be translated with an integer. If we want to affect all selections we simply use the maximum integer value, where all bits are one. Functions that act on a selection have been modified, if possible, to act on a mask of selections.

As Eigen does not currently have cwise bitwise operations, unary functions were used. These might not be as well optimized. However, when testing on the armadillo mesh [2.6] the interactions was still responsive enough.

Face or edge selection was not implemented as this is more involved and does not necessarily add more features for the current use case.

Transformations

In order to transform a mesh or part of the mesh there are two stages: finding which transformation should be done and how to apply it. For determining an affine transformation - translation, rotation, scaling - we are much more flexible in VR, as we have two controllers. Some inspiration was taken from Quill [1.4] for how the transformations behave.

Once the transformation is known we can either apply it to the mesh directly, which is done in C# with the Unity API. Or we can apply it to a vertex selection mask, which is done in C++ and modifies the vertex data. This implementation is more involved as transforming a selection mask needs to be done on the worker thread. It uses the Eigen geometry module [2.15].

When working with multiple meshes or multiple selections, we need to determine what to transform – a mask of meshes or selections. For this the sphere brush is used. If a mesh or selection is inside, it is affected. If there is nothing inside then the active mesh or selection is affected. This provides lots of control but also gives an intuitive experience. If both hands act on the same mask then we perform two handed transformations, such as scaling. This method provides and simple way for operating on multiple selections.

To provide more fine grained control, the amount by which the grip buttons are pressed is used as a smoothing factor. This works well, although it can be hard to control this smoothing factor precisely. It may make sense to apply a log or square root to the smoothing factor to counteract this.

Different pivot modes where tested: mesh center, selection center and hand center. Using the hand as the center appeared the most natural. For transforming selections, using the mesh as the center usually gave unintuitive results, particularly for smaller selections, see gallery.

Deformations

The libigl biharmonic deformation `igl::harmonic` can be toggled on. If enabled it will be run whenever the input arguments have been changed. In this case, when the boundary conditions have changed. This can be detected quite easily by checking the *`DirtyState`* of the mesh data have been modified when applying the mesh data in *`ApplyDirty()`*. The As-Rigid-As-Possible `igl::arap` deformation works very similarly, except that we need to check when the precomputation needs to be done. For details as well as diagrams see the documentation [2.16].

5.2.3 Documentation Process

There are several parts to the documentation process, all of which need to be equally addressed. It is important to make a distinction for how to:

1. Use the end product
2. Start development and understand the overall process
3. Use the existing code/API
4. Understand the development approach and an evaluation of the project (this report)

An important part is also that the documentation should be inlined as much as possible, so that it is made part of the source. This means it can be easily found when developing and that it is more easily maintained.

Most functions and types have an annotated docstring, in C# a xml-doc and javadoc in C++. This provides information on how to *use* the function/type. In the implementation, there are comments as required for how to *modify* the function/type. As in C# everything resides inside a class/struct/interface the docstring of the class is intended to give an overview of everything inside and its intention.

Additional markdown files are there to add an overview and provide general information not specific to a file or piece of code. These files are placed 'inline' next to the `.cs` or `.cpp` files. Within these, diagrams.net is used for flowcharts.

To compile all this information, Doxygen [2.17] and Sphinx [2.18] are used. Doxygen is used to extract the documentation from the code. This information in xml format is then used by Breathe (a Sphinx extension) to render it with Sphinx, which then combines it with the markdown files. Breathe [2.19] and the language domains ensure cross-referencing of items. For this to work with C#, the sphinx-csharp [2.20] and breathe projects where modified, see #8 and #550 respectively on GitHub.

ReadTheDocs [2.21] is used to host and compile the website output of Sphinx. This has continuous integration. Whenever a commit is pushed to the `read-the-docs` branch, the website is recompiled.

5.2.4 Miscellaneous Features

1. Importing of meshes into Unity, adjusting scaling, vertex buffer layout and materials
    1. Recognized file types use an asset post-processor
    2. Custom file types, e.g. `.off`, are imported via libigl within a scripted importer
    3. Meshes are validation before instantiation
2. UI to indicate when a thread is inside `Execute` for a longer time
3. Rendering with the Universal Render Pipeline (URP)
4. Environment modelled in Blender, ocean shader created using Shader Graph
5. Speech recognition for specific actions (disabled by default)
6. Cross-platform controller models

7. Different modes for editing selections: add, remove, invert, new/clear selection per stroke
8. Conversion of selection mask to vertex color

For more features see the documentation and repository.

5.3 Discussion & Future Work

Discrete orthogonal geodesic nets (DOG) [4.1] as another use case were briefly explored. However, the interface is more complicated and the project is currently not compatible with Windows without modifications to the code.

Motion sickness experienced in VR has been minimal as the developer is generally experiencing little movement. However, motion sickness for newer users could be further improved if needed. This can be done by reducing the field of view during fast movements with a vignette effect [4.2]. This already has a Unity implementation with the VR Tunneling Pro asset [4.3], so will require minimal work to add.

5.3.1 Synergy of VR and 2D Editors

VR editors can be powerful for viewing and manipulating 3D meshes. However, a VR editor should work in tandem with a 2D editor. Depending on the task, it may be easier to perform it in a 2D or VR scenario. Ideally, a developer should be able to easily switch between the 2D and VR editor as desired. This is not possible with Unity without reimplementing the 2D viewer in Unity.

Another takeaway from this project is that the iteration time when testing in VR is higher than with a 2D viewer. This is simply because a developer has to physically put on and off the VR headset and controllers repeatedly. If possible, testing new changes in VR is avoided.

5.3.2 Alternatives to Unity

A problem is that Unity is a game engine not a mesh editing software. Whilst it can provide a good interface to VR, there is a limit to how useful it can be, in terms of mesh editing, for libigl before common functionality has to be manually implemented, e.g. vertex selection. Finding workarounds for Unity incompatibilities was, in the end, a large focus of this thesis, see Backend.

Unreal Engine is an alternative to Unity. Its benefits lie in using C++ as well as having a lower-level open source mesh API exposing greater control. However, Unreal Engine is also a game engine. A more promising alternative is Blender. There are several interesting aspects to this:

- Open source
- Intended for editing meshes, common functionality is already available
- C++ can be used directly, otherwise the python interface from libigl can be used
- Cross-platform VR is in active development, although not production ready
- VR viewer is built on top of 2D viewer, enabling easy switching
- Blender is closer to a real world application of libigl

Both these alternatives will have a steeper learning curve, but will be better solutions in the end.

5.3.3 Use Cases & Shortcomings

This project can be used to visualize libigl functionality in VR. However, there is still a reasonable amount of work required to integrate new behavior. In most cases, this will not be worth the effort.

The most immediate value is in the independent components. Runtime mesh modifications can be done more easily in Unity, as the mesh interface between C++ Eigen matrices and Unity is now working. This is regardless whether in VR, with libigl or not. Developments made with the UnityNativeTool allow for easier C++ development within Unity for any project. This project can also serve as an example for C#/C++ development in Unity, as this is not common in open source. The methods used for UI can be used in other VR projects. Other projects using C# and C++ can use the same documentation generation process.

The major shortcoming of this project has already been introduced and lies in the choice of Unity as a basis. This is ultimately limits its future potential. Otherwise in comparison to the related work, it offers very little functionality, with only two types of operations to perform on a mesh. There is currently no way of exporting the meshes, although, this could be done in the future with the existing FBX, USD and Alembic exporters in Unity.

5.4 Conclusion

A documented and working VR editor for libigl has been produced with plans for expansibility for adding more libigl features. It provides multi-mesh support as well as standard ways of editing a mesh with multiple vertex selections. This enables the interactive use of biharmonic and As-Rigid-As-Possible deformations. By providing visual aids via tooltips and input hints, user-friendliness is improved.

The current 2D libigl editor still has a much larger feature set and there is a large barrier for converting applications to the VR viewer. As a result, the current application appears to be more suitable for demos. However, there is potential for this to see wider use cases in the future, if developed further.

Using Unity as a basis is not optimal but has several advantages, particularly because of its ease of use and flatter learning curve. Workarounds have been found for the immediate shortcomings with the mesh interface, API thread-safety limitation and C++ interface. The discussed alternatives should be considered first before continuing this project.

- See whole genindex

CHAPTER

SIX

FEATURES

See the Gallery for visual examples.

1. Run As-Rigid-As-Possible `igl::arap` or a biharmonic defomation `igl::harmonic` on a mesh and manipulate it in real-time with the VR controllers
2. Select vertices and transform them using VR controllers
3. Threaded geometry code, can handle armadillo with responsive VR
4. Multiple selections per mesh (using bitmasks)
5. Multi-mesh editing possible, easily swap out the mesh
6. Easy import process of new models
7. Easy UI generation (using prefabs and C#)

Compatible VR Headsets: Theoretically everything compatible with the Unity XR Plugin system and XR Interaction Toolkit. Tested on Oculus Rift S, likely to be compatible are Oculus Rift, Oculus Quest, Valve Index.

CHAPTER

SEVEN

TECHNICAL FEATURES

1. Unity/libigl interface for meshes
2. Unity C#/C++ interface
3. Handling of input with threaded geometry calls

CHAPTER

EIGHT

DEVELOPMENT TIMELINE

BIBLIOGRAPHY

[1.1] Roger Barton. 3D Modeling in Virtual Reality Documentation. https://vr-modeling.readthedocs.io/.

[1.2] Alec Jacobson, Daniele Panozzo, and others. libigl: a simple C++ geometry processing library. 2018. https://libigl.github.io/.

[1.3] Google. Google blocks. https://arvr.google.com/blocks/.

[1.4] Facebook Technologies. Quill. https://quill.fb.com/.

[1.5] Oculus. Oculus medium. https://www.oculus.com/medium/.

[1.6] David Farrell. Medium under the hood: part 1 - developing the move tool (oculus developer blog). https://developer.oculus.com/blog/medium-under-the-hood-part-1-developing-the-move-tool.

[1.7] Blender Foundation. Blender. https://www.blender.org/.

[1.8] Blender Foundation. Blender VR scene inspection. https://docs.blender.org/manual/en/dev/addons/3d_view/vr_scene_inspection.

[1.9] Blender Foundation. Blender VR scene inspection (release notes). https://wiki.blender.org/wiki/Reference/Release_Notes/2.83/Vi

[1.10] Epic Games. Unreal engine. https://www.unrealengine.com/.

[1.11] Epic Games. Unreal engine VR mode documentation. https://docs.unrealengine.com/en-US/Engine/Editor/VR/index.html.

[1.12] Unity Technologies. Unity. https://unity.com/.

[1.13] O. Sorkine, D. Cohen-Or, Y. Lipman, M. Alexa, C. Rössl, and H.-P. Seidel. Laplacian surface editing. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, SGP '04, 175–184. New York, NY, USA, 2004. Association for Computing Machinery. URL: https://doi.org/10.1145/1057432.1057456, doi:10.1145/1057432.1057456.

[1.14] Olga Sorkine and Marc Alexa. As-rigid-as-possible surface modeling. In *Proceedings of the Fifth Eurographics Symposium on Geometry Processing*, SGP '07, 109–116. Goslar, DEU, 2007. Eurographics Association.

[1.4] Facebook Technologies. Quill. https://quill.fb.com/.

[1.7] Blender Foundation. Blender. https://www.blender.org/.

[2.3] mcpiroman. Unity native tool. https://github.com/mcpiroman/UnityNativeTool.

[2.4] the Regents of the University of California The University of Utah and contributors. SWIG: simplified wrapper and interface generator. http://www.swig.org/, https://github.com/swig/swig.

[2.5] David Beazley. Automated scientific software scripting with swig. *Future Generation Comp. Syst.*, 19:599–609, 07 2003. doi:10.1016/S0167-739X(02)00171-1.

[2.6] Stanford University Computer Graphics Laboratory. The stanford 3D scanning repository. http://graphics.stanford.edu/data/3Dscanrep/.

[2.7] Gaël Guennebaud, Benoît Jacob, and others. Eigen v3. http://eigen.tuxfamily.org, 2010.

[2.8] Unity Technologies. NativeRenderingPlugin: native code (C++) rendering plugin example for unity. https://github.com/Unity-Technologies/NativeRenderingPlugin.

[2.9] Oculus. Oculus unity integration. https://assetstore.unity.com/packages/tools/integration/oculus-integration-82022.

[2.10] Unity Technologies. Unity XR plugin framework. https://docs.unity3d.com/Manual/XRPluginArchitecture.html).

[2.11] Unity Technologies. XR interaction toolkit (unity package). https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@

[2.12] Roger Barton. 3D Modeling in VR Documentation: Customizing Input. https://vr-modeling.readthedocs.io/docs/developer-guide/adding-functionality.html#customizing-input.

[2.13] Unity Technologies. UI toolkit (unity package). https://docs.unity3d.com/2020.1/Documentation/Manual/UIElements.html.

[2.14] Unity Technologies. KeywordRecognizer UnityEngine.Windows.Speech. https://docs.unity3d.com/ScriptReference/Windows.Speech.KeywordRecognizer.html.

[2.15] Gaël Guennebaud, Benoît Jacob, and others. Eigen geometry module. https://eigen.tuxfamily.org/dox/group__Geometry__Module.html.

[2.16] Roger Barton. 3D Modeling in VR Documentation: Custom Deformation. https://vr-modeling.readthedocs.io/docs/developer-guide/adding-functionality.html#custom-deformation.

[2.17] Dimitri van Heesch. Doxygen. https://www.doxygen.nl.

[2.18] Georg Brandl and others. Sphinx: python documentation generator. https://www.sphinx-doc.org/.

[2.19] Michael Jones and others. Breathe: ReStructuredText and sphinx bridge to doxygen. https://github.com/michaeljones/breathe.

[2.20] djungelorm and others. Sphinx-csharp: c# domain for sphinx. https://github.com/djungelorm/sphinx-csharp.

[2.21] Inc Read the Docs and contributors. Read the docs. https://readthedocs.org/.

[4.1] Michael Rabinovich. Discrete orthogonal geodesic net (DOG) editor. https://github.com/MichaelRabinovich/DOG-editor/.

[4.2] A. S. Fernandes and S. K. Feiner. Combating VR sickness through subtle dynamic field-of-view modification. In *2016 IEEE Symposium on 3D User Interfaces (3DUI)*, volume, 201–210. 2016. https://ieeexplore.ieee.org/document/7460053/.

[4.3] Sigtrap Games. VR tunnelling pro (unity asset). https://assetstore.unity.com/packages/tools/camera/vr-tunnelling-pro-106782.

INDEX

L

M

Q

R

S

T

U

V