作品集

视频演示：

1. 自动绑定肢体工具**(auto limb tool)**：

Github ：[vinL1234/auto-limb-tool: A modular Autodesk Maya tool for building IK/FK limb rigs, stretch systems, and roll joints.]

简介：

这是一个用Python script来解决在**maya**重复绑定劳动力的而开发的自动化绑定工具。该工具是将重复、复杂的肢体绑定流程标准化，减少手动创建骨骼、控制器和约束关系所需的时间。

工具支持根据用户设置快速生成初始人体肢体绑定结构，自动完成骨骼层级、控制器、IK/FK 系统及相关连接的搭建，并通过模块化代码组织提高工具的可维护性与扩展性。

设计：

首先，在了解了且亲自制作绑定人体角色的大概与整体流程后，思考 哪些 变量是得使用者输入 而不能 通过代码获取 比如 简单的人体绑定变量可以让用户输入关节的orientation获得(Primary axis, secondary axis, 和world secondary orientation) 。

- 胳膊和腿的绑定有些区别所以分开设定让使用者自由设定。因为每个角色的绑定orientation都有不同根据 引擎或者项目需求等等，所以orientation得让使用者自行输入设置。
- 因为这是基础的初始四肢设定所以根据三个joints来为基础的框架
- 考虑到会有复杂的绑定需求三个joints的基础框架不足以解决复杂的绑定需求 从而设置了自定义页面

- IK/FK 一般都是在一条骨骼上都是有开始到结束所以首先使用者先获取root joint 然后再进行选择joints从而防止 尽可能避免出现错误。

  并且提供选项可以添加是IK还是FK还是两个都要

- 设置 orientation，这样子控制器，限制，和节点设置才能得到正确的操作。

- 伸缩系统通常是根据距离变化来变化scale从而获取拉伸。所以得有一个目标点来获取joint所需要的拉伸。

- Roll joints 通常是基于更好模拟出骨骼带动肌肉变化来设定的helper joints。所以这里添加了不同的roll type供用户选择然后以后也可以自由添加新的类型来扩展（比如肩膀的三角肌和肱二头肌的运动规律，肱二头肌的拉伸鼓起旋转）

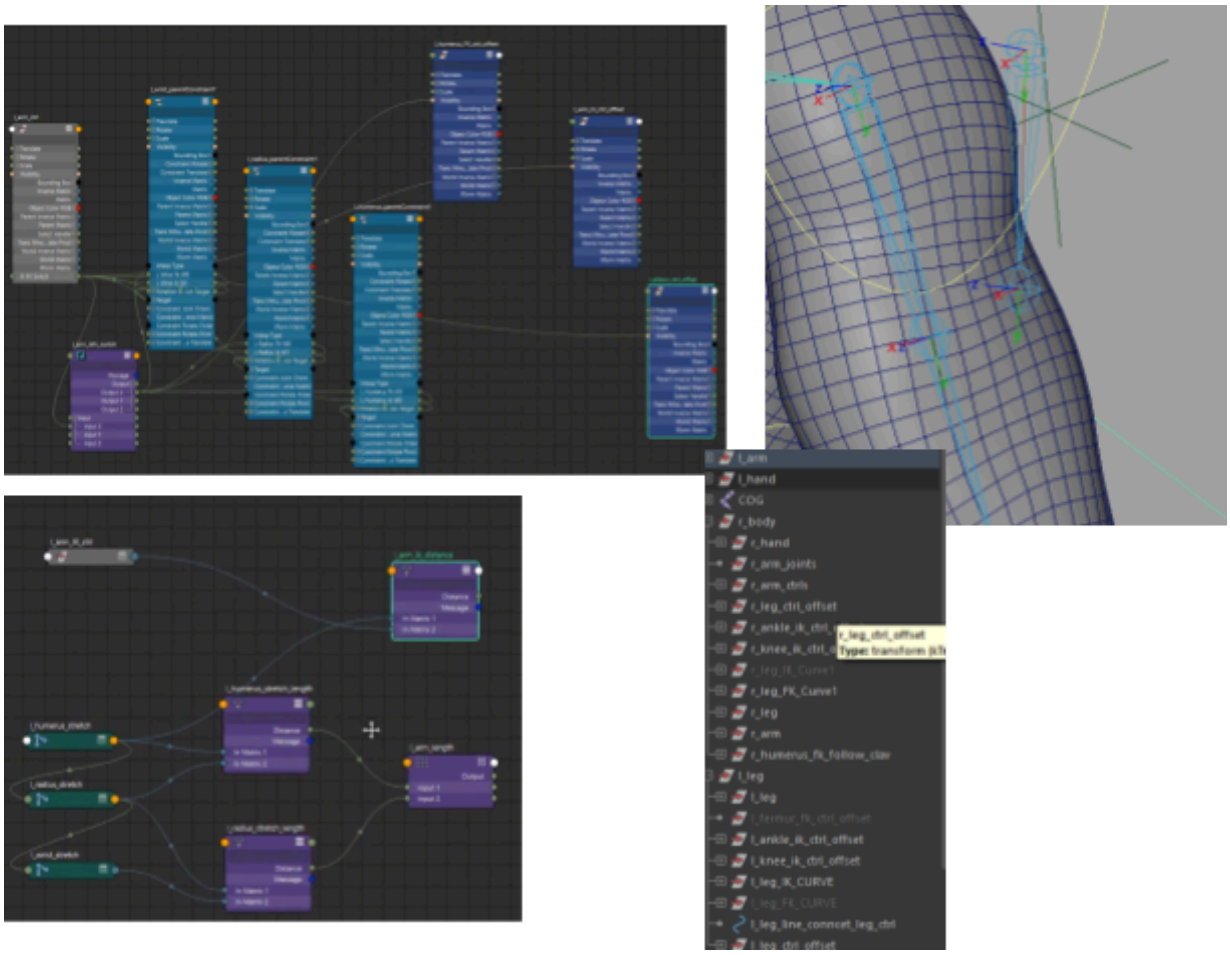
解决的问题：
（例子）

即便是基础的简单的人体肢体绑定 也会消耗大量的重复无用的时间来进行操作。比如，IKFK设置，需要复制 joints chain， 控制器的设定限制进行约束，节点设置，命名，父子关系层级关系，根据ik/fk 开关的操作，等等。

辅助joints的constraint复杂设置（例如，aim constraint限制三角肌的joint正常旋转根据locator可以把joint的旋转锁住），还有伸缩的复杂constraint设定（roll joint也得根据伸缩来进行变形）。

AI辅助编程，根据我设计好的框架来进行重复性的搭建，比如重复的按钮设置和一些重复的逻辑上的实现，还可以协助我解决一些重复code的refactor，甚至协助我完成一些复杂的逻辑实现，检查逻辑，获取信息，完善逻辑疏漏，等等。所以大幅度降低时间成本。

- 例子：复杂重复的绑定操作：



UI 代码设计：

User friendly UI: 根据用户的需求来 尽量简洁的设置 来完成完整的内容。之前已经解释过了。

- 代码实现细节（页面跳转）：

  首先，只有一个main window（主要窗口）：

```
def show(self):
    if cmds.window(WINDOW_NAME, exists=True):
        cmds.deleteUI(WINDOW_NAME)

    cmds.window(
        WINDOW_NAME,
        title="Auto Limb Builder",
        width=430, height=500
    )

    self.main_layout = cmds.columnLayout(
        adjustableColumn=True,
        rowSpacing=10
    )

    self.show_main_page()
```

  - 整体只有一个main window（主要的窗口）

- 主页面设定

```
def show_main_page(self):
    self.clear_ui()

    cmds.text(label="Limb Builder", height=30, align="center")
    cmds.separator(height=8, style="in")

    cmds.frameLayout(
        label="Standard Limb",
        collapsable=False,
        marginWidth=10,
        marginHeight=10
    )

    cmds.columnLayout(adjustableColumn=True, rowSpacing=8)
```

- 启动mainpage后，先清理所有ui elements. 因为我们是在main page 和 custom page 来回跳转。

```
def clear_ui(self):
    children = cmds.columnLayout(
        self.main_layout,
        query=True,
        childArray=True
    ) or []

    for child in children:
        if cmds.control(child, exists=True) or cmds.layout(child, exists=True):
            cmds.deleteUI(child)

    cmds.setParent(self.main_layout)
```

- 清除掉所有的ui设置在现在的页面里

```
cmds.button(
    label="Go To Custom Page",
    height=45,
    command=lambda *args: self.show_custom_page()
)
```

- 在main page里的这个按钮会让我们跳到custom page

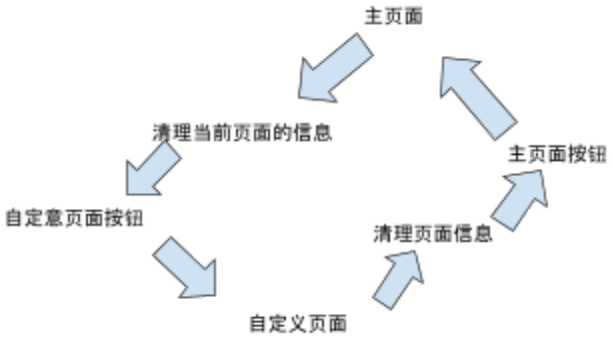
```
def show_custom_page(self):
    self.clear_ui()

    self.custom_root_joint = None
    self.custom_chain = []
```

- Custom page先清除所有ui

```
cmds.button(
    label="Back",
    height=35,
    command=lambda *args: self.show_main_page()
)
```

- 用back按钮再次返回主页面



- 以后能更方便的添加新的页面功能以后

Roll joint 整体实现过程：

- 实现的整体逻辑 是让三角肌的joint不会自我旋转模拟真实大臂肌肉，可以让它朝向肘部用aim constraint 使他的 object up vector 朝向我们设置的locator那里，从而实现单方面的锁住三角肌的joint的primary aixes。然而胳膊向里移动的时候因为locator没有移动，所以一直指向locator的轴向会翻转，因此设计了follow joints让locator在它的子级里跟随移动。follow joints的移动规律是根据放在肘部的ik handle 来进行跟随

```
# Upper segment remains a root + midpoint chain.
first_segment = self.create_simple_roll_segment(
    bind_chain[0],
    bind_chain[1],
    first_name
)

# Lower segment has two sibling roll joints:
# one at the midpoint and one at the wrist/end joint.
second_segment = self.create_forearm_roll_joints(
    bind_chain[1],
    bind_chain[2]
)
```

- 先创建两组roll joints 分别布置到在大臂和小臂从而影响权重

```
def create_simple_roll_segment(
    self,
    start_joint,
    end_joint,
    segment_name
):
    roll_root = self.duplicate_joint_only(
        start_joint,
        segment_name + "_roll_root_jnt"
    )

    temp_constraint = cmds.parentConstraint(
        start_joint,
        roll_root,
        maintainOffset=False
    )[0]
    cmds.delete(temp_constraint)
```

- 复制roll joint的起始，然后用parent constraint的方法对齐，达到对齐的目的后删除constraint（orientation也可以一致）

```
start_position = cmds.xform(
    start_joint,
    query=True,
    worldSpace=True,
    translation=True
)

end_position = cmds.xform(
    end_joint,
    query=True,
    worldSpace=True,
    translation=True
)

midpoint = [
    (start_position[0] + end_position[0]) * 0.5,
    (start_position[1] + end_position[1]) * 0.5,
    (start_position[2] + end_position[2]) * 0.5
]
```

- 根据起点和中点的位置获得中间距离，然后根据中间的距离放置第二个roll joint（如果是大臂的话就是肱二头肌那里）

```
# Move in the opposite direction of the selected secondary axis.
cmds.move(
    -secondary_vector[0] * distance,
    -secondary_vector[1] * distance,
    -secondary_vector[2] * distance,
    locator,
    relative=True,
    objectSpace=True
)

return locator
```

- 设置locator在根部roll joint相反的secondary aixes上移动，这里只要是不是primary aixes就行 主要是为了锁住roll joint的旋转

```
# Upper roll root aims toward elbow/knee.
cmds.aimConstraint(
    bind_chain[1],
    first_roll_root,
    maintainOffset=False,
    aimVector=aim_vector,
    upVector=up_vector,
    worldUpType="object",
    worldUpObject=aim_locators[0]
)
```

- 设置aim constraint来实现锁住roll joint的主轴的旋转

```
def create_upper_roll_follow_chain(
    self,
    first_roll_segment,
    upper_aim_locator,
    target_joint,
    secondary_axis,
    distance=2.0
):
```

- 创建follow joint

```
cmds.move(
    -secondary_vector[0] * distance,
    -secondary_vector[1] * distance,
    -secondary_vector[2] * distance,
    follow_root,
    relative=True,
    objectSpace=True
)
```

- 移动 follow joint相同的方向到locator和roll joint之间 这样子可以幅度不会太大，不容易导致变形

```
follow_ikh = cmds.ikHandle(
    name=root_name.replace("_jnt", "") + "_follow_ikh",
    startJoint=follow_root,
    endEffector=follow_mid,
    solver="ikRPsolver"
)[0]

# Zero the IK Handle pole vector values.
for axis in "XYZ":
    pole_attr = follow_ikh + ".poleVector" + axis

    if cmds.objExists(pole_attr):
        cmds.setAttr(
            pole_attr,
            0
        )
```

- 根据follow joint创建ik handle 随后把pole vector调整成0这样ikhandle就不会跟随elbow而内旋了 但是会跟着肘部移动

- 小臂的解决方法和大臂差不多 这里就不过多解释了

技术问题(代码细节)：

- 在做ikfk 开关的时候要让权重操控ik或者fk，但是从每个constraint上获取权重名称。
  因为没有固定权重名称 wo或者w1，而且ik 不一定对应着就是woik，有可能joint写反了，所以进行专门的查询是很有必要的

```
for i, constraint in enumerate(constraints):
    aliases = cmds.parentConstraint(
        constraint,
        query=True,
        weightAliasList=True
    ) or []

    targets = cmds.parentConstraint(
        constraint,
        query=True,
        targetList=True
    ) or []
```

- 通过weightAliasList获得对应的weight attribute的名称
- 通过targetList获得对应的parent constraint的驱动名称

```
for target, alias in zip(targets, aliases):
    target_short = self.short_name(target)

    if target_short == self.short_name(ik_chain[i]):
        cmds.connectAttr(
            switch_ctrl + ".ikFk",
            constraint + "." + alias,
            force=True
        )

    elif target_short == self.short_name(fk_chain[i]):
        cmds.connectAttr(
            reverse_node + ".outputX",
            constraint + "." + alias,
            force=True
        )
```

- 然后检查是不是同一个ik 或者fk joint对应着的 weight value。
- Reverse node是因为ik weight和fk的weight得相反。这样子一个开了另外一个就关了

- 在很多情况下内部存储的是dag的完整路线但是给user显示的是简短的名称

```
def short_name(self, obj):
    return obj.split("|")[-1]
```

- Short name 返回给dag的最后面的名字

```
for menu in [self.custom_start_menu, self.custom_end_menu]:
    items = cmds.optionMenu(menu, query=True, itemListLong=True) or []
    for item in items:
        cmds.deleteUI(item)

for jnt in self.custom_chain:
    cmds.menuItem(label=self.short_name(jnt), parent=self.custom_start_menu)
```

- 拿到短名字显示给user

```
def load_custom_root_chain(self, *args):
    root = self.selected_joint()
    if not root:
        return

    self.custom_root_joint = root
    self.custom_chain = self.get_first_child_chain(root)
```

- Custom_chain一般都是以完整的dag path存储的

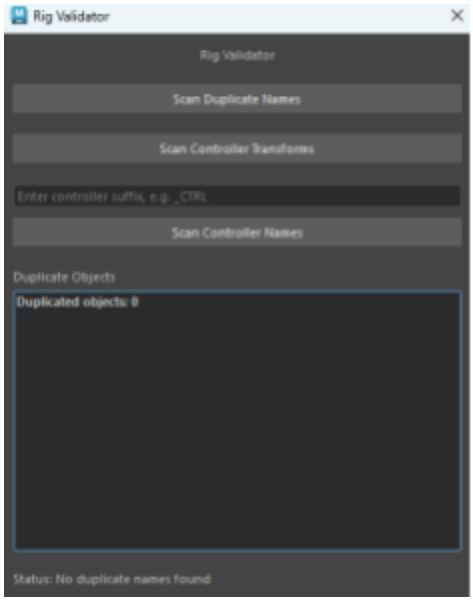
Rig validator（绑定检查器）：

Github: [vinL1234/maya-rig-validator: A PySide6 rig validation tool for Autodesk Maya.]

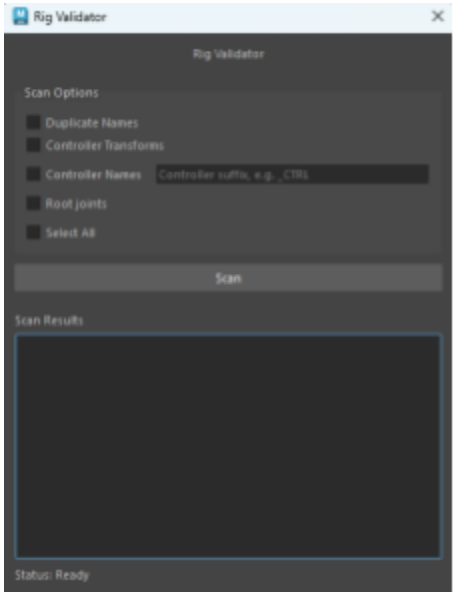
基于 pyside python开发的 Maya Rig 自动检查工具，支持 **Rig** 批量扫描、问题定位、批量修复、**Scene** 变化监听以及可扩展的 Validator 框架设计。

该工具用于解决复杂 Rig 中节点数量增加后，人工逐项检查效率低、容易遗漏且重复操作较多的问题，从而提高 Rig QA 效率

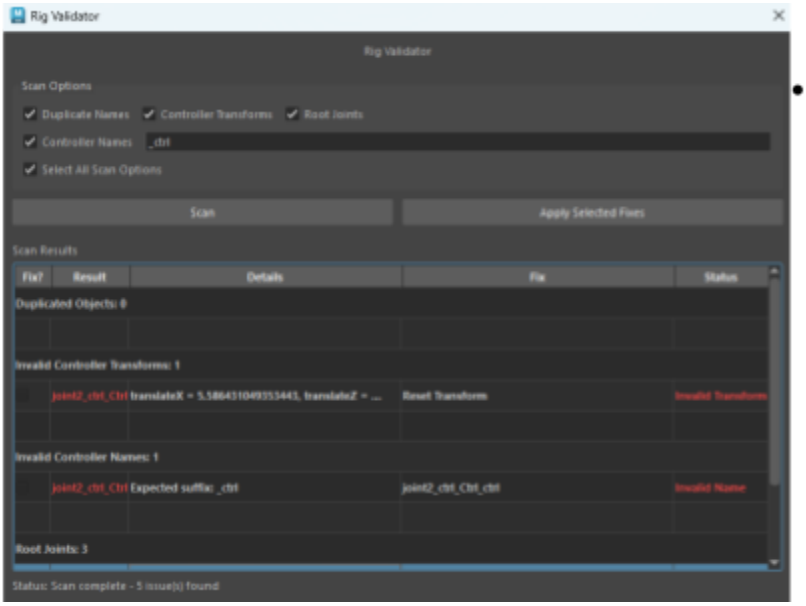
界面的迭代：



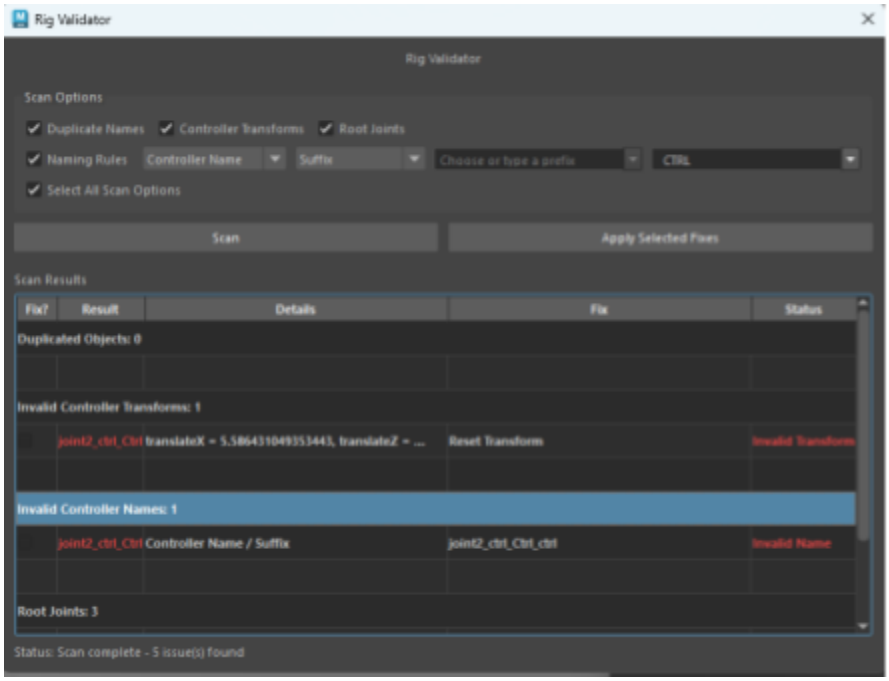
- 最开始的设计通过不同的按钮来实现功能
- 发现后续如果检查的功能越来越多按钮会很麻烦



- 二次修改 目的是可以让使用者有更多的选择，选择一项或者多项或者全部来进行扫描



- 继续进行迭代，更改后，用户可以根据table的提示和修改方式来修复错误
或者双击错误来自行更改初始的修复



- 最终修改，可以根据前缀或者后缀或者都选 来检查合规的名字
- 还可以选择检查mesh或者joint 的名称

整体的代码实现逻辑框架：

Maya主窗口 **—> UI**界面 **—>** 扫描调度层 **—>** 验证与修复层 **—>** 场景监听层

Pyside 实现逻辑：

通过get_maya_main_window()初始化窗口

```
maya_main_window = get_maya_main_window()
```

创建各种widget类

```
def create_widgets(self):
    """Create all widgets."""
    self.title_label = QtWidgets.QLabel("Rig Validator")
    self.title_label.setAlignment(QtCore.Qt.AlignCenter)
```

添加widget类

```
def create_layouts(self):
    """Arrange widgets."""
    main_layout = QtWidgets.QVBoxLayout(self)
    main_layout.addWidget(self.title_label)
```

连接实现各种功能的函数

```
def create_connections(self):
    """Connect signals."""
    self.scan_button.clicked.connect(self.scan_selected)
    self.apply_fixes_button.clicked.connect(
        self.apply_selected_fixes
    )
```

- Qt的话是通过slot 和signal来传递信息。signal就是像button那种user发出指令，然后slot就是connect的函数

Qdialog接受的对话操作（比如**button**，等等）来激活**connect**的函数

这里的目的是为了新创建的窗口跟随**maya**的主窗口进行操作，比如，关闭，移动

```
def get_maya_main_window():
    """Return Maya's main window as a Qt widget."""
    main_window_ptr = omui.MQtUtil.mainWindow()
    if main_window_ptr is None:
        return None
    return wrapInstance(int(main_window_ptr), QtWidgets.QWidget)


class RigValidatorWindow(QtWidgets.QDialog):

    def __init__(self, parent=get_maya_main_window()):
        super().__init__(parent)
```

- 通过omui.MQtUtil.mainWindow()返回个指向主window的指针，因为window是c++，我们得用wrapInstance来转换c++成为一个可以用python来设置的QT对象

技术细节：

```python
def add_result_row(
    self,
    result_text,
    details="",
    fix_text="",
    status="",
    node_name=None,
    fix_type=None,
    checkable=True,
    fix_editable=False,
    bold=False,
    color=None,
    tooltip=None,
):
    """Add one validation result row."""
    row = self.result_table.rowCount()
    self.result_table.insertRow(row)

    check_item = QtWidgets.QTableWidgetItem()
    result_item = QtWidgets.QTableWidgetItem(result_text)
    details_item = QtWidgets.QTableWidgetItem(details)
    fix_item = QtWidgets.QTableWidgetItem(fix_text)
    status_item = QtWidgets.QTableWidgetItem(status)

    result_item.setData(
        QtCore.Qt.UserRole,
        {
            "node": node_name,
            "fix_type": fix_type,
        },
    )
```

- 通过add_result_row()来添加不同search给的结果到table里，根据 传递不同的信息做每一排的规划

发现的问题和解决方法：

假如 我们保存了两个节点：
{
|char|ctrls|
|char|ctrls|l_arm_ctrl
}

→

如果先修改了第一个节点
ctrls->|char|CTRLS|
第二个节点要修改的时候
路线不对了

修改方法

选择修复list 然后根据 | 来找到最深的依次修复

```python
# Rename children before parents because parent renaming changes paths.
selected_fixes.sort(
    key=lambda data: (
        data["node"].count("|") if data["node"] else 0 #count the number of '|' in the node path to determine depth
    ),
    reverse=True,
)
```

- 这里用了AI因为我之前的代码太厚重了，然后AI给我改了 我检查了逻辑

还有一种问题 用户先扫描了整个界面，然后用户修改了某个名字，路线变了，如果这时候再次在工具里进行改名会搜索失败

大概的解决方案就是 建造个监听模式 监听所有的能更改的节点 一旦用户对名称进行更改 就锁住整个**table** 然后 用刷新按钮重新查询并获得节点完整的路径

实现过程：

```
self.results_are_stale = False
self.ignore_scene_changes = False
self.scene_callback_ids = []
self.watched_node_handles = set()
```

- Results_are_stale 判断现在的结果是否过期 来决定能运行apply_selected_fixes()
- Ignore_scene_changes 因为场景在调整的时候不需要触发提示，等修改完后再触发提示
- Scene_callback_ids 把所有callback的id保存起来
- Watched_node_handles 储存所有已经监听的节点

```
self.results_are_stale = False
self.refresh_button.setEnabled(False)
self.apply_fixes_button.setEnabled(True)
self.register_result_name_callbacks()
```

- scan_selected() 里面在最后用 register_result_name_callbacks() 去把查询把要修改的节点进行保存

```
row_data = result_item.data(QtCore.Qt.UserRole)
if not row_data:
    continue

node_name = row_data.get("node")
```

- 在register_result_name_callbacks()里用userRole把之前在里面存好的路径找到

```
for row in range(self.result_table.rowCount()):
    result_item = self.result_table.item(row, RESULT_COLUMN)
```

- 通过遍历整个table找到所有的需要监听的节点

```
parts = [part for part in node_name.split("|") if part]
```

```
current_path = ""
for part in parts:
    current_path += "|" + part
```

- 把路径的所有节点全部拆开，因为为了重新组成 所有parent 的路径，因为为了防止parent的节点被改 子节点无法成功查询 然后把所有路径都加入监听对象

```
selection = om.MSelectionList()
selection.add(node_path)
node_object = selection.getDependNode(0)
```

```
name_callback_id = om.MNodeMessage.addNameChangedCallback(
    node_object,
    self.on_node_renamed,
)
```

- 把str的路径改成openmaya能用的对象的路径 利用openmaya 来添加callback进行监听，这里就是一旦监听改名到就调用on_node_renamed

```
destroyed_callback_id = (
    om.MNodeMessage.addNodeDestroyedCallback(
        node_object,
        self.on_node_destroyed,
    )
)
```

- 当监听到节点被删除直接调用on_node_destroyed

```
def on_node_renamed(self, node, previous_name, client_data=None):
    """Handle a watched node or ancestor being renamed."""
    self.mark_results_stale("A watched node was renamed")

def on_node_destroyed(self, client_data=None):
    """Handle a watched result node being deleted."""
    self.mark_results_stale("A watched node was deleted")
```

- Rename 和 destroy都会叫 mark_results_stale()

```
self.results_are_stale = True
self.refresh_button.setEnabled(True)
self.apply_fixes_button.setEnabled(False)
self.status_label.setText(
    f"Status: {reason}. Please refresh the scan results."
)
```

- 在mark_results_stale()里可以调整被监听后的数值禁用 apply_selected_fix()根据 results_are_stale = True 开启刷新按钮，禁止apply按钮

当把一个子节点换到别的节点下面，父节点也是发生了变化，这时候**rename**的监听不行

```
def install_global_scene_callbacks(self):
    """Install callbacks for scene-wide structural changes."""
    self.remove_scene_callbacks()

    callback_specs = (
        (om.MDagMessage.addParentAddedCallback, self.on_parent_changed),
        (om.MDagMessage.addParentRemovedCallback, self.on_parent_changed),
    )
```

- 加了全局监听，任何parent变化都会被监听

当关闭窗口的时候**callback**没有被消除怎么办

```
name_callback_id = om.MNodeMessage.addNameChangedCallback(
    node_object,
    self.on_node_renamed,
)
destroyed_callback_id = (
    om.MNodeMessage.addNodeDestroyedCallback(
        node_object,
        self.on_node_destroyed,
    )
)
```

- 通过保存callback id，存储到 on_node_destroyed 这样子等结束窗口时会自动消除节点

细节问题：

```
handle = om.MObjectHandle(node_object)
handle_hash = handle.hashCode()
```

- 在我们添加监听的时候 要取得节点独特的hashcode，这样子可以去重，因为有可能table里会有相同的node被扫描出来

```
self.ignore_scene_changes = False
```

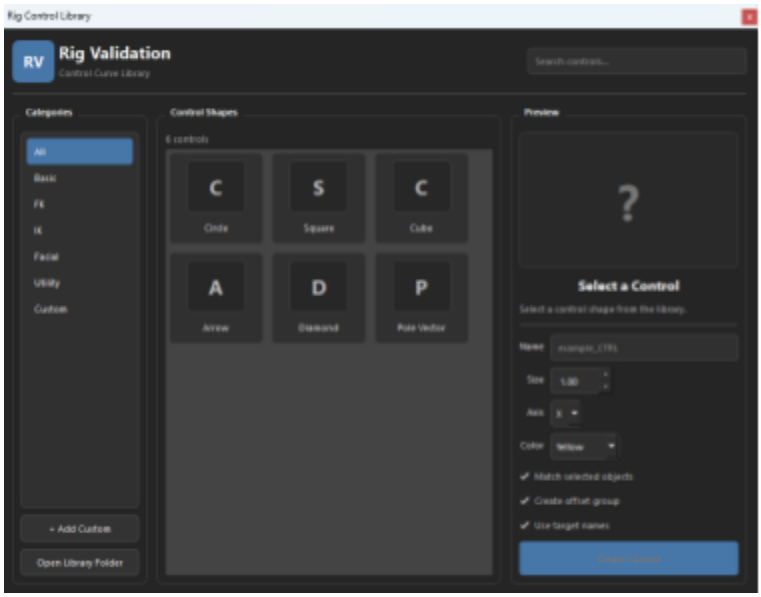
- 之前说过的这个，这个是为了让工具而改变的节点名称等等不会触发监听系统 比如说 用了apply然后名称改变了，如果没有判定的话call back直接进入监听模式，重新刷新

Control Library:

Github: [vinL1234/maya-control-library: A PySide6 control-curve library tool for Autodesk Maya.]

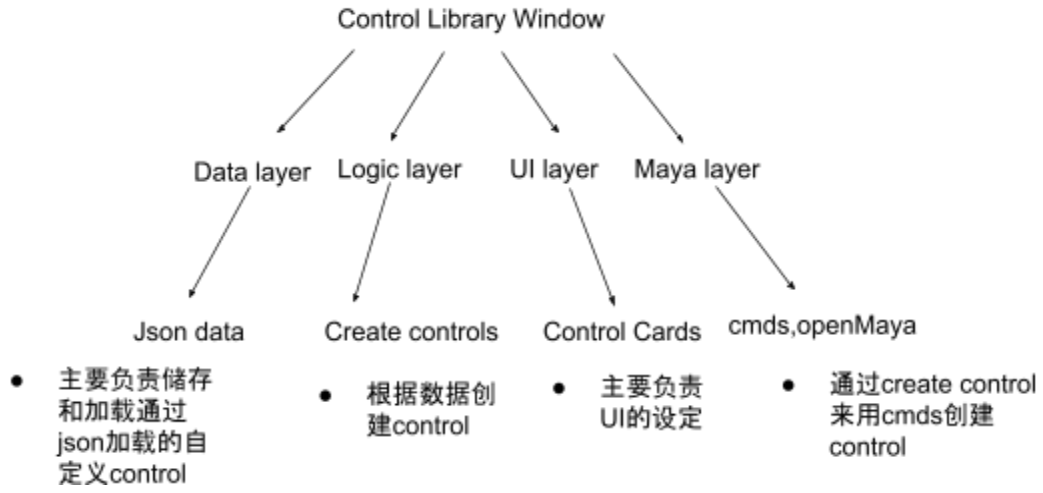
- 这是一款python pyside开发的一款绑定控制器收集和使用的工具
- 它支持内置控制器和用户自定义控制器，并通过 JSON 序列化/反序列化 保存和重建 NURBS Curve 数据，支持多形状 Controller

界面介绍:



- 页面的UI是主要是我来设计AI来完成的
- control shapes底下来选择不同形状的控制形状
- Categories 可以根据不同类型进行分类
- Match selected objects可以根据object的orientation来校准control的orientation
- Offset group 是为了让新创建的control归0
- Use target names 是根据object的名字自动对应相对的命名
- Add custom：是可以选择界面的形状线来加进库里

代码实现结构：



设计的原因是因为不同的层的修改不会干扰不同的层的修改

比如，如果改了UI 但是因为层是分开的 UI的层不会干扰其他层的函数 不用全部修改

数据存储结构(技术难点)：
（技术难点：保存和使用 自定义的控制）

```
BUILT_IN_CONTROL_DATA = [
    {
        "name": "Circle",
        "category": "Basic",
        "description": "A standard circular control for FK joints.",
        "icon": "",
        "custom": False,
        "control_type": "Circle",
    },
```

- 初始的内置的control被储存到字典里跟着脚本一起

```
self.add_custom_button.clicked.connect(
    self.add_custom_control
)
```

```
def get_selected_curve_data():
    """Extract serializable NURBS
    selected_objects = cmds.ls(
        selection=True,
        long=True,
        type="transform",
    ) or []

    if len(selected_objects) != 1:
        raise ValueError(
            "Please select exactly
        )

    control = selected_objects[0]

    shapes = cmds.listRelatives(
        control,
        shapes=True,
        noIntermediate=True,
        fullPath=True,
    ) or []
```

- 当用户按下add_custom按钮，首先要获得选择的curve的dag 完整路线。
- 先获得用户选择的transform，然后再通过transform获得包括shape的完整路线

```
selection_list = om.MSelectionList()
selection_list.add(shape)

dag_path = selection_list.getDagPath(0)
curve_function = om.MFnNurbsCurve(dag_path)

# Store CVs in object space so the shape can be rebuilt anywhere.
points = curve_function.cvPositions(
    om.MSpace.kObject
)
```

- 再把shape的string形式转换成openmaya能操作的对象
- 再根据获得的curve的内部数据结构来获取CV点

```
point_data = [
    [
        point.x,
        point.y,
        point.z,
    ]
    for point in points
]
```

- 再把CV点转换成能储存到Json的普通list

```
knot_data = [
    float(value)
    for value in curve_function.knots()
]
```

- 把获取的knots转换成python float

```
serialized_shapes.append(
    {
        "degree": int(
            curve_function.degree
        ),
        "form": int(
            curve_function.form
        ),
        "points": point_data,
        "knots": knot_data,
    }
)
```

- 再把所有获得的信息组成一个字典通过shape的信息

```
curve_data = (
    get_selected_curve_data()
)
```

```
control_data = {
    "name": display_name,
    "category": "Custom",
    "description": (
        description.strip()
        or "Custom control shape."
    ),
    "icon": "",
    "custom": True,
    "control_type": "Custom",
    "shapes": curve_data["shapes"],
}

self.custom_controls.append(
    control_data
)
```

- 把获取到的信息储存到字典里再储存到我们的control哪里，然后再存放到json里作为本地保存

```
json.dump(
    custom_controls,
    file,
    indent=4,
    ensure_ascii=False,
)
```

创造自定义控制器：

```
def create_custom_control_from_data(
    control_data,
    name,
    size=1.0,
    axis="Z",
):
    """Rebuild a custom controller from previously serialized shape data."""
    shape_data_list = control_data.get(
        "shapes",
        [],
    )
```

- 先收集"所有"shape的data(因为一个transform有可能由多个shape组成

```
for shape_data in shape_data_list:
    degree = int(
        shape_data.get("degree", 1)
    )

    form = int(
        shape_data.get(
            "form",
            int(om.MFnNurbsCurve.kOpen),
        )
    )

    original_points = shape_data.get(
        "points",
        [],
    )

    knots = shape_data.get(
        "knots",
        [],
    )

    if not original_points:
        continue

    scaled_points = [
        (
            point[0] * size,
            point[1] * size,
            point[2] * size,
        )
        for point in original_points
    ]
```

- 每到一个shape就会读取这个shape的degree form points knots信息

```
temporary_curve = cmds.curve(
    **curve_arguments
)
```

- 因为已经把curve的参数放进curve_arguments里了，所以直接解包dictionary，来让curve获得参数

```
if root_control is None:
    root_control = cmds.rename(
        temporary_curve,
        name,
    )
    continue

temporary_shapes = cmds.listRelatives(
    temporary_curve,
    shapes=True,
    noIntermediate=True,
    fullPath=True,
) or []

for temporary_shape in temporary_shapes:
    # Shape-parenting preserves multiple NURBS shapes under one controller.
    cmds.parent(
        temporary_shape,
        root_control,
        shape=True,
        relative=True,
    )
```

- 第一个shape作为controller。
- 接下来的control shape都会被储存到作为controller的transform的底下，因为cmds.parent(temporary_shape,root_control)

```
    cmds.delete(temporary_curve)

if root_control is None:
    raise ValueError(
        "Failed to create the custom control."
    )

if axis != "Z":
    orient_curve_to_axis(
        root_control,
        axis,
    )

return root_control
```

- 剩下的就是删除每次的临时curve然后 最后对准用户设置好的axis，返回control

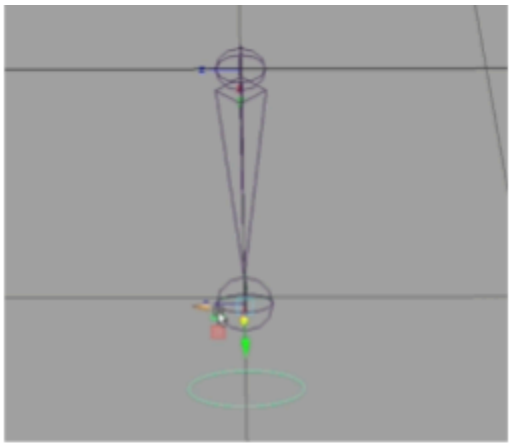
Lips stretch tool(嘴唇控制器）:
（小工具）（可以规划为自动绑定器的脸部模块 ）

这个工具是针对解决模拟嘴唇移动时候的肌肉形态所过多且重复的复杂的绑定骨骼（这个模块可以加进第一个自动化绑定）

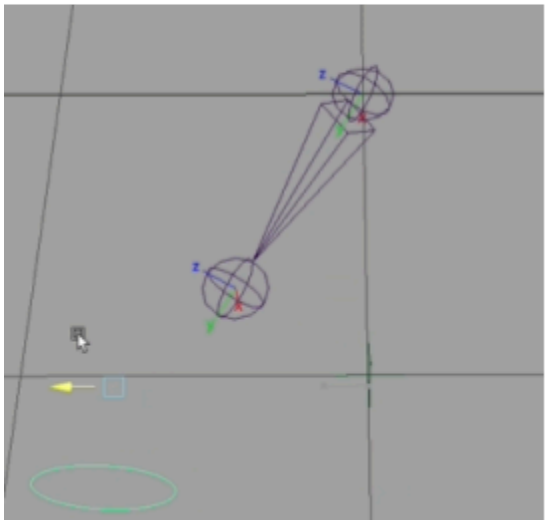
目的：当控制器往左右上下移动的时候骨骼要往后面缩小，后面的骨骼要跟随控制器的方向旋转。

节点实现原理：连接控制器的移动和前骨骼节点的旋转，通过distance between 节点算出初始距离，但是控制器会移动所以用locator来记录初始距离，根据初始距离再根据移动中的距离算出比例，根据比例让节点缩小。

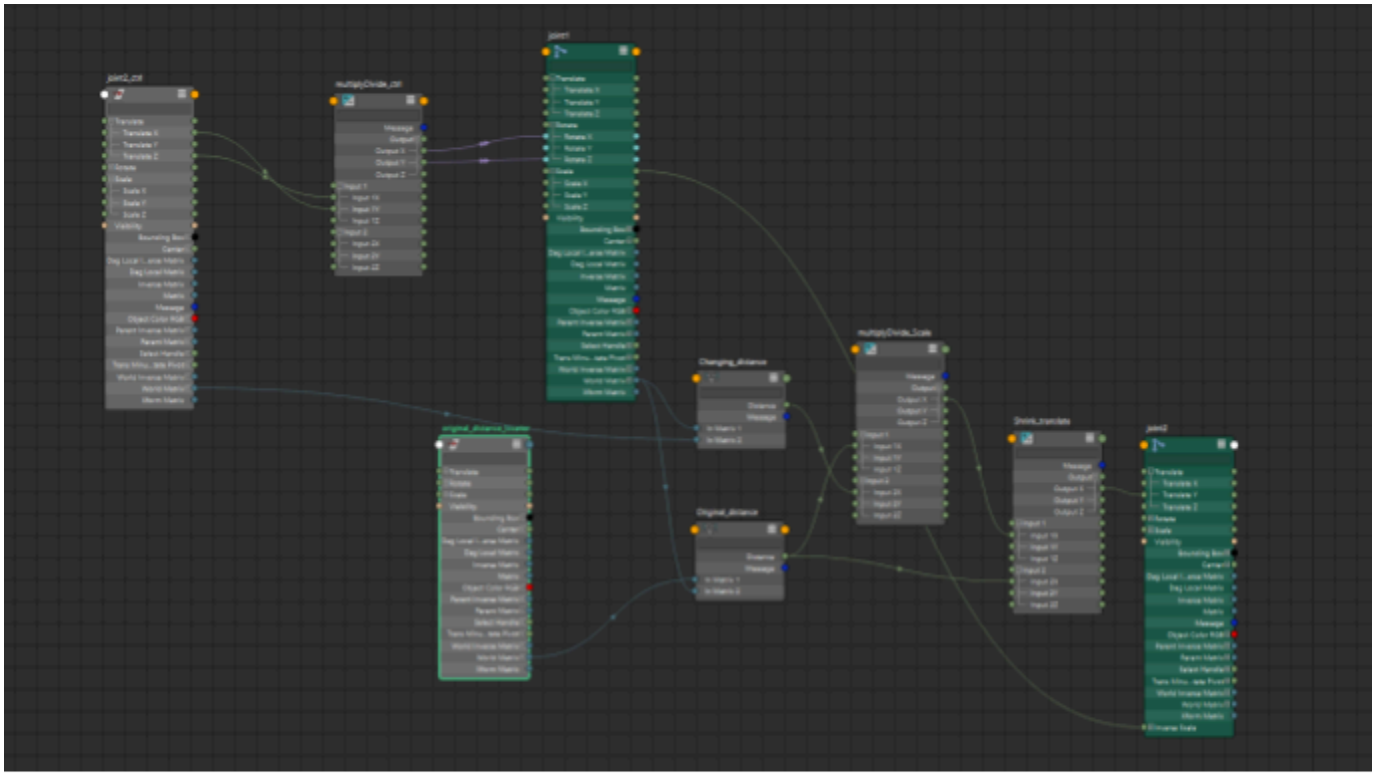
代码实现的区别：不需要设置locator来记录初始距离，直接记录初始距离来计算，然后使用的是aim constraint来让后面的骨骼跟随控制器旋转



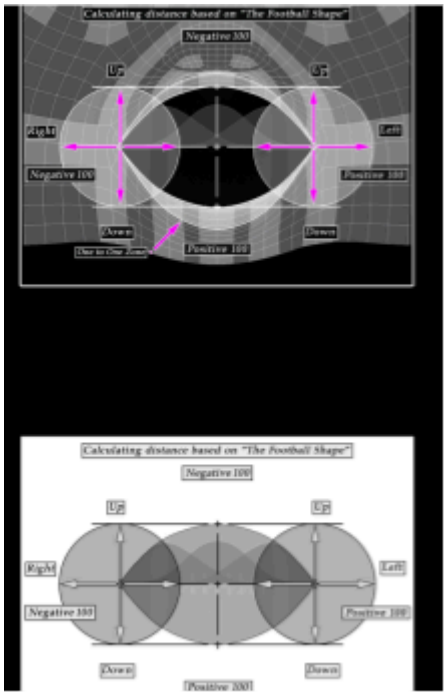
- 初始的骨骼



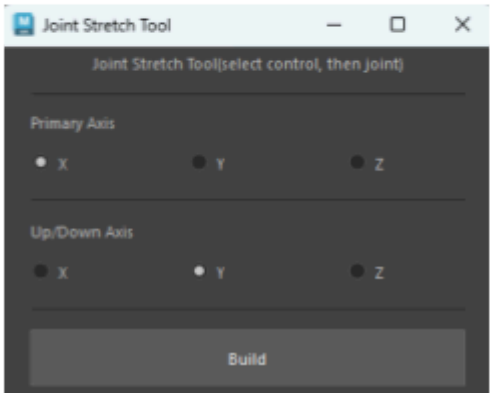
- 当控制器移动后面的骨骼跟着控制器旋转，前面的骨骼将会随着控制器的移动而向主轴后移动



- 节点实现图(因为嘴唇上所有节点都得进行繁琐设置 如果一个个连接节点将是大量的时间消耗)



- 参考图(可以用作调整嘴唇绑定和绑定控制器)



- 界面(使用者通过设置主轴，然后up down axis来进行部署)