VREngine Document

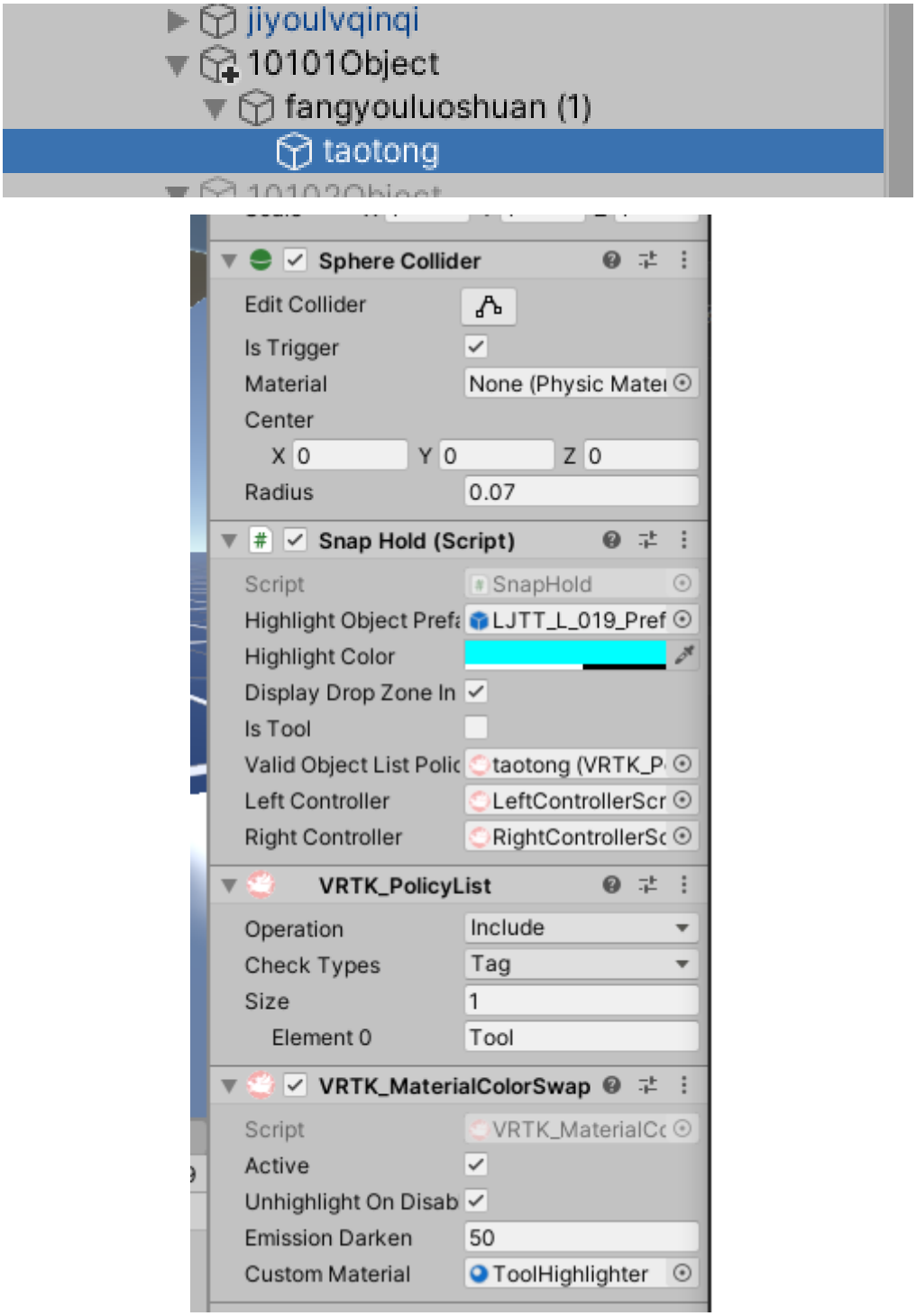
操作相关

添加一个需要放置的物体（例如拆卸时的套筒）

在需要对一些零件进行操作时，用一个空物体去容纳需要操作的物体，在操作时去显示需要操作的，而隐藏原有的，原因有二：(1) 不破坏原有的结构，prefab不允许删减层级；(2) 用空物体容纳之后，可以进行整体的，并让 localPosition 归零，可以让容纳的所有物体进行上下的平移（例如拧螺丝）。

> 使得新建的物体保持位置不变的方法：
> (1) 点想要复制的物体，按 `Ctrl + D` 在同级复制一遍；
> (2) 在父级下新建空物体；
> (3) 将复制的物体拖到空物体下，就可以保证相对位置不变（如果位置变成很小的偏差值，归零即可，不影响）

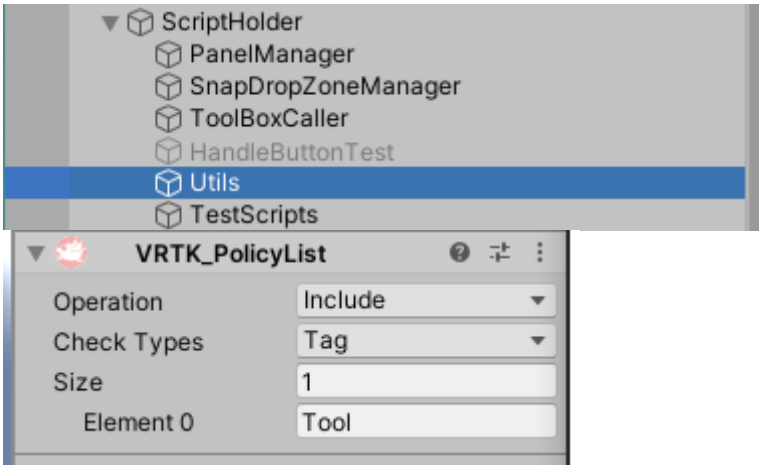
1. 在套筒空物体上添加以下脚本：



（1）`Collider`，通常是 `Sphere Collider` ，设置 `isTrigger` 为 `true` （两个Collider相碰撞，isTrigger为true的物体会触发 OnTriggerEnter 和 OnTriggerExit 方法，这两个方法属于 MonoBehaviour ，可在同级的脚本下重写）。`Radius` 可以自己调整；

（2）`SnapHold`，修改过的高亮脚本；

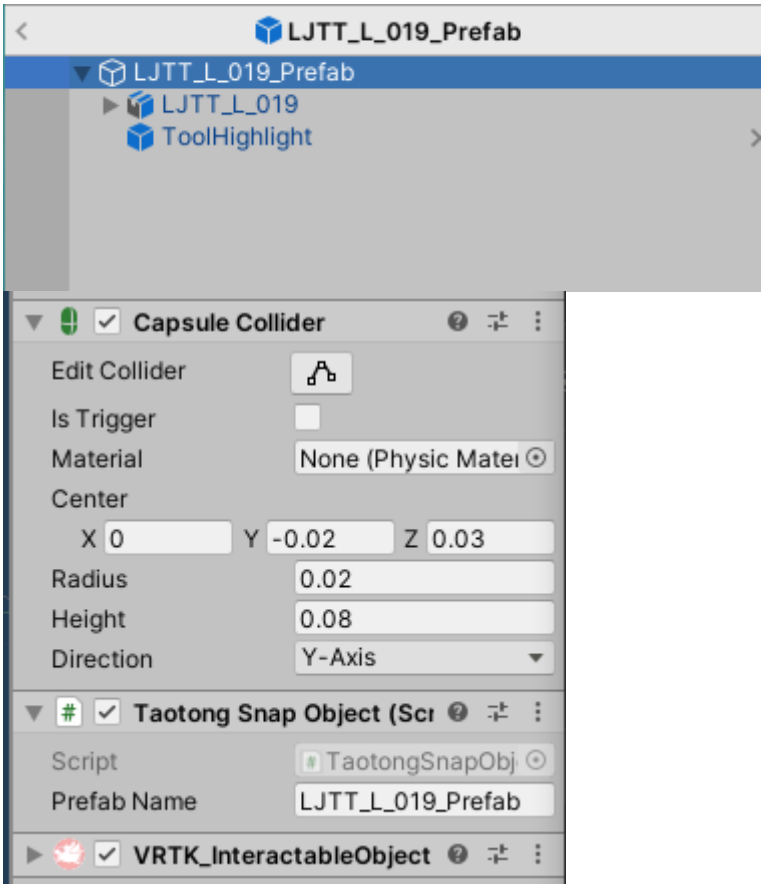
- 设置 `HighlightObjectPrefab` 为需要放置上去的零件
- `isTool` 设置为 `false` （false为按一下直接放置上去，true为按住时吸附上，松手时回到手上）。
- `ValidObjectListPolicy` 将 ScriptHolder/Utils 拖拽上去即可，或者像上图在当前物体添加一个 Component，将自己拖拽上去。之后可能改成运行时去寻找这个物体。



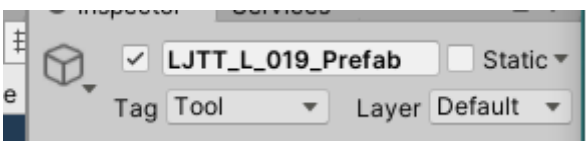
- 左右手控制器无需拖拽，在运行时自动寻找。

（3）`VRTK_MaterialColorSwapHighlighter` 按图中所示设置。

2. 高亮物体设置



（0）最重要的，把零件的Tag设置为 Tool：

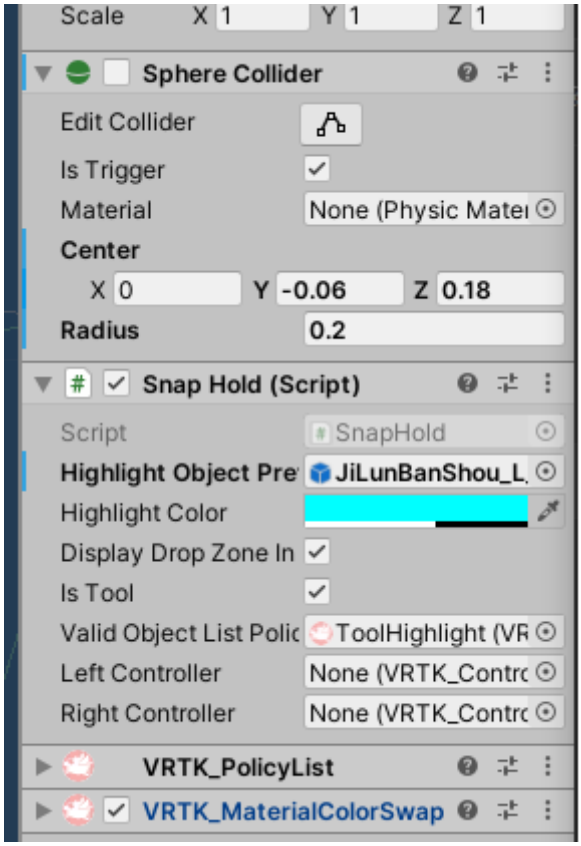


（1）添加 `Collider`，`isTrigger` 设置为 `false`（当然设为true之后写一些操作也可以），`Collider` 大小自己调节；

（2）添加一个 `SnapObjectBase` 子类，一般需要额外扳手操作的都是 `TaotongSnapObject`，名字设置为自己的Prefab名，以便控制高亮的唯一性；

（3）`VRTK_InteractableObject`，之前使用这个控制一些操作，也许之后可以去掉。

（4）如果需要额外的扳手操作，需要在Root节点下加一个 `ToolHighlight`（预制件）。和上面套筒类似（上面套筒应该也可以直接使用预制件）。需要注意的是扳手的话需要设置 `isTool` 为 `true`。



3. 调节高亮物体位置的方法：
    - 将所需高亮的物体拖拽到 `SnapHold` 脚本所在物体下；
    - 直接调节 `SnapHold` 所在物体的位置，使得物体到达它的目的位置。

流程控制

StepControllerBase

Attributes

1. protected string[] StepFunctions;

脚本使用反射通过字符串去调用每个步骤所使用的方法，例如在 `JiYouPaiFang.cs` 里：

```
protected override void Awake()
{
    base.Awake();
    StepFunctions = new string[]{"Origin", "FangYouLuoSai", "LvQingQiBanShou",
"LvQingQiChaiXie","Success"}; //后面会修改代码将第一个无用的去掉，或者作为Reset
    ConditionNum = new int[]{1, 1, 1, 1};
    Debug.Log("ChildAwake");
}

public void FangYouLuoSai() { }

public void LvQingQiBanShou() { }
```

```
public void LvQingQiChaiXie() { }

public virtual void Success() { }
```

2. protected int[] ConditionNum;

每一步骤需要满足的条件数，例如每个步骤需要放置多少个零件，拧多少个螺丝。**但是目前因为旋转螺丝需要一个全局的变量控制，可能需要每个螺丝分出单独的步骤，这一问题在后续考虑修复，如果无法修复，这个属性形同虚设。**

3. protected *Dictionary<GameObject, Vector3>* initPositions;

表示每个螺丝（以及相关的零件）的初始位置，用于控制拧螺丝的上下移动。

其他属性不多解释。

Methods

1. public virtual void Notify();

完成一个操作时，例如拧完一个螺丝，可以调用 `CommonUtil.NotifyStepController();` 来调用这个方法，以通知控制器一个操作已经完成。

2. public virtual void NextStep();

当某一步骤的条件数已满足时，会调用此方法，从 `StepFunctions` 取到下一步骤的方法并进行调用；

3. public virtual void Reset();

TODO

4. public virtual void ChangeTaotongHierarchy(string operateObjectName);

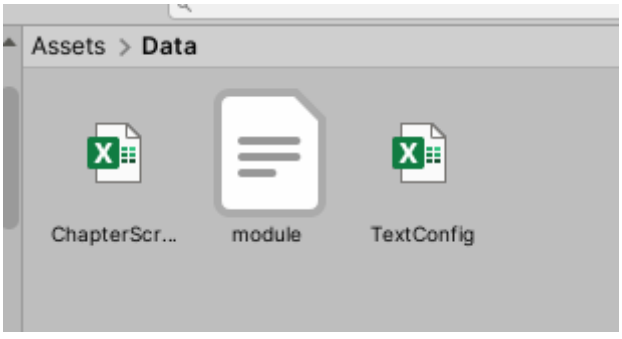
改变套筒放置后的层级结构，方便拧螺丝的操作（这里写得比较丑，如果有时间再做优化）。

StepScriptChooser

每一章节都有自己的一个控制脚本，这里通过查询 `ChapterScriptConfig` 去加载对应的控制脚本，控制脚本均继承自 `StepControllerBase`。

数据表

1. 原始数据



均放置在 `Assets/Data` 下。（之后如果多的话可以理一下结构）

2. 数据转换

因为打包之后，读取 Excel 和 json 的库均无法使用，因此将原始数据在编辑器下处理为 `.asset` 数据，在打包后去读取 `.asset` 数据。

> 编辑器下处理数据需要：
>
> ```
> #if UNITY_EDITOR
>         // Process Data
> #endif
> ```

- 准备条件
  - 所有需要读取的数据需要一个脚本作为 `DataHolder`，此脚本需要继承自 `ScriptableObject`，将需要保存的数据设计为 `DataHolder` 的属性，例如：

    ```
    using System.Collections.Generic;
    using UnityEngine;

    [System.Serializable]
    public class EngineModuleDataHolder : ScriptableObject
    {
        [SerializeField]
        public List<string> assembly;
        [SerializeField]
        public List<string> disassembly;
    }
    ```

  - 基本类型设置为public即可参与序列化，如果是 List<>等特殊类型就需要写 `[SerializeField]`。以防万一所有需要序列化的属性都写上即可。
  - 而 Dictionary 类型则不可参与序列化，如果是基本类型的映射，则只需要将键和值分别存到两个 List 内，在序列化前(`OnBeforeSerialize`)，和反序列化后(`OnAfterDeserialize`)，进行字典的转化和还原即可，例如 StepScriptDataHolder（调用这两个方法需要继承 `ISerializationCallbackReceiver` 接口）：

    ```
    [System.Serializable]
    public class StepScriptDataHolder : ScriptableObject,
    ISerializationCallbackReceiver
    {
        public Dictionary<string, string> scriptMap = new Dictionary<string,
    string>();

        [SerializeField]
        public List<string> keyList = new List<string>();
        [SerializeField]
        public List<string> valueList = new List<string>();

        public void OnBeforeSerialize()
        {
            keyList.Clear();
            valueList.Clear();

            foreach (var pair in scriptMap)
            {
                keyList.Add(pair.Key);
    ```

```
            valueList.Add(pair.Value);
        }
    }

    public void OnAfterDeserialize()
    {
        scriptMap.Clear();

        for (int i = 0; i < keyList.Count; ++i)
        {
            scriptMap[keyList[i]] = valueList[i];
        }
    }
}
```

- 而当有嵌套的数据结构，例如 string 映射到一个 List，则需要使用另外的类去容纳 List，例如 `LanguageDataHolder`：

```
public class LanguageDataHolder : ScriptableObject,
ISerializationCallbackReceiver
{
    [System.Serializable]
    public class LanguageDataItem
    {
        [SerializeField]
        public List<string> langData;

    }
    public Dictionary<string, List<string>> dict = new
Dictionary<string, List<string>>();
    [SerializeField]
    public List<string> keyList = new List<string>();
    [SerializeField]
    public List<LanguageDataItem> valueList = new List<LanguageDataItem>
();

    ...
}
```

注意，此处 `LanguageDataItem` 并未继承自 `ScriptableObject`。

- Excel 数据
  - 需要使用额外的库 `using System.Data;`
  - 使用 `ExcelUtil.ReadExcel()` 读取到 `DataRowCollection` 格式的数据（相当于一个二维数据，读取到原生的数据，包括第一行的key，在处理的时候需要去掉第一行）。用例：

```
string filePath = Application.dataPath +
"/Data/ChapterScriptConfig.xlsx";
int columnNum = 0, rowNum = 0;
DataRowCollection collection = ExcelUtil.ReadExcel(filePath, 0, ref
columnNum, ref rowNum);//获得行与列的值
StepScriptDataHolder holder =
ScriptableObject.CreateInstance<StepScriptDataHolder>();
holder.scriptMap = new Dictionary<string, string>();
for(int i = 1; i < rowNum; i++)
{
holder.scriptMap.Add(collection[i][0].ToString(), collection[i]
[1].ToString());
}

ExcelUtil.CreateAsset("ChapterScriptConfig", holder);
```

    - DataHolder 需要使用 `ScriptableObject.CreateInstance<>()` 创建对象，而不是 `new`；
    - 直接去操作 DataHolder 的属性，或者内部去提供接口均可；
    - 最后使用Excel.CreateAsset() 去将Excel保存为 `.asset`。
  - 在打包后运行时，使用如下格式去读取数据即可

```
StepScriptDataHolder holder = Resources.Load<StepScriptDataHolder>
(dataName);
```

- Json 数据
  - 需要使用额外的库 `using LitJson; using System.IO;`
  - 参考以下读取即可：

```
public EngineModuleDataHolder ReadJson(string str)
{
    StreamReader StreamReader = new StreamReader(Application.dataPath +
str);
    JsonReader js = new JsonReader(StreamReader);
    return JsonMapper.ToObject<EngineModuleDataHolder>(js);
}
```

  - 使用同Excel