

连接与集成开发笔记

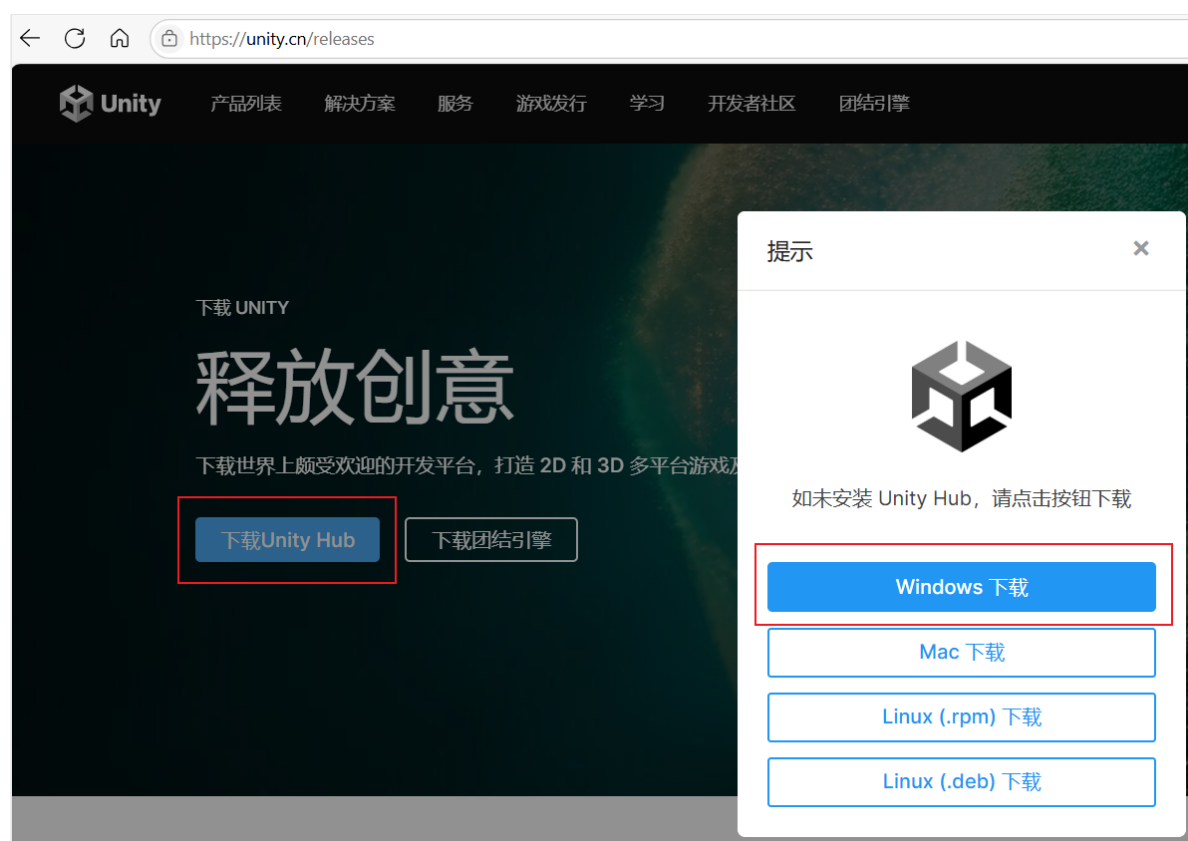
1. 开发环境搭建

1.1. 软件安装

1.1.1. Unity Hub安装

Unity Hub是一个多个版本工程管理的脚手架工具，**建议下载!!!**

下载链接直接搜索unity官网即可：<https://unity.cn/releases>

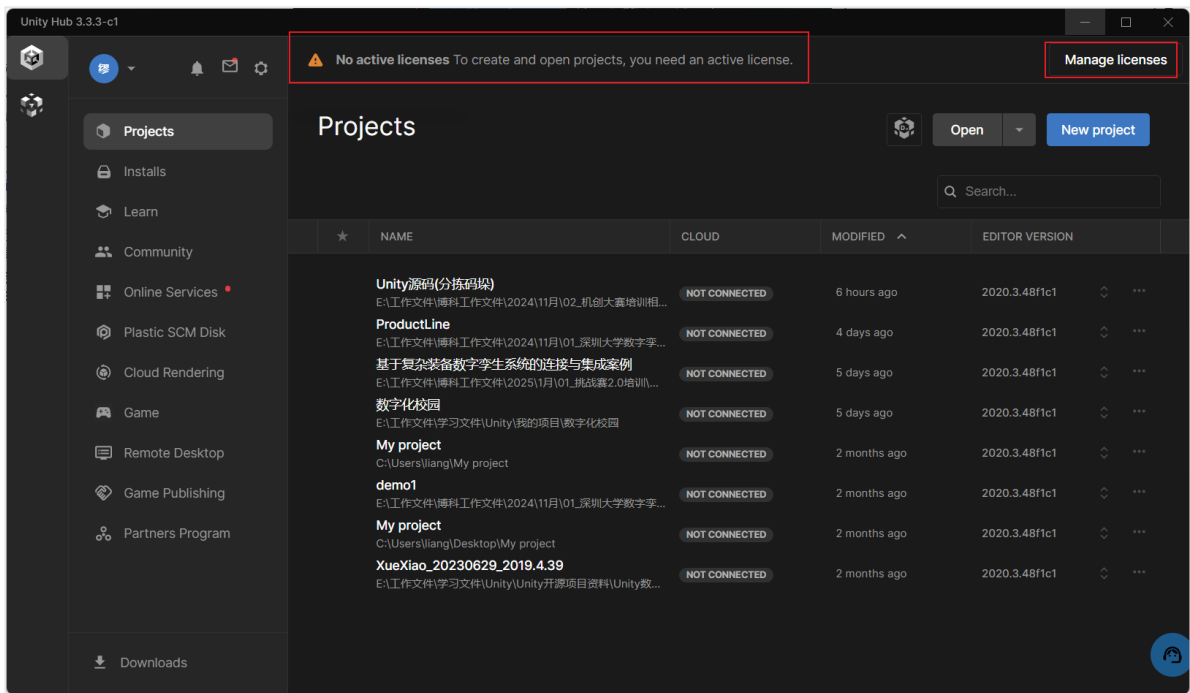


安装包下载以后，傻瓜式安装即可

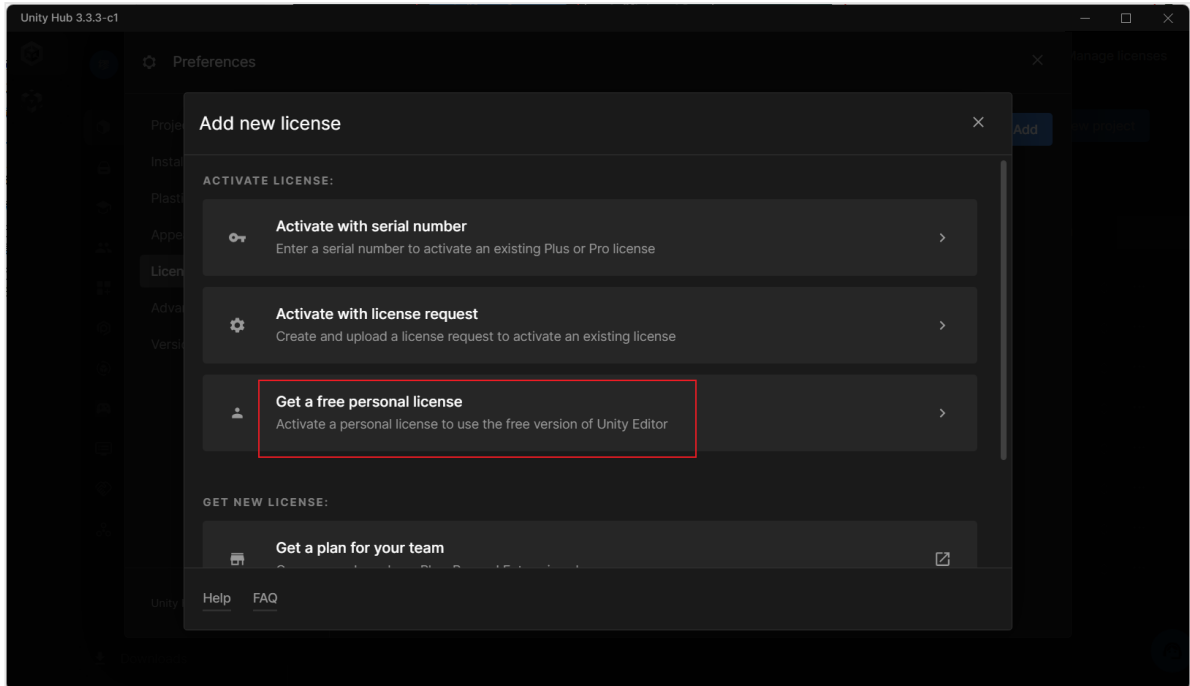
1.1.2. 获取Unity Hub许可证

Unity Hub下载之后，打开需要先注册登录账号 并添加许可证！

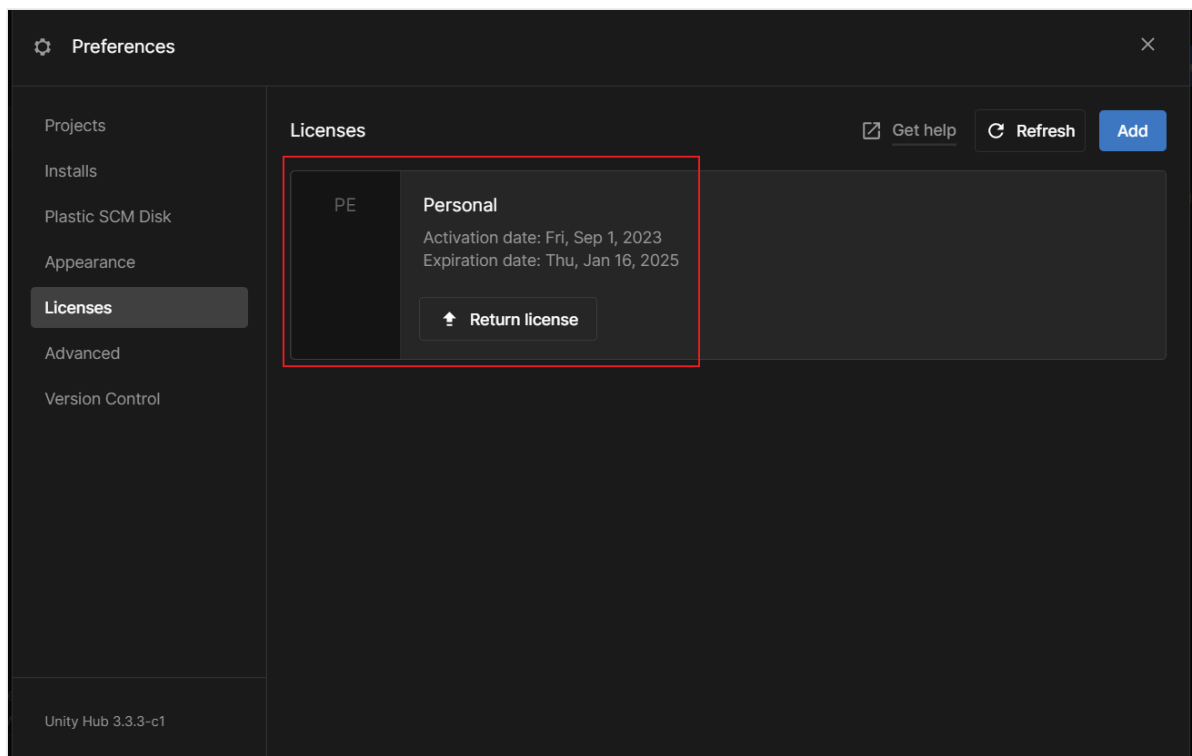
它这里就显示我没有激活许可证，**直接点击"Manage licenses"**，



点击获取个人开发许可证

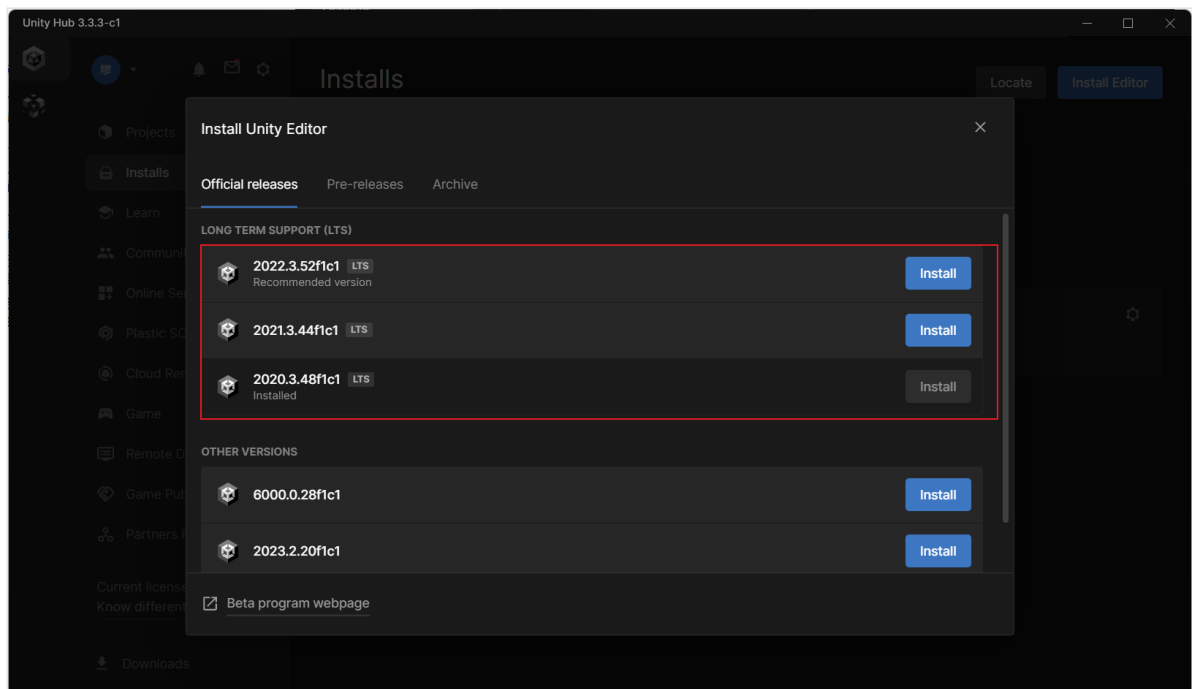


个人许可证每周要手动更新一次，如果不想这么麻烦，您可以动用您的“钞能力”直接购买开发版

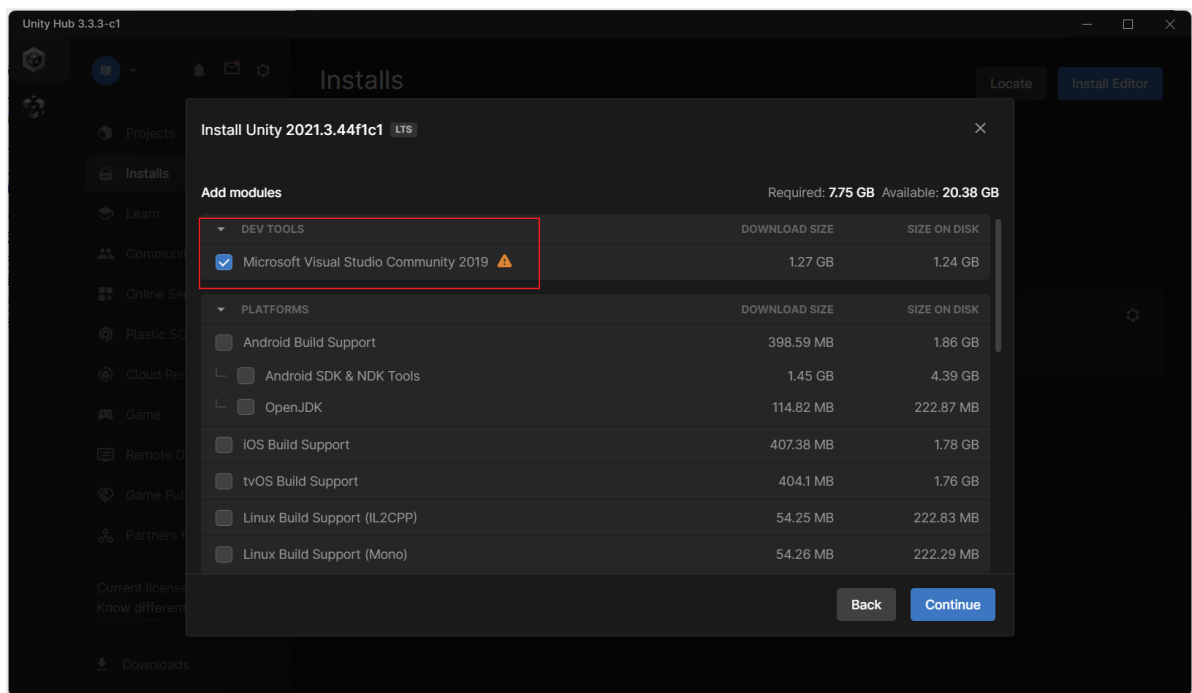


1.1.3. Unity安装

我们直接在Unity Hub中安装Unity游戏引擎即可,这里官方推荐了三个长期支持的版本，**任意选择一个版本下载即可**



这里的VS2019我们不用下载，取消勾选，后面我们直接下载VS2022

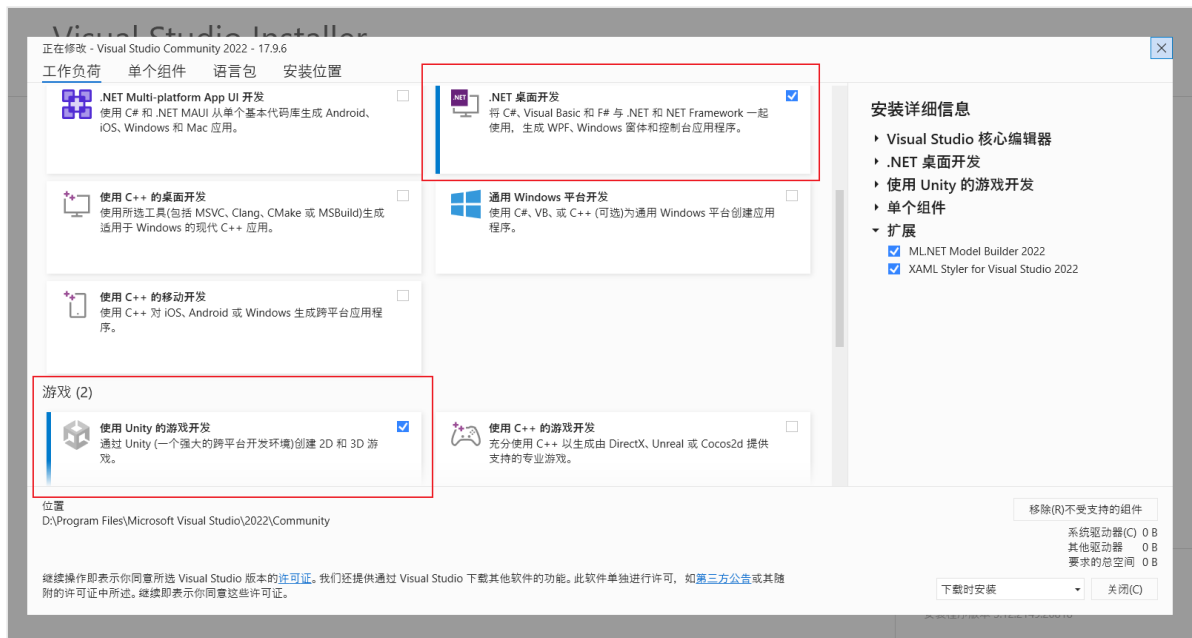


1.1.4. Visual Studio 2022安装

直接安装Visual Studio 2022 Community社区版本



VS2022安装的时候，这两个勾选上：“.NET桌面开发”和“使用Unity的游戏开发”



点击“单个组件”，.NET6.0和.NET8.0勾不勾选无所谓

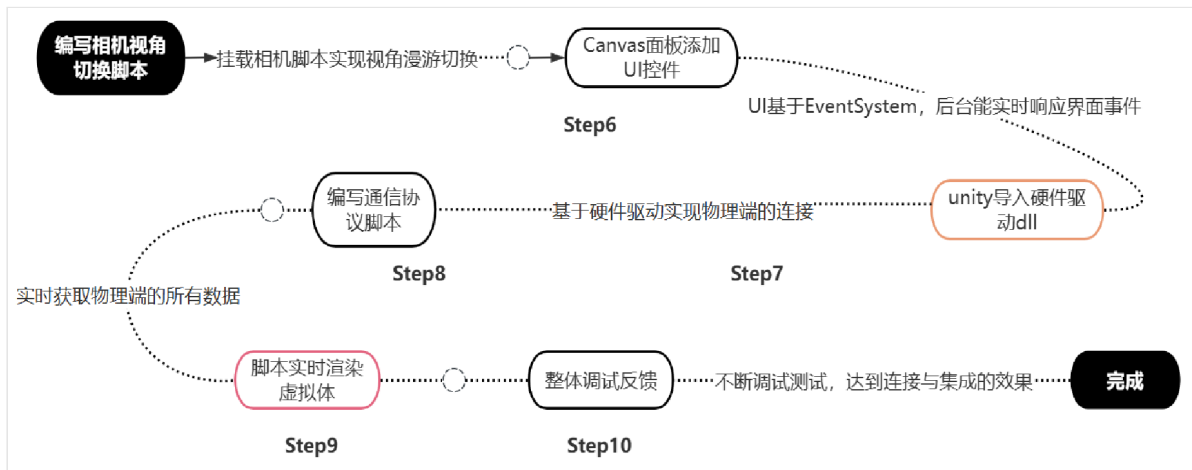
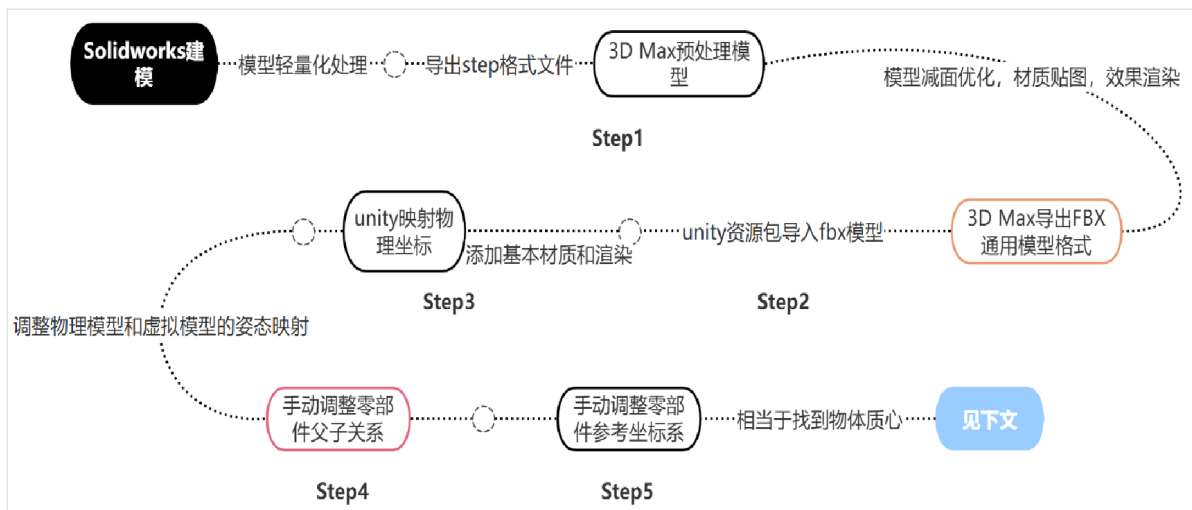
.NET Framework的组件一定要勾选上

搜索组件(Ctrl+Q) 

- ☒ .NET 6.0 运行时(长期支持)
- ☐ .NET 7.0 WebAssembly Build Tools
- ☐ .NET 7.0 运行时
- ☒ .NET 8.0 运行时(长期支持)
- ☐ .NET Core 2.1 运行时(不受支持)
- ☐ .NET Core 3.1 运行时(不受支持)
- ☐ .NET Framework 3.5 开发工具
- ☒ .NET Framework 4.6 目标包
- ☐ .NET Framework 4.6.1 SDK
- ☒ .NET Framework 4.6.1 目标包
- ☐ .NET Framework 4.6.2 SDK
- ☒ .NET Framework 4.6.2 目标包
- ☐ .NET Framework 4.7 SDK
- ☐ .NET Framework 4.7 目标包
- ☐ .NET Framework 4.7.1 SDK
- ☒ .NET Framework 4.7.1 目标包
- ☐ .NET Framework 4.7.2 SDK
- ☒ .NET Framework 4.7.2 目标包
- ☒ .NET Framework 4.8 SDK
- ☒ .NET Framework 4.8 目标包
- ☐ .NET Framework 4.8.1 SDK
- ☐ .NET Framework 4.8.1 目标包
- ☐ .NET Framework 项目和项模板

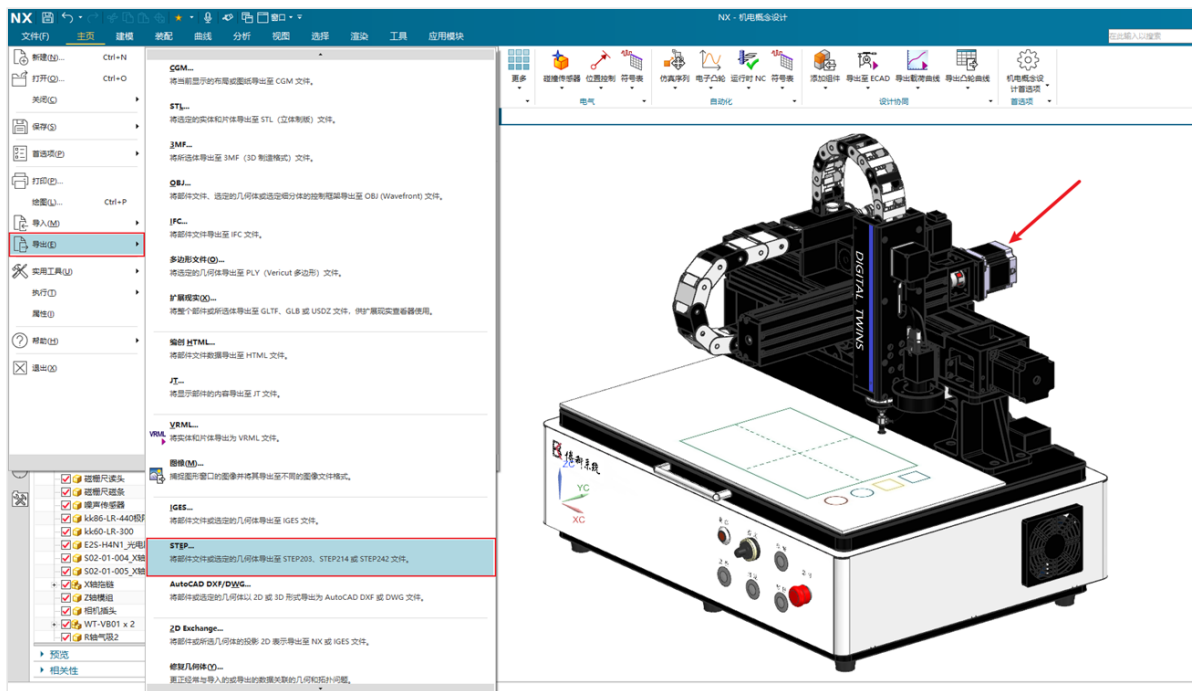
2. 开发流程简述

2.1. 流程框图



2.2. 具体细节

2.2.1. 模型的轻量化与导出



基于unity3D实现连接与集成

① 模型轻量化与导出

➤模型导出:

进入UG NX软件; 打开装配体, 选择导出STEP文件

➤轻量化处理:

选择“要导出的数据”, 只勾选“实体”, 这样做可以有效减小导出的模型体积



(选择AP214格式)



(只选择实体)

2.2.2. 模型减面与材质贴图

基于unity3D实现连接与集成

② 模型减面与材质贴图

➤模型减面:

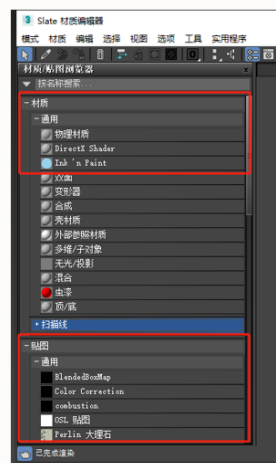
将模型导入到3d max软件中, 在修改器列表中选择“专业优化”, 并设置优化定点数

➤材质贴图:

按下快捷键M键, 为需要的模型选择合适的材质和贴图



(减面优化)



(材质贴图)

2.2.3. 导出FBX通用格式文件

基于unity3D实现连接与集成

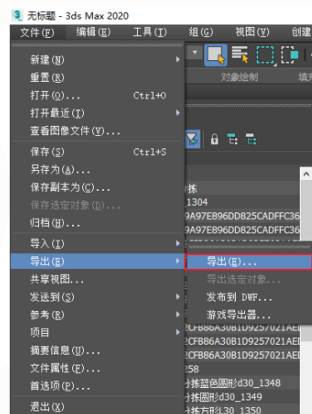
3 导出FBX通用格式文件

➤导出FBX文件:

点击3dmax软件“文件”选项, 点击“导出”

➤几何体轻量化:

在导出选项中勾选“涡轮平滑”和“保留边缘方向”, 这样可以最大限度减小模型体积



(选择AP214格式)



(只选择实体)

2.2.4. 物理端与虚拟端坐标系映射

基于unity3D实现连接与集成

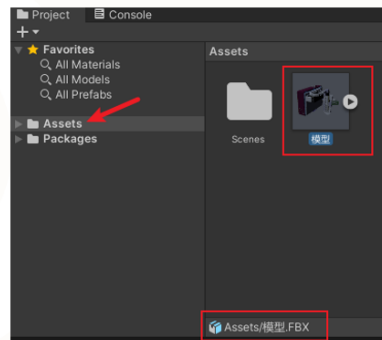
4 物理端与虚拟端坐标系映射

➤导入FBX文件:

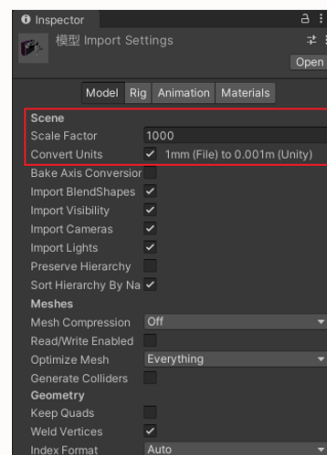
将3dmax转出的fbx文件导入到unity的Assets文件夹

➤设置单位和缩放:

在右边的Inspector面板设置合适的缩放比例因子和转换单位



(unity导入fbx文件)



(根据实际情况调整单位和缩放比例)

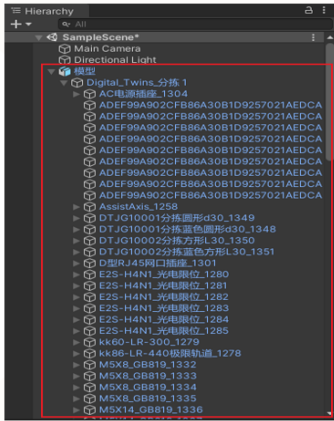
2.2.5. 调整所有零部件父子关系

基于unity3D实现连接与集成

5 调整所有零部件父子关系

➤添加父子关系:

在将fbx添加到场景视图后,我们对相应的运动模组添加父子关系,比如右图看到的X Y Z R模组



(调整前)



(调整后)

2.2.6. 编写并挂载相机脚本

基于unity3D实现连接与集成

6 编写并挂载相机脚本

➤编写相机脚本:

场景模型处理完毕之后,编写相机脚本,方便程序调试和运行时能够以各种视角观察模型状态

```
public class CameraController : MonoBehaviour
{
    /// <summary> 视角类型.
    private enum ViewType...

    /// <summary> 鼠标按键.
    private enum MouseButton...

    /// <summary> 指针样式.
    private enum CursorType...

    字段
    Unity_Method
    初始化相关
    切换视角
    拉近拉远
    拖动
    360° 旋转观察
    聚焦选中物体
    各种视图
}
```

(编写CameraController相机脚本)



(挂载相机脚本)

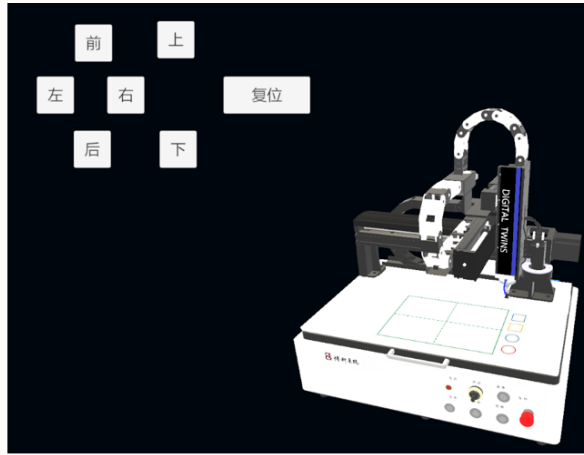
2.2.7. Canvas面板添加UI

基于unity3D实现连接与集成

7 Canvas面板添加UI

➤添加按钮:

添加人机交互的按钮以及注册后台事件系统,方便后续调试功能,达到虚控实,实控虚的效果。



(添加UI素材及UI事件)

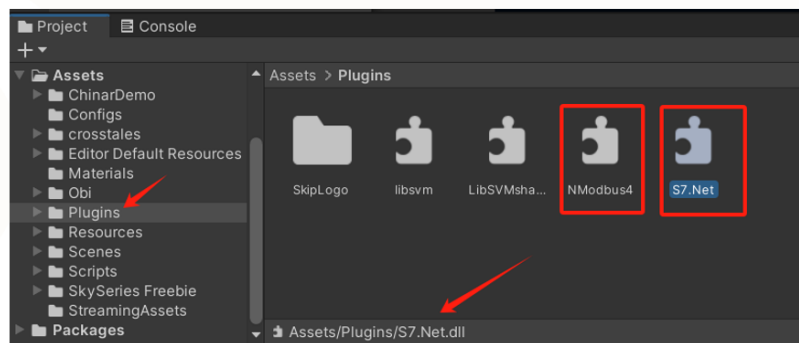
2.2.8. 导入硬件驱动dll

基于unity3D实现连接与集成

8 导入硬件驱动dll

➤导入驱动:

在Assets目录下新建Plugins目录,这个文件夹专门用来存放硬件驱动的dll,比如需要用到的S7.Net.dll和NModbus4.dll。



(导入驱动)

2.2.9. 编写通信协议脚本

基于unity3D实现连接与集成

9 编写通信协议脚本

➤PLC端部分:

在博途软件中新建Unity数据块, 这个数据块存放的就是虚拟端unity脚本需要通信拿到的内容。

名称	数据类型	偏移量	起始值	保持
Static				
X轴当前位置	Real	0.0	0.0	
Y轴当前位置	Real	4.0	0.0	
Z轴当前位置	Real	8.0	0.0	
R轴当前位置	Real	12.0	0.0	
区域位置计数	Dint	16.0	0	
分拣完成状态	Real	20.0	0.0	
全部分拣完成	Bool	24.0	false	
视觉心跳	Bool	24.1	false	
系统初始化完成	Bool	24.2	false	
识别到的工件数	Dint	26.0	0	
X轴剩余	Dint	30.0	100	
Y轴剩余	Dint	34.0	100	
Z轴剩余	Dint	38.0	100	
R轴剩余	Dint	42.0	100	
A产品产量	Dint	46.0	0	
B产品产量	Dint	50.0	0	
C产品产量	Dint	54.0	0	
D产品产量	Dint	58.0	0	
X轴当前位置_1	Real	62.0	0.0	

(PLC数据块)

基于unity3D实现连接与集成

9 编写通信协议脚本

➤虚拟端部分:

在unity脚本中, 先通过S7库实例化与PLC通信的一个基础类; 再实例化一个与PLC数据块相对应的类。

```
public static class S7_1200
{
    public static readonly Plc plc = new Plc(CpuType.S71200, "192.168.1.10", 0, 1);
    static S7_1200()
    {
        try
        {
            //S7_1200.plc.Open();
        }
        catch (System.Exception ex)
        {
            //Debug.Log(ex);
            //S7_1200.plc.Open();
        }
        Debug.Log($"this is S7_1200");
    }
}
```

(PLC通信类)

```
public class ProductHelper
{
    #region 要获取的信息项
    1 个引用
    public float xPosition_Axis { get; set; } //X轴实时位置
    1 个引用
    public float yPosition_Axis { get; set; } //Y轴实时位置
    1 个引用
    public float zPosition_Axis { get; set; } //Z轴实时位置
    1 个引用
    public float rPosition_Axis { get; set; } //R轴实时位置
    1 个引用
    public int AreaCount { set; get; } //区域数量
    1 个引用
    public float SortState { set; get; } //分拣状态
    1 个引用
    public bool IsOver { set; get; } //分拣是否完成
    1 个引用
    public bool IsProduct { set; get; } //是否扫描到工件
    1 个引用
    public bool IsResetOver { set; get; } //是否复位完成
    1 个引用
    public int ProductCount { set; get; } //产品数量
    1 个引用
    public int X_Remain { set; get; } //X轴剩余
    1 个引用
    public int Y_Remain { set; get; } //Y轴剩余
    1 个引用
    public int Z_Remain { set; get; } //Z轴剩余
    1 个引用
    public int R_Remain { set; get; } //R轴剩余
    1 个引用
    public int A_Count { set; get; } //A产品产量
    1 个引用
    public int B_Count { set; get; } //B产品产量
    1 个引用
    public int C_Count { set; get; } //C产品产量
    1 个引用
    public int D_Count { set; get; } //D产品产量
    1 个引用
}
```

(PLC数据块对应类)

2.2.10. 实时渲染虚拟体

基于unity3D实现连接与集成

10 实时渲染虚拟体

➤虚拟端部分:

在渲染脚本方法FixedUpdate中, 根据通信协议拿到的实时数据, 实时刷新模組的Transform位置和Rotate姿态信息, 达到物理端和虚拟端实时同步的效果。

```
private void FixedUpdate()
{
    if (true)
    {
        var goPosition = x_CurrentPosition - x_LastPosition;
        x_LastPosition = x_CurrentPosition;
        if (Math.Abs(goPosition) > 0)
        {
            X_Lead_screw_Transform.Rotate(0, 0, (float)(goPosition * 20.0));
            X_Coupling_Transform.Rotate((float)(goPosition * 20.0), 0, 0);
            x_Transform.Translate(new Vector3((float)(goPosition / 1.0), 0));
            if (IsHide == false)
            {
                IsHideComponent(true);
            }
            z_chain_transform.Translate(new Vector3(0, (float)(goPosition / 1.0), 0));
            z_chain_transform.Rotate(0, 0, 0);
            z_chain_board_transform.Translate(new Vector3((float)(goPosition / 1.0), 0));

            t1_transform.Translate(new Vector3((float)(goPosition / 1.0), 0));
            foreach (var item in listRigidbody)
            {
                item.WakeUp();
            }
        }
    }
}
```

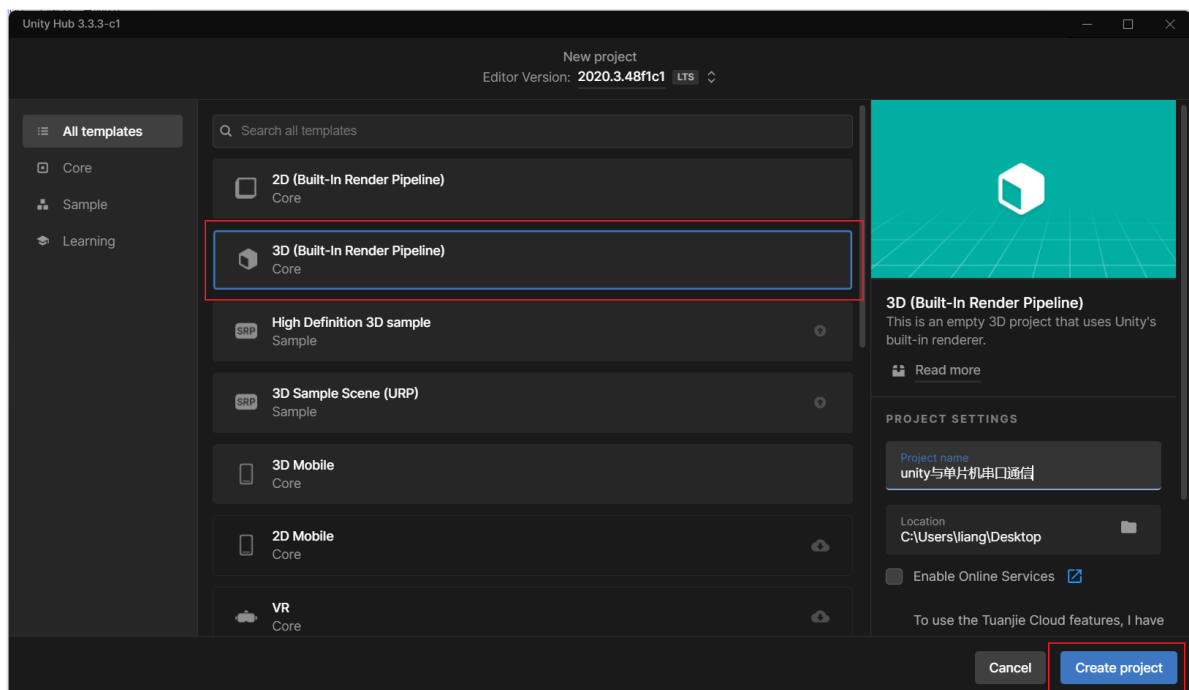
(脚本代码)

3. 简单示例：unity与单片机串口通信

3.1. 开发流程

3.1.1. 新建工程

选择 3D (Built-in Render Pipeline) 模板，项目名称和保存路径可自由选择,然后点击“Create project”,即可创建基本工程



3.1.2. 在Assets目录下新建文件夹

在Assets目录下新建的文件夹名称**不是随便取的，是有约定的，一定要保持一致**,这软件工程中称之为“约定大于配置”

"Scenes"文件夹：

"Scenes"文件夹是用来存放场景文件的，所谓的场景，就是游戏画面

新建"Scripts"文件夹：

在“Scripts”文件夹中存放的是工程所有的代码文件（或者说是脚本文件），**我们所有新建编写的脚本文件都要存放在这里**

新建“Plugins”文件夹：

这个“Plugins”文件夹用来**存放外部dll动态链接库的**，包括外部的插件，也包括和物理硬件通信的硬件驱动dll文件，都是放置在这个文件夹中

新建“Resources”文件夹：

在“Resources”文件夹中，我们**存放的是UI元素和预制体**

文件夹“Assets\Resources\Images”中存放的是Canvas UI面板中会使用到的图片素材 比如说背景图片

文件夹“Assets\Resources\Prefabs”中存放的是预制体的信息,也就是.prefab文件，这个预制体实际上就是提前准备的物体，比如“萝卜”，“野草”，“树林”，“土壤”等场景中元素

新建“Materials”文件夹：

在这个“Materials”文件夹中，我们存放的是材质球和贴图，这些东西都是我们用来给物体做外观渲染用的

新建“StreamingAssets”文件夹：

StreamingAssets 是 Unity 项目中的一个特殊文件夹，**用于存储需要在运行时以原始格式直接访问的文件**

在这个文件夹中的文件，**是不允许被“写入”的！！！！**，也就是只能读取

这个文件夹有什么用？

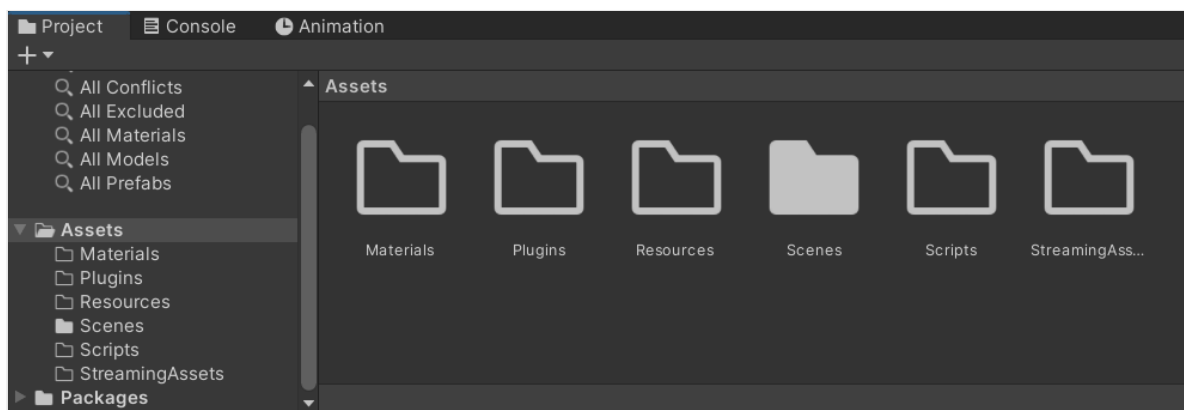
比如说，**我们的配置文件(json文件)**，就可以放置在这里,在项目运行的时候，读取配置内容

比如说，我们的多媒体资源(.mp4视频)，就可以放置在这里,在项目运行的时候，加载视频流

再次要强调一点：在StreamingAssets这个文件夹中的文件，是不允许被“写入”的！！！，也就是只能读取

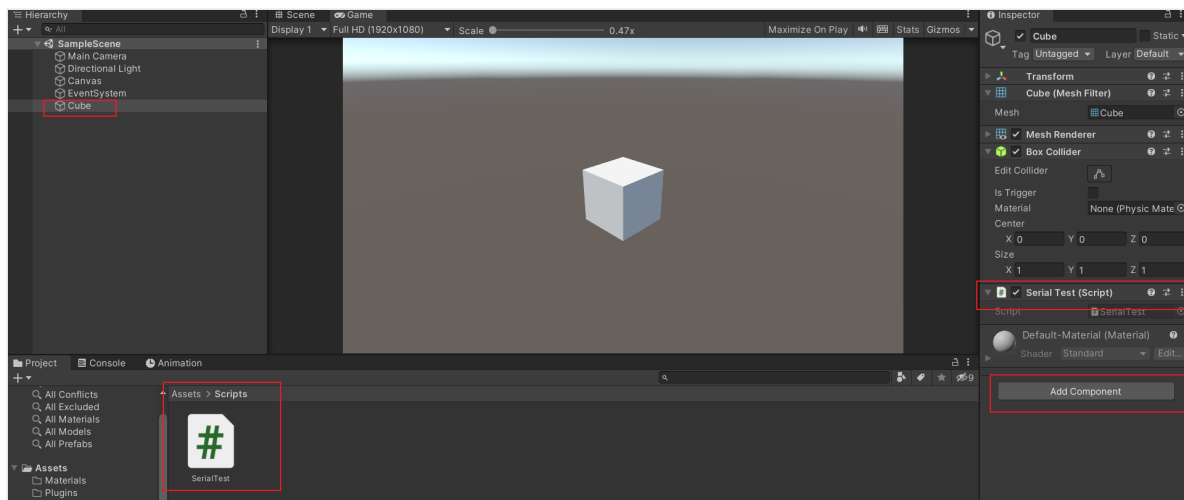
数据库也存放在这里

最终，文件夹目录结构如下：

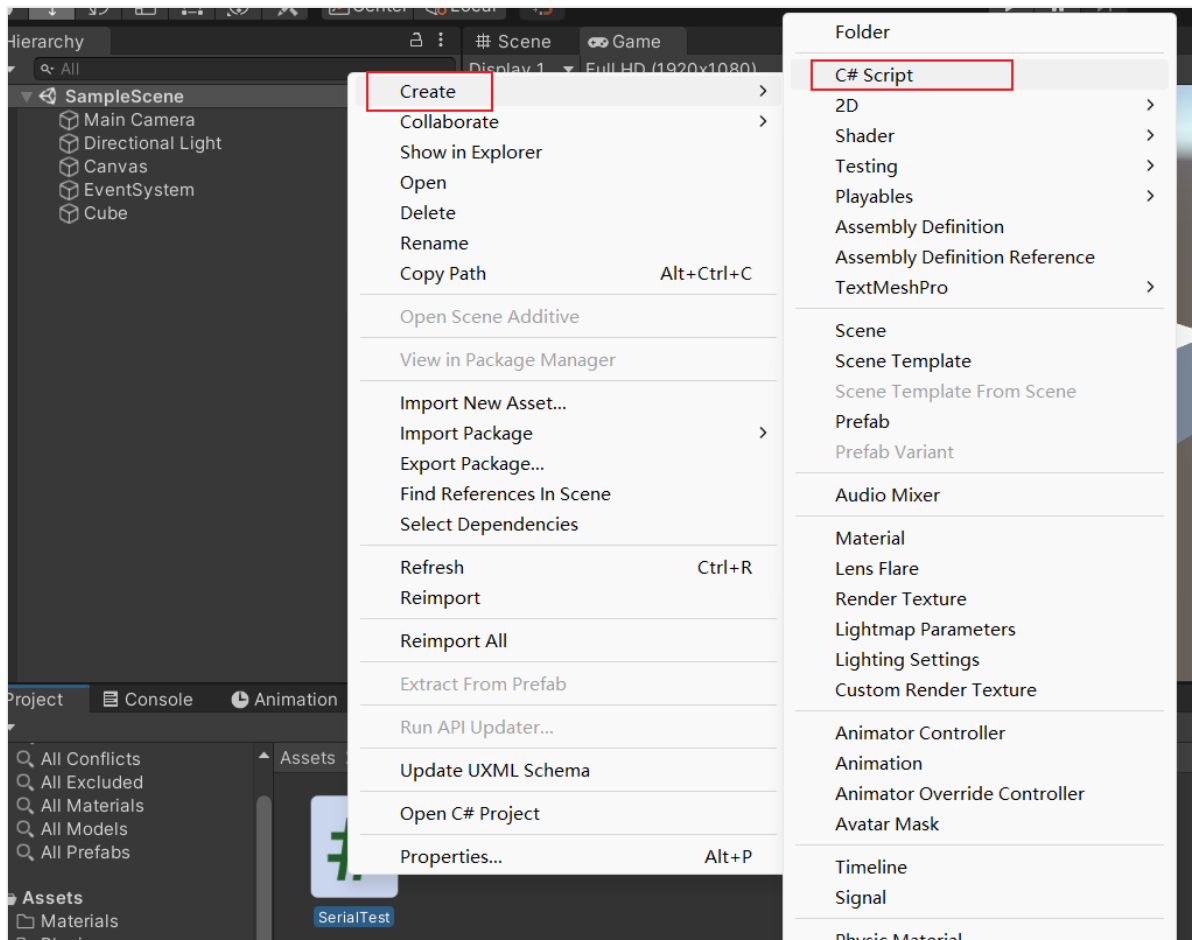


3.1.3. 新建“Cube”对象并挂载脚本

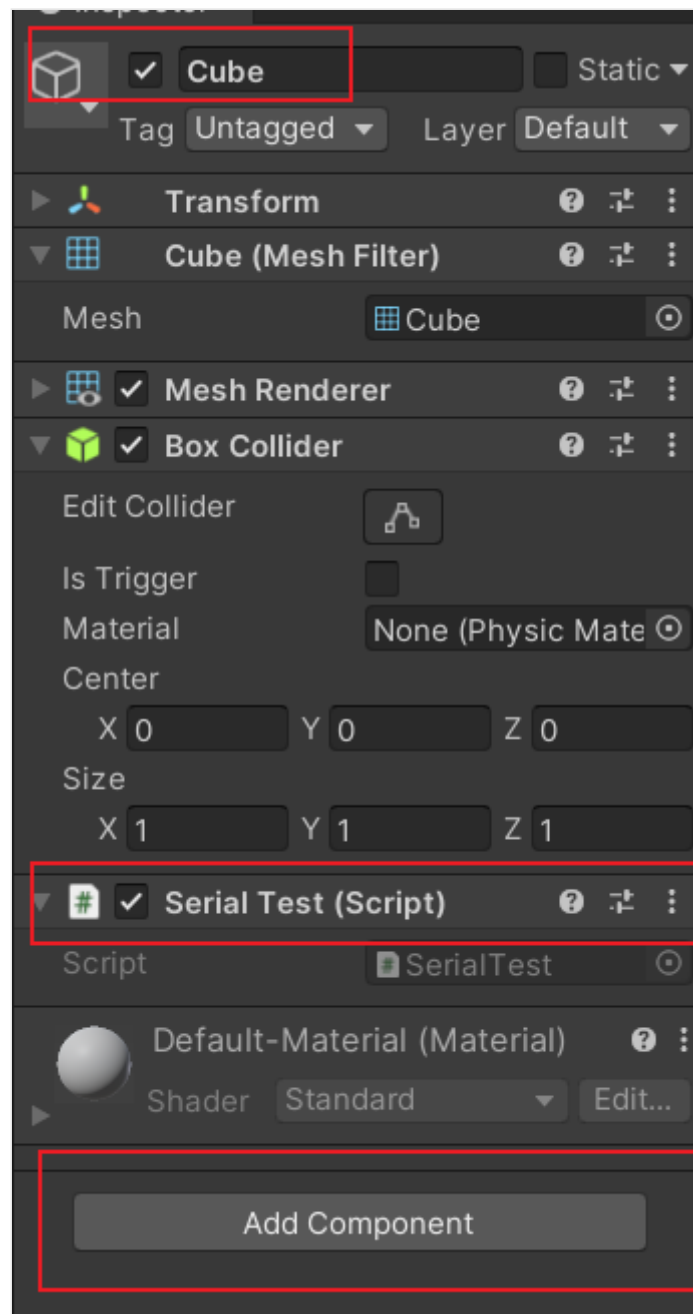
我们在场景中新建一个“Cube”对象用来挂载脚本



在Scripts文件夹中，新建脚本文件“SerialTest”



在“Cube”对象的Inspector面板中点击“Add Component”**添加脚本组件**,讲我们新建的脚本文件SerialTest挂载到Cube对象上,随后进行代码编写



3.1.4. 编写脚本串口通信程序

双击打开“SerialTest”脚本,编写如下代码:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.IO.Ports;
using System.Threading.Tasks;
using System;
using System.Text;
using UnityEngine.UIElements;

public class SerialTest : MonoBehaviour
```



```

        {
            m_str = str;
            Debug.Log("物体向右移动");
        }
        m_serial.DiscardInBuffer();
        m_serial.DiscardOutBuffer();
    }
}

});

}

}

//脚本每一帧更新执行的方法
void Update()
{

}

private void FixedUpdate()
{
    if (m_str == null) return;

    if (m_str.IndexOf("RIGHT") != -1)
    {
        m_transform.Translate(-1, 0, 0);
        m_str = null;
    }
    else if (m_str.IndexOf("LEFT") != -1)
    {
        m_transform.Translate(1, 0, 0);
        m_str = null;
    }
}

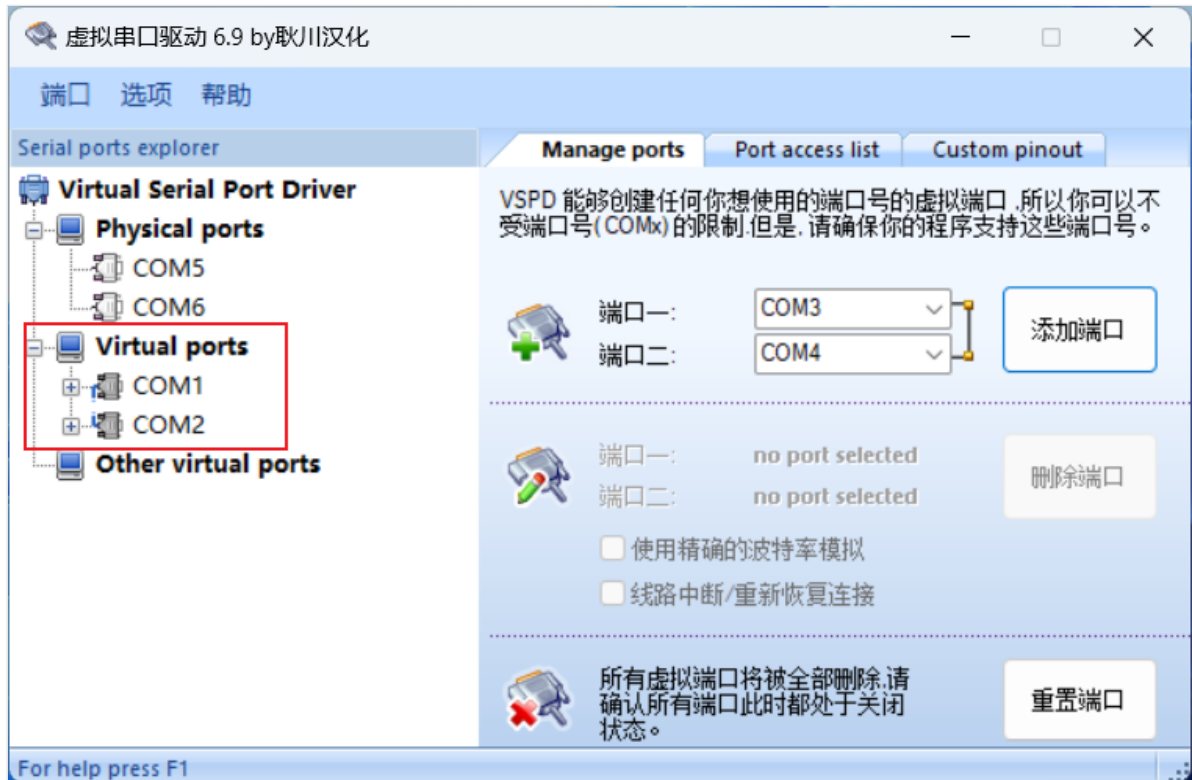
private void OnApplicationQuit()
{
    m_serial.Close();
    Debug.Log("串口已关闭");
}

```

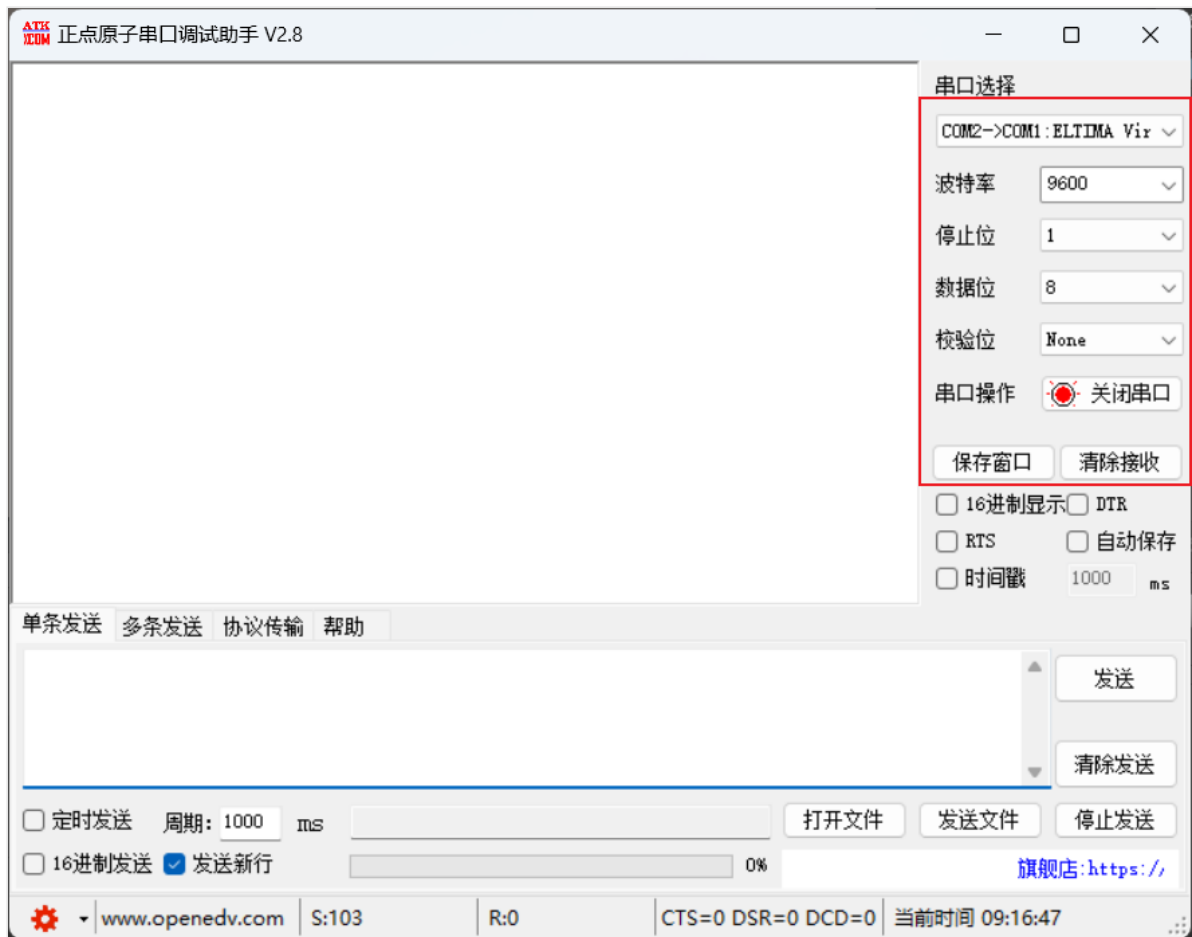
```
}
```

3.1.5. 测试

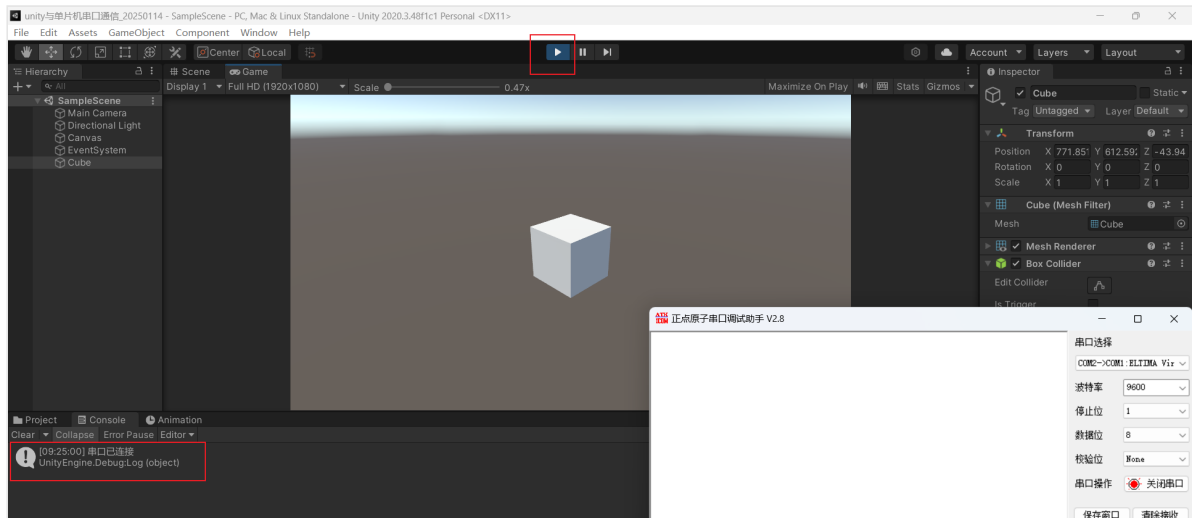
我这里没有单片机实物，直接在电脑上使用虚拟串口去模拟单片机，在PC端，我使用软件，模拟出来了串口1和串口2



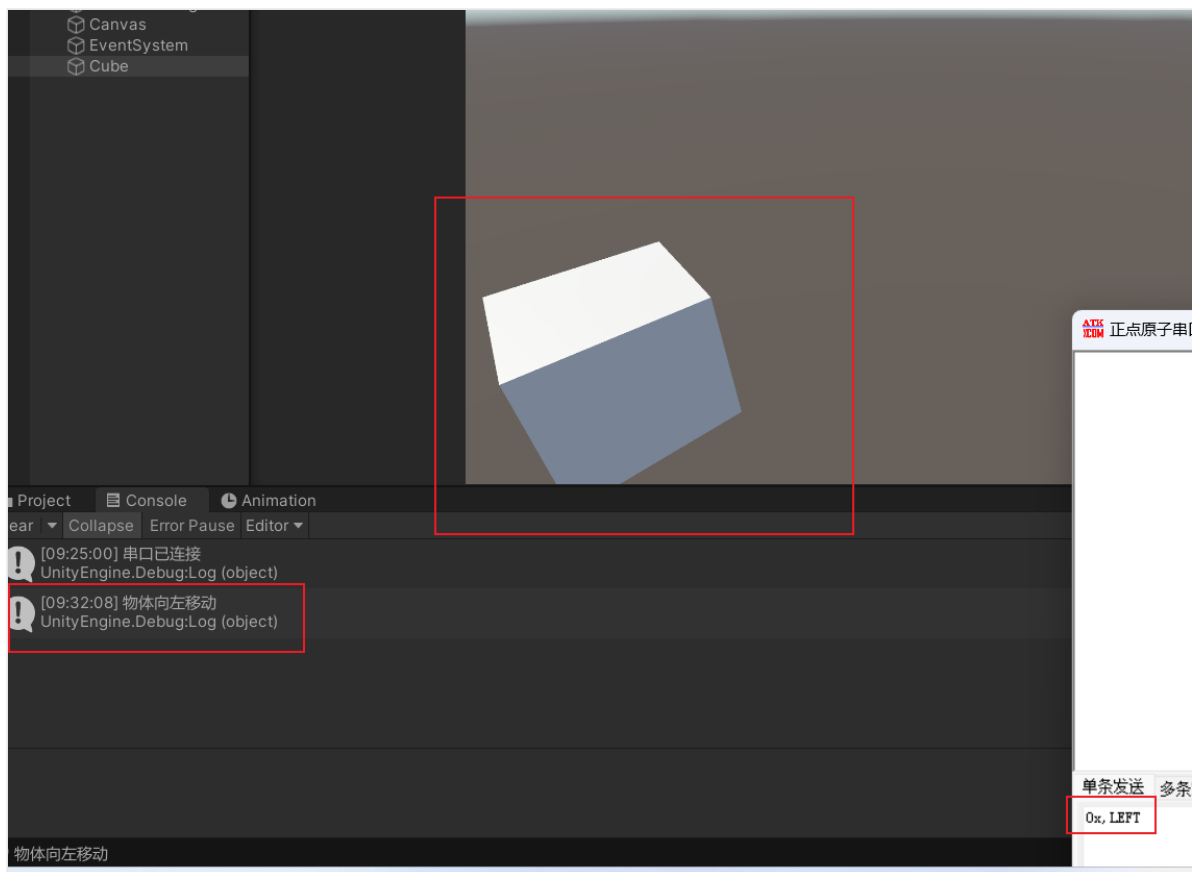
打开串口调试助手，串口参数设置如下：



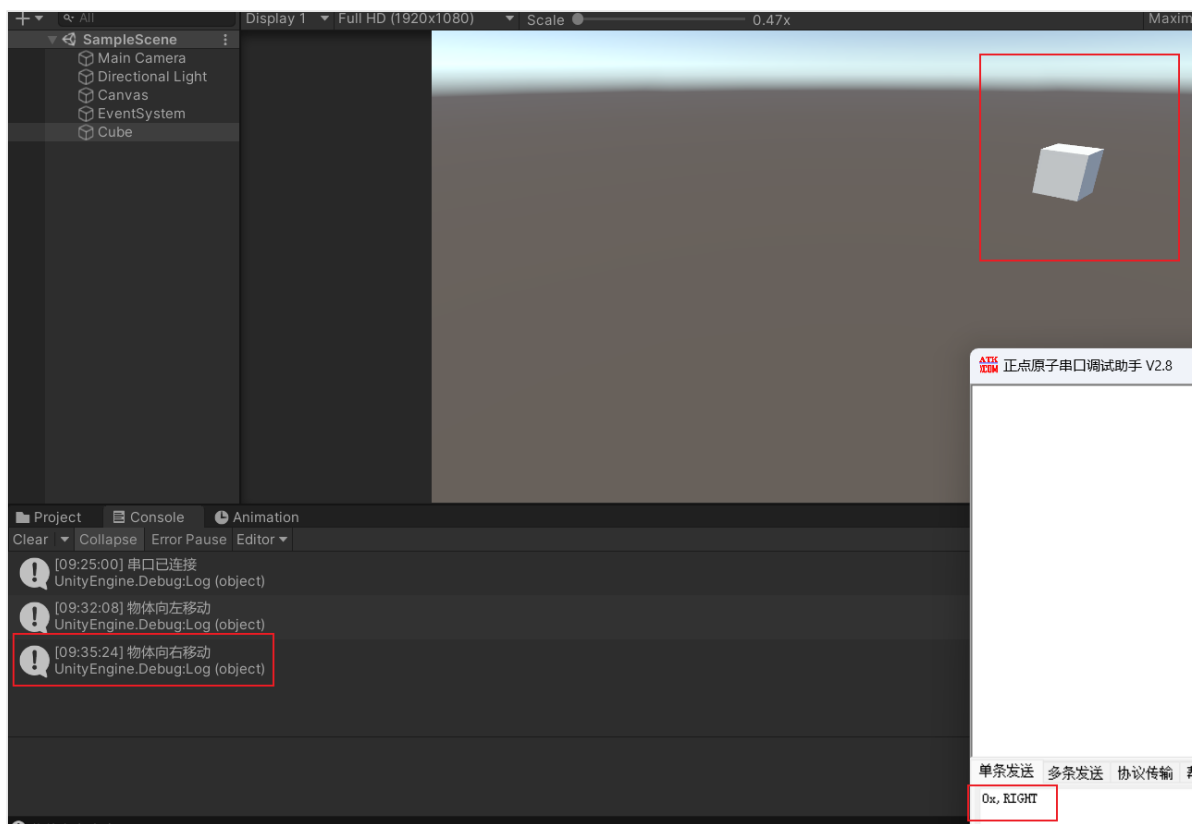
启动unity程序，控制台可以看到输出信息“串口已连接”



然后我们使用串口助手来模拟单片机与虚拟端进行通信，下图可以看到，串口助手给虚拟端发送“0x,LEFT”,方块便向左下角移动



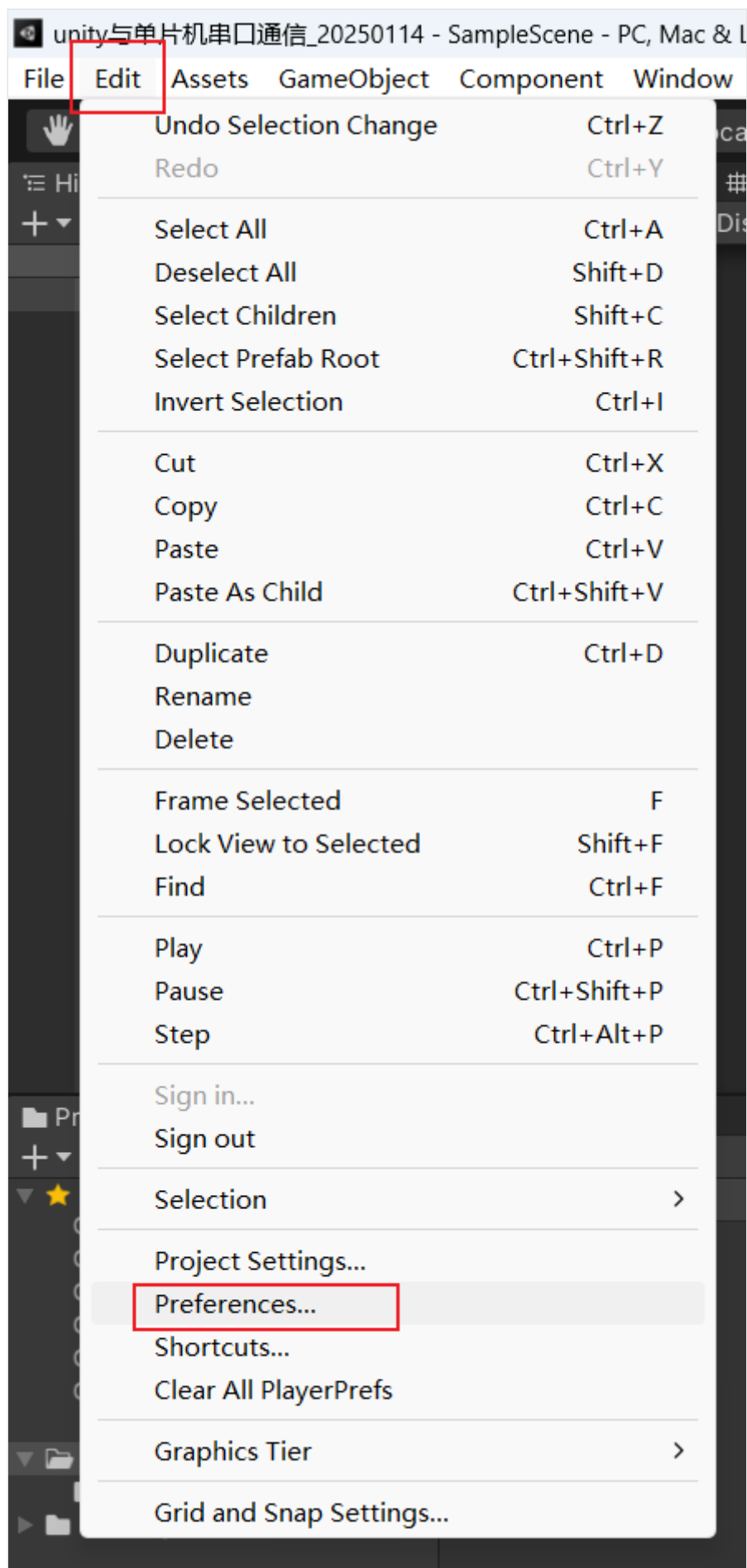
紧接着串口助手给虚拟端发送“0x,RIGHT”,可以看到，方块向右上角移动



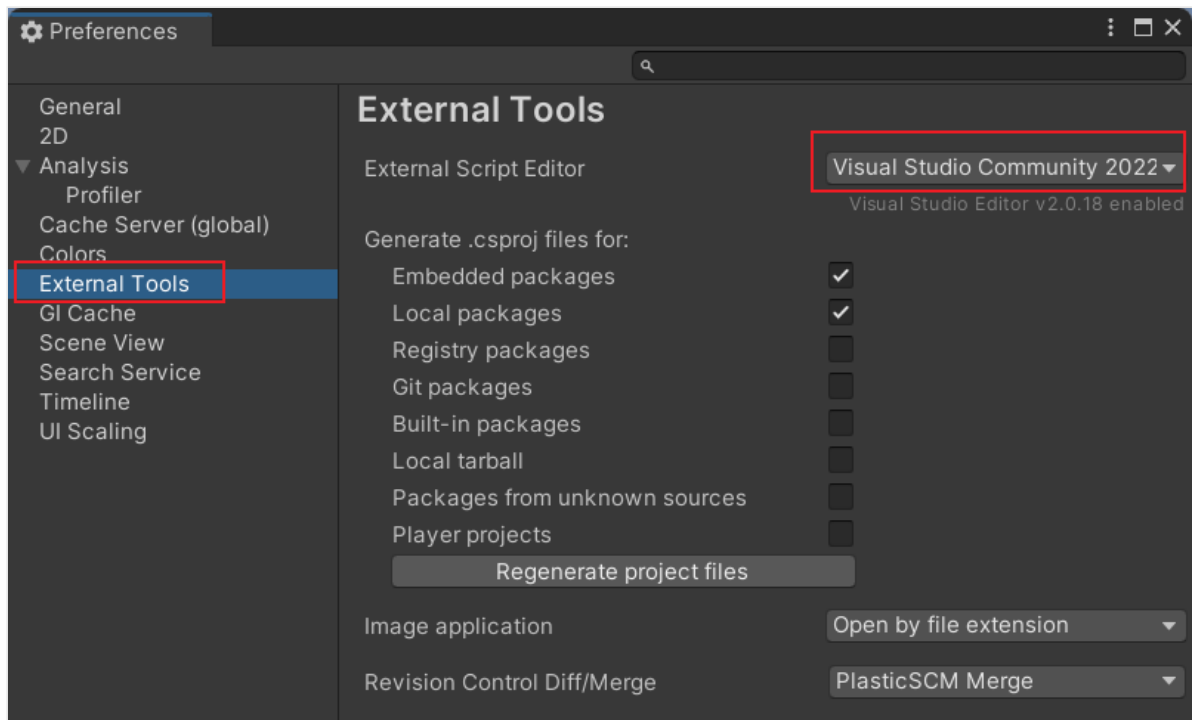
3.2. 注意事项

3.2.1. 在工程中设置默认的IDE编辑器

第一次的话，我们要在工程中设置默认的IDE编辑器，也就是我们前面下载的VS2022社区版



然后在“External Tools”选项中，选择VS2022社区版

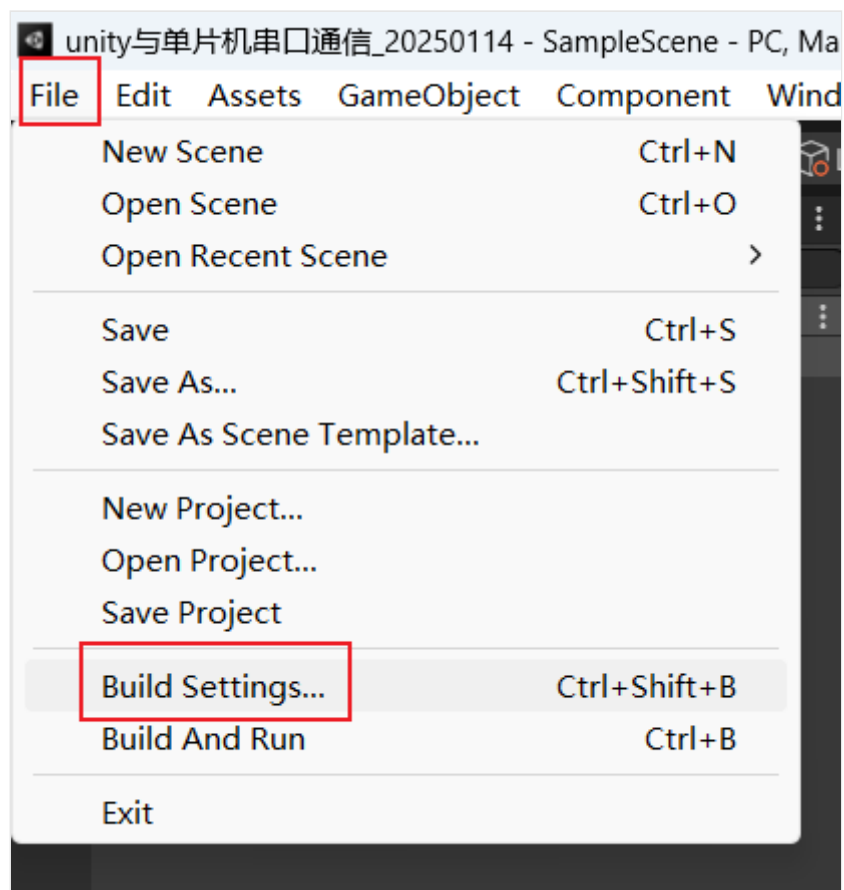


3.2.2. 设置项目API兼容级别

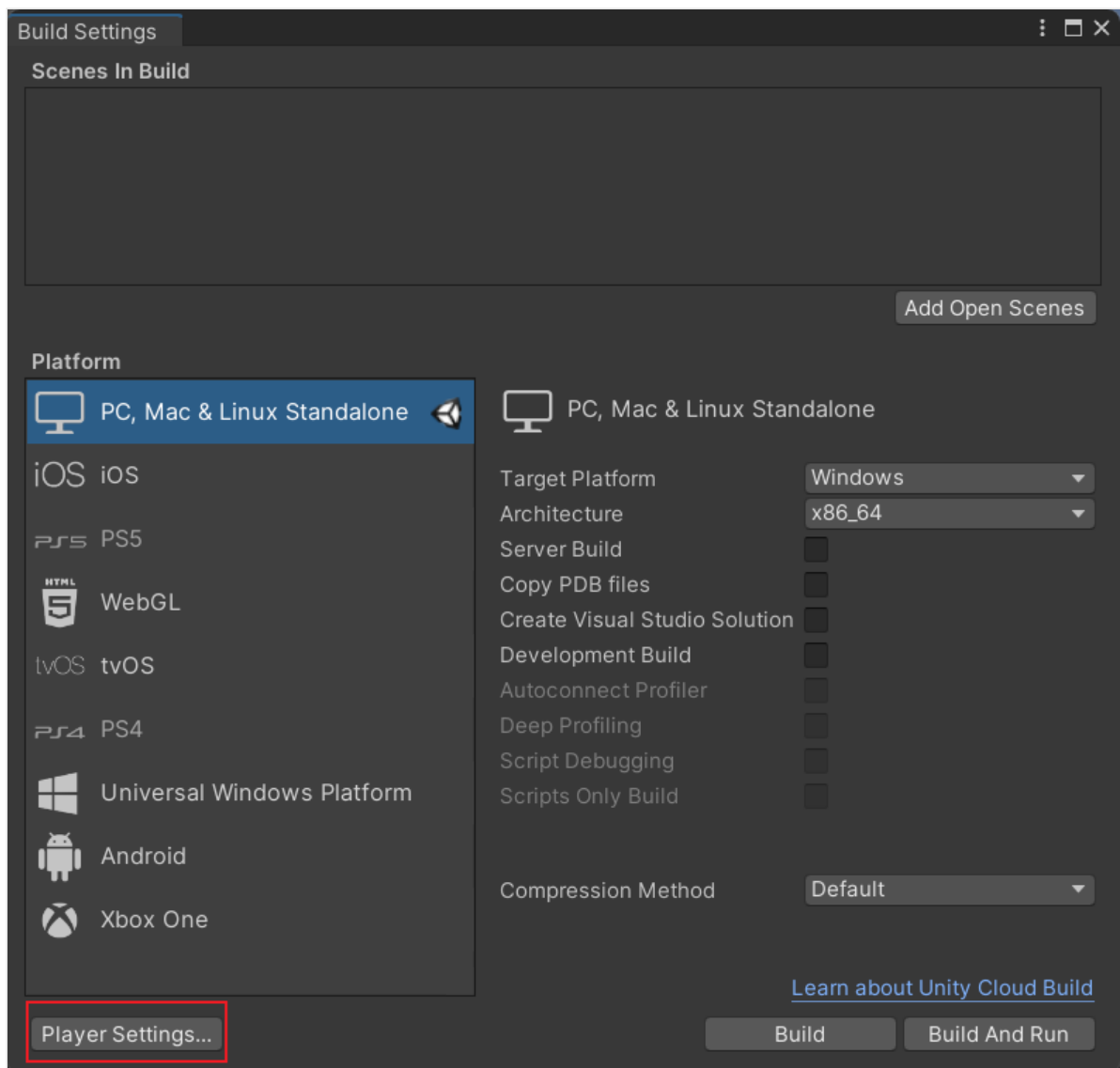
默认的unity项目，它的API兼容级别是是 **.NET Standard 2.0**，我们要把它改成**.NET 4.X**

不改成**.NET 4.X**的版本，很多API就无法使用

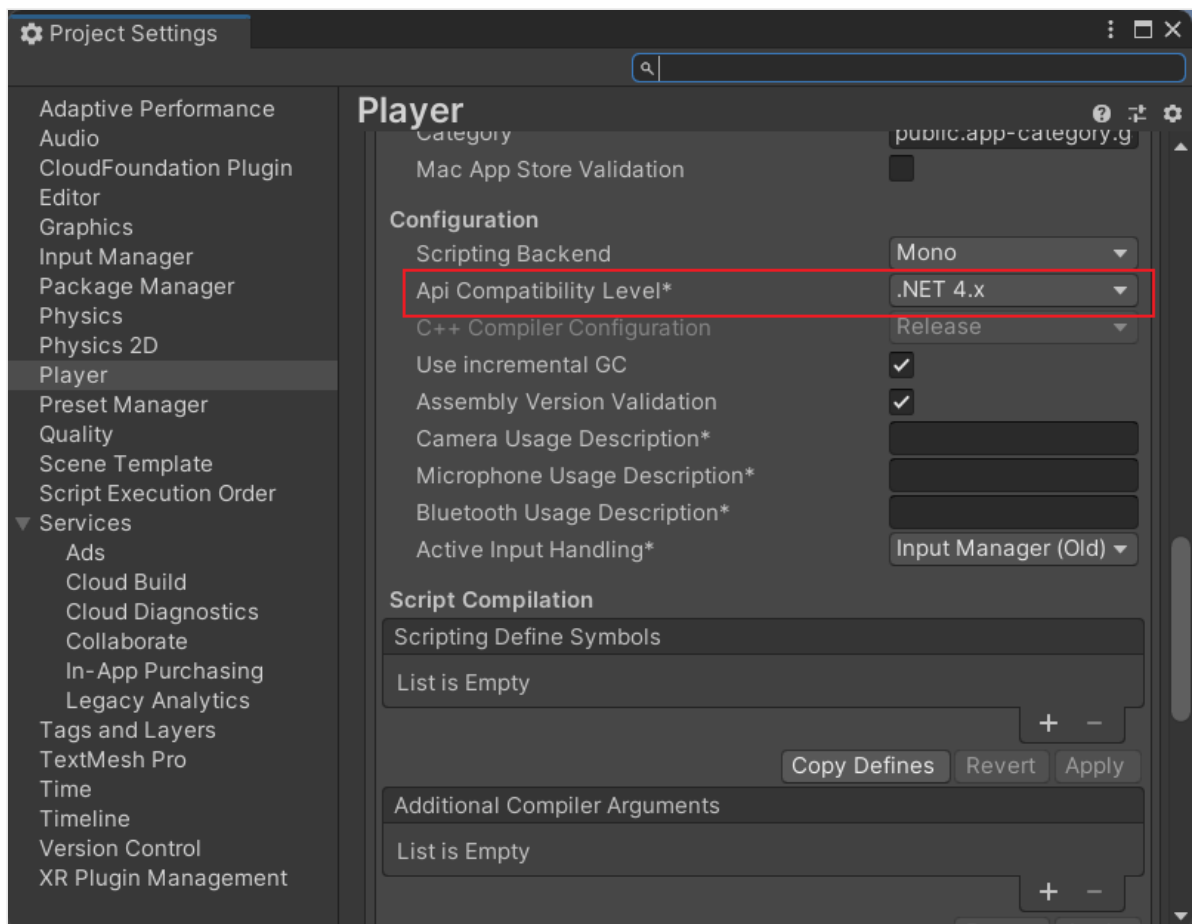
选择“Build Settings”



点击“Player Settings”



在这里，将API的兼容版本改为“**.NET 4.X**”



3.2.3. 安装串口驱动

如果使用STM32或者树莓派Uart串口，PC端需要安装串口驱动

直接去南京沁恒官网下载CH340驱动安装即可：

https://www.wch.cn/downloads/CH341SER_EXE.html

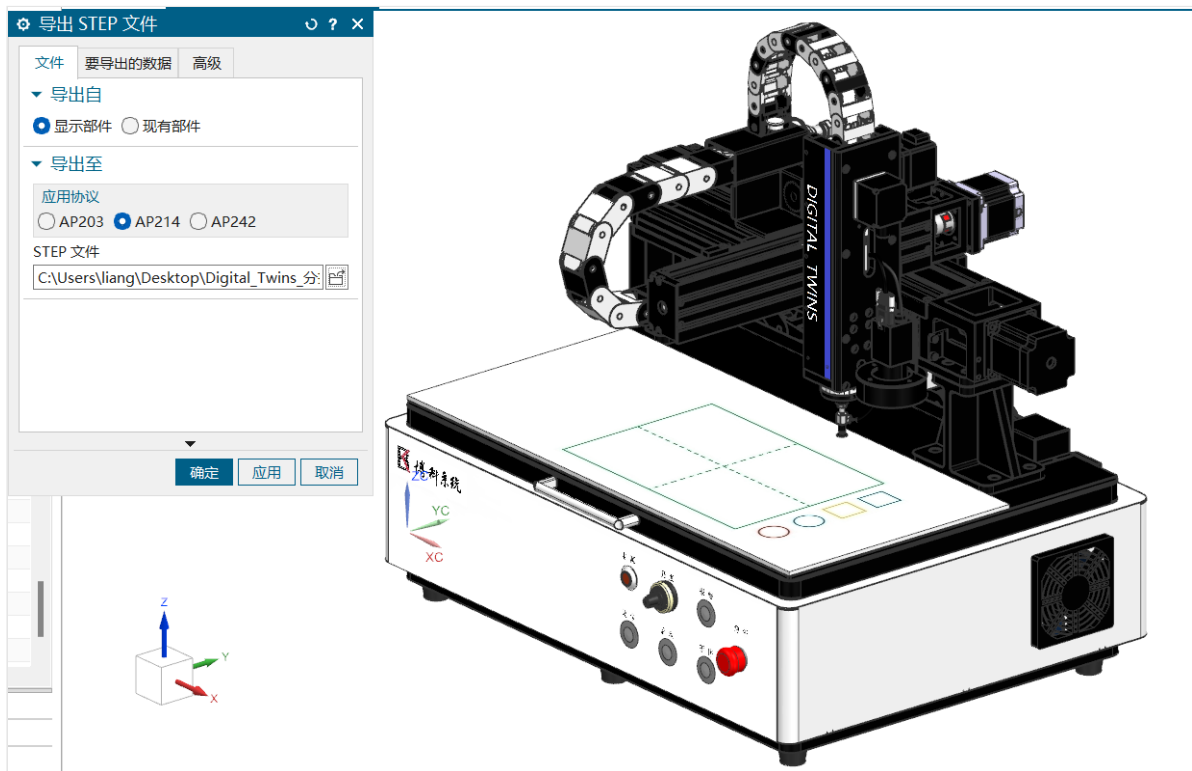


4. 综合示例1：基于复杂装备数字孪生系统实现连接与集成

4.1. 开发流程

4.1.1. 导出三维模型

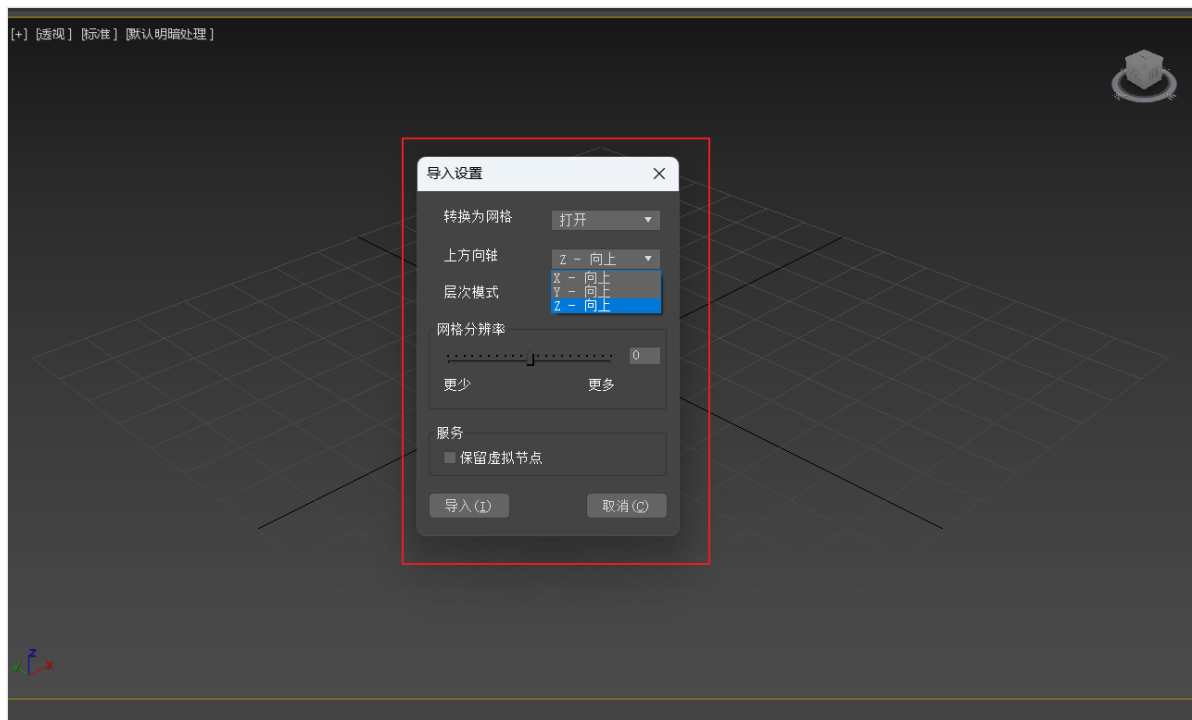
我们一般是使用solidworks或者UG NX进行了模型构建，为后续方便导入3D MAX或者MAYA进行预处理，要将三维模型导出成“STEP214”格式的文件



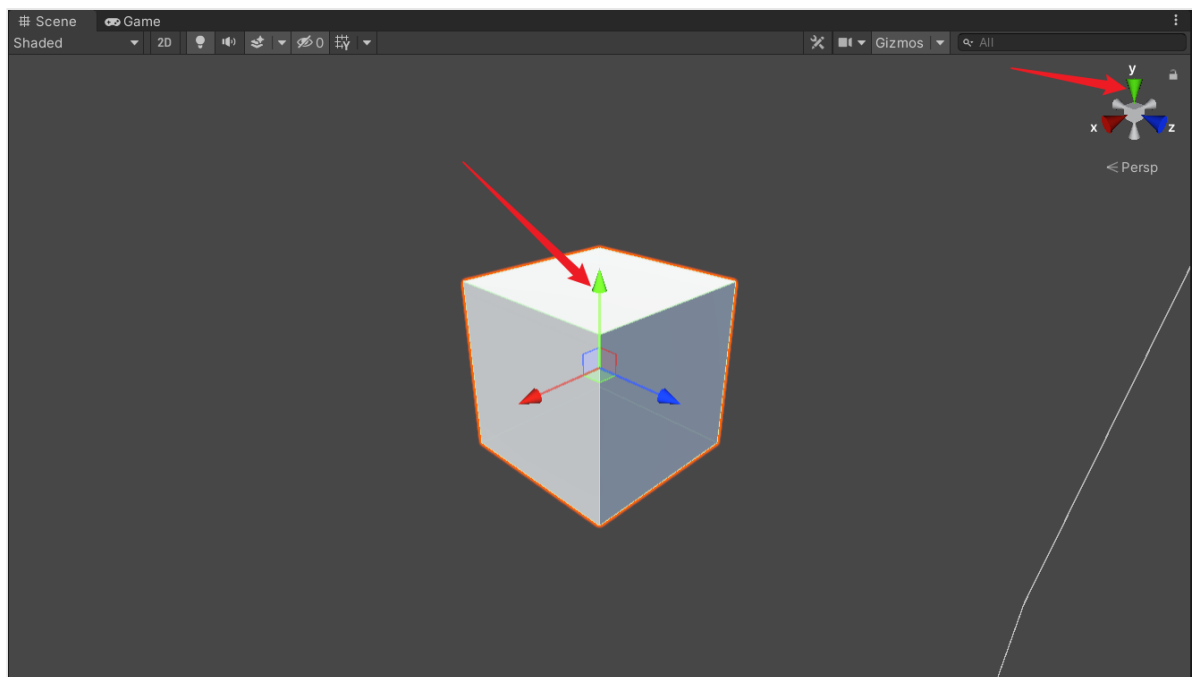
4.1.2. 导入到3D MAX进行预处理

3D MAX版本建议安装2020的版本!!!

直接点击导入前面一部导出的step文件，然后软件会提示这些参数如何设置，我们重点要看下选择哪个方向朝上



先打开unity，看看它的世界坐标系,我们可以很清楚的看到 在unity中，是Y轴朝上的

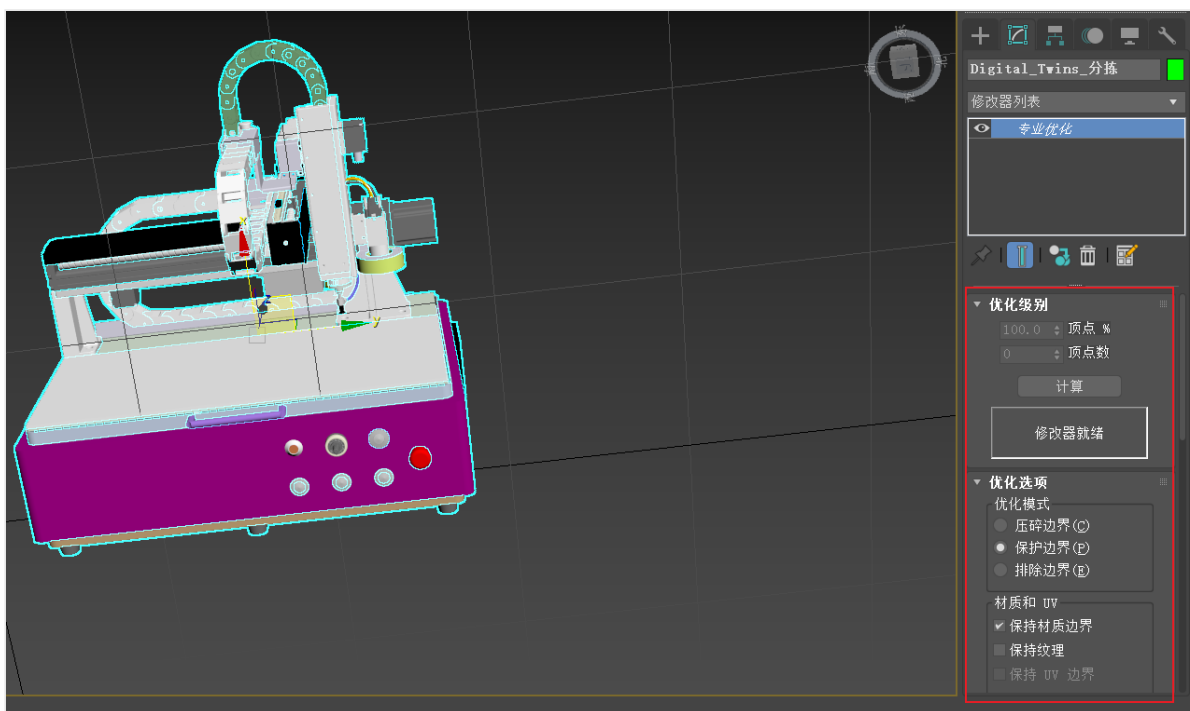


所以在导入进3d max时，为了保持坐标系方向一致性，设置“Y - 向上”



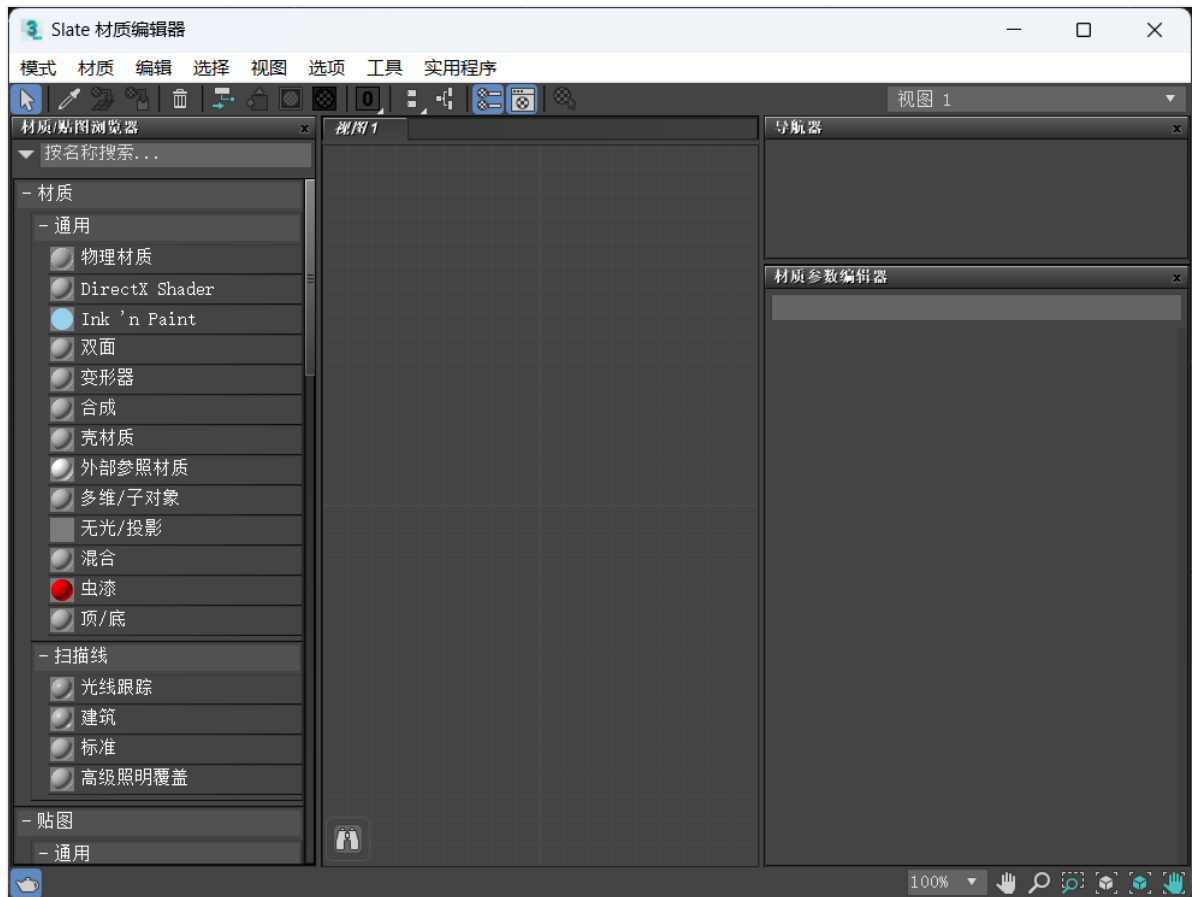
然后进行“**模型减面处理**”,做这一步主要是为了对模型进行轻量化

点击“**修改器列表**”，选择“**专业优化**”，可以设置要削减的面数和顶点数



然后我们可以在3d max中对模型进行材质的贴图和渲染（在unity中也可以对模型进行材质贴图）

在软件中，按下快捷键“M”，即可打开材质编辑器



然后就可以将**模型导出为FBX格式**(在unity中，处理的模型格式为FBX)

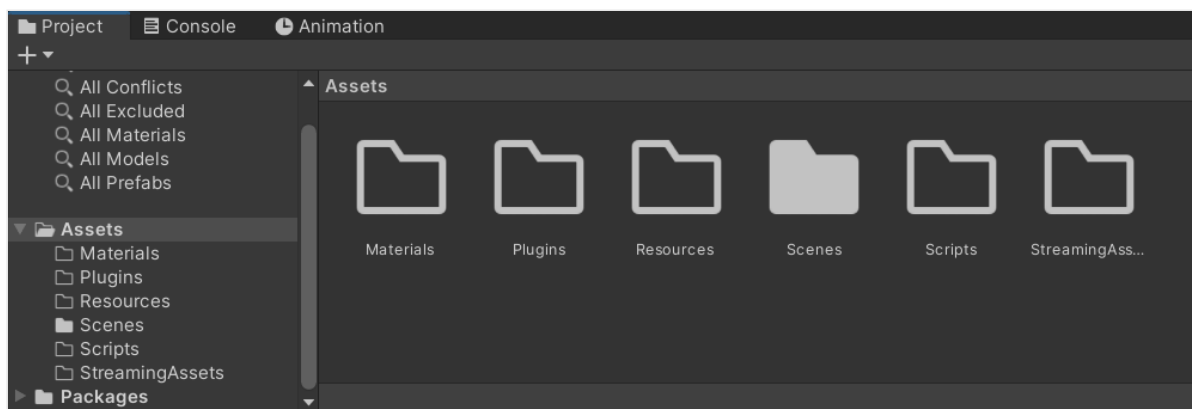
这里需要注意的是，**如果你在3d max中对模型进行了贴图和渲染，在导出FBX格式文件时，要勾选“嵌入的媒体”**（勾选此选项会将材质贴图嵌入到FBX文件中，方便导入到Unity时材质路径正确。否则，需要手动复制贴图文件到Unity项目中）



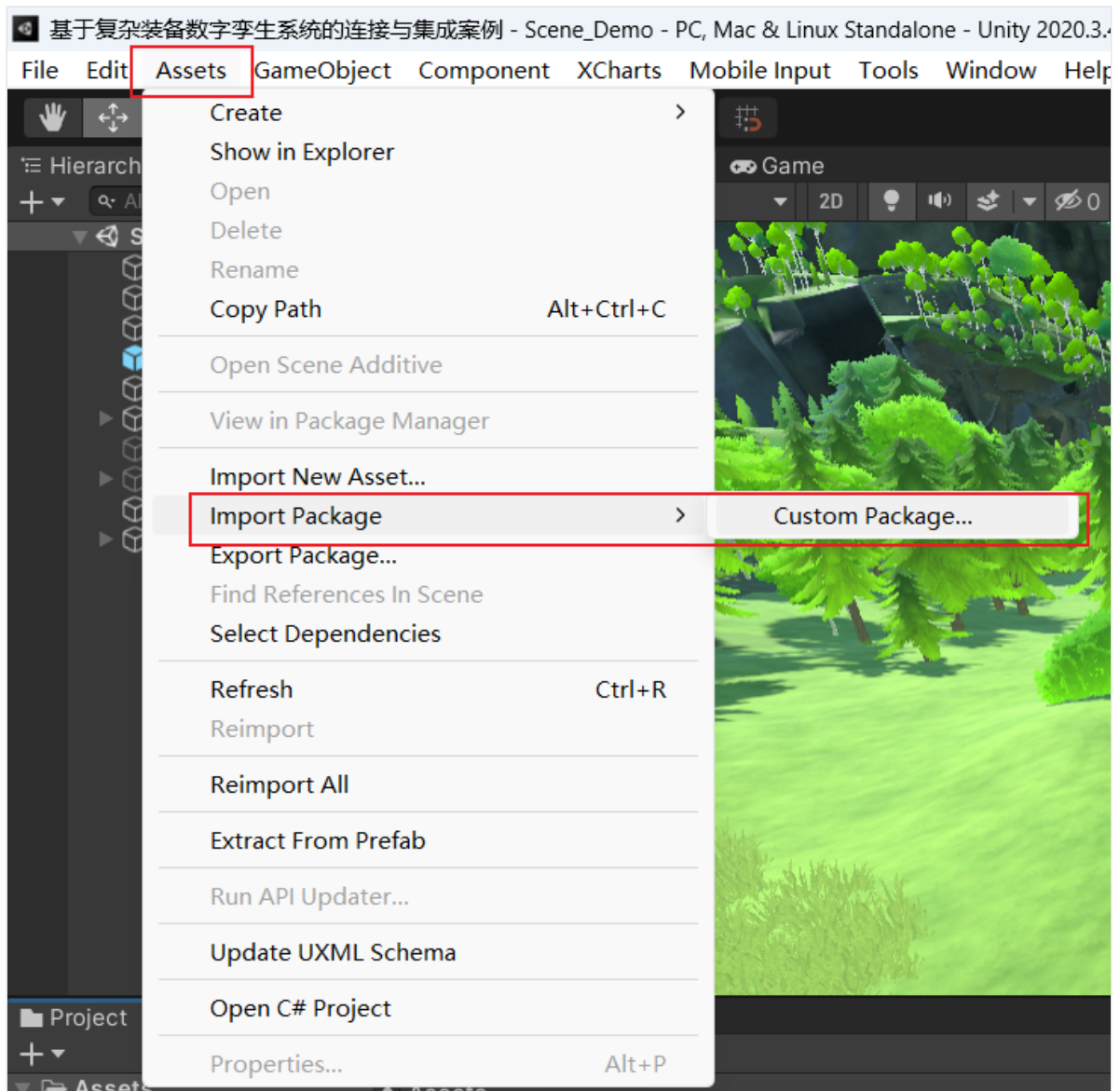
4.1.3. 新建工程导入FBX模型和场景

新建工程和前面一样不赘述

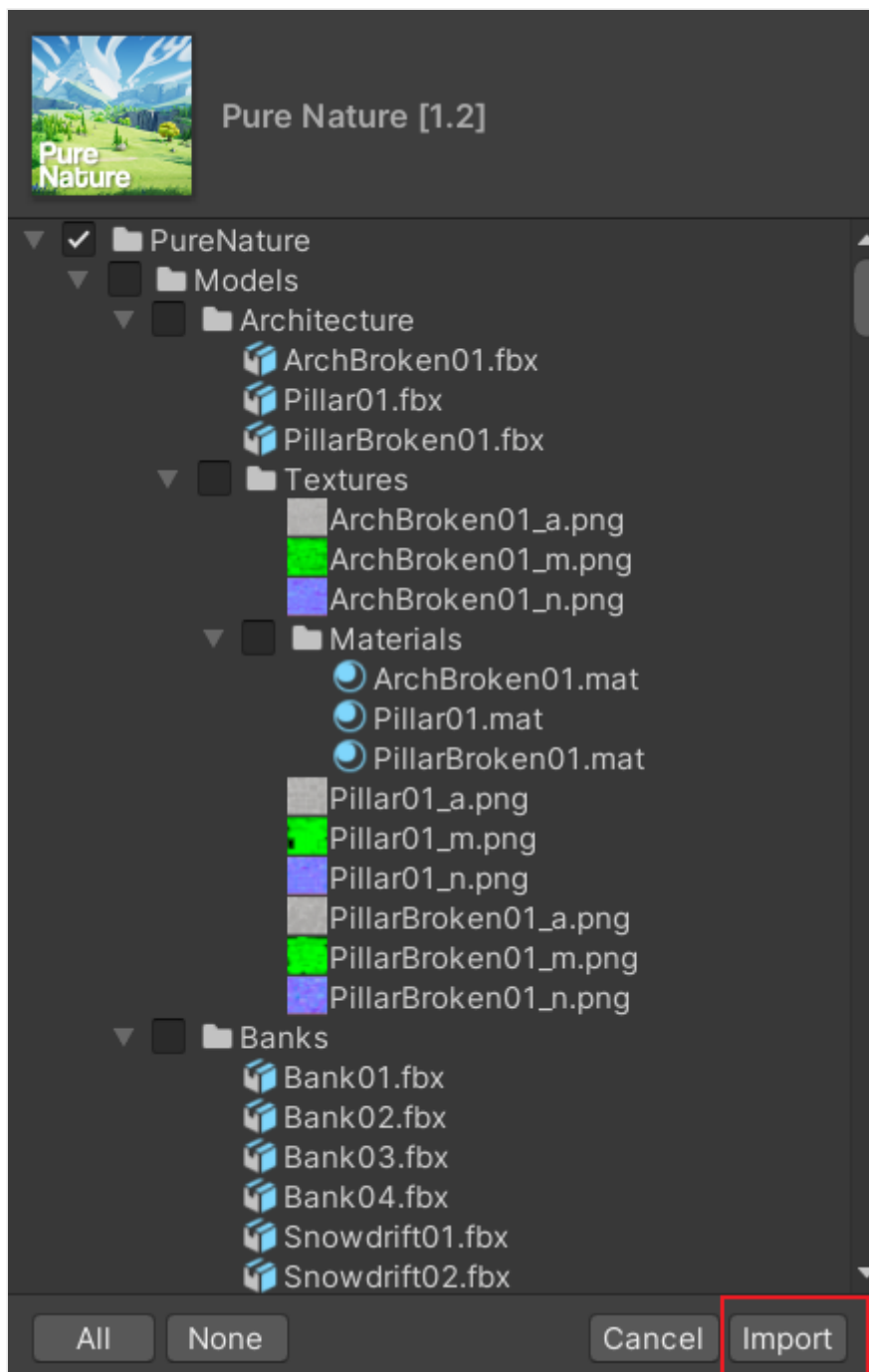
在新建工程的里面，依次添加文件夹



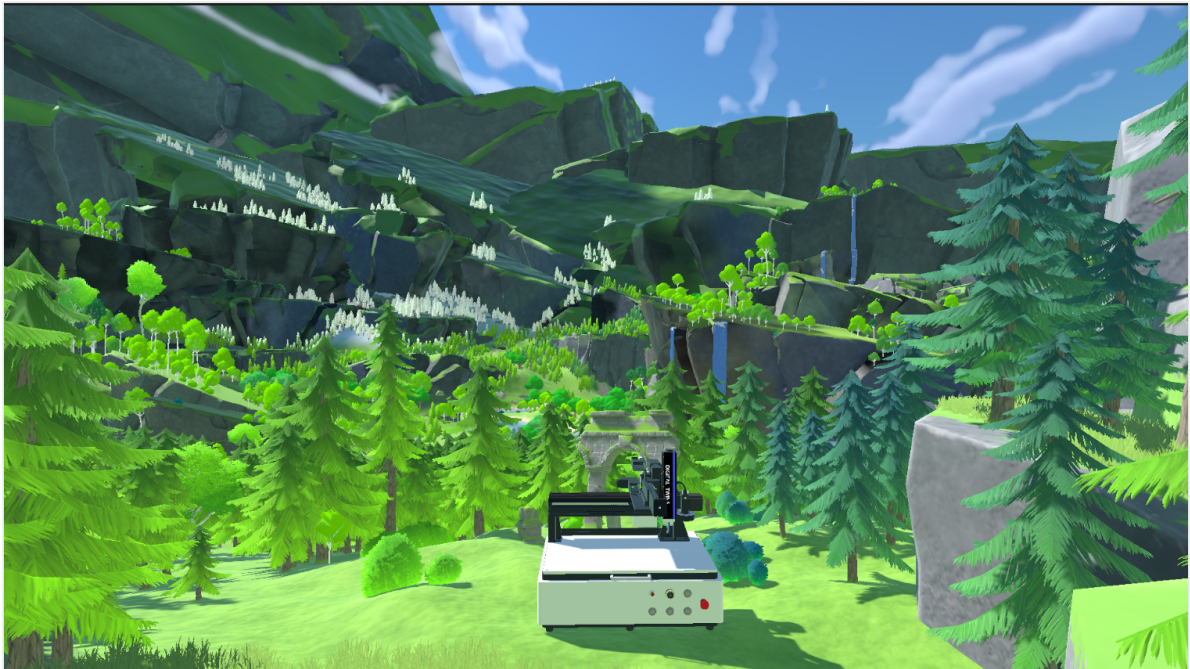
如果我们要使用外部场景的包，比如一个野外场景的包，



这是我从网上下载的一个野外场景的包,直接点击“Import”导入



最终的场景视图如图所示：

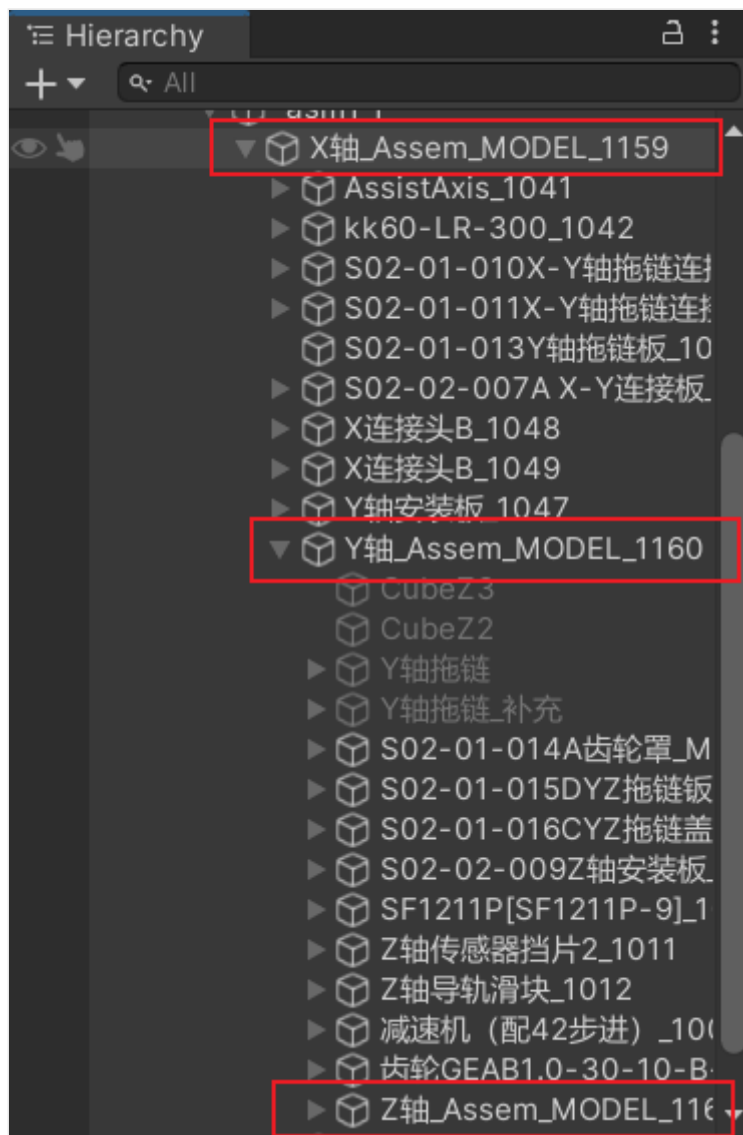


4.1.4. 调整所有零部件父子关系

我们知道 在一台设备中 有零件-部件-组件-整机这样的结构关系

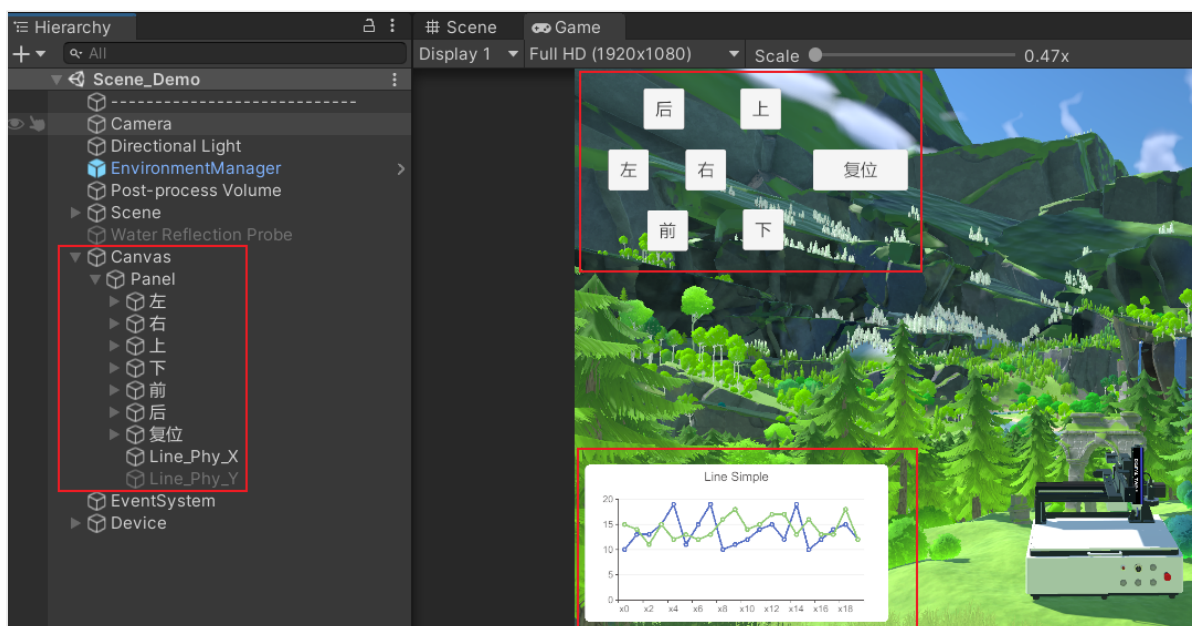
在unity中，我们需要手动调整零部件间的父子关系

当然了 在3d max去调整父子关系也是可以的 这个看个人操作意愿



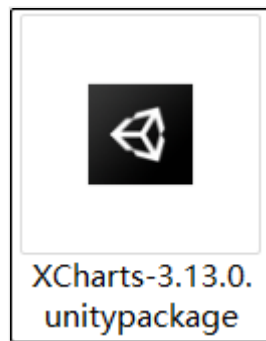
4.1.5. 添加UI面板

在Canvas面板添加按钮和图表等UI素材，UI的加入可以很方便我们进行虚实结合的双向交互调试

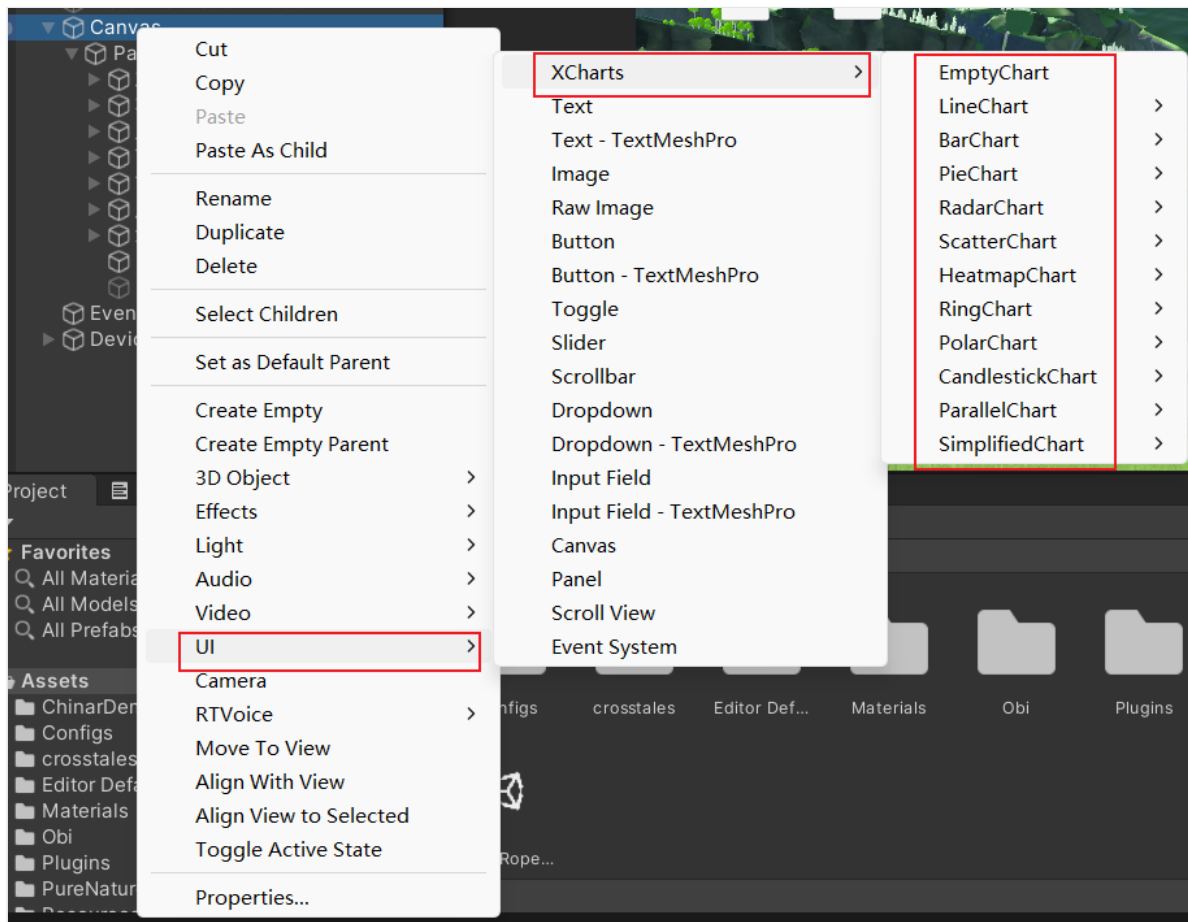


图表的包我是使用的XCharts-3.13.0，开源的包，免费使用

直接官网下载XCharts图表的包,导入进来就能用了



Import导入进来之后，直接就能在UI面板使用XCharts的图表



4.1.6. Plugins文件夹导入硬件驱动dll

在正式编写代码之前，将我们需要使用到的dll动态链接库放置到Plugins文件夹中，比如：

NModbus4.dll：和modbus协议通信的库，包含了modbus tcp和modbus rtu

S7.Net.dll: 和西门子硬件进行通信的动态库

sqlite3.dll: sqlite3数据库驱动

Newtonsoft.Json.dll: 一个处理json格式的动态库

4.1.7. 编写与控制器的通信代码

我这里使用的是西门子的1200，以此为例，

如果你用的树莓派或者单片机，思路是类似的

一定要先引用S7.Net.dll库

```
using UnityEngine;
using S7.Net;
namespace Common
{
    public static class S7_1200
    {
        public static readonly Plc plc = new
        Plc(CpuType.S71200, "192.168.1.10", 0, 1);
        static S7_1200()
        {
            try
            {
                //S7_1200.plc.Open();
            }
            catch (System.Exception ex)
            {
                //Debug.Log(ex);
                //S7_1200.plc.Open();
            }

            Debug.Log($"this is S7_1200");
        }
    }
}
```

我们通过访问PLC的DB块来拿到我们想要的数据,这是我们在1200自己定义的和Unity通信的DB块

Unity数据		名称	数据类型	偏移量	起始值
PLC 编程	1	Static			
	2	X轴当前位置	Real	0.0	0.0
	3	Y轴当前位置	Real	4.0	0.0
	4	Z轴当前位置	Real	8.0	0.0
	5	R轴当前位置	Real	12.0	0.0
	6	区域位置计数	DInt	16.0	0
	7	分拣完成状态	Real	20.0	0.0
	8	全部分拣完成	Bool	24.0	false
	9	视觉心跳	Bool	24.1	false
	10	系统初始化完成	Bool	24.2	false
	11	识别到的工件数	DInt	26.0	0
	12	X轴剩余	DInt	30.0	100
	13	Y轴剩余	DInt	34.0	100
	14	Z轴剩余	DInt	38.0	100
	15	R轴剩余	DInt	42.0	100
	16	A产品产量	DInt	46.0	0
	17	B产品产量	DInt	50.0	0

再新建一个cs类（单例）文件，和这个DB块的数据是一一对应的

```

public class ProductHelper
{
    #region 要获取的信息项
    public float xPosition_Axis { get; set; } //X轴实时位置
    public float yPosition_Axis { get; set; } //Y轴实时位置
    public float zPosition_Axis { get; set; } //Z轴实时位置
    public float rPosition_Axis { get; set; } //R轴实时位置

    public int AreaCount { set; get; } //区域数量
    public float SortState { set; get; } //分拣状态
    public bool IsOver { set; get; } //分拣是否完成
    public bool IsProduct { set; get; } //是否扫描到工件
    public bool IsResetOver { set; get; } //是否复位完成
    public int ProductCount { set; get; } //产品数量
    public int X_Remain { set; get; } //X轴剩余
    public int Y_Remain { set; get; } //Y轴剩余
    public int Z_Remain { set; get; } //Z轴剩余
    public int R_Remain { set; get; } //R轴剩余
    public int A_Count { set; get; } //A产品产量
    public int B_Count { set; get; } //B产品产量
    public int C_Count { set; get; } //C产品产量
    public int D_Count { set; get; } //D产品产量
    
```

置_1

速度

前里程

位

置_1

速度

前里程

位

置_1

速度

前里程

位

置_1

速度

前里程

```
public float X_Current_Pos { set; get; }//X轴当前位
```

```
public float X_Current_Speed { set; get; }//X轴当前
```

```
public float X_Current_Mileage { set; get; }//X轴当
```

```
public bool Is_X_Up_Limit { set; get; }//X轴上限位
```

```
public bool Is_X_Ref { set; get; }//X轴参考点
```

```
public bool Is_X_Down_Limit { set; get; }//X轴下限
```

```
public float Y_Current_Pos { set; get; }//Y轴当前位
```

```
public float Y_Current_Speed { set; get; }//Y轴当前
```

```
public float Y_Current_Mileage { set; get; }//Y轴当
```

```
public bool Is_Y_Up_Limit { set; get; }//Y轴上限位
```

```
public bool Is_Y_Ref { set; get; }//Y轴参考点
```

```
public bool Is_Y_Down_Limit { set; get; }//Y轴下限
```

```
public float Z_Current_Pos { set; get; }//Z轴当前位
```

```
public float Z_Current_Speed { set; get; }//Z轴当前
```

```
public float Z_Current_Mileage { set; get; }//Z轴当
```

```
public bool Is_Z_Up_Limit { set; get; }//Z轴上限位
```

```
public bool Is_Z_Ref { set; get; }//Z轴参考点
```

```
public bool Is_Z_Down_Limit { set; get; }//Z轴下限
```

```
public float R_Current_Pos { set; get; }//R轴当前位
```

```
public float R_Current_Speed { set; get; }//R轴当前
```

```
public float R_Current_Mileage { set; get; }//R轴当
```

```
public bool Is_R_Up_Limit { set; get; }//R轴上限位
```

```
public bool Is_R_Ref { set; get; }//R轴参考点
```

```

        public bool Is_R_Down_Limit { set; get; } //R轴下限
    位

    #endregion

    // 定义一个静态变量来保存类的实例
    private static ProductHelper uniqueInstance;
    // 定义一个标识确保线程同步
    private static readonly object locker = new
object();
    // 定义私有构造函数，使外界不能创建该类实例
    private ProductHelper()
    {

    }

    /// <summary>
    /// 定义公有方法提供一个全局访问点,同时你也可以定义公有属性
    来提供全局访问点
    /// </summary>
    /// <returns></returns>
    public static ProductHelper GetInstance()
    {
        // 当第一个线程运行到这里时，此时会对locker对象 "加
    锁",
        // 当第二个线程运行该方法时，首先检测到locker对象为"加
    锁"状态，该线程就会挂起等待第一个线程解锁
        // lock语句运行完之后（即线程运行完之后）会对象"解
    锁"

        if (uniqueInstance == null)
        {
            lock (locker)
            {
                // 如果类的实例不存在则创建，否则直接返回
                if (uniqueInstance == null)
                {
                    uniqueInstance = new
ProductHelper();
                }
            }
        }
    }
}

```

```

    }
    }
    return uniqueInstance;
}
}

```

编写好，我们直接在项目启动时，开启后台线程，实时从PLC的“Unity数据”DB块获取数据

```

Task.Run(async () =>
{
    while (cts.IsCancellationRequested == false)
    {
        if (Common.S7_1200.plc.IsConnected)
        {
            await
Common.S7_1200.plc.ReadClassAsync(ProductHelper.GetInstance(), 98, 0, cts.Token);
            x_CurrentPosition =
ProductHelper.GetInstance().xPosition_Axis / 100.0f;
            y_CurrentPosition =
ProductHelper.GetInstance().yPosition_Axis / 100.0f;
            z_CurrentPosition =
ProductHelper.GetInstance().zPosition_Axis / 100.0f;
            r_CurrentPosition =
ProductHelper.GetInstance().rPosition_Axis / 100.0f;
        }
        else
        {
            Common.S7_1200.plc.OpenAsync();
            await Task.Delay(500);
        }
    }
});

```

4.1.8. 编写和传感器的通信代码

如果传感器走的是标准modbus rtu协议(485硬件接口),可以很方便使用NModbus4.dll这个库来连接硬件

示例代码:

XXX.cs

```
using System.IO.Ports;
using Modbus.Device;

public class XXXX : MonoBehaviour
{
    //串口类
    private SerialPort serial = new SerialPort()
    {
        PortName = "COM5",
        DataBits = 8,
        ReadBufferSize = 128,
        BaudRate = 9600, //比特率
        Parity = Parity.None, //奇偶校验
        StopBits = StopBits.One, //停止位
        WriteTimeout = 1000,
        DtrEnable = false,
        RtsEnable = false,
        ReadTimeout = 1000
    };

    //串口主设备
    private ModbusSerialMaster serialMaster;

    void Start()
    {
        //创建串口主设备
        serialMaster =
ModbusSerialMaster.CreateRtu(serial);
        //开启后台线程,访问传感器获取数据
        Task.Run(async () =>
        {
            while (cts.IsCancellationRequested == false)
```

```

        {
            try
            {
                if (serial.IsOpen == false)
                {
                    //打开串口
                    serial.Open();
                }

                ushort[] temp2 = await
serialMaster.ReadHoldingRegistersAsync(0x51, 0x43, 0x03);
                vibration_value_Z = temp2[0] / 1.0;

                await Task.Delay(100);

                ushort[] temp1 = await
serialMaster.ReadHoldingRegistersAsync(0x50, 0x41, 0x03);
                vibration_value_X = temp1[1] / 1.0;

                await Task.Delay(100);
                ushort[] temp3 = await
serialMaster.ReadHoldingRegistersAsync(0x01, 0x06, 0x01);
                nosie_value = temp3[0] / 10.0;
            }
            catch (Exception)
            {
            }
        }
    });
}
}

```

代码中ReadHoldingRegistersAsync读取的是保持寄存器中的地址：

这个方法

第一个参数的含义是：传感器设备的地址

第二个参数的含义是：寄存器起始地址

第三个参数的含义是：寄存器个数

4.1.9. 实时渲染虚拟体

实时渲染虚拟体，主要是在FixedUpdate方法中，依靠我们前面获取到的运动状态，实时在虚拟端中，将物理端的运动状态给复现出来，核心就是调用Transform组件的Translate方法和Rotate方法

这里可以给一个伪代码的案例参考一下：

```
//下面是伪代码!!!
private void FixedUpdate()
{
    if (true)
    {
        //实时渲染虚拟端X轴模组的运动
        var goPosition = x_CurrentPosition -
x_LastPosition;
        x_Transform.Translate(new Vector3((float)
(goPosition / 1.0f), 0f));

        //实时渲染虚拟端Y轴模组的运动
        var goyPosition = y_CurrentPosition -
y_LastPosition;
        y_Transform.Translate(new Vector3(0f, -(float)
(goyPosition / 1.0f)));

        //实时渲染虚拟端Z轴模组的运动
        var gozPosition = z_CurrentPosition -
z_LastPosition;
        z_Transform.Translate(new Vector3(0f, 0f, -(float)
(gozPosition / 1.0f)));

        //实时渲染虚拟端R轴模组的运动
        var gorPosition = r_CurrentPosition -
r_LastPosition;
```

```

        r_Transform.Rotate(new Vector3(0, -(float)
(gorPosition / 1.0), 0));
    }

}

```

4.1.10. 数据的UI实时显示

UI主要是图表的实时刷新，我直接基于XCharts开源的案例，更改了一下代码：

Line_PHY_X.cs

```

using Assets.Scripts.Utils;
using System.Collections;
using UnityEngine;
#if INPUT_SYSTEM_ENABLED
using Input = XCharts.Runtime.InputHelper;
#endif
using XCharts.Runtime;
//[DisallowMultipleComponent]
//[ExecuteInEditMode]
public class Line_PHY_X : MonoBehaviour
{
    private LineChart chart;

    void Awake()
    {

    }

    private void Start()
    {
        Init();
        StartCoroutine(UpdatePos());
    }

    private IEnumerator UpdatePos()
    {

```

```

        while (true)
        {
            chart.RemoveData();
            chart.AddSerie<Line>();
            for (int i = 0; i < ModelData.Phy_X.Count;
i++)
            {
                chart.AddXAxisData("x" + i);
                chart.AddData(0, ModelData.Phy_X[i]);
            }
            ModelData.Phy_X.Clear();
            yield return new WaitForSeconds(1f);
        }
    }

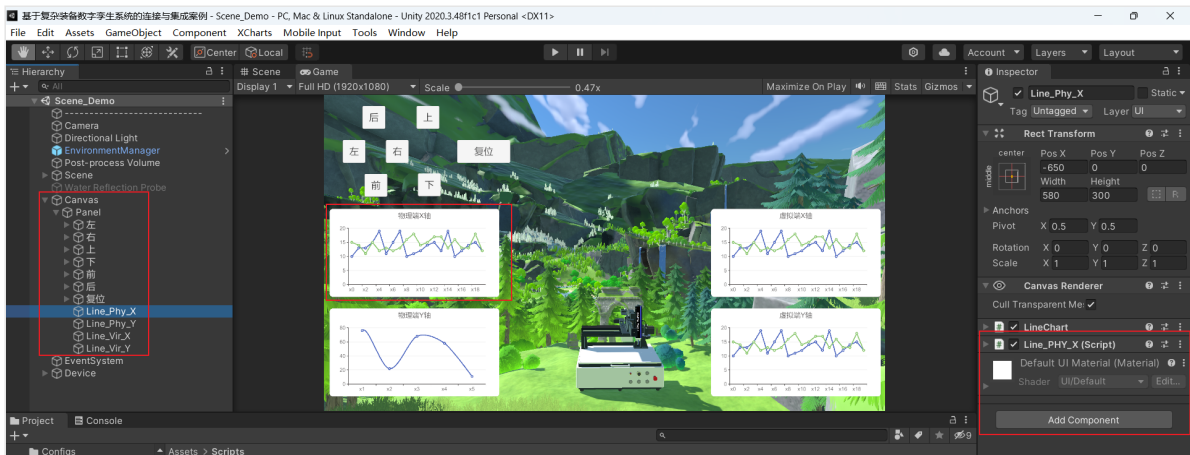
    void Init()
    {
        chart = gameObject.GetComponent<LineChart>();
        if (chart == null)
        {
            chart = gameObject.AddComponent<LineChart>();
            chart.Init();
        }
        chart.EnsureChartComponent<Title>().show = true;
        chart.EnsureChartComponent<Title>().text = "物理端X
轴";

        chart.EnsureChartComponent<Tooltip>().show = true;
        chart.EnsureChartComponent<Legend>().show = false;

        var xAxis = chart.EnsureChartComponent<XAxis>();
        var yAxis = chart.EnsureChartComponent<YAxis>();
        xAxis.show = true;
        yAxis.show = true;
        xAxis.type = Axis.AxisType.Category;
        yAxis.type = Axis.AxisType.Value;

        xAxis.splitNumber = 10;
        xAxis.boundaryGap = true;
    }
}

```



直接在Canvas面板创建图表，并且新建脚本Line_PHY_X.cs挂载上去，用于实时更新显示物理端X轴模组的数据

我这里要实时刷新4张图表，所以有4个独立的Line_XXX_X.cs脚本文件，代码都是大同小异

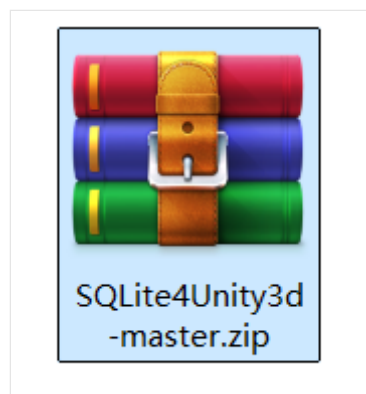
4.1.11. 数据实时保存到数据库

项目中我使用的数据库是sqlite3

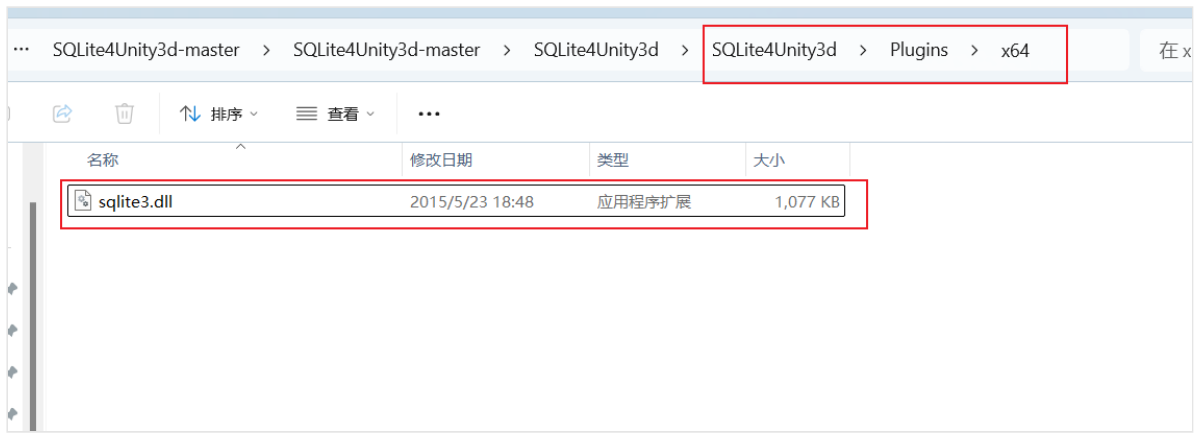
主要是sqlite这个数据库是免驱动的，移植起来很方便，而且体积很小

当然,unity也可以连接mysql和sql server数据库，看个人选择

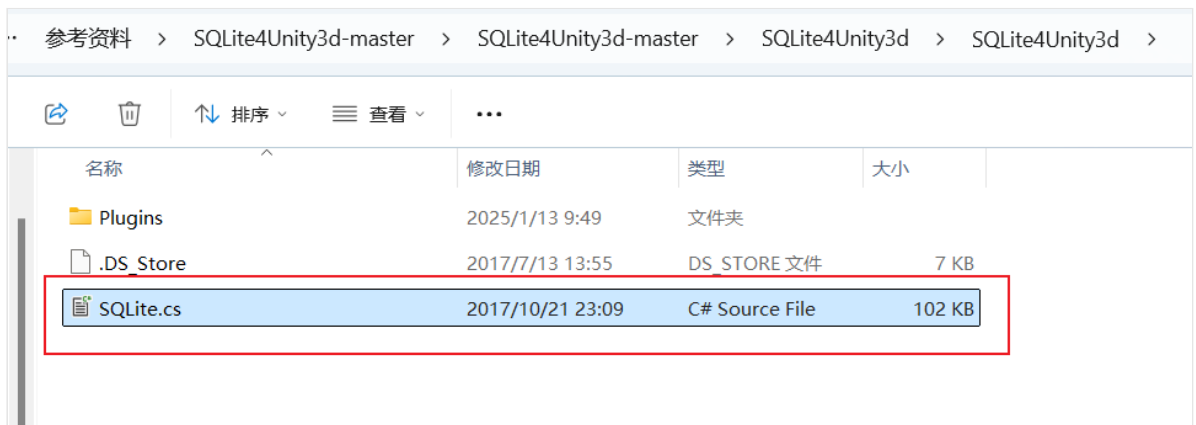
网络上直接由针对unity的开源sqlite数据库版本(名称就叫SQLite4Unity3d)，直接github下载（后面我也会打包到资料中去）



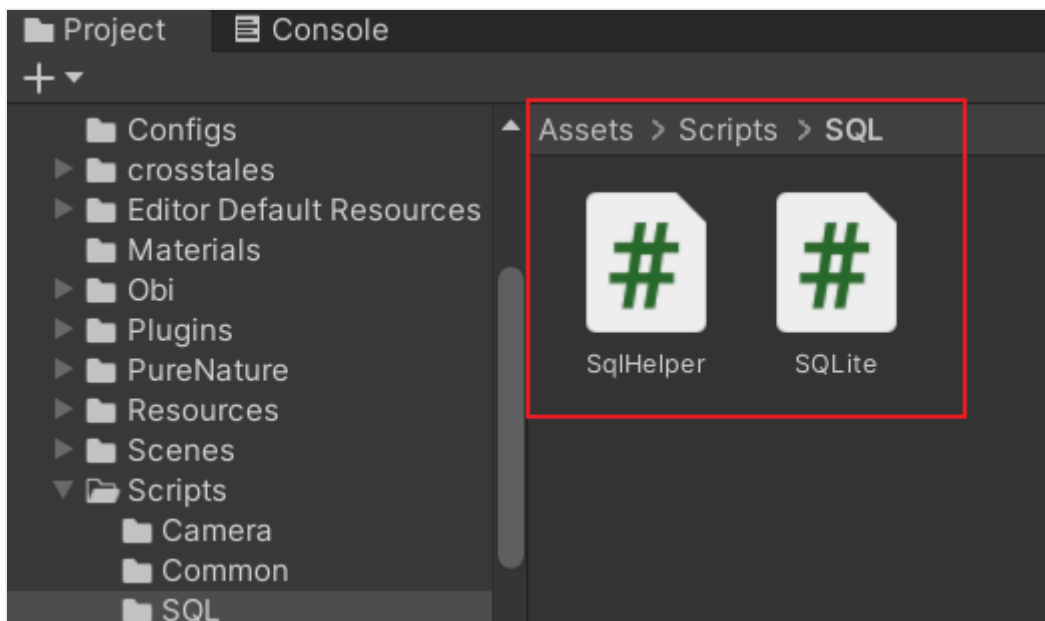
首先将sqlite3.dll这个库拷贝到unity的Plugins文件夹里面去



然后将SQLite.cs这个文件拷贝到Scripts文件夹里面去,这个SQLite.cs就是作者给我们封装好的去操作sqlite数据库的API文件



这是我放置的目录结构，数据库的SQL操作脚本，我都放置在一个文件夹里面



在原有API的基础上，我创建了一个类文件来完成基本的操作

SqlHelper.cs

```

using SQLite4Unity3d;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;
using static SQLite4Unity3d.SQLiteConnection;
namespace Assets.Scripts.SQL
{
    /// <summary>
    /// Sql脚本帮助类
    /// </summary>
    public class SqlHelper
    {
        /// <summary>
        /// 数据库连接
        /// </summary>
        public static SQLiteConnection Connection;
        //数据库存放路径以及名称
        private static string databasePath =
Application.streamingAssetsPath + "/digital_database.db";

        //私有构造方法，在构造方法中直接创建数据库
digital_database.db
        static SqlHelper()
        {
            Connection = new
SQLiteConnection(databasePath, SQLiteOpenFlags.ReadWrite |
SQLiteOpenFlags.Create);
        }

        //创建数据库的静态方法
        public static void CreateDb()
        {
            //参数1.数据库地址，一般放在StreamingAssets文件夹
            //中，2.开启读写和创建数据库权限
            Connection = new
SQLiteConnection(databasePath, SQLiteOpenFlags.ReadWrite |
SQLiteOpenFlags.Create);
        }
    }
}

```

```

        //创建表的静态方法
        public static void CreateTable<T>()
        {
            Connection = new
SQLiteConnection(databasePath, SQLiteOpenFlags.ReadWrite |
SQLiteOpenFlags.Create);
            Connection.CreateTable<T>(); //创建表
        }

        //向数据库中插入数据：多条数据插入
        public static void
InsertData(System.Collections.IEnumerable objects)
        {
            Connection.InsertAll(objects); //插入一个数组，
数组里是多条数据
        }

        //向数据库中插入数据：单条数据插入
        public static void InsertData(object obj)
        {
            Connection.Insert(obj);
        }
    }

    /// <summary>
    /// 虚拟端表：记录存放虚拟端的数据
    /// </summary>
    public class T_VirtualPos
    {
        [PrimaryKey, AutoIncrement] //设置主键 自动增长
        public int Id { get; set; } //Id作为主键
        public float VirtualPosX { get; set; } //虚拟端X轴模
组位置
        public float VirtualPosY { get; set; } //虚拟端Y轴模
组位置
        public float VirtualPosZ { get; set; } //虚拟端Z轴模
组位置
    }

```

```

        public float VirtualPosR { get; set; } //虚拟端R轴模
组位置

        public string Time { get; set; } //时间
    }

    /// <summary>
    /// 物理端表：记录存放物理端的数据
    /// </summary>
    public class T_PhyPos
    {
        [PrimaryKey, AutoIncrement] //设置主键 自动增长
        public int Id { get; set; } //Id作为主键
        public float PhyPosX { get; set; } //物理端X轴模组位置
        public float PhyPosY { get; set; } //物理端Y轴模组位置
        public float PhyPosZ { get; set; } //物理端Z轴模组位置
        public float PhyPosR { get; set; } //物理端R轴模组位置
        public string Time { get; set; } //时间
    }

    /// <summary>
    /// Sensor表：用来存放传感器的数据
    /// </summary>
    public class T_SensorData
    {
        [PrimaryKey, AutoIncrement] //设置主键 自动增长
        public int Id { get; set; } //Id作为主键

        public float SensorData1 { get; set; } //传感器数据1
        public float SensorData2 { get; set; } //传感器数据2
        public float SensorData3 { get; set; } //传感器数据3
        .....
        public string Time { get; set; } //时间
    }
}

```

在项目启动的时候，我们直接开启后台线程，更新数据库

```
#region 数据库线程更新
```

```

Task.Run(async () =>
{
    SqlHelper.CreateTable<T_VirtualPos>();
    SqlHelper.CreateTable<T_PhyPos>();
    while (cts.IsCancellationRequested == false)
    {
        await Task.Delay(100);
        SqlHelper.InsertData(new T_PhyPos() {

            PhyPosX =
ProductHelper.GetInstance().xPosition_Axis,
            PhyPosY =
ProductHelper.GetInstance().yPosition_Axis,
            PhyPosZ =
ProductHelper.GetInstance().zPosition_Axis,
            PhyPosR =
ProductHelper.GetInstance().rPosition_Axis,
            Time = DateTime.Now.ToString() });
        SqlHelper.InsertData(new T_VirtualPos()
        {
            VirtualPosX = virX,
            VirtualPosY = virY,
            VirtualPosZ = virZ,
            VirtualPosR = virR,
            Time = DateTime.Now.ToString()
        });
    }
});
#endregion

```

4.2. 注意事项

4.2.1. unity中关于虚拟体渲染和UI界面渲染的若干问题(见下)

4.2.2. 线程不能参与虚拟端的渲染与交互！！！！

在unity中，线程是不能够参与虚拟端的交互的，也就是说，在线程中，不能出现和unity游戏引擎相关的业务逻辑！！！！

要处理unity游戏引擎相关的业务逻辑,要么依靠脚本自带的生命周期方法,例如 Start Update FixedUpdate,要么依靠协程来处理关于unity的业务逻辑

错误示例：

```
Task.Run(async () =>
{
    while (cts.IsCancellationRequested == false)
    {
        virX = x_Transform.localPosition.x;
        virY = y_Transform.localPosition.y;
        virZ = z_Transform.localPosition.z;
        virR = r_Transform.localPosition.z;
    }
});
```

//这个错误示例 在于 在Task的后台线程，我们是不能够直接获取虚拟端运动模組的运动数据的，原因也讲了：在后台线程中，不能出现和unity游戏引擎相关的业务逻辑！！！！

我们要实时获取虚拟端运动模組的位置数据，这就涉及到了和unity相关的业务逻辑，这样的交互，要使用脚本自带的生命周期方法，或者使用协程(协程可以理解为是线程的子线程，虽然这样讲不严谨)

正确示例：

```
//开启协程方法
StartCoroutine(GetVirPos());
//定义的协程方法
private IEnumerator GetVirPos()
{
    while (cts.IsCancellationRequested == false)
```

```

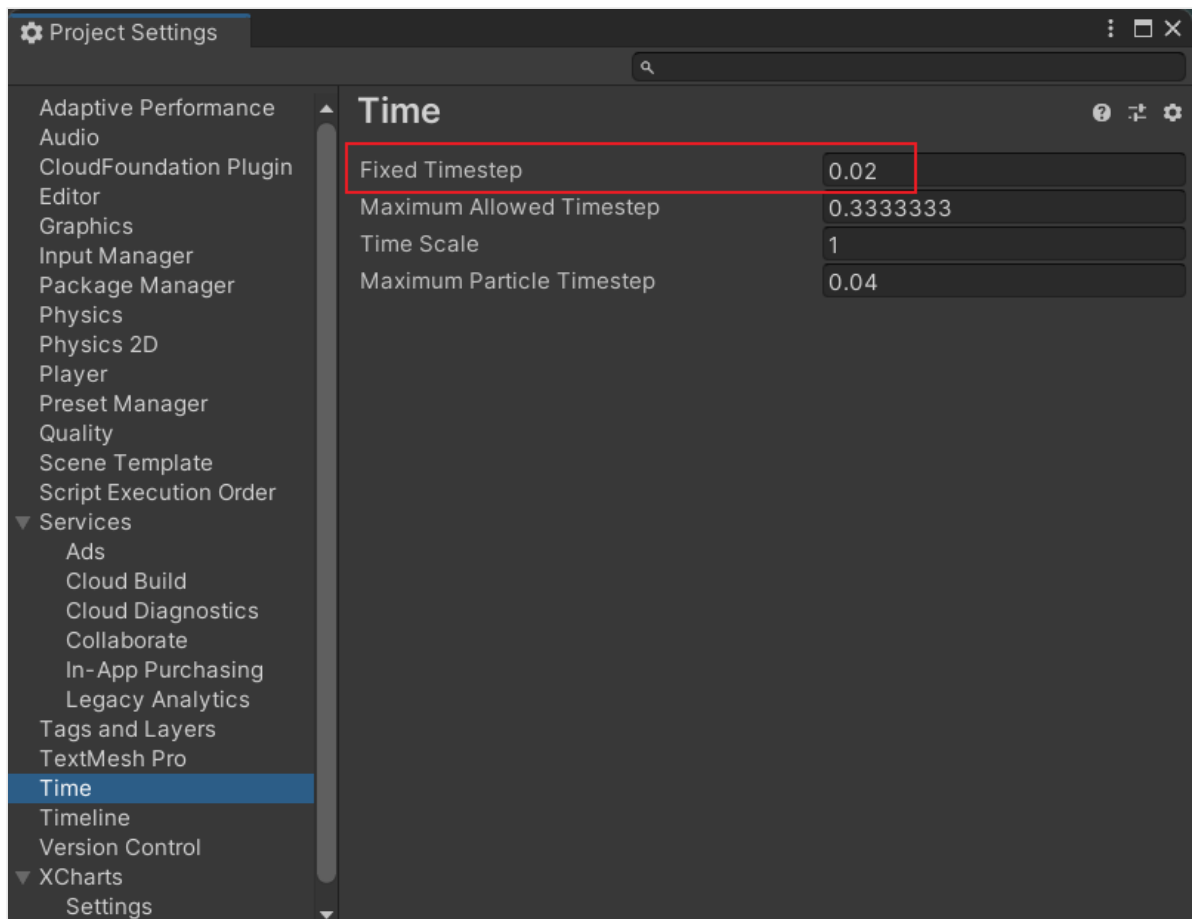
{
    //获取虚拟端X轴模组的数据
    virX =x_Transform.localPosition.x;
    //获取虚拟端Y轴模组的数据
    virY =y_Transform.localPosition.y,
    //获取虚拟端Z轴模组的数据
    virZ =z_Transform.localPosition.z;
    //获取虚拟端R轴模组的数据
    virR =r_Transform.localPosition.z;
    //通过迭代器关键字yield返回 实现1s读取1次虚拟端模组运
    动数据的效果
    yield return new WaitForSeconds(1f);
}
}

```

4.2.3. 虚拟体如何渲染

虚拟体的渲染，只推荐一种方式：**在脚本的生命周期方法FixedUpdate中去刷新虚拟体**

在unity中,FixedUpdate的刷新周期默认是0.02s,也就是挂载的脚本1s执行50次FixedUpdate方法



在unity中，凡是涉及到虚拟端对象会运动的状况，统一到FixedUpdate这个方法中去渲染

```
//下面是伪代码!!!
private void FixedUpdate()
{
    if (true)
    {
        //实时渲染虚拟端X轴模组的运动
        var goPosition = x_CurrentPosition -
x_LastPosition;
        x_Transform.Translate(new Vector3((float)
(goPosition / 1.0), 0));

        //实时渲染虚拟端Y轴模组的运动
        var goyPosition = y_CurrentPosition -
y_LastPosition;
        y_Transform.Translate(new Vector3(0, -(float)
(goyPosition / 1.0)));

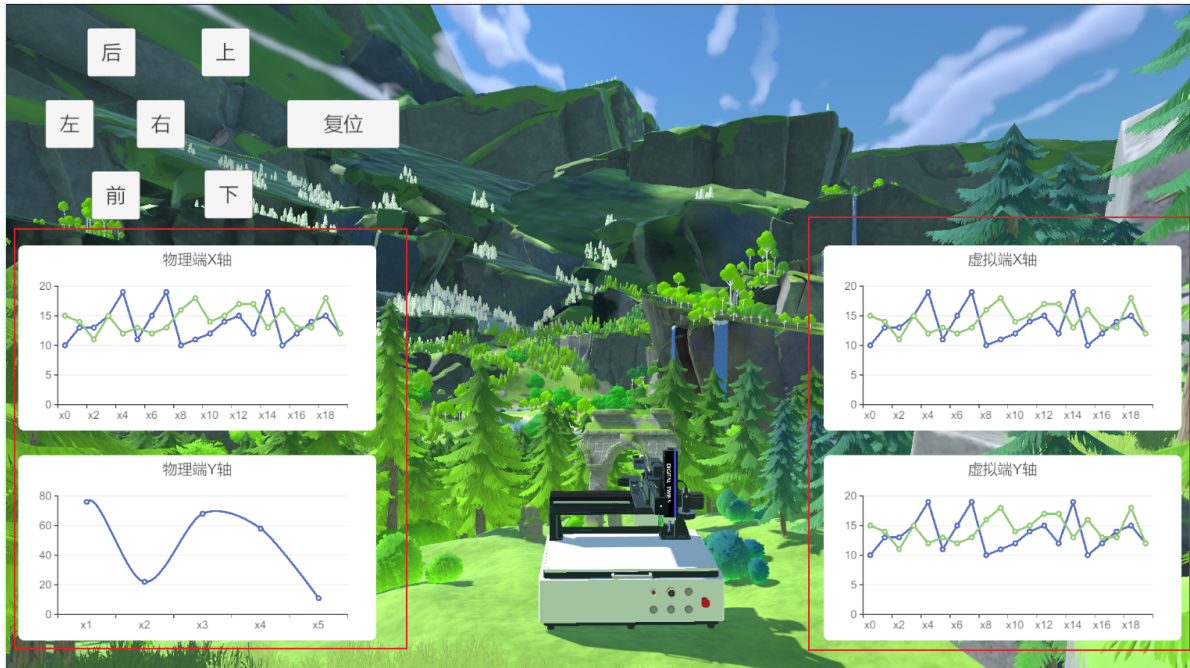
        //实时渲染虚拟端Z轴模组的运动
        var gozPosition = z_CurrentPosition -
z_LastPosition;
        z_Transform.Translate(new Vector3(0, 0, -(float)
(gozPosition / 1.0)));

        //实时渲染虚拟端R轴模组的运动
        var gorPosition = r_CurrentPosition -
r_LastPosition;
        r_Transform.Rotate(new Vector3(0, -(float)
(gorPosition / 1.0), 0));
    }
}
```

4.2.4. UI界面如何渲染

在unity中，UI界面如何渲染，这里的UI界面渲染更多的在项目中是指**图表如何实时渲染出来**

比如下面，我们要实时渲染出来物理端和虚拟端模组运动的图表



这种UI图表实时渲染，要么使用**协程**，要么在脚本生命周期方法**Update**中来完成

我这里使用了**协程**

示例代码：

```
using Assets.Scripts.Utils;
using System.Collections;
using UnityEngine;
#if INPUT_SYSTEM_ENABLED
using Input = XCharts.Runtime.InputHelper;
#endif
using XCharts.Runtime;
//[DisallowMultipleComponent]
//[ExecuteInEditMode]
public class Line_PHY_X : MonoBehaviour
{
    //定义图表字段
    private LineChart chart;
```

```

//在脚本生命周期方法中，先初始化Init图表
private void Start()
{
    Init();
    //再开启协程去更新图表
    StartCoroutine(UpdatePos());
}
//定义协程方法
private IEnumerator UpdatePos()
{
    while (true)
    {
        chart.RemoveData();
        chart.AddSerie<Line>();
        for (int i = 0; i < ModelData.Phy_X.Count;
i++)
        {
            chart.AddXAxisData("x" + i);
            chart.AddData(0, ModelData.Phy_X[i]);
        }
        ModelData.Phy_X.Clear();
        yield return new WaitForSeconds(1f);
    }
}

//图表参数的初始化
void Init()
{
    chart = gameObject.GetComponent<LineChart>();
    if (chart == null)
    {
        chart = gameObject.AddComponent<LineChart>();
        chart.Init();
    }
    chart.EnsureChartComponent<Title>().show = true;
    chart.EnsureChartComponent<Title>().text = "物理端X
轴";

    chart.EnsureChartComponent<Tooltip>().show = true;
    chart.EnsureChartComponent<Legend>().show = false;

```

```
var xAxis = chart.EnsureChartComponent<XAxis>();  
var yAxis = chart.EnsureChartComponent<YAxis>();  
xAxis.show = true;  
yAxis.show = true;  
xAxis.type = Axis.AxisType.Category;  
yAxis.type = Axis.AxisType.Value;  
xAxis.splitNumber = 10;  
xAxis.boundaryGap = true;  
}  
}
```

5. 综合示例2：仿生组-开发思路

5.1. 前言

ESP32+陀螺仪 +sensor1+sensor2

无线传输 (wifi bt)

仿生蝴蝶或者仿生青蛙要连接集成到unity，实现实时同步的效果，还是得依靠传感器拿到物理端的**位姿（位置和姿态）**

5.2. 方案一:IMU+摄像头

这种方法就是视觉slam导航那一套

IMU陀螺仪拿到物体的角速度和角加速度（也就是姿态），再通过视觉去计算物体的实时位置，最终也就拿到了位姿

这一套硬件方案是比较成熟，经济性也是最好的，方案技术资料也是很多的,但是问题是很难挂载到仿生机械上

如果做的是slam 你连接那个虚拟端就是ros->unity 3d

连接的都是树莓派 或者英伟达jerson nano

5.3. 方案二:IMU+激光雷达

思路和方案一差不多，通过**传感器融合技术**：

IMU获陀螺仪取实时的运动姿态

激光雷达拿到2D或者3D的点云数据

然后**通过融合算法进行地图构建**，也是slam那一套

问题和前面一样，**很难挂载到仿生机械上**，价格的话也会有点贵

5.4. 方案三:单IMU硬解算位姿

5.4.1. 前言

去网上看过一些资料，可以考虑只依靠IMU，硬解算物体**位姿**，但是这种解算只是估算，而且累计误差会越来越大，但是可行性会比较强

5.4.2. 数据采集

从IMU读取以下数据：

- **加速度数据 (A_x, A_y, A_z)**：单位为 m/s^2 。
- **角速度数据 (G_x, G_y, G_z)**：单位为 $^\circ/s$ 或 rad/s 。
- **采样频率 (f)**：单位为 Hz ，用于时间积分。

这些数据是我们能够从IMU中拿到的

5.4.3. 姿态计算 (Roll, Pitch, Yaw)

- **Pitch (俯仰角)**：基于加速度计数据（沿X轴的加速度与重力方向）来估算。

- **Roll (横滚角)**：基于加速度计数据（沿Y轴的加速度与重力方向）来估算。
- **Yaw (偏航角)**：通过陀螺仪的角速度积分得到，表示物体在水平方向的旋转。

如果是买那种商业化程度比较高的IMU传感器，这些姿态不需要我们自己算，厂家已经帮我们算好了

5.4.4. 位置变化估算 (x, y, z)

- **计算线性加速度 (去除重力分量)**：通过姿态 (Roll, Pitch, Yaw) 旋转矩阵将加速度转换到全局坐标系，并去除重力影响：

$$A_{linear} = A_{body} - G_{vector}$$

- **速度计算 (积分加速度)**：对线性加速度积分，得到速度：

$$v_x = \int A_x dt, v_y = \int A_y dt, v_z = \int A_z dt$$

- **位置变化 (积分速度)**：对速度积分，得到位置变化：

$$x = \int v_x dt, y = \int v_y dt, z = \int v_z dt$$

这里我们可以看到，位移数据或者说位置数据，是**通过两次积分运算来完成的**，先对加速度积分得到速度，再对速度积分得到位移

5.4.5. 坐标系转换

前面我们拿到的是积分出来的位移数据，我们应该还要进行坐标系转换

因为前面得到的数据是**基于陀螺仪自身的惯性参考坐标系**

但是我们知道，我们要的是基于**物体的质心坐标系**的运动参数

5.4.6. 伪代码

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

```

public class IMUData : MonoBehaviour
{
    // IMU数据变量
    private Vector3 acceleration; // 加速度 (Ax, Ay, Az)
    private Vector3 angularVelocity; // 角速度 (Gx, Gy, Gz)

    // 时间变量
    private float deltaTime; // 每帧的时间间隔

    // 姿态角 (roll, pitch, yaw)
    private float roll = 0f;
    private float pitch = 0f;
    private float yaw = 0f;

    // 速度和位置
    private Vector3 velocity = Vector3.zero; // 初始速度
    private Vector3 position = Vector3.zero; // 初始位置

    // 重力矫正
    private Vector3 gravity = new Vector3(0, -9.81f, 0);

    void Start()
    {
        // 初始化
        acceleration = Vector3.zero;
        angularVelocity = Vector3.zero;
    }

    void Update()
    {
        // 获取IMU数据
        acceleration = GetAcceleration(); // 获取加速度
        angularVelocity = GetAngularVelocity(); // 获取角速
        度

        // 时间间隔
        deltaTime = Time.deltaTime;

        // 更新姿态
        UpdateOrientation();
    }
}

```

```

        // 去重力后的加速度
        Vector3 linearAcceleration = acceleration -
gravity;

        // 更新速度（通过加速度积分）
        velocity += linearAcceleration * deltaTime;

        // 更新位置（通过速度积分）
        position += velocity * deltaTime;

        // 更新物体在Unity中的姿态和位置
        UpdateTransform();
    }

    // 更新姿态 (roll, pitch, yaw)
    private void UpdateOrientation()
    {
        // 通过加速度计算pitch和roll（基于重力方向）
        pitch = Mathf.Atan2(acceleration.x,
Mathf.Sqrt(acceleration.y * acceleration.y +
acceleration.z * acceleration.z)) * Mathf.Rad2Deg;
        roll = Mathf.Atan2(-acceleration.y,
acceleration.z) * Mathf.Rad2Deg;

        // 通过角速度积分yaw
        yaw += angularVelocity.z * deltaTime;
    }

    // 更新Unity物体的姿态和位置
    private void UpdateTransform()
    {
        // 更新物体位置
        transform.position = position;

        // 更新物体旋转
        transform.rotation = Quaternion.Euler(pitch, yaw,
roll);
    }

    // 获取加速度数据

```

```

private Vector3 GetAcceleration()
{
    return new Vector3(0f, 0f, -9.81f); // 默认静止状态
}

// 获取角速度数据
private Vector3 GetAngularVelocity()
{
    return new Vector3(0f, 0f, 0f); // 默认静止状态
}
}

```

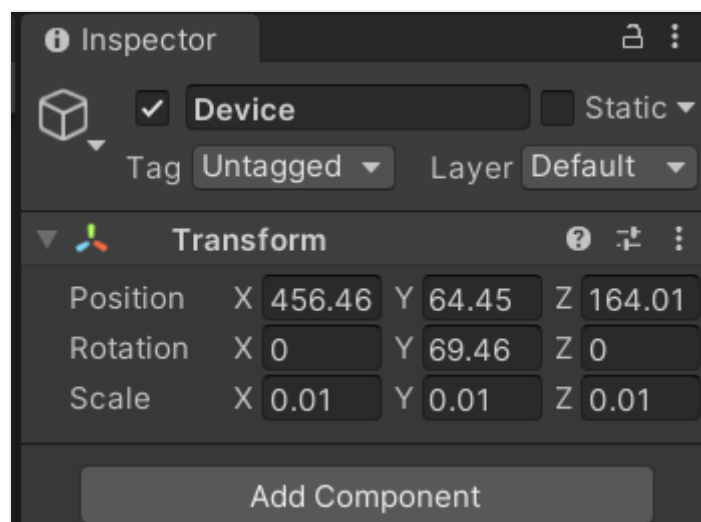
上述只是我的个人想法，不一定对，只是提供出来参考一下

5.5. 注意事项

5.5.1. unity中关于位姿的理解

在unity中，使用的是6自由度法来描述一个空间物体的位姿，即Tx Ty Tz
Rx Ry Rz

在unity中默认空间物体的姿态使用的欧拉角的描述方式

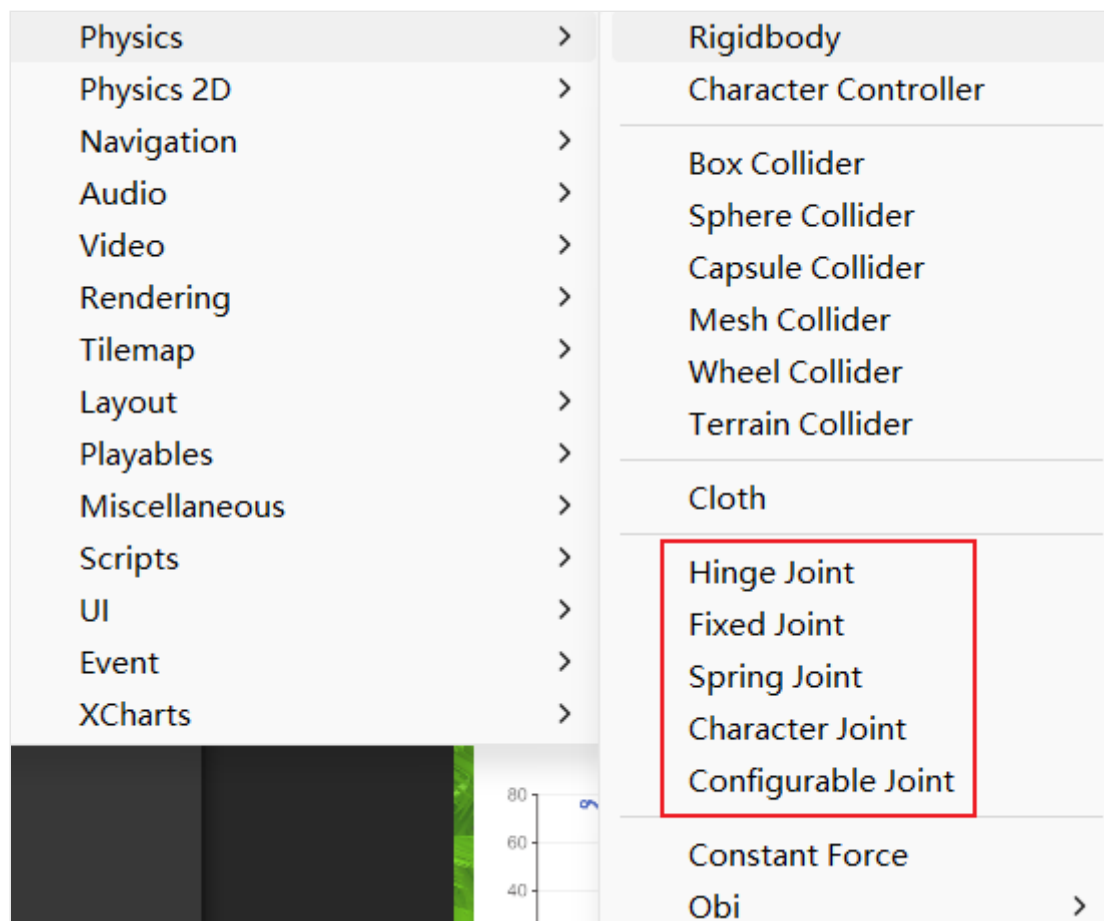


5.5.2. unity中关于运动副的理解

在unity中有Joint 组件可以供我们使用去实现运动副的连接：

- **Fixed Joint:** 固定两个物体，无相对运动。

- **Hinge Joint:** 模拟旋转副。
- **Spring Joint:** 模拟弹簧连接。
- **Configurable Joint:** 高度自定义，支持复杂约束（低副和部分高副）。
- **Character Joint:** 模拟骨骼和关节，适合角色物理。



6. 零基础学习路线推荐

6.1. C#语言

6.1.1. 书籍

看书的话，不是很推荐，没有看视频效果好，这里就推荐一本书，感兴趣的可以看看

《C#图解教程》

6.1.2. 视频

推荐大家去看刘铁猛老师的《C#语言入门详解》



刘铁猛老师讲得很好，直接去某站搜索视频观看即可

C#语言是特别基础，特别重要的内容，在unity3d中，我们实现的所有效果，都是要依靠C#编程来实现的，所以c#的语言基础特别重要，零基础的同学，建议好好看看刘铁猛老师的视频

6.2. unity3D

6.2.1. 书籍

推荐《Unity3D从入门到精通》和《Unity5.x完全自学手册》这两本书

6.2.2. 视频

unity3D的教学视频网络上比较多，大家可以去某站搜索一下，可以挑选自己认为讲得还不错的视频去观看

作者: 缪新亮

日期: 2025-01-16