PC 端 VR 实验文档

——使用 VRTK 开发 VR 汽车驾驶项目

前言——标识说明

该前言部分主要为项目与代码中的专用标识进行说明，方便同学们快速进行实验。其中文档中出现的标识种类包括：

a. **加粗**：突出不能被忽略的内容重点；

b. **红色加粗**：说明以下内容需要自行编写代码实现；

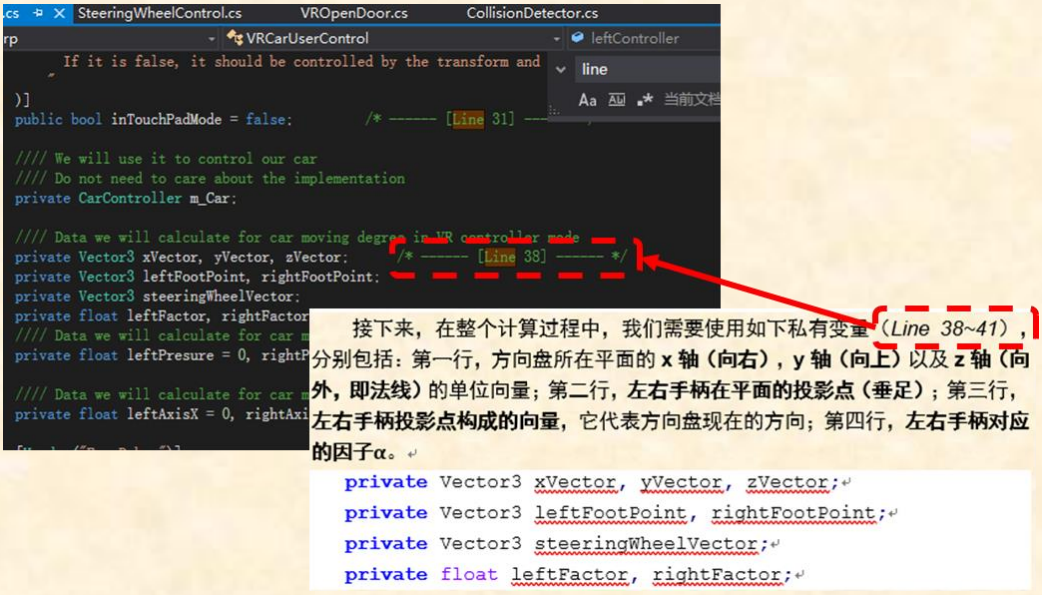
c. *Line*：指明该部分文档内容对应了脚本中的哪一行代码；

d. 绿色提示框：给出必要的提示信息，通常伴随**红色加粗**标识出现，给出代码所需的基础知识或提示思路。

代码中出现的标识种类包括：

a. [TODO: SECTION XXX]：TODO 标志表示以下代码需要修改才能完成相应功能，通常需要你解注释或自行编写部分代码。其中 Section 表示这一 TODO 部分对应的文档中的章节数，你不必立刻修改那些还未读到的章节的 TODO 代码。

b. [Line XXX]：该行代码在原脚本中的行数，与文档中的 *Line* 一一对应。由于脚本在编写过程中，行数会随着代码的增减发生偏移，因而你可以通过 Ctrl+F 检索[Line XXX]标识的方式对应文档与代码（如下图）。



项目简介

本次实验为 PC 端 VR 实验，将基于 HTC Vive 或 Oculus 设备，在 PC 端上开发一款简单的汽车驾驶项目。在本次实验中，我们的代码将更偏重 VRTK 的使用，而非游戏性上的编写。同学们将更加深入地了解 VRTK 这个通用 VR 开发包，使用它来获取底层交互数据，例如手柄的位置与按键等信息；或是其他高级交互功能，例如触碰（Touch）、抓取（Grab）与使用（Use）事件。

本次实验素材包有五个，包括一个基本的项目场景、VRTK 工具包、SteamVR 开发插件、Oculus 开发插件以及 VRTK—Windows MR 拓展插件。请先导入基本项目场景，并根据文档要求一步步导入、配置与编写 VRTK 相关内容，最后在测试运行前，根据需要使用的硬件种类，导入 SteamVR 开发插件、Oculus 开发插件或是 WMR 拓展插件。

实验要求

a. 完成汽车驾驶控制。包括使用手柄控制车辆转向与使用按键控制车辆加速；
b. 完成与车门的交互。包括手柄触碰高亮显示，以及开关车门；
c. 其他练习内容，包括手柄振动、方向盘控制、以及触摸板交互；
d. 进阶功能（加分项）

实验内容

1 准备工作

1.1 导入实验初始化场景

找到附件中的 **CarGame-init.unitypackage**，它是本次实验所要使用的初始场景。其中包含三个文件夹，首先 SampleScenes 与 Standard Assets 文件夹来源于 Unity 官方素材 Standard Assets，我们提取了其中的汽车驾驶部分以用于本次 VR 实验。然后是 MyAssets，用于存放我们自己的素材与脚本，其中包含一个高精度的汽车模型（Model）与其对应的预制体（Prefab），以及本次实验中，我们需要编写的各种脚本。

找到 **SampleScenes -> Scenes -> Car**，双击打开该场景。该场景改编自 Unity 官方 Car 场景，并使用高精度汽车模型替换原有的汽车模型（更加适合 VR 场景的需要）。其中所有的汽车控制、碰撞体适配等操作均由助教完成，如无特殊需求，请不要随意修改 Car 物体下原有的汽车控制脚本及其内容，以及其子物体的层级结构。

现在点击运行，你可以使用上下左右/WSAD 控制车辆的移动。接下来我们的任务即是将这个项目修改为在 VR 场景下运行的项目。

1.2 导入并设置 VRTK

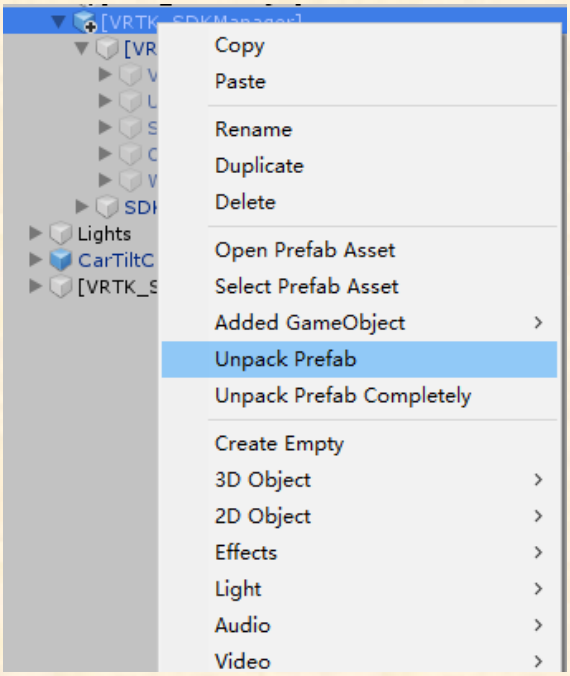
之前的实验我们提到过 VRTK。VRTK 是一款辅助构建 VR 项目的开发工具包，它的脚本可以方便、快速且通用地构建各种 VR 解决方案，而不用在意你所使用的 VR 硬件设备。在本次实验中，我们将使用它来进行各种交互事件的开发，

并使用它来同时支持 HTC Vive（SteamVR）以及 Oculus 设备。

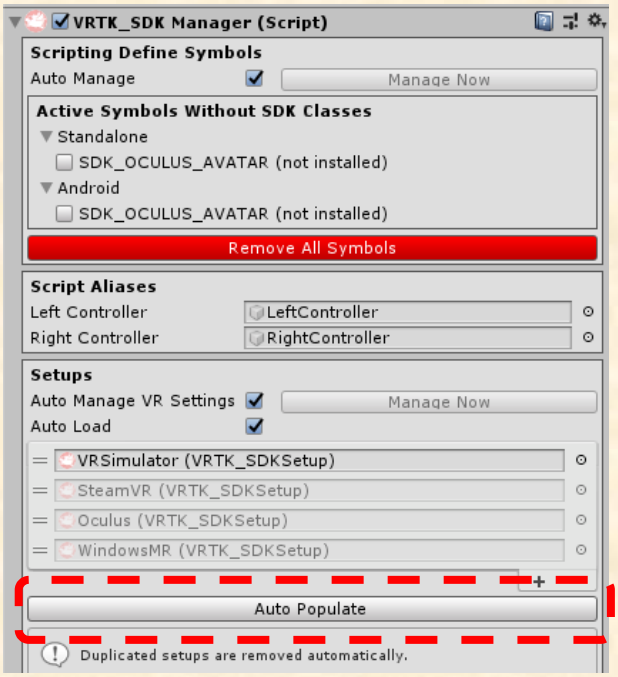
找到附件中的 **VRTK - Virtual Reality Toolkit - VR Toolkit.unitypackage**，将其导入至实验场景中。找到路径 **VRTK -> Examples -> ExampleResources -> SharedResources -> Prefabs -> SDKManager**，将**[VRTK_SDKManager]**物体拖入到场景中。[VRTK_SDKManager]是 VR 设备 SDK 的管理器，它会在运行时根据你所使用的设备，自动配置对应类型的 SDK，同时在当前位置生成 VR 边界（boundary）、头盔（headset）与手柄（controllers）。

为了保证驾驶汽车时头盔的视野一并移动，我们**将[VRTK_SDKManager]移动到 Car 下**，成为其子物体（即与 CarModel 同一层级），不要忘记将[VRTK_SDKManager]的位置**重置为(0，0，0)**。随后**禁用原来场景中的 Camera**（VRTK 会在运行时自动启用适合当前 VR 设备的 Camera）。

接下来，我们需要对其进行设置。但在修改一个来自预制体的实体前（特指删除其脚本、子物体或修改其层级结构），我们需要断开其与原预制体之间的联系，从而使得我们对实体的修改不会影响到原预制体。具体来说，我们需要右击[VRTK_SDKManager]，选择 **Unpack Prefab** 指令（如下图所示）。

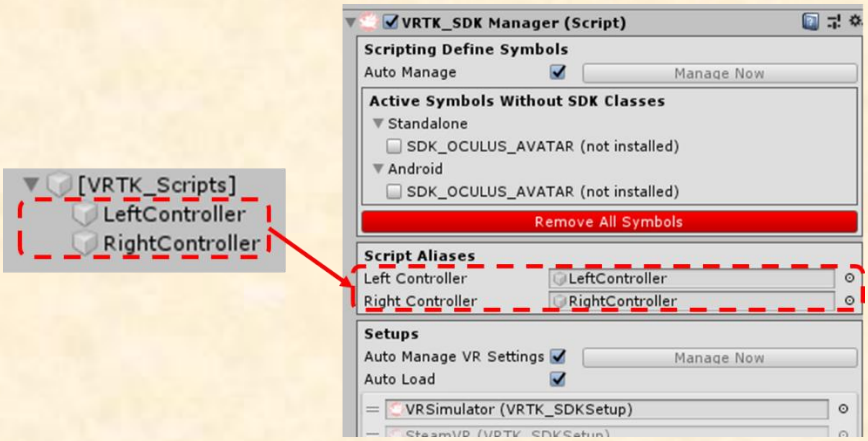


在 Unpack 之后，我们可以随意修改这个实体，而不会影响到预制体了。现在，我们将其子物体 **SDKSetupSwitcher** 删除，并将[VRTK_SDKSetups]下的 **UnityXR** 删除（只支持模拟器、HTC Vive、Oculus 以及 WindowsMR）。然后由于删除了 UnityXR，我们需要重新加载对应的 SDK。找到[VRTK_SDKManager]下的脚本 **VRTK_SDKManager**，点击 **Auto Populate** 按钮即可修正 Setups（如下图红框所示）。



接下来，我们需要设置手柄，正如之前所说，我们不知道自己会使用什么样的 SDK，所以自然也就不清楚对应的手柄物体在哪里，更别提要使用什么样的 API 了。而 VRTK 提供了一个统一的接口，它可以将你创建的物体与实际的手柄进行关联。

首先，我们创建一个空物体，取名为**[VRTK_Scripts]**。随后在其底下创建两个空物体，分别叫做 **LeftController** 与 **RightController**，我们希望使用这两个物体来与真实手柄建立映射关系。找到之前的[VRTK_SDKManager]下的 VRTK_SDKManager 脚本，并找到 **Scripts Aliases** 这一项，将之前的空物体**对应填入**即可（如下图所示）。现在，LeftController 与 RightController 就好像真实手柄的对应引用，你可以通过它们来获取所需的位置信息、监听按键信息以及获取触摸抓取等交互信息了。若无特殊说明，下文中所说的左右手手柄，均指代 LeftController 与 RightController 这两个物体。



2 汽车驾驶控制

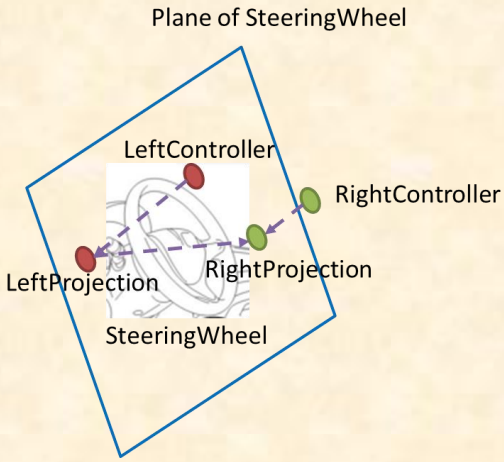
经过前面的设置，VR SDK 与手柄已经配置完成了，现在我们就可以开始与 VR 相关的交互行为开发了。首先我们来尝试使用手柄的位置与按键信息来开发 VR 场景下的

汽车驾驶控制。

找到Car物体上的**CarUserControl**脚本，将其注销。这个脚本使用上下左右/WSAD键控制汽车移动，在真正使用 VR 设备时是没有用的。随后，找到 **VRCarUserControl** 脚本，将其绑定在 Car 物体上，我们将使用它在 VR 场景下控制汽车驾驶。

2.1 使用手柄位置信息控制汽车转向

使用传统设备时，我们通常使用键盘上或是手柄上的按键来控制汽车移动。而在 VR 场景中，我们拥有了空间定位信息，这使得我们可以做一些更加有趣且逼真的交互设计。我们希望能根据**手柄在空间中的位置**模拟**汽车的方向盘的转动**。



具体来说如上图所示，我们将**左右手柄**对应**投影到方向盘所在的平面**上，然后使用**两个投影点形成的角度**控制方向盘的转向。具体步骤如下：

已知**方向盘的法线向量**$\vec{\mathbf{z}} = (\mathbf{x_z}, \mathbf{y_z}, \mathbf{z_z})$以及**方向盘原点**$\mathbf{O} = (\mathbf{x_0}, \mathbf{y_0}, \mathbf{z_0})$，可求方向盘平面：

$$\mathbf{x_z}(\mathbf{x} - \mathbf{x_0}) + \mathbf{y_z}(\mathbf{y} - \mathbf{y_0}) + \mathbf{z_z}(\mathbf{z} - \mathbf{z_0}) = \mathbf{0} \quad [1]$$

随后对于已知**左手手柄位置**$\mathbf{L} = (\mathbf{x_L}, \mathbf{y_L}, \mathbf{z_L})$，需求其在**该平面上的投影（垂足）**$\mathbf{F} = (\mathbf{x_F}, \mathbf{y_F}, \mathbf{z_F})$。代入公式**[1]**，并由垂线段与法线平行可以得到：

$$\mathbf{x_z}(\mathbf{x_F} - \mathbf{x_0}) + \mathbf{y_z}(\mathbf{y_F} - \mathbf{y_0}) + \mathbf{z_z}(\mathbf{z_F} - \mathbf{z_0}) = \mathbf{0} \quad [2]$$

$$\frac{\mathbf{x_L}-\mathbf{x_F}}{\mathbf{x_z}} = \frac{\mathbf{y_L}-\mathbf{y_F}}{\mathbf{y_z}} = \frac{\mathbf{z_L}-\mathbf{z_F}}{\mathbf{z_z}} \quad [3]$$

结合**[2][3]**联立方程得：

$$\mathbf{x_F} = \mathbf{x_L} + \frac{\mathbf{x_z}(\mathbf{x_0}-\mathbf{x_L}) + \mathbf{y_z}(\mathbf{y_0}-\mathbf{y_L}) + \mathbf{z_z}(\mathbf{z_0}-\mathbf{z_L})}{\mathbf{x_z^2}+\mathbf{y_z^2}+\mathbf{z_z^2}}\mathbf{x_z} \quad [4]$$

设中间的因子为$\boldsymbol{\alpha}$，则[4]改写为：

$$\mathbf{x_F} = \mathbf{x_L} + \boldsymbol{\alpha}\mathbf{x_z} \quad [5]$$

又因为$\boldsymbol{\alpha}$本身有对称性，根据对称性可得

$$\mathbf{F} = \mathbf{L} + \boldsymbol{\alpha}\vec{\mathbf{z}} \quad [6]$$

这样，通过**方向盘原点、法线**，结合**手柄位置**，我们能计算出其**投影（垂足）位置**。然后通过比较**左右两个垂足位置形成的向量**与**原方向盘 x 轴向量**的偏移，即可计算角度信息。

现在就让我们编写脚本来实现这一功能。首先，我们已经提供了一些变量与代码。打开 VRCarUserControl 脚本，在本小节中，我们将使用如下公有变量(*Line*

9~13, Line 19~22)，它们分别是**手柄的位置信息**与**方向盘的位置信息**，我们之后需要在 Inspector 上初始化它们，因而不需要通过代码初始化。

```
public Transform leftController;
public Transform rightController;

public Transform steeringWheel;
```

接下来，在整个计算过程中，我们需要使用如下私有变量（*Line 38~41*），分别包括：第一行，方向盘所在平面的 **x 轴（向右）**，**y 轴（向上）**以及 **z 轴（向外，即法线）**的单位向量；第二行，**左右手柄在平面的投影点（垂足）**；第三行，**左右手柄投影点构成的向量**，它代表方向盘现在的方向；第四行，**左右手柄对应的因子α**。

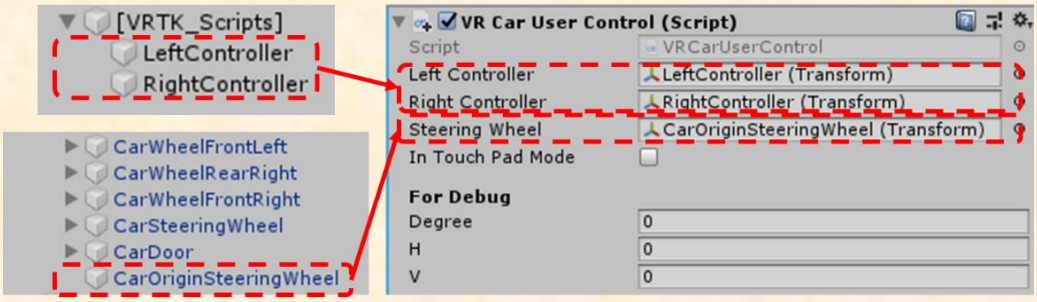
```
private Vector3 xVector, yVector, zVector;
private Vector3 leftFootPoint, rightFootPoint;
private Vector3 steeringWheelVector;
private float leftFactor, rightFactor;
```

对于计算出的结果，我们会保留两个值，分别为角度 degree（-90~90），以及归一化角度值 h（-1~1）（*Line 50*，*Line 52*）。

该功能对应代码的**主体范围**为 *Line 56~73*，以及 *Line 161~185* 。我们提供了其中部分代码，包括获取**方向盘平面的各向单位轴**信息（*Line 161~164*），以及**计算方向盘向量、角度与归一化角度**（*Line 175~184*），而你的任务就是**根据前面的数学描述补写这部分代码**。具体提示如下：

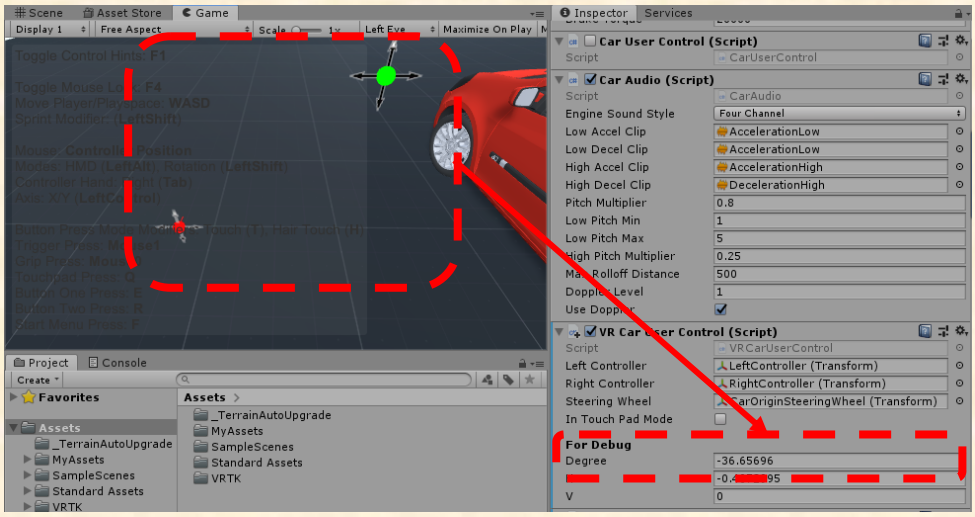
> **提示**：1. 根据公式[4][5]补写 Factor 函数（*Line 56~73*），它用来计算前面提及的因子**α**。
> 2. 使用 Factor 函数计算 leftFactor 与 rightFactor（*Line 166~174*）
> 3. 使用公式[6]计算 leftFootPoint 与 rightFootPoint（*Line 166~174*）

完成后，我们将脚本绑定在 Car 物体上，然后不要忘记在 Inspector 上**初始化手柄与方向盘位置**。如下图所示，其中方向盘位置实体使用 **Car → CarModel → CarOriginSteeringWheel**，这是方向盘原点的一个拷贝。注意**不要选择 CarSteeringWheel**，它是方向盘实体，控制一个方向盘模型，它的旋转轴是可能变化的（在下文的某个功能需求中，我们会修改它）。



最后我们可以开始使用 Simulator 进行测试。在**保证在 Inspector 上处于 VRCarUserControl 的位置时**，运行 Unity。使用 **WSAD** 调整当前位置至一个空旷位置。接下来按下 **Left Alt 切换至手柄模式**，接下来**按住 Left Ctrl，使用鼠标移动手柄**，观察手柄所成角度与脚本上 **For Debug** 列表中的 **Degree** 属性是否

吻合，以判断该功能是否编写正确（如下图所示，注：负的角度代表向左，正的角度代表向右）。



2.2 监听手柄按键信息以控制汽车加速度

接下来，我们要尝试**使用手柄按键**来**控制汽车的向前或向后移动**。首先，为了监听手柄按键，我们需要为手柄绑定事件脚本，具体来说，我们为**左右手手柄**绑定 **VRTK_ControllerEvents** 脚本。这个脚本基本包括了所有的手柄事件，可以被代码直接访问，同时也是其他所有高级交互方式（例如抓取、触碰与使用）的基础。

高级交互方式我们在下文中再做介绍，这里我们将使用代码来监听手柄的按键信息。我们使用**右手扳机键（Trigger）键**来控制**汽车向前的加速度**（看作油门），而使用**左手扳机键（Trigger）**控制**汽车向后的加速度**（看作刹车）。选择 Trigger 按键是因为该按键不仅有“**是否按下**”这个信息（True/False），还有“**按下程度**”这个信息（0~1 的 float 类型），这可以方便我们更加准确地控制汽车。

现在就让我们来看看该如何编写脚本。再次打开 VRCarUserControl 脚本。首先，由于 VRTK 插件的导入，我们可以解注释一些代码。首先是 *Line 4*：

```
// using VRTK;
```

VRTK 的所有脚本都在其同名的名字空间中，**解注释这句话**，才能使用 VRTK 插件的所有脚本。然后是 *Line 14~17*：

```
// private VRTK_ControllerEvents leftControllerEvents;
// private VRTK_ControllerEvents rightControllerEvents;
```

这两个即我们之前提到的**左右手的手柄事件脚本**，我们待会会在 Awake 函数中初始化它，**解注释**它们。随后我们还会用到如下变量（*Line 43*），这两个值用于存储从手柄那里**获取的 Trigger 按键程度**（0~1）

```
private float leftPresure = 0, rightPresure = 0;
```

现在，让我们来看代码主体。首先我们要在 Awake 函数中**初始化**两个手柄事件。然后，我们找到手柄事件下面的 **TriggerAxisChanged 代理事件**，将我们自己的处理函数**注册**上去，**解注释**如下语句（*Line 114~116, Line 128~130*）：

```
// leftControllerEvents =
leftController.GetComponent<VRTK_ControllerEvents>();
```

```
    // leftControllerEvents.TriggerAxisChanged +=
OnLeftTriggerAxisChanged;
    // rightControllerEvents =
rightController.GetComponent<VRTK_ControllerEvents>();
    // rightControllerEvents.TriggerAxisChanged +=
OnRightTriggerAxisChanged;
```

这里我们需要解释一下**代理事件**。代理事件其实与**关注—推送机制**类似，其运转机制如下：

a. 将自定义函数**注册**在代理事件上（类似关注行为）；
b. 在**定义好的某个特定情况**下，一个代理事件被触发；
c. 被触发的代理事件查找**所有已注册的函数**；
d. 对应**执行**这些函数（类似推送行为）。

回到刚才的代码，以左手端为例。leftControllerEvents 下面有许多这样的代理事件，它们在特定的情况下会触发，例如这里使用的名为 TriggerAxisChanged 的代理事件，它会在左手柄 Trigger 按键的轴信息（即“按下程度”）改变时触发。随后我们将 OnLeftTriggerAxisChanged 函数注册在 TriggerAxisChanged 事件下，那么在对应情况发生时，TriggerAxisChanged 事件就会自动调用该函数。注意，一个代理事件可以注册多个自定义函数。

那么接下来，我们就要写自定义函数了。找到 **OnLeftTriggerAxisChanged** 与 **OnRightTriggerAxisChanged** 函数（*Line 75~85*），**解注释**它们：

```
    //// When trigger axis of leftController was changed
    // [TODO: SECTION 2.2] Uncomment the function
    // private void OnLeftTriggerAxisChanged(object sender,
ControllerInteractionEventArgs e) {
    //     leftPresure = e.buttonPressure;
    // }

    //// When trigger axis of rightController was changed
    // [TODO: SECTION 2.2] Uncomment the function
    // private void OnRightTriggerAxisChanged(object sender,
ControllerInteractionEventArgs e) {
    //     rightPresure = e.buttonPressure;
    // }
```

这些自定义函数的格式相同，按键的信息会通过参数 e 返回给我们，这里我们只要在轴信息更改时记录其 **buttonPressure**（按下程度）即可。最后我们会在 Update 函数中使用 leftPressure 与 rightPressure 计算 v（*Line 157~158*）。由于 Simulator 无法模拟按下程度的多少，所以该部分必须在 VR 端进行测试。

3 车门控制

上一章节，我们通过 VRTK 的底层代码获取其位置信息与按键信息。而在本章中，我们将通过已有的 VRTK 脚本来快速构建一些高级功能。我们需要使用这些高级功能来实现对车门的控制，主要包含车门触碰高亮与开关车门这两个效果。

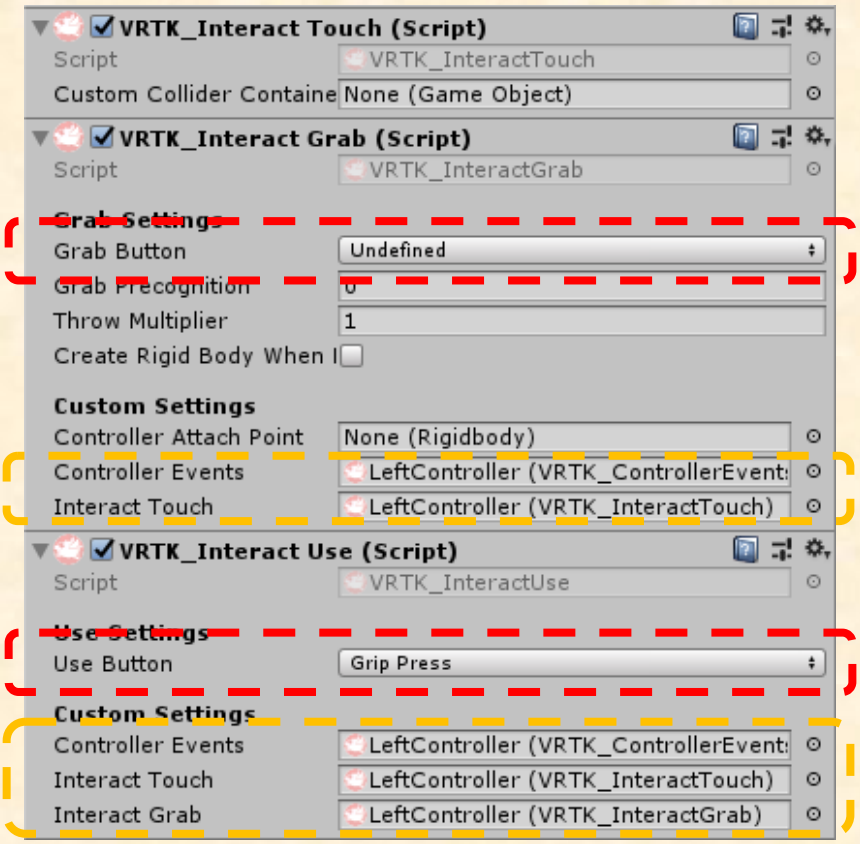
3.1 VRTK 手柄与物体交互功能

VRTK 手柄与物体的高级交互功能主要有三：

a. **触摸（Touch）**：用户在触摸物体时产生的事件，常见应用场景有触碰高亮等；
b. **抓取（Grab）**：用户在触摸物体的同时按下（或按住）对应按键，可以将物体抓取至手柄处（或其他固定位置），常见应用场景有拾取道具等。
c. **使用（Use）**：用户在触摸物体（或抓取物体，可设置）的同时按下对应按键会产生的事件，常见应用场景有场景事件触发（开关房门）、使用道具等；

这三种事件分别对应着三种 VRTK 脚本，分别为 **VRTK_InteractTouch**、**VRTK_InteractGrab** 以及 **VRTK_InteractUse**。它们有顺序依赖关系，即要实现后者需求时，必须同时绑定所有前者的脚本，同时注意 **VRTK_InteractGrab** 与 **VRTK_InteractUse** 脚本需要依赖于之前提到的 **VRTK_ControllerEvents** 来获取按键信息。

在本章节中，我们需要触摸（Touch）与使用（Use）这两个交互功能，因而我们需要包含所有这三个脚本。我们将所有脚本分别**绑定在左右手柄上**，然后按下图所示初始化脚本（以左手柄为例。红色是必改项；而橙色表示可选修改项目，你可以在此留空）。其中前者 Grab Button 为 **Undefined** 代表着我们**不使用抓取功能**，而 Use Button 为**Grip Press** 代表我们在满足条件下**使用握持键（Grip）**会触发使用事件。



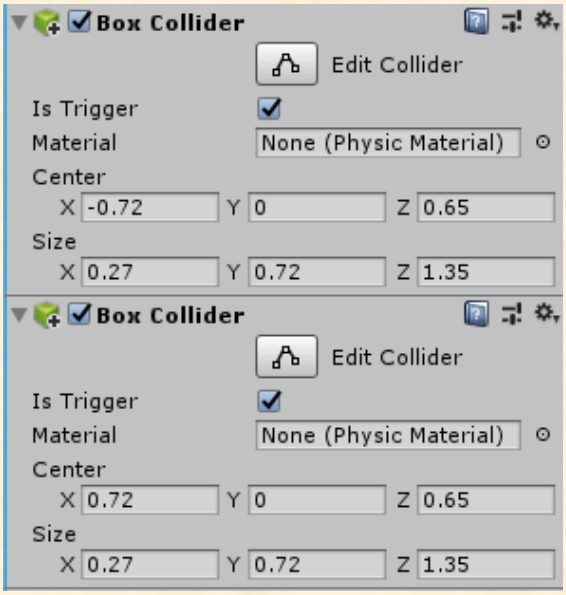
绑定了这些脚本之后，我们的手柄就有了触碰与使用的功能。但具体哪些物体可以与手柄交互，交互时会触发哪些功能，这就要到各自的物体下进行设置了。

3.2 触碰车门时的描边高亮

找到 **Car → CarModel → CarDoor**，该物体是已经分割的车门的实体，我

们要把它变为**可交互物体**。在 VRTK 中，一个可交互对象需要满足两个条件，一是必须包含至少一个**碰撞器**或**触发器**，二是必须包含 **VRTK_InteractObject** 脚本，这个脚本专门用于定义可交互对象的细节。

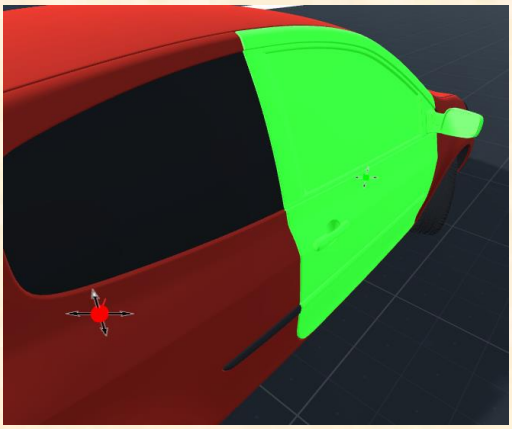
根据这一要求，我们需要先在车门实体上绑上碰撞体，我们使用两个 **BoxCollider** 来近似左右门的形状，并直接绑定在 CarDoor 上。其设置如图所示，注意其中 **IsTrigger** 选项必须勾选，IsTrigger 可以将该碰撞器转化为触发器，这样该物体会参与碰撞检测，但不会应用在 Unity 物理模拟中。



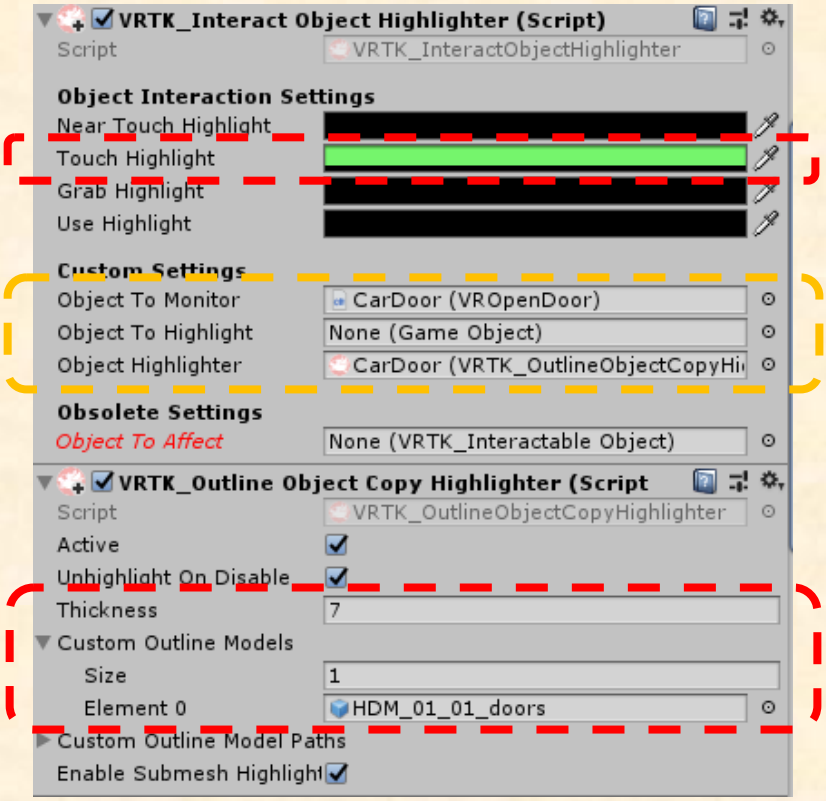
接下来，我们需要绑定 VRTK_InteractObject 脚本。由于我们之后要实现开关车门的操作，需要重写使用（Use）功能，因而我们实际需要绑定一个继承于 VRTK_InteractObject 脚本的自定义脚本来实现这一步骤。

找到 **VROpenDoor** 脚本，该脚本继承于 VRTK_InteractObject。反注释 using 语句（*Line 2*）以及整个脚本类（*Line 6~70*）。针对 VROpenDoor 的代码编写将放在下一小节处理，在本小节，我们只要将其视为一个 VRTK_InteractObject 脚本即可。我们将其直接**绑定在 CarDoor 上**。

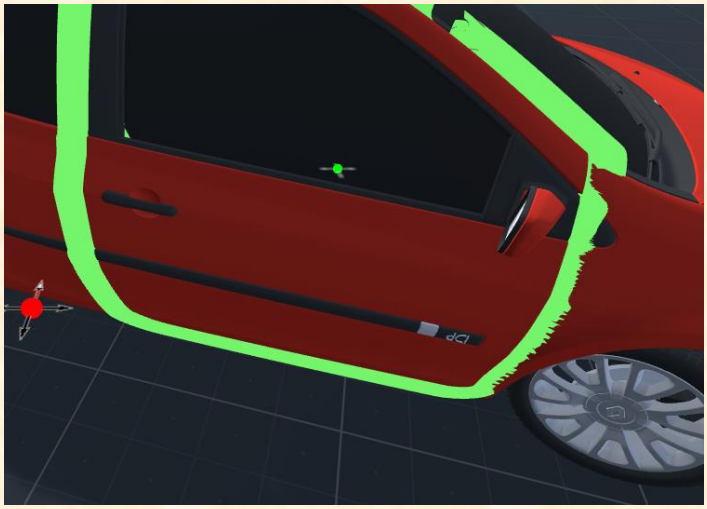
现在车门已经成为可交互物体了，我们先来实现触碰高亮的内容。VRTK 提供了一些已写好的脚本，我们可以直接使用。首先，绑定 **VRTK_InteractObjectHighlighter** 脚本。该脚本控制物体的高亮，会在手柄触碰（Touch）时触发，默认情况下会使用 **VRTK_MaterialColorSwapHighlighter** 脚本控制高亮（顾名思义，直接交换材质颜色）。效果如下图（需调整颜色）：



当然这里默认高亮脚本是**颜色替换脚本**，我们也可以改成其他类型脚本，例如**描边高亮脚本**。找到 **VRTK_OutlineObjectCopyHighlighter** 脚本，然后如下图所示，同时设置该脚本与 **VRTK_InteractObjectHighlighter** 脚本（红色是必改项；而橙色表示可选修改项目，你可以在此留空）。其中 **Touch Highlight** 默认为黑色，代表不进行高亮，你可以改成你喜欢的颜色。**Object To Monitor** 与 **Object Highlighter** 为可选设置，如果你留空，它们也会找到当前物体（CarDoor）下绑定的对应脚本；而 **Object To Highlight** 不需要设置（且设置也会无效），因为在绑定物体具体的高亮方法后（**Object Highlighter**），我们不再使用它指明高亮对象，而是使用对应脚本中的 **Custom Outline Models** 来指明高亮对象。**Thichness** 为高亮厚度，默认为 1，你需要根据物体的大小调整这个值，这里设置为 7。**Custom Outline Models** 指代的是高亮对象（必须包含面片，这里 HDM_01_01_doors 为门的面片网格），可以指代复数个。



设置完毕后，你可以使用 Simulator 来测试效果了，具体效果如下图所示：

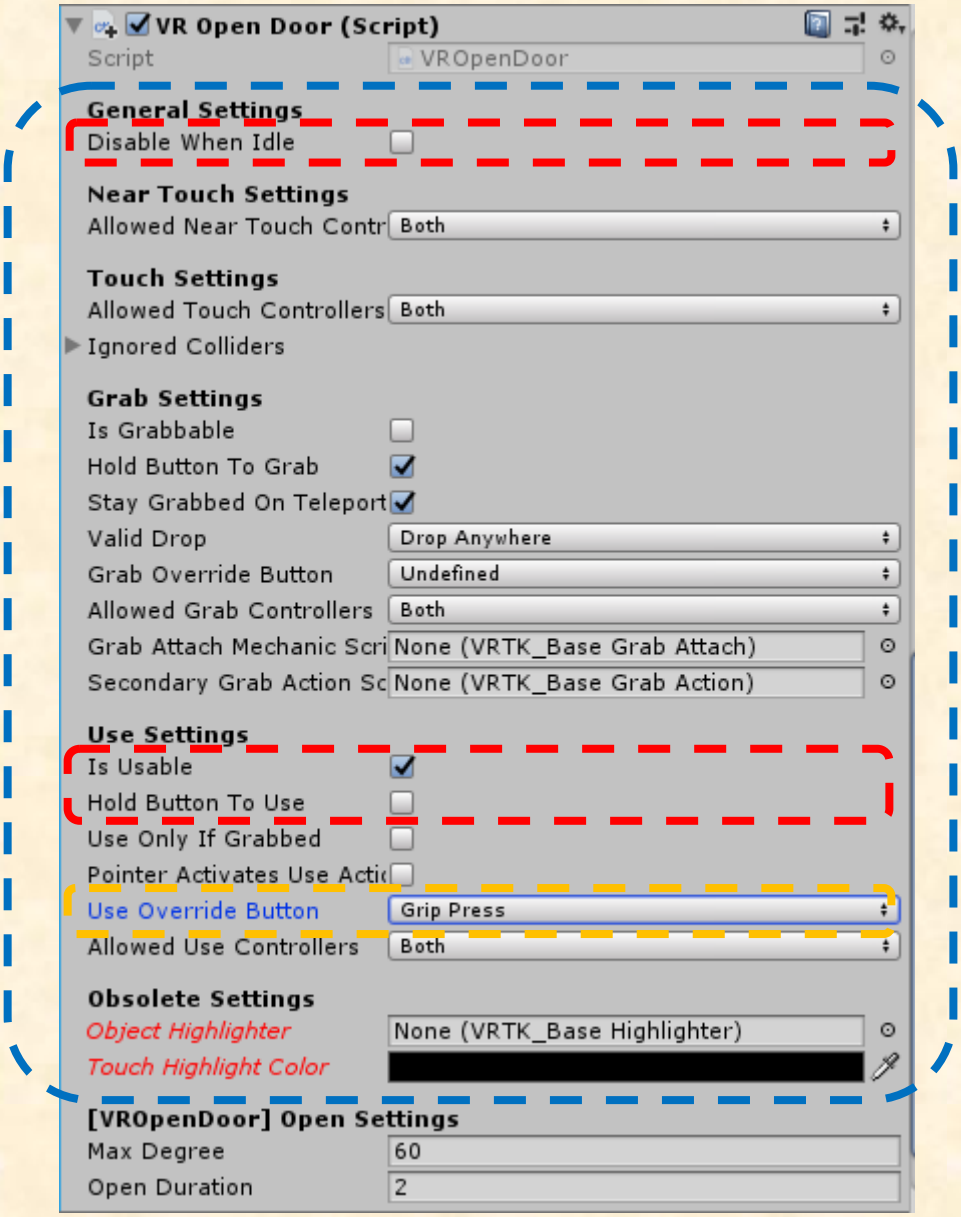


3.3 开关车门

与物体高亮不同，物体的使用（Use）事件多种多样，不同的应用可能有不同的需求。考虑到这点，VRTK 将设计的选择权交由用户。具体来说，VRTK 在 VRTK_InteractObject 脚本中提供了各种事件的**虚函数**，我们可以通过**函数重载**的方式来编写我们自己的交互事件。在本项目中，我们希望利用使用（Use）事件实现一个**开关车门的交互行为**，我们希望车门可以像跑车一样旋开/旋关，具体细节：

a. 车门已开启时，按动 Grip 键可以关闭车门；车门已关闭时，按动 Grip 按键可以开启车门；
b. 车门开启/关闭时需平滑处理，即在一定时间内（如 2s 内）从开启/关闭状态旋转到关闭/开启状态；
c. 由于车门的左右门 Mesh 未分离，不能左右开车门，需要像跑车一样旋开车门（使车门沿 x 轴旋转）。

了解了需求之后，让我们回到之前的 VROpenDoor 脚本，它继承自 VRTK_InteractObject。首先，为了正常进行使用（Use）事件的交互，我们需要先在检视板（Inspector）上对脚本进行设置。设置如下图，其中蓝色区域为继承的 VRTK_InteractObject 参数（红色是必改项；而橙色表示可选修改项，你可以保留默认值）。首先 **Disable When Idle** 表示当手柄不与该物体接触时，这个脚本会被自动禁用，我们不能勾选该选项，因为我们的脚本还需要在使用（Use）交互后控制门的开启与关闭（旋转动画）。接下来我们需要勾选 **Use Settings** 中的 **Is Usable** 选项来使用该功能，并取消 **Hold Button To Use** 选项，表明我们只要点击按钮就可以触发使用（Use）事件（而非按住按钮）。**Use Override Button** 属性与手柄上的 **VRTK_InteractUse** 上定义 **Use Button** 是相同的，如果两者都被定义，以物体上的 Use Override Button 为准；如果 Use Override Button 未被定义（Undefined），以手柄上的 Use Button 为准。



设置完毕后，我们现在就要开始编写代码了。打开 **VROpenDoor** 脚本，我们需要使用如下公有变量（*Line 10~14*），其中 **maxDegree** 代表门打开的最大角度，**openDuration** 为门从关闭到开启所需的时间（或相反）。

```
public float maxDegree = 60;
public float openDuration = 2.0f;
```

随后我们可能需要使用一些私有变量进行辅助（*Line 16~21*），我们可以使用 **origin** 变量存储车门的**初始旋转值**，使用 **currTime** 记录**当前车门的旋转用时**，使用 **isOpenning** 来记录当前的**车门状态**，是开还是关（默认为关）。你可以随意取用它们，或是增加其他你认为需要的变量：

```
private Quaternion origin;
private float currTime = 0;
private bool isOpenning = false;
```

现在，你的任务是**编写代码来重载 VRTK_InteractObject 提供给你的三个虚函数**，来完成我们对于车门开关交互的需求。这三个虚函数分别是：

a. **Awake** 函数（*Line 23~32*）：在应用开启时调用一次该函数。
b. **Update** 函数（*Line 34~60*）：在每一帧调用一次该函数。
c. **StartUsing** 函数（*Line 63~68*）：在使用（Use）事件发生时调用该函数。

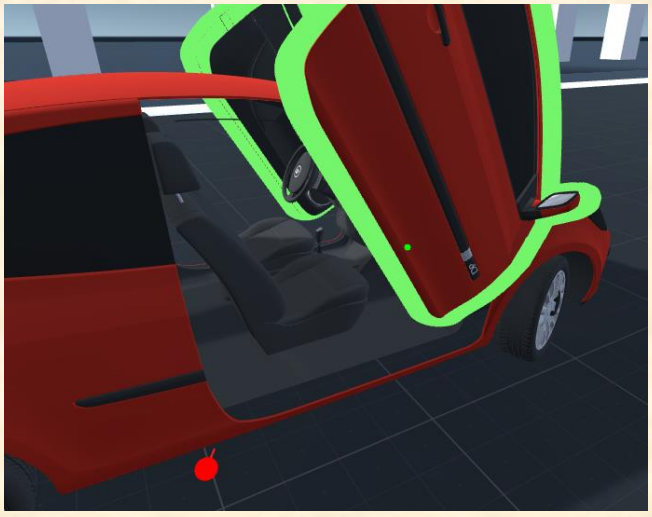
提示：你可能需要知道如下知识：

1. 注意 **transform.rotation** 与 **transform.localRotation** 的区别，**一个是世界坐标，一个是本地坐标（相对父物体）**。
2. Unity 使用**四元数（Quaternion）**类型存放旋转信息，它支持乘法操作，A*B 代表着**在 A 旋转值的基础上，再按 B 旋转值旋转**。
3. Quaternion 类型不直观，难以进行赋值，但你可以使用静态函数 **Quaternion.Euler(float x, float y, float z)**将 Euler 角转化为四元数。
4. Euler 角是一个很直观的概念，**Euler(x, y, z)**代表物体**沿 x 轴 y 轴与 z 轴分别旋转对应度数**。在该项目中，车门只沿本地坐标系的 x 轴进行旋转，因而你可以使用 **Euler(degree，0，0)** 表示其旋转度数。

提示：这里提供一种可行的解决思路：

1. 初始时，**记录车门的旋转值**。
2. 维护一个 **bool 标记**（isOpenning），在使用（Use）事件发生时，**切换其 True/False 值**。
3. 在每一帧，若门处于**正在开启**状态，增加**当前时间**直至到达最大时间；若门处于**正在关闭**状态，减少**当前时间**直至 0。
4. 使用**当前时间**计算**旋转角，当前旋转值=初始旋转值*旋转角**。

完成代码编写后，你可以使用 Simulator 进行测试。注意在按下 **Left Alt** 切换为手柄模式后，你可以使用**鼠标左键点击**来模拟 Grip 按键，效果如图所示：



4 其他练习内容

4.1 手柄振动反馈

在驾驶车辆时，我们会撞上各种障碍物，虽然车辆会震动或是被弹开，但在现实中我们却没有任何感觉，换句话说，我们需要一个**触觉上的碰撞反馈**。虽然现有的商用 VR 技术在触觉上的反馈手段非常有限，但是大部分 VR 设备厂商至少考虑到了一点，那就是通过**手柄振动**来给予用户反馈。

当然，VRTK 也为这些设备提供了一个通用的振动接口。与之前我们提到的通过脚本获取数据或是高级交互反馈不同，VRTK 提供了大量公用**静态函数**，使得用户可以不需要绑定特定的脚本就调用到该功能。

现在，找到 **CollisionDetector** 脚本，将其**绑定在汽车 Car 上**。你的任务是**编写代码实现车辆碰撞时的手柄振动**，记得在检视板（Inspector）上进行必要的初始化操作。

提示：你可能需要知道如下知识：

1. **VRTK_ControllerHaptics** 类控制手柄的振动，其中的静态函数 **TriggerHapticPulse(controllerRef, strength, duration, interval)**可以触发手柄的持续振动。其中 **cotrollerRef** 为手柄引用；**strength** 为振动强度（建议不要取太大，0.1~0.3 即可）；**duration** 为振动持续时间（以秒为单位）；**interval** 为本次持续振动中每个单脉冲的间隔时间（以秒为单位），具体来说一个**持续 1s** 的振动，可由 **20 个间隔 0.05s** 的单脉冲组成，也可由 **50 个间隔 0.02s** 的单脉冲组成。
2. controllerRef 是 **VRTK_ControllerReference** 类，它可通过其下的静态函数 **GetControllerReference(controller)**得到，其中 controller 为对应的手柄物体（即项目中的 leftController 或 rightController）。

4.2 方向盘旋转

之前提到了我们可以通过手柄的扳机控制油门与刹车，通过手柄的位置模拟汽车的转向。但现在还有一点美中不足的地方，那就是汽车的方向盘，它并**不随着手柄的移动而变动角度**。这使得游戏的真实性有所下降。

现在，找到 **SteeringWheelControl** 脚本，将其绑定在 **Car → CarModel → CarSteeringWheel** 上，**编写代码使得方向盘与手柄计算出的方向同步**，记得在检视板（Inspector）上进行必要的初始化操作。

提示：1. 不要忘记你在**章节 3.3** 所学到的知识。
2. **章节 2.1** 中提到的 **VRCarUseControl** 脚本可以给予你帮助，你可以直接使用其中的 **degree** 变量。
3. 在该项目中，方向盘只沿**本地坐标系的 z 轴**进行旋转。

4.3 使用触摸板控制汽车的行动

在本次 Lab 的编写中，对于汽车的控制方式，我们考虑了多种可行的设计方案。而最终，我们采取了使用手柄模拟方向盘的这种最贴近真实的交互方案。不过现在，为了考核你对代码的理解，**我们需要你实现另一种方案：使用触摸板控制**。具体细节：

使用触摸板（Touchpad）控制汽车的驾驶，**触摸左手触摸板的左右**分别对应**汽车方向的向左、向右**，**触摸右手触摸板的上下**分别对应**汽车的加速、减速**。

我们希望**允许新旧两种交互模式同时存在**，因而需要按键进行切换，即按下一个手柄按键（使用按键 **ButtonTwo**）时，可以在新旧两种交互方式间切换。

请在 **VRCarUserControl** 中直接进行更改，你可能会需要使用如下的变量进行辅助（*Line 31*，*Line 46*），其中公有变量 **inTouchPadMode** 用于切换这两种交换模式，私有变量 leftAxisX 与 rightAxisY 分别用于记录左手 Touchpad 的 X 值（横向，-1~1）以及右手 Touchpad 的 Y 值（纵向，-1~1）。

```
public bool inTouchPadMode = false;

private float leftAxisX = 0, rightAxisY = 0;
```

提示：1. 参照**章节 2.2** 分别为需要的**代理事件**注册对应的**自定义函数**
2. 编写自定义函数，包括**左手柄 Touchpad 轴值改变时**调用的函数，**右手柄 Touchpad 轴值改变时**调用的函数，以及 **ButtonTwo 按键被按下时**调用的函数。
3. 在 Update 中使用记录的值更新 **h**，**v** 以及 **degree**。

5 进阶要求（加分项）

本次实验进阶要求为**加分项**，且为开放性设计，不设置标准需求与结果。**强烈建议**各位同学先完成前面**所有的实验内容**，并通过实际 VR 设备测试后，再开始本阶段的内容。具体的，在使用 VR 设备测试前，你需要进行针对性的 **VR 底层 SDK 的配置工作**，请参见**第 6 章节**的相关内容。

当你确保完成了前面所有内容后，你可以尝试如下进阶挑战（完成一项即可）。在开发时，建议使用 VRSimulator 调试，并在最后再通过对应 VR 设备进行测试。

5.1 抓取功能

之前，我们实现了触摸（Touch）与使用（Use）这两个交互功能，但并没有讲关于**抓取（Grab）功能**的实现。在场景中，我们有许多黄色的障碍物方块，**请试着查询一些资料，完成对该类方块的操作，包括但不限于：**

a. 触摸方块使其高亮；
b. 触摸的同时**按住 Grip 键**抓取方块，松手则掉落；
c. 在抓取方块时，将其缩小为一定大小，松手时恢复原有大小；
d. 当你保持抓取时，方块会持续播放特殊效果（如旋转、变色、粒子效果等），当你释放后则不会。
e. 其他你能想到的与抓取、触摸或使用相关的、有趣的交互设计。

5.2 传送功能

由于 VR 设备的活动场地极为有限，因而传送功能是 VR 里最为常用的一种“代步工具”。由于篇幅所限，这里我们没有介绍关于传送的内容，**请试着查询一些资料，完成有关传送的功能设计，包括但不限于：**

a. 按下 Touchpad 来唤出传送射线，然后松手来传送到对应位置；
b. 可以传送到高处（即传送时考虑高度变化）
c. 设置一些禁止传送的物体或区域，例如禁止传送至障碍物方块上等
d. 改变游戏逻辑：在车内时，只可以进行驾驶，不可以传送，此时人的位置随车更新；在车外时，只可以传送，不可以驾驶，此时人的位置不随车更新。
e. 其他你能想到的与传送相关的、有趣的交互设计。

6 针对各 VR 设备的配置

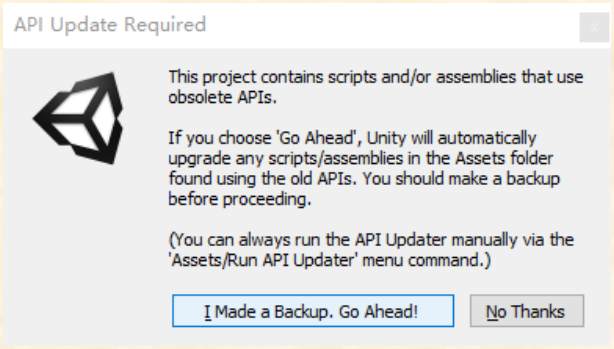
虽然 VRTK 提供了通用的开发接口，但由于底层接口的复杂性，以及各种设备开发所需的插件的版本更迭问题，我们在使用之前，要依据我们目标设备的不同进行修改与配置。在本次实验中，为了节省时间，我们**建议同学们始终选择一种设备**进行调试。

为了保险起见，各位同学在开始针对 VR 设备进行配置前，请先导出一个**仅包含项目与 VRTK 的 Unity package 作为 backup**，将其命名为 **Lab-primary-noSDK**。

6.1 SteamVR

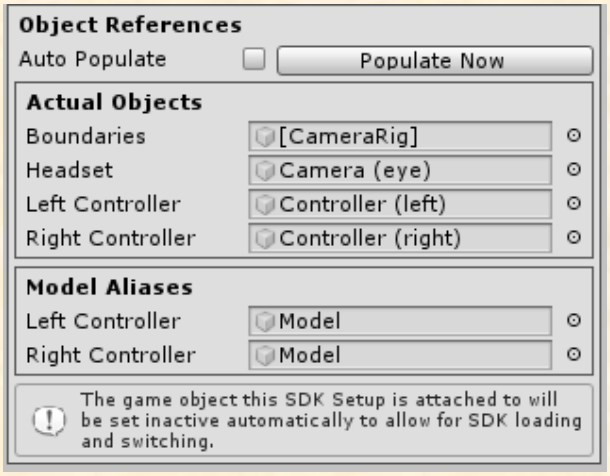
如果使用 HTC Vive 调试程序，请导入附件 **SteamVR.Plugin.unitypackage**。由于目前官方版本的 **VRTK 3.3.0** 不支持 **SteamVR 2+**的 SDK，我们这里导入的是 **SteamVR 1.2.3** 的老版本。

在导入过程中，可能出现如下对话框。这是由于 **SteamVR 1.2.3** 版本较低，一些 Unity API 已经弃用或更改。选择 **Go Ahead**，Unity 会帮助你自动转换。



接下来，我们需要进行场景上的设置了。在层级视图中找到 **[VRTK_SDKManager] → SteamVR**，由于版本匹配问题，VRTK 这里使用的 **[CameraRig]**与 SteamVR 提供的**[CameraRig]**并不是同一个物体，直接使用会出现**头盔视野旋转加倍**的现象（物理旋转 180° 对应 VR 场景中旋转 360°）。删除该 **[CameraRig]**，然后在 **Assets** 中找到 **SteamVR → Prefabs → [CameraRig]**，将该预制体绑定在 SteamVR 物体下，并 Unpack 该预制体。

然后不要忘了重新设置 SteamVR 的 **VRTK_SDKSetup** 脚本，如下图所示。注意 Model 分别为 **Controller(left)**与 **Controller(right)**下的 Model。



现在，全部设置已经完毕，你可以开始进行测试了。

6.2 Oculus

如果你选择使用 Oculus VR 设备调试该项目，请选择导入附件 **Oculus Integration.unitypackage**。在视图中找到**[VRTK_SDKManager] → Oculus**，由于版本匹配问题，**LocalAvatar** 无法起作用，因而我们直接删除它，但此时我们也就失去**绘制手柄模型**的功能。因而接下来，我们在项目 Assets 中找到 **Oculus → VR → Prefabs → OVRControllerPrefab**，将其直接拖入场景中，并 Unpack。选其下的 **OculusTouchForRiftLeftModel** 与 **OculusTouchForRiftRightModel**，分别对应地放在 **LeftHandAnchor** 与 **RightHandAnchor** 下，变成左右手柄模型，这样就解决了之前的问题，当然不要忘记**将两个手柄的 Transform 重置为 0**。

注意，我们修改了 Oculus 下的物体结构，因而回到**[VRTK_SDKManager] → Oculus**，将其下 **VRTK_SDKSetup** 脚本的 **ModelAliases** 的对应值改为刚才的两个手柄物体。

最后，由于 Oculus 设备设置时不会像 HTC Vive 一样**自动校准地面**，而是通过输入一个**使用者的高度**来反推地面位置，因而我们需要在代码中进行设置。找到**[VRTK_SDKManager] → Oculus → OVRCameraRig，将其 y 值设定为 1.8（本次实验中设定的使用者高度）**。

现在，全部设置已经完毕，你可以开始进行测试了。

6.3 Windows Mixed Reality

由于目前官方 VRTK 3.3.0 对于 WMR 设备的支持并不完善，我们并不推荐各位同学使用该设备进行调试。但若实验当天设备使用饱和，我们会将该设备作为备选调试方案来使用。

如果你使用 Windows MR 设备调试该项目，请选择导入附件 **VRTK-Windows-MR-Extension.unitypackage**。打开 **Build Settings**，将平台切换至 **Universal Windows Platform** 下。

到目前为止，我们了解到使用 VRTK 开发 WMR 项目时会存在**三个问题**：

a. **报错：ArgumentNullException: Value cannot be null**。这是 VRTK 3.3.0 的一个**已知 bug**，当手柄获取失败时，手柄信息为 null，然后 VRTK 将 null 直接传给了 Transform.Find 函数。请直接定位错误代码，将

```
  Tansform defaultAttachPoint =
controllerReference.model.transform.Find(elementPath);
```

改为：

```
  Transform defaultAttachPoint = elementPath != null ?
controllerReference.model.transform.Find(elementPath) :
controllerEvents.transform;
```

b. 无法使用 VRTK 获取**振动反馈**。在这种情况下请**直接联系助教**，由助教通过代码判断是否完成功能。

c. **手柄无法显示**。由于 VRTK 3.3.0 及其拓展插件已经有一段时间没有更新，其 glTF 模型库查询不到新的设备，因而不会进行渲染。一种折衷的办法是使用其他模型进行替代。在视图中找到**[VRTK_SDKManager] → WindowsMR → [WindowsMR_CameraRig] → ControllerManager**。你可以在其下的 **MotionControllerVisualizer** 脚本中，使用模型预制体定义 **AlternateLeftController** 与 **AlternateRightController**；或是直接将手柄模型移动到 **Controller(Left)**与 **Controller(Right)**下成为子物体（你可以用 Oculus 的手柄模型临时代替）。

最后，由于 Windows MR 在初始设置时**不会检测地面**，你需要根据实际情况，调整**[WindowsMR_CameraRig]**的高度。一种简单的做法是将头盔放置在地上，通过观察 **Head** 的 Y 轴高度来对应调整**[WindowsMR_CameraRig]**的偏移。

现在，全部设置已经完毕，你可以开始进行测试了。在日后的项目中，如果确定要使用 WindowsMR 设备进行开发，我们推荐使用 **Windows MRTK 2.0.0 RC1** 或是在 Git 上使用 **VRTK 4 Beta** 版本进行开发。

最终内容提交

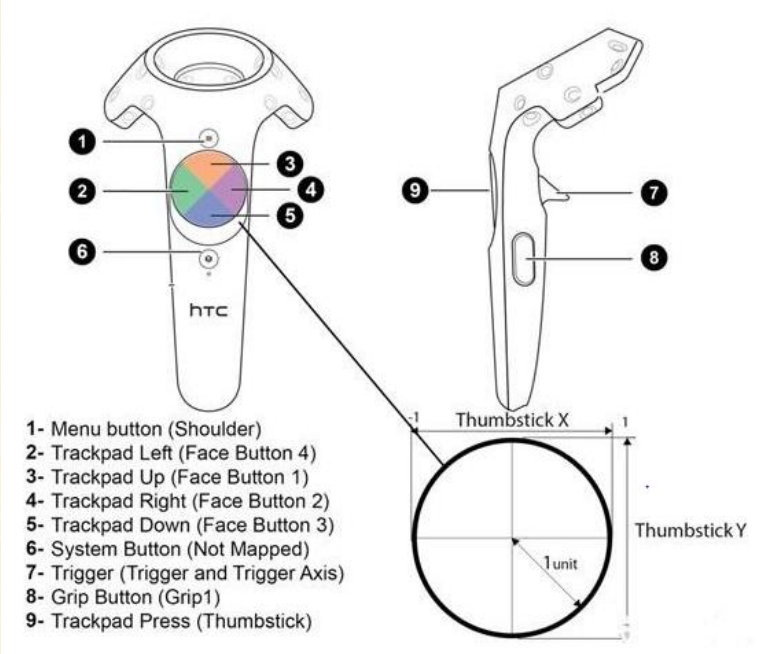
请提交一个命名格式为**组长学号_组长姓名全拼_Lab2.xxx** 的压缩包，内容以完成程度不同可分为：

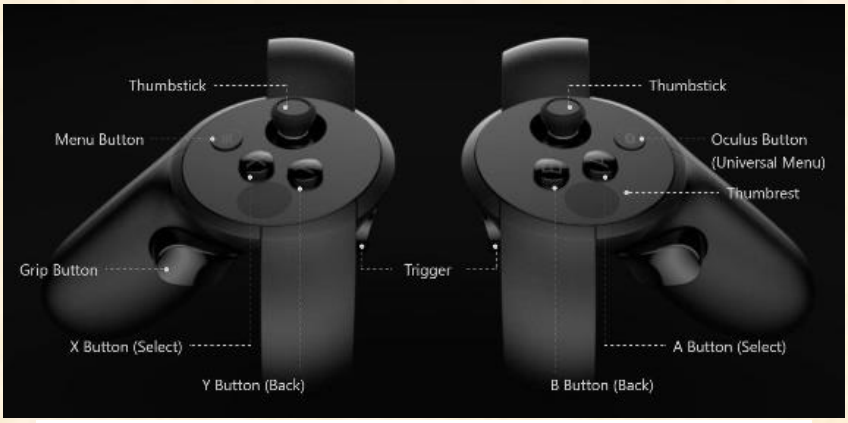
a. **未实现任何进阶要求**：直接提交无 SDK 的 backup 版本的 Unity package 即可（命名为 **Lab-primary-noSDK**），并辅以 **Readme 文档**来简要说明**完成情况，**助教将通过**代码**与**场景文件，**对 Readme 文档所提内容进行评判。

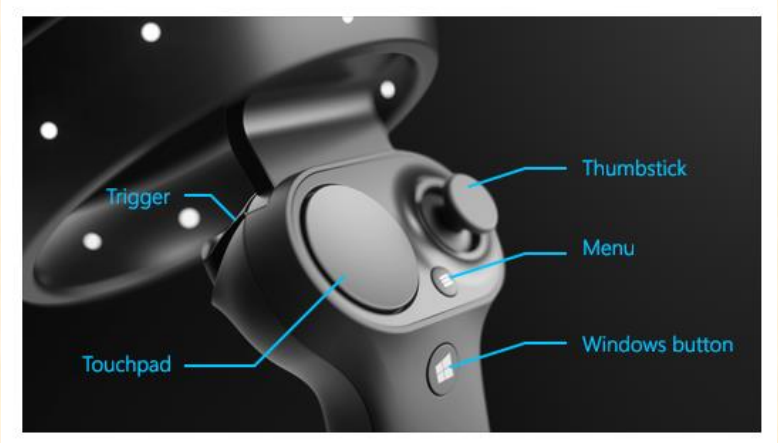
b. **实现部分/全部进阶要求**：提交最终版本（包含 SDK）的 Unity package（命名为 **Lab-final**），并辅以 **Readme 文档**来简要说明**测试用设备类型，基础完成情况**与**加分项完成情况**。助教将通过**实际运行效果，代码**与**场景文件，**对 Readme 文档所提内容进行综合评判。

附录 1——手柄按键

如下是各款手柄的按键图，从上到下分别是 HTC，Oculus 以及 Windows MR。







接下来我们给出各款手柄的按键在 VRTK 中的对应按键名称：

VRTK	HTC Vive	Oculus	Windows MR
Trigger	Trigger	Trigger	Trigger
Grip	Grip	Grip	Grip
Touchpad	Trackpad(Thumbstick)	Thumbstick	Touchpad
Touchpad Two	——	——	Thumbstick
Button One	——	Button A/X	——

Button Two	Menu	Button B/Y	Menu
Start Menu	——	Menu	Menu

附录 2——ControllerInteractionEventArgs

以下为 ControllerInteractionEventArgs 类的参数，以及其含义说明：

```
public VRTK_ControllerReference controllerReference; // 手柄引用
public float buttonPressure;        // 按键按下程度（-1~1）
public Vector2 touchpadAxis;        // 触摸板触碰的轴值（2D,-1~1）
public float touchpadAngle;         // 触摸板触碰点的角度（0~360）
public Vector2 touchpadTwoAxis;     // 二号触摸板触碰的轴值（同上）
public float touchpadTwoAngle;      // 二号触摸板触碰点的角度（同上）
```

附录 3——VRTK_ControllerEvents 代理事件

VRTK_ControllerEvents 代理事件数量非常多，对应着之前提到的所有按键，以及各种按键行为。以下为对各种按键行为的描述：

a. Pressed：当按键被按下一半以上（对应 Released）
b. TouchStart：当按键刚刚开始被按下（对应 TouchEnd）
c. HairlineStart：当按键按下超过 Hairline 阈值（对应 HairlineEnd）
d. Clicked：当按键完全被按下（对应 Unclicked）
e. AxisChanged：当轴值发生改变
f. SenseAxisChanged：当轴值发生微小改变