概览：

在VR项目中，我们需要在用户”凝视“某个物体时将其激活。在VRSamples中，我们构建了一个简单的，可拓展的轻度系统，让用户跟场景中的物体进行交互。其中包含了三个主要的脚本文件：VREyeRaycaster,VRInput和VRInteractiveItem，下面将对这三个重要的类进行简要的介绍。相关的源代码也做了注释。

VREyeRaycaster

该脚本需要和Main Camera绑定。在每次调用Update()时，该脚本都会使用Physics.Raycast来投射一条射线，以确认该射线是否命中任何碰撞物体。使用该脚本还可以排除特定的layers-在某些场景中，我们可能为了便于操作把所有的可交互对象移到一个单独的层。

如果某个碰撞体被射线命中，那么该脚本将尝试在GameObject上找到一个VRInteractiveItem组件。

C#脚本：

```
VRInteractiveItem interactible = hit.collider.GetComponent<VRInteractiveItem>();  //attempt to get the VRInteractiveItem on the hit object
```

从这里我们就可以判断用户究竟在”凝视“哪个物体，或是停止”凝视“某个物体。如果用户开始或停止”凝视“某个物体，那么我们就可以进行一些处理，比如调用一个方法。

VRInput

VRInput是个简单的类，可以判断用户在GearVR上（或是使用DK2时在PC上）所进行的一些简单操作，比如滑动、触碰、或双触。

我们可以直接在VRInput上注册事件：

C#脚本：

```
public event Action<SwipeDirection> OnSwipe;
// Called every frame passing in the swipe, including if there is no swipe.

public event Action OnClick;
// Called when Fire1 is released and it's not a double click.

public event Action OnDown;                     // Called when Fire1 is pressed.

public event Action OnUp;                       // Called when Fire1 is released.

public event Action OnDoubleClick;              // Called when a double click is detected.

public event Action OnCancel;                   // Called when Cancel is pressed.
```

关于注册事件的更多信息，请参考这里。http://unity3d.com/learn/tutorials/modules/intermediate/scripting/events

VRInteractiveItem

我们可以把该组件添加到任何希望在VR场景中进行交互的GameObject上，在该物体上需要绑定一个碰撞体。

对此我们可以选择注册六种不同的事件：

C#脚本：

```
public event Action OnOver;             // Called when the gaze moves over this object

public event Action OnOut;              // Called when the gaze leaves this object

public event Action OnClick;
// Called when click input is detected whilst the gaze is over this object.

public event Action OnDoubleClick;
 // Called when double click input is detected whilst the gaze is over this object.

public event Action OnUp;
// Called when Fire1 is released whilst the gaze is over this object.

public event Action OnDown;
// Called when Fire1 is pressed whilst the gaze is over this object.
```

此外还有一个布尔值用于判断用户是否”凝视“在当前物体上?

C#脚本：

```
public bool IsOver
{
    get{ return m_IsOver; }             // Is the gaze currently over this object?}
```

我们也可以创建自己的脚本对以上事件作出响应。下面是一个简单的示例：

C#脚本：

```
using UnityEngine;
using VRStandardAssets.Utils;

namespace VRStandardAssets.Examples
{
    // This script is a simple example of how an interactive item can
```

```
// be used to change things on gameobjects by handling events.
public class ExampleInteractiveItem : MonoBehaviour
{
    [SerializeField] private Material m_NormalMaterial;
    [SerializeField] private Material m_OverMaterial;
    [SerializeField] private Material m_ClickedMaterial;
    [SerializeField] private Material m_DoubleClickedMaterial;
    [SerializeField] private VRInteractiveItem m_InteractiveItem;
    [SerializeField] private Renderer m_Renderer;


    private void Awake ()
    {
        m_Renderer.material = m_NormalMaterial;
    }
```

如果想看到实际的示例，不妨看看VRSampleScens/Scens/Examples/的InteractiveItem场景。

SelectionRadial和SelectionSlider

我们同时利用了radial选择条（SelectionRadial），以及选择滑动条(SelectionSlider)，这样用户就可以按住Fire1来确认某个交互：

当按下输入键时，选择条会进行填充，并在填充完整后分发OnSelectionComplete或OnBarFilled事件。关于此部分的代码，可以在SelectionRadial.cs和SelectionSlider.cs中找到，并进行了详细的注释。在VR的世界里，从用户交互的角度看，用户需要时刻知道自己在做什么，而且可以掌控一切。通过这种held input的确认输入方式，可以确保用户不会出现误操作。

VR Sample项目中的交互示例

现在让我们来一起看看VR Sample项目中的部分交互示例。我们将提到每个场景中所使用的交互方式，以及具体实现的方式。

Menu 场景中的交互

每个menu场景都包含了几个组件，其中我们需要重点关注的是MenuButton,VRInteractiveItem和Mesh Collider。

MenuButton组件订阅了VRInteractiveItem组件上的OnOver和OnOut事件，这样当十字准星移到menu上时，selection radial会出现。当用户的实现离开菜单选项时，selection radial会消失。而当selection radial可见，且用户按住Fire1键时，则radial会自动填充：

C#脚本：

```
private void OnEnable ()
```

```
{
    m_InteractiveItem.OnOver += HandleOver;
    m_InteractiveItem.OnOut += HandleOut;
    m_SelectionRadial.OnSelectionComplete += HandleSelectionComplete;
}


private void OnDisable ()
{
    m_InteractiveItem.OnOver -= HandleOver;
    m_InteractiveItem.OnOut -= HandleOut;
    m_SelectionRadial.OnSelectionComplete -= HandleSelectionComplete;
}


private void HandleOver()
{
    // When the user looks at the rendering of the scene, show the radial.
    m_SelectionRadial.Show();

    m_GazeOver = true;
}


private void HandleOut()
{
    // When the user looks away from the rendering of the scene, hide the radial.
    m_SelectionRadial.Hide();

    m_GazeOver = false;
}


private void HandleSelectionComplete()
{
    // If the user is looking at the rendering of the scene when the radial's selection finishes,
activate the button.
    if(m_GazeOver)
        StartCoroutine (ActivateButton());
}


private IEnumerator ActivateButton()
{
    // If the camera is already fading, ignore.
    if (m_CameraFade.IsFading)
        yield break;

    // If anything is subscribed to the OnButtonSelected event, call it.
```

```
    if (OnButtonSelected != null)
        OnButtonSelected(this);

    // Wait for the camera to fade out.
    yield return StartCoroutine(m_CameraFade.BeginFadeOut(true));

    // Load the level.
    SceneManager.LoadScene(m_SceneToLoad, LoadSceneMode.Single);
}
```

让我们来看看Selection Radial的部分示例，注意截图中间的粉色元素：

当用户“凝视”菜单选项时，空白的Selection Radial可见。

Selection Radial 填充（当用户“凝视”菜单选项，且按下fire1输入键）

在整个示例项目中，我们尝试用同样的风格，也就是使用bar和radial以固定的速度进行填充。在此建议大家在开发自己的VR项目时注意到这一点，因为交互设计中的连贯性对用户很重要，特别是对于VR这种新媒介。

Maze场景中的交互

Maze（迷宫）游戏中提供了一个桌面式的交互示例，其中我们可以指引游戏角色到出口，并避免触发炮塔。

在选择角色的目的地时，会出现一个目的地标记，同时还会显示一个角色的路径。玩家可以通过在触摸板上使用swipe，按下方向键，或是使用游戏操纵杆上的左键来旋转视图。

在MazeFloor游戏对象上关联了MeshCollider和VRInteractiveItem，从而允许在VR场景中进行交互：

MazeCourse 游戏对象是一个parent对象，其中包含了MazeFloor和MazeWalls GameObjects，这两个对象依次包含了迷宫布局中的几何信息。

MazeCourse关联了一个MazeTargetSetting脚本，其中包含了对MazeFloor对象上VRInteractiveItem组件的引用。

MazeTargetSetting订阅了VRInteractiveItem上的OnDoubleClick事件，随后会分发OnTargetSet事件。该事件将把十字准星的Transform作为参数：

C#脚本：

```
public event Action<Transform> OnTargetSet;                    // This is triggered when a
destination is set.
```

```
private void OnEnable()
{
    m_InteractiveItem.OnDoubleClick += HandleDoubleClick;
}


private void OnDisable()
{
    m_InteractiveItem.OnDoubleClick -= HandleDoubleClick;
}


private void HandleDoubleClick()
{
    // If target setting is active and there are subscribers to OnTargetSet, call it.
    if (m_Active && OnTargetSet != null)
        OnTargetSet (m_Reticle.ReticleTransform);
}
```

MazeCharacter游戏对象上的Player组件和MazeDestinationMarketGUI游戏对象上的DestinationMarker组件都会订阅该事件，并作出相应的响应。

游戏角色可以使用Nav Mesh systems在迷宫中进行路径判断。Player组件使用HandleSetTarget函数来判断Nav Mesh Agent到十字准星间的方向，并更新Agent的轨迹-角色路径的视觉渲染。

C#脚本：

```
private void HandleSetTarget(Transform target)
{
    // If the game isn't over set the destination of the AI controlling the character and the trail showing its path.
    if (m_IsGameOver)
        return;

    m_AiCharacter.SetTarget(target.position);
    m_AgentTrail.SetDestination();
}


```

DestinationMarker可以将标记移动到Reticle的Transform位置：

C#脚本：

```
private void HandleTargetSet(Transform target)
{
    // When the target is set show the marker.
    Show();
```

```
    // Set the marker's position to the target position.
    transform.position = target.position;

    // Play the audio.
    m_MarkerMoveAudio.Play();

    // Play the animation on whichever layer it is on, with no time offset.
    m_Animator.Play(m_HashMazeNavMarkerAnimState, -1, 0.0f);
}
```

在下图中可以看到reticle,目的地标记，玩家和轨迹。
迷宫中的切换开关也是在VR中和物体进行交互的示例，其中用到了Collider，以及VRInteractiveItem，和SelectionSlider三个类。

正如上图中显示的，和其它交互对象一起，SelectionSlider脚本会监听由VRInteractiveItem和VRInput所分发的事件。

C#脚本：

```
private void OnEnable ()
{
    m_VRInput.OnDown += HandleDown;
    m_VRInput.OnUp += HandleUp;

    m_InteractiveItem.OnOver += HandleOver;
    m_InteractiveItem.OnOut += HandleOut;
}
```

Flyer场景中的交互

Flyer场景是一个计时”无尽飞行”游戏，在其中玩家可以通过四处看来引导飞船的方向，并使用Fire1输入键进行射击，通过击中陨石或是引导飞船穿越空中的门来得分，跟Pilotwings或Starfox这两款游戏有点类似。

在交互方面，Flyer使用了更简单的方式，也就是让FlyerLaserController订阅VRInput的OnDown事件，从而发射激光。

C#脚本：

```
private void OnEnable()
{
    m_VRInput.OnDown += HandleDown;
}
```

```
private void HandleDown()
{
    // If the game isn't running return.
    if (!m_GameController.IsGameRunning)
        return;

    // Fire laser from each position.
    SpawnLaser(m_LaserSpawnPosLeft);
    SpawnLaser(m_LaserSpawnPosRight);
}
```

Shooter180和Shooter360场景中的交互（Target Gallery/ Target Arena）。

在VR Samples包含了两个射击游戏，其中一个是回廊射击游戏，玩家在180度视角的走廊中对潜在目标射击。另外还有一个竞技场射击游戏，玩家被类似X战警场景的潜在目标包围。

这两款游戏中的每个目标对象都有一个Collider,VRInteractiveItem和ShootingTarget。

ShootingTarget组件订阅了VRInteractiveItem的OnDown事件，以判断目标是否被击中。该方法适用于瞬间命中（比如激光枪这种）的设定，如果要展示子弹时间，我们就需要考虑其它解决方案了。

现在我们应该对基本的VR交互组件有了大概的印象，包括任何在VR Samples项目中具体使用这些组件。现在让我们来看看VR Samples项目中如何使用gaze（凝视）和reticles（十字星）。

GAZE（凝视）

在VR应用中判断用户正在看什么很重要，可能是用于判断用户和游戏对象的交互，或是触发一个动画，也可能是向目标发射子弹。我们将VR中“看”这个动作定义为gaze（凝视），而在后续的教程中我们将频繁使用这个词。

考虑到目前大多数HMD头戴设备还不支持眼部追踪，因此我们只能估计用户的gaze（凝视）。透镜的扭曲意味着用户正看着正前方，有一个简单的解决方案。正如在概览中提到的，我们只需要从摄像机的中心发射一条射线，然后找到这条射线所碰撞的物体即可。当然，这就意味着所有要被碰撞（或是需要通过“凝视”进行交互）的对象都必须关联一个Collider组件。

Reticle（十字准星）

十字准星用于辅助标记用户视野的中心。十字准星的样式可能是简单的点，也可能是一个十字准线，具体形式取决于项目需求。

在传统的3D游戏中，十字准星被设置为空间中的固定点，比如通常是屏幕的中央。但是在VR中使用十字准星变得非常复杂：当用户在VR环境中四处观望时，双眼将汇集在靠近摄像机的物体上。如果十字准星处在一个固定的位置，那么用户会看到两个准星：我们在现实世界里面可以轻易模仿这种效果。把某个手指放在眼睛前面，然后聚焦到近处和远处的物体上。当我们聚焦在这个手指上时，就会看到两个背景，反之亦然。这就是传说中的“自发复视”现象。

为了避免用户在查看周围环境和注视不同距离的物体时看到两个准星，我们需要将准星放到3D空间的同一个点，也就是用户所关注对象的表面。

将准星放在空间的这个点意味着从远处看准星将非常小，当靠近时会变大。为了让准星的大小不随距离发生变化，我们需要根据它到摄像机的距离对其进行缩放。

为了说明这一点，我们从Examples/Reticle场景中找了一些例子，展示了处于不同距离和比例的准星。

准星放置在靠近摄像机的物体上：

准星放置在稍远的物体上：

准星放置在更远的物体上：

根据所处的位置和自身比例，用户在任何距离上看到的准星大小都是相同的。

如果没有击中任何对象，那么我们只需把准星放到一个预设的距离上。在室外环境中，可能会放在摄像机的Far clip plane前面，在室内场景中可能会近得多。

将十字准星渲染到其它游戏对象的表面

如果十字准星恰好和某个对象的位置相同，那么准星可能会嵌入到临近的对象中。

为了解决这个问题，我们需要确保将准星渲染到场景中所有对象的前面。在VR Samples中，我们提供了一个shader，基于Unity现有的名为UIOverlay.shader的”UI/Unlit/Text” shader。在选择某个材质的shader时，可以在”UI/Overlay”中找到。

这个shader对UI 元素和文本都适用，会在场景中其它物体的前面绘制。

将准星和场景中的游戏对象对齐

我们希望准星的旋转方向和它所命中的对象的法线相匹配。通过RaycastHit.normal就可以实现这一点，以下是具体的实现代码：

C#脚本：

```
public void SetPosition (RaycastHit hit)
{
   m_ReticleTransform.position = hit.point;
   m_ReticleTransform.localScale = m_OriginalScale * hit.distance;

   // If the reticle should use the normal of what has been hit...
   if (m_UseNormal)
      // ... set it's rotation based on it's forward vector facing along the normal.
      m_ReticleTransform.rotation = Quaternion.FromToRotation (Vector3.forward, hit.normal);
   else
      // However if it isn't using the normal then it's local rotation should be as it was originally.
      m_ReticleTransform.localRotation = m_OriginalRotation;
}
```

我们可以在Maze场景中看到这个action。

下图展示了准星如何匹配墙壁的法线：

下图展示了准星如何匹配地板的法线：

我们还提供了一个示例的Reticle脚本。该脚本可以跟VREyeRaycaster一起适用，从而将准星放置到场景的正确位置，并且可以选择跟所命中的对象法线帖齐。

以上内容都可以在VRSampleScens/Scens/Examples/中看到。

在VR项目中头部的旋转和位置

在头戴设备中跟踪头部的旋转和位置可以用沉浸式的体验来感受周围环境，但同时也可以让对象根据这些数值所相应的相应。

为了获取这些数值我们需要用到VR.InputTracking类，并指定我们要访问的VRNode。为了获取头部的旋转，我们会希望用到VRNode.Head，而不是两只眼睛。想了解更多的信息，可以参考Getting Started with VR Development一文中的Camera Nodes。

使用头部旋转作为输入方式的可能应用是精细旋转菜单或是其它对象。在VRSampleScenes/Examples/Rotation场景中可以看到这一点的示例。
下面是ExampleRotation的脚本：

C#脚本：

```
// Store the Euler rotation of the gameobject.
var eulerRotation = transform.rotation.eulerAngles;

// Set the rotation to be the same as the user's in the y axis.
eulerRotation.x = 0;
eulerRotation.z = 0;
eulerRotation.y = InputTracking.GetLocalRotation(VRNode.Head).eulerAngles.y;
```

下面的图展示了游戏对象如何根据用户所注视的位置来进行旋转：

在Flyer游戏场景中，我们将看到太空飞船基于头部的旋转来调整自身位置，具体参考FlyerMovementController:

C#脚本：

```
Quaternion headRotation = InputTracking.GetLocalRotation (VRNode.Head);
m_TargetMarker.position = m_Camera.position + (headRotation * Vector3.forward) *
m_DistanceFromCamera;
```

在VR游戏中使用触摸板和键盘进行交互

Gear VR在头戴设备的侧边配备了一个触摸板。Unity把这个触摸板当做鼠标来使用，所以我们可以使用以下方法：
Input.mousePosition
Input.GetMouseButtonDown
Input.GetMouseButtonup

在使用Gear VR时，开发者可能会希望从触摸板中获取swipe数据。我们提供了一个名为VRInput的示例脚本，可以处理swipe,触碰和双触。此外它还支持方向键和键盘上的左Ctrl键（在Unity中的默认输入术语是Fire1），或者是鼠标上的左键，以此来处罚swipe和触摸。

在Unity Editor中，我们可能会希望使用DK2来测试Gear VR的内容。因为目前暂时无法直接从Unity直接关联到Gear VR进行测试。考虑到Gear VR的触摸板作用跟鼠标类似，我们可以考虑使用鼠标来模拟输入。当用户佩戴HMD设备时操控键盘会更容易，因此VRInput同时也会讲方向键操作处理成swipe，将Left-Ctrl（Fire1）处理成触碰。

在使用游戏手柄时，左侧的stick可以用作swipe，其中的某个按键可以用作触碰。

关于如何处理swipe，可以参考VRSampleScenes/Scenes/Examples/Touchpad

以下是ExampleTouchpad脚本，其中根据swipe的方向将AddTorque方法作用于一个刚体，从而让对象选装。

C#脚本：

```
using UnityEngine;
using VRStandardAssets.Utils;

namespace VRStandardAssets.Examples
{
    // This script shows a simple example of how
    // swipe controls can be handled.
    public class ExampleTouchpad : MonoBehaviour
    {
        [SerializeField] private float m_Torque = 10f;
        [SerializeField] private VRInput m_VRInput;
        [SerializeField] private Rigidbody m_Rigidbody;


        private void OnEnable()
        {
            m_VRInput.OnSwipe += HandleSwipe;
        }


        private void OnDisable()
        {
            m_VRInput.OnSwipe -= HandleSwipe;
        }


        //Handle the swipe events by applying AddTorque to the Ridigbody
        private void HandleSwipe(VRInput.SwipeDirection swipeDirection)
        {
            switch (swipeDirection)
            {
                case VRInput.SwipeDirection.NONE:
                    break;
                case VRInput.SwipeDirection.UP:
                    m_Rigidbody.AddTorque(Vector3.right * m_Torque);
                    break;
                case VRInput.SwipeDirection.DOWN:
                    m_Rigidbody.AddTorque(-Vector3.right * m_Torque);
                    break;
                case VRInput.SwipeDirection.LEFT:
                    m_Rigidbody.AddTorque(Vector3.up * m_Torque);
```

```
                break;
            case VRInput.SwipeDirection.RIGHT:
                m_Rigidbody.AddTorque(-Vector3.up * m_Torque);
                break;
        }
    }
  }
}
```

VR Samples项目中的VRInput示例

正如上面所提到的，我们所有的示例游戏都使用VRInput来处理触摸屏和键盘的输入。Maze游戏中的摄像机也会对swipe作出响应：

Maze

在这个场景中，CameraOrbit对swipe进行监听，从而允许对视点进行调整：

C#脚本：

```
private void OnEnable ()
{
    m_VrInput.OnSwipe += HandleSwipe;
}


private void HandleSwipe(VRInput.SwipeDirection swipeDirection)
{
    // If the game isn't playing or the camera is fading, return and don't handle the swipe.
    if (!m_MazeGameController.Playing)
        return;

    if (m_CameraFade.IsFading)
        return;

    // Otherwise start rotating the camera with either a positive or negative increment.
    switch (swipeDirection)
    {
        case VRInput.SwipeDirection.LEFT:
            StartCoroutine(RotateCamera(m_RotationIncrement));
            break;

        case VRInput.SwipeDirection.RIGHT:
            StartCoroutine(RotateCamera(-m_RotationIncrement));
            break;
    }
}
```

如果想了解为什么场景中的摄像机按照某个轨道运行，而不是旋转迷宫本身，可以参考Movement这篇文章。

在看完本文后，我们应该对VR Sample场景中的基本交互有更深入的理解。虽然还有其它方式来实现VR交互，但这种方法是最简单易行的。在下一篇文章中，我们将讨论不同类型的VR用户界面。如果在学习中有任何问题，可以到Unity 论坛中参与讨论。

对VR开发感兴趣的朋友可以通过邮件(eseedo@gmail.com)或微信(iseedo)联系我，希望跟大家一起学习。另外在我的网站(http://www.cylonspace.com)和博客(http://blog.sina.com.cn/eseedo)上也会放VR/AR开发的相关内容。