《游戏编程》课程设计之《鬼水怪谈》

开发文档

开发者：王孙洪

目　录

一、 项目概述

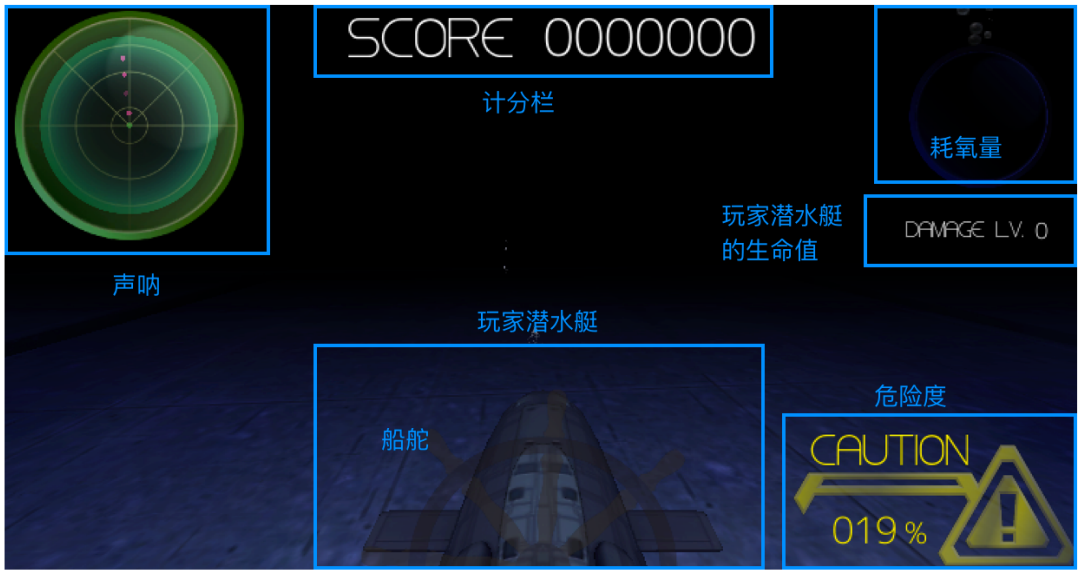
1. 项目简介

在生活中，我们通常都是根据声音判断周围环境中出现的人或物，Unity 中也具有 **3D 声音**功能，它可以通过声源和摄像机的位置关系，自动计算音量和声像（PAN，左右声道的音量平衡），并实时显示出声音传来的方向。通过这个 3D 声音特性启发灵感开发这一款游戏——《鬼水怪谈》。

这是一款轻量级恐怖心理冒险类游戏，玩家在潜水艇里，潜入海底寻找埋藏在深海之中的秘密。通过声呐和反射音寻找敌人位置，体验在黑暗的空间中只依靠声音前进并判断敌人的位置的紧张和刺激。

2. 玩法介绍

游戏主界面布局：



通过鼠标、键盘 B 键和空格键操作：

- 前后拖动鼠标可以控制速度；
- 左右拖动鼠标可以转弯；
- 通过 B 键发射鱼雷；
- 通过空格键切换声呐的类型。

主动声呐的使用：按住空格键，切换到主动声呐。通过主动声呐，可以在声呐上显示敌人或物体。但使用主动声呐时更容易被敌人发现。

危险值（CAUTION）：CAUTION 的值和玩家的潜水艇和敌舰的距离成反比，当玩家和敌机距离越近，CAUTION 值越高，最高 100%。

任务：每一关的任务都有所不同，例如：“击沉敌舰”、“寻找特定物品”等。玩家需完成任务后才能进入下一关。

二、 项目实现

1. 脚本一览

文件/脚本		说明
Player/	PlayerCollider.cs	玩家被鱼雷集中后的处理
	PlayerController.cs	玩家操作（旋转、发射鱼雷等）
	SonarCamera.cs	声呐对象进入声呐显示范围内时被显示，超出该范围时不被显示的提示事件（根据 Collider 判断）
Common/	ColorFader.cs	声呐上显示的点的淡出控制，显示和不显示
	Note.cs	控制对象发出的声音
Enemy/	EnemyBehavior.cs	控制敌舰的前进速度和转弯等动作
	EnemyCaution.cs	用于管理敌人的 CAUTION 值
Torpedo/	TorpedoBehavior.cs	控制鱼雷的行为（前进、超出范围时删除、对象的销毁）
	TorpedoGenerator.cs	生成鱼雷的脚本（敌人和玩家使用同样的脚本）
	TorpedoCollider.cs	鱼雷的碰撞处理（根据鱼雷的发射者和被击中者发出相应的消息）
UI/Sonar	ActiveSonar.cs	主动声呐
	SonarSwitcher.cs	主动声呐和被动声呐的切换
	SonarEffect.cs	主动声呐和被动声呐的纹理放大缩小的效果
Item/	ItemCollider.cs	物体和玩家接触时，物体发出的通知的内容
Airgage/	Airgage.cs	调整 Air 的上升状态（如根据 DamegeLv 增加上升值）
	AirgageBubble.cs	Airgage 的气泡效果设定（根据 DamegeLv 增加气泡数量）
UI/	Controller.cs	操舵轮的纹理显示以及旋转控制

本项目脚本较多，这里仅列出一些具有代表性的脚本，代码量共 4000 行***

2. 3D 声音的定位和控制

《鬼水怪谈》是一款只根据声音来探测敌人或物体位置的游戏。如果玩家听了 Unity 中 3D 声音的效果，就能够很明显地感觉到根据距离和方向的不同，实时声音也会不同。

Unity 3D 的声音具有如下**特性**：

① 声源和监听器之间的距离近则声音较大，距离远则声音较小

② 如果声源位于监听器的左侧，则左声道的声音较大，如果位于右侧，则右声道的声音较大

③ 声音接近监听器时音调变高，远离监听器时音调变低

这些特性主要与声源和监听器的位置有关。

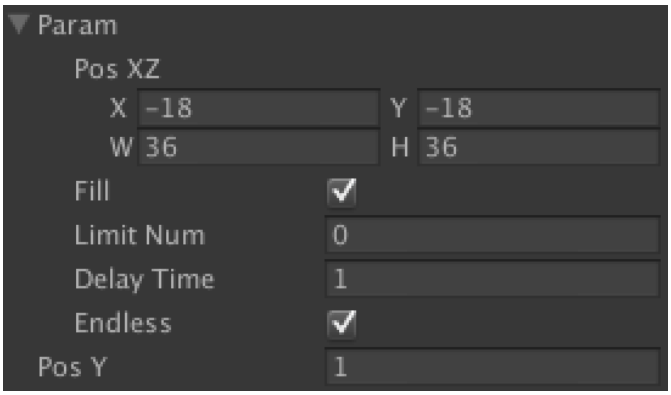
游戏声源：在 Unity 中使用 AudioSource 组件的游戏对象都可以做声源。

游戏监听器：听到声音的人或者用于录音的麦克风等物体。在 Unity 中 AudioListener 组件具有监听器的功能，游戏中将监听器同摄像机捆绑在一起。

在游戏中，玩家仅依靠声音判断方向，然后前往声源对象的位置，其中声源对象隐藏不显示在画面中，玩家成功到达对象所在的位置后将发出光效，之后游戏对象消失。

通过左右声道的音量差来确定声源方向称为**声像**（PAN）。声像只能产生音量差，玩家无法分辨前方和后方的区别，这种情况下可稍微前后移动，依靠多普勒效应导致的高音变化来判断。

在本项目中，声源属性值如下：



① PosXZ：生成对象范围。以（X，Y）为原点，在（Width，Height）范围内随机选取位置生成对象

② Fill：值为 true 时表示该范围内所有的位置都允许生成对象，值为 false 时则只能在 PosXZ 所确定区域范围的边缘部分生成对象

③ Limit Num：允许生成对象的最大数量

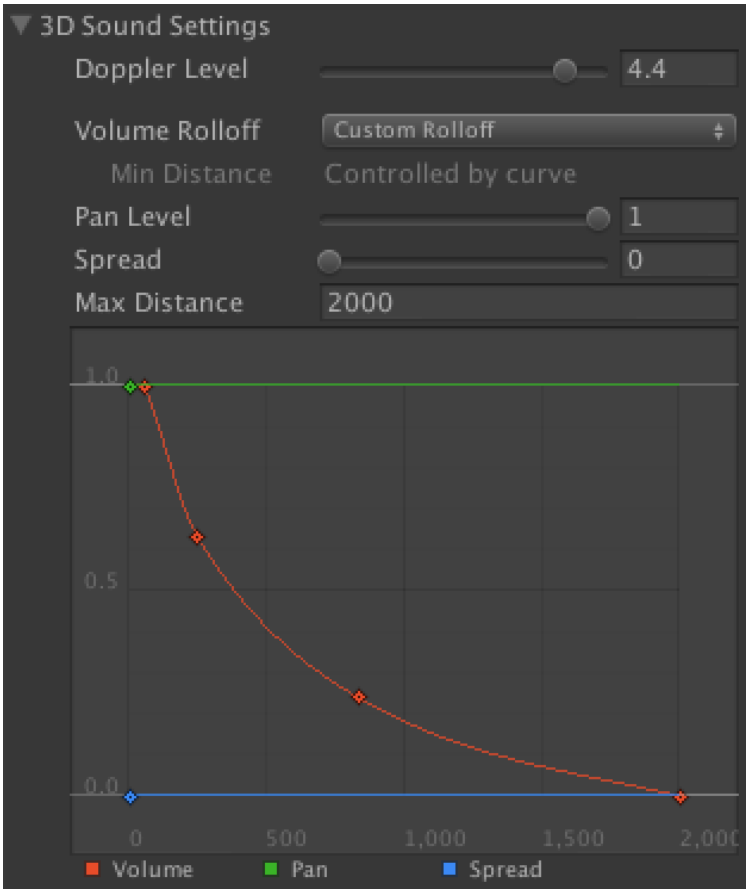
④ Delay Time：生成对象的时间间隔

⑤ Endless：值为 true 时，将持续生成对象，值为 false 时，在生成数量达到 Limit Num 时，将停止生成新对象

Unity 中的 3D 声音部分有很多可以由开发人员进行设置的参数，其中，对于定位而言非常重要的是距离衰减。

距离衰减描述的是远离声源时音量以何种规律降低。Unity 中可以通过表示距离和音量的关系的图形形状来调整距离衰减。

Unity 中检视面板有 3D Sound Setting，可以看到距离衰减图中的曲线：



3D 声音设置（3D Sound Settings）面板各属性如下：

平衡调整级别 （Pan Level）：设置多少，3D 引擎在音频源上有效果。

扩散 （Spread）：设置 3D 立体声或者多声道音响在扬声器空间的传播角度。

多普勒级别 （Doppler Level）：决定了多少多普勒效应将被应用到这个音频信号源（如果设置为 0，就是无效果）。

最小距离 （Min Distance）：在最小距离之内，声音会保持最响亮。 在最小距离之外，声音就会开始衰减。

最大距离 （Max Distance）：声音停止衰减距离。超过这一点，它将在距离侦听器最大距离单位，保持音量，不在衰减。

衰减模式 （Volume Rolloff）：声音淡出的速度有多快。该值越高，越接近侦听器最先听到声音（这是由图形决定）：

① 对数衰减（Logarithmic Rolloff）：当你接近的音频源，声音响亮，但是当你远离对象，声音下降显着快。

② 线性衰减（Linear Rolloff）：越是远离音频源的，你可以听到的声音越小。

③ 自定义衰减（Custom Rolloff）：根据你设置的衰减图形，来自音频源的声音行为。

在 Custom Rolloff 图形面板中，允许自由改变其形状，改变顶点的位置和斜度，其中有两个特点：

①近处的衰减比较剧烈

②远处的衰减较为缓和

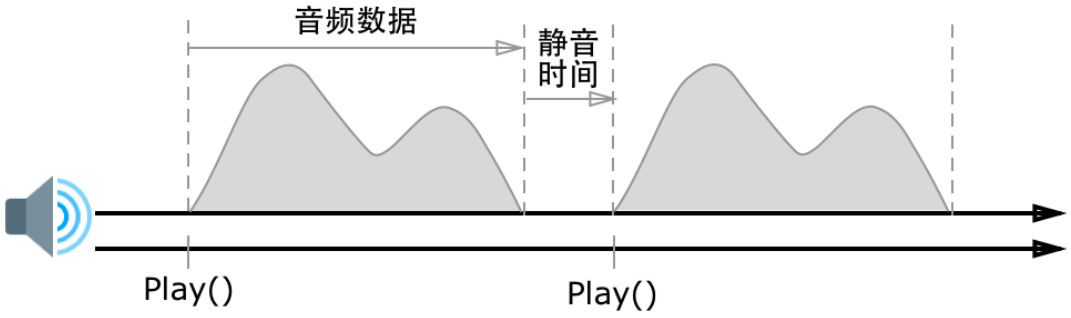
音量的变化对距离的变化越敏感，距离感就越容易把握。因为最终必须要定位到声源所在地，所以物体越接近声源，对其移动的正确性的要求就越高。与此相反，当物体距离声源较远时，相比距离导致的音量变化，确保物体即使距离声源很远也能够听到声音则尤为重要。因此我们将近处的距离衰减设置得强烈一些，远处的距离衰减设置得缓和一些。虽然距离较远时难于把握到声源的距离，不过只要能够确定方向，物体就可以朝着声源的方向移动。

在设置完毕 3D 声音的控制后，还需要让声音随着一定的时间间隔发出“滴咚、滴咚”的背景音。本项目采用单次音的音频素材。

相比循环音持续不断地发出声音，单次音会有一段静音时间。这样我们就可以设定静音的时长，给玩家制造一种失去声源方向的紧张感。

对音频数据的后半部分做消音处理，也可以达到制造静音时段的效果。不过，随着程序来控制静音时段，会更便于做细微的调整，在游戏中也可以根据情况来随时改变静音时段。

使用计时器进行计时，每经过一定的时间就通过 Play 方法播放声音。



按一定的时间间隔播放声音

图片制作：王孙洪

Note.Clock 方法

```
private void Clock(float step)
  {
    // (a)累加时间间隔（从播放声音起的时间）
    counter += step;
    // （b）计时器超过“播放间隔”后，再次播放声音
    if (counter >= interval)
    {
      audio.Play();
      // （c）清零计时器
      counter = 0.0f;
    }
  }
```

程序设计思路：

（a） 更新用于记录声音开始播放后所经时间的 counter 变量值。这里累加的 step 值是上一帧起到现在经过的时

间，该值可以用 Time.deltaTime 参数指定。

（b） counter 超过 interval 后，就意味着距离上次播放声音的时间已经超过了指定时间，再次播放声音。

（c） 最后将计时器的值重新设置为 0。

在物体和敌机等声源消失后，其发出的声音也将同时消失，这时，如果强行中断声音的播放，根据声音种类的不同有时候可能会产生噪音，于是，在声源对象消失后，让正在播放的声音慢慢**淡出**从而消失。

如果在播放声音时通过 Stop（）方法停止 AudioSource 组件，根据声音种类的不同可能会产生“啪嗒”的噪音。可以让声音消失的过程拉长，先慢慢降低音量，然后淡出，淡出处理可以通过在协程中慢慢降低音量来实现。

Note.Fadeout 方法

```
private IEnumerator Fadeout(float duration)
  {
    // 淡出
    float currentTime = 0.0f;
    float waitTime = 0.02f;
    //（a）淡出开始的音量
    float firstVol = audio.volume;
    //（b）在 duration 内循环
    while (duration > currentTime)
    {
       //（c）慢慢降低音量
      audio.volume = Mathf.Lerp(firstVol, 0.0f, currentTime / duration);
      //（d）中断处理一段时间
      yield return new WaitForSeconds(waitTime);
      currentTime += waitTime;
    }
    // 淡出处理完全结束后销毁对象
    if (hitEffector)
    {
      while (hitEffector.IsPlaying())
```

```
        {
            yield return new WaitForSeconds(waitTime);
        }
    }
    // 发送销毁对象消息
    transform.parent.gameObject.SendMessage("OnDestroyLicense");
}
```

程序设计思路：

（a） 首先记录下淡出开始时的音量值。

（b） duration 表示淡出过程中截止到音量变为 0 所经过的时长，currentTime 表示从开始淡出到现在所经过的时间。在 duration 内，while 循环将一直执行。

（c） **通过 Math.lerp 方法慢慢降低音量。**第 3 个参数表示补间率，它的值通过当前时间 currentTime 除以淡出的时长 duration 计算而来。

（d） 为了让音量慢慢降低，需要中断处理一段时间。

Math.Lerp 方法简介：

Unity 中提供插值与平滑的运算。插值的作用是根据比率 t 返回 from 到 to 之间的某个值。本项目运用 Math.Lerp 方法计算两个小数间的插值，实现声音的趋近平滑效果。

平滑插值算法伪代码：

要修改的数据 = Math.Lerp(要修改的数据, 目标值, Time.deltaTime * 变化率);

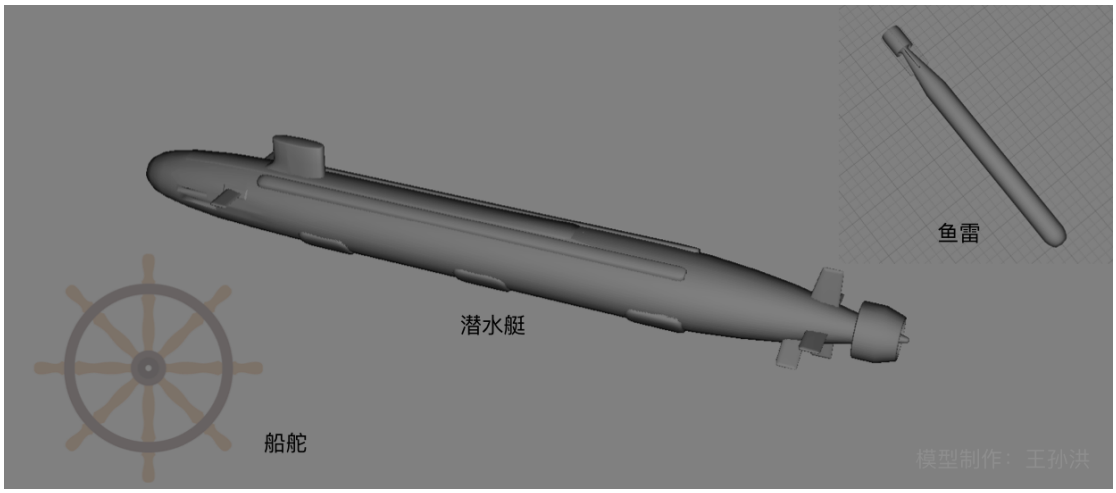
3. 潜水艇的操纵

在本项目的游戏中，为了应用“只依靠声音来确定位置”这一规则，选取潜水艇为交通工具，为了还原实际场景，潜水艇具备下面两个特性：

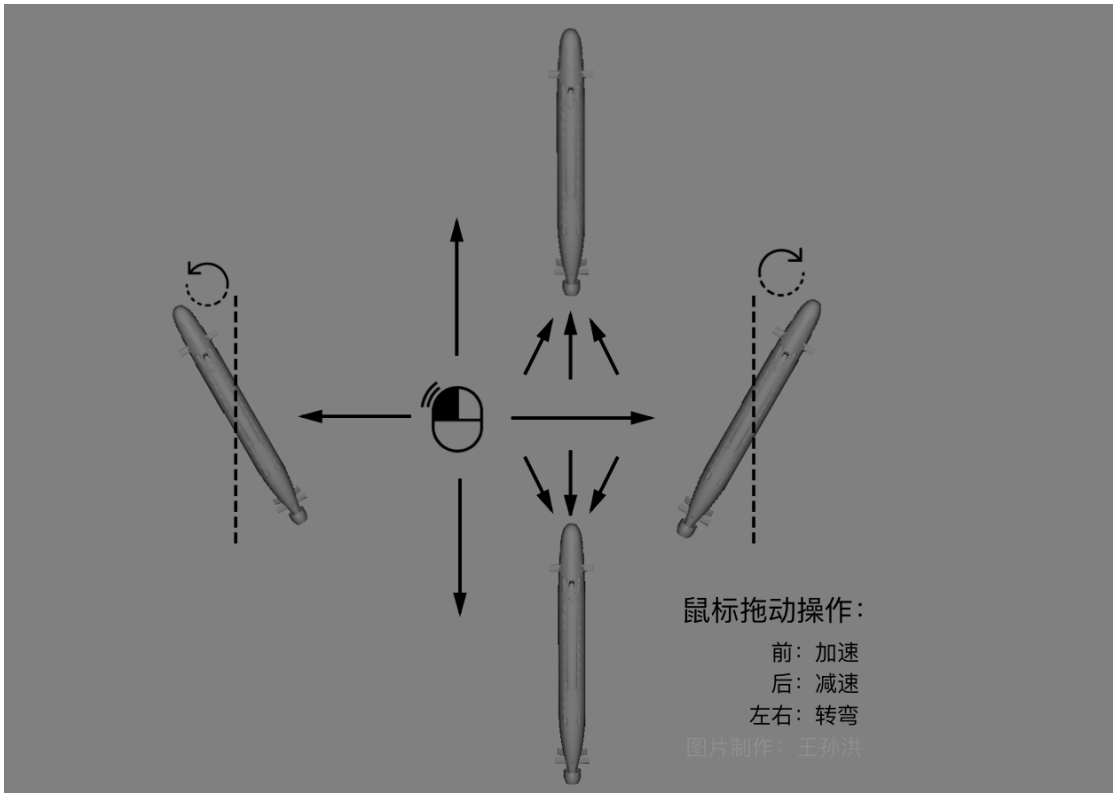
① 对玩家的操作不能快速地做出反应

② 只有持续地操作鼠标潜水艇才能转弯

由于水的阻力比空气大，所以在水中运动往往没有那么灵活。另外，因为潜水艇自身的重量导致其惯性大，所以不能很快停下来。而且前后细长的外形还会导致它在转弯时会遇到比前进时更大的阻力。



玩家的潜水艇可以通过点击鼠标的左键并拖动来操作。将鼠标往前拖动会加快前进的速度，往后拖动则会降低速度，左右拖动则可以实现转弯。



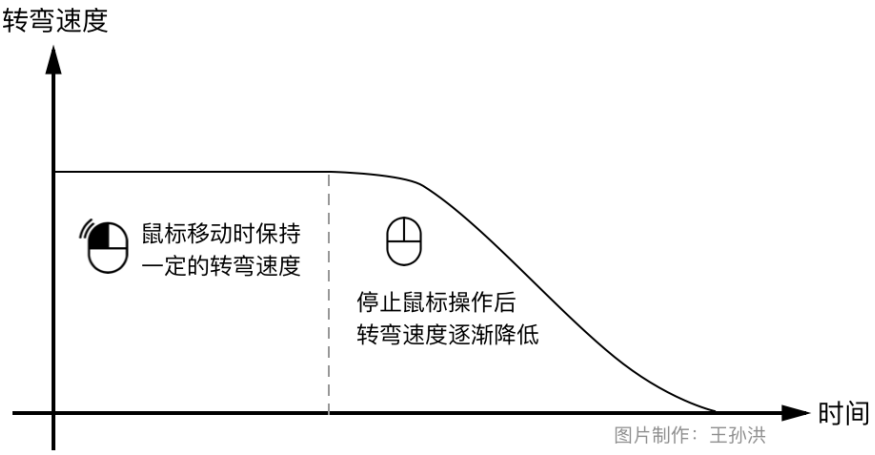
前后拖动控制速度和左右拖动控制转弯，在鼠标停下来时以及放开鼠标左键时，二者的行为有细微差别。

往前拖动提高速度后，即使停止拖动鼠标或者松开按键，潜水艇也能保持原速度继续前进。

转弯的情况下，左右拖动鼠标时潜水艇将持续转弯，停止操作鼠标后速度将慢慢变下，过一阵后将自动停止转弯。

如果松开鼠标按键，转弯速度的衰减将变缓，因为松开按键后，由于惯性所致潜水艇将继续保持转弯一段时间。这种情形还原了真实物理行为，给玩家真实的体验。

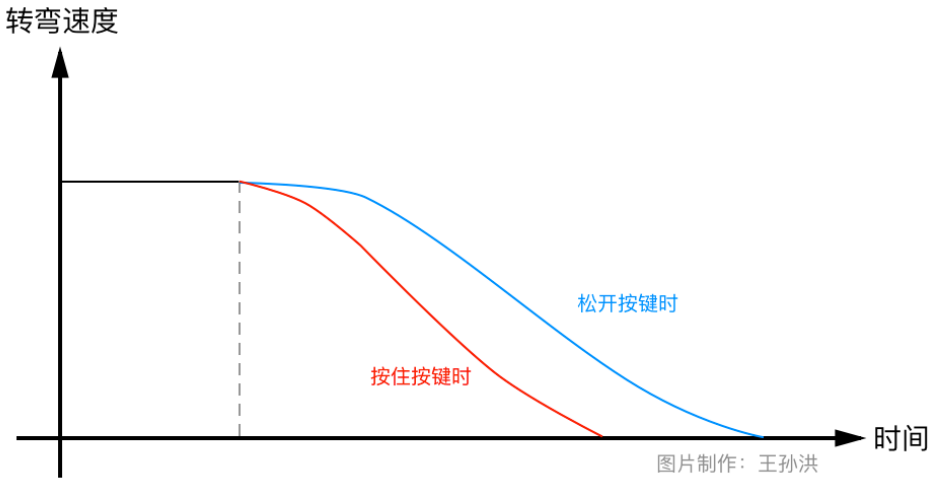
下图表示的是持续拖动鼠标一段时间后再放开时转弯速度的变化情况：



虚线处是停止鼠标操作的时刻。虚线左侧表示鼠标正在被拖动，转弯速度保持在一定的值。当停止鼠标操作后，也就是虚线右侧的区域，转弯速度逐渐降低直到变位 0。

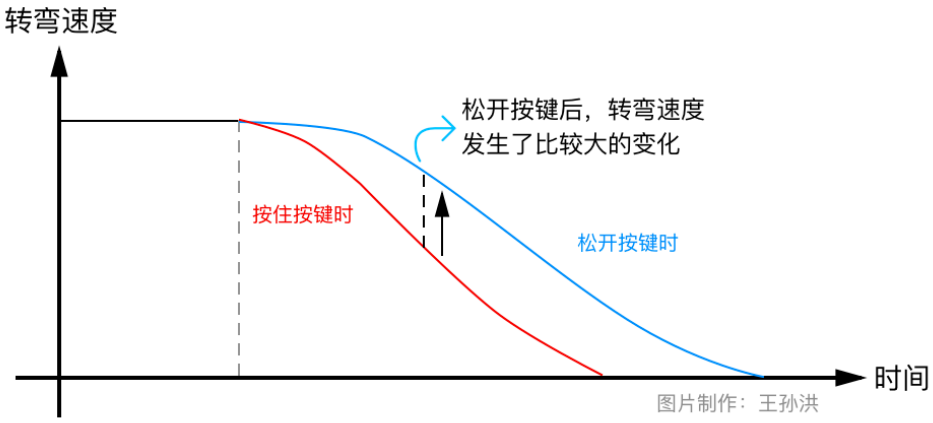
松开鼠标按键后，不管是否继续拖动鼠标，转弯速度的衰减都将变缓。

下图的曲线表示按住鼠标按键时转弯速度的变化情况：

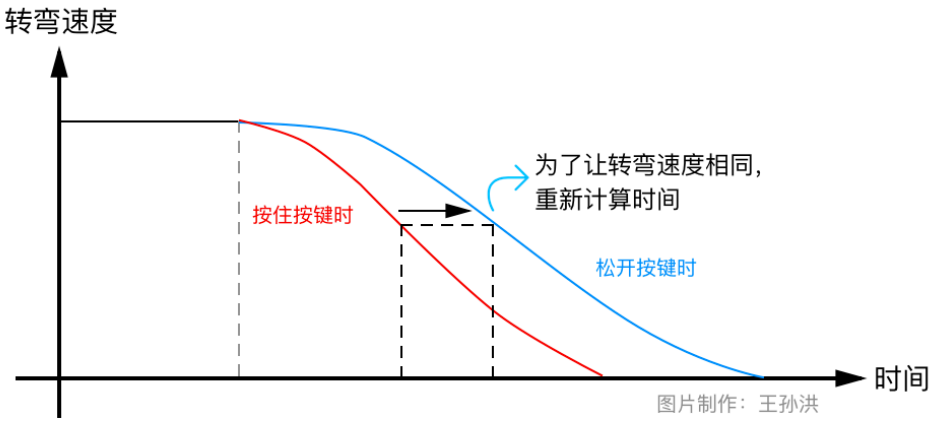


红色曲线表示按住鼠标按键时转弯速度的变化情况，蓝色曲线表示松开按键时的情况，可以看到，松开鼠标按键情况下，转弯速度衰减到 0 所花费的时间更长。

下图表示停止鼠标拖动后过一阵再松开鼠标按键时的转弯速度：



（1）只改变衰减率的情况



（2）重新计算时间的情况

在松开按键前，转弯速度沿着下面的曲线衰减，松开按键后则沿着上面的曲线衰减。在松开按键的瞬间，下面的曲线向上面的曲线演变，速度发生了较大的变化。

上面的曲线是松开按键后的转弯速度，下面的曲线是按住按键时的转弯速度。这两条曲线的最大值和最小值都相等，区别只在于速度衰减到 0 所需要的时间。

在上面的曲线和下面的曲线中，一定存在某个时刻二者的转弯速度是相同的。中间的那根曲线表示松开按键的时刻，它和下面的曲线的交点表示松开按键瞬间的转弯速度。在该处画一横线，在横线和上面的曲线的交汇处，二者的转弯速度相同。这两点的横轴值，也就是“从衰减开始经历的时间”是不同的。

控制转弯速度的主要代码如下：

PlayerController.FixedUpdate 方法

```
void FixedUpdate ()
  {
    // （a）转弯速度慢慢降低
    rot.Attenuate(Time.deltaTime);

    if (valid)
    {
      // 发射鱼雷
      if (Input.GetKeyDown(KeyCode.B))
      {
        //Debug.Log("B ender : " + Time.time);
        torpedo.Generate();
      }

      // （b）拖动中，使用鼠标在 X 轴方向的移动量来更新转弯速度
      if (Input.GetMouseButton(0))
      {
// Debug.Log("MouseButton");
        // 旋转
        rot.Change(Input.GetAxis("Mouse X"));
```

```
            // 加速
            speed.Change(Input.GetAxis("Mouse Y"));
        }

        // （c）在按下左键的瞬间设定转弯速度的衰减系数
        if (Input.GetMouseButtonDown(0))
        {
//Debug.Log("MouseDown");
            rot.BrakeAttenuation();
        }
        // （d）在松开按键的瞬间设定转弯速度的衰减系数
        if (Input.GetMouseButtonUp(0))
        {
            rot.UsualAttenuation();
        }
    }
    // （e）转弯处理
    Rotate();
    // 前进
    MoveForward();
}
```

RotationValue 类

```
public class RotationValue
  {
    public Vector3 current = Vector3.zero;
    private float attenuationStart;
    private float attenuationTime = 0.0f;
    // 当前的衰减率（attenuationRot/slowdownRot）
    private float currentRot;

    [SerializeField]
    private float max = 30.0f;
    [SerializeField]
    private float blend = 0.8f;
    [SerializeField]
    private float margin = 0.01f;
```

```
    [SerializeField]
    // “未按下按键”时的衰减率
    private float attenuationRot = 0.2f;
    [SerializeField]
    // “按下按键，未移动鼠标”时的衰减率
    private float slowdownRot = 0.4f;

    public void Init()
    {
      currentRot = attenuationTime;
    }

    /// <summary>
    /// 改变旋转量
    /// </summary>
    // （b）使用鼠标在 X 轴方向的移动量来更新转弯速度
    public void Change(float value)
    {
      // 当鼠标轻微移动时不更新
      //（为了确保不持续移动鼠标将会停止旋转）
      //（b1）如果鼠标移动量 value 的绝对值小于 margin 则退出（目的是为了让鼠标停止移动后最终能够停止转弯）
      if (-margin < value && value < margin) return;

      // 混合旋转量
      //（b2）更新转弯速度
      current.y = Mathf.Lerp(current.y, current.y + value, blend);
      if (current.y > max) current.y = max;
      // 重置衰减
      attenuationStart = current.y;
      //（b3）重置用于衰减的计时器
      attenuationTime = 0.0f;
    }
    /// <summary>
    /// 衰减
    /// </summary>
```

```
    /// <param name="time">时间变量</param>
    /// <returns>衰减中 / 不衰减</returns>
    // （a）转弯速度逐渐变慢
    public bool Attenuate(float time)
    {
      if (current.y == 0.0f) return false;
      attenuationTime += time;
      // （a1）用比率 “currentRot*attenuationTime” 对 attenuationStart 和 0.0f 进行补间
      current.y = Mathf.SmoothStep(attenuationStart, 0.0f, currentRot * attenuationTime);
      return true;
    }
    // （c）设定转弯速度的衰减率（按下按键的瞬间）
    public void BrakeAttenuation()
    {
      currentRot = slowdownRot;
    }
    // （d）设定转弯速度的衰减率（松开按键的瞬间）
    public void UsualAttenuation()
    {
      // （d1）为了不让转弯速度变化，重新计算从衰减开始经过的时间
      attenuationTime = (slowdownRot * attenuationTime) / attenuationRot;
      currentRot = attenuationRot;
    }

    public void Stop()
    {
      current = Vector3.zero;
    }
};
```

程序设计思路：

（a） 转弯速度逐渐变慢，进行衰减处理。

（a1）通过 Mathf.SmoothStep 方法求出当前时间 attenuationTime 对应的转弯速度。如果按

Mathf.SmoothStep（from，to，t） 的形式调用 Mathf.SmoothStep 方法，t 小于 0 时将返回 from，大于 1 时将返回 to，位于 0 到 1 之间时则返回从 from 到 to 之间的平滑插值。

（b） 使用鼠标在 X 方向的移动量来更新转弯速度。

（b1）如果鼠标的移动量 value 的绝对值小于 margin，则退出更新转弯速度的处理。这是为了让鼠标停止移动后最终能够停止转弯。

（b2）将鼠标的移动量乘以一定值后加上转弯速度，求出新的转弯速度。

（b3）将衰减用的计时器 attenuationTime 重新设置为 0 这个值表示转弯速度的衰减图形中的横轴“时间”。在鼠标移动时这个值一直保持为 0，因此（a1）的 Mathf.SmoothStep 的返回值总是等于 attenuationStart，不会做衰减处理。

（c） 将（a1）的计算中用到的衰减率的值改为松开按键时的值 slowdownRot。

（d） 和（c）相同，将衰减率的值改为松开按键时的值 slowdownRot。

（d1）为了使松开按键的瞬间转弯速度不发生剧烈变化，需要重新计算从衰减开始到现在经过的时间。转弯速度的计算公式：

Mathf.SmoothStep(attenuationStart，0.0f，currentrot*attenuationTime)；

为了使衰减率 currentRot 变化时上述计算式也能保持结果值不变，就需要确保 currentRot*attenuationTime 的值保持不变。这样，当 currentRot 从 slowdownRot 变为 attenuationRot 时，假设衰减率变化后的经过时间为

attenuationTime0，那么就有

slowdownRot*attenuationTime=attentionRot*attenuationTime0

对上式加以变形，即可得出

attenuationTime0=slowdownRot*attenuationTime/attenuationRot。

4. 声呐制作

在游戏中，画面的左上角会显示声呐探测到的敌机和鱼雷。虽然这对于游戏来说只是增加了一个小功能，但制作过程会用到很多重要的技术：

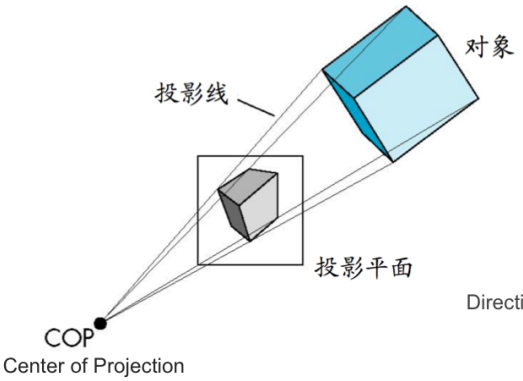
① 摄像机的视图变换

② 根据 layer 指定绘制对象

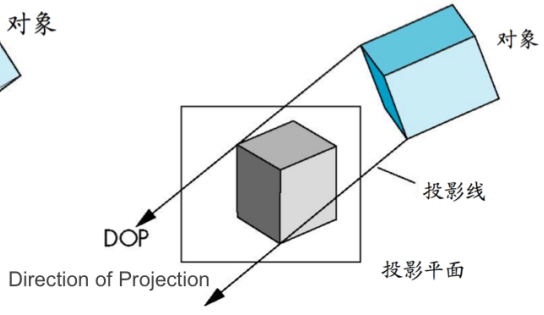
③ 通过 viewport 指定绘制位置

游戏中的摄像机具有**视图变换**的功能能够将三维坐标转换成二维坐标。玩家角色和地形等模型数据都位于三维空间内，而用于绘制它们的屏幕使用的则是二维坐标系。

摄像机的视图变换有两种类型，Perspective（透明视图）和Ortho（平行投影视图）。



(1) 透视投影示意图



(2) 平行投影示意图

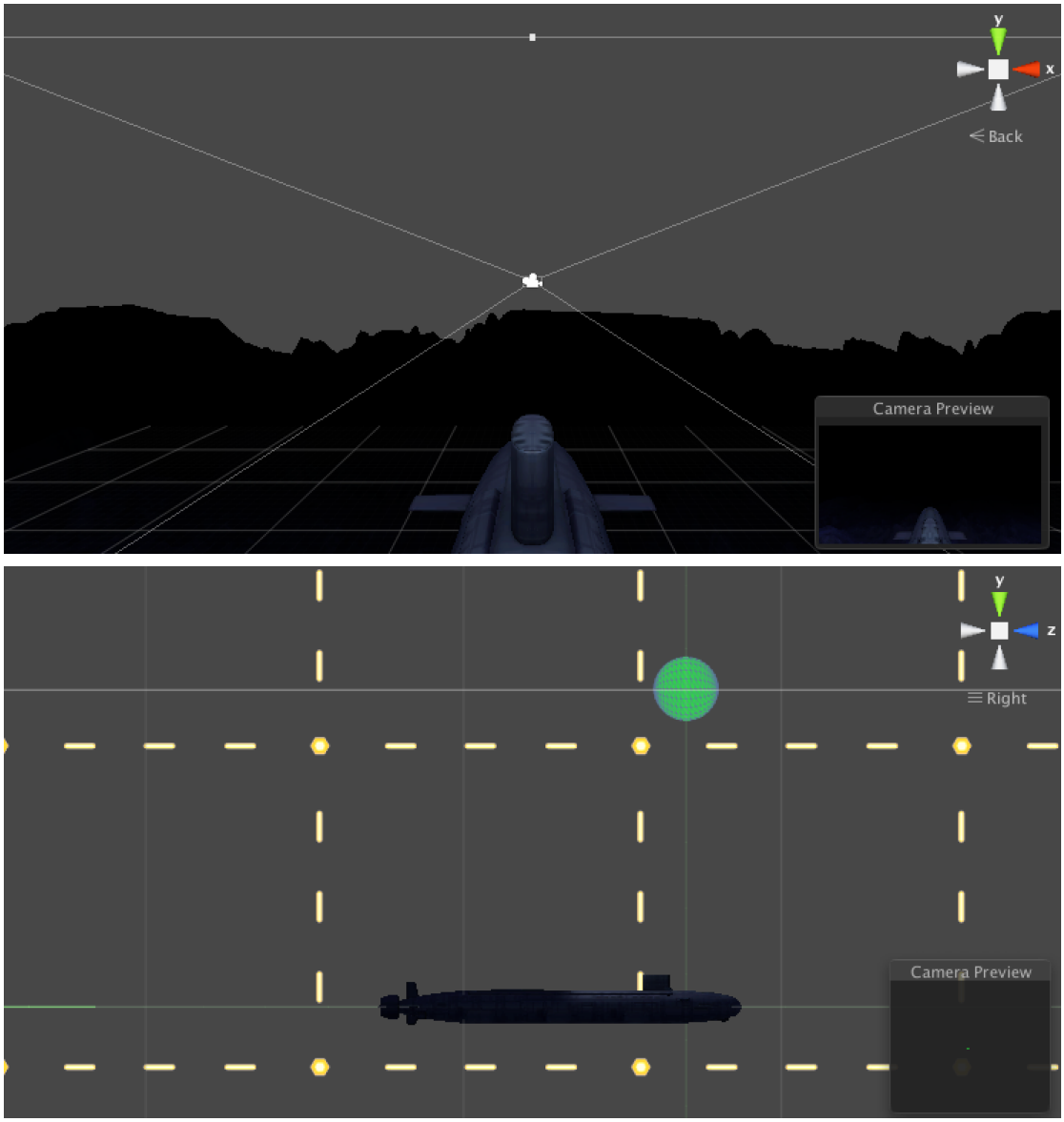
在透视投影视图中，物体离得越远显示在画面上的尺寸越小。透视视图的特性非常接近人眼。远处的物体看起来比较小，符合日常的经验。相反，平行投影视图因为比较容易调整物体显示在画面

上的大小。

因此，在本项目中，摄像机位于玩家潜水艇的后上方。一台是用于绘制 3D 主场景画面的主摄像机，另一台是用于绘制声呐的声呐摄像机。绘制游戏的主画面的摄像机用到透视视图，声呐画面用平行投影视图。

摄像机在决定将图像绘制在画面的何处时，使用了 viewport（视口）属性，通过设置不同的 viewport 参数，使主摄像机在整个画面中绘制，声呐摄像机在画面左上角的一部分区域中绘制。

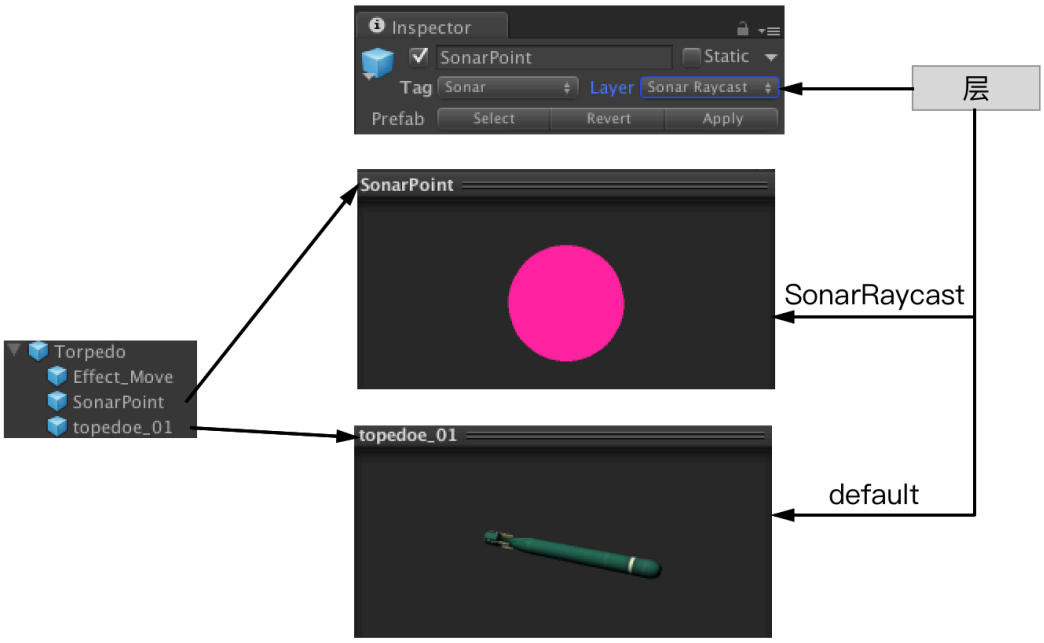
主摄像机位于玩家潜水艇的上部中央位置，总是朝着前进的方向。声呐摄像机位于玩家潜水艇的正上方，向下俯视。两个摄像机的位置和朝向是不同的。



主摄像机和声呐摄像机都是潜水艇的子对象，它们都随着玩家潜水艇一起移动。

游戏中主摄像机和声呐摄像机分别绘制不同的画面。**游戏中的物体在主摄像机上被绘制成 3D 模型，而在声呐上只显示一个点**。这中方法的实现使用了层（layer）的特性。

下图是鱼雷模型的层次关系：

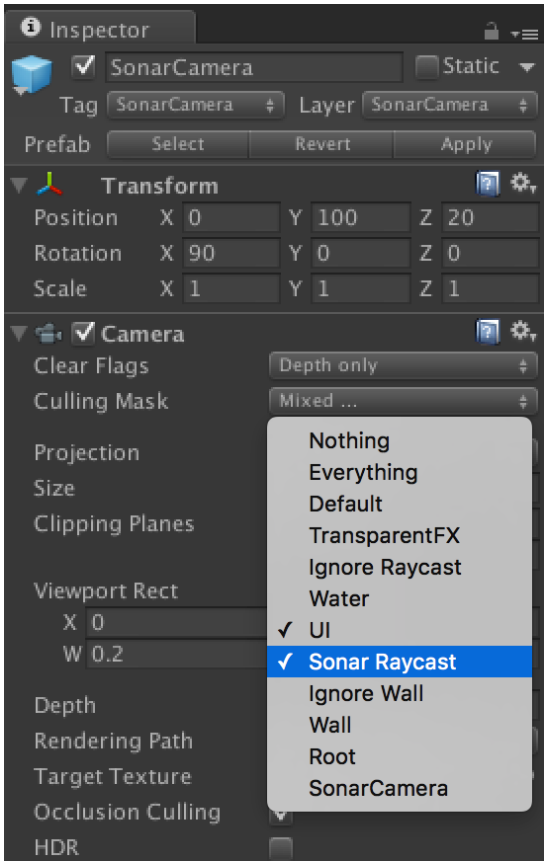


父对象 Torpedo 下有 topedoe_01 和 SonarPoint 两个子对象，其中 toepedoe_01 是鱼雷自身的 3D 模型，而 SonarPoint 则是单纯的球体模型，用于显示在声呐上。

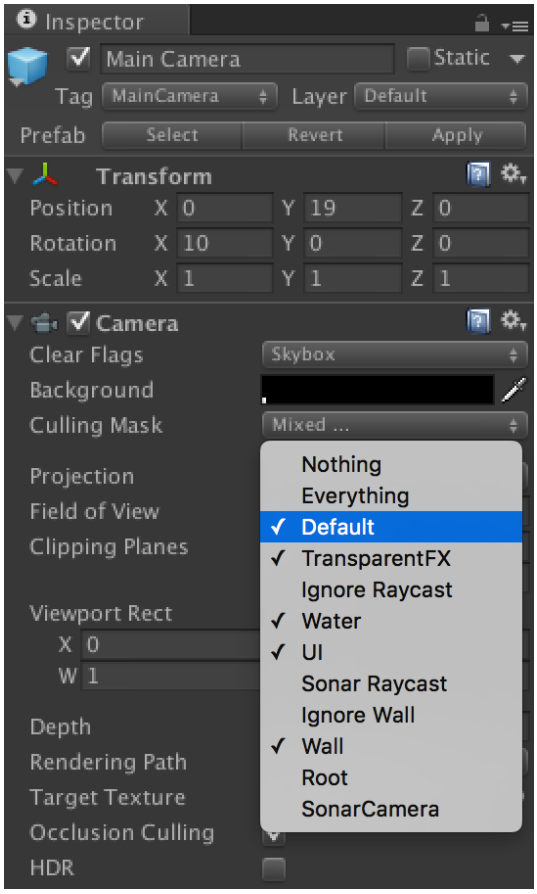
topedoe_01 和 SonarPoint 分别设定了不同的层，在检视面板中，SonarPoint 的层为 SonarRaycast，topedoe_01 的层为 Default。

再利用 **CullingMask （消隐遮罩）**这个用来选择性的渲染部分场景：

声呐摄像机中的裁剪蒙版只有 Sonar Raycast 项被选中

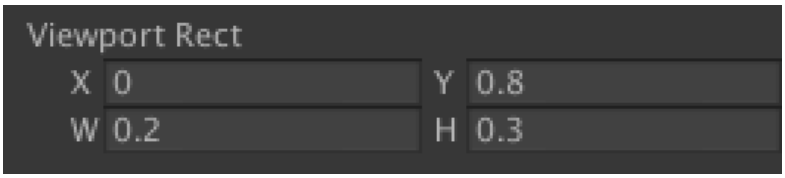


主摄像机中则选中了很多项



鱼雷模型 topedoe_01 和声呐上的小点 SonarPoint 总是同时存在，而且位置相同，通过对摄像机和对象进行层值的设定，就能在不同的摄像机中看到不同的画面。

最后，通过 **viewport rect（视口）**这个属性决定摄像机的图像在画面的何处显示，以及按多大的尺寸显示。声呐摄像机的 viewport rect 参数设置如下：



View Port 参数说明：

View Port Rect：标明这台相机视图将会在屏幕上绘制的屏幕坐标(值 0 - 1) 的 4 个值。视口坐标是标准的和相对于相机的。相机的左下角为（0，0）点，右上角为（1，1）点，Z 的位置是以相机的世界单位来衡量的。

X：相机视图的开始水平位置。

Y：相机视图的开始垂直位置。

W：摄像机输出在屏幕上的宽度。

H：摄像机输出在屏幕上的高度。

这样，声呐摄像机会显示在画面的左上角，实现“画中画”效果。

三、 项目小结

1. 技术总结

除了 Unity 3D 场景的天空盒、光源、地形引擎、角色动画、着色器等基本操作外，本项目的开发另外运用到了以下理论及技术：

① Unity 中 3D 声音应用。

② 利用插值与平滑的算法处理声音、文字、场景的淡入淡出及潜水艇的速度。

③ 在模拟潜水艇运动及相关操作时运用到游戏开发中的物理学，达到真实体验。

④ 根据投影变换绘制两个不同的摄像机显示在同一个画面中，并用 Layer（层）和 Culling Mask （剔除遮罩）的高级特性控制外观，选择性的照射物体，并通过视口绘制画中画。

2. 发展规划

在理论及技术支持下，本项目未来规划如下：

① 添加更为丰富的海底声音，有营造更逼真的海洋环境

② 添加海洋元素，如穿梭其中的危险生物以及奇珍异宝，打造更神秘莫测的海底世界，吸引玩家潜入探险

③ 引入虚拟现实技术，让玩家身临其境，享受视听盛宴

④ 移植到 Mobile 端，并通过网络进行多人协作，寻找财富，躲避危险物，合作共赢！

3. 参考资料

本项目的理论研究和技术实现参考了以下资料：

网络资源：

- Unity 3D sound
  http://wiki.etc.cmu.edu/unity3d/index.php/3D_Sound
- Unity Audio Source
  http://docs.unity3d.com/Manual/class-AudioSource.html
- Unity3D 声音格式和导入
  http://www.cnblogs.com/fortomorrow/archive/2012/10/31/unity06.html
- Unity GAME CONTROLLER

http://unity3d.com/cn/learn/tutorials/projects/stealth-tutorial-4x-only/game-controller

- Unity 简明教程——插值与平滑
  https://li-kang.gitbooks.io/unity-concise-course/content/Summary/Lerp.html

图书资源：

- 《游戏中的数学与物理学（第 2 版）》作者：（美国）John Patrick Flynt （美国）Danny Kodicek，清华大学出版社
- 《3D 数学基础:图形与游戏开发》作者：（美国）邓恩 （美国）帕贝利 译者：史银雪 陈洪 王荣静，清华大学出版社