# DOG SOULS 狗狗之魂

《游戏项目实践》课程期末项目报告

姓名：俞辰杰　　学号：10192100571　　专业：计算机科学与技术

## 1. 项目简介

《DOG SOULS》是我组自主研发的一款 Unity 的 3D 魂 like 游戏。游戏发生在一个被称作「麟齾堡垒」的地下城堡，在这里，你将扮演一位双手持剑的神秘角色，在地堡的旅行中邂逅旅行中的导师和同伴们，在他们的助力下一起击败强敌，离开恐怖的地堡——同时，逐步发掘「麟齾」的真相。

## 2. 游戏创意

在《DOG SOULS》立项之初，准备做一款类似于《黑暗之魂》的动作冒险游戏，在游戏中玩家将作为一个囚禁的外来者一路解开这个堡垒的秘密。在这个基础上对于整个游戏框架进行了设计：1）玩家和敌人的差距不能过大，不能随便就能打败所有敌人，有一定的探索成本；2）需要存在死亡惩罚，让玩家的失败惩罚高过一般游戏，要做慎重决定；3）在游戏中包含一定量的探索要素，让玩家充分探索这个地图；4）需要锁定敌人的机制，为了防止视角等原因导致攻击没有命中；5）需要对于玩家有一定的补偿机制，让玩家不会过长时间的卡在某一结点上....

同时还有部分创意由于时间缘故或者技术受限没有完成，比如敌人的随机物品掉落，角色的自定义装备和武器选择，打斗时的场景破坏来表现动作，这部分可以等到未来进行添加。

## 3. 游戏各模块介绍

### 1. 交互模块：


与门的交互


与宝箱的交互

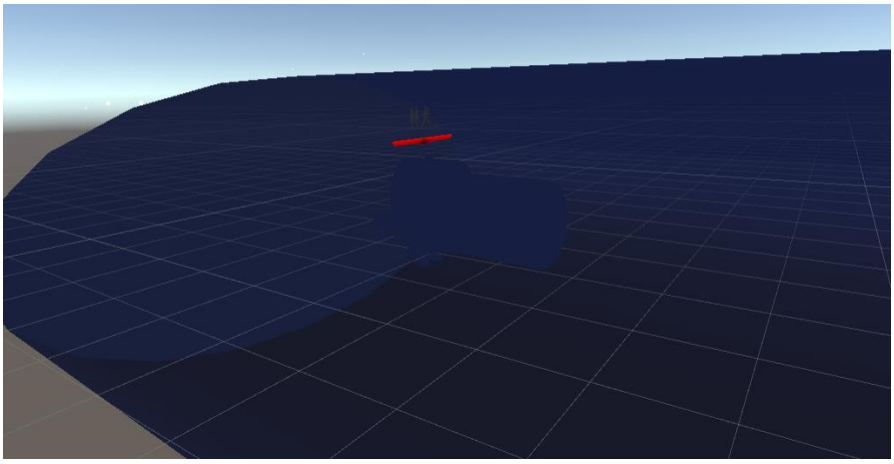
交互模块指玩家在游戏主场景内部和可交互部件进行互动，完成部分操作，包括可以“打开获得道具的宝箱交互”和“可开启与不可开启的门交互”，进入交互范围时会显示可交互的 UI，按下对应按钮即可完成交互。

### 2. 同伴模块：


在 NPC 同伴处进行升级

同伴模块指在游戏场地中和玩家互动，用于强化玩家的 NPC，在 NPC 可交互范围内交互时可以和玩家实现对话，以及使用玩家的消耗道具用于强化升级。

**3. 敌人模块：**



敌人的检测范围，外侧为警戒范围，内侧为攻击范围

敌人模块包括区别一般敌人以及 Boss。两者的共同之处包括敌人的追踪玩家的模块，攻击玩家的模块，以及受击和死亡的模块。前两个模块分别包含了警戒区，和攻击区用于检测。受击是基于敌人的本身碰撞箱完成交互，敌人死亡则会消除敌人的碰撞箱。

在进入对于检测区域后，对人会根据不同区域的脚本设计完成不同的任务。同时为了让敌人的攻击更加自然，攻击采用了随机数使得敌人的攻击更难被玩家预测和躲避。

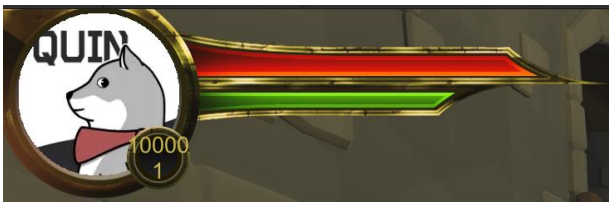
**4. 音乐模块：**

音乐模块包括了在三个场景中会循环播放的 BGM，以及在不同操作时候会播放的音效，包括玩家攻击受伤等玩家音效，以及开门、走路等环境音效。

采用多个音乐模块同时开启而非复用的方式，用以规避多音效情况下发生的音效的冲突。

**5. 玩家模块：**

**1）玩家 UI**




玩家的状态栏


回复道具

包括玩家的等级、血量和耐力的显示（使用 Slider 条用于展示剩余数值），回复道具的数量，以及收集的可消耗物品的数量。当玩家死亡或者是完成特定任务的时候还会在屏幕中央跳出对应的成就条。设置了黑幕转场，用于平滑的过渡场景切换。

**2）玩家操作**

攻击操作

翻滚操作

玩家的操作通过响应式操作进行。也即当玩家企图进行某种操作的时候，先根据优先级对此操作判断能否进行切入（例如翻滚的时候不能进行移动，移动的时候不能进行攻击），再进入到某个动作的处理模块，按逻辑判断是否有额外功能（攻击激活触发器，翻滚隐藏用户碰撞箱），同时播放对应动画。

**3）索敌模块**


锁定状态下角色攻击必定面向被锁定的敌人

在玩家前侧设置锁定范围以及维护一个可锁定目标列表，敌人进入或者离开锁定范围将会修改锁定目标列表，当玩家进行锁定的时候会从锁定列表中选择一个距离最近的敌人进行锁定，若当前目标因为死亡或离开范围导致丢失则会自动更换为下一个最近的目标锁定。

**4）等级模块**

玩家使用消耗道具进行升级的时候可以提升玩家自身的属性，分别用于提升玩家的血量耐力或伤害，升级后这些属性会被保存下来，不会随着玩家的退出或者游戏内死亡而消失。

**5）新手提示模块**

在玩家第一次进入游戏的时候，根据玩家有无存档记录判断其是否需要新手教程单元，之后便不会再出现新手教程。同时在完成新手教程之前，做了确保无法离开新手区域的设置。

**6. 功能模块：**

功能模块包括用于保存玩家游玩记录数据，用于在下次游玩时进行加载。此外还包括一个总的 GameManager，此模块在场景切换时候不会被销毁，这个模块用于进行场景切换同时记录玩家状态。

## 4. 核心编程模块

由于各种模块功能较多且复杂，此处仅简单介绍玩家操作的角色所涉及到的功能代码。

### 1. 玩家私有函数（CharacterStateBeta.cs）

```
void Update(){
    if(!isDead){
        Drink();
        Breathe();
        Exit();
    }
    else{
        ProcessDeath();
    }
}
```

IsDead 用于标记是否玩家死亡，非死亡状态下的 Drink，Breathe，Exit 函数分别代表着“玩家体力恢复”，“玩家血量回复”和“玩家离开游戏”三个事件的监听；死亡状态下只运行 ProcessDeath 用于处理死亡事件。

### 1.1 Drink 血量回复

```
    private int yspNum = 5;
    void Drink(){
        if(Input.GetKeyDown(KeyCode.K)){
            if(yspNum > 0){
                yspNum -= 1;
                canvas.transform.Find("RecoverHP/Text").gameObject.GetCo
mponent<Text>().text = "原素瓶 " + yspNum + "/5";
                recoverHP(this.gameObject.GetComponent<LevelAndMoney>().
getLevel() * 3f + 60);
                AM.transform.Find("SoundEffect").gameObject.GetComponent
<AudioSoundEffect>().AudioPlayDrink();
            }
        }
    }
```

Drink 函数中监听用户是否执行恢复（GetKeyDown），同时判断是否可以恢复（yspNum）。此函数会修改 UI 界面中显示剩余可用数量（Text），根据用户等级动态调整恢复数量（recoverHP），同时播放回血音效（AudioPlay）。

```
    void recoverHP(float cost){
        playerHP = (playerHP + cost > playerHPMax) ? playerHPMax :
playerHP + cost;
        healthBar.GetComponent<Image>().fillAmount = playerHP /
playerHPMax;
    }
```

Drink 函数中使用的 recoverHP 在增加血量的同时，可以修改 UI 界面中玩家状态栏的血量百分比条。

### 1.2 Breathe 体力恢复

```
    private float dt_forRecovery = 0f;
    private float recoveryEnduranceSpeed = 5f;
    void Breathe(){
        dt_forRecovery += Time.deltaTime * 0.05f;
        recoverEndurance(dt_forRecovery * recoveryEnduranceSpeed);
    }
```

Breathe 函数中没有玩家的交互操作，而是持续不断的为玩家恢复体力条，

其中 dt_forRecovery 用于线性增加玩家恢复体力的时间。而当玩家进行某些操作的时候就会重置 dt_forRecover 为零（见 2.1）。

```
    void recoverEndurance(float cost){
        playerEndurance = (playerEndurance + cost > playerEnduranceMax) ? playerEnduranceMax : playerEndurance + cost;
        manaBar.GetComponent<Image>().fillAmount = playerEndurance / playerEnduranceMax;
    }
```

recoverEndurance 和 recoverHP 的作用类似。

### 1.3 Exit 离开游戏

```
    private bool inPage = false;
    void Exit(){
        if(Input.GetKeyDown(KeyCode.Backspace)){
            inPage = !inPage;
            UI.transform.Find("Exit").gameObject.GetComponent<Canvas>().enabled = inPage;
        }

        if(inPage && Input.GetKeyDown(KeyCode.Return)){
            UI.transform.Find("SceneMaskCanvas/SceneMask").gameObject.GetComponent<SceneFade>().SceneFadeOut();
            AM.transform.Find("SoundEffect").gameObject.GetComponent<AudioSoundEffect>().AudioPlayButtonClick();
            this.gameObject.GetComponent<LevelAndMoney>().saveData();
            Invoke("backToHead", 2f);
        }
    }
```

Exit 函数监听用户的退出事件（GetKeyCode），从而修改用于显示退出界面的 inPage 属性。选择退出后 UI 会黑屏淡出（FadeOut），播放退出音效（AudioSource），保存玩家数据（saveData）最后切换至主界面（backToHead）。

```
    void backToHead(){
        GM.GetComponent<GameManager>().SetSceneNum("HeadPage");
    }
```

BackToHead 调用 GameManager 进行切换场景。

### 1.4 ProcessDeath 死亡事件

```
private bool isPlayingMusic = false;
void ProcessDeath(){
    if(!isPlayingMusic){
        AM.GetComponent<AudioManager>().AudioPlayDeath();
        isPlayingMusic = true;
    }
    cam.GetComponent<CameraController>().enabled = false;
    canvas.GetComponent<Canvas>().enabled = false;
    transform.Find("PlayerCore").position = new Vector3(0f,1000f,0f);

    Vector3 pos = transform.position;
    gameObject.GetComponent<LevelAndMoney>().saveDie(pos.x,pos.y,pos.z);

    GameObject background = gameOver.transform.GetChild(0).gameObject;
    background.GetComponent<DeathUI>().playDeathAnimation();
    GameObject textHighlight = gameOver.transform.GetChild(1).gameObject
    textHighlight.GetComponent<DeathUI>().playDeathAnimation();
    GameObject text = gameOver.transform.GetChild(2).gameObject;
    text.GetComponent<DeathUI_Text>().playDeathAnimation();

    Invoke("MaskFade", 3.5f);
    Invoke("GameRestart", 8f);
}
```

ProcessDeath 函数主要处理一系列玩家死亡后的事件：

1.播放死亡音效（AudioPlay），同时为了防止音效重复播放，所以设置一个标志 isPlayingMusic 表示不需要重复。

2.解除相机的绑定，使得无法在死亡之后再操作相机位置（CameraControll）

3.隐藏玩家的状态栏等 UI（Canvas），移除玩家的碰撞箱（Position）

4.保存玩家的等级，同时在死亡的地方记录生成遗留物的位置（saveDie）

5.播放死亡动画（playDeathAnimation， MaskFade）

6.重新开始游戏（GameRestart）

```
    void GameRestart(){
        GM.GetComponent<GameManager>().SetSceneNum("MainPage");
    }
```

### 2.玩家公有函数（CharacterStateBeta.cs）

### 2.1 Attack 攻击（CharacterController.cs）

```
public void Attack(int attackNumber, Side attackSide, Weapon
leftWeapon, Weapon rightWeapon, float duration)
{
    if(gameObject.GetComponent<CharacterStateBeta>().TryAttack()){
        animator.SetSide(attackSide);
        _isAttacking = true;
        Lock(true, true, true, 0, duration);
        var attackTriggerType = AnimatorTrigger.AttackTrigger;
        animator.SetActionTrigger(attackTriggerType, attackNumber);
        gameObject.GetComponent<CharacterStateBeta>().DoAttack();
    }
}
```

Attack 的调用入口在判断是否可以攻击（TryAttack）后加载攻击动画，同时进行攻击操作（DoAttack）。

```
    private float attackEnduranceCost = 30f;
    public bool TryAttack(){
        return hasEnoughEndurance(attackEnduranceCost);
    }
    bool hasEnoughEndurance(float cost){
        return playerEndurance > cost;
    }
```

TryAttack 函数的功能是比较当前体力值剩余和攻击所需体力值剩余（hasEnoughEndurance）。

```
    public void DoAttack(){
        weapon.GetComponent<BoxCollider>().enabled = true;
        decreaseEndurance(attackEnduranceCost);
        Invoke("ForceStopAttack", 2f);
    }
    void decreaseEndurance(float cost){
        dt_forRecovery = 0f;
        playerEndurance = (playerEndurance - cost < 0.1f) ? 0 :
(playerEndurance - cost);
        manaBar.GetComponent<Image>().fillAmount = playerEndurance /
playerEnduranceMax;
    }
```

```
    void ForceStopAttack(){
        weapon.GetComponent<BoxCollider>().enabled = false;
    }
```

DoAttack 函数的功能是激活武器上的碰撞箱（BoxCollider）用于造成伤害，消耗体力（decreaseEndurance），同时为了防止空挥的情况，在一定时间后会强制关闭碰撞箱（ForceStopAttack）。

可以看到在 decreaseEndurance 中会清空 dt_forRecovery 的记录。

### 2.2 DiveRoll 翻滚（CharacterController.cs）

```
public void DiveRoll(DiveRollType rollType){
    if(gameObject.GetComponent<CharacterStateBeta>().TryRoll()){
        if(GetComponent<RPGCharacterInputController>().HasAimInput()){
         GetComponent<RPGCharacterInputController>().SetAimInput(false);
            var angles = transform.rotation.eulerAngles;
            if(Input.GetKey(KeyCode.S)){
                angles.y += 180f;
            }
            else if(Input.GetKey(KeyCode.A)){
                angles.y += 270f;
            }
            else if(Input.GetKey(KeyCode.D)){
                angles.y += 90f;
            }
            transform.rotation = Quaternion.Euler(angles);
            Invoke("RecoverAim", 1.05f);
        }
        animator.TriggerDiveRoll(rollType);
        Lock(true, true, true, 0, 1f);
        SetIKPause(1.05f);
        gameObject.GetComponent<CharacterStateBeta>().DoRoll();
    }
}
```

和 Attack 类似，DiveRoll 函数首先尝试能否进行翻滚（TryRoll），然后判断当前是否处于锁定状态（HasAimInput）。若存在，需要先解除锁定（SetAimInput（false）），由于翻滚方向是基于“角色的前方”而非“摄像机的

前方”，所以需要在翻滚时候额外添加一个旋转量（rotation），之后重新锁定（RecoverAim）。

同时在翻滚操作的时间内，暂停所有其他企图切入的动作（SetIKPause），最后执行翻滚操作（DoRoll）。

```
    private float rollEnduranceCost = 20f;
    public bool TryRoll(){
        return hasEnoughEndurance(rollEnduranceCost);
    }
    public void DoRoll(){
        Core.localScale = new Vector3(0, 0, 0);
        decreaseEndurance(attackEnduranceCost);
        Invoke("recoverRolling", 1f);
    }
    private void recoverRolling(){
        Core.localScale = new Vector3(1f, 0.4f, 3f);
    }
```

Roll 函数和 Attack 函数功能类似，不同在于 DoRoll 的功能是将玩家的碰撞箱缩小（localScale）到忽略不计，以此规避伤害，同时又不至于消失使得敌人丢失锁定。

**3.玩家锁定敌人（TargetLock.cs）**

```
public bool LookAt = false;
private void Update() {
    if(Input.GetKeyDown(KeyCode.Tab)){
        LookAt = !LookAt;
    }
    if(LookAt){
        if(getCloest()){
            Player.GetComponent<RPGCharacterController>().SetTarget(clos
estTarget.transform);
            Player.GetComponent<RPGCharacterController>().SetAimInput(cl
osestTarget.transform.position);
        }
        else{
            Player.GetComponent<RPGCharacterController>().ClearTarget();
            LookAt = false;
        }
    }
}
```

LookAt 用来表示当前玩家是否处于锁定状态。当按下 Tab 键时候，玩家会进行锁定切换。

若此时为锁定状态，首先会判断是否待锁定列表中存在可锁定对象（getCloset)。如果存在，则会持续更新最近的敌人的位置并面向（SetTarget，SetAimInput)；如果不存在，那么将清除对象（ClearTarget）并结束 LookAt。

## 5. 运行环境配置及游戏测试

本游戏在 Window10 环境下的 PC 可以正常运行，同时还支持 Google Chrome 版本的 WebGL 版本，Unity 编辑器版本为 2020.2.7f1c1

游戏视频：https://www.bilibili.com/video/BV1yM411m7uT/

## 6. 课程小结

《DOG SOULS》的定位之初是一款动作冒险游戏，但是想要完成这样一类游戏，其最核心的难度在于动作的设计和流畅性转换，同时还要保证画风和游戏内容的一致性。

由于 Unity 提供的支持有限，一开始（CharacterStateOld.cs）使用了和游戏内敌人模型一致的玩家主角“dog”。在动作设计的时候，我们尝试过用 Animator 来衔接玩家的动作，但是过于复杂的动作衔接，例如：攻击接跑步，防御接重击等都需要在动作之间做切换，导致整个 Animator 过于复杂。

于是在第二版方案（CharacterStateNew.cs）中我替换为了 BlenderTree 来进行动作的衔接，效果较第一版有进步，但是缺少动画经验的我们在企图制作扩展动作的时候发现制作出来的动画的效果并不好。例如，翻滚，以及更多种类的攻击动作都缺少。

在最终呈现出来的方案（CharacterStateBeta.cs）中，修改为了像素风格的方块人，其最终呈现出来的效果比较能符合我的预期，所以将最终确定了第三版。

在角色动画之外，地图设计是另一大难点，即使现有资源给出了一系列优秀且符合预期的模型，但是将其拼搭成我们所需要的内容并且保证质量并不是一件容易的事情，相较于使用“DIY 地图”最终还是使用了资源自带的城堡地图，虽然不能展现出更深一步的设计所在，但是能够让整幅地图更加流畅。

以上这些在完整的游戏项目开发中应归属为美术部分的内容，包包括 UI 设计对于编写程序和设计者是一件费时费力可能也达不到满意效果的部分。

同时对于 Unity 来说，能够恰当使用管线渲染也是能为游戏的风格化和视觉效果增光添彩的部分，但是作为一个初学者很难在一个学期的时间之中获得所有

游戏开发相关的知识。

很兴奋在自己的努力下，能够完成自己的第一份游戏，虽然看上去问题重重，也有许多可以改进和需要改进的地方，但是更多的是一个游戏诞生于自己手中的快乐。在创作的同时了解到自己对于游戏开发的可以提升以及需要进一步了解的地方，希望能做出让更多人喜欢的，好玩的游戏。