浙江大学

题　　目：游戏设计技术文档

任课教师：王锐 霍宇驰

组　　长：李逸东

组　　员：祝舟航 王天昊 易桂光

作业完成时间：2023.6.25

目录

一、关卡设计 3

1. 主要脚本介绍 3
2. 功能实现 3

二、 动作与战斗系统 8

1. 主要脚本介绍 8
2. 人物按键输入控制 9
3.动画实现 9
4. 玩家状态管理 12
5. 武器性能与攻击 13

三、怪物系统 15

1. 脚本介绍： 15
2. 动画状态机介绍： 19
3. 怪物配置 19

四、游戏 UI 与数值设计 20

1. 游戏 UI 20
2. 游戏数值 23

技术文档

一、关卡设计

1. 主要脚本介绍

与关卡设计有关的脚本如下：

ShowHint.cs
用于显示在靠近机关时跳出的提示 UI。

DoorOpen.cs DoorNeedKey.cs ChestOpen.cs
用于实现无钥匙的门、需要钥匙的门的开启、箱子打开的功能

CrystalLit.cs CrystalData.cs
用于实现水晶点亮、水晶传送、记录上一次点亮的水晶的位置等功能

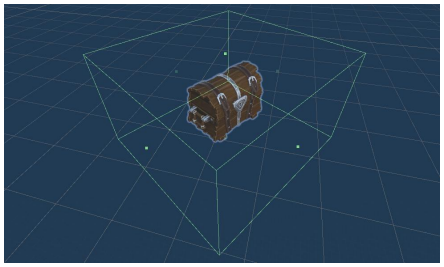
ElevatorControl.cs Press.cs AtUp.cs AtDown.cs
SwitchUpSide.cs SwitchDownSide.cs
用于实现电梯的整体移动

NPC.cs DialogData.cs
用于实现 NPC 的对话交互

2. 功能实现

(1) 靠近跳出提示

以宝箱为例



首先需要为宝箱设置 2 个 Collider，其中一个是宝箱自身的 Component，设置为 Trigger，范围较大，并且和脚本共同作用于触发提示 UI，另外添加一个 EmptyObject 带有 Collider，实现贴近宝箱时的碰撞。

脚本实现上，进入 Trigger 时，将一个 bool 值 isTriggered 设置为 True，在 Update 中检测 isTriggered，检测为 True 后，若按下 F 则播放开启动画，同时设置 UI 提示和获得物品。

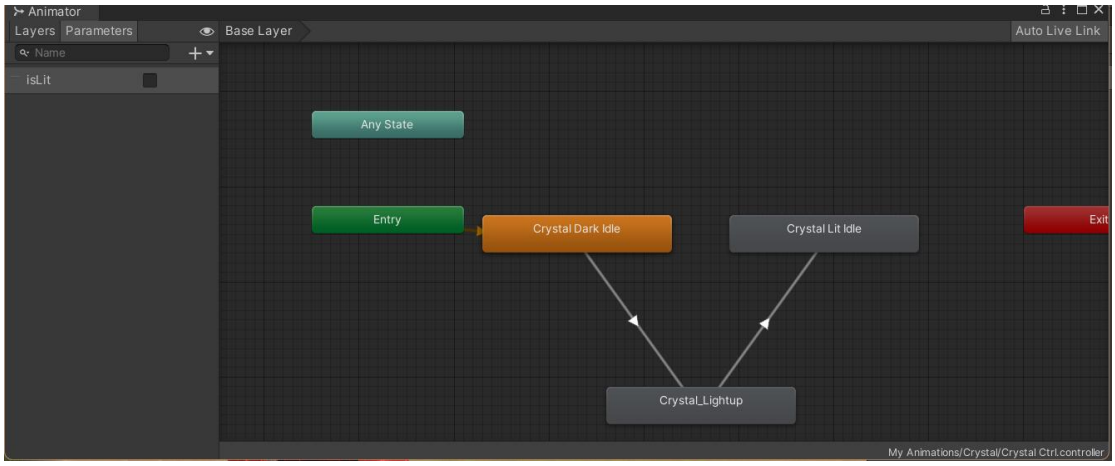
```
private void OnTriggerEnter(Collider other)
    {
        Debug.Log(other.gameObject.name);
        if (other.gameObject.tag == "Player")
```

```
        {
            Debug.Log("Entered!");
            isTriggered = true;
            other.gameObject.GetComponent<ShowHint>().Show();
        }
    }
private void Update()
    {
        if (isAfterOpen)
        {
            Destroy(gameObject,4);
        }
        if (isTriggered && Input.GetKeyDown(KeyCode.F))
           GetComponent<Animator>().SetTrigger("isOpen");
            if (Content != null)
            {
                Content.isAcquired = true;
             //ShowUI()
             }
        }
    }
```

对于需要钥匙的门，会在玩家的道具信息 PlayerInventory.cs 中添加一个 bool 变量，用来代表是否获得钥匙，在获得钥匙后，将此变量设置为 true，然后在开门时检测此变量即可。对于恢复药剂的处理和钥匙类似。

(2) 动画实现

使用 Animation 组件为机关录制简单的动画，然后使用 Animator 为动画设定触发器。以水晶为例给出示例图。



(3) 水晶管理

首先，建立了一个水晶管理器的 GameObject，其中挂载了 CrystalData.cs，包含两个列表，一个包含所有水晶，另一个用来记录被点亮的水晶。为每一个水晶设定一个 ID，然后

使用委托，每当有水晶被点亮，则调用委托，在列表中加上新点亮的水晶 ID。

委托的创建和声明如下所示：

```
public delegate void AddtoList(int ID);
   public class CrystalLit : MonoBehaviour
   {
       [SerializeField] public int ID;
       public AddtoList addtoList;
   }
```

在游戏开始时，在 CrystalManager 中将水晶加入列表并注册委托。每当委托触发时，则调用 OnCrystalLit()，将水晶 ID 添加到已点亮列表中。

```
private void Awake()
   {
       for(int i = 0; i < transform.childCount; i++)
       {
           CristalList.Add(transform.GetChild(i).gameObject);
           CristalList[i].GetComponent<CrystalLit>().addtoList += OnCrystalLit;//调
用委托
       }
    }
public void OnCrystalLit(int CrystalID)
   {
       isLitList.Add(CrystalID);
   }
```

(4) 水晶传送

水晶点亮后，靠近水晶点击 F 可以跳出 UI，然后点击水晶的按钮，即可传送。由于全部地图都放在同一个场景内，所以在传送时只需要修改角色的 position 即可。传送时添加了一个黑色遮罩，是通过在 UI 面板的最前方加入一张全黑色的图片，并将其设置成可视/不可视来完成的。

```
public void tpUI()
   {
           CrystalUI.transform.parent.GetChild(3).gameObject.SetActive(true);
           Invoke("Maskoff", 2f);
   }
```

由于角色的移动收到 StarterAssests 中的 Controller 控制，而 Controller 禁用了对 transform 的直接修改，所以直接修改角色的 transform 是不可以的。因此，在角色控制的脚本中添加了一个 teleport()函数，通过改变_controller 的位置来传送。

```
public void teleport(int ID) {
           switch (ID) {
               case 0:
                   _controller.transform.position = new Vector3(342.79f, 21.2f,
73.5f);
```

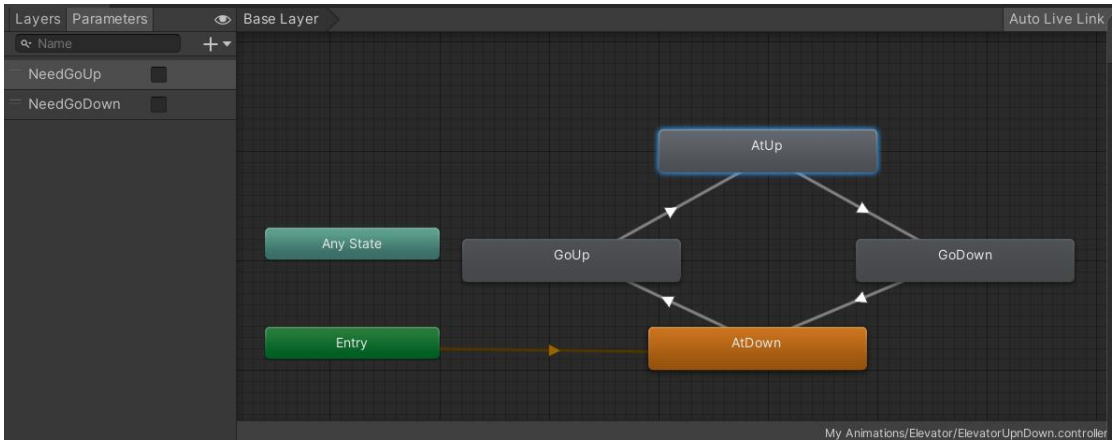
```
                break;
            ...  ...
```

(5) 电梯

电梯由三部分组成：踏板和上下两个拉杆。

踏板的逻辑为：按下一次后切换电梯状态，用两个 bool 变量 NeedGoUp 和 NeedGoDown 实现。踏板是否按下用 bool 变量 IsPressed 判断。踏板按下后，如果电梯在上方，就给出向下方运行的需求，然后电梯用动画实现下降。

```
void Update()
    {
        if(isUp||isDown)
            Press.GetComponent<Press>().ElevatorStable = true;
        else
            Press.GetComponent<Press>().ElevatorStable = false;

        if (isDown && Press.GetComponent<Press>().isPressed)
        {
            animator.SetTrigger("NeedGoUp");
            animator.ResetTrigger("NeedGoDown");
        }
        if (isUp && Press.GetComponent<Press>().isPressed)
        {
            animator.SetTrigger("NeedGoDown");
            animator.ResetTrigger("NeedGoUp");
        }
    }
```

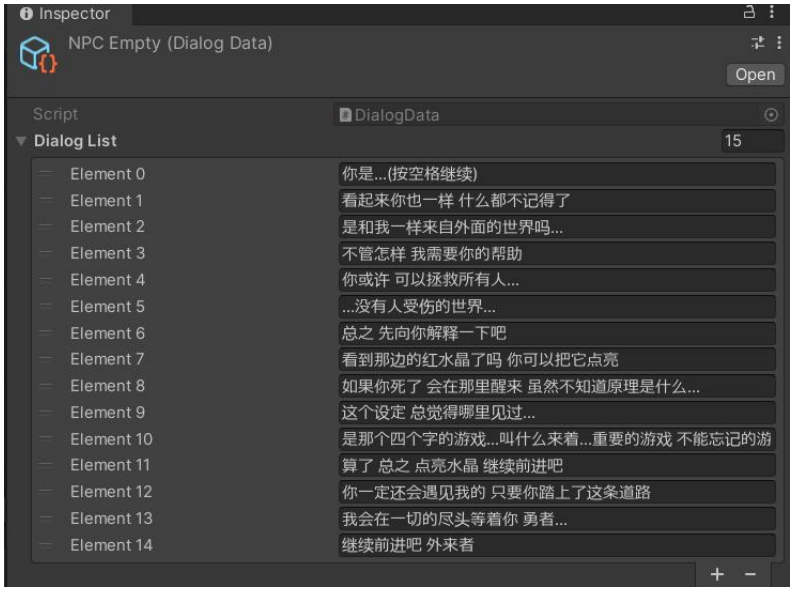


拉杆对电梯的控制是先检测电梯在上方还是下方。如果电梯在上方，那就使能下方拉杆。拉动拉杆后，令 IsPressed 变为 true，相当于又踩了一次踏板。此后的运行逻辑与踏板相同。

(6) NPC

游戏中设置了一个 NPC。其中靠近触发对话的功能与宝箱等相同，不再赘述。按 F 触发对话后，会跳出一个新的对话框，上面显示对话内容。按下 space 可以继续对话。

```
[CreateAssetMenu(fileName = "DailogData", menuName = "EnvironmentAssests/DailogData")]
public class DialogData : ScriptableObject
{
    public List<string> dialogList;
}
```



对话内容的实现使用了 Unity 的 ScriptableObject。其中包含一个字符串列表，内容即为按句子划分的对话内容。对话内容的填入可以直接在 Unity 内部写入，也可以用文本文件导入。

对话过程的实现上，由于游戏本身就是一个循环，如果每帧都重写字符串的话可能会造成性能下降，所以使用了一个 DialogPhase 的 int 变量，每次按下空格键，DialogPhase 自加，然后将其作为下标，从字符串列表中取出对应的文本并展现在文本框中。

结束对话后，DialogPhase 值最大，并直接取出最后一句话，作为结束语。

```
    public int talkPhase = 0;
    private void Update()
    {
        if (isTriggered)
        {
            if (Input.GetKeyDown(KeyCode.F))
            {
                GetComponent<Animator>().SetTrigger("EnterTalk");
                TextBox.SetActive(true);
                isTalking = true;
            }
            if (isTalking)
            {
                if (finishedTalking)
                {
                    talkPhase = 14;
```

```
                if (Input.GetKeyUp(KeyCode.Space))
                {
                    TextBox.SetActive(false);
                    isTalking = false;
                }
            }
            if (talkPhase<=13&&Input.GetKeyUp(KeyCode.Space))
            {
                talkPhase++;
                Invoke("DiaglogTalk",0f);
                if (talkPhase > 13)
                {
                    TextBox.SetActive(false);
                    isTalking = false;
                    finishedTalking = true;
                }
            }
        }
    }
}
```

二、 动作与战斗系统

1. 主要脚本介绍

与人物动作与战斗系统有关的脚本如下：

ThirdPersonController.cs

第三人称控制系统，绑定于玩家人物模型，用于处理输入、并控制角色根据输入做出相应的动作

PlayerBasics.cs

玩家状态管理系统，绑定于玩家人物模型，用于记录和处理玩家的基本信息，如血量、耐力值、经验值等

PlayerInventory.cs

玩家武器道具管理系统，绑定于玩家人物模型，用于记录和管理玩家可以使用的武器、道具

WeaponHolder.cs 和 WeaponHolderManager.cs

武器拿持系统，WeaponHolder.cs 绑定于玩家手部模型，主要用于将武器模型加载出来并绑定在人物手部模型手上，WeaponHolderManager 是 PlayerInventory 的一个子类，用于管理左右手的 WeaponHolder，同时负责了伤害的打开与关闭

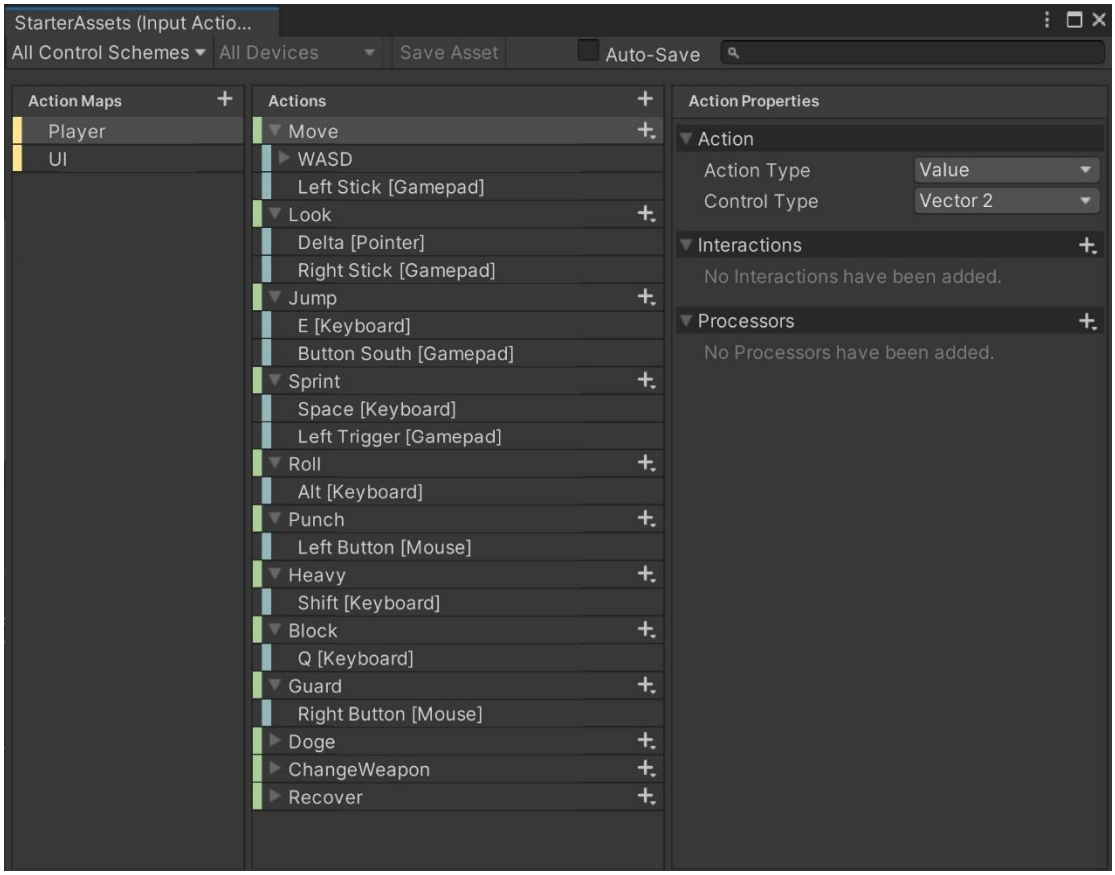
Weapons.cs

武器基本信息系统，每一种武器都对应着一个 Weapons 类的可编程对象，记录武器的基本信息，包括武器模型、种类、ID、基础伤害和基础耐力消耗等

Damages.cs
伤害处理系统，绑定于武器模型，用于处理武器接触到人物或者敌人模型时的伤害

2. 人物按键输入控制

在项目开发中，人物控制的按键输入主要通过 Unity Starter Asserts 中新的输入系统，部分键位绑定如下：

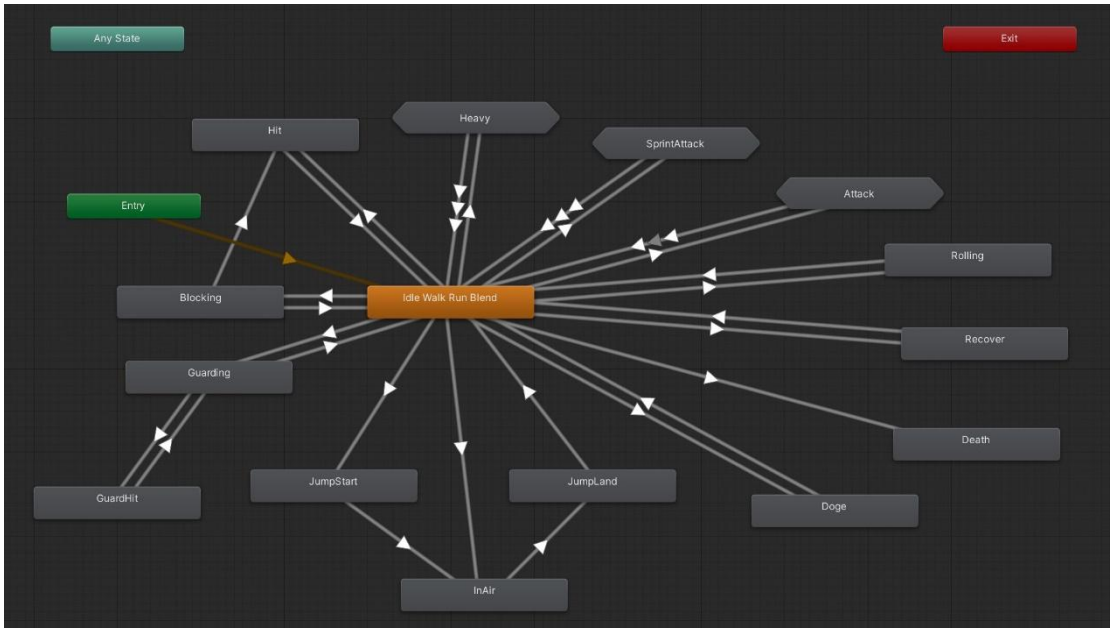


在第三人称人物控制脚本 ThirdPersonController.cs 中监听 input 的结果，并根据不同的监听结果进行不同的操作，并将动画状态机中相应的状态转换条件变量置为 true。

3.动画实现

(1) 人物动作动画触发

在 Unity 中，玩家当前执行的动画是通过状态机进行管理的，玩家人物动画状态机如下：

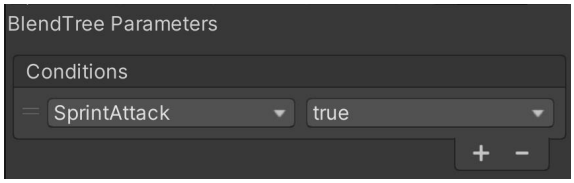


玩家默认处于 Idle Walk Run Blend 状态，该状态是原地静止、步行和奔跑状态的结合。之后根据状态机各个参数值的变化来决定状态转换。人物动画状态机的参数列表如下：

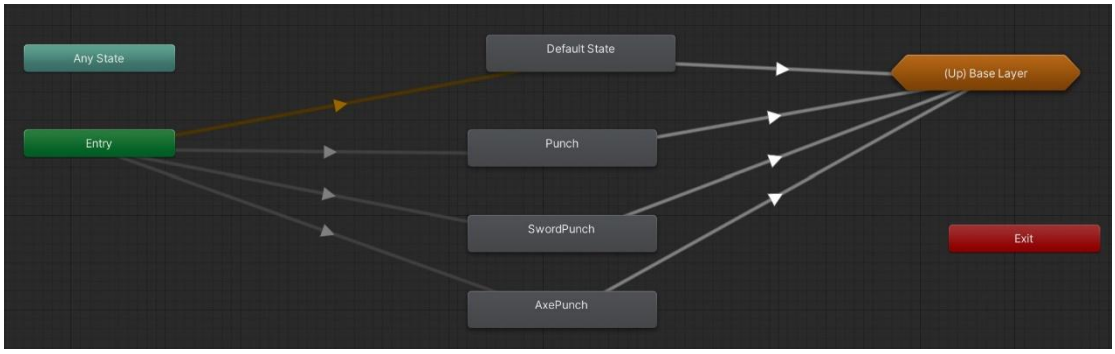
接下来以攻击动作为例详细解析状态机的状态转换。

(2) 攻击动作动画实现

当鼠标左键被按下，input 受到输入 Punch 后，根据此时是否监听到 shift 被按下(重击)以及是否监听到 space 被按下(奔跑)，决定是进入轻击、重击还是跑攻的子状态机，具体就是将相应的状态转移条件变量置 true，比如从 Idle Walk Run Blend 转移到 SprintAttack 子状态机的条件是。SprintAttack 为 true，而在 ThirdPersonController.cs 脚本中如果在按下鼠标左键时还按着空格就将该参数置为 true，因此状态机会进入 SprintAttack 子状态机。



对于普通攻击、跑攻和重攻击三种攻击方式，对于不同的武器类型需要运用不同的人物攻击动画，因此在子状态机中也应该分情况进行处理。这里运用了三种攻击模组，即重武器、轻武器、空手攻击，每一种状态绑定的是对应相应武器的轻攻击动画。同样在 ThirdPersonController.cs 脚本中有记录当前人物装备的武器类型，并在每次 Update 时检查更新状态机中的参数布尔值。以轻攻击为例，子状态机如下：



接下来以翻滚动作为例解析 ThirdPersonController.cs 脚本中对状态机参数根据输入的操作。

(3) 翻滚动作动画实现

在人物动画状态机中，最普遍出现的动画就是按下一个按键然后触发一段动画，比如翻滚、攻击、弹反、上步、回血等，对这些动画状态的条件参数值的管理本质上都是相同的，因此这里以翻滚动作为例进行介绍。

在触发了一次动画以后，必须要等待一段时间才能够再次触发，这里使用的是 Timeout 机制，每次新触发一次动画，将 TimeoutDelta 置为倒计时的值，将已触发标志置为 true，随着每次 Update 调用，TimeoutDelta 都相应减少，只有当它减少到 0 的时候，已触发标志才置为 false，相应的状态机参数和输入信号也置为 false，这个时候才能够触发下一次的动画。

翻滚动作是运用这种机制的动画中实现最为简单的之一，因为它只涉及一个参数即 Roll 的管理，实现具体代码如下：

```
private void RollForward()
        {
            // roll satisfied
            if (_input.roll && _rollTimeoutDelta <= 0.0f && rolled == false)
            {
                _animator.SetBool(_animIDRoll, true);
                _rollTimeoutDelta = RollTimeout;
                rolled = true;
            }
            else if(_rollTimeoutDelta <= 0.0f)
            {
```

```
                _animator.SetBool(_animIDRoll, false);
                _input.roll = false;
                rolled = false;
            }

            // roll timeout
            if (_rollTimeoutDelta >= 0.0f)
            {
                _rollTimeoutDelta -= Time.deltaTime;
            }
        }
```

4. 玩家状态管理

PlayerBasics.cs 脚本中记录着玩家的基本状态，包括血量、耐力值、经验值、是否收到伤害等信息以及对这些基本信息的管理。

```
public class PlayerBasics : MonoBehaviour
{
    public Animator _animator;
    public bool _hasAnimator;

    public BarSlide healthBar;
    public int currentHealth = 100;
    public int maxHealth = 100;
    public int currentStamin = 100;
    public int maxExperience = 100;
    public int currentExperience = 0;
    public bool isHit = false;
```

当玩家收到伤害时，调用 takeDamage() 方法，要注意当玩家处于翻滚状态时是应该不受到任何伤害的，而处于格挡状态时受到的伤害应该减半，我们通过判断玩家此时是否正在执行相应的动画来实现。

```
public void takeDamage(int damage)
        {
            //If the player is rolling or doging, ignore the damage
            if (_animator.GetCurrentAnimatorStateInfo(0).IsName("Rolling") ||
                 _animator.GetCurrentAnimatorStateInfo(0).IsName("Doge")) {
                return;
            }
            //If the player is guarding, the damage is smaller
            if (_animator.GetCurrentAnimatorStateInfo(0).IsName("Guarding"))
            {
                currentHealth -= damage/2;
            }
            else currentHealth -= damage;
            healthBar.SetCurrentHealth(currentHealth);
```

```
            isHit = true;
        }
```

在 ThirdPersonController.cs 脚本中，每次 update 时都应该检查 PlayBasics 中的 isHit 是否被置为 true，如果为 true 则应该将触发被击中动画的参数置为 true，从而使得玩家做出被击中的动画，实现玩家被击中的反馈。

5. 武器性能与攻击

(1)武器性能

Weapons 类继承自 ScriptableObject 类 Item，记录武器的基本数据，包括武器模型的 prefab 绑定、武器攻击类型（决定攻击动画模组）、武器具体类型（决定攻击能够造成的伤害和损失的耐力值）、武器是否已经获得以及武器能够造成的基础伤害值和玩家装备该武器时攻击会消耗的基础耐力值。每次在 Awake()时，应该根据武器的 ID 以及玩家的等级信息初始化其 baseDamage 和 baseStamin。Weapons 类成员变量如下：

```
public class Weapons : Item
{
    //game model prefab
    public GameObject modelPrefab;
    //unarmed signal
    public bool isUnarmed;

    // 1 for sword
    // 2 for axe
    public int kind;
    // 1 for sword
    // 2 for mace
    // 3 for helberd
    // 4 for bigsword
    // 5 for axe
    public int ID;

    //if the weapon is acquired
    public bool isAcquired;
    public int baseDamage;
    public int baseStamin;
```

(2)武器切换与获取

在 PlayerInventory.cs 脚本中记录了玩家获取武器和道具信息。该脚本类的成员变量如下：

```
public class PlayerInventory : MonoBehaviour
{
    public WeaponHolderManager weaponHolderManager;
    public Weapons right_weapon;
    public Weapons left_weapon;
    public Weapons unarmed_weapon;

    //The weapon slot in both hands
    public Weapons[] right_weapons_slot = new Weapons[1];
    public Weapons[] left_weapons_slot = new Weapons[1];

    //The index for weapon currently loaded
    public int cur_right_index = 0;
    public int cur_left_index = 0;
```

right_weapon 和 left_weapon 代表玩家左右手当前装备着的武器，right_weapons_slot 和 left_weapon_slot 是玩家的武器槽，存储着五种玩家能够使用的武器。脚本中还有一个 changeRightWeapon()方法用于切换右手武器（左手暂时只能够装备盾牌），当监听到 input

中有鼠标滚轮向下滚动时调用该方法，首先将 cur_right_index 加 1，如果下一个武器槽中的武器的 isAcquired 值为 true，即该武器已经获得，那么 right_weapon 就置为当前武器槽的武器，否则再次将 cur_right_index 加 1，一直循环直到找到 isAcquired 值为 true 的武器；同时如果 cur_right_index 大于武器槽的总长度，那么将 cur_right_index 置为-1，即空手不装备武器的状态，攻击方式也变为空手攻击。

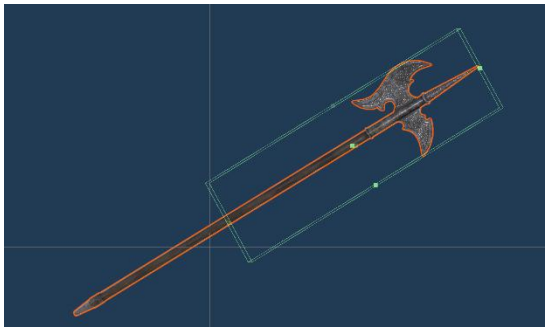
关于人物拿取武器的具体实现，主要是通过在人物手部绑定一个 WeaponHolder.cs 脚本：

```
public class WeaponHolder : MonoBehaviour
{
    public Transform parentOverride;
    public GameObject currentWeapon;
    public bool isright;
    public bool isleft;
```

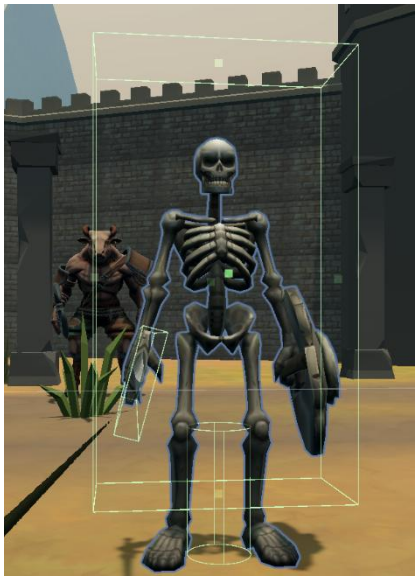
在脚本中载入并实例化当前装备的武器模型 prefab，并将其 transform.parent 设为手部模型，这样在载入武器模型后就能够随着手部的运动而运动，另外调整武器的大小和位置，从而实现手持武器的效果。

(3) 碰撞检测与伤害

不管是玩家攻击敌人还是敌人攻击玩家，是否造成了伤害都是通过在武器上绑定碰撞盒，检测是否与玩家或者敌人的碰撞体发生碰撞来实现的。例如长戟的碰撞盒如下：



怪物的碰撞盒如下：

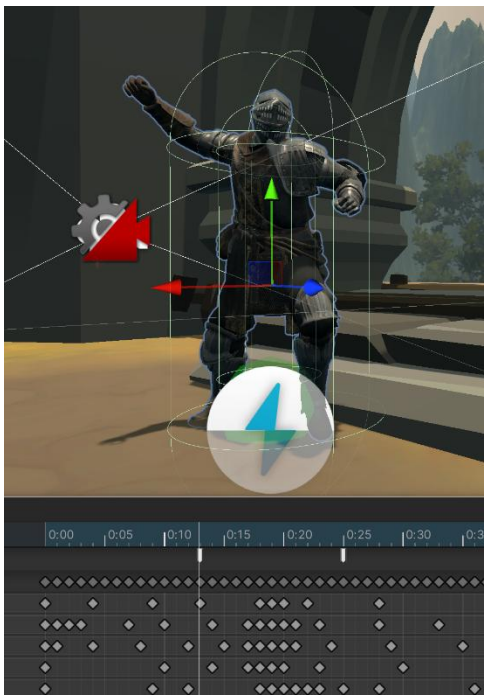


以玩家攻击怪物为例，我们在武器模型上绑定 Damages.cs 脚本进行玩家攻击伤害处理，当武器的碰撞盒碰撞到了敌人的碰撞盒，触发器方法 OnTriggerEnter(Collider collision) 就会

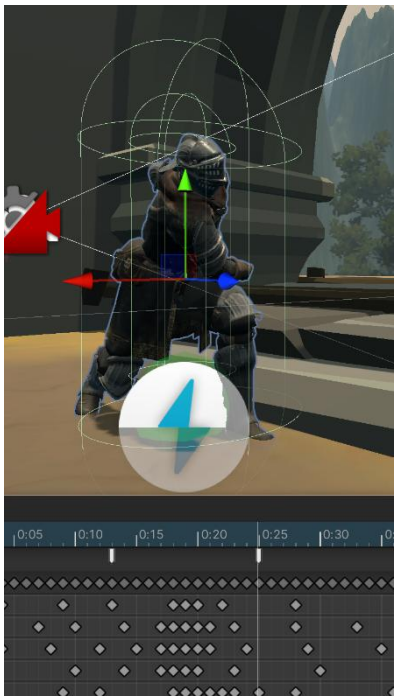
生效，collosion 就是和武器发生碰撞的怪物模型，因此我们就能够直接对怪物的血量数值进行操作，从而达到对敌人造成伤害的效果，比如当敌人的武器击中玩家模型后，我们应该调用玩家 PlayerBasics.cs 中的 takeDamage() 方法，玩家对敌人造成伤害也亦然。

另外，玩家对敌人造成的伤害还应该根据玩家装备的武器以及攻击动作方式的不同（轻击重击跑攻）来设定，项目中同样通过判断玩家是否处于相应的动画状态并赋予不同的伤害值来实现。

还值得一提的是，玩家对敌人或者敌人对玩家造成伤害应该是在武器挥砍的那一个动作才能够触发，假如不加以约束的话会出现敌人没有发动攻击动作而玩家只是碰到了敌人就出现受到伤害的情况，这肯定是不合理的。因此我们为每个攻击动画都设置两个 AnimationEvent，在开始挥砍时打开碰撞体，在结束挥砍时关闭碰撞体，这样只有玩家或敌人处于挥砍过程中才会造成伤害。



玩家开始挥砍



玩家结束挥砍

三、怪物系统

1. 脚本介绍：

EnemyController.cs

用于控制怪物状态的基类，确定怪物的状态。首先在脚本中创建枚举类 State，规定了怪物存在 Idle 空闲状态、Attack 攻击状态、Damage 受击状态、Dead 死亡状态。并在开始时规定怪物处于 Idle 状态同时搜索 player 绑定为侦测对象。

```
public enum State
{
    Idle,Attack,Damage,Dead,
}
private void Start()
{
```

```
    CurrentState = State.Idle;
    player = GameObject.FindWithTag("Player").transform;
}
```

同时定义了四个状态，以及四种状态转换方法，具体状态行为和转换的调用在子类具体怪物中实现。

```
public virtual void StateIdle() { }
public virtual void StateAttack() { }
public virtual void StateDamage() { }
public virtual void StateDead() { }
public virtual void ChangeIdle()
{
    CurrentState = State.Idle;
}
public virtual void ChangeAttack()
{
    CurrentState = State.Attack;
}
public virtual void ChangeDamage()
{
    CurrentState = State.Damage;
}
public virtual void ChangeDead()
{
    CurrentState = State.Dead;
}
```

在 Update 中，不断检测怪物和玩家的距离，同时检测当前状态，在不同状态执行不同的行为逻辑。

```
private void Update()
{
    distance = Vector3.Distance(player.transform.position, transform.position);

    switch (CurrentState)
    {
        case State.Idle:
            StateIdle();
            break;
        case State.Attack:
            StateAttack();
            break;
        case State.Damage:
            StateDamage();
            break;
```

```
        case State.Dead:
            StateDead();
            break;
        default:
            throw new ArgumentOutOfRangeException();
    }
}
```

Minotaur.cs 和 Skelton.cs

分别为牛头人和骷髅怪的脚本实现。首先绑定武器和伤害脚本，并且在动画中绑定开启伤害以及关闭伤害，防止在攻击之外产生碰撞造成意外伤害。

```
public GameObject Weapon;
public DamagesOpen damage;
public void OpenDamage()
{
    damage.EnableDamage();
}
public void CloseDamage()
{
    damage.DisableDamage();
}
```

空闲状态时允许怪物进行巡逻，通过绑定目标点，使怪物往返于各目标点之间，并且允许在目标点停留一定时间。

```
public GameObject[] Target;
private float time = 2.0f;
public override void StateIdle()
{
    • • •
    if (time <= 0)
    {
        time = 2.0f;
        var nexti = Random.Range(0, Target.Length);
        while (nexti == i)
        {
            nexti = Random.Range(0, Target.Length);
        }
        i=nexti;
    }
    agent.SetDestination(Target[i].transform.position);
    • • •
}
```

攻击状态时，通过施加攻击锁防止不断造成攻击伤害，在攻击时通过异步执行协程进行攻击和状态转换。

```
private Coroutine async;
private bool AttackLock;
```

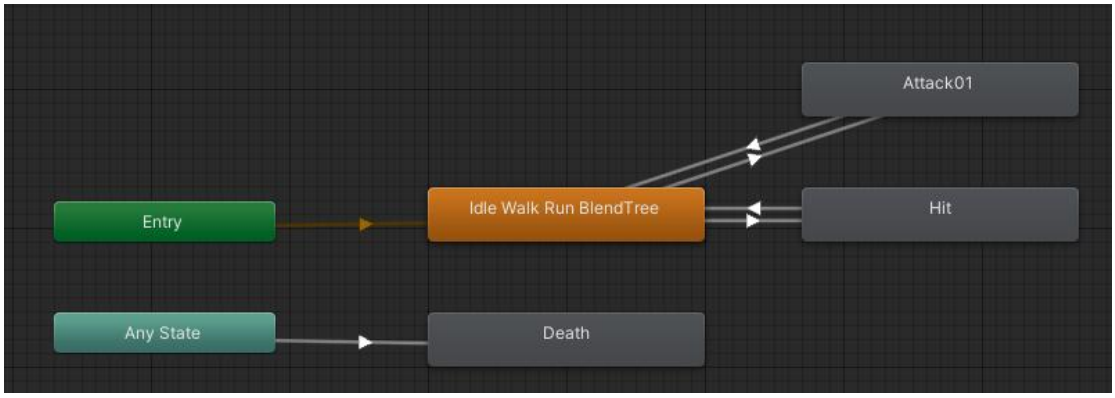
```
private float CD = 3f;
public override void StateAttack()
{
    if (AttackLock == false)
    {
        AttackLock = true;
        agent.speed = 0f;
        gameObject.transform.LookAt(player.transform.position);
        ani.SetTrigger("Attack1");

        if (async != null)
        {
            StopCoroutine(StateChange());
        }
        async=StartCoroutine(StateChange());
        ani.SetFloat("Speed", agent.speed, 0.1f, Time.deltaTime);

    }
    . . .
}
private IEnumerator StateChange( )
{
    yield return new WaitForSeconds(CD);
    AttackLock = false;
    async = null;
}
```

伤害采用碰撞检测，当检测到碰撞盒且 tag 为玩家时，判断玩家的状态是不是格挡，如果不是格挡则造成伤害。

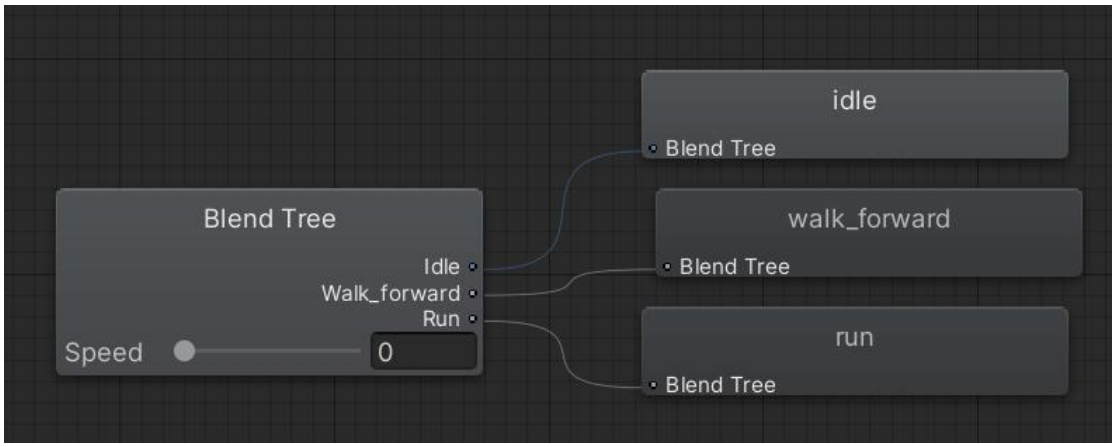
```
private void OnTriggerEnter(Collider collision)
{
    if (collision.tag == "Player")
    {
        PlayerBasics playerBasics = collision.GetComponent<PlayerBasics>();
        if (playerBasics != null)
        {
            if
(playerBasics._animator.GetCurrentAnimatorStateInfo(0).IsName("Blocking"))
            {
                Debug.Log("Blocked !");
            }
            else
            {
                playerBasics.takeDamage(currentDamage);
```

```
            }
        }
    }
}
```

2. 动画状态机介绍：



动画状态机与代码中的状态相符，空闲状态使用 BlendTree，包含空置、行走、奔跑三种状态。在速度为 0 时处于原地保持不动，速度为 2 为行走状态，速度为 6 为奔跑状态，巡逻时为行走，追逐玩家时为奔跑状态。

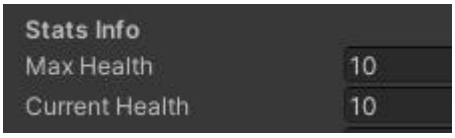


3. 怪物配置

怪物配置采用 ScriptableObject 进行配置，该文件配置最大血量和当前血量，攻击力数据绑定在武器之上。

```
public class EnemiesData : ScriptableObject
{
    [Header("Stats Info")]
    public int maxHealth;
    public int currentHealth;
}
```

牛头怪和骷髅怪属性分开配置，分别绑定在怪物身上。



在游戏开始时，通过脚本读取数据，使得当前生命值等于最大生命值。

```
public EnemiesData data;
public  int currentHealth;

private void Awake()
{
    currentHealth=data.maxHealth;
}
```

四、游戏 UI 与数值设计

1. 游戏 UI

在游戏开始时将会首先出现主菜单，这个菜单放置在游戏的主场景之外，在点击进入游戏按钮时加载主场景。使用场景管理器即可完成场景之间的跳转。

```
public void Jogar()
{   // load the main scene
    SceneManager.LoadScene("SampleScene");
    Cursor.lockState = CursorLockMode.None;
}
```

游戏中菜单的初始化，由于主场景中直接开始游戏，所有菜单组件要设置为不可见，等待玩家按键激活。

```
void Start()
{   // hide all the component of menu and unlock the cursor
    canvasMenuPrincipal.SetActive(false);
    canvasMenuInstrucoes.SetActive(false);
    canvasMenuCreditos.SetActive(false);
    canvasMenuHistoria.SetActive(false);
    canvasDescricaoPoder.SetActive(false);
    Cursor.lockState = CursorLockMode.None;
}
```

点击“玩法介绍”按键跳转到帮助页面；“退出游戏”按键则退出游戏。在退出游戏时要及时存档水晶编号数据。水晶是游戏中的复活点，在这里记录玩家退出游戏前最后一次点亮的水晶编号，作为下次开始游戏的复活点。

```
public void Sair()   // exit the game
{
    Application.Quit();   // quit the application
    storeDocument(crystalId); // store the crystal id to save it
```

```
#if UNITY_EDITOR            // in the editor mode or the release version, use different quit
            UnityEditor.EditorApplication.isPlaying = false;
        #else Application.Quit();
        #endif
    }
```

在游戏中按 M 键唤醒游戏菜单，游戏菜单和主菜单功能大体相同，区别是，在主菜单中开始游戏将加载一个新游戏，而在游戏菜单中游戏开始将会从当前游戏存档最近点亮的复活水晶处复活。

```
    public void Jogar() // start the game
    {
        // hide the main menu
        canvasMenuPrincipal.SetActive(false);
        Time.timeScale = 1; // set the time scale to normal value
        readDocument(); // read the last lighted crystal ID
        Cursor.lockState = CursorLockMode.Locked; // lock the cursor
    }
```

按 M 键唤醒菜单。按 B 键显示背包。

```
void Update()
    {
        if (Input.GetKeyDown(KeyCode.M))
        {   // press M, then show the main menu
            Cursor.lockState = CursorLockMode.None; // unlock the cursor
            Time.timeScale = 0;  // stop the game
            canvasMenuPrincipal.SetActive(true); // show the menu
        }
        else if (Input.GetKeyDown(KeyCode.B))
        {
            canvasMenuPrincipal.SetActive(false);
            canvasMenuHistoria.SetActive(true); // show the backpack
        }
    }
```

存档和读档函数。目前的实现比较简单，记录一个 int 类型的水晶 id 值。

```
    public void storeDocument()
    {   // store the value
        PlayerPrefs.SetInt("crystalId", crystalId);
        PlayerPrefs.Save();
    }
    public void readDocument()
    {   // read the value out
        this.cystalId = PlayerPrefs.GetInt("crystalId");
    }
```

背包是一个九宫格贴图，背景色是白色，表示没有获得该物品，它的 icon 属性是显示

对应物品的图片。在击杀怪物或者开宝箱之后，调用脚本中的显示函数在对应位置显示物品，在消耗资源如血瓶之后调用脚本中的隐藏函数表示资源已使用。

```
    void Start()
    { // hide all the object
        holder1.transform.Find("Icon").GameObject().SetActive(false);
        holder2.transform.Find("Icon").GameObject().SetActive(false);
        holder3.transform.Find("Icon").GameObject().SetActive(false);
        holder4.transform.Find("Icon").GameObject().SetActive(false);
        holder5.transform.Find("Icon").GameObject().SetActive(false);
        holder6.transform.Find("Icon").GameObject().SetActive(false);
        holder7.transform.Find("Icon").GameObject().SetActive(false);
        holder8.transform.Find("Icon").GameObject().SetActive(false);
        holder9.transform.Find("Icon").GameObject().SetActive(false);
    }
    public void showImage(int item)
    {
        switch (item)
        {
            case 1: // show the icon of ith image
                holder1.transform.Find("Icon").GameObject().SetActive(true);
                break;
            ……
        }
    }
    public void hideImage(int item)
    {
        switch (item)
        {
            case 1: // hide the icon of ith image
                holder1.transform.Find("Icon").GameObject().SetActive(false);
                break;
            ……
        }
    }
```

游戏中的在左上角显示的玩家血条、耐力条和经验值条，它们都只需要设置最大值和当前值两个函数。对于耐力条而言，它还需要随时间回复它的值，不过需要判定玩家当前状态，当玩家处于攻击动作施放中的状态时不可回复耐力值。

```
public class BarSlide : MonoBehaviour
{
    public Slider sliderHP; // health bar stamin bar and experience bar
    public Slider sliderST;
    public Slider sliderEX;
    public void SetMaxHealth(int maxHealth)//each bar has setmax, setcurrent method
```

```
    {
        sliderHP.maxValue = maxHealth;
        sliderHP.value = maxHealth;
    }
    public void SetCurrentHealth(int currenthealth)
    {
        sliderHP.value = currenthealth;
    }
}
```

随时间回复耐力值。在玩家处于非攻击动作时计时并按时间回复。

```
        private void UpdateStamin()
        {// check whether in the spare state
            if (_animator.GetCurrentAnimatorStateInfo(0).IsName("Idle Walk Run Blend")
&& !punched) {
                deltaStamin += Time.deltaTime; // calculate the lasting time
                if(currentStamin < totalStamin && deltaStamin > 0.1f)
                {   // recover the stamin value by time and set it to the UI
                    currentStamin += Mathf.RoundToInt(deltaStamin * recoverStamin);
                    if (currentStamin > totalStamin) currentStamin = totalStamin;
                    staminBar.SetCurrentStamina(currentStamin);
                    deltaStamin = 0; // after recovery, recalculate the value
                }
            }
            else
            {   // in the motion, set the time to be 0
                deltaStamin = 0;
            }
        }
```

2. 游戏数值

游戏玩家可以设置的参数包括普攻伤害、重击伤害、普攻耐力消耗、重击耐力消耗、武器附加伤害等，在击杀怪物或开宝箱获取经验值后可以升级。目前玩家设定为 5 个级别，使用静态数组存储了玩家每个级别所对应的强度，作为一种简单的实现。

```
    public int playerLevel = 1;
    public int[] maxHealth = { 100, 120, 140, 160, 180 };
    public int[] maxStamin = { 100, 120, 140, 160, 180 };
    public int[] recoverStamin = { 20, 25, 30, 35, 40 };
    public int[] consumeStaminPunch = { 30, 25, 23, 20, 18 };
    public int[] consumeStaminHeavy = { 35, 30, 25, 23, 20 };
    public int[] damagePunch = { 15, 20, 15, 30, 35 };
    public int[] damageHeavy = { 20, 30, 40, 50, 55 };
    public int[] injury = { 20, 18, 16, 14, 10 };
```