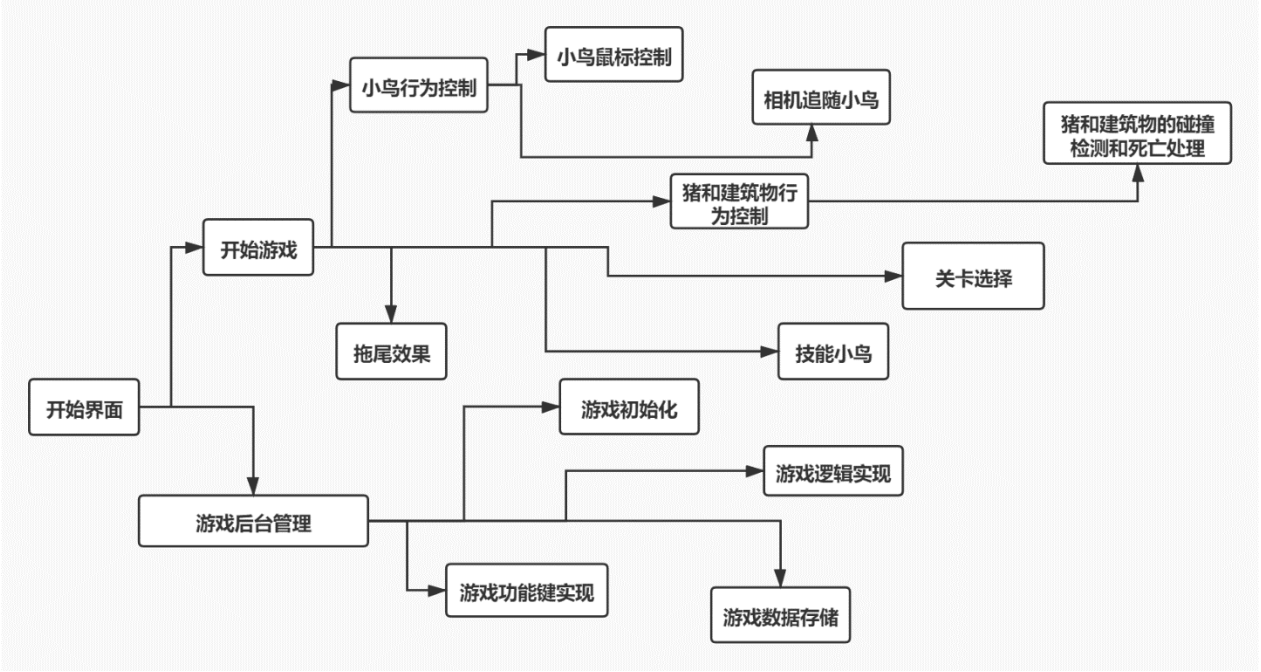
AngryBirds(Seasons) 软件设计说明书

20373623 软件学院 周恩申

1 总体设计

1.1 模块组成



1.2 模块结构说明

游戏分为可供玩家操控的游戏模块和不可被玩家操控的游戏后台管理板块组成，这两个模块在游戏软件打开的时候，就开始一同工作，密不可分。

2 程序描述

2.1 游戏后台管理模块

2.1.1 功能

实现游戏的全自动化管理，让游戏在不被外界干扰的情况下自行运转和调整。

2.1.2 输入项目

鼠标点击按钮

2.1.3 输出项目

游戏按逻辑正常进行；游戏页面跳转；游戏功能键按键反馈；界面更新；

2.1.4 算法

（1） 游戏的初始化操作

（2） 游戏逻辑的实现

（3） 游戏功能键的设置

（4） 游戏的数据存储

2.1.5 代码

```
public List<Bird> birds;    //用列表的形式表示小鸟的集合
public List<Pig> pig;       //用列表的形式表示猪的集合
public static GameManager _instance;    //当前的游戏管理对象

private Vector3 originPos;    //初始小鸟的位置

public GameObject win;       //获取面板的输赢状况
public GameObject lose;

public GameObject[] stars;  //从面板中获取星星集合

private int starNum = 0;    //记录每一关的得到星星数（方便之后数据的存储）

private int totalNum = 15;  //该场景下所有的关卡数
```

GameManager.cs 变量申明

下面是游戏的初始化：

```
private void Awake()         //初始化
{
    _instance = this;            //读取游戏管理对象
    if (birds.Count > 0)         //如果有小鸟
    {
        originPos = birds[0].transform.position;    //初始位置为第一只小鸟的位置
    }

}
Unity 消息|0 个引用
private void Start()
{
    Initialized();           //初始化
}
2 个引用
private void Initialized()  //初始化
{
    for (int i = 0; i < birds.Count; i++) //遍历小鸟集合
    {
        if (i == 0)  //如果是第一只小鸟
        {
            birds[i].transform.position = originPos;    //把小鸟放置在初始位置，使其更柔和
            birds[i].enabled = true;    //该小鸟可用
            birds[i].sp.enabled = true; //该小鸟的物理效果开启（sp是在Bird脚本里的弹动与摆动的物理效果）
            birds[i].canMove = true;    //小鸟可以被鼠标点击
        }
        else
        {
            birds[i].enabled = false;    //该小鸟不可用
            birds[i].sp.enabled = false; //该小鸟的物理效果关闭
        }
    }
}
```

GameManager.cs 初始化

下面是具体游戏逻辑实现：

```
public void NextBird() //判断游戏逻辑（如果已经获胜了，就不再需要进行Initialized()的初始化操作了，此时剩余的小鸟都在地面上）
{
    if (pig.Count > 0)
    {
        if (birds.Count > 0) //准备初始化下一只小鸟
        {
            Initialized();  //初始化
        }
        else//输了
        {
            lose.SetActive(true);   //启动输了的画面
        }
    }
    else//猪已经全部消灭，获得胜利
    {
        win.SetActive(true);        //启动赢了的画面
    }
}

1 个引用
public void ShowStars()     //胜利时出现星星
{
    StartCoroutine("show");    //开启协程
}

0 个引用
IEnumerator show()
{
    for (; starNum < birds.Count + 1; starNum++)
    {
        if (starNum >= stars.Length)
            break;                    //如果超过三个星星就直接跳出循环
        else
        {
            yield return new WaitForSeconds(0.2f);  //等待0.2s
            stars[starNum].SetActive(true);              //开启星星特效
        }
    }
}
```

GameManager.cs 游戏逻辑

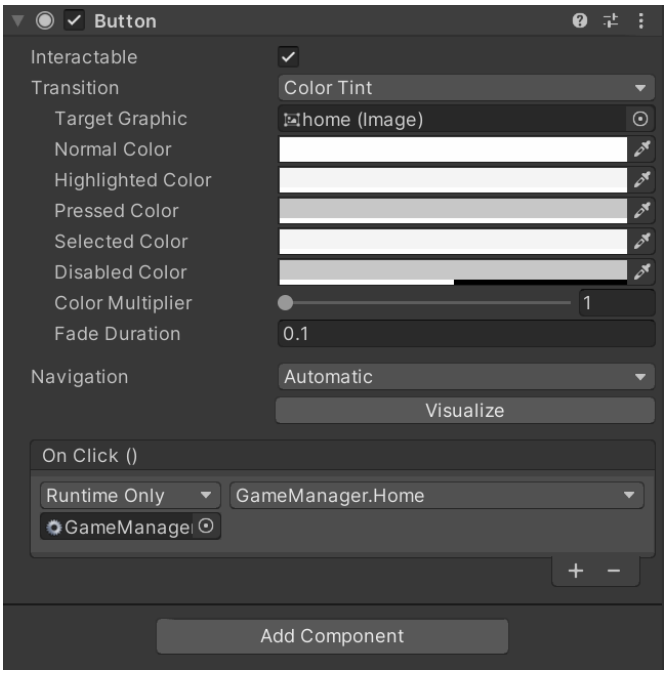
下图是游戏功能键的设置：

```
public void StartTOPlay()
{
    SceneManager.LoadScene(1);  //File中Build Setting中对于01-Level场景的模块序号为1
}
0 个引用
public void Replay()    //如果按下重玩按钮
{
    SavaData();                 //保存数据
    Time.timeScale = 1;
    SceneManager.LoadScene(2);  //File中Build Setting中对于02-Game场景的模块序号为2
}

0 个引用
public void Home()
{
    SavaData();                 //保存数据
    Time.timeScale = 1;
    SceneManager.LoadScene(1);  //File中Build Setting中对于01-Level场景的模块序号为1
}

0 个引用
public void NextLevel()    //如果按下下一关的按钮
{
    SavaData();
    Time.timeScale = 1;
    string levelNum = PlayerPrefs.GetString("nowLevel");
    //去掉字符串里带level的字符，即得到当前是第几关
    levelNum = levelNum.Replace("level", "");
    //关卡数加一 这里还要判断一下当前i是否大于当前地图里边最大的关卡数
    int i = int.Parse(levelNum) + 1;
    levelNum = "level" + i.ToString();
    PlayerPrefs.SetString("nowLevel", levelNum);
    SceneManager.LoadScene(2);
}
```

GameManager.cs 功能键设置



按钮添加 GameManager.cs 功能键函数

下图是游戏数据存储：

```
public void SavaData()
{
    //PlayerPrefs.SetInt("string",int num)              通过键值对存储该关卡的星星数量
    /*
     * 使用本地持久化类PlayerPrefs完成Unity整个游戏的数据存储
     * 很有技巧的数据存储
     * 避免的外部数据库的使用
     */
    if (starNum > PlayerPrefs.GetInt(PlayerPrefs.GetString("nowLevel")))    //如果获得的星星大于历史记录
    {
        PlayerPrefs.SetInt(PlayerPrefs.GetString("nowLevel"), starNum);      //获取nowLevel对应的level + 该关卡的序号，存下该关卡的获得星星数
    }
    //存储所有的星星个数
    int sum = 0;
    for (int i = 1; i <= totalNum; i++)
    {
        sum += PlayerPrefs.GetInt("level" + i.ToString());              //累加该场景目前所有星星数（由于显示text中的分子）
    }


    PlayerPrefs.SetInt("totalNum", sum);            //通过键值对存储该场景totalNum个关卡所获得的所有星星数
    //print(PlayerPrefs.GetInt("totalNum"));
}
```

GameManager.cs 游戏数据存储

2.2 游戏模块

2.2.1 功能

（1） 控制小鸟的物理组件

（2） 控制相机追随小鸟移动

（3） 控制猪和建筑物的物理组件

（4） 控制猪和建筑物的受伤或死亡状态

（5） 用鼠标控制小鸟技能的释放

（6） 控制场景与关卡的开启和关闭

2.2.2 输入项目

鼠标的点击和拖拽

2.2.3 输出项目

小鸟位置的改变；小鸟图片的更改；猪和建筑物位置的改变；猪和建

筑物的图片更改；得分方式的反馈；按钮反馈；音效反馈

2.2.4 算法

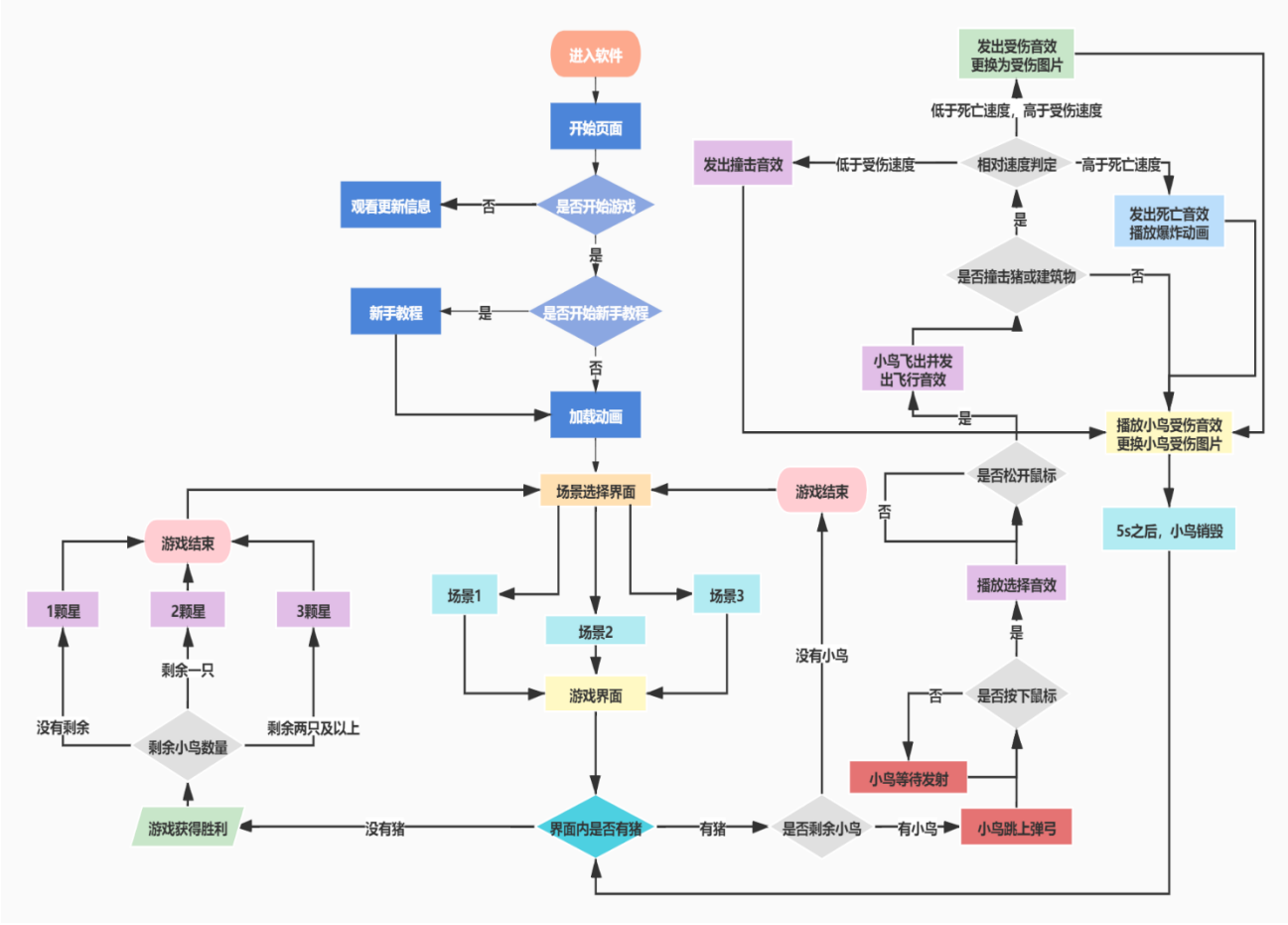
（1） 采用 OnMouseDown()和 OnMouseUp()来控制鼠标行为

（2） 运用动量守恒定律与 collision.relativeVelocity()碰撞组件完成碰撞检测

（3） 采用前驱法完成对场景和关卡的开启判断

（4） 利用 Bird.cs 中的虚方法 ShowSkill()，方便进行重载和继承，完成小鸟技能的实现

2.2.5 程序逻辑



2.2.6 代码

```
private void Awake()            //初始化
{
    sp = GetComponent<SpringJoint2D>(); //获取当前弹动与摆动的物理效果
    rg = GetComponent<Rigidbody2D>();   //获取当前刚体的物理运动状态
    myTrail = GetComponent<TestMyTrail>();//获取当前拖尾状态
    render = GetComponent<SpriteRenderer>();    //获取当前小鸟的状态的图片
}
Unity 消息|0 个引用
private void OnMouseDown()  //鼠标按下
{
    if(canMove)
    {
        Audioplay(select);      //开启选择音效
        isClick = true;
        rg.isKinematic = true;  //开启物理动力学
    }

}
Unity 消息|0 个引用
private void OnMouseUp()    //鼠标抬起
{
    if (canMove)
    {
        isClick = false;
        rg.isKinematic = false; //关闭物理动力学
        Invoke("Fly", 0.1f);    //延迟0.1s，给予足够多的时间进行物理计算

        //禁用划线组件
        right.enabled = false;  //弹弓右子中心的划线效果消失
        left.enabled = false;   //弹弓左子中心的划线效果消失

        canMove = false;        //飞出去的小鸟不再被鼠标控制
    }

}
```

```
void Fly()
{
    isFly = true;           //在飞行途中
    isReleased = true;      //已经被释放，此时与弹弓没有联系
    Audioplay(fly);         //开启飞行音效
    myTrail.StartTrails();  //开启拖尾效果
    sp.enabled = false;     //弹动与摆动的物理效果消失
    Invoke("Next", 5);      //小鸟飞出5s之后，分别依次执行移除列表中的小鸟，销毁小鸟，出现小鸟消失特效
}
1 个引用
void Line() //划线操作（两点确定一条直线）
{
    //启用划线组件
    right.enabled = true;       //弹弓右子中心的划线效果开启
    left.enabled = true;        //弹弓左子中心的划线效果开启

    right.SetPosition(0, rightPos.position);        //两点确定一条直线
    right.SetPosition(1, transform.position);

    left.SetPosition(0, leftPos.position);          //两点确定一条直线
    left.SetPosition(1, transform.position);
```

Bird.cs 鼠标控制

下图是相机追随小鸟移动的部分：

```
private void Update()
{
    if (EventSystem.current.IsPointerOverGameObject())      //是否在点击UI界面（解决暂停后点击会触发小鸟技能的bug）
        return;
    if (isClick)    //鼠标一直按下,进行位置跟随
    {
        transform.position = Camera.main.ScreenToWorldPoint(Input.mousePosition);   //坐标系转换

        //transform.position += new Vector3(0, 0, 10);
        transform.position += new Vector3(0, 0, -Camera.main.transform.position.z); //小鸟图层深度的改变

        if(Vector3.Distance(transform.position,rightPos.position) > maxDis) //进行位置限定
        {
            Vector3 pos = (transform.position - rightPos.position).normalized;  //获得从弹弓子中心指向小鸟的单位向量
            pos *= maxDis;  //最大长度向量
            transform.position = pos + rightPos.position;   //限制小鸟的活动范围
        }

        Line();     //开始划线，出现皮筋的效果
    }

    //相机跟随
    float posX = transform.position.x;  //获取到待飞小鸟的一维x轴的位置
    float posY = transform.position.y;  //获取到待飞小鸟的一维x轴的位置

    //Lerp( , , )通过向量插值实现主相机平滑地跟随，第一个点是当前目标点，第二个是目的地，第三给为当前运动速率(平滑度 * 时间间隔)
    //Mathf.Clamp(value,min,max)     把value限制在min和max之间                                    相机x的范围

    Camera.main.transform.position = Vector3.Lerp(Camera.main.transform.position, new Vector3(Mathf.Clamp(posX, 0, 40), Mathf.Clamp(posY, 0, 15),
        Camera.main.transform.position.z), smooth * Time.deltaTime);

    if (isFly)       //如果处于飞行状态
    {
        if(Input.GetMouseButtonDown(0))      //此时单击鼠标
        {
            ShowSkill();            //触发小鸟技能
        }
    }
```

图 5.9 Bird.cs 相机移动

下图是碰撞检测和死亡处理部分：

```
private void OnCollisionEnter2D(Collision2D collision)//碰撞检测
{
    //print(collision.relativeVelocity.magnitude);      //显示相对速度，合理选取受伤和死亡的取值范围
    //collision.relativeVelocity（相对速度）是向量，需要转换为标量
    if(collision.gameObject.tag == "Player")    //如果小鸟碰撞的是猪或者建筑物
    {
        Audioplay(birdCollision);    //播放小鸟碰撞时使用的音乐组件
        collision.transform.GetComponent<Bird>().Hurt();  //更新小鸟受伤图片
    }

    if (collision.relativeVelocity.magnitude > maxSpeed)//碰撞相对速度大于最大速度，直接死亡
    {
        Dead();     //猪死亡的效果处理
    }
    else if (collision.relativeVelocity.magnitude > minSpeed && collision.relativeVelocity.magnitude < maxSpeed) //相对速度位于最大和最小速度之间为受伤状态
    {
        render.sprite = hurt;   //图片更新为受伤图片
        Audioplay(hurtClip);    //播放猪或者建筑物碰撞时使用的音乐组件
    }
}

2 个引用
public void Dead()
{
    isDead = true;      //已经死亡
    if(isPig)
    {
        GameManager._instance.pig.Remove(this);     //如果是猪，从列表中移除它
    }
    Destroy(gameObject); //猪或建筑物死亡后销毁
    Instantiate(boom, transform.position,Quaternion.identity);  //产生爆炸效果

    GameObject go = Instantiate(score, transform.position + new Vector3(0,0.5f,0), Quaternion.identity);    //产生得分效果
    Destroy(go,2);   //销毁得分特效

    Audioplay(dead); //播放猪死亡或者建筑毁坏是使用的音乐组件
}
```

Pig.cs 碰撞检测和死亡处理

下图是场景与关卡开启部分：

```
private void Start()
{
    if (transform.parent.GetChild(0).name == gameObject.name)    //如果是第一关
    {
        isSelect = true;                //该关卡可以被选择
    }
    else
    {
        //前驱算法开启下一关
        int beforeNum = int.Parse(gameObject.name) - 1; //获得前一关的关卡名
        if (PlayerPrefs.GetInt("level" + beforeNum.ToString()) > 0) //使用前驱算法，判断是否满足开启条件
        {
            isSelect = true;        //可选择
        }
    }

    if (isSelect)
    {
        image.overrideSprite = levelBG; //替换为可选择后的关卡图片
        transform.Find("num").gameObject.SetActive(true);    //激活num对应的关卡图片

        //通过字符串拼接获得关卡名字，然后获得该关卡对应的星星数量
        int count = PlayerPrefs.GetInt("level" + gameObject.name);
        //
        if (count > 0)                    //如果该关卡星星数大于零
        {
            for (int i = 0; i < count; i++)  //遍历，同时让相等数量的星星显示在选择关卡界面
            {
                stars[i].SetActive(true);   //显示星星
            }
        }
    }
}
```

LevelSelect.cs 场景与关卡开启

```
11 个引用
public virtual void ShowSkill()      //小鸟各种不同的技能(虚方法，方便重载继承)
{
    isFly = false;           //飞行过程结束，不可再实现小鸟技能
}
```

Bird.cs 中 ShowSkill()的声明

```
public class YellowBird : Bird      //继承一般红鸟的脚本
{
    7 个引用
    public override void ShowSkill()     //重写技能板块
    {
        base.ShowSkill();          //继承基类
        rg.velocity *= 2;          //改变子类，使其速度变为原来的两倍
    }
}
```

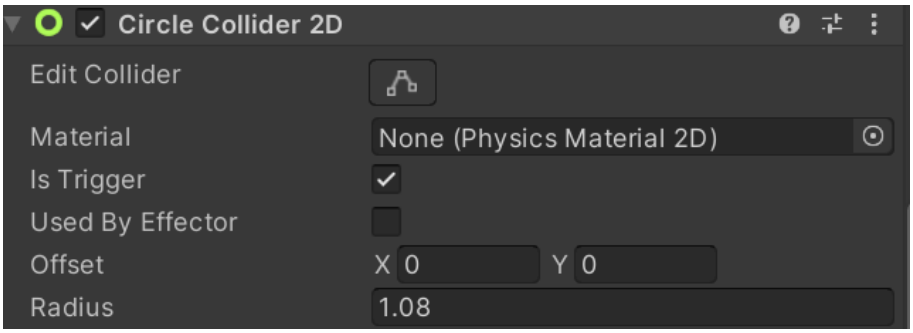
YellowBird.cs 中 ShowSkill()的重写

```
public class GreenBird : Bird      //继承一般红鸟的脚本
{
    7 个引用
    public override void ShowSkill()    //重写技能板块
    {
        base.ShowSkill();   //继承基类
        Vector3 speed = rg.velocity;    //修改速度
        speed.x *= -0.5f;                   //x轴反向
        //y轴变为0（按照原游戏，应该是加速度朝上的匀减速运动，之后再做加速度向上的加速运动，画出一个类圆弧的形状）
        speed.y = 0;
        rg.velocity = speed;
    }
}
```

GreenBird.cs 中 ShowSkill()的重写

```
public class OrangeBird : Bird      //继承一般红鸟的脚本
{
    private CircleCollider2D circle;    //未膨胀前的圆形碰撞范围
    public Sprite clickOrange;          //点击鼠标膨胀后的图片
    private SpriteRenderer render_orange;
    7 个引用
    public override void ShowSkill()    //重写技能板块
    {
        base.ShowSkill();       //继承基类
        circle = GetComponent<CircleCollider2D>();      //获取未膨胀前的圆形碰撞范围
        render_orange = GetComponent<SpriteRenderer>(); //获取未膨胀前的图片信息
        circle.radius += 1;     //膨胀后圆形半径增加
        render_orange.sprite = clickOrange;             //更换图片
    }
}
```

OrangeBird.cs 中 ShowSkill()的重写



Circle Collider 2D 的注意事项

```
public class BlackBird : Bird                //继承一般红鸟的脚本
{

    public List<Pig> blocks = new List<Pig>(); //这个集合里存放以黑炮为圆心的圆中可爆炸销毁猪和建筑物（有Enemy的标签）
    Unity 消息|0 个引用
    private void OnTriggerEnter2D(Collider2D collision) //进入圆圈里的触发区域
    {
        if (collision.gameObject.tag == "Enemy")  //如果标签是Enemy
        {
            blocks.Add(collision.gameObject.GetComponent<Pig>());
        }
    }
    Unity 消息|0 个引用
    private void OnTriggerExit2D(Collider2D collision)  //离开圆圈里的触发区域
    {
        if (collision.gameObject.tag == "Enemy")  //如果标签是Enemy
        {
            if (collision.GetComponent<Pig>().isDead == false)   //之前没有触发Dead方法
                blocks.Remove(collision.gameObject.GetComponent<Pig>());
        }
    }

    7 个引用
    public override void ShowSkill()        //重写技能板块
    {
        base.ShowSkill();                    //继承基类
        if (blocks.Count > 0 && blocks != null)     //如果存在障碍物
        {
            for (int i = 0; i < blocks.Count; i++)  //爆炸清除障碍物
            {
                blocks[i].Dead();    //爆炸清除障碍物
            }
        }
        OnClear();  //处理后事
    }
```

BlackBird.cs 中 ShowSkill()的重写

3 系统数据结构设计

3.1 逻辑结构设计

（1） 小鸟状态

```
public class Bird : MonoBehaviour   //实现小鸟的拖拽
{
    public float maxDis = 1.5f;         //公共变量，用于输入最大长度以限制小鸟位置

    private bool isClick = false;   //bool变量判断鼠标是否点击
    [HideInInspector]
    public bool canMove = false;    //保证小鸟飞出去之后与下一只小鸟上弹弓之间该小鸟不可以被鼠标控制
    public bool isFly = false;      //判断是否处于飞行状态，这样方便让黄锋（黄鸟）实现继承，同时加上飞行过程中单击左键实现加速的效果

    [HideInInspector]//判断小鸟在弹弓上时，非暂停情况下有没有拉扯然后松开（用于解决在暂停页面也可以点击小鸟放在弹弓上以至于取消暂停后小鸟失效的bug）
    public bool isReleased = false;


    [HideInInspector]              //把公共变量隐藏起来，不让用户改动
    public SpringJoint2D sp;       //用于控制弹动与摆动的物理效果(设置为公共是因为可以传输至GameManager中管理待发射小鸟的物理效果)
    protected Rigidbody2D rg;      //用于控制飞出后小鸟刚体的状态(设置为保护类型，外界看不到，但是内部可以调用)

    public LineRenderer right;      //弹弓右边划线皮筋
    public LineRenderer left;       //弹弓左边划线皮筋
    public Transform rightPos;      //公共变量，用于传入弹弓右子中心的位置
    public Transform leftPos;      //公共变量，用于传入弹弓左子中心的位置

    public Sprite hurt;             //小鸟受伤的图片
    protected SpriteRenderer render;  //当前小鸟的状态图片
    public GameObject boom;        //小鸟消失的特效
    protected TestMyTrail myTrail;    //小鸟拖尾效果

    public float smooth = 3.5f;         //解决相机移动的平滑度值

    public AudioClip select;        //选择小鸟是使用的音乐组件
    public AudioClip fly;           //小鸟飞行是使用的音乐组件
```

（2） 猪和建筑物状态

```
public class Pig : MonoBehaviour
{
    //使用动量守恒：设小鸟质量为m，速度为v，猪质量为M，碰撞后小鸟的速度为（M-m)*v/(M+m)，猪的速度为2mv/(M+m)
    //因此比较小鸟与猪碰撞前的相对速度和比较碰撞后猪的速度的变化量大小其实一样，都可以判断猪的状态

    public float maxSpeed = 10;
    public float minSpeed = 2;
    public Sprite hurt;             //猪受伤的图片
    private SpriteRenderer render;  //当前猪的状态图片
    public GameObject boom;         //猪的死亡爆炸效果
    public GameObject score;        //猪死亡的得分效果

    public AudioClip hurtClip;       //猪或者建筑物碰撞时使用的音乐组件
    public AudioClip dead;           //猪死亡或者建筑毁坏是使用的音乐组件
    public AudioClip birdCollision;  //小鸟碰撞时使用的音乐组件

    public bool isPig = false;      //由于猪和木块的效果差不多，只是相对速度有所差别，所以用一个bool判断是否为猪即可，其他的都可以共同调用
    public bool isDead = false;     //判断当前物体是否已经触发Dead方法
```

（3） 场景与关卡选择

```
public class MapSelect : MonoBehaviour
{
    public int starNum = 0;          //解锁新场景需要的星星数量
    private bool isSelect = false;   //判断是否可以开启新场景
    public bool isThird = false;     //判断是否是第三个场景，因为三个场景的星星总数不同

    public GameObject locks;         //场景封锁
    public GameObject stars;         //该场景获得的星星数量

    public GameObject panel;         //场景下的关卡（完成与LevelSelect的交互）
    public GameObject map;           //整个游戏的所有场景

    public Text starsText;           //处理选择场景表面的星星比值
    public int startNum = 1;        //起始关卡数
    public int endNum = 5;           //终止关卡数
```

（4） 游戏后台管理

```
public class GameManager : MonoBehaviour
{
    public List<Bird> birds;     //用列表的形式表示小鸟的集合
    public List<Pig> pig;        //用列表的形式表示猪的集合
    public static GameManager _instance;     //当前的游戏管理对象

    private Vector3 originPos;     //初始小鸟的位置

    public GameObject win;        //获取面板的输赢状况
    public GameObject lose;

    public GameObject[] stars;  //从面板中获取星星集合

    private int starNum = 0;     //记录每一关的得到星星数（方便之后数据的存储）

    private int totalNum = 15;  //该场景下所有的关卡数
```

3.2 逻辑结构设计

此游戏并未涉及到负载的数据结构，唯一重点在于场景与关卡的存储以及过关数据的保留和场景与关卡之间的跳转。

由于 Unity 的本地持久化类 PlayerPrefs 包含类似于 Hash 一样的键值对存储，所以合理利用内置的数据结构可以高效的完成设计目的。

存储：本地持久化类 PlayerPrefs 在存储方面有两个方法。

（1） PlayerPrefs.SetInt("string",int num)
通过键值对存储该关卡的星星数量，具体方式为 PlayerPrefs 在内部存储空间申请一个空间，并命名为“string”，然后在该空间内部存储整形数据 num，表示该关卡的星星数量。

（2） PlayerPrefs.SetString("nowLevel", levelNum);
通过键值对存储玩家需要打开的关卡编号，具体方式为 PlayerPrefs 在内部存储空间申请一个空间，并命名为“nowLevel”，然后在该空间内部存储字符串类型数据 levelNum，表示玩家需要打开的关卡编号。

访问：本地持久化类 PlayerPrefs 在获取方面有两个方法。

（1） PlayerPrefs.GetInt(string);
通过键值对获取该关卡的星星数量，具体方式为 PlayerPrefs 在内部存储空间访问命名为“string”的存储空间，然后取出在该空间内部存储整形数据 num。

(2) PlayerPrefs. GetString("nowLevel");

通过键值对获取玩家需要打开的关卡编号，具体方式为 PlayerPrefs 在内部存储空间访问命名为“nowLevel”的存储单元，然后获取在该空间内部存储字符串类型数据 levelNum。

下面为具体使用案例：

```
if (starNum > PlayerPrefs.GetInt(PlayerPrefs.GetString("nowLevel")))     //如果获得的星星大于历史记录
{
    PlayerPrefs.SetInt(PlayerPrefs.GetString("nowLevel"), starNum);      //获取nowLevel对应的level + 该关卡的序号，存下该关卡的获得星星数
}
//存储所有的星星个数
int sum = 0;
for (int i = 1; i <= totalNum; i++)
{
    sum += PlayerPrefs.GetInt("level" + i.ToString());               //累加该场景目前所有星星数（由于显示text中的分子）
}


PlayerPrefs.SetInt("totalNum", sum);            //通过键值对存储该场景totalNum个关卡所获得的所有星星数
//print(PlayerPrefs.GetInt("totalNum"));
```

```
SavaData();
Time.timeScale = 1;
string levelNum = PlayerPrefs.GetString("nowLevel");
//去掉字符串里带level的字符，即得到当前是第几关
levelNum = levelNum.Replace("level", "");
//关卡数加一 这里还要判断一下当前i是否大于当前地图里边最大的关卡数
int i = int.Parse(levelNum) + 1;
levelNum = "level" + i.ToString();
PlayerPrefs.SetString("nowLevel", levelNum);
SceneManager.LoadScene(2);
```