《软件设计文档》

一、技术选型理由

（一）选择 Unity3d 作为游戏开发引擎

全平台支持：

Unity3D 游戏引擎支持的平台：PC, Mac OS, Web, iOS, Android, XBOX360, PS3, Wii。这种跨平台能力，让人很难再挑剔了。特别要关注的是 Web，iOS 和 Android 平台，这几个平台的重要性不用多说了。

全新的授权、盈利模式

过去的游戏引擎都是通过卖 license 赚钱的，现在 Unity3D 打破了这一常规，除了收费的 license 还提供了完全免费的简化版本，而且另外提供了 Union 和 Asset Store 销售平台，任何游戏制作者都可以把自己的游戏放到 Union 平台销售，赚到的钱二八分成。Asset Store 更是为单个的美术资源提供了一个销售平台，一个模型或一个骨骼动画都可以拿到这里销售，为游戏开发者提供了一站式的销售、开发平台。

免费的 license 一方面扩大了用户群，另一方面也可以通过 Union 平台来为 Unity3D 带来额外的收入，可以说是一石二鸟。

开发：

脚本语言在 Unit3D 游戏开发中占据了主角的位置。Unity3D 提供了三种脚本语言的支持：Javascript、C#、Boo，Boo 是 Python 在.Net 上的实现。值得注意的是 Unity3D 通过 Mono 实现了.Net 代码的跨平台。这样对数据库、xml、正则表达式等技术的支持都因为采用了.Net 而得到完美的解决。

脚本语言的动态特性让我们可以方便的通过名称、层次结构、tags 等方式访问所有的对象。当然更大的好处是脚本语言的跨平台性，绝大部分平台相关的代码都放到了引擎的内部，而游戏内容相关的代码都可以跨平台执行。游戏开发者终于可以不再为跨平台头疼了。

渲染：

支持 100 多种光照材质 shader，20 多种后期处理效果。Unity3D 的 surface shader 还是比较灵活的，可以非常自由的定制。不过不清楚后期处理有没有提供这么灵活的扩展能力。当然 Unity3D 提供的各种后期处理效果已经非常优秀了，扩展能力更多的是体现架构的可扩展性。

Unity3D 的渲染性能优化也是比较有自己的特色的。

其他：

Unity3D 对网络通信的支持比较全面，不过开发 MMO 的话还是不能满足需求的。因此 Unity3D 推荐了几个 MMO 的服务器平台可以配合使用，包括 Electrotank Universe Platform, Photon Socket Server,

Smartfox Server。另外 Unity3D 可以直接运行在浏览器页面内也是未来的一个趋势。

Unity3D 提供的 Unit Asset Server 方便了对游戏资源的管理和版本控制。

Unity3D 提供了包括编辑器、beast 渲染器、tree creator 等大量的辅助工具

（二）选用 MVC 框架

1、耦合性低

2、重用性高

3、生命周期成本低

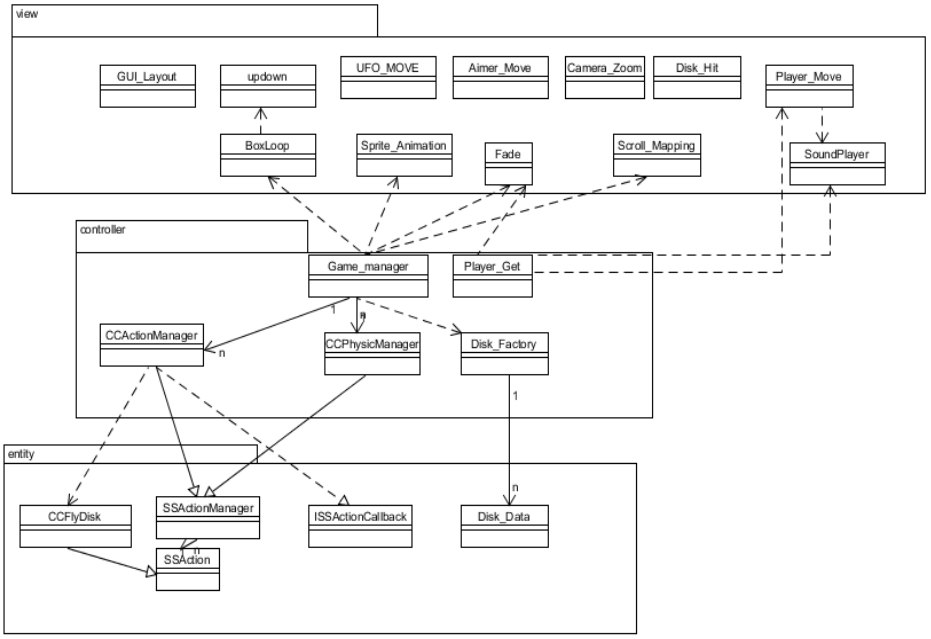
4、部署快

5、可维护性高

6、有利软件工程化管理

二、架构设计

（一）MVC 架构



值得说明的是 view 中的有些类是直接绑定的 unity 对象上的，是对象属性本身，不需要和控制器交互.

场景的实现通过 Game_Manager 控制器来实现。为了更好的进行迭代或者扩展维护，我们采用二级控制器动作管理类、人物控制类、工厂类来实现代码的更高效重用和更易维护。

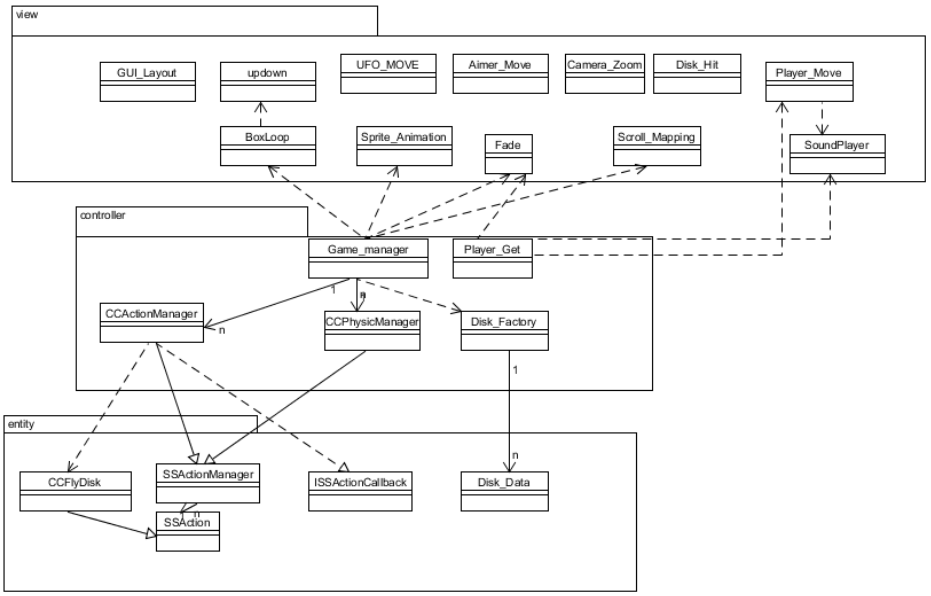
Game_Manager 中的动作均通过动作管理类来实现。这样做的好处是当我们需要增加新的动作时，只需要建立一个新的 CCAtion（继承自 SSAction）通过 CCActionManager 来调用它即可；非常方便和高效。同样当我们需要维护或者修改某个动作时，只需要修改其对象的 CCAction 类即可。

人物控制类的诞生是因为在我们游戏中，主角是唯一的也是核心，所以进行单独的管理。

工厂类的目的是为了节约系统资源和便捷的提供游戏对象供 Game_Manager 使用。在 Unity 引擎中，如果不断的在运行时不断消耗系统资源创建新对象，其代价是昂贵的。我们通过工厂类，使得部分资源可以被回收利用（并不是说销毁这部分资源，而是当他们闲置的时候加入重新加入队列以供再此使用）

三、模块设计

（一）程序类图到包图的映射



在 view 模块中的类，将游戏数据以一定的方式展示在游戏中。

类名	用途
GUI_Layout	这个类负责提供用户交互界面：菜单之类
UFO_MOVE	实现场景内飞碟的随机运动
Box_Loop	组合不同预制形成游戏场景总的地面运动
UpDown	实现地面的抬升和下降
Sprite_Animation	播放精灵的动画

Fade	淡化功能，游戏结束时会发生
Scroll_Mapping	实现背景地图的滚动播放，达到卷轴 2d 的目标
Camera_Zoom	拉近或者放远镜头，实现 3D 的视角变化
Disk_Hit	飞碟碰撞的效果
Player_Move	人物移动的方法
Sound_Player	音效播放
Aimer_Move	准心移动

在 controller 模块中的类，从 view 中得到消息，从 model 中加载数据传递给 view，在场景中做改变

类名	用途
Game_Manager	游戏场景的控制类
CCActionManager	游戏对象运动管理类
CCPhysics	游戏对象的物理运动管理类（和上区分，物理引擎是 Unity 中一种特殊的运动引擎）
Player_Get	玩家控制类
Disk_Factory	飞碟工厂类，提供飞碟，回收飞碟

model 模块负责提供数据给 Controller

类名	用途
CCFlyDisk	飞碟的运动类
SSAction	动作类的基类
SSActionManager	动作管理类的基类
ISSActionCallback	消息接口，当动作完成时调用，每个 ActionManager 都需要实现
Disk_Data	飞碟数据类，每个飞碟都有，记录飞碟数据

（四）软件设计技术

说明：若源码中注释有乱码可以换打开的文本编辑器，就能正常显示

一、面向对象的编程

采用继承的方法来实现动作的可扩展

SSAction.cs 中，我们定义了一个动作的基类。包括该动作状态信息，成员变量 transform,gameobejct,callback 回调。定义了虚函数 start(),update()（这是 Unity Monobehavior 自带的）。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SSAction : ScriptableObject {
    public bool enable = true;
    public bool destory = false;

    public GameObject gameobject { get; set; }
    public Transform transform { get; set; }
    public ISSActionCallback callback { get; set; }

    protected SSAction() {}

    // Use this for initialization
    public virtual void Start () {
        throw new System.NotImplementedException();
    }

    // Update is called once per frame
    public virtual void Update () {
        throw new System.NotImplementedException();
    }
}

```

当我们需要建立一个新的动作，比如飞碟运动的时候：我们需要先写一个飞碟运动的类，CCDiskFly.cs。我们只需要让 CCDiskFly 继承 SSAction 类，重定义 start(),update()即可。GetSSAction（）为外部类创建一个 CCFlyDisk 实例提供方法。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

//□□□□□□碟的□□作
public class CCFlyDisk : SSAction {
    private float acc;
    private Vector3 force;

    private float time;
    // Use this for initialization
    public override void Start () {
        enable = true;
        this.gameobject.transform.position = new Vector3 (Random.Range (-2f, 5f), Random.Range (7f, 10f), 0);
        this.gameobject.GetComponent<Rigidbody> ().velocity = new Vector3 (Random.Range (-3f, 3f), Random.Range (-1f, -3f), 0);
        this.gameobject.GetComponent<Rigidbody> ().AddForce (new Vector3 (Random.Range (-1.0f, 1.0f), Random.Range (2.0f, 5.0f), 0));
    }

    // Update is called once per frame
    public override void Update () {
    }

    public static CCFlyDisk getSSAction() {
        CCFlyDisk ac = ScriptableObject.CreateInstance<CCFlyDisk> ();
        return ac;
    }
}
```

在程序中还有这样的例子，比如 SSActionManager 父类，ISSSceneCallback 接口和 CCActionManager 的关系。

面向对象的编程使得我们可以更加有效实现代码复用和更加高效的实现程序的扩展和维护。

二、结构化编程-自顶向下

在程序设计上，我们采用结构化编程。

在场景控制中，我们采用将动作管理，人物管理，游戏对象管理,场景布置，声效播放，动画播放，UI 交互分离的做法。Game_Mangaer 类通过 CCActionManager 实现将动作，Player_Get 来控制玩家角色，Disk_Factory 管理游戏对象的生产, Scrop_Mapping 来控制背景的卷动, Box_Loop 来产生地形关卡，Sound_Player 来播放音效，Sprite_Animation 来控制动画播放，GUI_Layout 来实现交互。

然后在每个类中在细化。比如 Box_Loop（），就将 box 的产生分为诞生，销毁，移动等动作。

```
    public GameObject MakeBox()
    {

        GameObject x_Box;

        //随机□□□□□□□□不同的地面□□□□□体□□□□行□□□□初化，增加游□□□□□□□□景的多□□□□性
        int ran = Random.Range(0,5);
        x_Box = Instantiate(Box[ran], new Vector3(30,0,0),transform.rotation) as GameObject;
        return x_Box;

    }

    public void SetBox(GameObject x_Box)
    {
        B_Box=A_Box;
        A_Box = x_Box;

    }

    public void Death(GameObject b_Box)
    {
        //销毁指定□□□□□体，□□□□□初化新的地面□□□□□体
        Destroy(b_Box);

        GameObject temp_Box;
        temp_Box = MakeBox();
        SetBox(temp_Box);

    }

    public void MoveForward()
    {

        //相□□□□世界坐□□□□系，将地面□□□□□体按一定速度往主平移
        A_Box.transform.Translate(Vector3.left * Speed *Time.deltaTime,
            Space.World);
        B_Box.transform.Translate(Vector3.left * Speed *Time.deltaTime,
            Space.World);

        TimeToUpdate(A_Box);

    }

    void TimeToUpdate(GameObject A_Box)
    {
        //当地面A□□□□□体超出x=0的位置，□□□□□将B□□□□□体□□□□□□□□
        if(A_Box.transform.position.x <= 0)
        {
            Death(B_Box);
        }
    }


}
```

这些关系从设计架构中就能看出来，也可以通过查看源码的类来查看。我们就是通过分而治之的办法使得游戏设计模块化，每个人能在开发中找到自己的角色。

三、Design-Pattern

设计模式在源码的开发中就随处可见了。比如在 Disk_Factory 中，我们是这样设计的。

在前文中已经提到了，在 UNITY 中创建一个新的对象代价是比较大的，尤其是当我们创建的对象在游戏中经过一定操作后会失效，但

又需要不断创建的情况时。一方面，我们需要不断有对象资源可供程序使用，另一方面我们又想避免不断创建新对象的高昂代价。

所以结合在课程中所学，我们有两个列表 used 表示正在使用的游戏对象，free 表示闲置可以被使用的对象。每当 free 不为空时，当我们需要新的对象时，我们直接在 free 中获取，从而避免从外存资源中重新创建对象。当 used 中对象不需要时，并不是直接销毁，而是使其失效而后加入到 free 列表中。

```
4
5 //碟工厂
6 public class Disk_Factory : MonoBehaviour {
7     public GameObject diskPrefab;
8
9     private List<GameObject> used = new List<GameObject>();
.0    private List<GameObject> free = new List<GameObject>();
.1    private GameObject bullet;
.2
.3    void Awake() {
.4        diskPrefab.SetActive (false);
.5    }
.6    // Use this for initialization
.7    void Start () {
.8
.9    }
!0    // Update is called once per frame
!1    void Update () {
!2
!3    }
!4    /*
```

如图：当游戏 Game_Manager 需要获取飞碟时，我们先从 free 中获取，如果没有就只能创建对象。并将对象加入 used 中

```
*/
public GameObject getDisk() {
    GameObject cy;
    Disk_Data ddata;
    int round = 2;
    int selectColor = 0, color = 0;
    if (free.Count == 0) {
        cy = (GameObject) GameObject.Instantiate(diskPrefab);
        cy.AddComponent<Disk_Data> ();
    } else {
        cy = free [0];
        free.RemoveAt (0);
    }
    ddata = cy.GetComponent<Disk_Data> ();
```

```
}
used.Add (cy);
cy.SetActive (true);
cy.name = cy.GetInstanceID ().ToString ()
return cy;
```

如图：Disk_Factory 提供了一个释放飞碟对象的函数，当场景中飞碟无效时，我们将他回收到 free 中，而不是从游戏中销毁。

```
//□□放□□碟□□源，加入到free列表中
public void freeDisk(GameObject cy)
{
    if (cy != null) {
        used.Remove(cy);
        cy.SetActive (false);
        free.Add (cy);
    }
}
```

不得不说这种工厂设计模式在游戏运行中的效果非常显著。如果没有工厂类，飞碟资源将会被不断创建。而工厂类使得我们节约了很多资源

四、单例模式

我们先写了个单例模板，在程序设计过程中，人物使用 Unity 中自带的 find()函数更容易实现单例模式，相比起来的这种方法的劣势在于 find()的代价更大点。

比如说在 player_Get 代码中，获取 Game_Manager 的方法，是通过 GameObject.Find();来找到的。

```
    //sound effects player
    public Sound_Player _SP;
    //player acction manager
    //
    public Player_Move _PM;
    //the get coin nums
    public int Get_Coin_Count;
    //controller
    public Game_Manager _gm;
    //
    public Fade _fade;

    void Start()
    {
        //找到控制器□□象，和□□景□□象
        GameObject a = GameObject.Find("05_GameManager");
        if(a!=null)
        {
            _gm = a.GetComponent<Game_Manager>();
        }
```

在其他类中，若要获得 Game_Manager 控制器也需要如此获得，保证了一个场景中只有一个场景控制器。