中山大學
SUN YAT-SEN UNIVERSITY

《One Color》

设计文档

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	14 级软件工程
成　　员 ：	欧伟杰 14331217
	叶海涛 14331342
	苏锡恺 14331240
	李纬航 14331138

一、技术选型理由

我们团队选择了 Unity3D 作为游戏的开发引擎。

1.1 提供简便的图形操作

从游戏开发的角度而言，Unity3d 则是一个高效的 IDE+代码库。它很好地封装了底层代码，提供许多简便的图形操作，还有商业级的高级功能。并且提供了完全免费的简化版本，以及带有 Asset Store 销售平台，使用资源商店可以节省时间和精力，直接从 Unity 编辑器或 web 浏览器在资源商店中购买。有数千多种现成的免费或可购买的资源和生产工具为游戏开发者提供了便利地编辑器扩展、插件、环境和模型及更多功能的购买。

与另外一个游戏开发引擎 Cocos2d-x 相比，Cocos2d-x 是一个“纯正”的引擎——仅仅只是代码库。虽然可以利用 CocosBuilder 和其他一些工具进行图形化操作，但效率始终不够 Unity3d 高。而且暴露过多的底层代码，对于研究是一件大大的好事，但是对于创作而言，却未必是一件好事。

1.2 多平台支持

Unity3D 游戏引擎支持的平台有:PC, Mac OS, Web, iOS, Android, XBOX360, PS3, Wii。 无可挑剔的跨平台能力，

1.3 代码驱动的开发模式

正如 Unity3D 官网所说，灵活，快速，高端 是它的优势。Unity 是一个现成的解决方案，可以直观的使用和深度定制。代码驱动的开发模式，可以做到快速的游戏开发。

这种模式，可以使我们快速地构建一个原型。对于 U3D 中的 MonoBehaviour 来说，它扮演的，就是如何驱动它的目标对象。 因此，你可以将你的对象的各种能力分配到不同的脚本组件中，然后根据对象的需求来挂接。编辑后立即运行，还能在运行过程中时实编辑，查看效果。

1.4 基于 Mono 的开发脚本

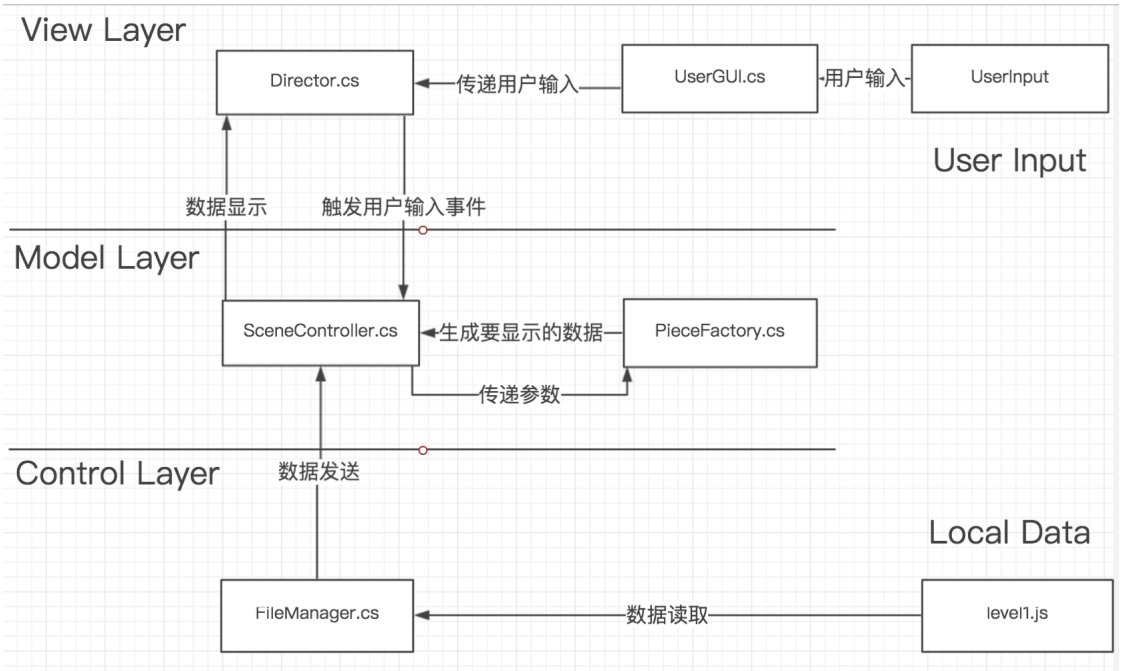
Mono的桥接，使得高效的C++图形引擎与带GC的内存安全语言进行结合。不仅减少了安全隐患，也使得大家编写跨平台代码时更佳容易。 同时，这类语言的反射机制，更适合做编辑器。而比起先前的一些DIY语言和像LUA这样的小巧型语言，Mono使脚本编程可以进行DEBUG，而不单纯的靠PRINT输出。脚本语言在 Unit3D 游戏开发中占据了主角的位置。Unity3D 提供了 三种脚本语言的支持:Javascript、C#、Boo，

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。

二、架构设计

在架构方面选择了 MVC 架构作为开发架构。1，主要游戏操作对象开发所使用的架构：以 mvc 架构为主，整个框架就是一个 mvc 架构； 2，游戏环境开发使用的框架（例如背景音乐，视觉切换，交互设计）：整个框架使用一个平行结构。

2.1 主要游戏操作对象开发架构(整个架构就是一个 mvc 架构)



如架构图所示：

UserGUI.cs 通过 OnGUI()获取用户输入并通过调用 Director getInstance()接口传递用户输入给 Director.cs。

Director.cs 通过 Director getInstance().currentSceneControl 接口传递用户输入给 SceneController.cs。

SceneController.cs 通过 director.currentSceneControl.LoadResources()经由 Director.cs 实现数据显示。创建新的 PieceFactory 对象来进行传递参数让 PieceFactory.cs 生成需要显示的对象。

SceneController.cs 通过调用 stageLevel() 经由 FileManager.cs 通过 loadLevelJson()读取本地数据之后，

因为主要游戏操作对象开发内容多，范围广，需要多模块进行开发，利用 MVC 架构进行开发可以将数据层，视图层和控制层分开进行工作。可以清晰地将开发任务模块化，使得共同开发对接更加容易。

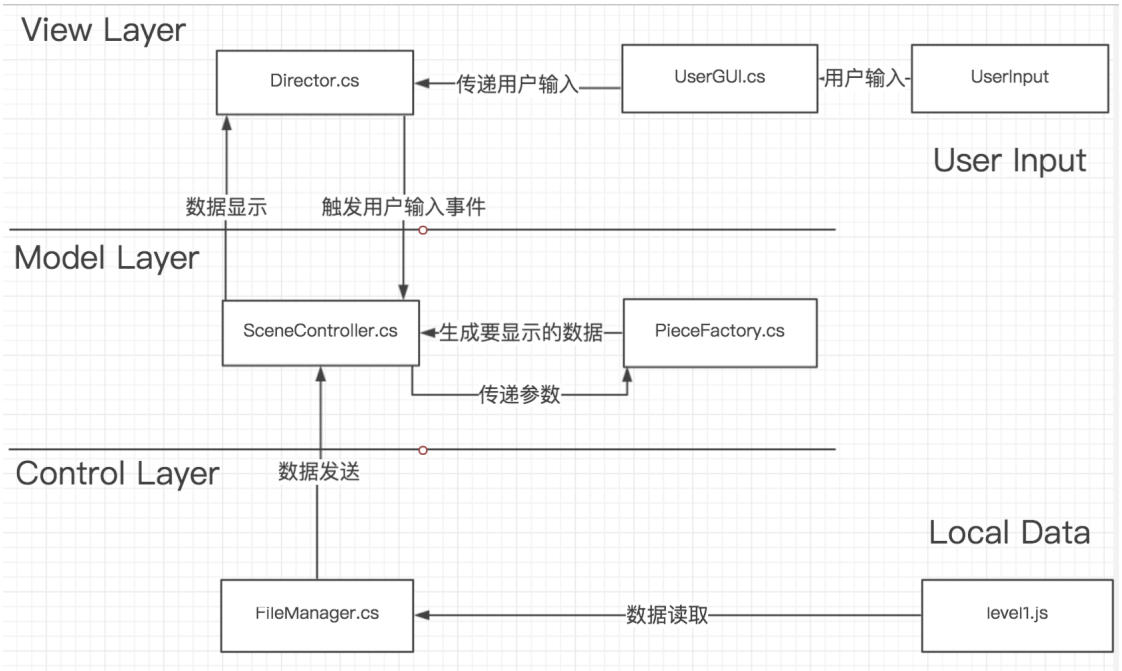
2.2 游戏环境开发架构（例如背景音乐，视觉切换，交互设计）：整个框架使用一个平行结构。

因为游戏环境开发架构（例如背景音乐，视觉切换，交互设计）的开发与游戏主要操作对象独立，且各个方面互相独立，不需要与其他接口对接，直接挂载在 unity3D 的 scene 上即可，因此无需对整个框架使用 mvc 架构，使用一个平行结构进行开发即可。

三、模块划分

3.1 模块划分情况：

游戏主要操作对象模块划分如图：



其他的模块划分有：audio.cs, CameraControl, Singleton，以及直接在 Unity3D Scene 进行的交互设计。

3.2 各个模块的具体内容：

UserGUI.cs 通过按钮的点击来获取用户想要选择的颜色。
Director.cs 提供一个到达场景的接口。
SceneController：

方法名	具体内容
`public void stageLevel(string json)`	对通过FileManager的接口获取的数据进行加工，然后传递给PieceFactory.cs。
`public void selectPiece(Vector3 pos)`	1.将所选中片的数据传递给PieceFactory.cs；2.判断当前是否已经达成胜利条件。
`public void selectColor(Color color)`	提供接口给UserGUI传递用户所选择颜色
`public void LoadResources()`	调用FileManager的接口，实现对json文件的读取。

PieceFactory:

方法名	具体内容
`private IEnumerator delay`	延时变色功能。
`private void paint`	用特定的颜色列表给piece上色。
`public void reset()`	初始化游戏变量。
`public void LoadResources()`	对SceneController传来的参数进行处理之后，根据参数生成特定size,特定颜色的cube。
`public List<GameObject> GetList(GameObject obj)`	返回所选中piece所在列表
`public int SetPieceColor void GetSameColorPiece`	实现对选中的piece及其相邻相同颜色的piece变换成所选择颜色的功能。
`public bool isOneColor()`	判断所有piece的颜色是否一致。

FlieManager:对 json 文件的读取。
audio.cs：增加播放背景音乐功能，实现了播放、暂停、结束音乐以及控制音量的功能。
CameraControl：实现了滚轮控制摄像头缩放。
Singleton：实现了单例模式的操作。

四、项目中使用的软件技术

4.1 Object-Oriented Programming

piecefactory.cs利用了面向对象的编程思想，对象为组成cube的所有片，存储了所有片的颜色；

```
private static List<GameObject> pieceList = new List<GameObject>();
private static List<GameObject> pieceList_left = new List<GameObject>();
private static List<GameObject> pieceList_right = new List<GameObject>();
private static List<GameObject> pieceList_back = new List<GameObject>();
private static List<GameObject> pieceList_top = new List<GameObject>();
private static List<GameObject> pieceList_bottom = new List<GameObject>();
private static List<int> colorList = new List<int> ();
private static List<int> color_left = new List<int>();
private static List<int> color_right = new List<int>();
private static List<int> color_top = new List<int>();
private static List<int> color_bottom = new List<int>();
private static List<int> color_back = new List<int>();
```

提供了对它们上色的方法；

```
private void paint(List<GameObject> target_list, List<int> color, int i, int j) {
    switch (color[i*size+j])
    {
    case 0:
        target_list[i*size+j].GetComponent<Renderer>().material.color = color_1;
        break;
    case 1:
        target_list[i*size+j].GetComponent<Renderer>().material.color = color_2;
        break;
    case 2:
        target_list[i*size+j].GetComponent<Renderer>().material.color = color_3;
        break;
    case 3:
        target_list[i*size+j].GetComponent<Renderer>().material.color = color_4;
        break;
    }
}
```

提供了初始化它们颜色的方法；

```
public void reset() {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            paint(pieceList, colorList, i, j);
            paint(pieceList_top, color_top, i, j);
            paint(pieceList_bottom, color_bottom, i, j);
            paint(pieceList_back, color_back, i, j);
            paint(pieceList_left, color_left, i, j);
            paint(pieceList_right, color_right, i, j);
        }
    }
}
```

提供了对选中的piece及其相邻相同颜色的piece变换成所选择颜色的功能；

```
//改变与选中片颜色相同且互相连接的片的颜色
public int SetPieceColor(GameObject slt_piece, Color slt_color)
{
    List<GameObject> new_list = GetList (slt_piece);
    Color old_color = slt_piece.GetComponent<Renderer>().material.color;
    if (old_color == slt_color) return 0;
    int piece_id = 0;
    for (int i = 0; i < pieceList.Count; i++)
    {
        if (slt_piece.GetInstanceID() == new_list[i].GetInstanceID())
        {
            piece_id = i;
        }
    }
    new_list[piece_id].GetComponent<Renderer>().material.color = slt_color;
    count = 0;
    GetSameColorPiece(new_list, piece_id, old_color, slt_color);
    return 1;
}
```

```
//改变一个片周围颜色相同的片的颜色
void GetSameColorPiece(List<GameObject> new_list, int id, Color old_color, Color new_color)
{
    List<int> id_list = new List<int>();
    count++;
    if ((id % size) > 0)
    {
        if (new_list[id - 1].GetComponent<Renderer>().material.color == old_color)
            id_list.Add(id - 1);
    }
    if ((id % size + 1) < size)
    {
        if (new_list[id + 1].GetComponent<Renderer>().material.color == old_color)
            id_list.Add(id + 1);
    }
    if ((id / size) > 0)
    {
        if (new_list[id - size].GetComponent<Renderer>().material.color == old_color)
            id_list.Add(id - size);
    }
    if ((id / size + 1) < size)
    {
        if (new_list[id + size].GetComponent<Renderer>().material.color == old_color)
            id_list.Add(id + size);
    }
    if (id_list.Count > 0)
    {
        for (int i = 0; i < id_list.Count; i++)
        {
            new_list[id_list[i]].GetComponent<Renderer>().material.color = new_color;
            if (new_list [id_list [i]].transform.rotation.z < 90) {
                new_list [id_list [i]].transform.Rotate (0, 0, Time.deltaTime);
            }
        }
        for (int i = 0; i < id_list.Count; i++)
        {
            StartCoroutine(delay(new_list ,id_list[i], old_color, new_color));
        }
    }
    else
```

提供了判断颜色是否一致的方法。

```
//判断颜色是否一致
public bool isOneColor()
{
    Color color = pieceList[0].GetComponent<Renderer>().material.color;
    for (int i = 0; i < pieceList.Count; i++)
    {
        if (pieceList[i].GetComponent<Renderer>().material.color != color)
        {
            return false;
        }
        if (pieceList_left[i].GetComponent<Renderer>().material.color != color)
        {
            return false;
        }
        if (pieceList_right[i].GetComponent<Renderer>().material.color != color)
        {
            return false;
        }
        if (pieceList_top[i].GetComponent<Renderer>().material.color != color)
        {
            return false;
        }
        if (pieceList_bottom[i].GetComponent<Renderer>().material.color != color)
        {
            return false;
        }
    }
    return true;
}
```

4.2单例模式

单例模式的目的是使得类的一个对象成为系统中的唯一实例。要实现这一点，可以从客户端对其进行实例化开始。因此需要用一种只允许生成对象类的唯一实例的机制，“阻止”所有想要生成对象的访问。使用工厂方法来限制实例化过程。单例模式可以保证这个类没有其他实例被创建，并且它可以提供一个访问该实例的方法。

在 oneColor 项目里 Director 与 PieceFactory 这两个类智能有一个实例，通过单例模式可以轻松地完成只允许生成对象类的唯一实例并且可以轻松访问。

实现：

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

public class Singleton<T> : MonoBehaviour where T : MonoBehaviour
{

    protected static T instance;

    public static T Instance
    {
        get
        {
            if (instance == null)
            {
                instance = (T)FindObjectOfType(typeof(T));
                if (instance == null)
                {
                    Debug.LogError("An instance of " + typeof(T)
                        + " is needed in the scene, but there is none.");
                }
            }
            return instance;
        }
    }
}
```

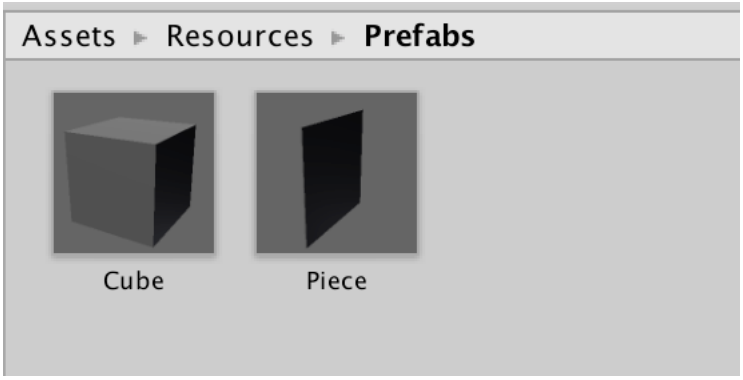
应用：

```
void OnGUI() {
    if (GUI.Button(new Rect(Screen.width - 280, Screen.height - 60, 60, 40), "Reset"))
    {
        Debug.Log ("reset");
        PieceFactory pf = Singleton<PieceFactory>.Instance;
        pf.reset();
        used_step = 0;
        maintext.text = "";
    }
}
//选择改变颜色的片
public void selectPiece(Vector3 pos)
{
    Ray ray = Camera.main.ScreenPointToRay(pos);

    RaycastHit hit;
    if (Physics.Raycast(ray, out hit) && hit.collider.gameObject.tag == "Piece" && is_slt_color)
    {
        PieceFactory pf = Singleton<PieceFactory>.Instance;
        used_step += pf.SetPieceColor(hit.collider.gameObject, slt_color);
        if (pf.isOneColor())
        {
            if (used_step > max_step)
                maintext.text = "Pass But Overstep";
            else
                maintext.text = "Perfect!";

        }
    }
}
```

4.3工厂模式

工厂模式是一种实现了“工厂”概念的面向对象设计模式。就像其他创建型模式一样，它也是处理在不指定对象具体类型的情况下创建对象的问题。工厂方法模式的实质是“定义一个创建对象的接口，但让实现这个接口的类来决定实例化哪个类。工厂方法让类的实例化推迟到子类中进行。”

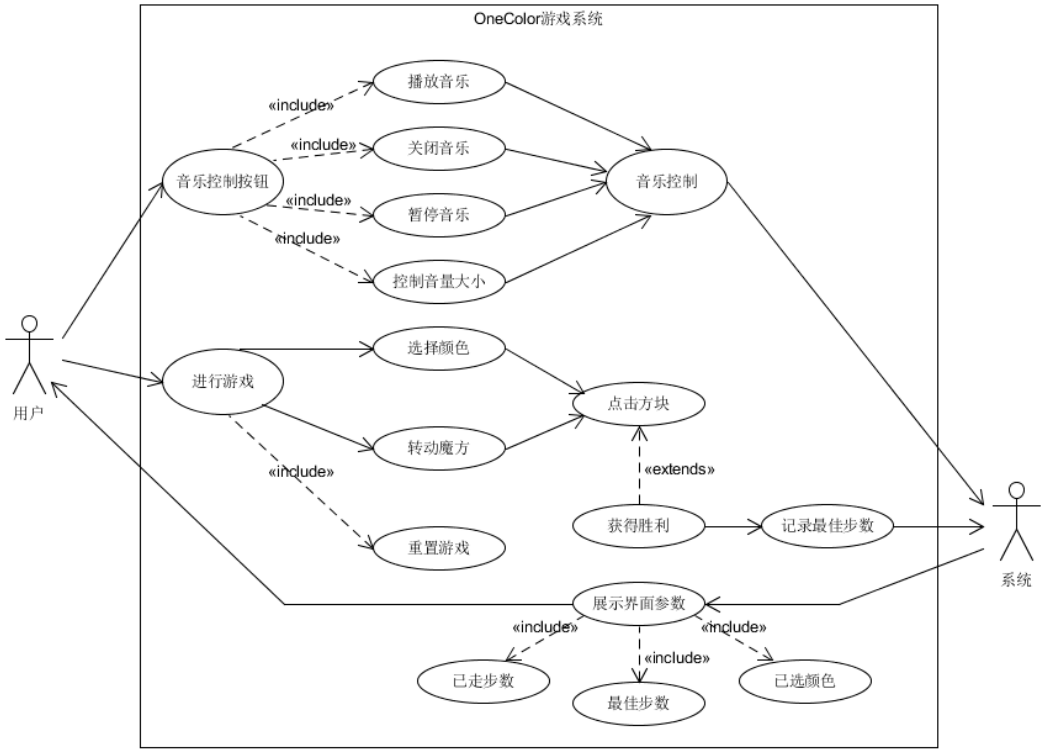
在 oneColor 项目中需要生成许多不定数量不定属性的 piece 对象，因此采用工厂模式，首先在 Asset/Resource 下创建 prefabs，创建一个对象。



然后再 PieceFactory 里利用这个对象创建多个实例化的类。

```
for (int i = 0; i < size; i++)
{
    for (int j = 0; j < size; j++)
    {
        Vector3 pos = new Vector3(start + 1.0f * j, start + 1.0f * i, 0);
        pieceList.Add(Instantiate(Resources.Load("Prefabs/Piece"), pos, Quaternion.identity) as GameObject);
        paint (pieceList, color, i, j);
    }
}
```

用例图:


OneColor游戏系统
播放音乐
关闭音乐
暂停音乐
控制音量大小
音乐控制按钮
音乐控制
«include»
«include»
«include»
«include»
选择颜色
进行游戏
点击方块
转动魔方
重置游戏
«include»
«extends»
获得胜利
记录最佳步数
展示界面参数
«include»
«include»
«include»
已走步数
最佳步数
已选颜色
用户
系统