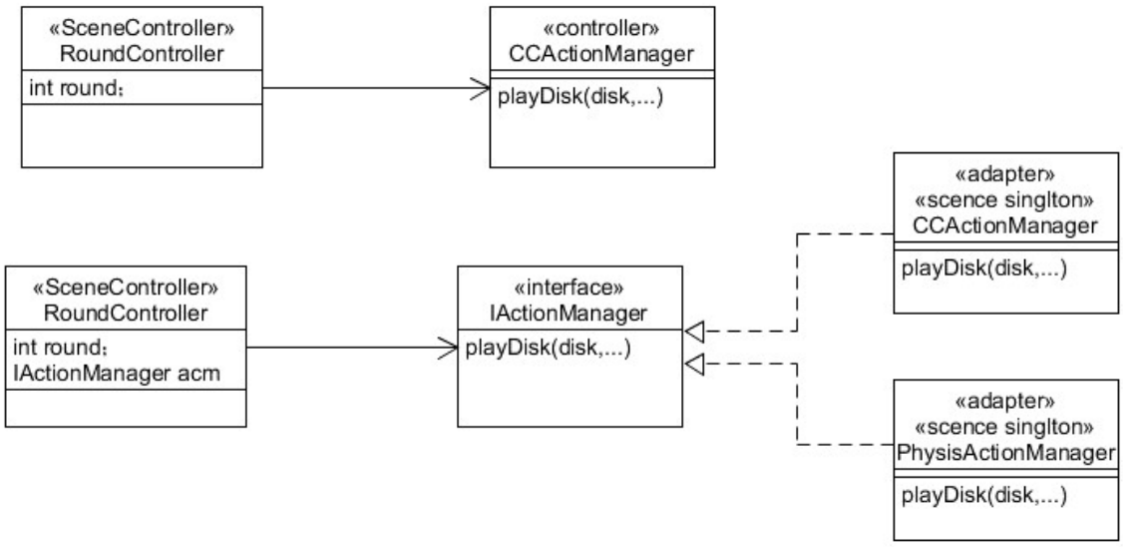
改进打飞碟游戏（可选）：

□游戏内容要求：
□按下面adapter模式设计图修改飞碟游戏
□使它同时支持物理运动与运动学（变换）运动



对象回收

在游戏中，对象的新建和销毁的开销是巨大的，是不可忽视的。对于频繁出现的游戏对象，我们应该使用**对象池**技术缓存，从而降低对象的新建和销毁开销。在本游戏中，飞碟是频繁出现的游戏对象，我们使用**带缓存的工厂模式**管理不同飞碟的生产和回收。对于该飞碟工厂，我们使用**单例模式**。

在 `UFOFactory` 中，我们使用两个列表来分别维护**正在使用**和**未被使用**的飞碟对象。我们对外提供了 `Get` 获取飞碟对象和 `Put` 回收飞碟对象的基本接口。由于使用了单例模式，我们还对外提供了 `GetInstance` 的静态方法。

```
public class UFOFactory
{
    // 定义飞碟颜色。
    public enum Color
    {
        Red,
        Green,
        Blue
    }

    // 单例。
    private static UFOFactory factory;
    // 维护正在使用的飞碟对象。
    private List<GameObject> inUsed = new List<GameObject>();
    // 维护未被使用的飞碟对象。
    private List<GameObject> notUsed = new List<GameObject>();
    // 空闲飞碟对象的空间位置。
    private readonly Vector3 invisible = new Vector3(0, -100, 0);

    // 使用单例模式。
    public static UFOFactory GetInstance()
```

```
    {
        return factory ?? (factory = new UFOFactory());
    }

    // 获取特定颜色的飞碟。
    public GameObject Get(Color color)
    {
        GameObject ufo;
        if (notUsed.Count == 0)
        {
            ufo = Object.Instantiate(Resources.Load<GameObject>
("Prefabs/UFO"), invisible, Quaternion.identity);
            ufo.AddComponent<UFOModel>();
        }
        else
        {
            ufo = notUsed[0];
            notUsed.RemoveAt(0);
        }

        // 设置 Material 属性（颜色）。
        Material material = Object.Instantiate(Resources.Load<Material>
("Materials/" + color.ToString("G")));
        ufo.GetComponent<MeshRenderer>().material = material;

        // 添加对象至 inUsed 列表。
        inUsed.Add(ufo);
        return ufo;
    }

    // 回收飞碟对象。
    public void Put(GameObject ufo)
    {
        // 设置飞碟对象的空间位置和刚体属性。
        var rigidbody = ufo.GetComponent<Rigidbody>();
        // 以下两行代码很关键！我们需要设置对象速度为零！
        rigidbody.velocity = Vector3.zero;
        rigidbody.angularVelocity = Vector3.zero;
        rigidbody.useGravity = false;
        ufo.transform.position = invisible;
        // 维护 inUsed 和 notUsed 列表。
        inUsed.Remove(ufo);
        notUsed.Add(ufo);
    }
}
```

在 `UFOFactory` 中，我们使用两个列表来分别维护**正在使用**和**未被使用**的飞碟对象。我们对外提供了 `Get` 获取飞碟对象和 `Put` 回收飞碟对象的基本接口。由于使用了单例模式，我们还对外提供了 `GetInstance` 的静态方法。

我们在场景控制器 `GameController` 的 `Update` 函数，对用户的输入进行监听。在特定游戏状态下（避免在本次 Trial 飞碟未销毁的情况下，进入下一 Trial），当用户按下空格键时，我们触发飞碟的发射函数 `ruler.GetUFOs` 。

我们使用 `Ruler` 管理飞碟特性（颜色、分值、同时出现的数目）与关卡进度的关系，其对外提供了 `GetUFOs` 方法，用于发射飞碟。

```
public class UFOModel : MonoBehaviour
{
    // 记录当前飞碟的分数。
    public int score;
    // 记录飞碟在左边的初始位置。
    public static Vector3 startPosition = new Vector3(-3, 2, -15);
    // 记录飞碟在左边的初始速度。
    public static Vector3 startSpeed = new Vector3(3, 11, 8);
    // 记录飞碟的初始缩放比例。
    public static Vector3 localScale = new Vector3(1, 0.08f, 1);
    // 表示飞碟的位置（左边、右边）。
    private int leftOrRight;

    // 获取实际初速度。
    public Vector3 GetSpeed()
    {
        Vector3 v = startSpeed;
        v.x *= leftOrRight;
        return v;
    }

    // 设置实际初位置。
    public void SetSide(int lr, float dy)
    {
        Vector3 v = startPosition;
        v.x *= lr;
        v.y += dy;
        transform.position = v;
        leftOrRight = lr;
    }

    // 设置实际缩放比例。
    public void SetLocalScale(float x, float y, float z)
    {
        Vector3 lc = localScale;
        lc.x *= x;
        lc.y *= y;
        lc.z *= z;
        transform.localScale = lc;
    }
}

if (model.game == GameState.Running)
{
    if (model.scene == SceneState.Waiting && Input.GetKeyDown("space"))
    {
        model.scene = SceneState.Shooting;
        model.NextTrial();
        // 添加此判断的原因：对于最后一次按下空格键，若玩家满足胜利条件，则不发射飞碟。
        if (model.game == GameState.Win)
        {
            return;
        }
        UFOs.AddRange(ruler.GetUFOs());
    }
}

// 在用户按下空格键后被触发，发射飞碟。
```

```
public List<GameObject> GetUFOs()
{
    List<GameObject> ufos = new List<GameObject>();
    // 随机生成飞碟颜色。
    var index = random.Next(colors.Length);
    var color = (UFOFactory.Color)colors.GetValue(index);
    // 获取当前 Round 下的飞碟产生数。
    var count = GetUFOCount();
    for (int i = 0; i < count; ++i)
    {
        // 调用工厂方法，获取指定颜色的飞碟对象。
        var ufo = UFOFactory.GetInstance().Get(color);
        // 设置飞碟对象的分数。
        var model = ufo.GetComponent<UFOModel>();
        model.score = score[index] * (currentRound + 1);
        // 设置飞碟对象的缩放比例。
        model.SetLocalScale(scale[index], 1, scale[index]);
        // 随机设置飞碟的初始位置（左边、右边）。
        var leftOrRight = (random.Next() & 2) - 1; // 随机生成 1 或 -1 。
        model.SetSide(leftOrRight, i);
        // 设置飞碟对象的刚体属性，以及初始受力方向。
        var rigidbody = ufo.GetComponent<Rigidbody>();
        rigidbody.AddForce(0.2f * speed[index] * model.GetSpeed(),
ForceMode.Impulse);
        rigidbody.useGravity = true;
        ufos.Add(ufo);
    }
    return ufos;
}
```

在游戏中，玩家通过鼠标点击飞碟，从而得分。这当中涉及到一个**点击判断**的问题。在 Unity 中，我们可以调用 `ScreenPointToRay` 方法，**构造由摄像头和屏幕点击点确定的射线，与射线碰撞的游戏对象即为玩家点击的对象。**

我们在 Assets Store 下载 "Particle Dissolve Shader by Moonflower Carnivore" 素材库，引入其中的爆炸效果 `Prefab` 。由于，我们期待爆炸效果能够持续一段时间，然后停止，因此，我们使用**协程**对爆炸效果进行管理。

我们在 `OnHitUFO` 函数中，创建协程。在该协程中，我们实例化预制件，并赋予其被点击的 UFO 的位置，随后调用 `WaitForSeconds` 方法，并让出协程。在重新获得执行机会后，我们调用 `Destroy` 方法，销毁爆炸效果对象。

```
// 光标拾取单个游戏对象。
// 构建射线。
Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
// 当射线与飞碟碰撞时，即说明我们想用鼠标点击此飞碟。
if (Physics.Raycast(ray, out RaycastHit hit) && hit.collider.gameObject.tag
== "UFO")
{
    OnHitUFO(hit.collider.gameObject);
}

// 在用户成功点击飞碟后被触发。
private void OnHitUFO(GameObject ufo)
{
    // 增加分数。
    model.AddScore(ufo.GetComponent<UFOModel>().score);
```

```
    // 创建协程，用于控制飞碟爆炸效果的延续时间。
    StartCoroutine("DestroyExplosion", ufo);
}

// 该协程用于控制飞碟爆炸效果。
private IEnumerator DestroyExplosion(GameObject ufo)
{
    // 实例化预制。
    GameObject explosion = Instantiate(explosionPrefab);
    // 设置爆炸效果的位置。
    explosion.transform.position = ufo.transform.position;
    // 回收飞碟对象。
    DestroyUFO(ufo);
    // 爆炸效果持续 1.2 秒。
    yield return new WaitForSeconds(1.2f);
    // 销毁爆炸效果对象。
    Destroy(explosion);
}
```

当玩家**点击飞碟**或**错失飞碟**时，场景控制器 `GameController` 会调用裁判类的 `AddScore` 和 `SubScore` 方法，更新玩家分数。

```
// 增加玩家分数。
public void AddScore(int score)
{
    this.score += score;
    // 通知场景控制器需要更新游戏画面。
    onRefresh.Invoke(this, EventArgs.Empty);
}

// 扣除玩家分数。
public void SubScore()
{
    this.score -= (currentRound + 1) * 10;
    // 检测玩家是否失败。
    if (score < 0)
    {
        Reset(GameState.Lose);
    }
    onRefresh.Invoke(this, EventArgs.Empty);
}

// 通知场景控制器更新游戏画面。
public EventHandler onRefresh;
// 通知场景控制器更新 Ruler 。
public EventHandler onEnterNextRound;
```

物理引擎是一个软件组件，将游戏世界对象赋予现实世界物理属性（重量、形状等），并抽象为**刚体**模型，使得游戏物体在力的作用下，仿真现实世界的运动及其之间的碰撞过程。

我们已使用了**动力学模型**实现飞碟的运动。现在，我们使用 **Adapter** 设计模式，为游戏添加**运动学模型**的支持，实现二者模型的自由切换。

动力学模型：

```
public class PhysicActionManager : IActionManager
{
    public void SetAction(GameObject ufo)
    {
        var model = ufo.GetComponent<UFOModel>();
        var rigidbody = ufo.GetComponent<Rigidbody>();
        // 对物体添加 Impulse 力。
        rigidbody.AddForce(0.2f * model.GetSpeed(), ForceMode.Impulse);
        rigidbody.useGravity = true;
    }
}
```

运动学模型：

```
public class CCActionManager : IActionManager
{
    private class CCAction : MonoBehaviour
    {
        void Update()
        {
            // 关键！当飞碟被回收时，销毁该运动学模型。
            if (transform.position == UFOFactory.invisible)
            {
                Destroy(this);
            }
            var model = gameObject.GetComponent<UFOModel>();
            transform.position = Vector3.MoveTowards(transform.position, new
Vector3(-3f + model.GetID() * 2f, 10f, -2f), 5 * Time.deltaTime);
        }
    }

    public void SetAction(GameObject ufo)
    {
        // 由于预设使用了 Rigidbody ，故此处取消重力设置。
        ufo.GetComponent<Rigidbody>().useGravity = false;
        // 添加运动学（转换）运动。
        ufo.AddComponent<CCAction>();
    }
}
```

Adapter 接口

首先，我们定义 Adapter 接口 `IActionManager` ，其含有 `SetAction` 方法，用于设置游戏对象的运动学模型。

```
public interface IActionManager
{
    void SetAction(GameObject ufo);
}
```

最终呈现效果：

参考：https://github.com/Jiahonzheng/Unity-3D-Learning/tree/master/HW5