# 3d游戏作业六

# 3d游戏设计读书笔记六

## 一、改进飞碟（Hit UFO）游戏：

游戏内容要求：

按 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

1. \*\*更改原 UFO\_action 类 为 UFO\_Kinematics\_action 类\*\*

将原来的类改为 运动学action ，并创建一个新的动力学action。以下是修改与增加部分，其余不变：

```java

public void SetSpeed(int speed)

{

this.speed = speed;

}

public void SetRunning(bool b)

{

this.running = b;

}

public GameObject getPlayer()

{

return this.player;

}

public void setPlayer(GameObject g)

{

this.player = g;

}

// Start is called before the first frame update

public void Start()

{

\_director = Director.getInstance();

start = new Vector3(Random.Range(-20f,20f), Random.Range(-20f, 20f), 0);

if (start.x < 10 && start.x > -10)

start.x \*= 4;

if (start.y < 10 && start.y > -10)

start.y \*= 4;

end = new Vector3(-start.x, -start.y, 0);

player.transform.position = start;

setColor();

Rigidbody rigit = player.GetComponent<Rigidbody>();

if (rigit != null)

{

Destroy(rigit);

}

}

```

ps：当我们修改脚本名称时，要对应修改其类的名称，否则会报错。

2. \*\*添加UFO\_Dinamics\_action 类\*\*

仿照 UFO\_Kinematics类 的设计。

注意： 我们这里在 Start函数 中为飞碟添加了 刚体组件 ，并且为其设置了初速度。所以在这个action被创建的时候，飞碟就开始飞了。

(1) Start()

```java

public void Start()

{

\_director = Director.getInstance();

if (player.GetComponent<Rigidbody>() == null)

player.AddComponent<Rigidbody>();

start = new Vector3(Random.Range(-20f, 20f), Random.Range(-20f, 20f), 0);

if (start.x < 10 && start.x > -10)

start.x \*= 4;

if (start.y < 10 && start.y > -10)

start.y \*= 4;

end = new Vector3(0, 0, 20);

player.transform.position = start;

setColor();

Rigidbody rigit = player.GetComponent<Rigidbody>();

rigit.velocity = (end - start) \* speed \* Random.Range(0.0f, 0.10f);

rigit.useGravity = false;

}

```

(2) Update()

start函数 中已经为飞碟设置了初速度，所以在就不用像 Kinematics\_action 那样每一帧都更新飞碟位置。我们要做的是检查飞碟是否飞出屏幕外。

因为飞碟的位置是随机的，且它们是刚体，很多时候会产生碰撞而改变运动轨迹。所以在 update函数中记录了其调用次数。若是调用次数 超过300 ，则将该飞碟推入 not\_hit 函数。

```java

public void Update()

{

framecount++;

if(framecount>300)

this.\_director.currentController.\_UFOfactory.not\_hit(this.player);

Rigidbody rigit = player.GetComponent<Rigidbody>();

if (running == false)

{

rigit.velocity = Vector3.zero;

framecount = 0;

}

if (player.transform.position.x < -100 || player.transform.position.x > 100 || player.transform.position.x < -100 || player.transform.position.x > 100 || player.transform.position.x < -100 || player.transform.position.x > 100)

{

rigit.velocity = Vector3.zero;

this.\_director.currentController.\_UFOfactory.not\_hit(this.player);

}

}

```

其它函数与UFO\_Kinematics\_action类相同。

3. \*\* UFO\_action 接口（Adapter模式）\*\*

由于我们拥有两个 Action类 ，所以按照 \*\*热拔插\*\* 的设计思想，我们要设计一个接口来包装这两个 Action类。我们通过抽象 UFOFactory 中两个类共有的方法，即可以抽象出该接口。

```java

public interface UFO\_action

{

void SetSpeed(int speed);

void Start();

void SetRunning(bool b);

GameObject getPlayer();

void setPlayer(GameObject g);

void Update();

}

```

在两个Action类里面继承这个接口，并实现这些函数。(这里的函数实现是一致的)

```java

public void SetSpeed(int speed)

{

this.speed = speed;

}

public void SetRunning(bool b)

{

this.running = b;

}

public GameObject getPlayer()

{

return this.player;

}

public void setPlayer(GameObject g)

{

this.player = g;

}

```

接着再在 UFOFactory类 里面更新函数实现，将从前指定 action 编程使用接口来调用函数。

```java

public class UFOFactory : MonoBehaviour

{

public int choose = 0; //0 for Kinematics\_actions 1 for Dynamic\_actions

public List<GameObject> used;

public List<GameObject> not\_used;

public List<UFO\_Kinematics\_action> Kinematics\_actions;

public List<UFO\_Dynamics\_action> Dynamics\_actions;

public List<UFO\_action> UFO\_actions;

public int round = 0;

public int score = 0;

private void Start()

{

used = new List<GameObject>();

not\_used = new List<GameObject>();

Kinematics\_actions = new List<UFO\_Kinematics\_action>();

Dynamics\_actions = new List<UFO\_Dynamics\_action>();

UFO\_actions = new List<UFO\_action>();

for (int i = 0; i < 10; i++)

{

not\_used.Add(Object.Instantiate(Resources.Load("Prefabs/UFO", typeof(GameObject)), new Vector3(0, -20, 0), Quaternion.identity, null) as GameObject);

Dynamics\_actions.Add(ScriptableObject.CreateInstance<UFO\_Dynamics\_action>());

Kinematics\_actions.Add(ScriptableObject.CreateInstance<UFO\_Kinematics\_action>());

}

if (choose == 1)

{

for (int i = 0; i < 10; i++)

{

UFO\_actions.Add(Dynamics\_actions[i]);

}

}

else

{

for (int i = 0; i < 10; i++)

{

UFO\_actions.Add(Kinematics\_actions[i]);

}

}

for(int i = 0; i < 10; i++)

{

UFO\_actions[i].setPlayer(not\_used[i]);

UFO\_actions[i].Start();

}

}

private void Update()

{

}

private void FixedUpdate()

{

if (round <= 10)

{

for (int i = 0; i < 10; i++)

UFO\_actions[i].Update();

if (not\_used.Count == 10)

{

round += 1;

if (round <= 10)

get\_ready(round);

}

}

}

public void hitted(GameObject g)

{

if (round <= 10)

{

if (g.gameObject.GetComponent<MeshRenderer>().material.color == Color.red)

score += 3;

else if (g.gameObject.GetComponent<MeshRenderer>().material.color == Color.yellow)

score += 2;

else if (g.gameObject.GetComponent<MeshRenderer>().material.color == Color.blue)

score += 1;

}

this.used.Remove(g);

g.transform.position = new Vector3(0, -20, 0);

for(int i = 0; i < 10; i++)

{

if (UFO\_actions[i].getPlayer() == g)

{

UFO\_actions[i].SetRunning(false);

Rigidbody rigit = UFO\_actions[i].getPlayer().GetComponent<Rigidbody>();

if (rigit != null)

{

rigit.velocity = Vector3.zero;

}

}

}

this.not\_used.Add(g);

}

public void not\_hit(GameObject g)

{

this.used.Remove(g);

g.transform.position = new Vector3(0, -20, 0);

for (int i = 0; i < 10; i++)

{

if (UFO\_actions[i].getPlayer() == g)

{

UFO\_actions[i].SetRunning(false);

Rigidbody rigit = UFO\_actions[i].getPlayer().GetComponent<Rigidbody>();

if (rigit != null)

{

rigit.velocity = Vector3.zero;

}

}

}

this.not\_used.Add(g);

}

public void get\_ready(int round)

{

for(int i = 0; i < 10; i++)

{

used.Add(not\_used[0]);

not\_used.Remove(not\_used[0]);

UFO\_actions[i].SetSpeed(round + 2);

UFO\_actions[i].Start();

UFO\_actions[i].SetRunning(true);

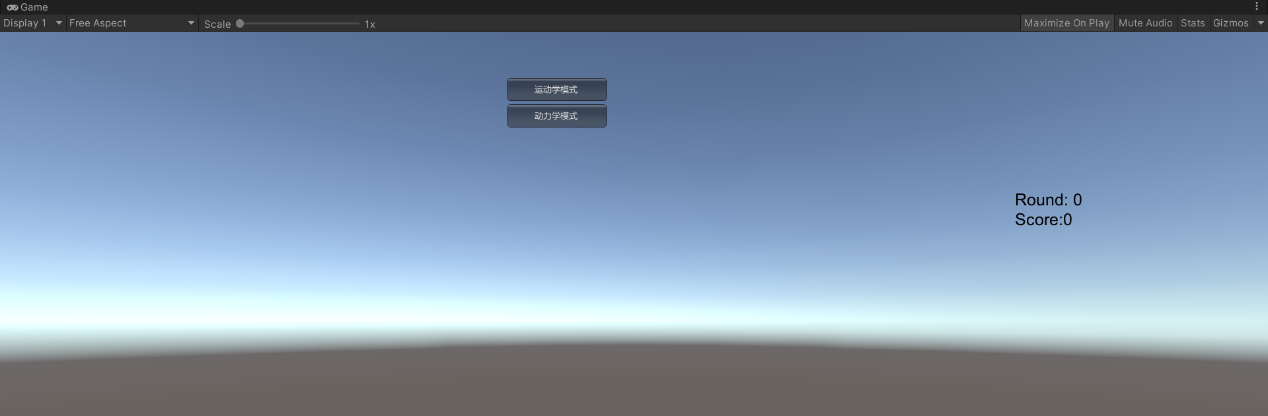
}

}

}

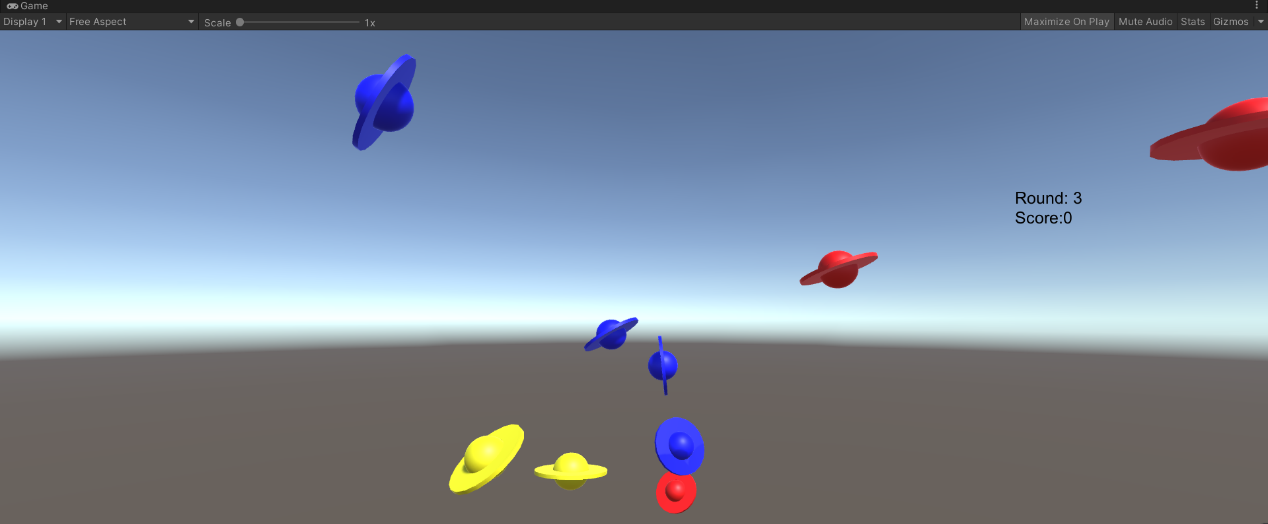
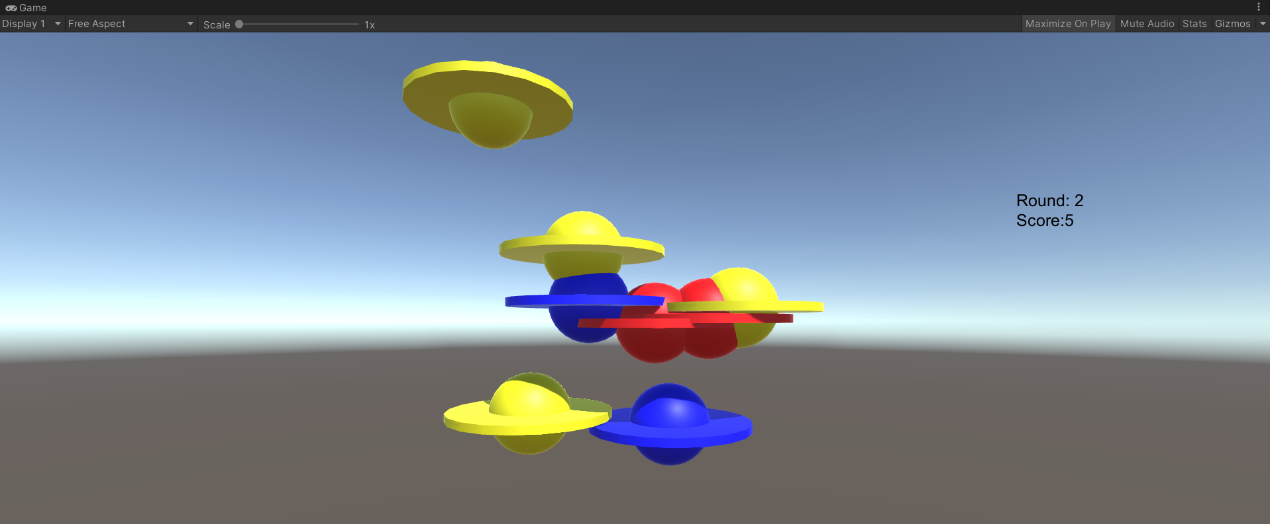
```

4. \*\*游戏展示\*\*



动力学模式：

运动学模式：



## 二、打靶游戏（可选作业）：

游戏内容要求：

靶对象为 5 环，按环计分；

箭对象，射中后要插在靶上

增强要求：射中后，箭对象产生颤抖效果，到下一次射击 或 1秒以后

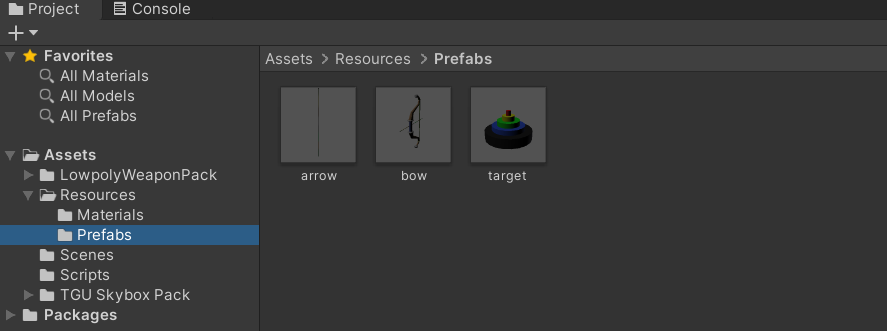
游戏仅一轮，无限 trials；

增强要求：添加一个风向和强度标志，提高难度

(基本参考[师兄博客](https://blog.csdn.net/C486C/article/details/80058316?utm\_medium=distribute.pc\_aggpage\_search\_result.none-task-blog-2~all~first\_rank\_v2~rank\_v25-5-80058316.nonecase&utm\_term=unity%20%E6%89%93%E9%9D%B6&spm=1000.2123.3001.4430))

### 1. 预置

弓箭、弓和靶。



（1）对于弓箭来说，其由箭头和剑身组成，在空对象arrow下创建一个柱体Cylinder和球Sphere构成箭身和箭头。给空对象arrow加上刚体Rigidbody并且勾选Is Kinematic，即开始时候为运动学刚体；给箭身加碰撞器，箭头也加上加碰撞器并且勾选Is Trigger，同时挂载检测碰撞的脚本。

（2）靶上面要有5环，故在一个空对象target下面创建了5个Cylinder子对象，各自带上Mesh Collider网格碰撞器，并且对Mesh Collider的Convex选项打勾，即为凸的网格，这样才能跟其他碰撞器产生碰撞作用。而且挂载了CollisionDetection和RingData脚本。RingData脚本上只有一个分数属性，代表了这一环的分数。然后将target移入Prefabs设为预置。

### 弓箭移动

这部分实现了当用户按下键盘的WSAD或上下左右键时，弓箭移动，并且保持相机与弓箭的相对位置不变，实现相机跟随弓箭的效果，并且弓箭移动范围不是无限的，所以限定了一个移动范围

#### 1. UserGUI

在UserGUI的Update中，当游戏进行时每一帧获取是否按下方向键，使用 \*\*Input.GetAxis\*\* 得到按下方向键后的虚拟轴中的值，然后通过接口去调用场景控制器中移动弓的方法。

```java

void Update()

{

if(game\_start && !action.GetGameover())

{

if (Input.GetButtonDown("Fire1"))

{

action.Shoot();

}

//获取方向键的偏移量

float translationY = Input.GetAxis("Vertical");

float translationX = Input.GetAxis("Horizontal");

//移动弓箭

action.MoveBow(translationX, translationY);

}

}

```

#### 2. FirstSceneController

继承了IUserAction接口，并实现其方法，其中MoveBow方法实现了根据获取的虚拟轴的值移动弓。

```java

public void MoveBow(float offsetX, float offsetY)

{

//游戏未开始时候不允许移动弓

if (game\_over || !game\_start)

{

return;

}

//弓是否超出限定的移动范围

if (bow.transform.position.x > 5)

{

bow.transform.position = new Vector3(5, bow.transform.position.y, bow.transform.position.z);

return;

}

else if(bow.transform.position.x < -5)

{

bow.transform.position = new Vector3(-5, bow.transform.position.y, bow.transform.position.z);

return;

}

else if (bow.transform.position.y < -3)

{

bow.transform.position = new Vector3(bow.transform.position.x, -3, bow.transform.position.z);

return;

}

else if (bow.transform.position.y > 5)

{

bow.transform.position = new Vector3(bow.transform.position.x, 5, bow.transform.position.z);

return;

}

//弓箭移动

offsetY \*= Time.deltaTime;

offsetX \*= Time.deltaTime;

bow.transform.Translate(0, -offsetX, 0);

bow.transform.Translate(0, 0, -offsetY);

}

```

#### 3. CameraFlow

挂载在主相机上，根据初始的时候与弓的偏移量，在弓的位置变化的时候，保持偏移量不变，从而产生跟随效果。这里的bow在场景控制器中设置。

```java

public class CameraFlow : MonoBehaviour

{

public GameObject bow; //跟随的物体

public float smothing = 5f; //相机跟随的速度

Vector3 offset; //相机与物体相对偏移位置

void Start()

{

offset = transform.position - bow.transform.position;

}

void FixedUpdate()

{

Vector3 target = bow.transform.position + offset;

//摄像机自身位置到目标位置平滑过渡

transform.position = Vector3.Lerp(transform.position, target, smothing \* Time.deltaTime);

}

}

```

### 弓箭飞行

这部分实现了用户按下鼠标左键，然后箭实现飞行动作

#### 1. ArrowFlyAction

在Start中为箭的Rigidbody设置一个初始力，让箭射出，在FixedUpdate中给箭一个持续的风力。(在开始时候给箭的预制体添加Rigidbody组件，不使用重力，并且预制体的Collider勾选isTrigger选项，作为触发器使用)

```java

public class ArrowFlyAction : SSAction

{

public Vector3 force; //初始时候给箭的力

public Vector3 wind; //风方向上的力

private ArrowFlyAction() { }

public static ArrowFlyAction GetSSAction(Vector3 wind)

{

ArrowFlyAction action = CreateInstance<ArrowFlyAction>();

//给予箭z轴方向的力

action.force = new Vector3(0, 0, 20);

action.wind = wind;

return action;

}

public override void Update(){}

public override void FixedUpdate()

{

//风的力持续作用在箭身上

this.gameobject.GetComponent<Rigidbody>().AddForce(wind, ForceMode.Force);

//检测是否被击中或是超出边界

if (this.transform.position.z > 30 || this.gameobject.tag == "hit")

{

this.destroy = true;

this.callback.SSActionEvent(this,this.gameobject);

}

}

public override void Start()

{

gameobject.transform.parent = null;

gameobject.GetComponent<Rigidbody>().velocity = Vector3.zero;

gameobject.GetComponent<Rigidbody>().AddForce(force, ForceMode.Impulse);

}

}

```

#### 2. ArrowFactory

在场景控制器需要箭的时候，从空闲的箭队列拿箭或者实例化新的箭放在场景中。

```java

public class ArrowFactory : MonoBehaviour {

public GameObject arrow = null; //弓箭预制体

private List<GameObject> used = new List<GameObject>(); //正在被使用的弓箭

private Queue<GameObject> free = new Queue<GameObject>(); //空闲的弓箭队列

public FirstSceneController sceneControler; //场景控制器

public GameObject GetArrow()

{

if (free.Count == 0)

{

arrow = Instantiate(Resources.Load<GameObject>("Prefabs/arrow"));

}

else

{

arrow = free.Dequeue();

//如果是曾经射出过的箭

if(arrow.tag == "hit")

{

arrow.GetComponent<Rigidbody>().isKinematic = false;

//箭头设置为可见

arrow.transform.GetChild(0).gameObject.SetActive(true);

arrow.tag = "arrow";

}

arrow.gameObject.SetActive(true);

}

sceneControler = (FirstSceneController)SSDirector.GetInstance().CurrentScenceController;

Transform temp = sceneControler.bow.transform.GetChild(2);

//设置新射出去的箭的位置在弓箭上

arrow.transform.position = temp.transform.position;

arrow.transform.parent = sceneControler.bow.transform;

used.Add(arrow);

return arrow;

}

//回收箭

public void FreeArrow(GameObject arrow)

{

for (int i = 0; i < used.Count; i++)

{

if (arrow.GetInstanceID() == used[i].gameObject.GetInstanceID())

{

used[i].gameObject.SetActive(false);

free.Enqueue(used[i]);

used.Remove(used[i]);

break;

}

}

}

}

```

#### 3. FirstSceneController

用户按下鼠标左键后，场景控制器从箭工厂得到箭，然后生成一个风力，传递给动作管理器，让箭飞行，并开启副相机。相关代码如下。

```java

public void Shoot()

{

if((!game\_over || game\_start) && arrow\_num <= 10)

{

arrow = arrow\_factory.GetArrow();

arrow\_queue.Add(arrow);

//风方向

Vector3 wind = new Vector3(wind\_directX, wind\_directY, 0);

//动作管理器实现箭飞行

action\_manager.ArrowFly(arrow, wind);

//副相机开启

child\_camera.GetComponent<ChildCamera>().StartShow();

arrow\_num++;

}

}

```

#### 4. ChildCamera

在设定的时间内，显示副相机，可以让用户看清楚射出去箭在靶上的位置，挂载在副相机上。

```java

public class ChildCamera : MonoBehaviour

{

public bool isShow = false; //是否显示副摄像机

public float leftTime; //显示时间

void Update()

{

if (isShow)

{

leftTime -= Time.deltaTime;

if (leftTime <= 0)

{

this.gameObject.SetActive(false);

isShow = false;

}

}

}

public void StartShow()

{

this.gameObject.SetActive(true);

isShow = true;

leftTime = 2f;

}

}

```

### 5. 弓箭中靶

弓箭中靶后会出现颤抖效果，并检测射中哪一环

#### 1.CollisionDetection

前面说过，箭分为两个部分，箭头和箭身。当箭头进入每一环碰撞器的时候，会消失。然后根据触发了哪一环的碰撞器，来计分。

```java

public class CollisionDetection : MonoBehaviour

{

public FirstSceneController scene\_controller; //场景控制器

public ScoreRecorder recorder; //记录员

void Start()

{

scene\_controller = SSDirector.GetInstance().CurrentScenceController as FirstSceneController;

recorder = Singleton<ScoreRecorder>.Instance;

}

void OnTriggerEnter(Collider arrow\_head)

{

//得到箭身

Transform arrow = arrow\_head.gameObject.transform.parent;

if (arrow == null)

{

return;

}

if(arrow.tag == "arrow")

{

//箭身速度为0，不受物理影响

arrow.GetComponent<Rigidbody>().velocity = new Vector3(0, 0, 0);

arrow.GetComponent<Rigidbody>().isKinematic = true;

recorder.Record(this.gameObject);

//箭头消失

arrow\_head.gameObject.gameObject.SetActive(false); ;

arrow.tag = "hit";

}

}

}

```

#### 2. ArrowTremble

弓箭中靶后，通过回调函数告诉动作管理器，去执行箭颤抖动作。箭颤抖可以通过短时间内上下快速移动实现的。

```java

public class ArrowTremble : SSAction

{

float radian = 0; // 弧度

float per\_radian = 3f; // 每次变化的弧度

float radius = 0.01f; // 半径

Vector3 old\_pos; // 开始时候的坐标

public float left\_time = 0.8f; //动作持续时间

private ArrowTremble() { }

public override void Start()

{

//将最初的位置保存

old\_pos = transform.position;

}

public static ArrowTremble GetSSAction()

{

ArrowTremble action = CreateInstance<ArrowTremble>();

return action;

}

public override void Update()

{

left\_time -= Time.deltaTime;

if (left\_time <= 0)

{

//颤抖后回到初始位置

transform.position = old\_pos;

this.destroy = true;

this.callback.SSActionEvent(this);

}

// 弧度每次增加

radian += per\_radian;

//y轴的位置变化,上下颤抖

float dy = Mathf.Cos(radian) \* radius;

transform.position = old\_pos + new Vector3(0, dy, 0);

}

public override void FixedUpdate()

{

}

}

```

#### 3. SSActionManager

之前创建了ISSActionCallback接口。在箭飞行后会执行一个回调函数SSActionEvent，传递现在中靶的GameObject。实现这个回调函数就可以让箭颤抖动作开始了。

```java

public class SSActionManager : MonoBehaviour, ISSActionCallback

{

private Dictionary<int, SSAction> actions = new Dictionary<int, SSAction>(); //将执行的动作的字典集合

private List<SSAction> waitingAdd = new List<SSAction>(); //等待去执行的动作列表

private List<int> waitingDelete = new List<int>(); //等待删除的动作的key

protected void Update()

{

foreach (SSAction ac in waitingAdd)

{

actions[ac.GetInstanceID()] = ac;

}

waitingAdd.Clear();

foreach (KeyValuePair<int, SSAction> kv in actions)

{

SSAction ac = kv.Value;

if (ac.destroy)

{

waitingDelete.Add(ac.GetInstanceID());

}

else if (ac.enable)

{

ac.Update();

}

}

foreach (int key in waitingDelete)

{

SSAction ac = actions[key];

actions.Remove(key);

DestroyObject(ac);

}

waitingDelete.Clear();

}

protected void FixedUpdate()

{

foreach (SSAction ac in waitingAdd)

{

actions[ac.GetInstanceID()] = ac;

}

waitingAdd.Clear();

foreach (KeyValuePair<int, SSAction> kv in actions)

{

SSAction ac = kv.Value;

if (ac.destroy)

{

waitingDelete.Add(ac.GetInstanceID());

}

else if (ac.enable)

{

ac.FixedUpdate();

}

}

foreach (int key in waitingDelete)

{

SSAction ac = actions[key];

actions.Remove(key);

DestroyObject(ac);

}

waitingDelete.Clear();

}

public void RunAction(GameObject gameobject, SSAction action, ISSActionCallback manager)

{

action.gameobject = gameobject;

action.transform = gameobject.transform;

action.callback = manager;

waitingAdd.Add(action);

action.Start();

}

public void SSActionEvent(SSAction source, GameObject arrow = null)

{

//回调函数，如果是箭飞行动作做完，则做箭颤抖动作

if(arrow != null)

{

ArrowTremble tremble = ArrowTremble.GetSSAction();

this.RunAction(arrow, tremble, this);

}

else

{

//场景控制器减少一支箭

FirstSceneController scene\_controller = (FirstSceneController)SSDirector.GetInstance().CurrentScenceController;

scene\_controller.CheckGamestatus();

}

}

}

```

#### 4. FirstSceneController

在箭颤抖动作做完后，动作管理器调用了场景控制器的ReduceArrow方法，弓箭减少一支，并且生成新的风向。

```java

void Update ()

{

if(game\_start)

{

for (int i = 0; i < arrow\_queue.Count; i++)

{

GameObject temp = arrow\_queue[i];

//场景中超过5只箭或者超出边界则回收箭

if (temp.transform.position.z > 30 || arrow\_queue.Count > 5)

{

arrow\_factory.FreeArrow(arrow\_queue[i]);

arrow\_queue.Remove(arrow\_queue[i]);

}

}

}

}

public void Shoot()

{

if((!game\_over || game\_start) && arrow\_num <= 10)

{

arrow = arrow\_factory.GetArrow();

arrow\_queue.Add(arrow);

//风方向

Vector3 wind = new Vector3(wind\_directX, wind\_directY, 0);

//动作管理器实现箭飞行

action\_manager.ArrowFly(arrow, wind);

//副相机开启

child\_camera.GetComponent<ChildCamera>().StartShow();

//用户能射出的箭数量减少

recorder.arrow\_number--;

//场景中箭数量增加

arrow\_num++;

}

}

public void CheckGamestatus()

{

if (recorder.arrow\_number <= 0 && recorder.score < recorder.target\_score)

{

game\_over = true;

return;

}

else if (recorder.arrow\_number <= 0 && recorder.score >= recorder.target\_score)

{

round++;

arrow\_num = 0;

if (round == 4)

{

game\_over = true;

}

//回收所有的箭

for (int i = 0; i < arrow\_queue.Count; i++)

{

arrow\_factory.FreeArrow(arrow\_queue[i]);

}

arrow\_queue.Clear();

recorder.arrow\_number = 10;

recorder.score = 0;

recorder.target\_score = targetscore[round];

}

//生成新的风向

wind\_directX = Random.Range(-(round + 1), (round + 1));

wind\_directY = Random.Range(-(round + 1), (round + 1));

CreateWind();

}

```

增加风向文本，通过判定风力值的大小来显示是哪个方向的风，风力几级。UserGUI可以通过IUserAction接口得到风力文本，实现是在FirstSceneController中。

```java

//根据风的方向生成文本

public void CreateWind()

{

string Horizontal = "", Vertical = "", level = "";

if (wind\_directX > 0)

{

Horizontal = "西";

}

else if (wind\_directX <= 0)

{

Horizontal = "东";

}

if (wind\_directY > 0)

{

Vertical = "南";

}

else if (wind\_directY <= 0)

{

Vertical = "北";

}

if ((wind\_directX + wind\_directY) / 2 > -1 && (wind\_directX + wind\_directY) / 2 < 1)

{

level = "1 级";

}

else if ((wind\_directX + wind\_directY) / 2 > -2 && (wind\_directX + wind\_directY) / 2 < 2)

{

level = "2 级";

}

else if ((wind\_directX + wind\_directY) / 2 > -3 && (wind\_directX + wind\_directY) / 2 < 3)

{

level = "3 级";

}

else if ((wind\_directX + wind\_directY) / 2 > -5 && (wind\_directX + wind\_directY) / 2 < 5)

{

level = "4 级";

}

wind = Horizontal + Vertical + "风" + " " + level;

}

```

### 结果



