# 12 示例教程—跑酷游戏

在第6章，我们已经简单演示了如何播放玩家动画。在本游戏教程中，玩家的动作与之前例子中并无什么区别。于是我们在原先工程的基础上，演示本次教程。

## 工具准备

在开始制作游戏之前，我们先准备好我们游戏中需要用到的工具。在这里我们使用了一款用于动画制作的插件—DoTween插件。下载地址：<http://dotween.demigiant.com/download.php>

下载完成后，在我们刚建好的工程中新建一个文件夹并命名为Plugins，将DoTween插件解压后的文件放入这个文件夹当中。不明白文件夹有什么用的参考这篇文章：<http://www.xuanyusong.com/archives/3229>

## 需求分析

工具准备完成后。现在需要分析我们这款跑酷游戏的需求，或者说实现功能。我们以手机上经典的跑酷游戏—神庙逃亡为例。进行我们游戏的需求分析。在这款游戏上主要实现了以下功能：

1. 人物动作——跳跃，跑步，下滑。
2. 地图生成——前进地图，转向地图，道具。
3. 背景音效——音乐。
4. 得分显示——游戏分数。

但是原版游戏的地图不够炫酷，我们这边将地图生成改变一下，做成道路动起来，能够自己到脚下铺路的这种。

## 人物动作

由于之前在第6章，我们已经搭建好了玩家的动画状态机，这里我们就不再进行演示。

### 代码控制

虽然之前我们已经能够控制玩家的动画进行播放与关闭，但是对于一个游戏来说，这样还是远远不够。于是我们需要用复杂一点的方法来进行我们游戏中玩家的动画播放与关闭，如下：

|  |
| --- |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  using DG.Tweening; //引入DoTween插件  public class PlayerController : MonoBehaviour {  public Animator playAnimtor; //动画控制组件  public float jumpPower; //玩家的跳跃高度  RuntimeAnimatorController nowController; //现在的运行时动画控制器  AnimationClip[] cilps; //现在播放的动画数组  void Start ()  {  playAnimtor = GetComponent<Animator>();  nowController = playAnimtor.runtimeAnimatorController;  cilps = nowController.animationClips;  } |

获取动画组件

由于官方提供的动画与我们的游戏不太匹配，我们需要对关闭动画的时间进行微调，达到满意的效果，所以我们所用了更为复杂的方法来进行动画关闭的控制。详情如下：

|  |
| --- |
| void Start ()  {  for (int i=0;i<cilps.Length; i++)  {  if (cilps[i].events.Length<=0)  {  switch (cilps[i].name)  {  case "JUMP00": //如果当前是跳跃动画  AnimationEvent endEvent = new AnimationEvent();  endEvent.functionName = "JumpEnd";  endEvent.time = cilps[i].length - (20.0f / 56.0f) \* 1.83f;//设置事件的执行时间 cilps[i].length表示动画的播放时长  cilps[i].AddEvent(endEvent); //把事件添加进跳跃动画当中  break;  }  }  }  }  public void JumpEnd()  {  playAnimtor.SetBool("IsJump", false);  } |

通过时间关闭动画

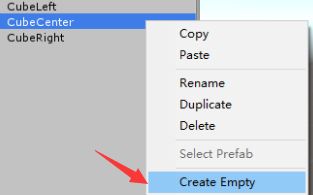
在上面的程序中，我们以跳跃动画为例子。通过动画的名字，在动画播放到某一个时间添加了动画事件，进行动画关闭。那么程序运行后，就会在我们设置好的时间，执行这个事件，进行动画的关闭。这样我们就能够流畅的进行游戏中动画的切换了。

## 地图生成

### 地图模板创建

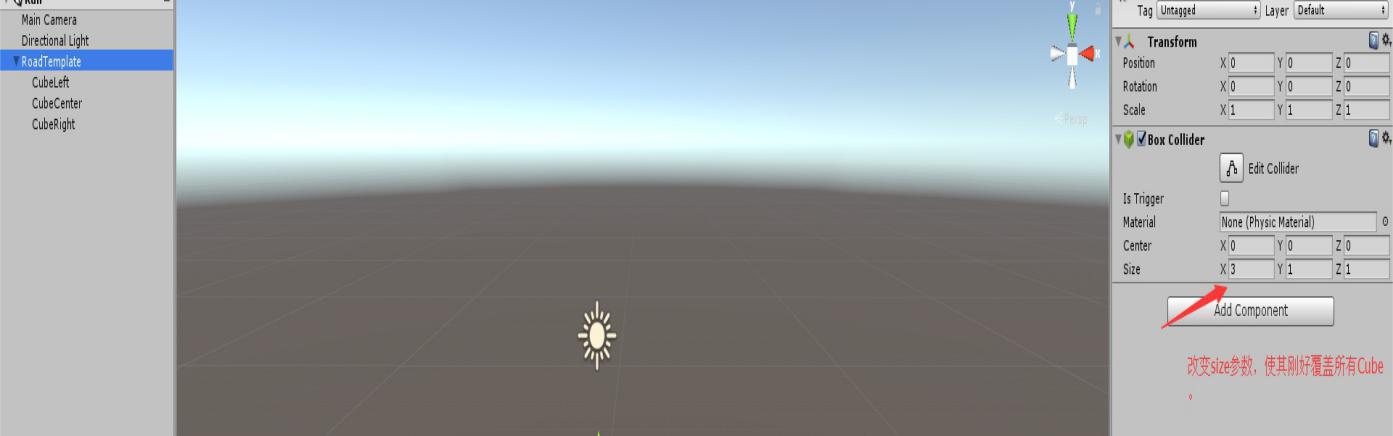
在神庙逃亡游戏中，游戏地图是由几个小一点地图模板，随机组合生成的。我们的轻量版跑酷游戏当中，不需要这么多的地图模板，只需要一个就够了。

首先我们在场景视图中创建一个Cube，并将他的TransForm.position改成0，0，0.同时更改这个Cube的名字为CubeCenter,去掉这个Cube上面的BoxCollider。创建第二个Cube，并将他的TransForm.position改成1，0，0.同时更改这个Cube的名字为CubeRight,去掉这个Cube上面的BoxCollider。创建第三个Cube，并将他的TransForm.position改成-1，0，0.同时更改这个Cube的名字为CubeLeft,去掉这个Cube上面的BoxCollider。在CubeCenter下面创建一个空物体，并命名为RoadTemplate。



创建空物体

将这个空物体设为3个cube的父节点，同时添加一个BoxCollider，并将这个碰撞盒的Size改成3，1，1.使其刚好覆盖3个Cube。



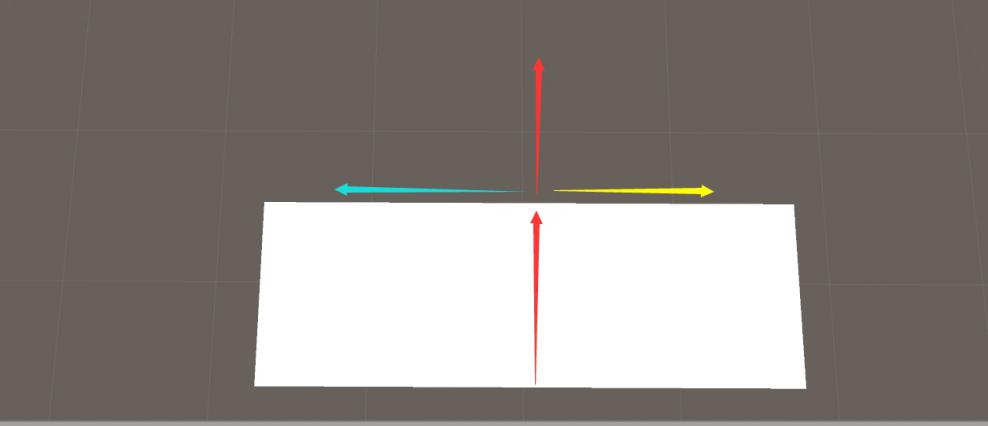
设置Cube参数

然后将这个道路模板保存为预制体。

### 地图规则

在跑酷游戏中，玩家的运动方向始终只有3个方向：

1. 直线方向——foward。
2. 相对于自己的左方向——left。
3. 相对于自己的右方向——right。



运动方向

那么要生成能够使用的，无限的地图，那么我们必须制定游戏道路的生成规则与我们玩家的运动方向相匹配，才能达到这个目标。那么规则如下：

1. 在你生成一次转向道路后，10回合之内不能再进行转向。（如果还能继续转向会出现无法使用的道路）
2. 只有在你完全经过一个道路进入下一个道路后，才能进行下一个道路的生成。（防止无限生成，占用不必要的资源。同时也防止道路生成过长，出现道路重叠，影响游戏体验）
3. 转向道路转向角度固定为90度。

依照上面的规则，我们创建一个GameMode.cs文件，在里面实现我们的道路生成方法。如下：

|  |
| --- |
| public void BuidRoad()  {  int turnSeed = Random.Range(1,10); //随机数用于确定是否转向  if (turnSeed == 1 && buidfound<=0) //是转向且当前离上一次生成差了10次  {  buidfound = 10; //回合数更新  int dictSeed = Random.Range(1,3);  for (int i = 0; i < 3; i++) //先生成3个格子的道路，作为转向区  {  var tmpRoad = Instantiate(roadTemplate, guideTrs.position, guideTrs.rotation);  PlayRoadAnimator(tmpRoad);  roads.Add(tmpRoad);//添加当前生成的道路进入可还原列表  guideTrs.position += guideTrs.forward;  }  if (dictSeed == 1)  {  guideTrs.position -= guideTrs.forward \* 2; //转向区域生成完成后，引导物体回退2格  guideTrs.Rotate(Vector3.up, 90); //转向  guideTrs.position += guideTrs.forward \* 2; //转向后引导物体的forward轴改变，改变pos值，到达下一个道路的生成地点  }  else  {  guideTrs.position -= guideTrs.forward \* 2;  guideTrs.Rotate(Vector3.up, -90);  guideTrs.position += guideTrs.forward \* 2;  }  }  else  {  var tmpRoad = Instantiate(roadTemplate, guideTrs.position, guideTrs.rotation);  PlayRoadAnimator(tmpRoad);  roads.Add(tmpRoad);//添加当前生成的道路进入可还原列表  guideTrs.position += guideTrs.forward; //每生成一个道路，引导物体的位置改变  }  buidfound--;  } |

地图生成

在上面的代码中，我们通过使用一个引导物体—guideTrs，用来确定接下来道路的位置与旋转等信息，在这个基础上进行道路生成。

### 运动的地图

经过了上面的步骤，我们已经得到了一个能够游玩的跑酷游戏地图。但是我们的目标是做一个运动的，能够自己到脚下铺路的地图。现在的地图还远远达不到我们的要求，这个时候就需要用到我们刚开始下载的插件，来完成这样一个动画的效果。

我们先创建一个RoadChildrenChange.cs文件。进行游戏道路的信息改变。如下：

|  |
| --- |
| using UnityEngine;  using DG.Tweening;  public Vector3 firstLocalPos; //保存最开始的相对坐标  public Quaternion firstLocalRotation; //保存最开始的相对旋转  private void Awake()  {  firstLocalPos = transform.localPosition;  firstLocalRotation = transform.localRotation;  }  public void PosChange() //位置改变  {  int changeValueUp = 0;  int chageValueRight = 0;  while (Mathf.Abs(changeValueUp)<=4.0f) //限定改变坐标的Y轴的范围  {  changeValueUp = Random.Range(-10,10);  }  while (Mathf.Abs(chageValueRight) <= 4.0f)//限定改变坐标的X轴的范围  {  chageValueRight = Random.Range(-10, 10);  }  transform.localPosition += transform.up \* changeValueUp;  transform.localPosition += transform.right \* chageValueRight;  }  public void ChangeRotate() //旋转改变  {  transform.Rotate(transform.right, Random.Range(0, 180));  transform.Rotate(transform.forward, Random.Range(0, 180));  transform.Rotate(transform.up, Random.Range(0, 180));  }  public void Init() //初始化信息，回到一开始的坐标与旋转  {  transform.DOLocalMove(firstLocalPos, time); //使用插件在time时间后回到初始坐标  Tween t = transform.DOLocalRotateQuaternion(firstLocalRotation, time);//回到初始旋转  t.OnComplete(InitGold);  } |

随机改变子物体信息

在上面的程序中，我们在一开始保存了当前物体的坐标坐标与旋转信息。并声明了3个函数，完成我们想要的功能。如下：

1. PosChange()——用于改变物体当前的位置即坐标。
2. ChangeRotate()——用于改变物体当前的旋转。
3. Init()——用于初始化物体的信息，当进行铺路时进行调用，通过插件做出动画效果。

以上就完成了我们想要的动画效果，然后将其挂载到我们上面创建好的地图模板—RoadTemplate下的3个子物体Cube上。

虽然我们现在已经完成了动画效果方法的创建，但是细心的读者会发现，我们并没有地方在使用我们这创建好的3个方法。所以，接下来我们要做的就是，控制使用这3个方法，达成我们的目的。首先，新建一个RoadControl.cs，挂载到我们的地图模板上，在这里我们这我们主要是获取子物体上的方法，并进行调用。那么实现内容如下：

|  |
| --- |
| using UnityEngine;  public class RoadController : MonoBehaviour {  public RoadChildrenChange[] childrens; //子物体上的脚本数组  void Awake()  {  childrens = GetComponentsInChildren<RoadChildrenChange>(); //一开始获取所有子物体上的脚本  }  public void ChangeChildrens() //执行所有子物体上的改变信息的方法  {  for (int i=0;i< childrens.Length;i++)  {  childrens[i].nowType = RoadType.road;  childrens[i].PosChange();  childrens[i].ChangeRotate();  childrens[i].isTurn = true;  }  }  public void InitChildrens() //初始化所有子物体的信息  {  for (int i = 0; i < childrens.Length; i++)  {  childrens[i].Init();  childrens[i].isTurn = false;  }  } |

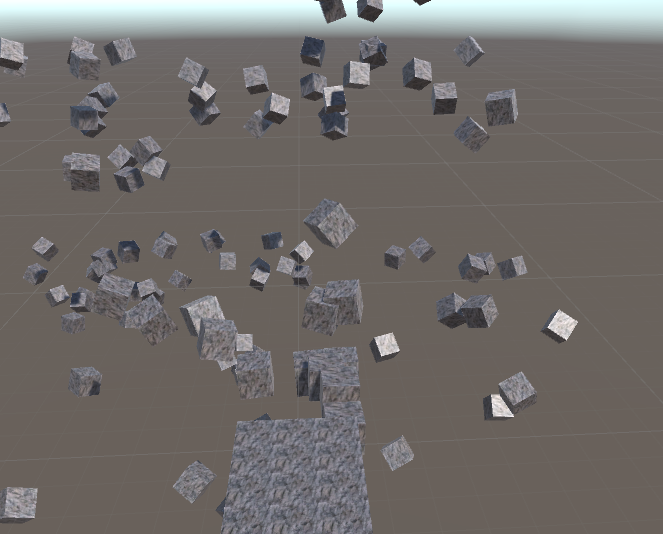
执行子物体上的方法

完成后，虽然看起来我们还是没有在调用这个方法，其实我们在地图生成的程序中已经在使用这个方法，完成我们的目的了，就是PlayRoadAnimator(tmpRoad)这一句，其实现方法如下：

|  |
| --- |
| public void PlayRoadAnimator(GameObject road) //播放道路的动画  {  var tmpRoadController = road.GetComponent<RoadController>();  if (tmpRoadController==null) { return; }  tmpRoadController.ChangeChildrens();  } |

播放道路动画

这样我们在道路一开始生成好之后，就在生成位置的基础上，随机改变坐标旋转信息，造成一种散乱的场景，如下图。关于方块的坐标与旋转信息的还原，这个与玩家当前的移动位置有关，在稍后的人物控制章节会进行说明。



散乱的地图

### 道具生成

跑酷游戏中的道具多种多样，我们以金币举例进行说明。首先在网上找到金币的模型，如下图。然后放入场景当中，调整大小相对游戏来说合适就好。完成后，保存为预制体，放入Resourecs文件夹下，没有的话，在Assets文件夹下创建一个同名文件夹放入即可。



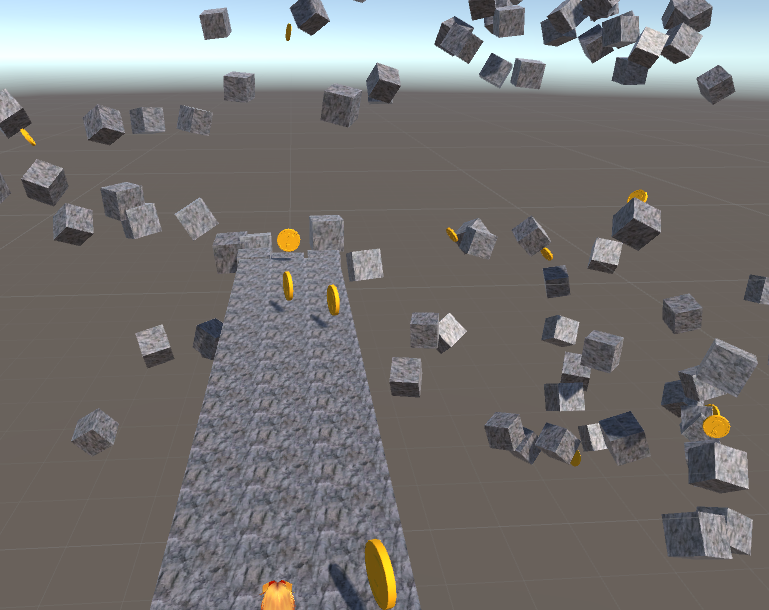
金币

接下来就是生成金币的环节了。在跑酷游戏当中，金币总是生成在道路上方，只是高度不同而已。我们的游戏中暂时设定金币生成的高度总是在道路上面就行。由于我们的金币生成位置总是与道路位置相似，只是高度不同。于是我们在RoadChildrenChange.cs文件中，加入生成金币的方法，如下：

|  |
| --- |
| private void Start()  {  gold = Resources.Load("Gold") as GameObject; //读取文件金币的预制体  bool isCreat = Random.Range(1, 10) == 3 ? true : false; //是否生成金币  if (isCreat)  {  Vector3 tmpPos = transform.position;  gold = Instantiate(gold, tmpPos += transform.up \* 1.2f, Quaternion.identity);  gold.transform.SetParent(transform);  nowQuat = gold.transform.rotation; //保存当前金币的旋转，方便初始化。  }  } |

金币生成

这样我们就在一开始道路生成的时候，在每一个方块上进行金币的随机生成。效果如图：



金币生成效果

### 复杂地形

#### 地形分析

现在我们已经得到了一个有金币，有道路的跑酷游戏场景。但是现在还不够，因为我们现在只能够在平地上进行跑酷，没有难度。我们需要加入复杂的地形，提高游戏的趣味性。这里暂时制作5种地形，供读者参考。如下：

1. 上升地形
2. 下降地形
3. 左斜地形
4. 右斜地形
5. 陷阱地形

#### 方向地形

方向地形的制作比较简单，只需要在引导物体更换坐标时，改变下一次的相应方向的值就行。我们在GameMode.cs文件中新声明地形的生成函数，以上升地形举例。如下：

|  |
| --- |
| public void BuidUpTerrain() //生成上升地形  {  var tmpRoad = Instantiate(roadTemplate, guideTrs.position, guideTrs.rotation);  PlayRoadAnimator(tmpRoad);  roads.Add(tmpRoad); //添加当前生成的道路进入可还原列表  guideTrs.position += guideTrs.forward;  guideTrs.position += guideTrs.up \* 0.2f;  } |

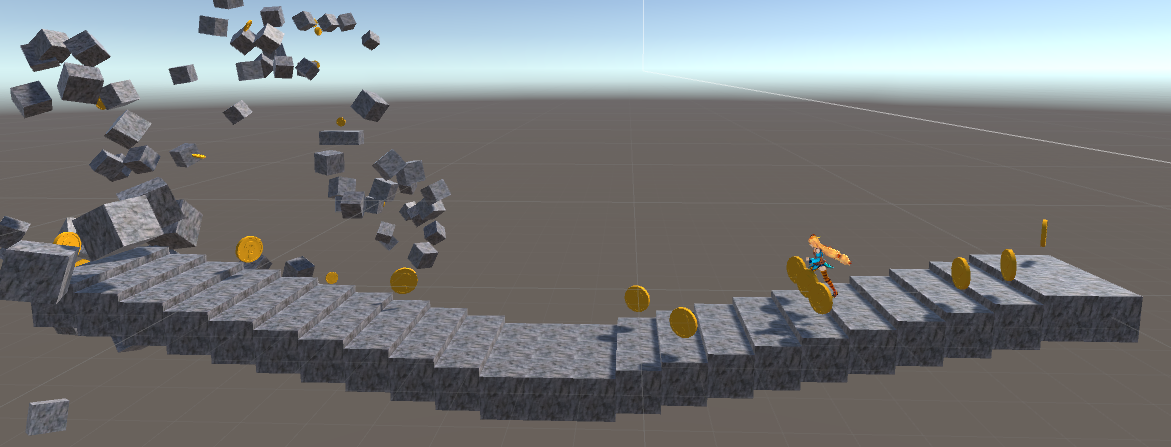
生成上升地形

但是地形地图需要连续一段才能看出效果，我们就设定任何一种地形都是至少由10个道路组成，这样的话我们就必须在13.6.3小节中的地图生成程序中重新加入新的限制生成条件，才能完成我们想要的效果，如下：

|  |
| --- |
| public bool isBuidDirRoad; //是否生成方向道路  int dirRoadType; //方向道路的类型  int dirRoadNumber; //方向道路的数量  public void BuidRoad() //生成道路的方法  {  if (isBuidDirRoad && dirRoadNumber > 0)  {  switch (dirRoadType) //通过道路的类型来进行生成  {  case 1:  BuidUpTerrain();  dirRoadNumber--;  break;  case 2:  BuidDownTerrain();  dirRoadNumber--;  break;  case 3:  BuidLeftTerrain();  dirRoadNumber--;  break;  case 4:  BuidRightTerrain();  dirRoadNumber--;  break;  }  if (dirRoadNumber <= 0)  {  isBuidDirRoad = false;  }  } |

生成地形限制

在之前的程序中我们已经进行了道路生成的限制，记录了上一次生成的地形类型—dirRoadType，以及已经生成的数量—dirRoadNumber 。如果生成的数量小于10个，在下次就继续生成，已经超过10的话，就关闭地形生成。完成后，效果如图：



上升与下降地形

#### 陷阱地形

陷阱地图生成虽然较为简单，只需要改变一下道路模板的自身旋转，但是有3种模式，如下：

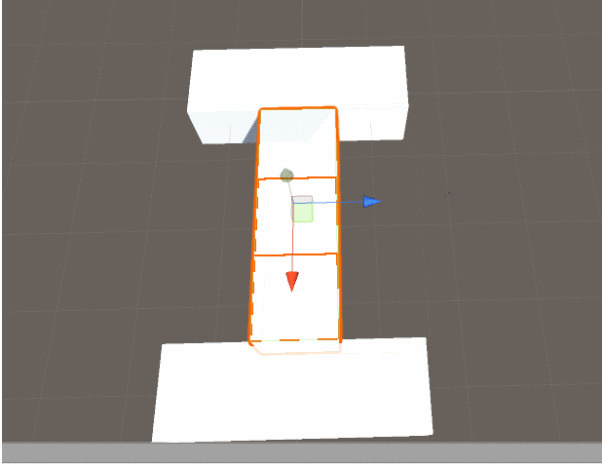
1. 相对于道路的右边
2. 相对于道路居中
3. 相对于道路右边

知道了这几种模式，我们就能在代码中进行实现，在GameMode.cs文件中声明新的陷阱地形的生成函数，如下：

|  |
| --- |
| public void BuidTrapRoad()  {  guideTrs.position += guideTrs.forward;  var tmpRoad = Instantiate(roadTemplate, guideTrs.position, guideTrs.rotation);  var tmpController = tmpRoad.GetComponent<RoadController>();  tmpController.ChangeRoadType();  tmpRoad.transform.Rotate(Vector3.up,90);  int trapType = Random.Range(1,4); //用于确定陷阱的相对位置--1左边,2居中,3右边！  switch (trapType)  {  case 1:  tmpRoad.transform.position += tmpRoad.transform.forward;  break;  case 2: //居中类型不做操作  break;  case 3:  tmpRoad.transform.position -= tmpRoad.transform.forward;  break;  }  guideTrs.position += guideTrs.forward \* 2.0f;  PlayRoadAnimator(tmpRoad);  roads.Add(tmpRoad); //添加当前生成的道路进入可还原列表  } |

陷阱地形生成

我们在这个函数里面对生成的道路进行了一个转向处理，并随机移动了以下位置，使得生成的陷阱地形，更具有随机性，而不是固定的处于道路的一个地方。如图：



居中的陷阱地形

## 人物控制

### 人物动作分析

到了这一步，我们完成游戏的大部分内容了，接下了来就是游戏中的角色控制了，参考神庙逃亡，我们可以分析游戏中有以下动作：

1. 向前移动
2. 左右移动
3. 左转
4. 右转
5. 跳跃
6. 下滑

在之前第八章节，我们已经小小的演示了动画功能，并准备好了大量的动作，这里我们可以直接开始制作。

### 角色控制器添加

关于游戏中玩家的移动控制，我们使用Unity提供角色控制器进行人物的控制。选中场景中的玩家，在Inspector视图中给玩家添加上角色控制器组件—Character Controller,然后调整大小刚好覆盖玩家模型即可。

### 向前移动

游戏中玩家向前移动，本质上就是一直朝着自己的Z轴跑动。这样的话，我们需要在PlayerController.cs文件中获取角色控制器组件，然后进行玩家移动处理。如下：

|  |
| --- |
| public CharacterController playController;  public Vector3 MoveIncrements;  public float moveSpeed = 6.0f; //玩家的游戏移动速度  void Start ()  {  playController = GetComponent<CharacterController>();  }  void Update ()  {  moveSpeed += Time.deltaTime\*0.3f;  MoveIncrements = transform.forward \* moveSpeed \* Time.deltaTime;  playController.Move(MoveIncrements); //玩家移动  playAnimtor.SetFloat("MoveSpeed",playController.velocity.magnitude); //跑步动画更新  } |

玩家向前移动

此处我们使用的是charactercontroller.move（）函数。这个函数移动是通过调用函数时的参数来进行移动的，它是一个增量的移动函数。如果你在玩家的坐标为（1，1，1）时调用这个函数，并给了一个（1，1，1）参数。那么下一帧，玩家的坐标就会改变为（2，2，2）。我们在代码中更改了MoveIncrements的相对于玩家的正方向轴的值，那么下一帧玩家的坐标就会增加我们设好的参数—MoveIncrements这么多，达成我们的移动效果。同时这个时候我们也在通过玩家的移动速度来更新玩家的跑步动画。

### 左右移动

左右移动的本质是朝自己的X轴进行移动，与前面向前移动差不多，只不过这个需要玩家的按键输入。这样的话，我们只需要在之前的移动函数中小小的加上几段代码就可以实现这个功能，如下：

|  |
| --- |
| float transverseSpeed = 5.0f; //玩家横向的移动速度  void Update ()  {  float moveDir = Input.GetAxis("Horizontal"); //获取玩家的键盘输入A与D键  MoveIncrements += transform.right \* transverseSpeed \* Time.deltaTime\*moveDir; //改变移动增量  } |

左右移动

### 左转与右转

左转与右转情况类似，都是玩家进行转向，只是转向的方向不同，但是转向的角度都是一样的，都是90度。这里我们以左转举例，如下：

|  |
| --- |
| if (Input.GetKeyDown(KeyCode.J) && isTurnleftEnd)  {  isTurnleftEnd = false; //更新转向状态  transform.Rotate(Vector3.up,-90);  Quaternion tmpQuaternion = transform.rotation; //计算转向后的四元数并保存  transform.Rotate(Vector3.up, 90); //角度回滚  Tween tween = transform.DORotateQuaternion(tmpQuaternion, 0.3f); //使用DoTween插件进行转向的平滑运动  tween.OnComplete(() => isTurnleftEnd = true); //动画结束后转向状态更新  } |

左转

上面使用了DoTween插件进行了玩家的平滑转向，而且新增加了一个状态—isTurnleftEnd用来确定当前转向是否完成，防止出现由于转向没有完成又继续转向造成的错误。这样我们就能在玩家按下J键的时候进行游戏人物左转。

### 跳跃与下滑

跳跃功能与下滑功能原理相似，我们以跳跃功能举例进行说明。

游戏中玩家的跳跃是由以下2部分组成：

1. 跳跃动作播放
2. 玩家位置升高

首先我们来实现玩家跳跃动作的播放。虽然我们在之前的动画小例子中已经简单的演示了如何播放玩家的跳跃动画，但是游戏中跳跃动画的播放有许多限制，只有将这些限制条件一并添加上去，游戏才能正常运行。于是我们在之前移动程序的基础上添加播放跳跃动画的代码。如下：

|  |
| --- |
| bool isJumpState; //现在是否是转向状态  if (Input.GetKeyDown(KeyCode.Space) && playController.isGrounded)//按下空格且在地面  {  isJumpState = true; //更新跳跃状态  playAnimtor.SetBool("IsJump", true); //播放跳跃动画  } |

玩家跳跃

现在运行游戏并按下空格，就会看到玩家已经能够播放跳跃这个动作了，但是这还不够，我们还需要将玩家的位置升高才算完成，那么我们这个时候还需要这上面的写入位置改变的代码，才能够实现这个功能。如下：

|  |
| --- |
| if (isJumpState) //如果现在正在进行跳跃  {  MoveIncrements.y += jumpPower\*Time.deltaTime;//根据设置好的跳跃高度进行平滑运动  }  else  {  MoveIncrements.y += playController.isGrounded ? 0f : -5.0f \* Time.deltaTime \* 1f; //更新重力  } |

玩家高度变化

### 道路动画播放

在上面的小节中，我们提到了方块的坐标与旋转信息的还原与玩家的移动位置有关，这里我们就来实现根据玩家当前位置来播放道路的动画效果。

首先我们需要删除玩家当前脚下的道路。我们通过角色控制器提供碰撞函数来实现这个功能。如下：

|  |
| --- |
| private void OnControllerColliderHit(ControllerColliderHit hit)  {  if (hit.gameObject!=nowRoad) //去重复避免删除错误  {  nowRoad = hit.gameObject;  Destroy(hit.gameObject,1.0f);  GameMode.instance.BuidRoad(); //生成道路  GameMode.instance.CloseRoadAnimator();  }  } |

在上面的删除代码中，我们设置好了玩家当前道路的删除时间。同时我们调用GameMode代码中的道路生成函数，以及动画关闭函数—CloseRoadAnimator()。即方块的坐标信息还原。所以我们还需要在GameMode的代码的基础上加入新的代码，实现我们的功能。如下：

|  |
| --- |
| public static GameMode instance; //声明当前脚本的单例  public List<GameObject> roads; //保存现在能够进行还原的道路  public void CloseRoadAnimator()  {  if (roads.Count<=0) { return; }  var tmpRoadController = roads[0].GetComponent<RoadController>();//取一个进行还原  if (tmpRoadController!=null)  {  tmpRoadController.InitChildrens(); //执行当前道路的还原函数  }  roads.RemoveAt(0);  } |

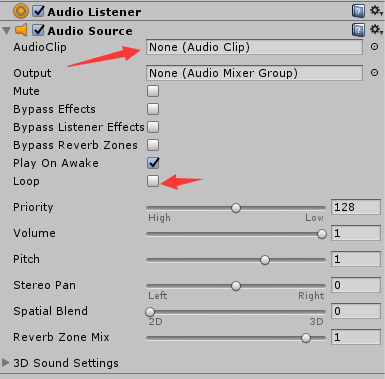
在上面的代码中。我们通过声明了当前脚本的单例，来使得外部的代码文件能够使用GameMode中的函数。同时使用了一个列表将之前所有已经播放过动画的道路给保存。在我们执行关闭动画函数的时候，取其中的第一个来进行当前道路信息的还原。这样就能通过玩家移动的位置来进行道路的还原以及无限道路的生成了。

## 游戏音效

经过上面的一系列实现，现在我们的游戏已经能够进行简单试玩了。但是我们还是缺少一点生气，这个时候就需要加入一点音效，让我们的游戏更具有生机。

### 背景音效

第一步我们添加游戏的背景音效。Unity支持绝大部分的音频格式，不用担心格式的问题，我们只需要找到适合游戏的背景音乐就行。将找到的音频放入工程的Aseet目录下，Unity就会自动识别。然后我们选中游戏中的相机物体，并給这个物体添加上音频播放组件—AudioSource。如图：



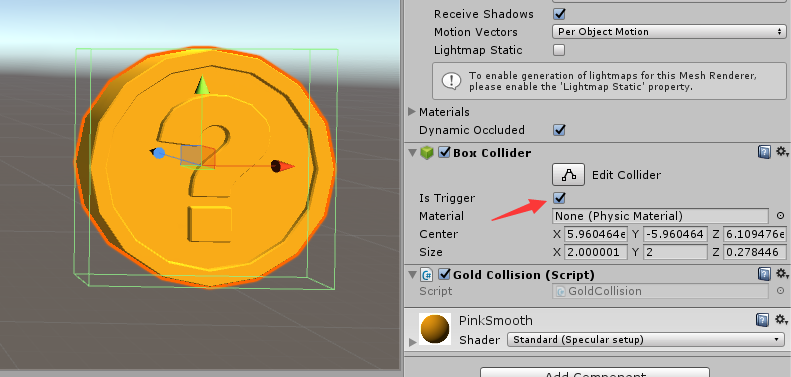
音频播放组件

其中AudioClip选项是我们的播放音源，Play On Awake按钮是表示游戏一开始就进行播放，Loop按钮表示循环播放。由于我们的背景音乐是循环播放不停歇的，所以将我们的背景音乐拖入AudioClip选项，并勾选循环按钮—Loop，运行游戏后。我们就能够听到我们选择的游戏背景音乐了。

### 道具音效

由于我们游戏当中只有金币这一种道具，于是我们使用一种简单音频播放方式来进行道具音效的播放。

首先还是得网上找到我们吃金币的音效，将其放入Resources目录下。然后在我们在金币的预制体上添加一个碰撞器，调整碰撞器大小刚好覆盖金币模型，并勾选为触发器—IsTrigger。接着创建一个GoldCollision.cs文件，将其挂载到金币的预制体上。如图：



金币设置

准备完成后，我们在GoldCollision.cs文件中写入以下代码，来进行道具音效的播放。如下：

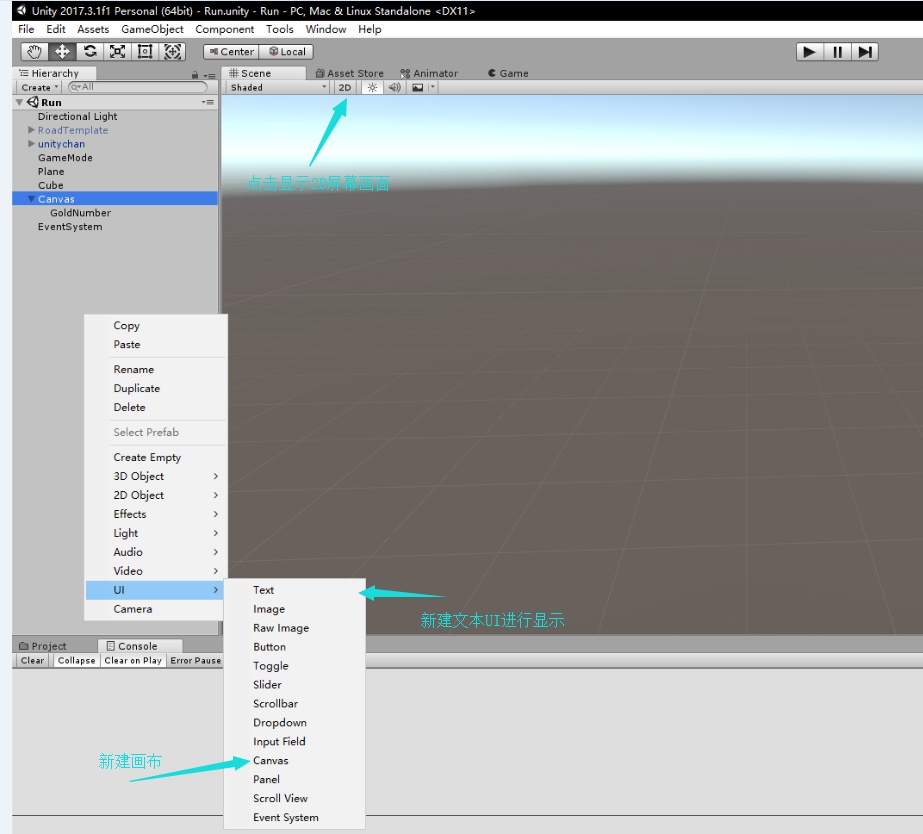
|  |
| --- |
| using UnityEngine;  public GameMode gameMode;  AudioClip clip; //道具音效  void Start ()  {  gameMode = GameObject.FindObjectOfType<GameMode>();  clip = Resources.Load("Clip2",typeof(AudioClip)) as AudioClip; //读取音效  }  private void OnTriggerExit(Collider other)  {  var player = other.gameObject.GetComponent<PlayerController>();  if (player)//如果是玩家触发  {  gameMode.goldNumber++; //金币数量增加  AudioSource.PlayClipAtPoint(clip,Camera.main.transform.position,0.8f); //播放音效  Destroy(gameObject);//删除自己  }  } |

播放道具音效

这里并没有添加音频播放组件来进行音乐播放，而是使用了AudioSource.PlayClipAtPoint()这个方法来完成我们的目的，这个方法的3个参数分别是音源，播放位置，以及音量。

## 得分显示

跑酷游戏的得分基本上是由吃到的金币数与跑的距离进行结合运算出的一个数字，我们这边就简单的使用吃到的金币数作为我们游戏中的得分。关于分数在游戏中的显示，首先，我们的新建一个Canvas，在Canvas下面新建一个Text的UI。打开2D视图进行物体的位置改变，调整至游戏画面的左上方。如图：



UI创建

接下来就是根据金币的数量来改变显示。在之前的程序当中已经将金币数量累积的方法写好，然后我们现在需要做到的就是获取这个金币数量。于是我们在GameMode.cs文件中加入以下代码。如下：

|  |
| --- |
| public int goldNumber; //吃到的金币数  public Text numberText; //数量的显示文本  private void Update()  {  numberText.text ="当前金币数："+ goldNumber.ToString();  } |

UI显示更新

## 触摸控制

经过上面的步骤，我们已经完成了整个游戏的制作。在windows平台上已经能够运行，甚至现在还可以使用unity的强大功能，将你的游戏打包到安卓平台上。虽然能够在安卓平台上运行，但是我们的操作方式是使用键盘上的输入来进行游戏的控制，在安卓平台上使用键盘来进行控制是不行的。所以，我们需要在我们的代码上重新写入第二套控制方式，来进行游戏在安卓平台运行时，游戏人物的控制。

### 向量的点乘

在安卓平台上，玩家的输入是触摸屏上手指的点击与滑动。如何将这些输入转换成我们需要的操作信息。这个时候就需要使用向量的点乘来完成我们的目的。

向量的点乘:a \* b

公式：a \* b = |a| \* |b| \* cosθ

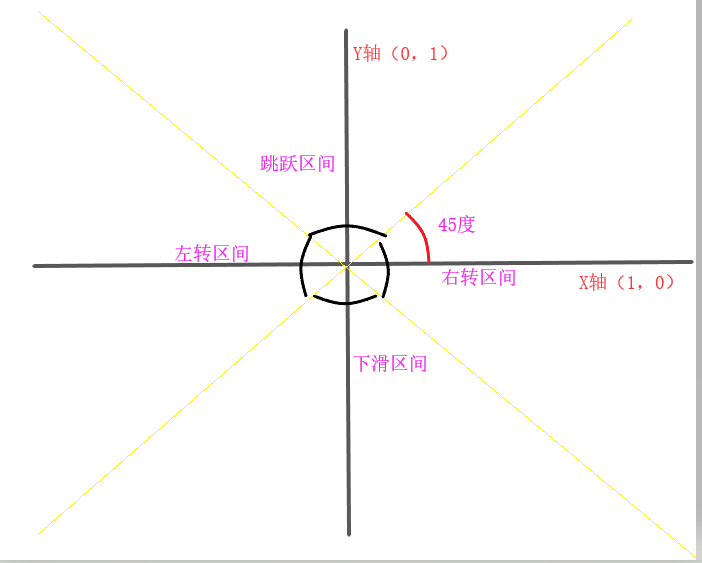
上面的列举出了我们需要用到的公式。在手机上。我们可以通过玩家在手机上的滑动，获得一条向量。通过这条向量，我们通过点乘可以计算出，当前向量与屏幕坐标系X轴与屏幕坐标系Y轴之间的夹角。通过它们之间的夹角来判断我们游戏中的人物此时应该如何移动，或者说该出现什么效果。

既然我们是通过玩家滑动方向与XY轴的夹角来判断运动方向，那么我们需要制定什么样的夹角范围来确认当前游戏中玩家的移动方向。通过滑动方向的向量与Y轴的夹角，来确定方向是朝上一点，还是朝下一点。与X轴的夹角来确定是朝左还是右边。

于是我们进行规定：

1. 跳跃操作—与Y轴的夹角大于等于0度小于等于45度，且处于1，2象限内。
2. 右转操作—与Y轴的夹角小于135度大于45度，且处于1，4象限内。
3. 左转操作—与Y轴的夹角小于135度大于45度，且处于2，3象限内。
4. 下滑操作—与Y轴的夹角大于等于135度小于等于180度，且处于3，4象限内。

图示如下：



### 代码实现

知道了我们现在需要的完成的目标，接下来就是在代码中进行实现。

由于我们之前已经在代码中实现了我们的控制代码，现在要做的是在手机上触摸控制的代码，如果直接写入代码文件中，会出现冲突以及错误。所以我们需要让编译器区分我们后面写入的代码与前面的代码。在本例中我们使用了unity提供的预处理指令来进行平台的区分。

首先在PlayerController.cs文件中的Update函数中，写入预处理指令。如下：

|  |
| --- |
| #if UNITY\_STANDALONE\_WIN //如果是在Windows平台  /\*在这个范围的代码只会在windows平台上才会执行\*/  #endif |

在上面的代码中，我们使用预处理指令让编译器识别我们在windsow上平台才会执行的代码。接下来就需要把我们之前写好的控制代码，剪切到这个代码块中即可。

然后就进行我们触摸控制的实现，在PlayerController.cs中写入以下代码：

|  |
| --- |
| public Vector2 ScreenAixY = new Vector2(0, 1); //屏幕坐标系Y轴  public Vector2 ScreenAixX = new Vector2(1, 0); //屏幕坐标系X轴  Vector2 startPos; //开始点  Vector2 EndPos; //结束点  bool isInput = false; //是否触摸  float angle; //夹角度数 |

这是我们在进行实现触摸控制所需要使用到的字段，我们还需要写入预处理指令来区分我们在安卓平台上执行的代码，如下：

|  |
| --- |
| #if UNITY\_ANDROID  /\*这个范围的代码只会在安卓Android平台上执行\*/  #endif |

这样我们就可以在这个预处理指令的代码块内，写入我们的触摸控制代码。但是，建议读者

在写触摸控制代码之前，先将我们的目标平台切换至安卓（可参考打包章节）。因为当我们的目标平台不是Android，在这个预处理指令当中写入代码，编译器会默认当前为假。就不会对这里面代码提供联想，以及其余功能。如果我们切换了目标平台，这样的话我们就能在预处理指令的代码块中，继续使用VS的提供的强大功能。接着在这个预处理指令中写入我们的控制代码。

首先第一步，获取玩家滑动触摸的2个点的坐标。如下：

|  |
| --- |
| #if UNITY\_ANDROID  MoveIncrements = transform.forward \* moveSpeed \* Time.deltaTime;  MoveIncrements += transform.right \* transverseSpeed \* Time.deltaTime \* Input.acceleration.x; //获取重力感应数值  if (Input.touchCount == 1)  {  if (Input.touches[0].phase == TouchPhase.Began)  {  startPos = Input.touches[0].position;  }  if (Input.touches[0].phase == TouchPhase.Ended && Input.touches[0].phase != TouchPhase.Canceled)  {  EndPos = Input.touches[0].position;  isInput = true; //更新触摸状态  }  } |

上面我们使用了Touch类，来进行移动端的触摸操作控制。其中Input.touchCount指的是触摸数量，即当前有几根手指在进行触摸操作。Input.touches[0].phase指的是获取当前帧，第一个手指与屏幕的触摸点的触摸状态。我们通过当前点的触摸状态来确定滑动的起始点与滑动的结束点。

第二步,通过获得的滑动向量点乘计算。如下：

|  |
| --- |
| if (isInput)  {  Vector2 nowDir = EndPos - startPos; //计算手指滑动之间的向量  float cosValueX = Vector3.Dot(nowDir, ScreenAixX) / nowDir.magnitude \* ScreenAixX.magnitude; //与屏幕坐标的X轴的余弦值  float cosValueY = Vector3.Dot(nowDir, ScreenAixY) / nowDir.magnitude \* ScreenAixY.magnitude; //与屏幕坐标的Y轴余弦值  angle = Mathf.Acos(cosValueY) \* Mathf.Rad2Deg; //通过余弦值计算角度 |

在上面，我们通过向量的点乘公式与点乘运算方法—Vector3.Dot计算处了滑动向量与XY轴之间夹角的余弦值。还将与Y轴之间的夹角计算出来，用于后续计算。

第三步，通过余弦值与夹角区分操作，先进行2，3象限中的区分。代码如下：

|  |
| --- |
| if (cosValueX < 0)  {  if (angle > 45 && angle < 135)  {  if (isTurnleftEnd)  {  isTurnleftEnd = false; //更新转向状态  transform.Rotate(Vector3.up, -90);  Quaternion tmpQuaternion = transform.rotation; //计算转向后的四元数并保存  transform.Rotate(Vector3.up, 90); //角度回滚  Tween tween = transform.DORotateQuaternion(tmpQuaternion, 0.3f); //使用DoTween插件进行转向的平滑运动  tween.OnComplete(() => isTurnleftEnd = true); //动画结束后转向状态更新  }  }  else if (angle >= 0 && angle <= 45)  {  if (playController.isGrounded)  {  isJumpState = true; //更新跳跃状态  playAnimtor.SetBool("IsJump", true); //播放跳跃动画  }  }  else if (angle >= 135 && angle <= 180)  {  playAnimtor.SetBool("IsSlide", true);  }  } |

由于余弦函数的特性，在0-90为正，90-180为负数。这样就可以在不用取度数值的情况下，通过与X轴的余弦值确定当前向量的分布范围是在2，3象限。还是1，4象限。然后我们在这个范围里面通过与Y轴夹角在进行细分，通过之前制定好的规则来进行区分当前滑动是跳跃是左转还是下滑。

第4步，1，4象限区分。代码如下：

|  |
| --- |
| else  {  if (angle > 45 && angle < 135)  {  if (isTurnRightEnd)  {  isTurnRightEnd = false;  transform.Rotate(Vector3.up, 90);  Quaternion tmpQuaternion = transform.rotation;  transform.Rotate(Vector3.up, -90);  Tween tween = transform.DORotateQuaternion(tmpQuaternion, 0.3f);  tween.OnComplete(() => isTurnRightEnd = true);  }  Debug.Log("右转" + " " + angle);  }  else if (angle >= 0 && angle <= 45)  {  if (angle > 0)  {  if (playController.isGrounded)  {  isJumpState = true; //更新跳跃状态  playAnimtor.SetBool("IsJump", true); //播放跳跃动画  }  Debug.Log("跳跃" + " " + angle);  }  else if (angle >= 135 && angle <= 180)  {  playAnimtor.SetBool("IsSlide", true);  Debug.Log("下蹲" + " " + angle);  }  }  }  isInput = false; //更新触摸状态  }  #endif |

1，4象限与X轴的夹角小于90度，余弦函数的值大于0。这样我们就完成了游戏在触摸屏上的操作代码。

接下来就可以通过之前打包与发布游戏章节提供的方法，将我们的游戏打包至安卓手机上运行游玩了。