# 课程设计概述

## 设计内容

1. 基于Unity和Kinect开发水果忍者游戏，使用Unity完成游戏画面设计，实现使用手势控制游戏开始和退出以及左右手切水果等功能。

2. 基于Unity和Kinect完成跑酷游戏体感控制。在已有跑酷游戏的基础上，添加Kinect交互功能，实现左右挥手、下蹲和跳跃来控制游戏角色向左向右、下蹲和跳跃。

## 开发工具及运行环境

### 开发工具和环境

Unity3D 2017、Visual Studio 2017、Kinect 2.0、Windows11

### 开发语言

本项目采用C#进行开发。

# 系统整体方案设计

## 水果忍者整体方案设计

水果忍者包括开始界面、游戏界面和结束界面，其设计流程如图1所示。

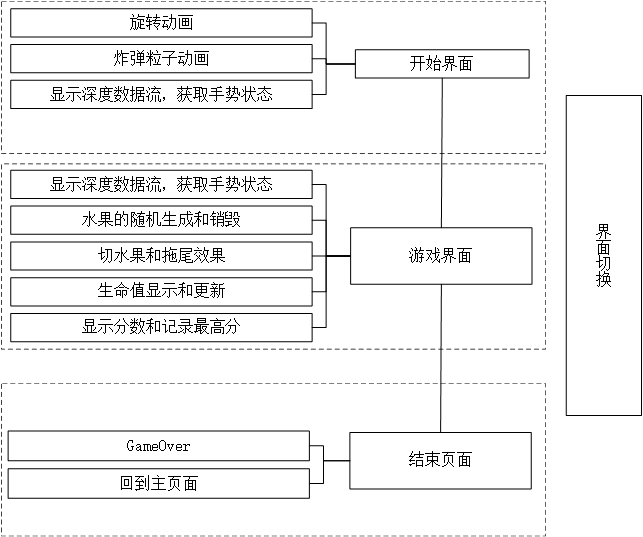


图1 水果忍者流程图

## 跑酷游戏整体方案设计

跑酷游戏包括开始界面、游戏界面、暂停界面和结束界面，其设计流程如图2所示。

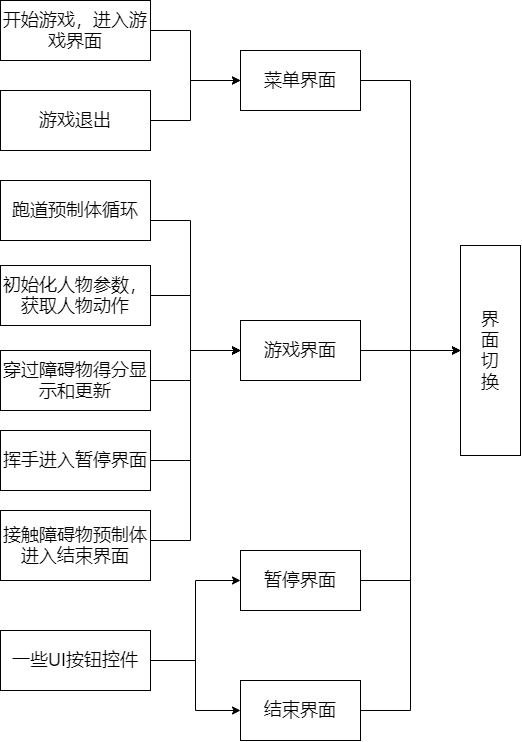


图2 跑酷游戏流程图

# 设计原理

## 获取并显示深度数据流

在本项目中，UsersLblTex是一个纹理（Texture2D）类对象，用于存储和展示深度图像，以实时显示捕获的用户和环境的深度信息。

Kinect 使用时间飞行（ToF）技术生成整个场景的深度图。该技术通过向场景发射红外光脉冲，并测量光波从发射到被场景中的对象反射回传感器所需的时间来工作。反射时间与光速成比例，光速是已知常数，因此可以直接计算出光波行进的距离，进而得到物体的深度信息。为了从深度图中区分不同的用户，Kinect 进一步处理深度数据，使用用户索引图来标识图像中的不同用户。每个用户被分配一个唯一的索引，该索引与深度值结合，形成了可以用于进一步处理的深度-用户图。

为了将这些信息可视化，UsersLblTex通过用遍历所有深度数据点，计算一个直方图，该直方图基于深度值的频率分布对颜色进行映射。这种映射不仅区分了不同的深度级别，还能通过不同的颜色区分不同的用户。例如，第一个用户可能被映射为蓝色，第二个用户为绿色，背景等可能为黑色。

将其应用到 GUI 组件上，使得用户可以实时看到深度感测的结果，如用户的位置、移动和其他交互动作。

## 获取右手关节位置并操控图标实现鼠标模拟

系统检查是否有用户被Kinect设备检测到，如果检测到用户，系统将获取当前活动用户的唯一标识符（UserId）。系统获取右手在三维空间中的坐标，并将其转换为屏幕坐标，再转换为用户界面的本地坐标，以便在界面上定位图标。此外，根据手的开闭状态，更新图标的视觉表现，闭合状态下图标更换为闭合手的图标，为用户提供即时的视觉反馈。实现流程如下：

如果 Kinect 检测到用户:

获取用户的ID

设置要跟踪的关节为右手

如果右手关节被跟踪:

获取右手关节在世界坐标中的位置

将世界坐标转换为屏幕坐标

将屏幕坐标转换为2D平面坐标

如果右手在画布内:

将2D坐标映射到图标上

设置图标的位置为右手的位置

设置图标的初始状态为“开”

获取右手的状态（如开或闭）

如果右手状态为闭:

改变图标状态为“闭”

标记为“选中状态”

## 水果忍者设计原理

### 开始界面

#### 旋转动画设计

在Unity中使用Animator组件，设计旋转动画的步骤如下：

1. 在Project窗口中创建一个新的动画剪辑文件，在Inspector窗口中添加Animator组件，并将动画剪辑拖到Animator组件的Controller字段。
2. 打开Animation窗口，选择包含Animator组件的GameObject，进入录制模式。在时间轴上选择一个时间点，设置旋转角度。然后，移动时间轴上的滑块到另一个时间点，设置目标旋转角度。Unity会在这些时间点自动创建关键帧，并插入中间帧，使旋转动画变得平滑。
3. 在Animation窗口中切换到Curves模式，查看和编辑动画曲线。通过调整曲线的形状，改变旋转的速度和加速度，在本项目中，我们设置旋转速度不变。

图形用户界面, 应用程序, Word

描述已自动生成

图3 Animator组件示意图

#### 炸弹粒子特效

在Unity中制作粒子特效主要通过Particle System组件来实现，步骤如下：

1. 在Hierarchy窗口中创建一个新的Particle System。
2. 在Inspector窗口中配置发射器，定义粒子的生成方式和数量，例如设置发射率、发射形状和发射方向等参数。
3. 调整粒子的生命周期、速度、大小、旋转和颜色等属性，通过这些参数可以实现各种不同的粒子效果，如烟雾、火焰和爆炸等。

Particle System提供了多个模块，如Shape模块控制粒子的发射形状，Velocity over Lifetime模块控制粒子在生命周期内的速度变化，Color over Lifetime模块控制粒子的颜色变化等。为了丰富粒子的外观，可以在Renderer模块中为粒子添加纹理和材质，使其看起来更加真实。

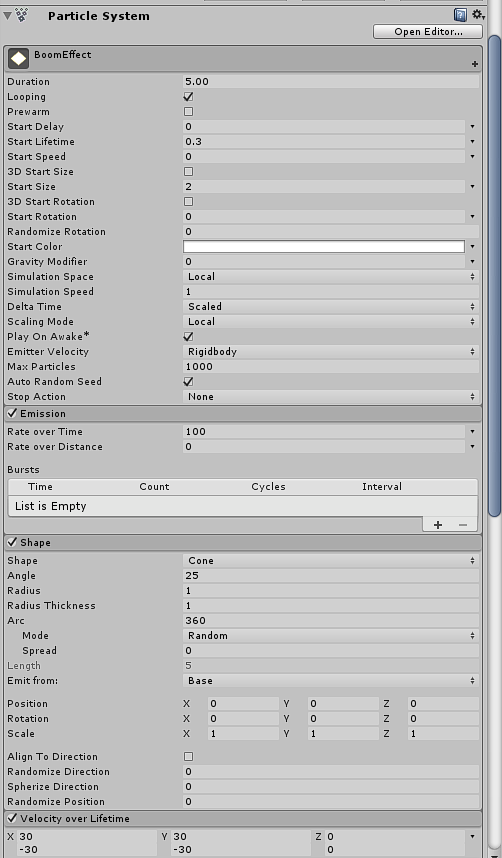


图4 Particle System 模块

#### 游戏开始

通过上文实现深度数据流显示和右手控制鼠标后，当图标位置在New Game框里面，并且图标处于选中状态后，游戏开始，开始界面销毁，创建游戏界面。

### 游戏界面

* + - 1. 水果生成和销毁

水果对象的初始化位置通过将其3D锚点位置置为原点来确保。通过在一个预定义范围内随机生成X坐标，并结合一个固定的负Y值，水果能够在屏幕上的指定区域内生成。

水果类型的选择通过数组对水果类的索引实现，例如苹果、香蕉等，并从中随机选择一个元素作为当前生成的水果类型。

水果的动态行为通过向其Rigidbody2D组件添加力实现。具体的力的方向和大小根据水果的X坐标的符号来确定，以模拟水果在空中的抛物线移动。这种物理效果不仅增加了视觉上的动感，还提升了玩家的游戏体验。

在游戏中，本项目设计了一个碰撞域，位于界面底部，当水果没有被切到时，水果与碰撞域产生碰撞，进行销毁操作，确保内存回收。

* + - 1. 切水果处理

在上文中获取了关节位置坐标后，当左右手与水果位置重合，便视作切水果，同时给左右手的运动赋予拖尾效果，提升视觉体验。

切中水果后涉及创建水果的左右两个部分，并设置相应的物理和视觉属性。通过复制预设的水果对象生成左右两个部分。物理动态通过为左部分添加向左的力，为右部分添加向右的力，同时两者都受到向上的力，模拟切割动作产生的动态效果。这种设计不仅使得水果在被切割时能够表现出真实的物理反应，增强游戏的互动性和视觉吸引力，还允许通过控制力的大小和方向调整切割后水果部分的移动速度和方向，提供更加丰富和逼真的游戏体验。

* + - 1. 生命值和分数的更新显示

通过生命值类创建生命值对象，并通过Text组件创建计分器，在游戏过程中对生命值和分数进行实时更新，本文对生命值和分数的更新规则如下：

1. 切中水果+1分，并通过PlayerPrefs.HasKey在本地用键值对的形式保存最高分。
2. 没切中水果-1生命值
3. 切中炸弹游戏结束

### 结束界面

当生命值为0，游戏结束，出现GameOver界面，并在两秒后自动跳转至开始界面。

## 跑酷游戏设计原理

### 资源导入与动画设计

首先我们从Unity AssetStore导入游戏人物资源包（Character Pack: Free Sample）以及跑道和装饰物资源包（Voxel Universe Free），结合其提供的基础动画设计跑酷Menu界面的人物动画，我们在其基础上使用Unity的Animator组件来控制人物的动画状态和转换，分别建立了wave（挥手）、HANDUP00\_R（挥右手向前跑）、RUN00\_F（向前跑）、WIN00（挥手+跳跃）、UMATOBI00（跳跃）五种动画状态，并将其用Transition（平滑过渡）连接起来，并沿着一个方向无条件进行循环，这样就能实现进入Menu菜单界面后五个动作的循环连续播放。

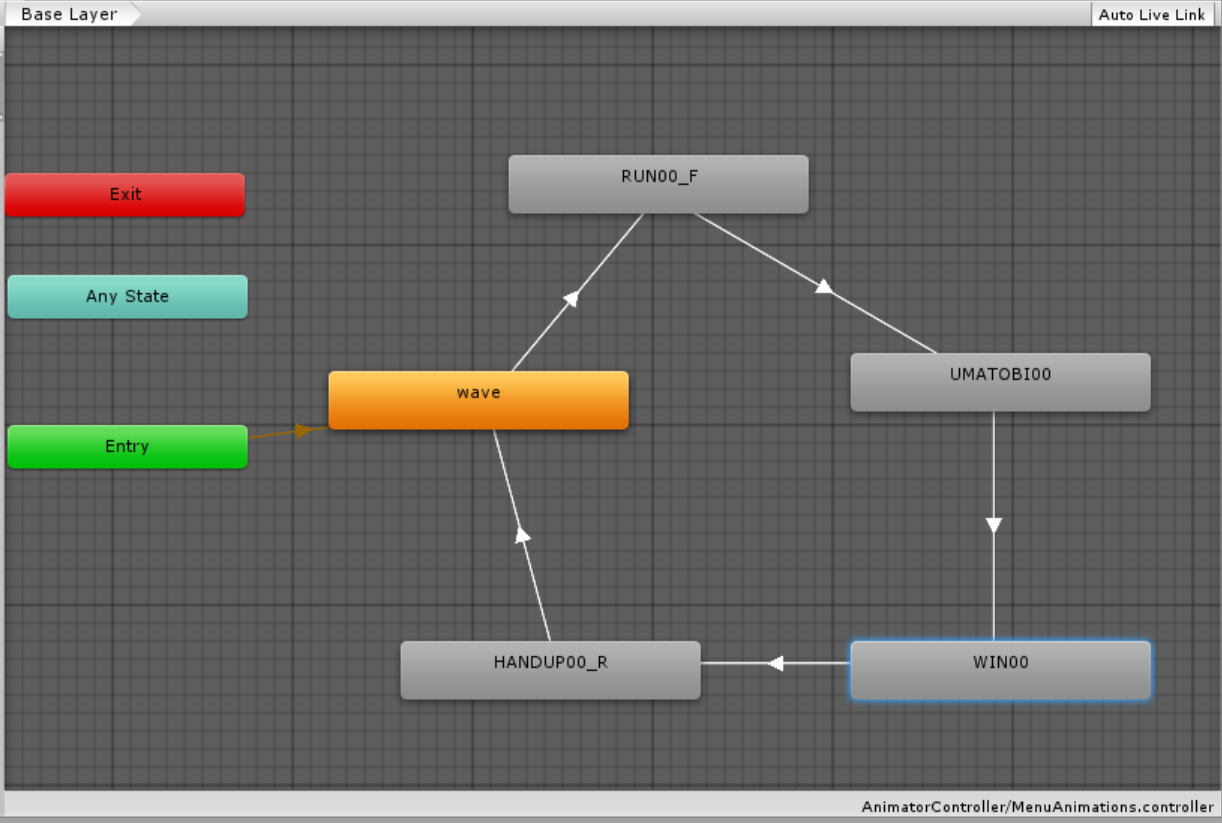


图5 Menu动画示意图

此外，我们还对游戏界面的人物动画进行了设计，与Menu界面类似，我们使用了Animator组件来控制动画状态的转换，分别建立了junp-down、jump-float、jump-up、walk、run等十一种动画状态，并利用布尔值Ground来判断是否落地从而实现jump类的动画与run类的动画的交互，值得一提的是，我们采用了Blend Tree（混合树）来控制不同速度的动画间的过渡，能够让角色在游戏中根据速度进行实时调整，让动画效果看起来更加逼真。

图片包含 室内, 飞行, 飞机, 不同

描述已自动生成

图6 游戏人物动画示意图

### Menu界面的鼠标控制与功能选择

通过上文实现右手控制鼠标后，我们在Menu界面放置了三个图标框，分别为“Start”、“Exit”、“Introduction”，我们利用Unity中的RectTransformUtility.RectangleContainsScreenPoint API来获取鼠标图标是否在功能图标框里，并从Kinect管理器获取当前用户右手的状态，如果是握拳则执行相应功能的函数，进入相应的界面。

### PlayerController脚本

该脚本用于控制游戏中角色的行为和动画，通过Awake方法中获取玩家角色的Rigidbody组件、游戏控制器、动画控制器等组件的引用，可用于后续代码中控制玩家角色的运动和状态。

1. 移动和跳跃

根据玩家角色的当前位置和目标位置，通过插值方法逐步将玩家角色移动到目标位置，这样可以保证这样可以确保无论帧率是多少，角色每秒移动的距离都是一样的，从而实现了匀速移动的效果。同时，如果Kinect检测到跳跃手势，并且角色在地面上，则给角色施加一个向上的力，实现跳跃。

1. 碰撞检测

通过Linecast方法检测角色的脚部是否与地面接触，从而确定角色是否在地面上。如果角色在地面上，则可以执行跳跃和下蹲动作。另外，还通过OnTriggerEnter方法检测是否与障碍物发生碰撞，如果碰撞到障碍物，则游戏结束（若角色是下蹲状态，调整某些角色的碰撞体，不触发碰撞事件）。

1. 动画控制

根据角色的状态（是否在地面上、是否进行了下蹲动作等），更新动画状态机中的参数，以控制角色的动画。

### GameController脚本

该脚本用于管理游戏的流程，包括游戏的暂停、继续。

定义了Pause、Continue等方法，用于控制游戏的暂停、继续，如果检测到挥手手势，则切换游戏的状态。在暂停游戏时，调用Pause方法将Unity控制游戏时间流逝的缩放因子设为0，同时加入暂停页面；在继续游戏时，调用Continue方法将缩放因子设为1，并退出暂停页面。

### TrackController脚本

该脚本用来控制游戏中跑道的行为和状态，实现跑步的效果。具体实现如下：

先初始化跑道的速度，并定义了Stop和Continue方法，用于在游戏暂停和继续时控制跑道的运动状态。在暂停时，将当前速度设置为0，停止跑道运动；在继续时，将当前速度设置为之前保存的速度，恢复跑道运动。跑道Roll类用于处理道路滚动的逻辑。它在 Update方法中更新道路的位置，使其沿 x 轴方向移动，当超出设定的道路长度时销毁当前对象。同时，在当前预制体末端重新创建新的道路。

### ScoreController脚本

1. 得分更新

在Update方法中，检查当前得分和上一次更新时的得分是否相等。如果不相等，表示得分发生了变化，更新preScore为当前得分，并将当前得分显示在UI文本中，这样做是为了避免每一帧都去更新UI文本，提高性能。

1. 得分增加

定义了AddScore方法，用于在游戏中增加得分。当调用这个方法时，将score增加传入的scoreValue，表示增加相应的得分。道具props类可用于处理玩家与游戏中的道具发生碰撞时的逻辑，当玩家与道具发生碰撞时，道具对象会被销毁，并调用ScoreController的AddScore方法，将道具的得分值添加到游戏得分中。

# 核心代码

## 获取深度数据流并显示

1. if(KinectImage!=null && KinectImage.texture == null)
2. {
3. Texture2D usermap = KinectManager\_fruit.Instance.GetUsersLblTex();//获得深度数据信息(检测到人）
4. KinectImage.texture = usermap;//把kinect获得到的人的图像放到ui里面
5. }

## 获取右手关节位置并用右手控制图标

1. //获取用户id
2. **long** UserId = KinectManager\_fruit.Instance.GetPrimaryUserID();
3. //获取右手关节
4. **int** JType = (**int**)KinectInterop.JointType.HandRight;
5. //追踪关节位置
6. **if** (KinectManager\_fruit.Instance.IsJointTracked(UserId, JType))
7. {
8. Vector3 rightjointpos= KinectManager\_fruit.Instance.GetJointKinectPosition(UserId, JType);//世界坐标
9. Vector3 screenpos = Camera.main.WorldToScreenPoint(rightjointpos);//世界坐标-相机坐标
10. Vector2 screenpos2D = **new** Vector2(screenpos.x, screenpos.y);//3D-2D 手的相机坐标
11. Vector2 GUIPOS;//相机坐标-平面坐标
12. //手在不在画布里
13. **if**(RectTransformUtility.ScreenPointToLocalPointInRectangle(Canv, screenpos2D, Camera.main, **out** GUIPOS))
14. {
15. RectTransform HandRTF = Hand.transform **as** RectTransform;
17. HandRTF.anchoredPosition = GUIPOS;//手坐标赋给图标
18. }
19. Hand.sprite = HandState[0];//初始右手图标为打开

22. **bool** isChoosed = **false**;//初始状态未选中
23. KinectInterop.HandState RightHandState=KinectManager\_fruit.Instance.GetRightHandState(UserId);//右手状态
24. **if**(RightHandState== KinectInterop.HandState.Closed)
25. {
26. Hand.sprite = HandState[1];
27. isChoosed = **true**;
28. }

## 水果忍者核心代码

### 界面切换

1. **private void Start()**
2. {
3. CreateStartPanel();
4. }
6. //创建开始面板
7. **public** **void** CreateStartPanel()
8. {
9. RectTransform startPanel = Instantiate(startPanelPrefab);
10. startPanel.SetParent(transform);
11. startPanel.offsetMin = Vector2.zero;
12. startPanel.offsetMax = Vector2.zero;
13. startPanel.anchoredPosition3D = Vector3.zero;
14. startPanel.localScale = Vector3.one;
15. }
16. //创建游戏面板
17. **public** **void** CreateGamePanel()
18. {
19. RectTransform gamePanel = Instantiate(gamePanelPrefab);
20. gamePanel.SetParent(transform);
21. gamePanel.offsetMin = Vector2.zero;
22. gamePanel.offsetMax = Vector2.zero;
23. gamePanel.anchoredPosition3D = Vector3.zero;
24. gamePanel.localScale = Vector3.one;
25. }

### 水果生成和销毁

1. //水果生成
2. **void** CreateFruit()
3. {
4. **if** (LifeNum > 0)
5. {
6. CurFruit = Instantiate(FruitPrefab);//克隆水果
7. CurFruit.transform.SetParent(transform);//将水果的父类设置为当前界面对象
9. //水果出现
10. CurFruit.GetComponent<RectTransform>().anchoredPosition3D = Vector3.zero;
11. **int** fruitX = Random.Range(fruitXpostion, fruitXpostion);
12. CurFruit.GetComponent<RectTransform>().anchoredPosition = **new** Vector2(fruitX, -fruitYpostion);
13. CurFruit.transform.localScale = Vector3.one;
14. **int**[] types = { FruitBoomR.BOOM, FruitBoomR.APPLE, FruitBoomR.BANANA, FruitBoomR.CAOMEI, FruitBoomR.PEACH, FruitBoomR.WATERMELON };
15. **int** FruitType = types[Random.Range(0, types.Length)];
16. CurFruit.SetType(FruitType);
17. **if** (FruitType == FruitBoomR.BOOM)
18. {
19. CurFruit.transform.Find("BoomEffect").gameObject.SetActive(**true**);
20. }
21. **else**
22. {
23. CurFruit.transform.Find("BoomEffect").gameObject.SetActive(**false**);
24. }
25. //水果上抛
26. Rigidbody2D rig = CurFruit.GetComponent<Rigidbody2D>();
27. **if** (fruitX > 0)
28. {
29. rig.AddForce(**new** Vector2(-forceX, forceY));
30. }
31. **else**
32. {
33. rig.AddForce(**new** Vector2(forceX, forceY));
34. }
35. audioSource.PlayOneShot(create);
36. }
37. }
38. //水果销毁
39. **if** (CurFruit.GetComponent<RectTransform>().anchoredPosition.y < -300)
40. {
41. Destroy(CurFruit.gameObject);
43. **if** (CurFruit.type != FruitBoomR.BOOM)
44. {
45. LifeNum--;
46. }
47. CreateFruit();
48. }

### 切水果处理

1. //切到水果处理
2. **void** cut()
3. {
4. **if** (CurFruit.type == FruitBoomR.BOOM)
5. {
6. Destroy(CurFruit.gameObject);
7. CreateFruit();
8. //碰到炸弹做相应处理
9. LifeNum = 0;
10. audioSource.PlayOneShot(boom);
11. }
12. **else**
13. {
14. ScoreNum++;
15. **if** (!PlayerPrefs.HasKey("maxScore") || ScoreNum > PlayerPrefs.GetInt("maxScore"))
16. {
17. PlayerPrefs.SetInt("maxScore",ScoreNum);
18. }
20. ProcessText();
21. cuttedFruit();
22. Destroy(CurFruit.gameObject);
23. audioSource.PlayOneShot(cutfruit);
24. CreateFruit();
26. }
27. }
28. //生成两半的水果
29. **void** cuttedFruit()
30. {
31. Fruit Leftonef = Instantiate(FruitPrefab);
32. Leftonef.SetType(CurFruit.type + 1);
33. Leftonef.transform.Find("FruitEffect").gameObject.SetActive(**true**);
34. initCuttedFruit(Leftonef, **true**);
35. Fruit Rightonef = Instantiate(FruitPrefab);
36. Rightonef.SetType(CurFruit.type + 2);
37. Rightonef.transform.Find("FruitEffect").gameObject.SetActive(**true**);
38. initCuttedFruit(Rightonef, **false**);
39. }
41. /\*
42. fuction:初始化切成两半的水果 初始化位置、施加两边力
43. fruit:切了之后的水果 左边一半 右边一半
44. isleft:是否是左边一半
45. \*/
46. **void** initCuttedFruit(Fruit fruit,**bool** isleft)
47. {
48. fruit.transform.SetParent(transform);
49. RectTransform rt = fruit.GetComponent<RectTransform>();
50. rt.anchoredPosition3D = Vector3.zero;
51. rt.anchoredPosition = CurFruit.GetComponent<RectTransform>().anchoredPosition;
52. rt.localScale = Vector3.one;
53. **if** (isleft)
54. {
55. fruit.GetComponent<Rigidbody2D>().AddForce(**new** Vector2(-forceX, cutForceY));//两半的水果施加加速度让其往两边走
57. }
58. **else**
59. {
60. fruit.GetComponent<Rigidbody2D>().AddForce(**new** Vector2(forceX, cutForceY));//两半的水果施加加速度让其往两边走
61. }
63. }

### 分数和生命值

生命值类：

1. **public** **class** Life : MonoBehaviour
2. {
3. **public** Toggle[] life;
4. **public** **void** SetLife(**int** LifeNum)//0
5. {
6. **int** MaxLife = life.Length;//3
7. **int** i = 0;
8. **while**(i < (MaxLife-LifeNum))//0-3 0 1 2
9. {
10. life[i].isOn = **false**;
11. i++;
12. }
13. }
14. }

最高分记录和显示：

1. **if** (!PlayerPrefs.HasKey("maxScore") || ScoreNum > PlayerPrefs.GetInt("maxScore"))
2. {
3. PlayerPrefs.SetInt("maxScore",ScoreNum);
4. }
5. **if** (PlayerPrefs.HasKey("maxScore"))
6. {
7. maxNum = PlayerPrefs.GetInt("maxScore");
9. }
10. maxText.text = "最高分：" + maxNum.ToString();

## 跑酷游戏核心代码

### 游戏进程控制

下面代码实现了游戏界面的手势控制功能，包括暂停和继续。通过检测挥手姿势和 Psi 手势实现相应的功能：当游戏运行时，如果检测到用户进行挥手姿势，则游戏将暂停，再次挥动则继续游戏，代码如下：

1. **void** Update()
2. {
3. **if** (!gestureListener)
4. **return**;
6. **if** (gestureListener.IsWave())
7. {
8. **if** (!isPause)
9. {
10. Pause();
11. isPause = **true**;
12. }
13. **else**
14. {
15. Continue();
16. isPause = **false**;
17. }
19. }
20. **else** **if**(gestureListener.IsPsi())
21. {
22. Exit();
23. }
25. //是否检测到用户

28. }

在暂停功能中，游戏的时间尺度被设置为 0，轨道控制器停止轨道的移动，同时暂停菜单界面被激活显示；而在继续功能中，游戏的时间尺度被设置为 1，轨道控制器继续轨道的移动，同时暂停菜单界面被隐藏。若检测到用户进行 Psi 手势（双手过肩1秒），则游戏将退出，代码如下：

1. //游戏暂停调研的函数
2. **public** **void** Pause()
3. {
4. Time.timeScale = 0;
6. trackCtrl.Stop();
8. pausedMenu.SetActive(**true**);
9. }
11. //游戏继续调研的函数
12. **public** **void** Continue()
13. {
14. Time.timeScale = 1;
16. trackCtrl.Continue();
18. pausedMenu.SetActive(**false**);
19. }
21. **public** **void** Exit() {
22. Application.Quit();
23. }

### 角色初始化

下面代码片段实现了对角色的初始化操作，包括给角色添加重力以及初始化角色在 Z 方向上的前进速度。具体而言，首先通过 body.AddForce 方法给角色施加重力，这是通过向下的一个力来模拟重力效果，gravity 是重力的大小。然后，通过 Mathf.MoveTowards 方法计算角色在 Z 方向上移动的目标位置 zPos，目标位置是当前位置向目标位置 target 移动的一小步，步长由 changeTrackSpeed 乘以 Time.deltaTime 控制。接着，通过 transform.position 来更新角色的位置，将当前位置的 x 和 y 分量保持不变，只更新 z 分量为目标位置 zPos，从而实现角色在 Z 方向上的移动。最后，通过 transform.Translate 方法使角色在 Z 方向上以 forwardSpeed 的速度前进，乘以 Time.deltaTime 以确保在不同帧之间移动的平滑性，代码如下：

1. //给角色添加重力
2. body.AddForce(Vector3.down \* gravity);
4. **float** zPos = Mathf.MoveTowards(transform.position.z, target, changeTrackSpeed \* Time.deltaTime);
5. transform.position = **new** Vector3(
6. transform.position.x,
7. transform.position.y,
8. zPos
9. );
10. transform.Translate(Vector3.forward \* forwardSpeed \* Time.deltaTime);

### 手部控制变道

下面代码实现了游戏界面的手势控制功能，用于控制玩家在游戏中进行左右移动及变道操作。通过检测 GestureDetect 中的布尔值 swipeLeft 和 swipeRight 来判断玩家手势的方向，从而控制玩家的移动方向。如果检测到玩家进行左滑手势（swipeLeft 为真），则将输入值 input 设置为 -1，表示向左移动；而如果检测到玩家进行右滑手势（swipeRight 为真），则将输入值 input 设置为 1，表示向右移动，这将为下面控制变道的逻辑部分提供输入，代码如下：

1. //使用kinect作为输入
2. /\*input = avatarCtrl.deltaPosition();\*/
3. input = 0;
4. **if**(gestureListener.IsSwipeLeft())
5. {
6. input = -1;
7. }
8. **if**(gestureListener.IsSwipeRight())
9. {
10. input = 1;
11. }

下面的代码用于确保在 Kinect 检测到用户想要变道时（input 不等于 0），只有当玩家已经到达目标位置且上一个变道操作已完成时才能执行下一个变道操作。这通过两个布尔值 hasChangedPosition 和 arrived 进行监控实现。具体来说，当 input 不等于 0 且 hasChangedPosition 为假且已经到达目标位置时，将新的目标位置计算为当前目标位置加上 input 方向乘以跑道宽度，并使用 Mathf 库的 Clamp 函数限制目标位置在跑道范围内，然后将 hasChangedPosition 设置为 true，表示变道操作已开始。而当 input 等于 0 且 hasChangedPosition 为真时，将 hasChangedPosition 设置为 false，表示变道操作已完成。这样确保了在一次变道操作完成之前，不会执行下一次变道操作，并且变道范围受到了限制，不会移出跑道，代码如下：

1. **if** (input != 0 && !hasChangedPosition && arrived)
2. {
3. target = target + Mathf.Sign(input) \* trackWidth;
4. target = Mathf.Clamp(target, -trackWidth, trackWidth);
5. hasChangedPosition = **true**;
6. GetComponent<AudioSource>().Play();
7. }
9. **if** (input == 0 && hasChangedPosition)
10. {
12. hasChangedPosition = **false**;
13. }

### 控制角色上跳下蹲

下面代码实现了当角色脚部与地面接触且已到达目标位置时，根据 Kinect 检测到的用户动作来控制角色的跳跃和下蹲功能。具体而言，通过 avatarCtrl 组件来捕获用户的动作传输到角色模型上。如果检测到用户执行了跳跃动作（avatarCtrl.isJumping() 为真），则给角色施加一个向上的速度，速度大小为 jumpForce。如果检测到用户执行了下蹲动作（avatarCtrl.isCrouch() 为真），则调用 Squat 方法，该方法会利用触发器调整角色碰撞器的状态，使得角色进入下蹲状态，并播放相应的动画。如果没有检测到下蹲动作，则将角色的碰撞器状态设置为使其能够与其他物体碰撞，代码如下：

1. //跳跃和下蹲都只有在人物脚在平面上的时候才能做
2. //变道过程中不能跳跃或者蹲下
3. **if** (grouded && arrived)
4. {
5. //使用Kinect作为输入
6. **if** (avatarCtrl.isJumping())
7. {
8. body.velocity = **new** Vector3(0, jumpForce, 0);
9. GetComponent<AudioSource>().Play();
10. }
11. **if** (avatarCtrl.isCrouch())
12. {
13. Squat();
14. }
15. **else**
16. {
17. RunCollider.isTrigger = **true**;
18. SquatCollider.isTrigger = **false**;
19. }
20. }
21. }
23. **public** **void** Squat()
24. {
25. RunCollider.isTrigger = **false**;
26. SquatCollider.isTrigger = **true**;
27. animator.Play("SLIDE00");
28. }

上跳下蹲控制原理：

上跳：通过检测双脚的抬高以及头部抬高到一定位置来确定。具体而言，isJumping 方法中的条件是当前左脚和右脚的关节位置大于 -0.65（即相对地面的高度），同时头部关节位置大于 0.7，这表明角色的双脚已经离开地面，而头部已经抬高到一定程度，符合跳跃的动作特征。

下蹲：通过检测头部和膝盖的关节位置来确定。具体而言，isCrouch 方法中的条件是当前头部关节位置小于 0.3 且当前膝盖关节位置小于 0，这表明角色的头部已经低于一定程度，而膝盖处于弯曲状态，符合下蹲的动作特征，代码如下：

1. **public** **bool** isMoving(){
2. **return** (System.Math.Abs(currentLeftFootJointSpeed) > 0.2) && (System.Math.Abs(currentRightFootJointSpeed) > 0.2);
3. }
5. **public** **bool** isJumping(){
6. **return** currentLeftFootJointPosition > -0.65 && currentRightFootJointPosition > -0.65 && currentHeadJointPostion > 0.7;
7. }
9. **public** **bool** isCrouch(){
10. **return** currentHeadJointPostion < 0.3 && currentKneeJointPositon < 0;
11. //return true;
12. }

### 判断角色是否触地

下面代码用于根据角色是否着地来控制角色的动画状态机。具体实现是通过检测角色的中心与足部的连线上是否与地面层相交来判断是否着地。这通过 Physics.Linecast 方法实现，该方法会从角色的当前位置向足部位置发出一条射线，检测是否与地面层相交，返回一个布尔值，表示是否与地面相交。这个布尔值被赋值给 grounded 变量，然后通过 animator.SetBool 方法将 grounded 的值传递给动画状态机的 Grounded 参数，用于控制角色动画的播放状态，代码如下：

1. //是否着地来决定角色的动画状态机
2. animator.SetBool("Grounded", grouded);
3. grouded = Physics.Linecast(transform.position, foot.position,
4. 1 << LayerMask.NameToLayer("Ground"));

### 跑道预制体更新

跑道预制体在游戏中具有一定的长度，当跑道预制体的位置超过了预定的长度时，会销毁该跑道（将其 transform.position 归零），同时在新的位置实例化一个新的跑道预制体，以确保游戏中始终有新的跑道生成。

具体而言，Update 方法中首先通过将当前对象的位置向 x 轴正方向移动 trackCtrl.currentSpeed 的距离来模拟跑道的移动。然后检查当前跑道的位置是否超过了预定的长度 length，如果超过则销毁当前跑道对象。接着，通过检查是否已经创建了新的跑道（hasCreated）以及当前位置是否大于 0 来确定是否需要在新的位置创建新的跑道。如果条件满足，则调用 CreateTrack 方法来在新的位置实例化一个随机选择的跑道预制体，确保游戏中始终有新的跑道生成。代码如下：

1. **private** **void** Update() {
2. transform.position = **new** Vector3(
3. transform.position.x + trackCtrl.currentSpeed,
4. transform.position.y,
5. transform.position.z
6. );
8. **if**(transform.position.x > length) {
9. Destroy(**this**.gameObject);
10. }
12. **if**(!hasCreated && transform.position.x > 0) {
13. hasCreated = **true**;
15. CreateTrack();
16. }
17. }
19. **private** **void** CreateTrack() {
20. **int** index = Random.Range(0, tracks.Length);
21. **float** xPos = transform.position.x - length;
23. Vector3 pos = **new** Vector3(xPos, transform.position.y, transform.position.z);
25. Instantiate(tracks[index], pos, Quaternion.identity);
26. }

### 计分功能

分数的更新流程如下：

首先，在 Start 方法中，初始化 score 和 preScore 变量为 0，用于存储分数值。然后，在 Update 方法中，通过检查 preScore 是否与 score 不同来判断分数是否有更新。如果分数发生了变化，将当前分数值赋给 preScore，并将该分数值显示在 scoreText 文本框中。AddScore 方法用于增加分数，当调用该方法时，将传入的 scoreValue 加到当前分数 score 中。

最后，在 OnTriggerEnter 方法中，当检测到碰撞器进入触发器时，首先检查碰撞器是否为玩家。如果是玩家，则调用 AddScore 方法增加分数，然后销毁当前的道具对象，代码如下：

1. **private** **void** Start()
2. {
3. score = preScore = 0;
4. }
6. **private** **void** Update()
7. {
8. **if** (preScore != score)
9. {
10. preScore = score;
12. scoreText.text = "Score：" + preScore;
13. }
14. }
16. **public** **void** AddScore(**int** scoreValue)
17. {
18. score += scoreValue;
19. GetComponent<AudioSource>().Play();
20. }
22. **private** **void** OnTriggerEnter(Collider other) {
23. **if** (other.CompareTag("Player")) {
24. scoreCtrl.AddScore(scoreValue);
26. Destroy(gameObject);
27. }
28. }

# 测试结果

## 获取并显示深度数据流，右手模拟鼠标

图片包含 桌子, 男人, 女人, 标志

描述已自动生成

图7 获取并显示深度数据流，右手模拟鼠标

## 水果忍者

在水果忍者的测试中，游戏能够使用体感控制开始和切水果的操作，在游戏全程也能正常显示深度数据流，测试动图如下。

### 游戏开始

图片包含 室内, 橱柜, 小, 桌子

描述已自动生成

图8 水果忍者游戏开始

### 游戏过程

图片包含 桌子, 灯光, 小, 苹果

描述已自动生成

图9 水果忍者游戏过程

### 游戏结束

夜晚钟表

中度可信度描述已自动生成

图10 水果忍者游戏结束

## 跑酷

Menu界面的人物和游戏界面人物的动画能够流畅播放，Menu菜单界面能够正常实现Kinect检测，并能够用用户右手控制界面上的鼠标图标，也能够实现右手握拳手势检测与功能选择，基本实现菜单界面以及跳转功能的设计要求。游戏界面也能够正常运行。

### 菜单界面



图11 跑酷游戏菜单界面效果

### 游戏界面



图12 跑酷游戏控制人物跳跃



图13 跑酷游戏控制人物移动

# 遇到的主要问题及解决办法

1. **问题：**在水果忍者游戏中，本想通过调用API中的MouseController来实现鼠标的调用，但无法成功实现。

**解决方案：**用图标来模拟鼠标位置，实时将右手位置传递给图标来进行实时移动；用右手的张合模拟鼠标的选中。

1. **问题：**用了上述方案后发现图标不跟手。

**解决方案：**经查阅后发现获取关节位置需要调用GetJointKinectPosition()而不是GetJointPosition()，并且需要将主相机的位置归0。

1. **问题：**跑酷游戏中，asset store 模型下载后出现 missing prefab 报错。

**解决方案：**微软网络组件.NET 3.5 没法读取到我导入的dll文件，需要改成.NET4.0。

1. **问题：**在实现跑酷菜单界面的手、鼠标同步功能的时候遇到了鼠标无法跟着手移动的问题。

**解决方案：**经过我们的排查，发现是由于Unity中Main Camera的位置没有重置的问题，经过重置解决了问题