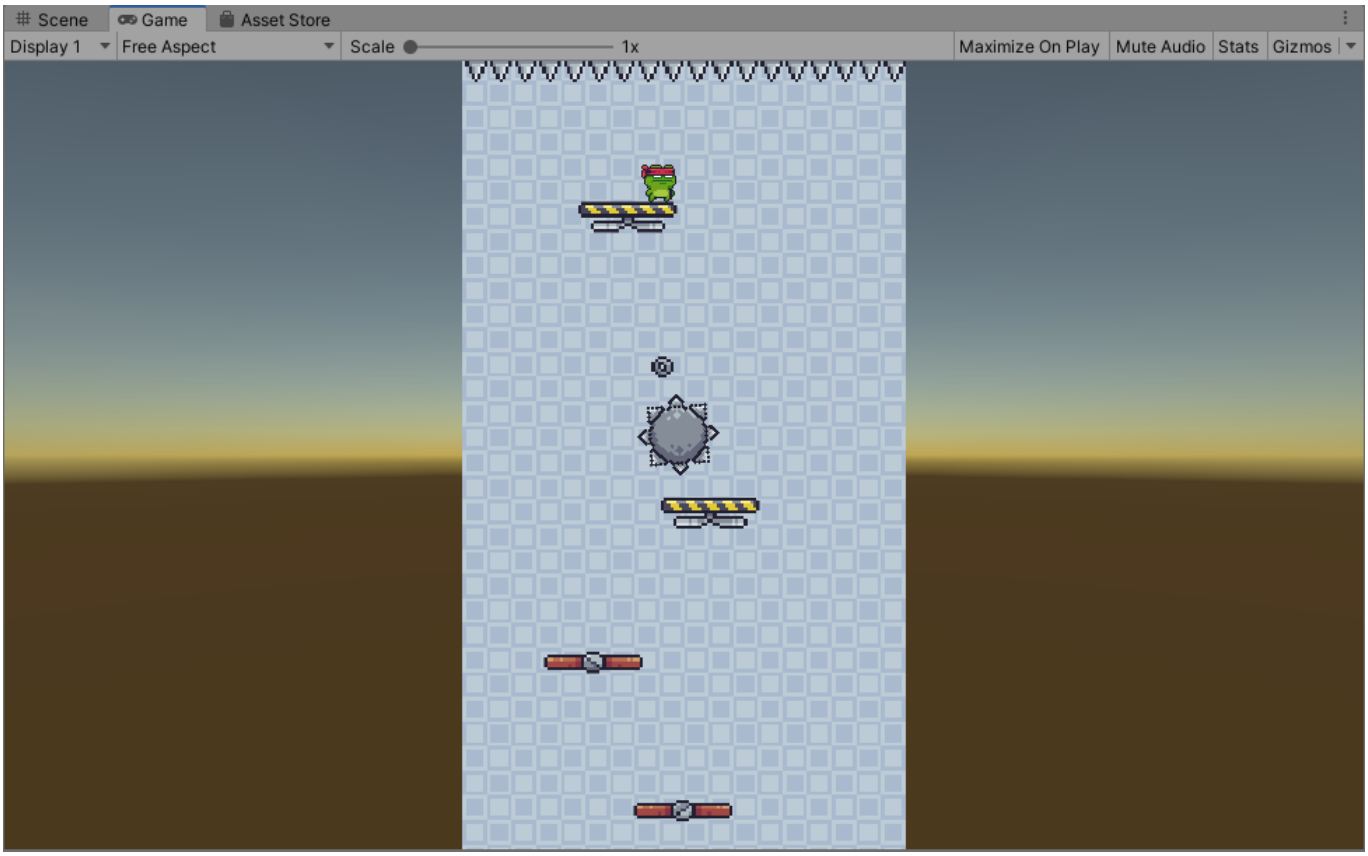
2D 游戏教程

0. 简介

本次课堂作业要实现一个2D的游戏。游戏的基本玩法是玩家通过左右移动，不断向下跳到不同平台上，躲避障碍物并避免跌落，避免被屏幕顶端的刺刺伤。游戏主要使用了 `Pixel Adventure 1` 的素材包，可以在 Asset Store 上免费下载，也可以在作业附件中下载。

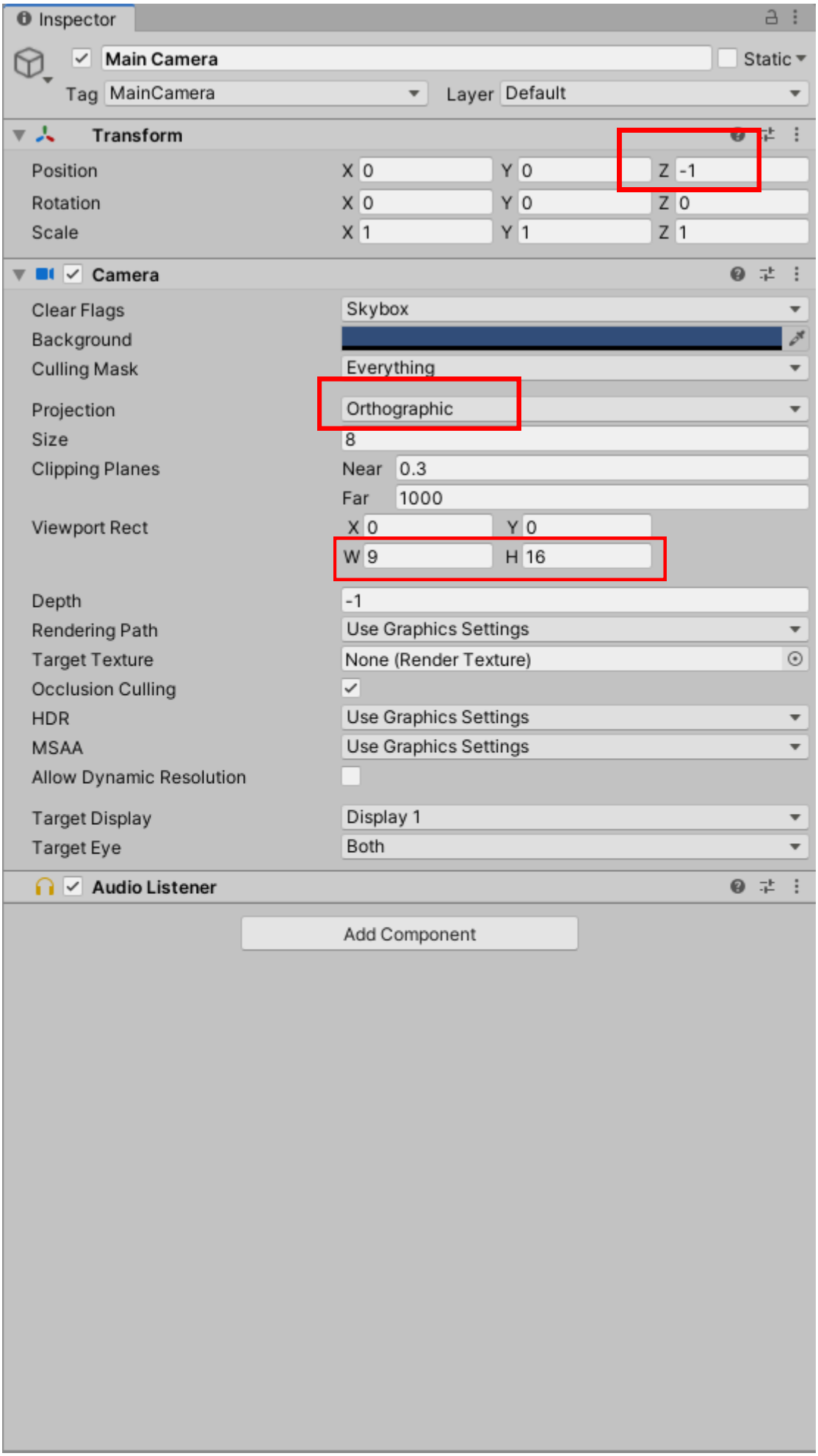
本次作业主要参考了@M_Studio制作的《Speed Down》系列视频教程，并做了一些修改以适应作业时间和难度。



1. 背景

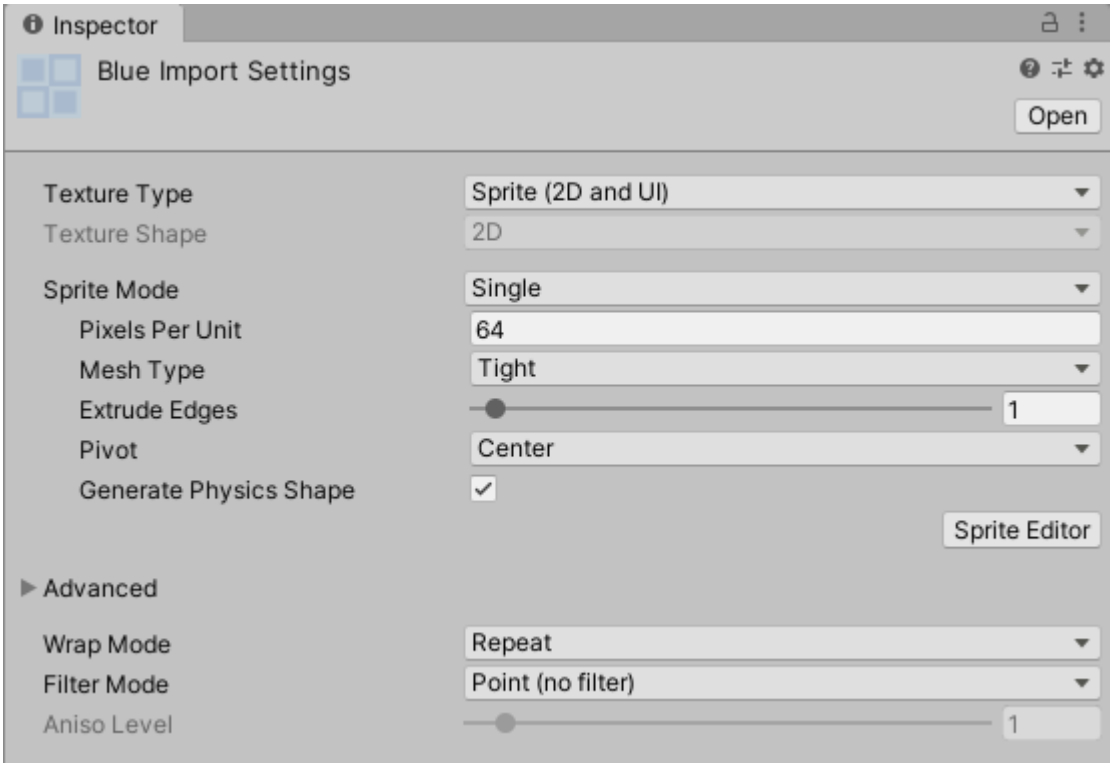
背景图片

背景设置 因为要做的是竖版的跑酷游戏，需要调整 Main Camera，使屏幕可见区域适应竖屏。如果在项目创建时，创建了2D项目，只需要设置 `Viewport Rect->w/h=9/16` 或 `10/16`；如果一开始创建的是3D项目，还需要设置 `Camera->Projection=Orthographic`，`Transform->Position->Z=-1`。


Inspector
Main Camera
Static
Tag MainCamera
Layer Default
Transform
Position X 0 Y 0 Z -1
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Camera
Clear Flags Skybox
Background
Culling Mask Everything
Projection Orthographic
Size 8
Clipping Planes Near 0.3
Far 1000
Viewport Rect X 0 Y 0
W 9 H 16
Depth -1
Rendering Path Use Graphics Settings
Target Texture None (Render Texture)
Occlusion Culling
HDR Use Graphics Settings
MSAA Use Graphics Settings
Allow Dynamic Resolution
Target Display Display 1
Target Eye Both
Audio Listener
Add Component

背景材质

在 `Pixel Adventure 1/Assets/Background/` 中选择喜欢的背景贴图，设置 `Sprite Mode->Pixels Per Unit=64` 调整它在游戏中显示的大小，`Warp Mode=Repeat` 使之能以平铺方式覆盖背景。

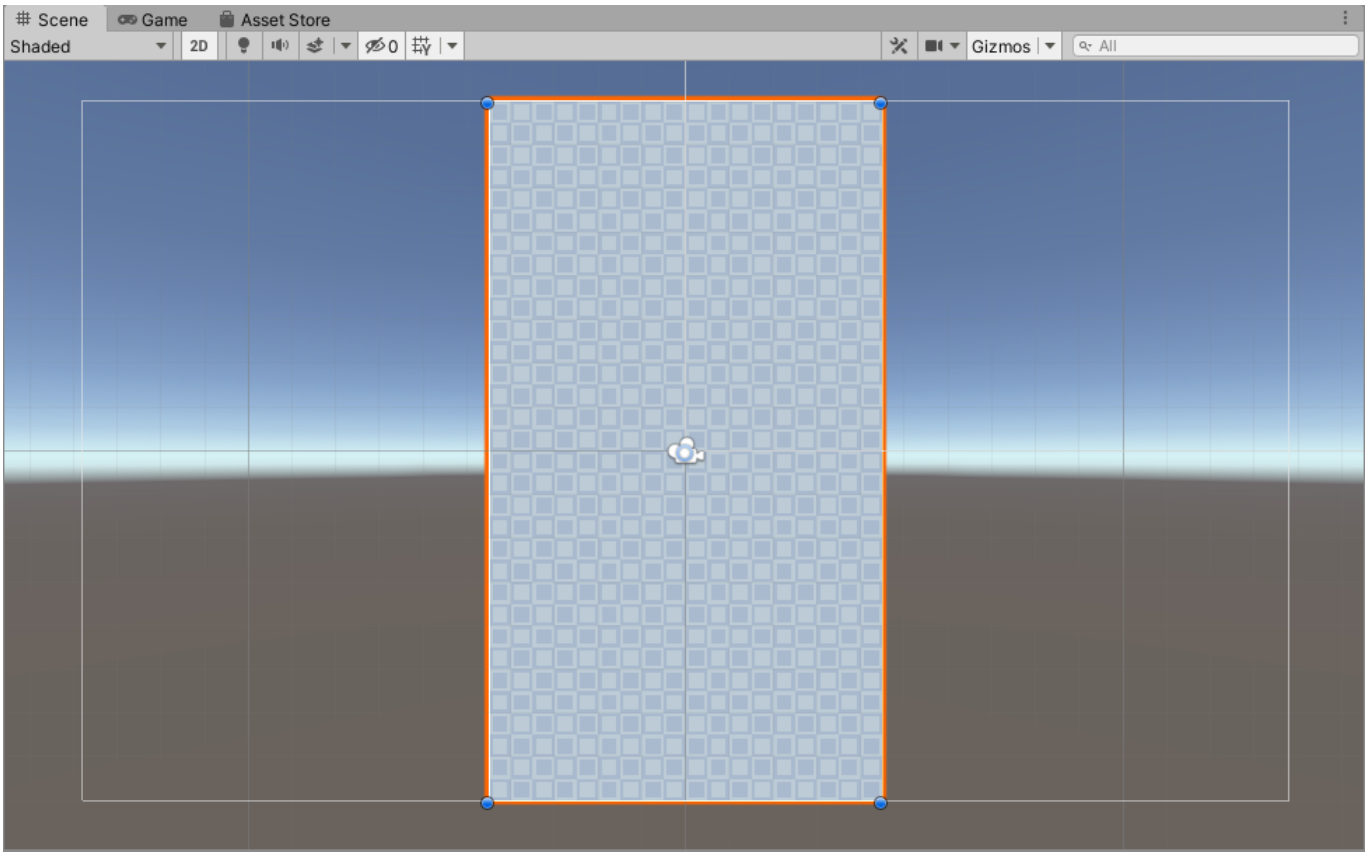


场景中新建 `Quad`，名为 Background，来表示背景，其中 `Transform->Scale=(9,16,1)` 以充满视窗。

文件夹中新建 `Material`，名为 BackgroundMat，并选为 `Unlit/Texture` 形式，选择之前的背景贴图作为材质的 `Base(RGB)`，并设置 `Tiling=(9,16)` 以满足背景比例。将材质 BackgroundMat 赋给 Background 。

测试

运行游戏，可以看到背景贴图，调整 BackgroundMat 中的 `Offset` 即为调整贴图的UV坐标，可以看到背景贴图的平移。如果你的背景贴图不是如下图所示填满方框，**那么检查Background的`Transform->Position`的z值是否为-1.**



背景运动

为了更好地实现背景卷轴化的运动，在 Background 下新建脚本。脚本内获取当前物体上的材质，并修改材质中贴图的UV坐标偏移量 `material.mainTextureOffset` 来实现背景移动。代码如下：

```
public Vector2 speed;
Material material;
Vector2 movement;

// Start is called before the first frame update
void Start()
{
    material = GetComponent<Renderer>().material;
}

// Update is called once per frame
void Update()
{
    movement+=speed;
    material.mainTextureOffset = movement;
}
```

测试

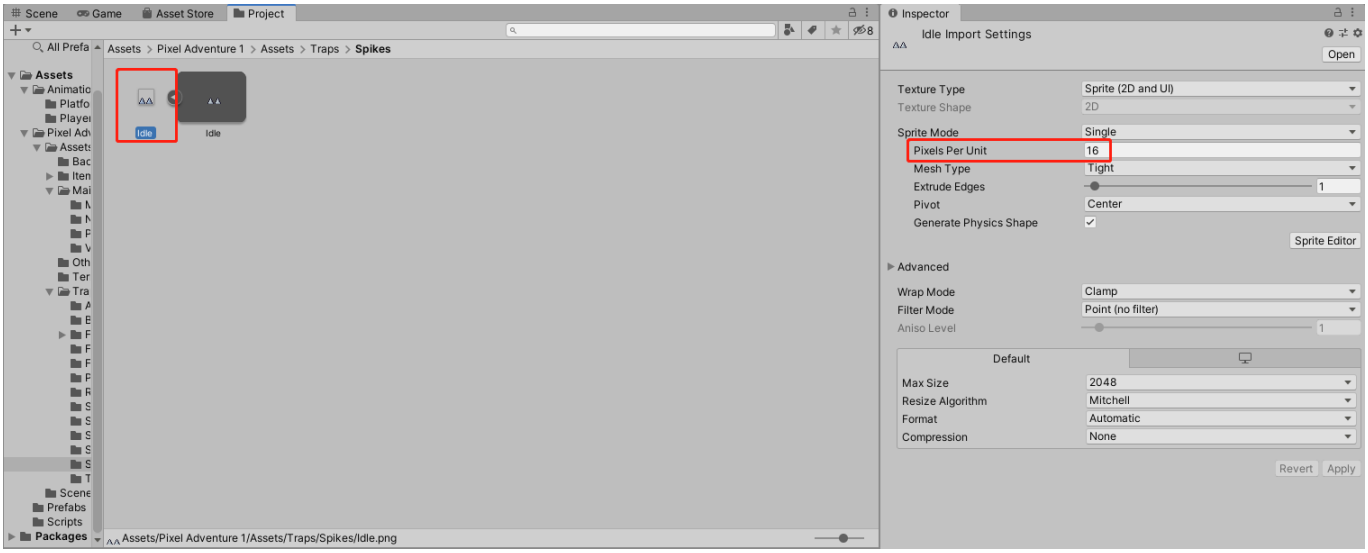
在 Inspector 中设置 Background 下脚本的 `speed` ，运行游戏，可以看到背景贴图，调整 BackgroundMat 中的 `Offset` 即为调整贴图的UV坐标，可以看到背景贴图的平移。

> Note: 我们也可以通过实现一个Shader来制作背景滚动动画，这会在渲染的章节介绍。

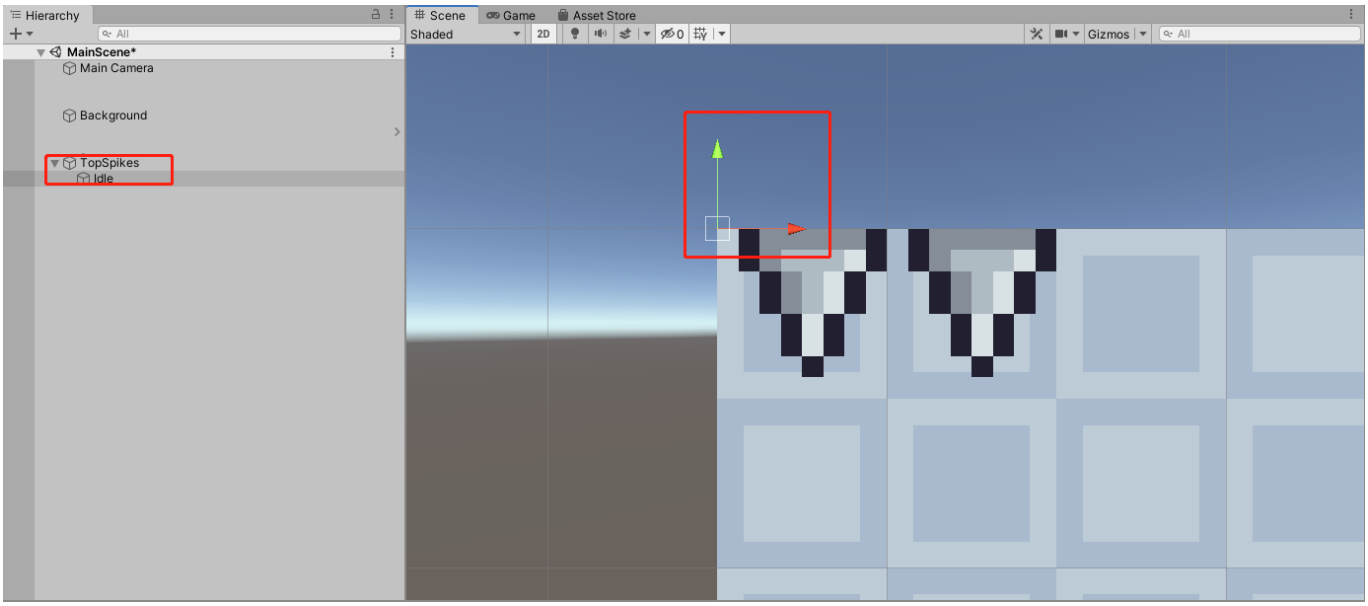
顶部尖刺

创建顶部尖刺

新建一个空物体，名为 TopSpikes 。然后打开 `Pixel Adventure 1/Assets/Traps/Spikes` ，选择其中的 `Idle` ，设置 `SpriteMode->Pixels Per Unit=16`（16比较适合本次作业，注意之后的每个2D素材都需要设置该项为16）。将其从文件浏览器拖入场景中，此时场景中出现了一个刺形状，设为 TopSpikes 的子对象，并将 `Transform->Position->z` 设为 `0` ，以防无法和后续物体发生碰撞（注意之后每个2D物体都需要设定位置的z坐标为0）。



设置 `Transform->Rotation->z=180` ，使之方向向下，将其放在背景顶部。为了方便对齐，可以在使用 `Move Tool` 的情况下，按住键盘 `V` ，尝试选中尖刺的左上锚点来移动物体，这时物体会自动吸附对齐到其他物体上。

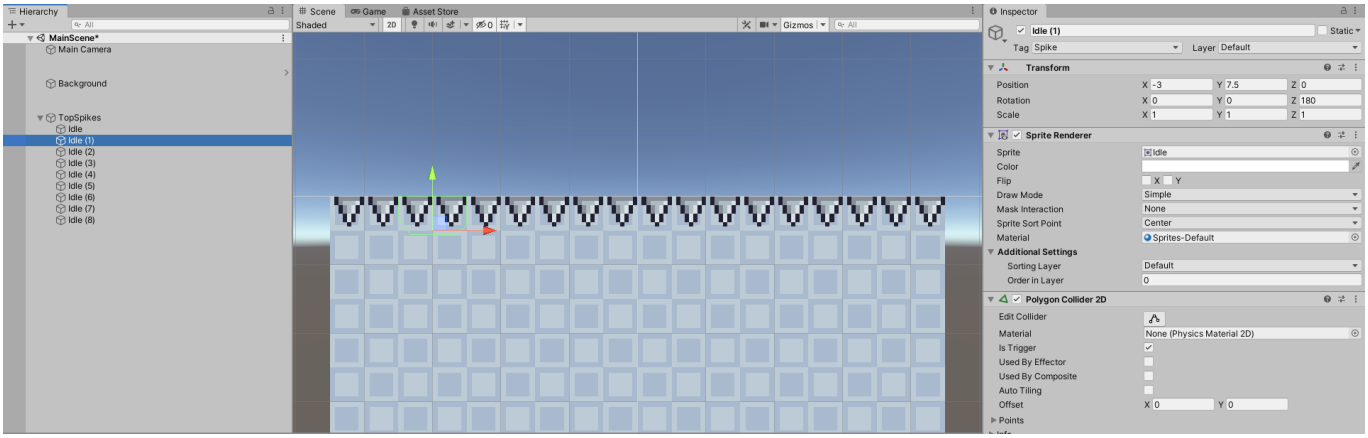


复制尖刺，利用对齐小技巧，铺满背景顶部。

添加碰撞体

选中所有的尖刺，添加PolygonCollider2D，以便后续计算碰撞。

> Note: 打开Scene Tab右上角的Gizmos选项，可以方便随时随地查看物体的碰撞体等信息。其中碰撞体以绿色线标出。

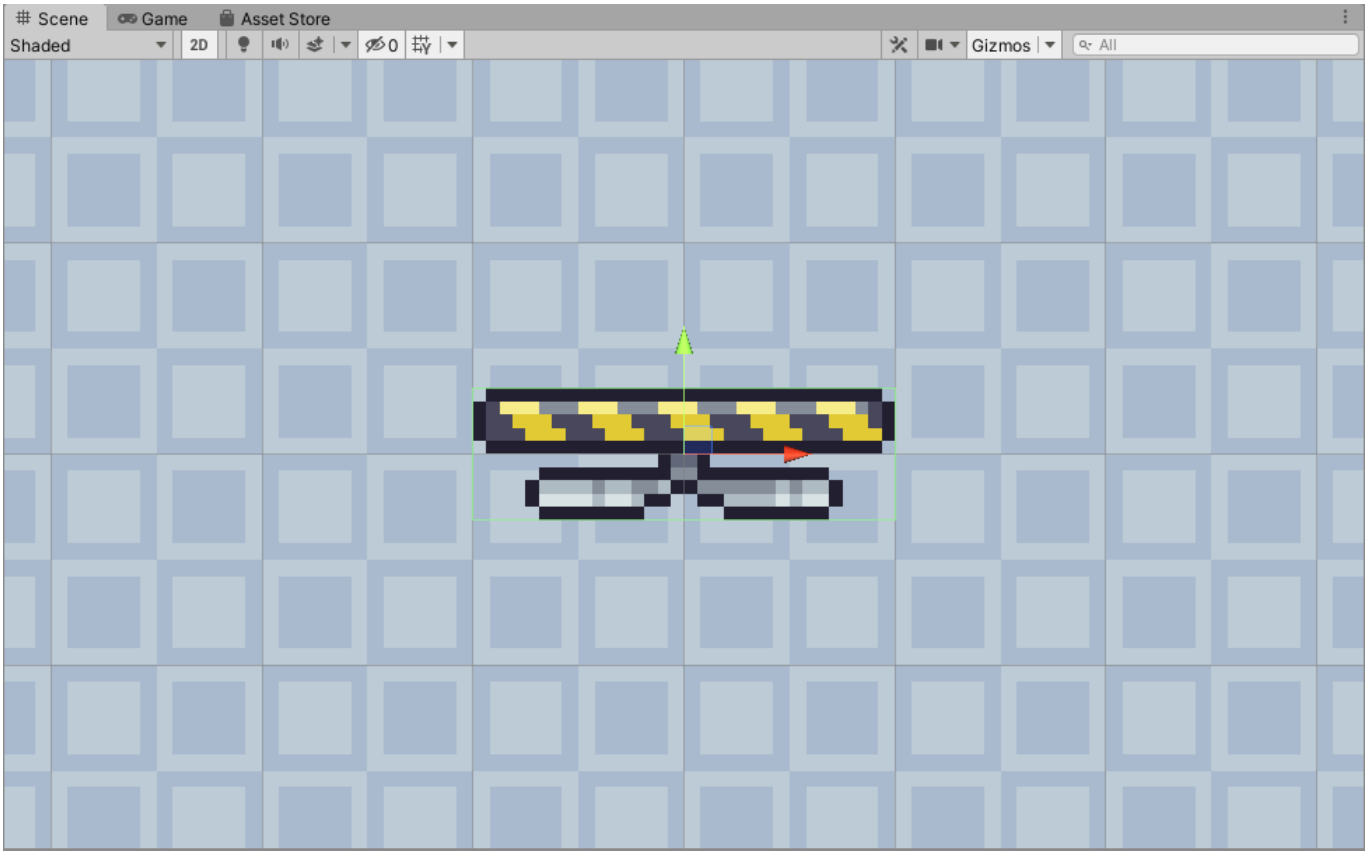


> Note: Collider 是 Unity 实际用于物理计算的组件。 Unity 中提供了几种基本形状的 Collider ， 如 Sphere Collider/Box Collider/Capsule Collider ， 这几种 Collider 在碰撞计算中开销较低，常常会使用在游戏中。在刚体物理仿真中，还可以通过为 Collider 赋上 Physic Material 来指定物体的摩擦系数、弹力等来模拟不同材质的物体。

2. 平台

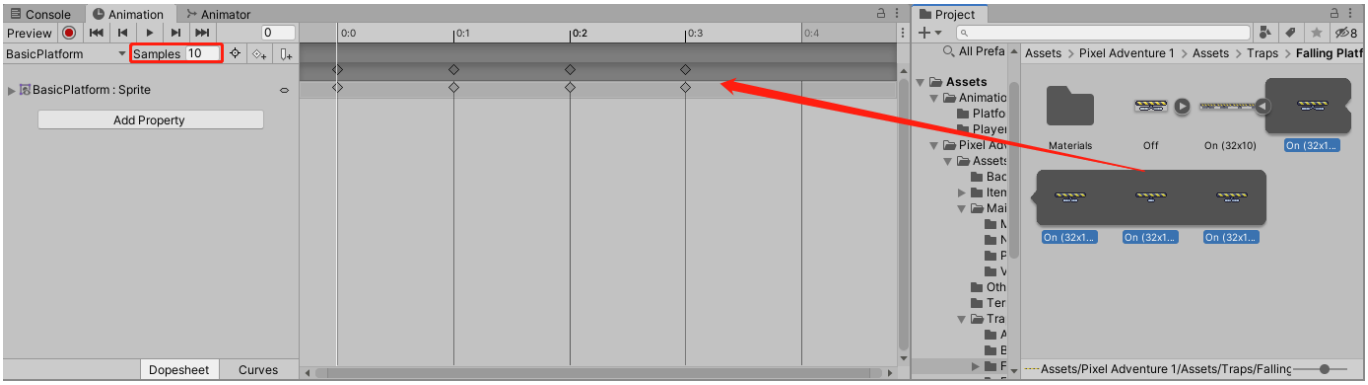
普通平台

创建平台 普通平台来自于文件中的 `Pixel Adventure 1/Assets/Traps/Falling Platform/On` 。同样的将其`SpriteMode->Pixels Per Unit` 设为16，并复制其中的 `On(32x10)_0` 到场景中，命名为Basic Platform。为了让它在游戏中可以托住角色，为其添加碰撞体 `BoxCollider2D` ，并使用 Edit Collider 调整使其碰撞体贴合模型。将普通平台命名为 Basic Platform。



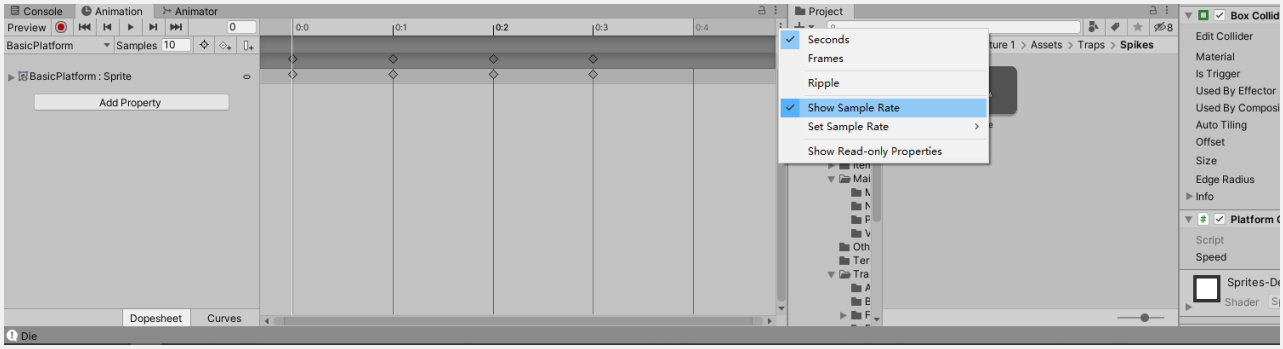
平台动画 菜单栏 `Window->Animation->Animation` 打开动画窗口，在 Hierarchy Tab 中选中 Basic Platform ，点击 Animation Tab 中的 Create 为其添加一个新的 Animation Clip 动画片段。选择 `Samples=10` ，这个帧率比较符合像素风格的游戏。

在 `Pixel Adventure 1/Assets/Traps/Falling Platform/` 中选中 `On` 中的4帧图片，拖拽到Animation窗口空轨道中。



测试

运行游戏，可以看到平台的帧动画，平台下方的风扇会转动。

> Note: 在一些新版本的 Unity 中，Sample Rate 窗口不是默认显示的，可以通过如下界面打开。
>
> 
>

> Note: Animation 可以为模型/图片增添动画效果，使得游戏更加生动。在游戏中往往会使用多个 Animation Clip 分别表示一个角色或物体的几个动作，然后用 Animator 来管理这些动作，实现在角色不同状态下切换播放不同的 Animation Clip，实现角色停走跑跳、攻击、防御等动作。

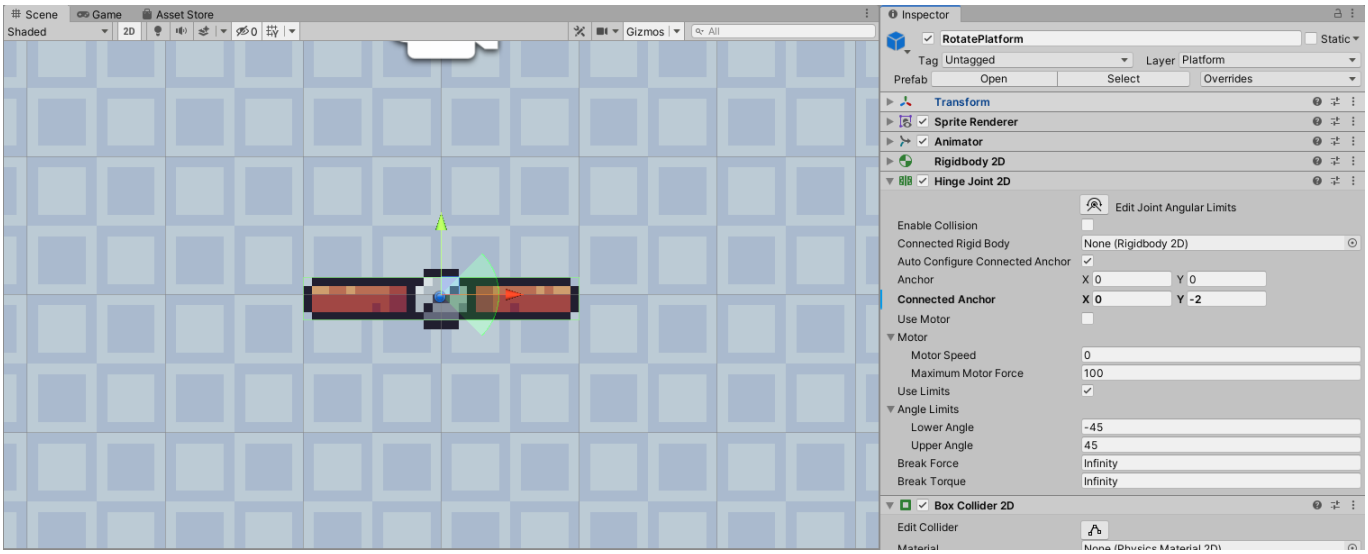
旋转平台

创建平台

这一节将制作一个可以绕着中间轴旋转的旋转平台。参照上一节，选择 `Pixel Adventure 1/Assets/Traps/Platforms/Brown on` ，按照上述普通平台的方法制作平台，命名为 `Rotate Platform` 。同样为其添加动画。

物理铰链

为了使平台能够绕着中间轴旋转，需要添加铰链组件，即 `HingeJoint2D` ，注意此时该物体会自动被添加一个 `Rigidbody2D` 的组件用于刚体运动计算。然后为其增加一个 `BoxCollider2D` 使之能与其他物体发生碰撞。



测试

为了测试效果，我们给 Basic Platform 加上 `Rigidbody2D` ，并放置在 Rotate Platform 侧上方，然后运行游戏，会看到 Basic Platform 掉落，并击中 Rotate Platform 使之绕轴转动。`HingeJoint2D->Angle Limits` 还可以控制模型旋转的角度。在测试完成后记得去掉 Basic Platform 中的 `Rigidbody2D`。

> Note: Joint 是 Unity 物理系统的一部分，其中包括 Hinge Joint / Spring Joint / Fixed Joint 等，2D版本还有 Distance Joint 2D 等。这些铰链实现了不同的物理约束，为游戏提供了实现丰富机关的可能。

链球障碍

在一些游戏中还会出现链子连接的球作为障碍物。在 `Pixel Adventure 1/Assets/Traps/Spiked Ball/` 中提供了相关素材。新建一个空物体，命名为 SpikedBall 。将 `Pixel Adventure 1/Assets/Traps/Spiked Ball/` 中的 `Chain` 和 `Spiked Ball` 作为 SpikedBall 的子对象，名为 Chain 和 Ball 。

在 Chain 和 Ball 中分别添加组件 `Rigidbody2D` ，其中 Chain 的 `Rigidbody2D->BodyType=static` 以防其随重力掉落，而Ball中 `BodyType=Dynamic` ，因此它能因重力下摆。

为了实现 Ball 绕着 Chain 转动，为 Ball 添加组件 `DistanceJoint2D` ，其中 `Connected Rigid Body` 设为 Chain 。在打开 Gizmos 时可以看到一条绿线表示连接。此时运行游戏可以看到 Ball 绕着 Chain 来回摆动。为了使之和角色发生碰撞，还需要为 Ball 添加 `CircleCollider2D` 。

平台移动

在该游戏中，场景中的平台会随着时间向上移动，形成玩家在不停向下闯关的错觉。为实现这个效果，新建一个名为 `PlatformController.cs` 的脚本，其中代码为：

```
public float speed;
Vector3 speedVec3;

// Start is called before the first frame update
void Start()
{
    speedVec3.y=speed;
}

// Update is called once per frame
void Update()
{
    Move();
}

void Move(){
    transform.position += speedVec3 * Time.deltaTime;
}
```

将该脚本挂在在所有的平台、障碍物上，并在 Inspector 中设置一个合适的速度值 `PlatformController->speed=1`。

测试

运行游戏，可以看到场景中的平台都随着时间上移了。

注意此时，由于所有平台都设置了碰撞体，因此当他们移动到顶端会和 Top Spikes 发生碰撞（特别是 Spiked Ball ）。因此设置 Top Spikes 中的所有刺的 `Polygon Collider 2D->Is Trigger=true` 来避免。

> Note: Collider 的 isTrigger 属性决定了物体碰撞时的效果。当 isTrigger=false ， 当两个物体碰撞时会产生碰撞效果，并且会触发 OnCollisionEnter/Stay/Exit 函数；而当 isTrigger=true ， 当两个物体碰撞时不会出现碰撞效果，且触发的是 OnTriggerEnter/Stay/Exit 函数。本次作业中将 TopSpikes 设成 trigger 使之不与 Spiked Ball 发生碰撞，而其他平台障碍物则是普通 collider ，用于计算物理碰撞。

拓展

现在你已经学会了所有平台的实现，你可以根据提供的其他素材，尝试不同的物理组件，来创建其他的平台或障碍。（加分项）

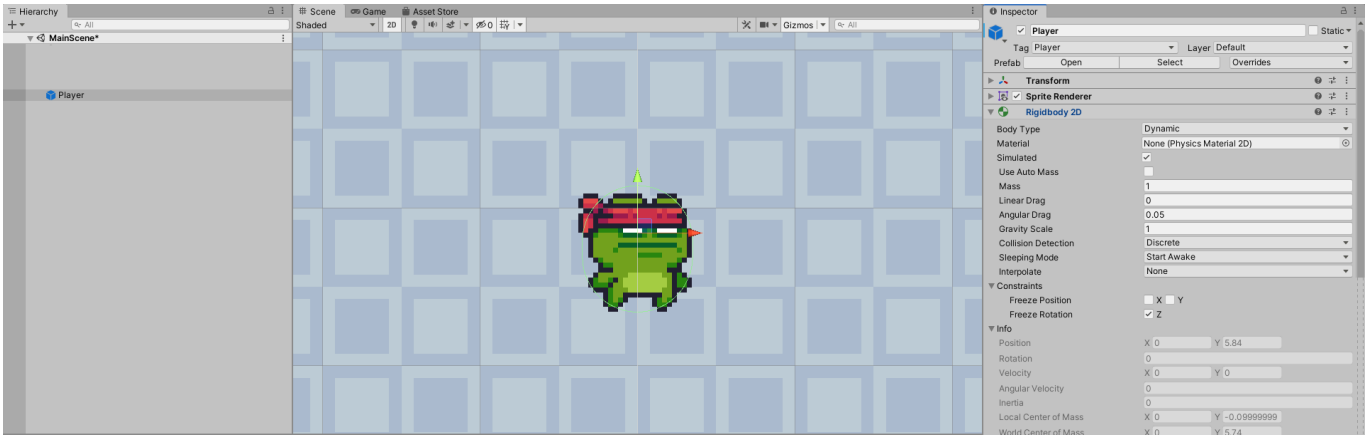
3. 角色

创建角色

类似于之前的建立平台，我们在 `Pixel Adventure 1/Assets/Main Characters/` 中选择自己喜欢的主角，并选择其 Idle 动画的第一帧创建物体，命名为 Player 。

角色移动

首先给角色添加组件 `Rigidbody2D` ，并设置 `Body Type=Dynamic` ， `Constraint->Freeze Rotation Z=true` ，使之会随重力下落，并不会再过程中左右转动。为其添加组件 `CapsuleCollider2D` ，设置其中的 `Offset` 和 `Size` 使得碰撞体位置大小基本贴合角色模型。



然后我们要实现角色的运动。在新建一个 `PlayerController.cs` 的脚本挂在 Player 下，其中添加代码：

```
Rigidbody2D rb;
public float speed;

// Start is called before the first frame update
void Start()
{
    rb = GetComponent<Rigidbody2D>();
}

// Update is called once per frame
void Update()
{
    Move();
}

void Move()
{
    float xInput = Input.GetAxisRaw("Horizontal");
    rb.velocity = new Vector2(xInput*speed, rb.velocity.y);
```

```
    if(xInput != 0)
        transform.localScale = new Vector3(xInput, 1, 1);
}
```

其中因为获得的 `xInput` 是 `1` 或 `-1`，通过 `transform.localScale` 我们可以设置角色的左右朝向。在Inspector中，设置 `Player Controller->speed=5`，比较适合游戏中的角色移动速度。

测试

在场景中，将 Basic Platform 移动到 Player 下方，运行游戏，使 Player 落在 Basic Platform 上，通过键盘可以控制其左右运动，当其到平台边缘时会坠落。

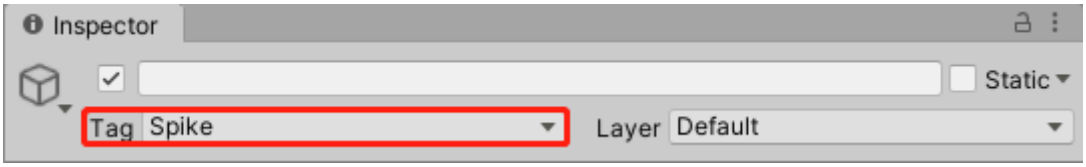
角色死亡

当角色触碰 TopSpikes 或 SpikedBall 或坠落时，触发死亡。为了检测角色死亡，在 `PlayerController.cs` 中实现一个函数来简单输出Log。

```
public void Die(){
    Debug.Log("Die");
}
```

由于 Top Spikes 设置了 `isTrigger=true`，因此我们需要用 Trigger 的方法实现碰撞检测。2D对象碰撞时，在被 IsTrigger 物体（Spikes）碰撞的物体（Player）中，`OnTriggerEnter2D` 会被调用，其中会有参数 `Collider2D other` 表示触发的对象。与 Roll-A-Ball 教程中类似，为所有的 Spikes 加上名为 `Tag=Spike`，并在 `PlayerController.cs` 中实现如下代码：

```
void OnTriggerEnter2D(Collider2D other)
{
    if(other.CompareTag("Spike"))
    {
        Die();
    }
}
```



而对于 SpikedBall->Ball，它并非一个 Trigger，因此在碰撞发生时，`OnCollsiionEnter2D` 会被调用到。同样地，为 SpikedBall->Ball 加上 `Tag=Spike`，并在 `PlayerController.cs` 中实现如下代码：

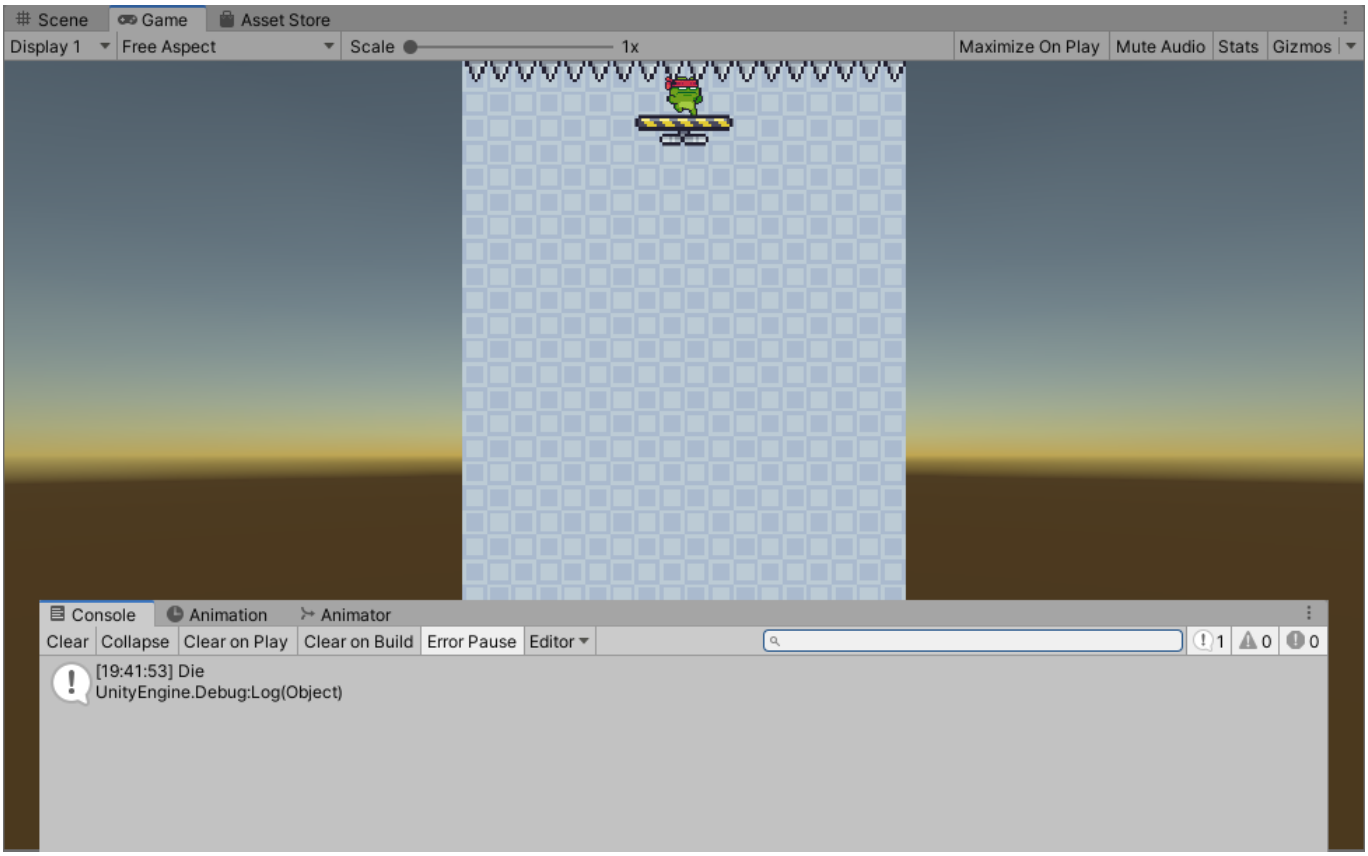
```
void OnCollisionEnter2D(Collision2D other)
{
```

```
        if(other.gameObject.CompareTag("Spike"))
        {
            Die();
        }
    }
```

对于坠落情况，新建一个空物体，名为 BottomLine ，并将其放置在可视范围以下，如`(0,-10,0)`，为其建立一个 `BoxCollider2D` ，并将其 `Size` 设置足够宽，如 `(20,1)` 。设置 `Tag=Spike` ，这样当玩家坠落至该 Collider 位置时，也触发 Die() 函数使得游戏结束。

测试

运行游戏，尝试碰撞 Top Spike或SpikedBall，会发现Console中输出相应的Log。

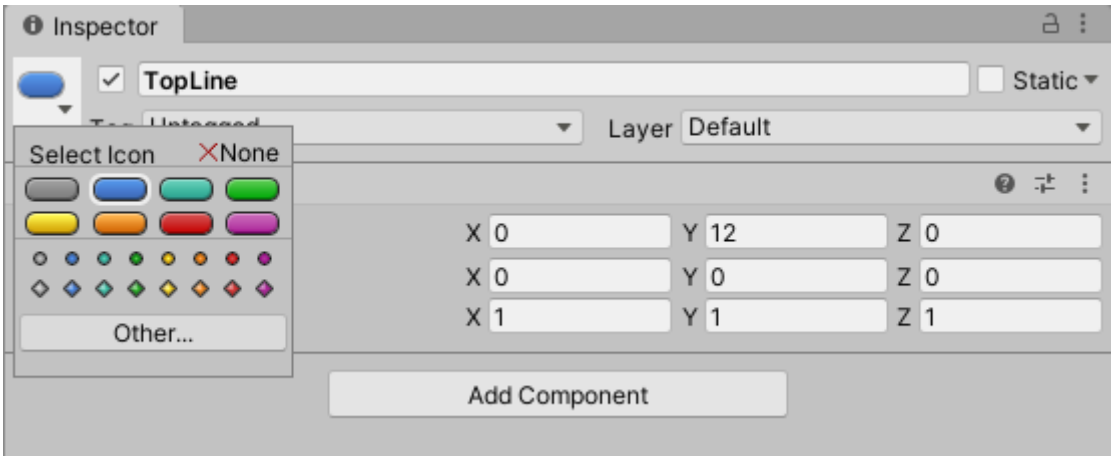


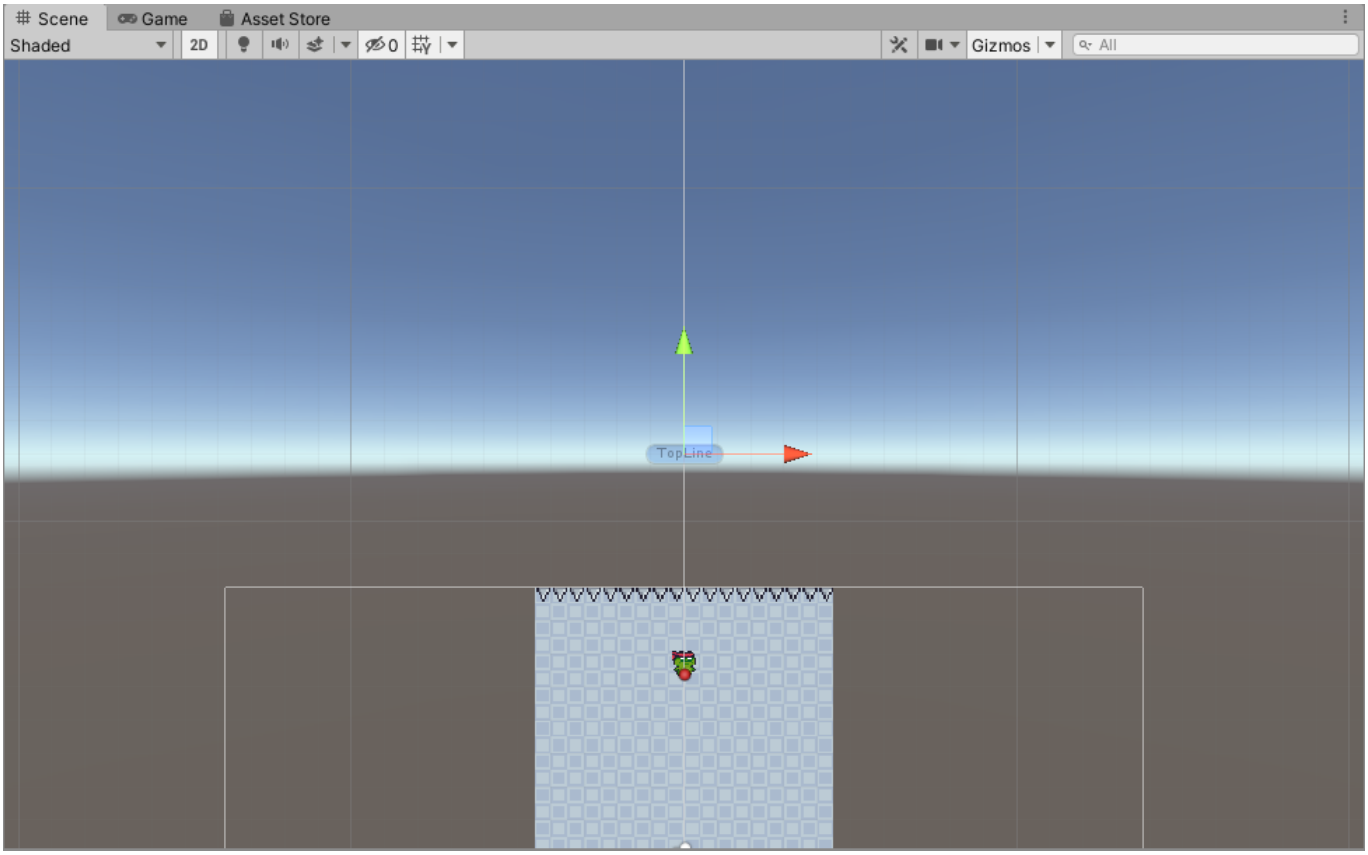
4. 平台随机生成

接下来，实现平台的无尽随机生成。

出界平台销毁

当上升的平台运动到屏幕以外时，需要将其删除，以免创建对象过多带来太大的内存开销。为此，我们可以在画面中创建一个空物体，将其名为 TopLine ，作为平台移动上限的标志物。为其选择一个醒目的 Icon 方便查看，调整其至屏幕上方合理的位置，如 `(0,12,0)` 。





在 `PlatformController.cs` 中，在游戏运行时查找场景中名为 TopLine 的对象：

```
GameObject topLine;

// Start is called before the first frame update
void Start()
{
    ...
    topLine = GameObject.Find("TopLine");
}
```

并在每次更新时判断当前平台/障碍物的高度是否从超过了 topLine 。一旦超过，销毁当前对象。

```
void Move(){
    ...
    if(transform.position.y>topLine.transform.position.y)
        Destroy(gameObject);
}
```

测试

运行游戏，可以在 Scene Tab 看到当平台上升到 TopLine 之上时，就会被销毁。

新平台生成

参考 Roll-A-Ball 教程，将之前的几个平台和障碍物都制作为 Prefab 。保留一个 Basic Platform ，删去现有场景中其他的平台和障碍物，并调整位置保证玩家一开始会掉落到该平台上。

方便起见，直接利用之前的 BottomLine 来作为刷新点，生成新的平台。为 BottomLine 新建一个脚本 `Spawner.cs`（在 Minecraft 中 Spawner 为刷怪笼），并在其中实现平台生成的逻辑。为了告诉 `Spawner.cs` 现在有的平台预设体，新建 `public List<GameObject> Platforms`，在 Inspector 中填入已有的几个预设体。然后实现一个函数 `SpawnPlatform()` 来实现平台的产生。

首先随机生成刷新位置，然后通过随机数生成要刷新的 Prefab ，并将其挂在当前对象下。

```
public void SpawnPlatform()
{
    Vector3 spawnPosition = transform.position;
    spawnPosition.x = Random.Range(-3.5f, 3.5f);

    int index = Random.Range(0, platforms.Count);
    GameObject go = Instantiate(platforms[index], spawnPosition,
Quaternion.identity);

    go.transform.SetParent(this.gameObject.transform);
}
```

要实现隔一段时间刷新一个平台，需定时调用 `SpawnPlatform()` ：

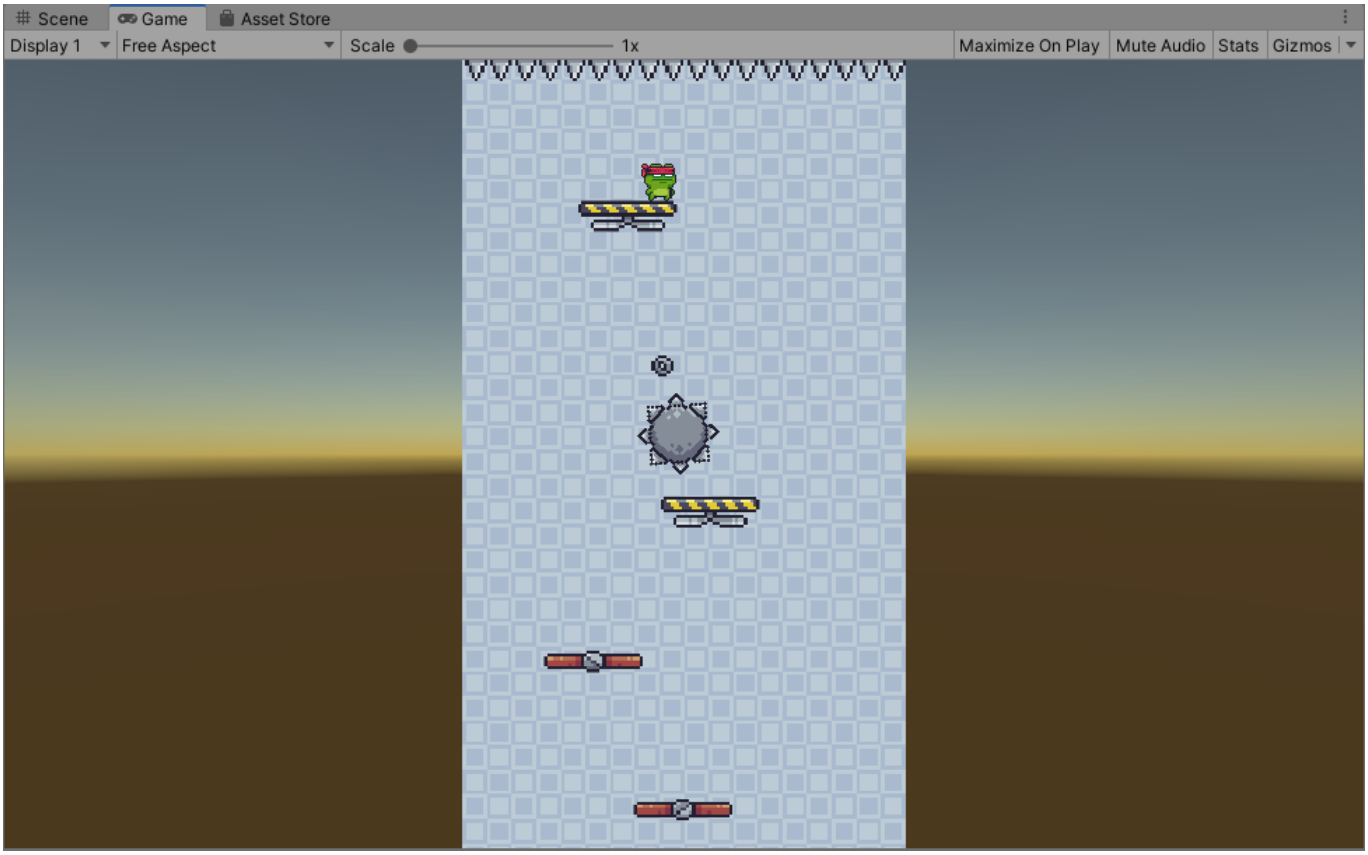
```
public float spawnTime;
private float countTime;

void Update()
{
    countTime += Time.deltaTime;

    if(countTime >= spawnTime)
    {
        SpawnPlatform();
```

```
            countTime = 0;
        }
    }
```

> Hint: 在许多项目中，还会使用对象池的方法管理大量生成的物体。这种方法大大地节省了物体创建、初始化、销毁的开销。

测试

设置 `Spawner.cs->spawnTime=3`，运行游戏，即可得到一个新手友好的游戏。通过调整随机数选择策略、刷新时间，还可以设计不同的游戏难度。

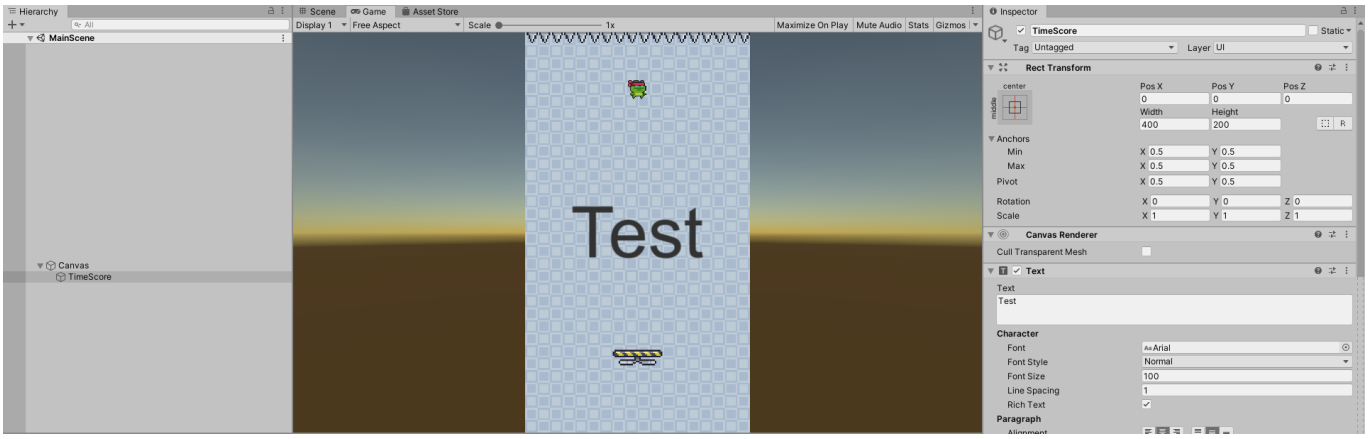


5. UI/游戏管理

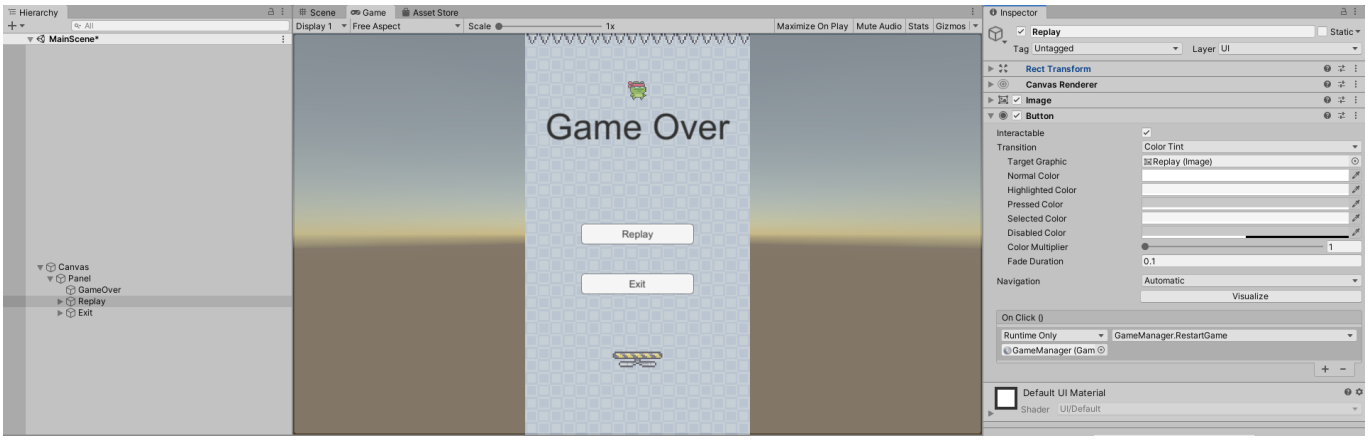
最后，为游戏添加UI和游戏管理功能。

UI界面

实现计时分数。在 Hierarchy 中建立一个 `UI->Text` 物体，命名为 `TimeScore`。调整其位置和文字大小（由于是UI，需要切换到 Game Tab 下才能查看效果）。



实现游戏结束页面。在 Canvas 下新建一个 Panel ， 调整颜色。在 Panel 下建立两个按钮，一个为 Restart ， 一个为 Exit 。



在游戏开始时，表示游戏结束界面的 Panel 不应该出现，因此一开始将其 Inspector 里面的勾去掉，来隐藏这个 Panel 。

游戏管理

单例模式

游戏UI和游戏状态受到游戏管理器的控制。新建一个空物体，名为 GameManager ， 并在其下建立 `GameManager.cs`脚本。

一般地，这种 Manager 脚本会被实现为单例模式，以便其他脚本调用并避免重复的生成。一个简单的单例模式可以实现如下：

```
static GameManager instance;
private void Awake()
{
    if(instance!=null)
        Destroy(gameObject);
    instance = this;
}
```

> Note：单例模式（Singleton pattern）是一种常用的设计模式，它能够保证一个类在程序运行中仅有一个实例。从而用户可以直接通过一个全局的接口来访问这个实例（如静态共有函数）。

分数显示

这个游戏中使用游玩时间表示分数。要使 `GameManager.cs` 控制分数变化，需要代码：

```
public Text timeScore;

void Update()
{
    timeScore.text = Time.timeSinceLevelLoad.ToString("00");
}
```

游戏控制

游戏结束时，将时间暂停，并显示 Panel 。因为有了单例模式，这部分函数可以实现为 `static` ，被其他对象调用。

```
public static void GameOver(bool dead)
{
    if(dead)
    {
        instance.gameOverPanel.SetActive(true);
        Time.timeScale = 0;
    }
}
```

然后在 `PlayerController.cs` 的 `Die()` 中即可以调用 `GameManager.GameOver(bool dead)` 即可实现死亡时显示游戏结束画面，并使时间停止。

退出游戏可以实现为

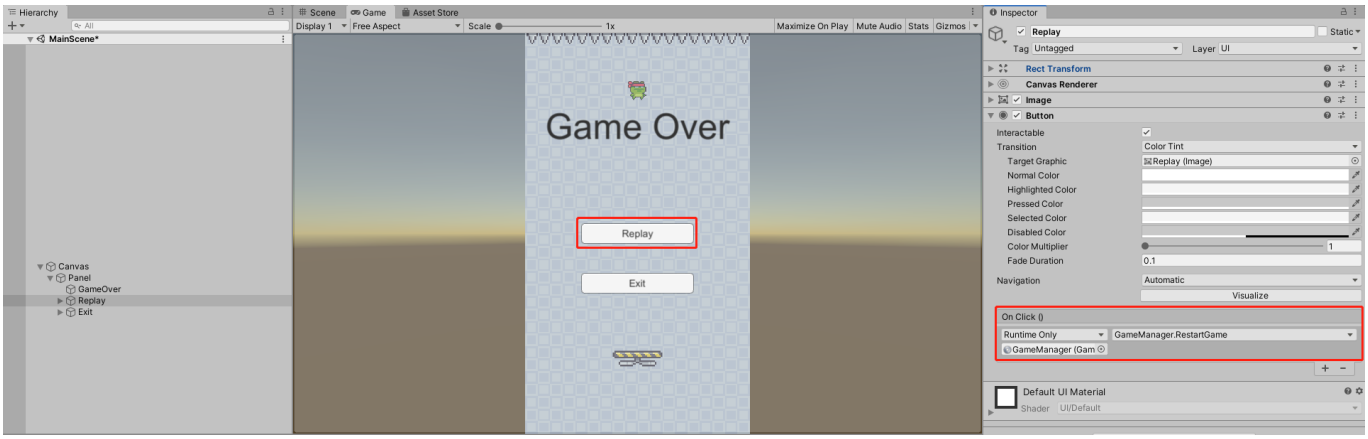
```
public void ExitGame()
{
    Application.Quit();
}
```

注意，退出游戏的功能只在打包发布后才能生效。

重启游戏可以实现为

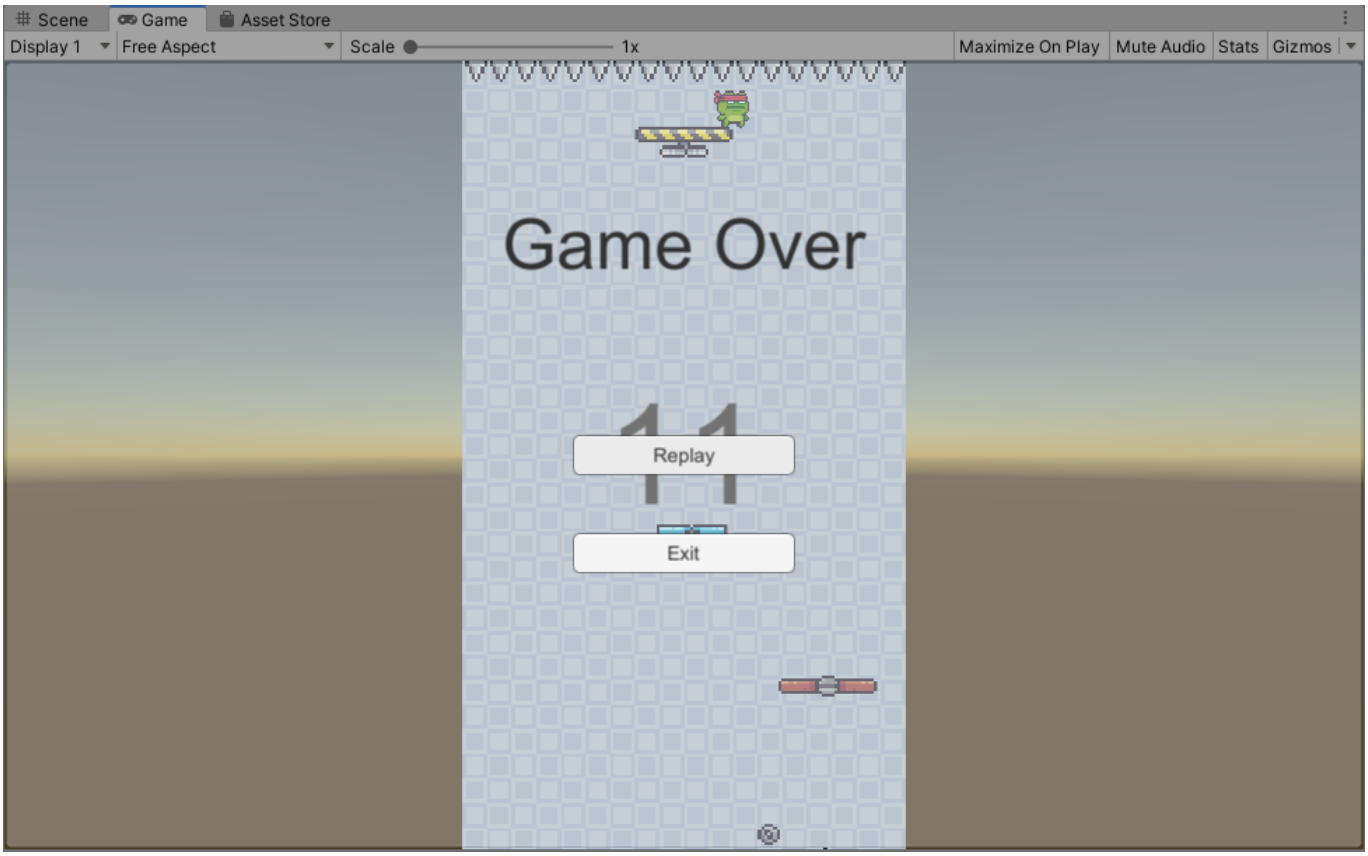
```
public void RestartGame()
{
    SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    Time.timeScale = 1;
}
```

为了将重启和退出游戏的功能和UI中的按钮关联起来，我们需要在相应按钮中找到组件上 `Button->On Click()`，将其设置为 GameManager 物体下 `GameManager.cs` 脚本的对应函数。



测试

此时运行游戏，会发现游戏不仅可以显示游戏时间，还会在死亡后出现重玩界面等。



至此，你已经完成了一个相对完整的游戏了。

6. 角色动画

一个只能漂移的角色还不够生动，这节将使用动画状态机丰富其角色动画。

首先选中 Player，使用 `Pixel Adventure 1/Assets/Main Characters/` 中的素材为其添加 `PlayerIdle/PlayerRun/PlayerFall` 的 Animation Clip 。

通过 `Window->Animation->Animator` 打开 Animator Tab。选择 Player，此时会在 Animator Tab 中看到 Entry/Any State 以及刚刚创建的几个动画状态。查看文件夹，可以发现除了后缀名为 `.anim` 的 Animation Clip

文件，还自动新建了一个名为 `Player.controller` 的文件，这个文件即为在 Animator Tab 中显示的 动画控制器（Animator Controller），它可以用来控制 Animation Clip 之间的切换。

> Note: 动画状态机（Animator）将不同的动画片段表示为不同的状态，通过设定状态之间的转化条件实现不同动画的切换。其中还可以通过设定不同的Layer来实现角色不同部分分别执行不同动画片段的效果。

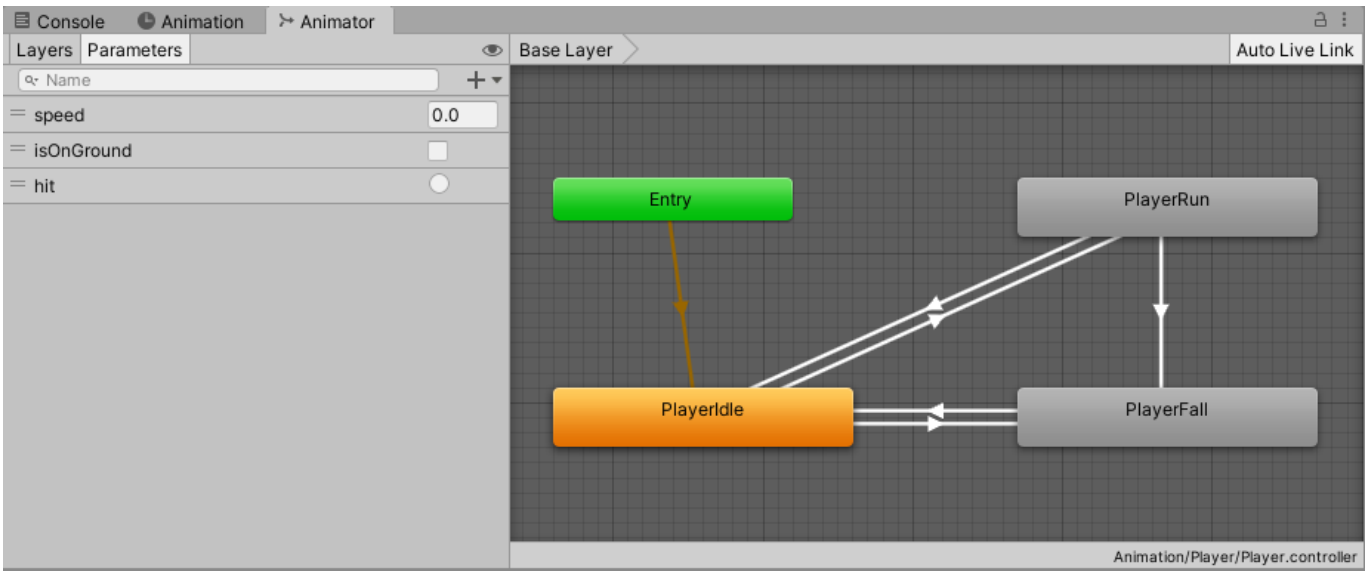
我们希望让动画状态机在一些条件下，切换播放不同的动画，比如角色移动时播放运动的动画，停下时播放站立的动画。默认条件下，应该播放站立动画，因此让 Entry （绿色）连接到 PlayerIdle 状态上，此时 PlayerIdle 状态显示为橙色，表示它是默认状态。（如果创建动画时先创建 PlayerIdle 动画，则自动会被设为默认状态）

然后在 Animator Tab 左侧选择 Parameter 界面，点击+来新建两个参数，一个是 `float` 类型的 `speed` ，表示当前角色运动速度；一个是 `bool` 类型的 `IsOnGround` ，表示当前角色是否在地面上。

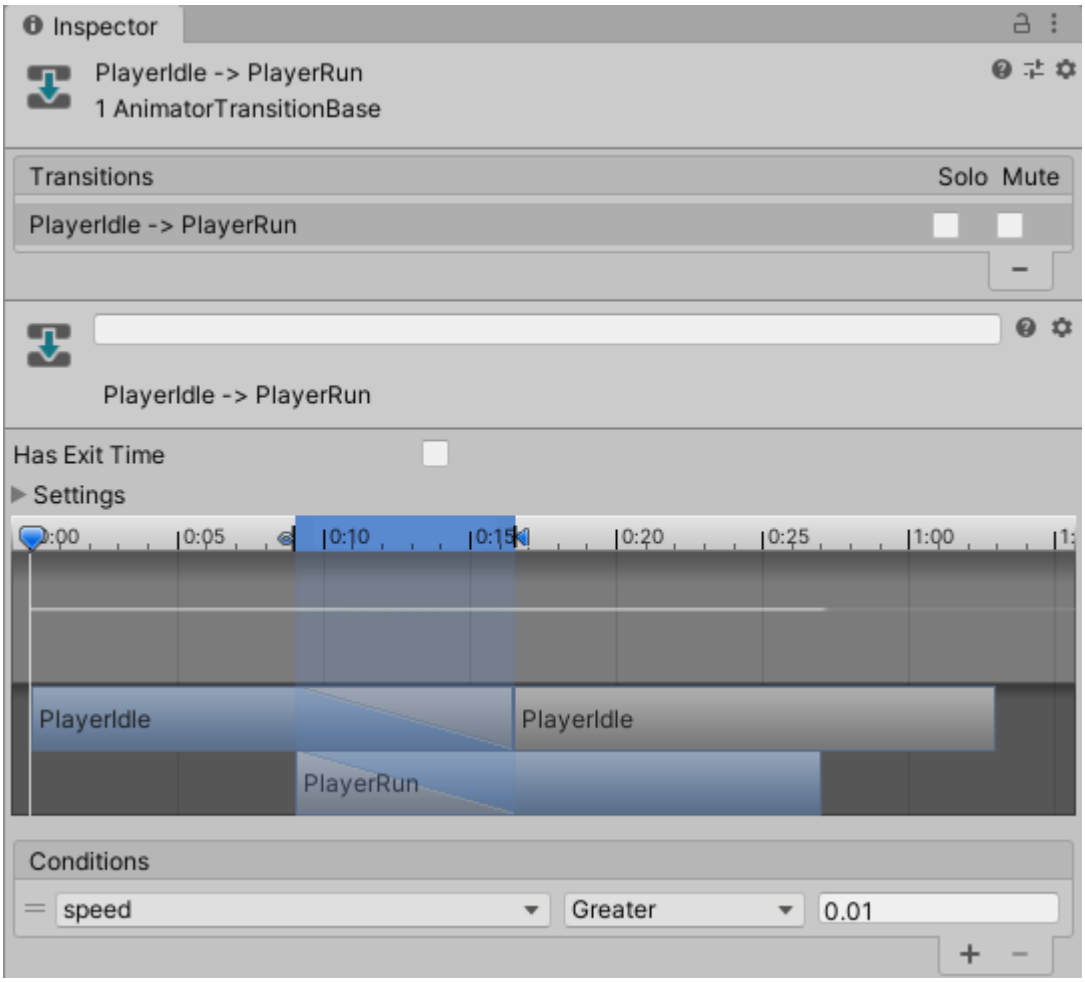
设计状态机逻辑，在 PlayerIdle 状态下，当 `speed>0.01` 时，应切换到 PlayerRun 状态；反之在 PlayerRun 状态当 `speed<0.01` ，应切换回 PlayerIdle 状态。同理，总结状态切换如下：

From State	Condition	To State
PlayerIdle	speed>0.01	PlayerRun
PlayerRun	speed<0.01	PlayerIdle
PlayerIdle	! isOnGround	PlayerFall
PlayerFall	isOnGround	PlayerIdle
PlayerRun	! isOnGround	PlayerIdle

在场景中，为了实现状态之间的跳转，在一个 From State 上右键，选择 `Make Transition` ，然后将箭头移到 To State 上。



选中表示 Transition 的箭头，可以看到它的详细参数。其中在 Settings 里面提供了前后两个状态切换时的混合效果，这有点像视频剪辑中的前一个片段淡出，后一个片段淡出的切换效果。由于本作业使用的是像素化的美术风格，动画帧率较低，选择去掉切换时的混合效果，即去掉 `Has Exit Time` 后的勾选框。然后，在 Conditions 中按照上表添加 Transition 的条件。



此时动画状态机里的参数还没有和实际游戏中的参数对应上。为此，需要修改 `PlayerController.cs` 代码，在 `Update()` 中更新动画状态机中的参数。

其中速度的设置实现如下：

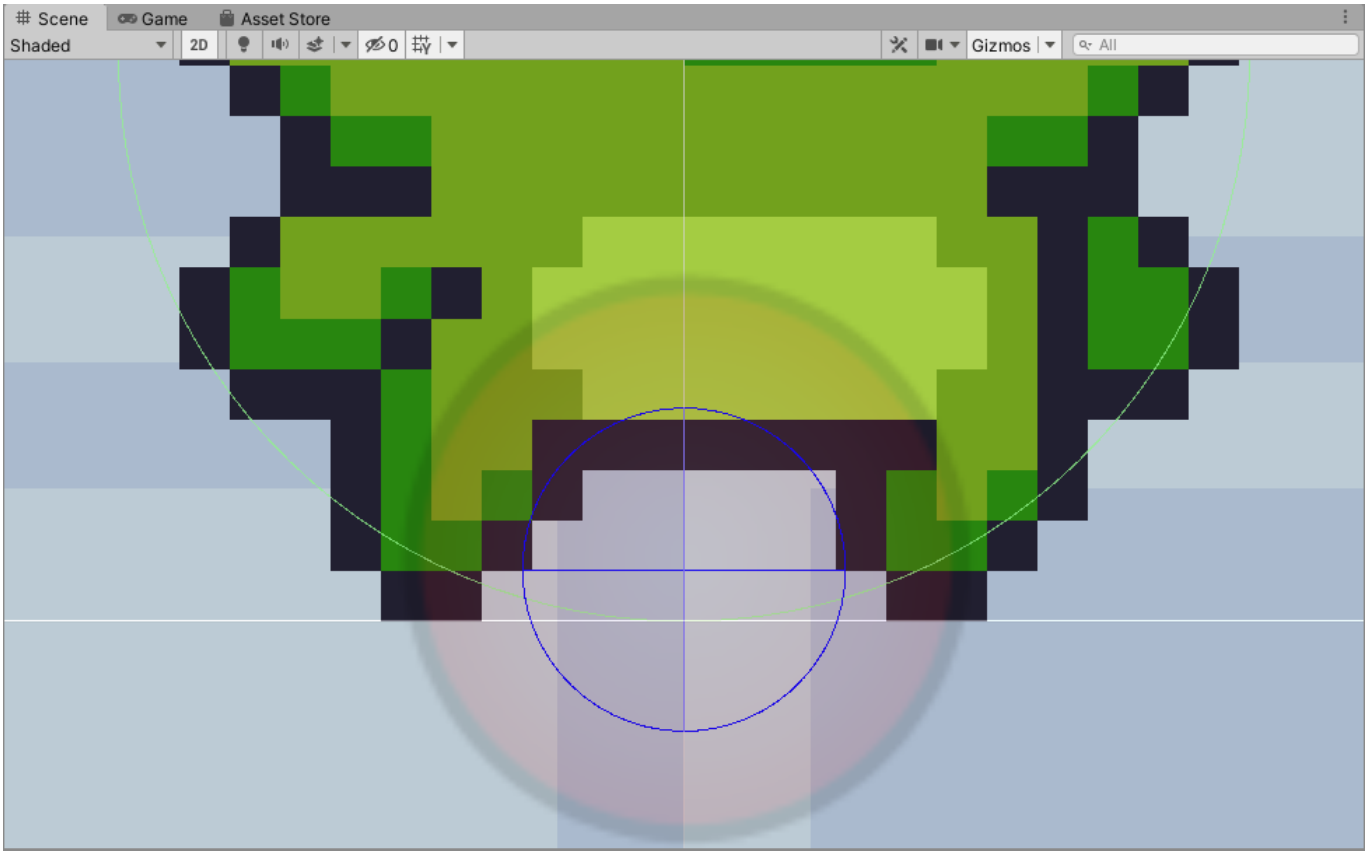
```
Animator animator;
void Start()
{
    ...
    animator = GetComponent<Animator>();
}
void Move()
{
    ...
    animator.SetFloat("speed", Mathf.Abs(rb.velocity.x));// run/idle
}
```

`isOnGround` 的判断则稍为复杂，它需要检测角色是否站在平台上。首先对每个平台的 Prefab ， 设置 `Layer=Platform`。（这一步与修改 Tag 类似） 在 Player 下建立一个空物体，名为 CheckPoint ， 并调整其位置至角色下方，可以使用一个 icon 方便查看。然后再代码中通过检测 CheckPoint 周围是否有 `Layer=Platform` 的物体实现平台的检测，如下：

```
public float checkRadius;
public LayerMask platformMask;
public GameObject checkPoint;
public bool isOnGround;
void Update()
{
    isOnGround = Physics2D.OverlapCircle(checkPoint.transform.position,
checkRadius, platformMask);
    animator.SetBool("isOnGround", isOnGround);
    Move();
}

private void OnDrawGizmosSelected()
{
    Gizmos.color = Color.blue;
    Gizmos.DrawWireSphere(checkPoint.transform.position, checkRadius);
}
```

这里为了方便调整 `checkRadius` 和检查点的位置，使用 `OnDrawGizmosSelected()` 来在 Gizmos 开启的情况下显示一个辅助圆。

完成之后在对 `PlayerController.cs` 的参数 `Check Radius/Platform/Check Point` 进行设置。 选中 Player ，在打开 Gizmos 时，可以看到 CheckPoint 上显示的蓝色圆圈，表示现在用于检测平台的范围。调整 CheckPoint 的位置和 `checkRadius` 的大小使之合适。



测试

运行游戏，可以看到当角色在平台上不动时，会播放 PlayerIdle ；角色左右移动时会播放 PlayerRun ；角色下落时会播放 PlayerFall 。

7. 总结

实验得分是根据最终提交的 exe/unitypackage 中包含内容来评价的。

1. 背景 (10%) ：实现可以移动的背景
2. 平台 (20%) ：实现教程中提到的三种平台
3. 角色 (15%) ：实现角色移动
4. 生成 (20%) ：实现无限平台生成和销毁
5. UI/游戏管理 (20%) ：实现计分；实现游戏管理脚本
6. 角色动画 (15%) ：实现角色动画切换
7. 加分项(+10%) ：实现不同平台或障碍物，其中需要用到其它的物理组件或编写脚本实现平台特效等，可以参考原作者的视频教程实现风扇弹跳机关，也可以参照其他平台跳跃游戏实现其他平台或障碍。实现其他对游戏机制的改进，也将根据实现难度和创意给予附加分。