

第四次实验： 2D 游戏开发

学号：22920212204392 姓名：黄勛

一、 实验目的

- Sprite 编程技术
- 掌握 Unity 2D 游戏开发方法

二、 实验条件

- 系统环境：Windows 10 21H2
- 软件环境：Unity 3D 2021.3.14f1c1

三、 实验内容

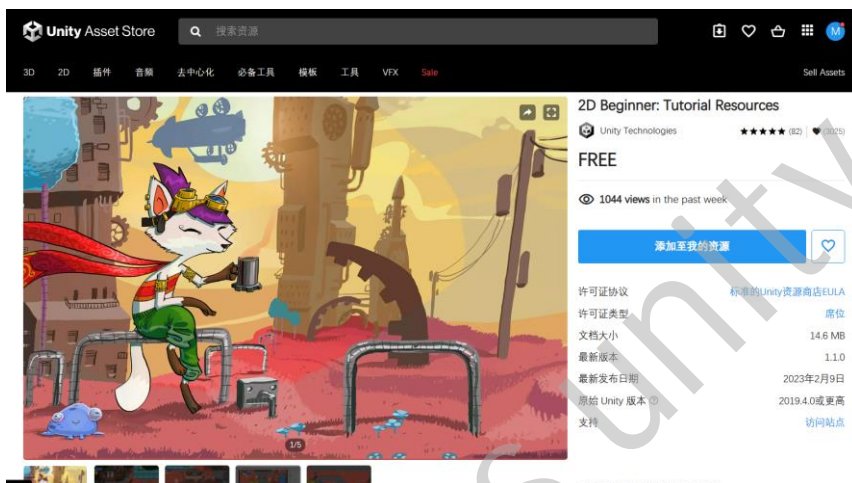
- 参考 Unity 教程，选择游戏 *Ruby's Adventure: 2D Beginner* 实现
 - ◆ 1. 实验步骤基本涉及对脚本、Inspector 属性、Project_Settings 这三个部分的编辑，实验步骤以简书/官方教程步骤的大标题为单位，每个大标题步骤进行必要说明和相应工作的截图
 - ◆ 2. 对于每个大标题，需解释或复制教程中的关键句，配以一张及以上自己操作的截图，若是代码则高亮出与本步骤相关的关键段落
 - ◆ 3. 选择 SpaceShooter 项目的同学，实验报告需要包含的项目步骤有：
除去 “教程(一) 1~3 步+教程(五) 第 7 步” 之外的所有大标题，步骤的说明和截图内容参照第 2 点
 - ◆ 4. 选择 Ruby's Adventure 项目的同学，工程至少完成到 [世界交互-伤害区域和敌人] 及其之前的内容。实验报告需要包含的项目步骤有：
“角色控制与键盘输入、装饰世界、可收集对象、伤害区域和敌人” 的所有大标题（可不含总结），步骤的说明和截图内容参照第 2 点

四、 实验项目步骤:

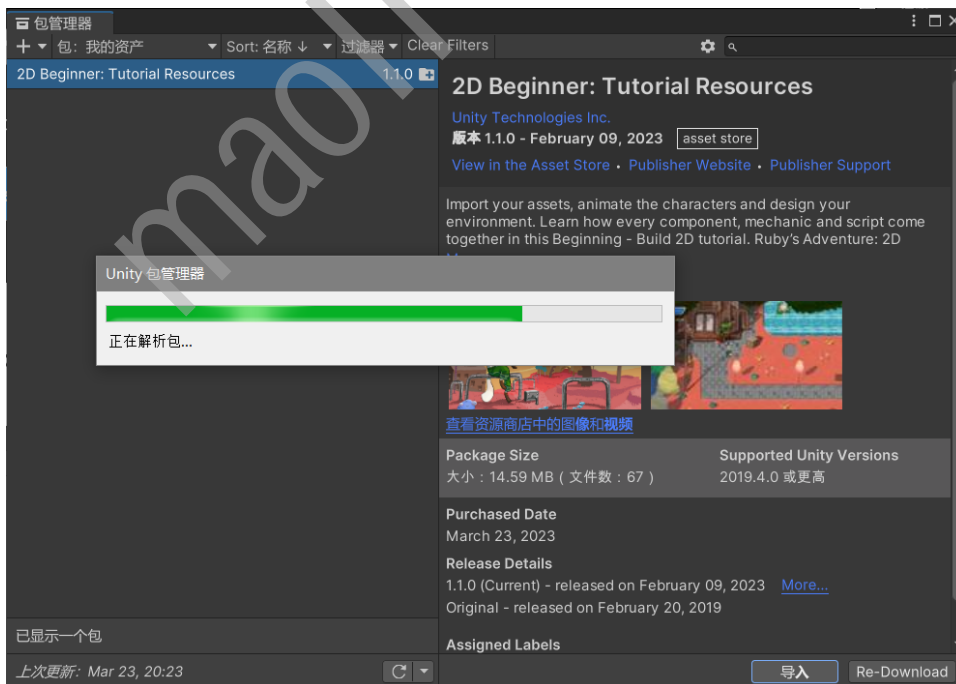
为了实现实验内容，主要的工作是编写 2D 模板游戏编辑项目和拓展脚本，主要内容与注释如下：

(一) 准备工作

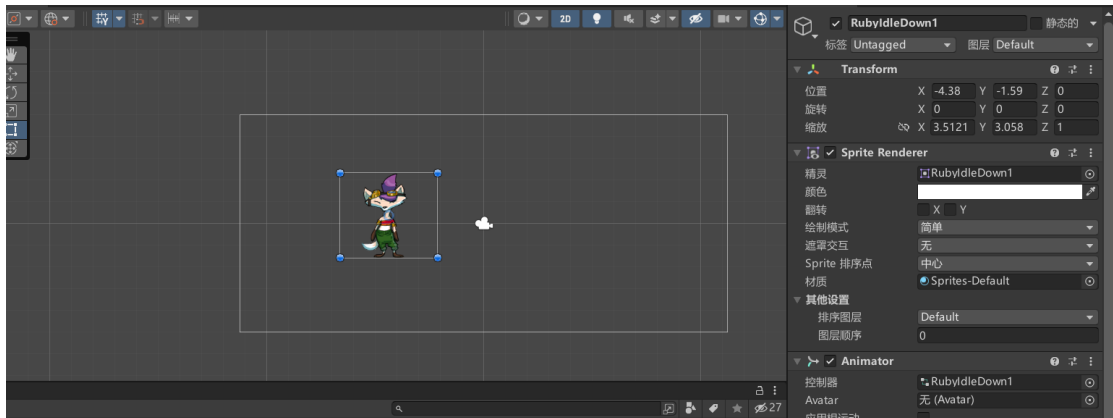
①导入资源包



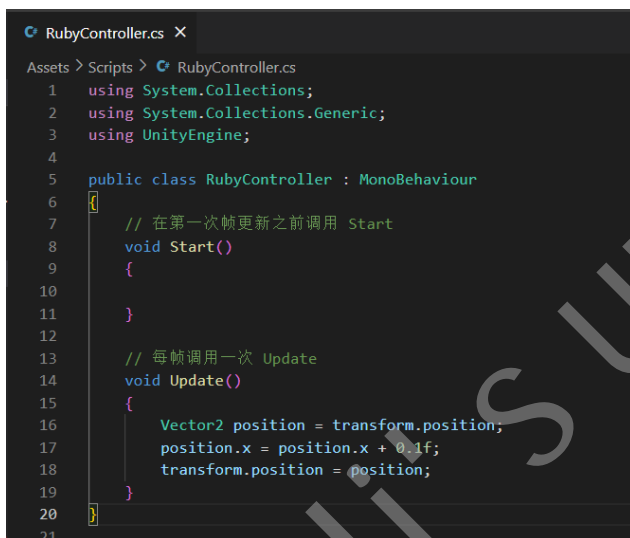
将资源加载到场景中



添加 Ruby



添加初始移动脚本



(二) 角色控制与键盘输入

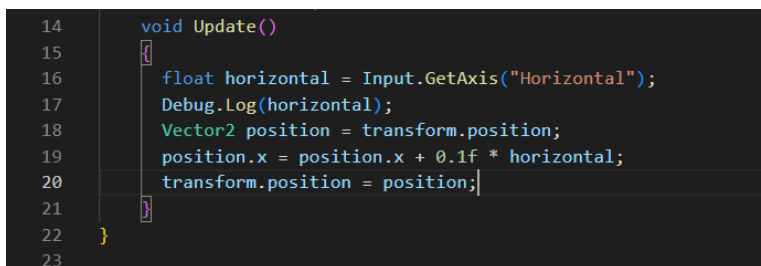
修改代码以使用键盘输入移动

我首先更改代码，让角色水平移动。

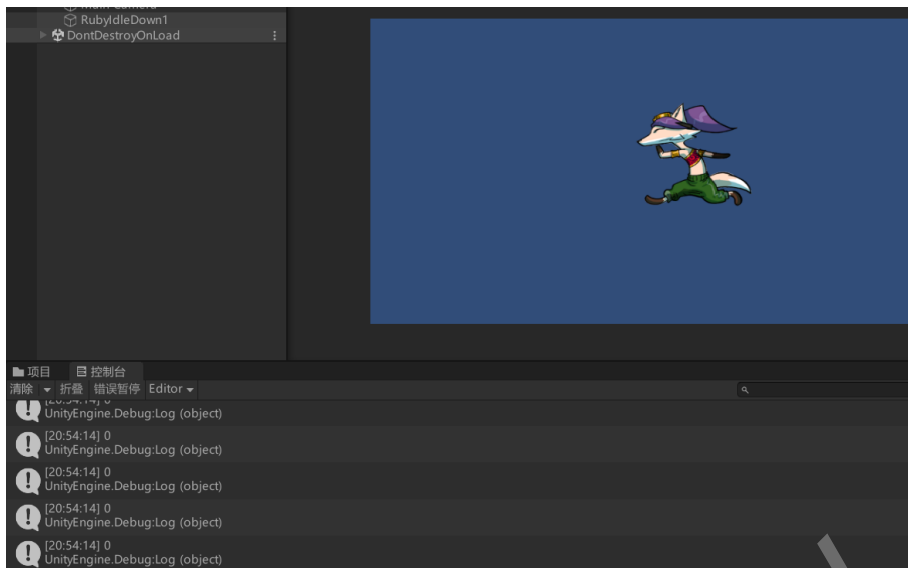
此次不是像上一教程中的代码那样每帧向 x 坐标加 0.1，而是仅在玩家按下键盘上的向右箭头键时才加 0.1。

玩家按向左箭头键时，应该减小 0.1，以便对象向左移动。

为此，我将使用 Unity 输入系统。该系统包含输入设置和输入代码（由 Unity 提供给我用于查询该帧某个轴的值）。



角色成功跑了起来：



修改代码以使用轴

添加垂直轴并以单位/秒表示 Ruby 的移动速度

```
14 void Update()  
15 {  
16     float horizontal = Input.GetAxis("Horizontal");  
17     float vertical = Input.GetAxis("Vertical");  
18     Vector2 position = transform.position;  
19     position.x = position.x + 3.0f * horizontal * Time.deltaTime;  
20     position.y = position.y + 3.0f * vertical * Time.deltaTime;  
21     transform.position = position;  
22 }
```

调用 `GetAxis` 函数

✧ `GetAxis` 与我在上一教程中遇到的 `Start` 和 `Update` 一样也是函数。

此处的区别在于我无需编写函数内部的代码，因为 **Unity** 工程师已经为我完成此工作。我在脚本中书写此类函数的名称时，就是在调用函数。

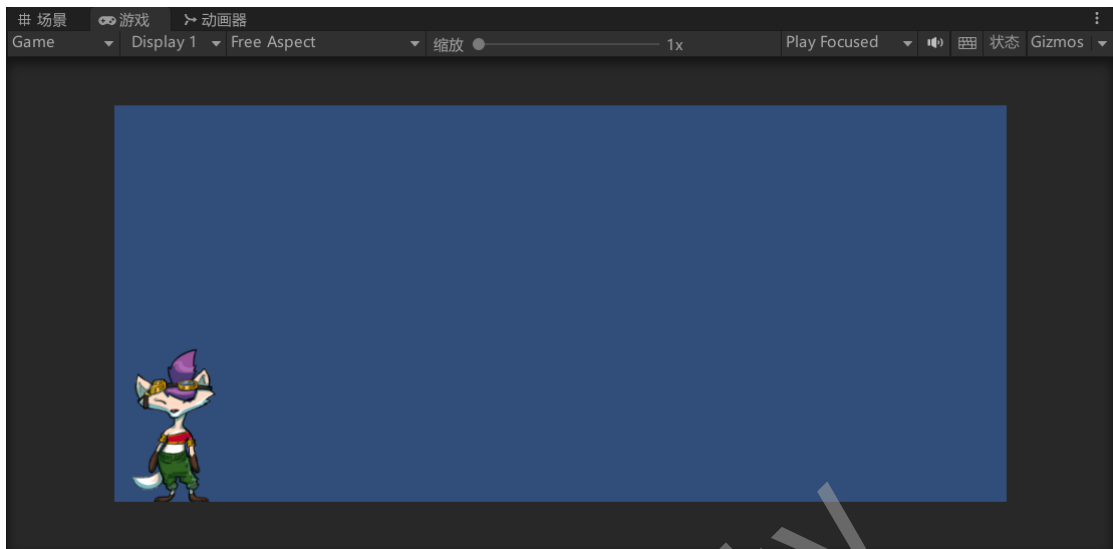
提供参数

✧ 圆括号之间的单词被称为**参数**。向函数提供此信息的目的是指定函数应执行的操作。在此示例中，我希望获得一个轴的值，所以告诉了函数这个轴的名称。我需将名称放在引号之间，以便编译器知道这是一个单词，而不是一个特殊关键字（例如变量名或类型）。

测试我的更改

1. 返回到 **Unity** 编辑器。
2. 按 **Play** 以进入运行模式。
3. 按下键盘上的向左和向右箭头键。我按下这些键时，Ruby 应该朝着正确方向移动，而不按下这些键时，Ruby 就不应移动。
4. 查看 **Console** 窗口。现在，按向左或向右箭头键时，该窗口中应该会填入变化的值：

此时 Ruby 可以移动到左下角了

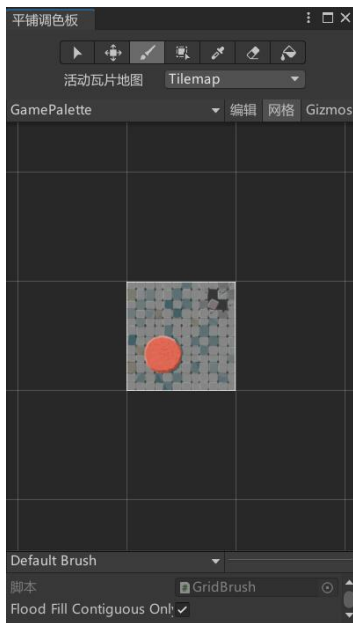


(三) 世界设计-瓦片地图

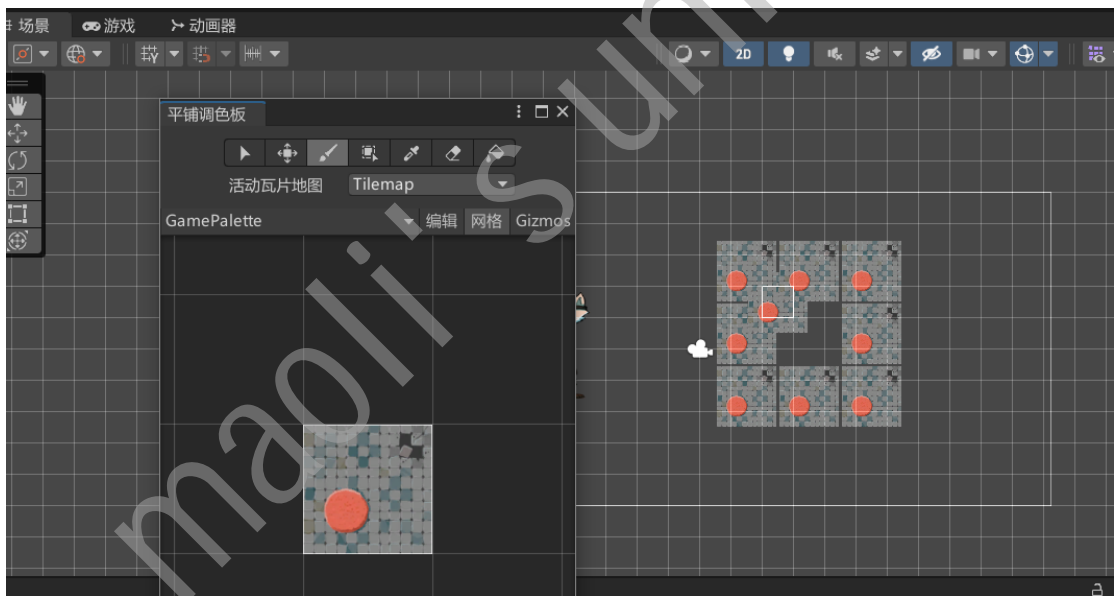
添加 firsttile



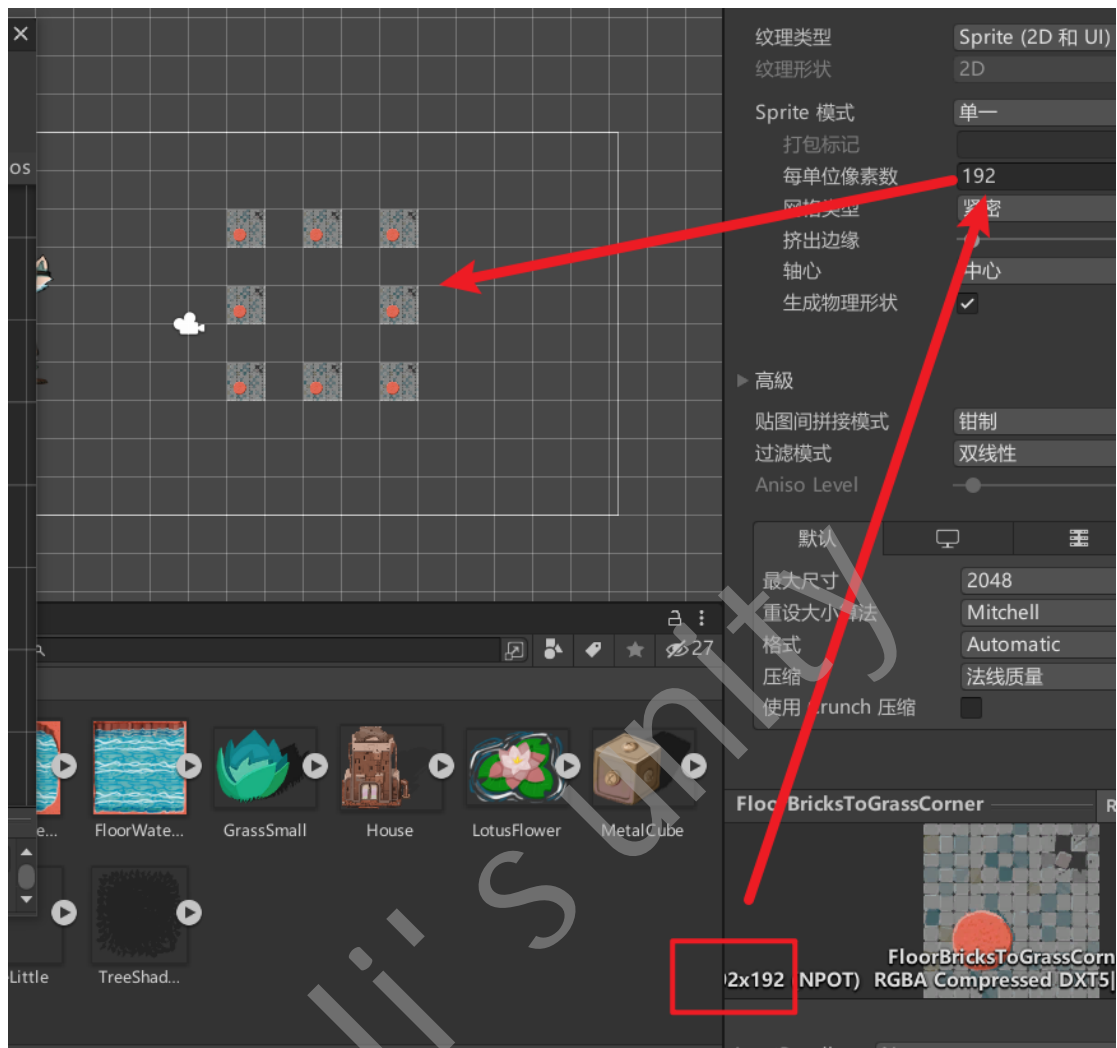
将 FirstTile 添加到面板



在 Scene 视图内的网格上进行绘制



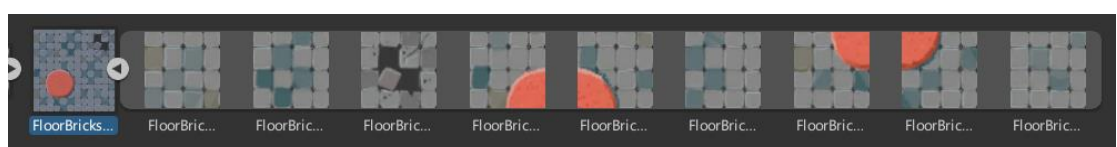
更改 Pixel Per Unit (PPU) 值之后，单击 Inspector 底部的 Apply。现在所有精灵都恰好填满网格



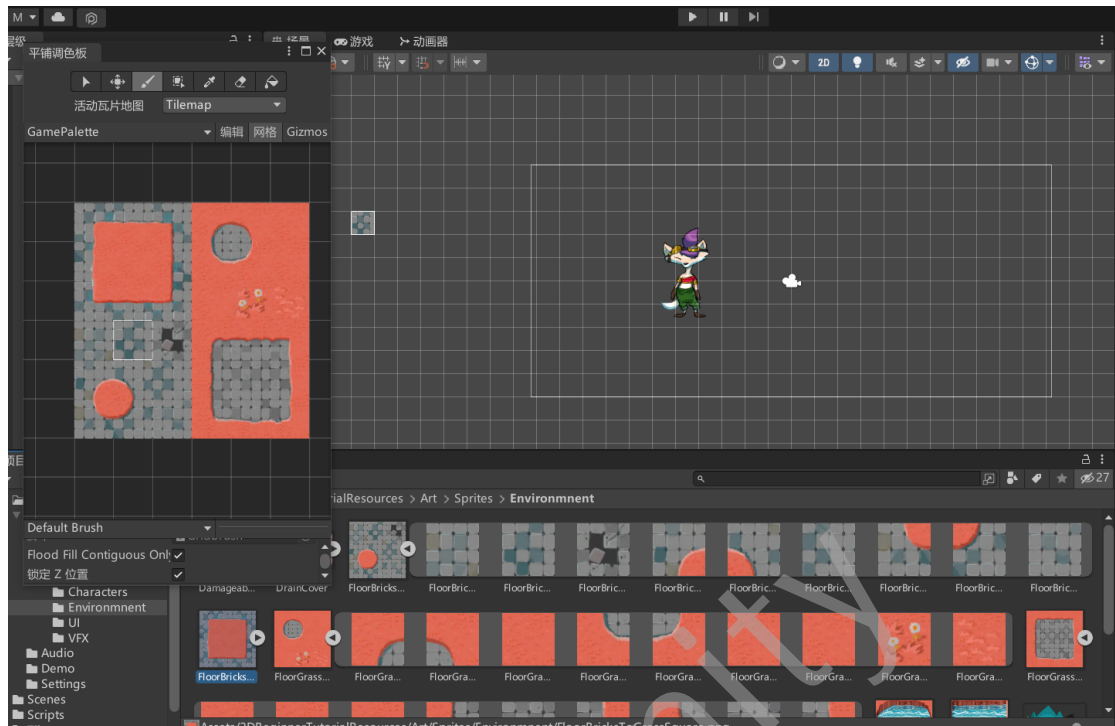
调整瓦片集



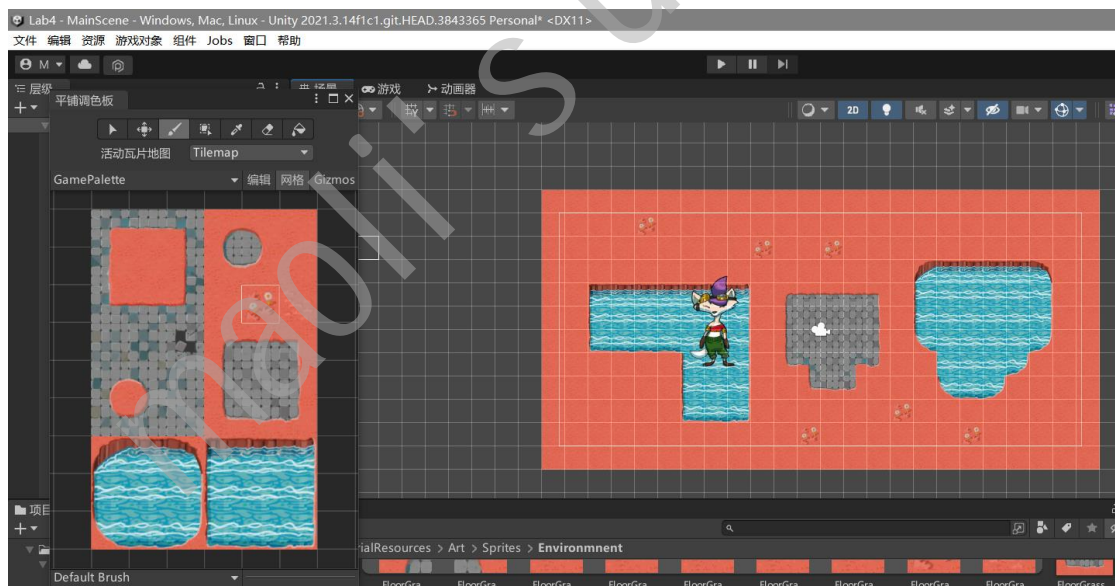
将精灵切片



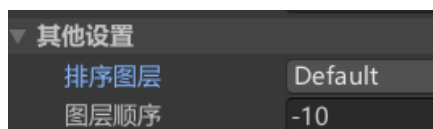
添加到 Palette 窗口，并重复操作，得到最后可以自由绘制的界面



绘制瓦片地图



设置图层顺序，保证地图在最底部



(四) 装饰世界

添加装饰

1. 在 Project 窗口中, 选择 Assets > Art > Sprites > Environment。选择 MetalCube 精灵。
2. 将 MetalCube 拖到 Hierarchy 窗口 以将其添加到场景中。
3. 使用移动工具 (快捷键: w), 将 MetalCube 置于场景所需的位置。
4. 按 Ctrl + S (Windows/Linux) 或 Cmd + S (macOS) 来保存更改。
5. 单击 Play 以进入运行模式, 并围绕着立方体移动我的角色!



更改图形设置

要指示 Unity 根据游戏对象的 y 坐标来绘制游戏对象, 请执行以下操作:

1. 选择 Edit > Project Settings。
2. 在左侧类别菜单中, 单击 Graphics。
3. 在 Camera Settings 中, 找到 Transparency Sort Mode 字段。此字段决定了精灵的绘制顺序。使用下拉菜单将此设置从 Default 更改为 Custom Axis。



4. 在 Transparency Sort Axis 中添加以下坐标:

x = 0

y = 1

z = 0

此设置告诉 Unity 在 y 轴上基于精灵的位置来绘制精灵。

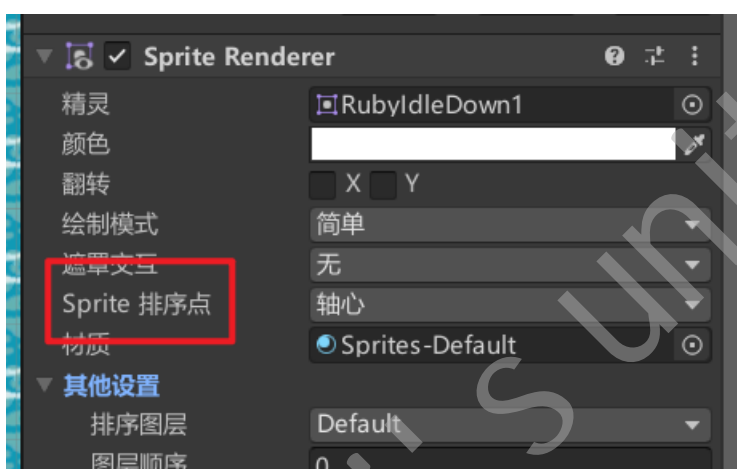
5. 关闭 Project Settings 窗口并保存我的更改。
6. 按 Play 以进入运行模式并测试我的更改。现在, 我的角色比箱子高时, 角色应该会绘制在箱子的后面; 而角色比箱子低时, 绘制在箱子的前面。
这只是一个开始, 但并不完美, 因为在箱子后面 (而不是前面) 绘制 Ruby 的时机似乎过早。为此解决此问题, 我需要调整 Sprite Renderer 组件。

成功实现角色比箱子高时, 角色应该会绘制在箱子的后面; 而角色比箱子低时, 绘制在箱子的前面。



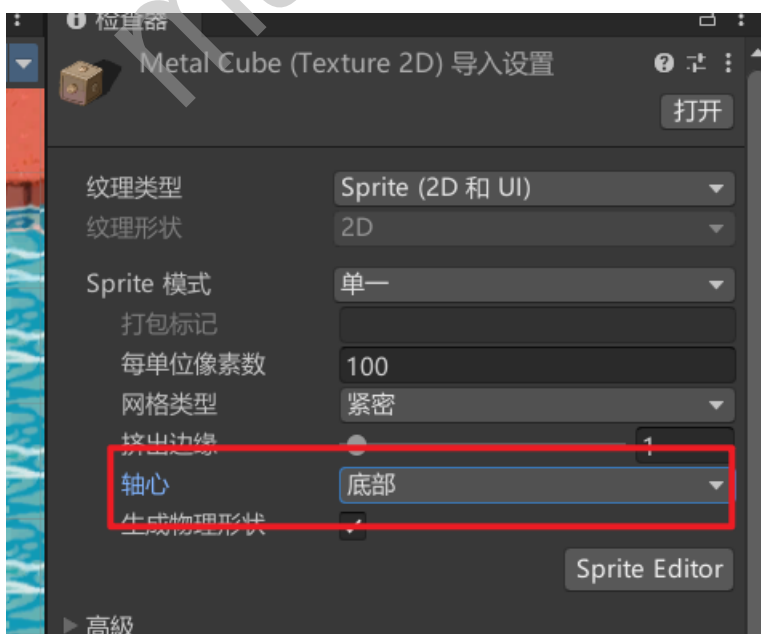
调整精灵设置

1. 在 Hierarchy 中，选择 Ruby 游戏对象。
2. 在 Inspector 中，找到该游戏对象的 Sprite Renderer 组件。
3. 找到 Sprite Sort Point 字段。目前，此字段设置为 Center，这意味着会使用精灵的中心点来决定这个游戏对象应该在另一个游戏对象的前面还是后面。
4. 将 Sprite Sort Point 更改为 Pivot。



调整单个精灵轴心

1. 在 Project 窗口中，选择 Assets > Art > Sprites > Environment。选择 MetalCube 精灵。
2. 在 Inspector 中，找到 Pivot 字段。使用下拉菜单将此字段设置为 Bottom。



使用 Sprite Editor 来更改轴心

更改轴心的另一种方法是使用 Sprite Editor:

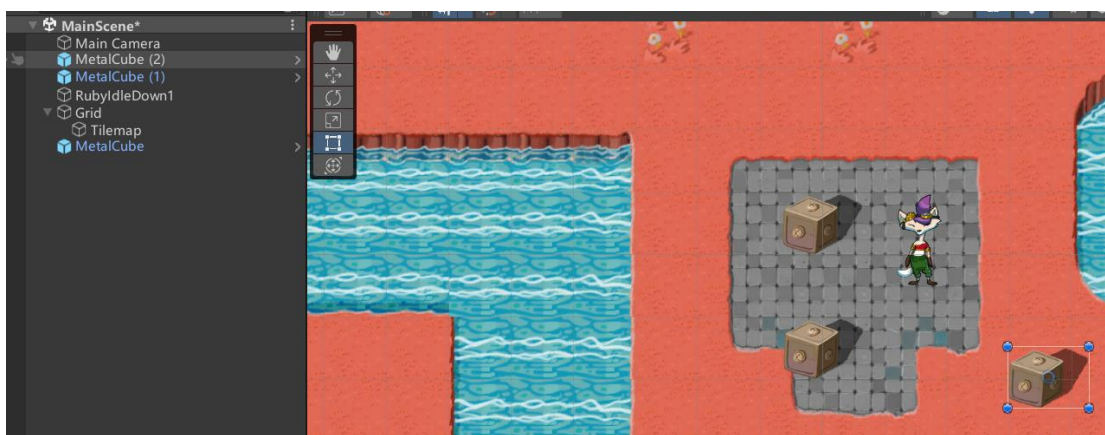
1. 在 Project 窗口中, 选择 Assets > Art > Sprites。选择 Ruby 精灵。
2. 在 Inspector 中, 单击 Sprite Editor 按钮。此时将打开我在先前教程中使用的 Sprite Editor。
3. 单击图像以显示精灵边框。我应该会在中心看到一个小的蓝色圆圈。这是轴心的当前位置。
4. 将 Custom Pivot 设置为 0.5 (x 轴) 和 0 (y 轴)。由于 Pivot Unit Mode 设置为 Normalized, 0 是最小值 (因此, x 为左侧, y 为底部), 1 是最大值 (x 为右侧, y 为顶部), 0.5 位于中间。
5. 单击 Apply 以保存我的更改。
6. 单击 Play 以进入运行模式, 然后试玩游戏。现在, 一切都应该可以正确显示!



将蓝色空心圆圈调整到底部

创建预制件

1. 在 Project 窗口中, 找到顶级文档夹 (Assets)。
2. 创建新文档夹并将其命名为 “Prefabs”。
3. 将 MetalBox 游戏对象从 Hierarchy 拖到新的 Prefab 文档夹中。



调整预制件设置

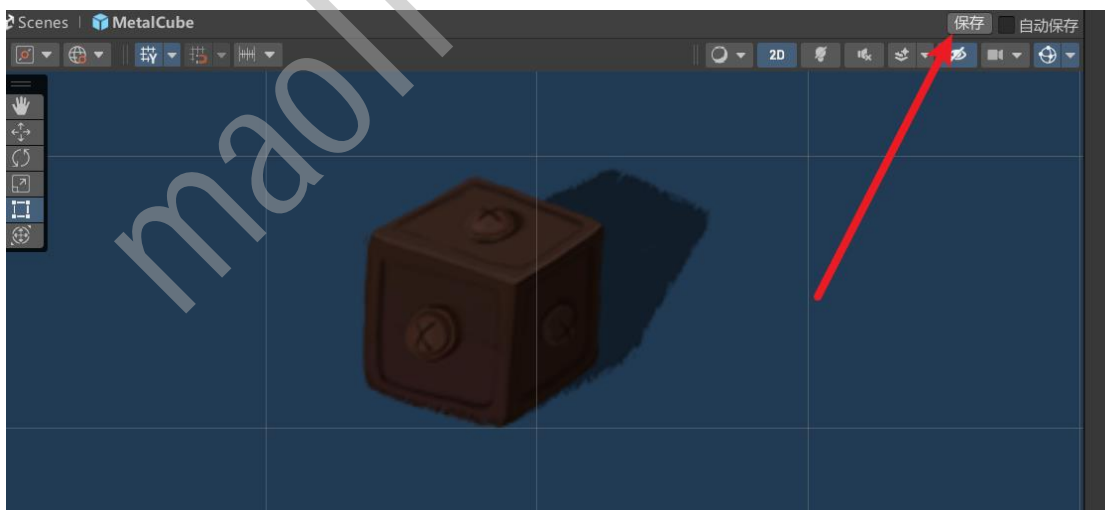
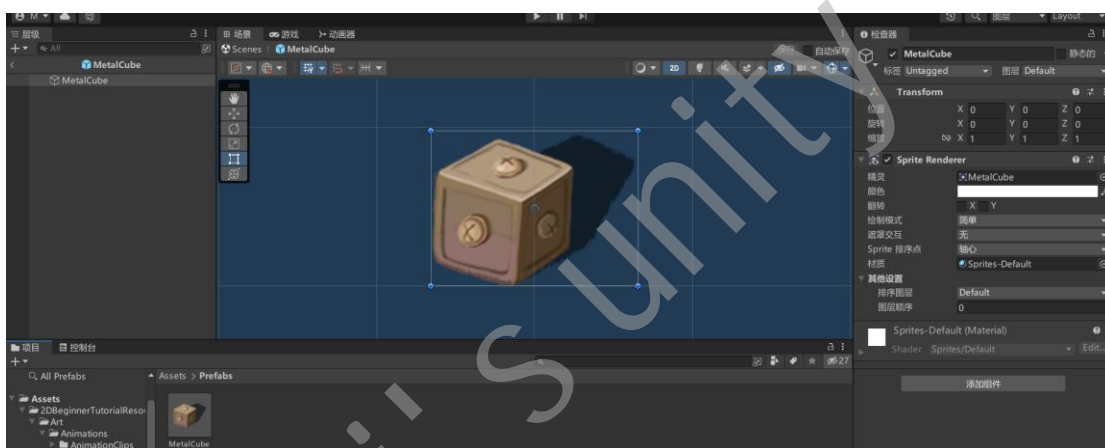
1. 在 Project 窗口中，双击 MetalBox 预制件。或者，选择预制件，然后在 Inspector 中单击 Open Prefab。我会发现 Scene 视图和 Hierarchy 现在仅包含金属箱。这是预制件模式。

2. 在 Inspector 中，找到 SpriteRenderer 组件中的 Color 字段。单击此字段，更改为其他颜色（例如，红色）。

3. 现在，我应该会看到 Scene 视图右上方的 Save 按钮不再灰显。这意味着我已经更改预制件，并且需要保存更改，因此请单击 Save 按钮。

注意：默认情况下已启用 Auto Save 按钮。这将确保我会自动保存执行的所有更改，如果未启用该功能，则必须按照上述说明来手动保存。

4. 在 Scene 视图中，单击 Scenes 导航路径以返回到所编辑的场景。或者，也可以单击 Hierarchy 中的箭头，沿着导航路径返回。

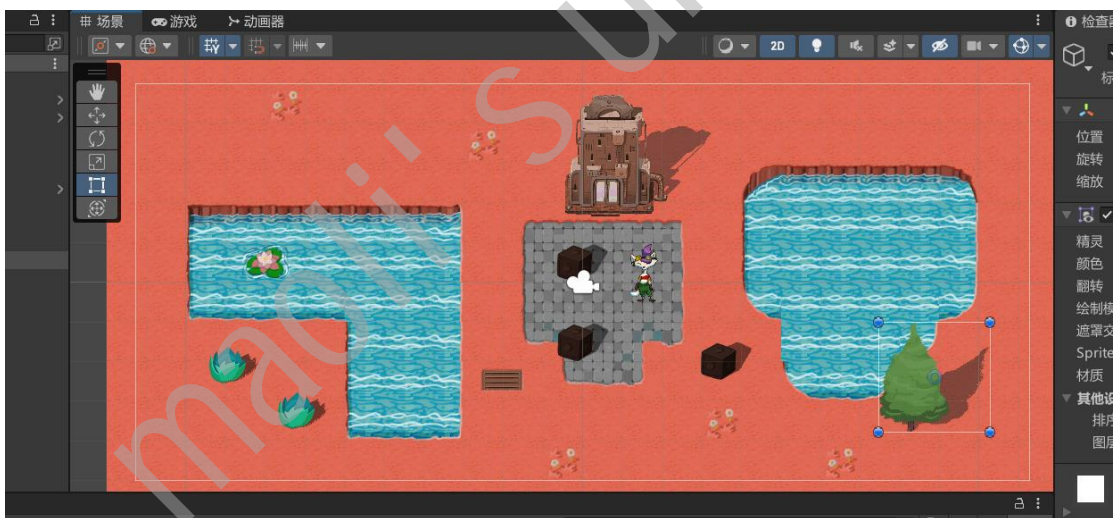




效果

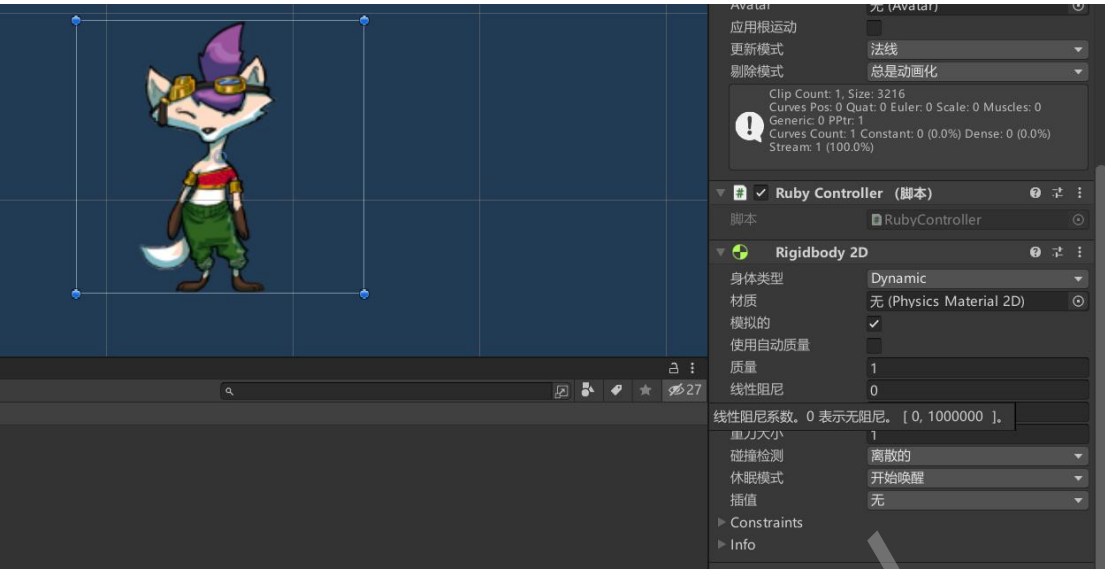
创意时刻

现在,我可以向游戏中添加自己的装饰!在 Art > Sprites > Environment 文档夹中,我会发现有很多项,例如房屋、树木和渠盖 (DrainCover)。



(五) 世界交互 - 阻止移动

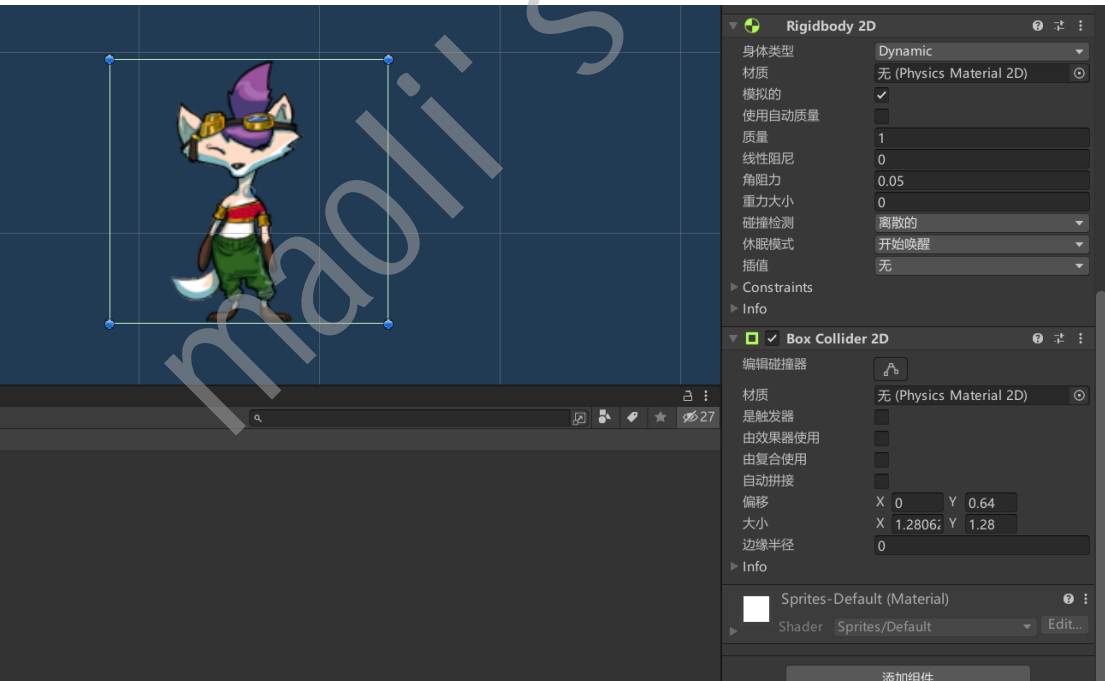
添加 Rigidbody 2D 组件

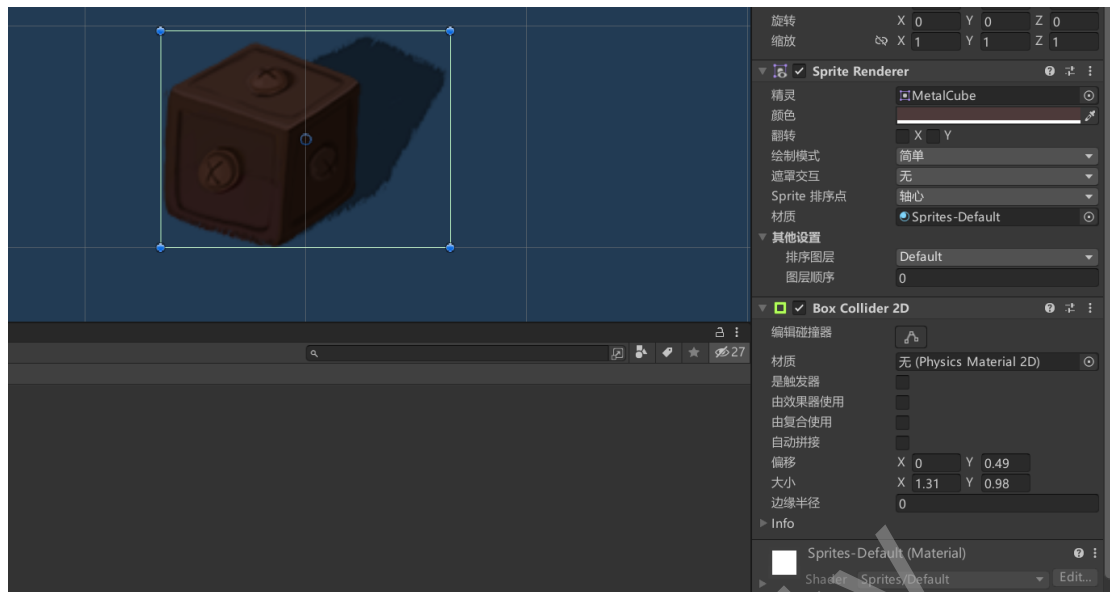


设置重力



向游戏对象添加碰撞体





无法穿过箱子



解决 Ruby 的旋转问题

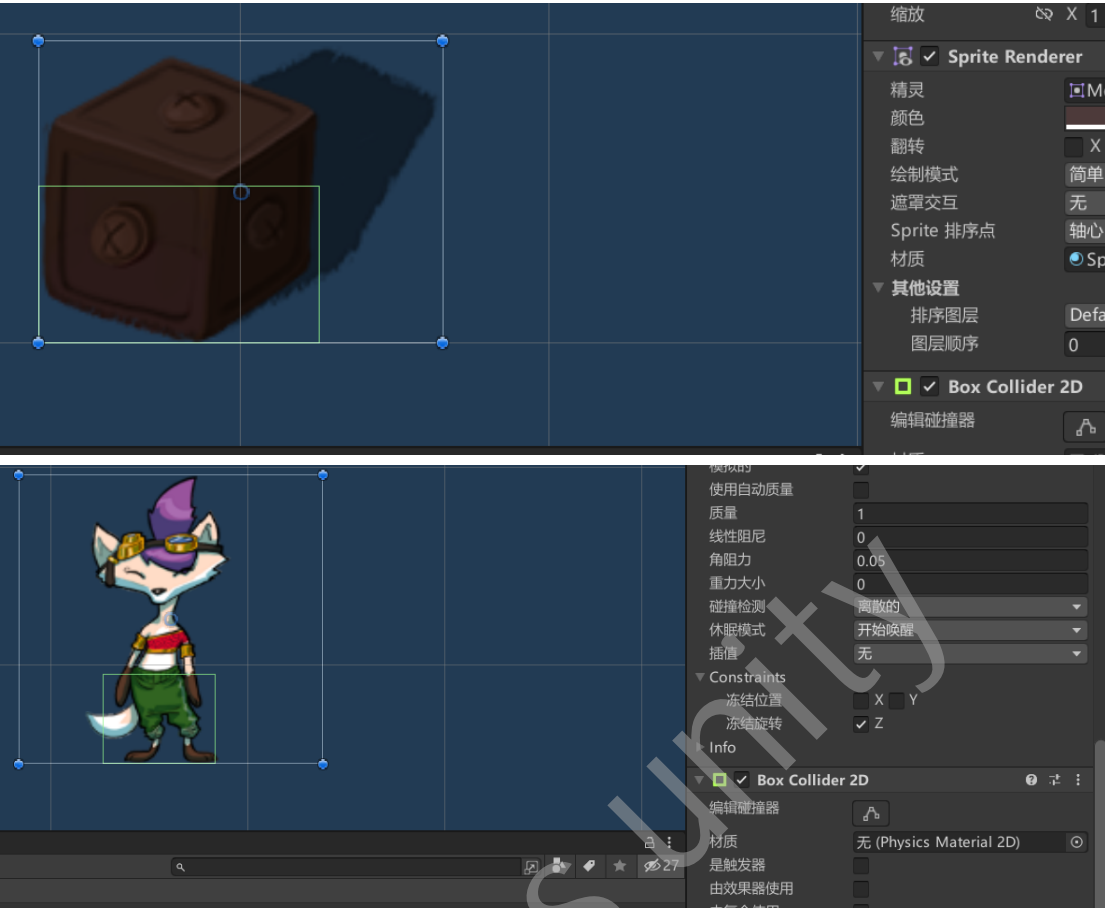


解决 Ruby 的抖动问题 (修改 controller)

```
RubyController.cs X
Assets > Scripts > RubyController.cs
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class RubyController : MonoBehaviour
6  {
7      Rigidbody2D rigidbody2d;
8      float horizontal;
9      float vertical;
10
11     // 在第一次帧更新之前调用 Start
12     void Start()
13     {
14         rigidbody2d = GetComponent<Rigidbody2D>();
15     }
16
17     // 每帧调用一次 Update
18     void Update()
19     {
20         horizontal = Input.GetAxis("Horizontal");
21         vertical = Input.GetAxis("Vertical");
22     }
23
24     void FixedUpdate()
25     {
26         Vector2 position = rigidbody2d.position;
27         position.x = position.x + 3.0f * horizontal * Time.deltaTime;
28         position.y = position.y + 3.0f * vertical * Time.deltaTime;
29
30         rigidbody2d.MovePosition(position);
31     }
32 }
```

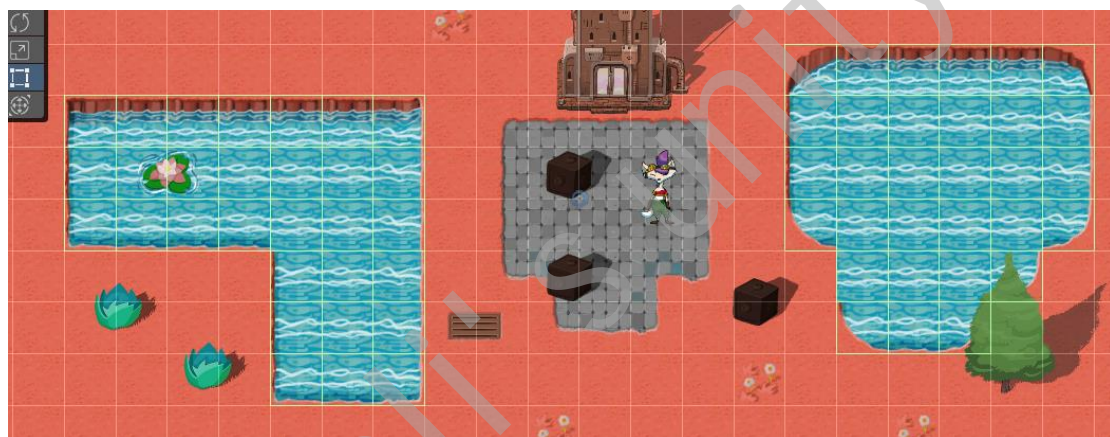
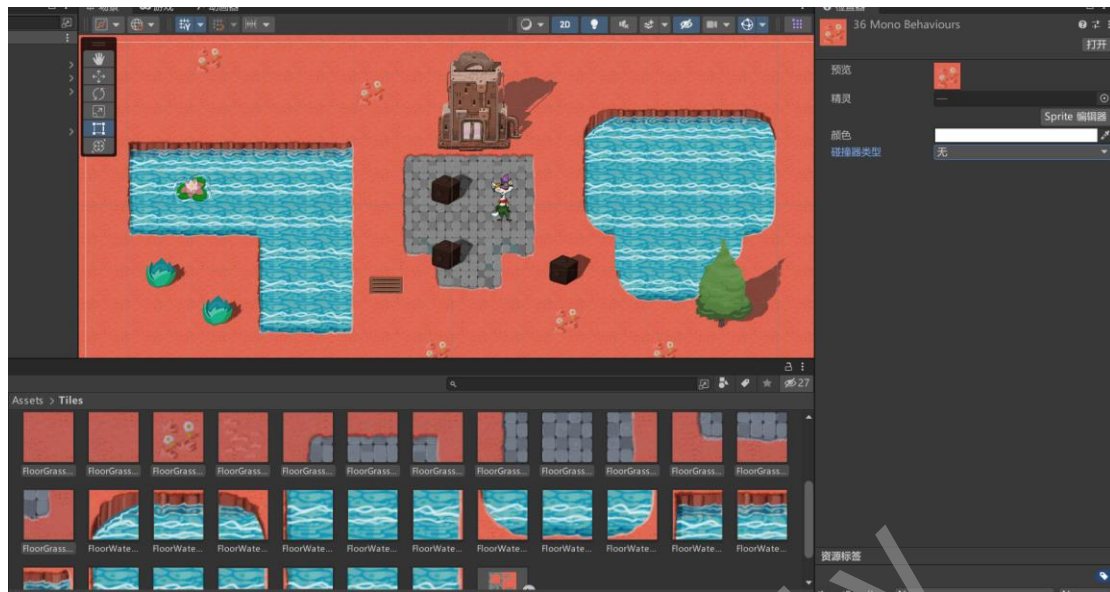
- Rigidbody2D rigidbody2d; 此行代码将创建一个新变量（名为 rigidbody2d）来存储刚体并从脚本中的任何位置访问刚体。
- float horizontal; float vertical; 这两行代码将创建两个新变量来存储输入数据。这些变量曾在 Update 函数中声明过，但是由于我现在需要从另一个函数(FixedUpdate) 访问这些变量，因此在此处又声明了这些变量。
- rigidbody2d = GetComponent<Rigidbody2D>(); 此代码位于 Start 函数内，因此在游戏开始时仅执行一次。此代码要求 Unity 向我提供与脚本附加到同一游戏对象（即我的角色）上的 Rigidbody2D。
- 在 Update 函数中，我删除了与移动相关的所有代码。我保留了用于读取输入的代码，这次是位于先前声明的两个变量中。
- Vector2 position = rigidbody2d.position; 在 FixedUpdate 函数中，我添加了曾位于 Update 函数中的代码行，并调整了此代码以使用刚体位置。
- rigidbody2d.MovePosition(position); 同样，我现在使用刚体位置，而不是使用 transform.position = position; 来设置新位置。这行代码会将刚体移动到我想要的位置，但是如果刚体在移动中与另一个碰撞体碰撞，则会中途停止刚体。

调整碰撞体的大小



添加瓦片地图碰撞





现在角色无法走到箱子和水上

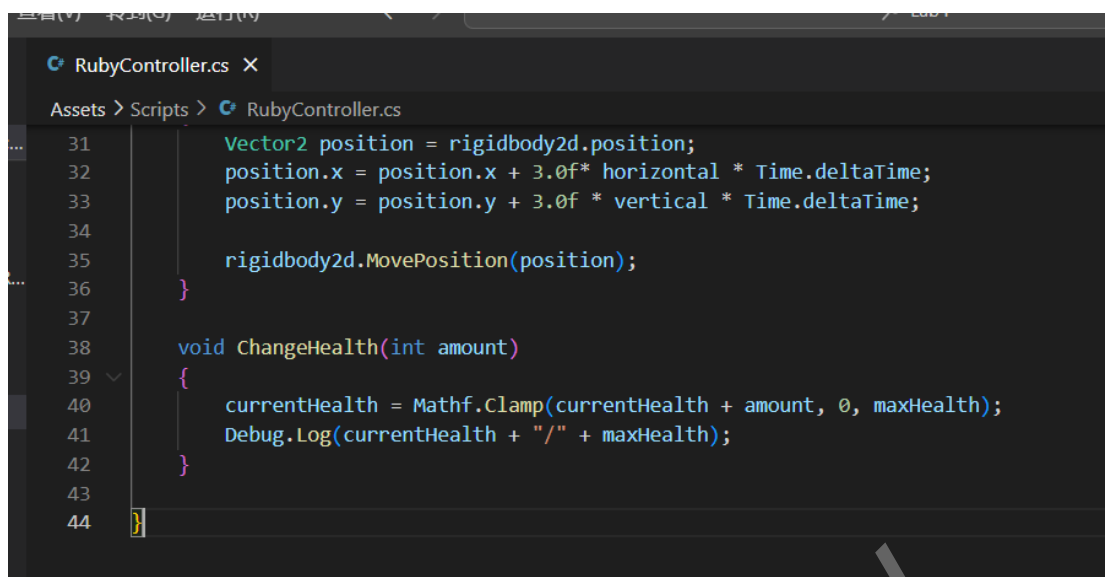


(六) 可收集对象

向 Ruby 添加生命值统计功能

修改我的角色脚本，向角色添加生命值：

1. 打开 RubyController 脚本。
2. 修改脚本：



maxHealth 变量存储 Ruby 可以拥有的最大生命值。

- 前缀为 public。我将在本教程的后面部分中找到有关此内容的更多信息。
- 类型为 int，这是我以前没遇到过的新类型。这种类型告诉计算机我要存储一个整数，即不带小数点的整数。Ruby 不能具有 4.325 生命值或 2.97412 生命值，在此示例中，角色的最大生命值为 5。

✧ 游戏开始时设置满血生命值

✧ 添加函数来更改生命值

ChangeHealth 函数这里使用了另一个名为 Mathf.Clamp 的内置函数来设置当前生命值。这是因为，在 Ruby 处于生命值满血状态时，如果我尝试将生命值增大 2，生命值便会超过最大值。

同样，如果 Ruby 剩下的生命值为 1，而我尝试减少 2，则 Ruby 的生命值将为负。

钳制功能 (Clamping) 可确保第一个参数 (此处为 currentHealth + amount) 绝不会小于第二个参数 (此处为 0)，也绝不会大于第三个参数 (maxHealth)。因此，Ruby 的生命值将始终保持在 0 与 maxHealth 之间。

```
void ChangeHealth(int amount)
{
    currentHealth = Mathf.Clamp(currentHealth + amount, 0, maxHealth);
    Debug.Log(currentHealth + "/" + maxHealth);
}
```

✧ 在 Console 窗口中显示生命值

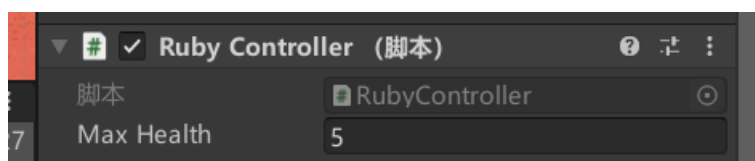
通过添加相关代码，借助 Debug.Log 在 Console 窗口中显示当前生命值。每次生命值变化时，此脚本都会更新控制台。

在我们的游戏中没有任何图形或文本可以显示 Ruby 的生命值，因此我可以检查控制台来查看一切是否正常。

为了更容易在控制台中读取生命值，我使用了 + 将字符串合并为一个方便阅读的短语。在此示例中，我在 currentHealth 和 maxHealth 之间插入了一个斜杠 (/)。

现在让我们回到 Unity 编辑器，看看 Unity 如何编译对脚本所做的更改，并查看我的角色。

在 Unity 编辑器中检查我的更改



脚本公开了一个 Max Health 属性。这是因为我在 maxHealth 变量前面添加了“public”。public 意味着我可以从脚本外部访问变量，因此 Unity 会将变量显示在 Inspector 中。此变量当前设置为 5，因为这是我在脚本中定义的默认值。

创建可收集的生命值游戏对象

1. 首先，重复我在上一教程中遵循的所有步骤以制作金属箱：

在 Project 窗口中，选择 Assets > Art > Sprites > VFX，并找到 CollectibleHealth。

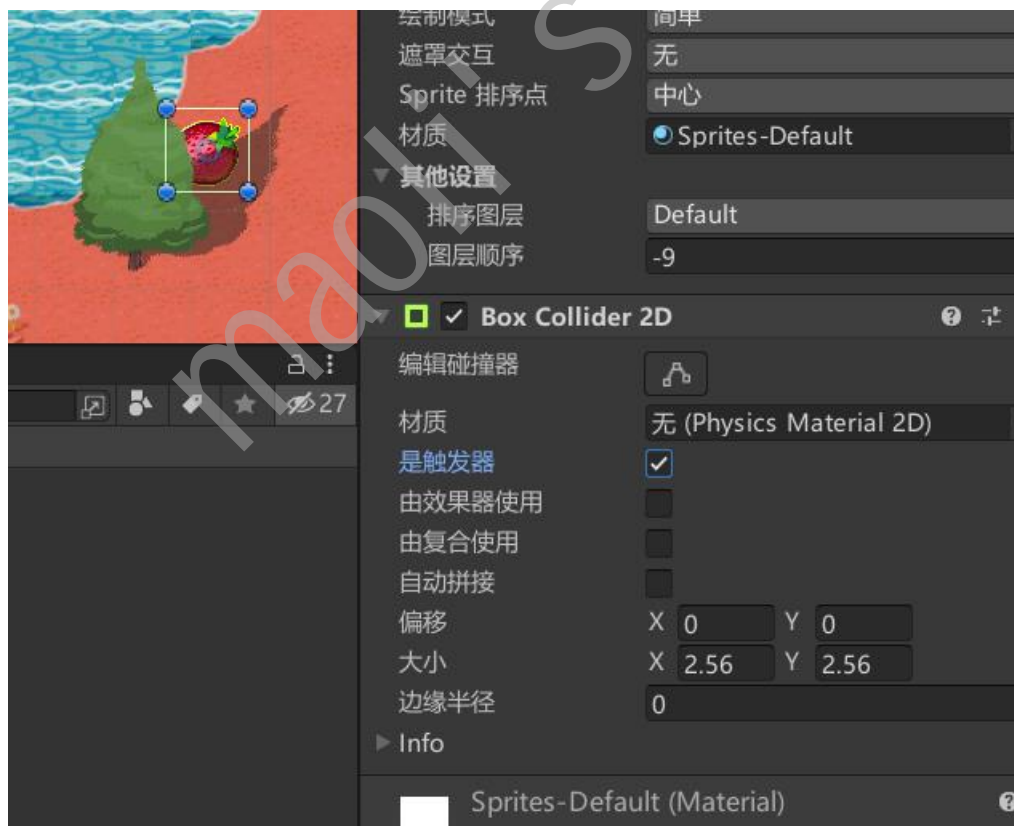
将这个游戏对象导入到场景中，并调整 PPU 值以设置为适当大小。

将 Box Collider 2D 组件添加到新游戏对象，调整碰撞体大小，以便更好地适应精灵。

2. 现在，如果我单击 Play，就像 Ruby 会与箱子碰撞一样，她也会与生命值可收集对象碰撞。但是，这不是我需要的效果。

3. 退出运行模式。

4. 在 Inspector 中，找到 Box Collider 2D 组件。启用 Is Trigger 属性复选框。

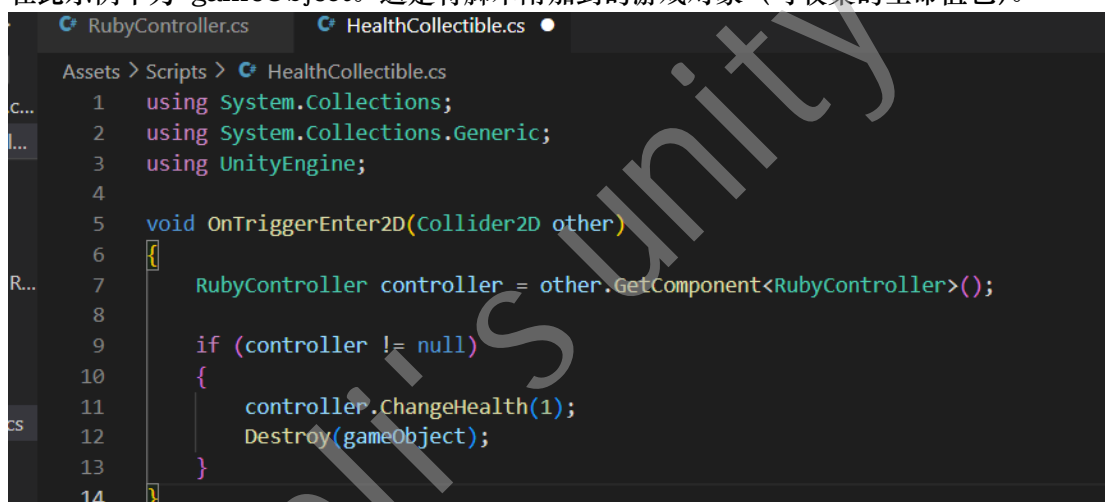


创建可收集对象脚本

现在，我可以添加代码来处理碰撞：

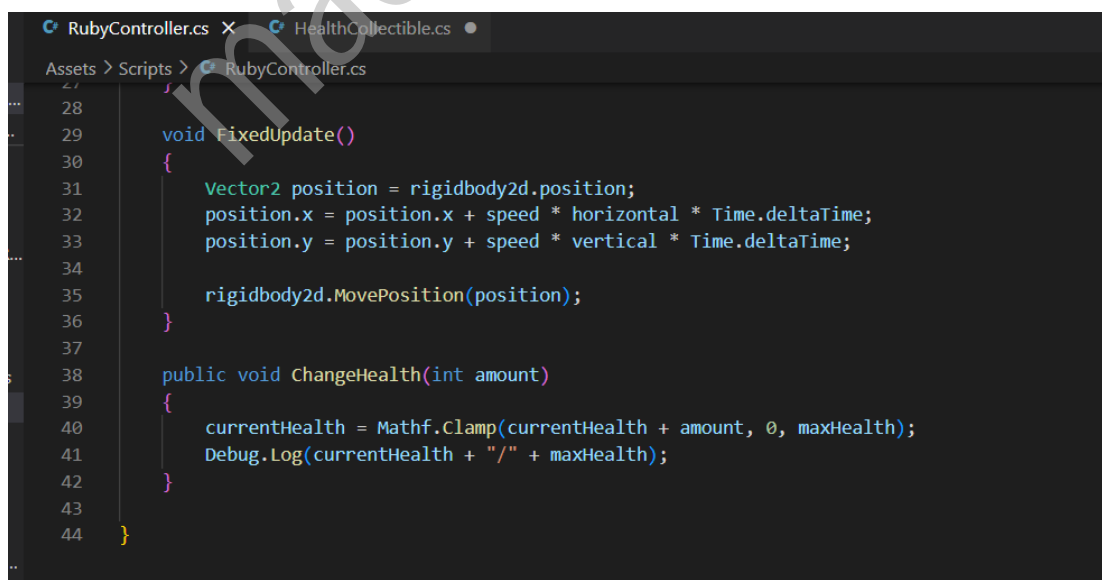
1. 在 Project 窗口中, 选择 Assets > Scripts。
 2. 右键单击, 然后选择 Create > C# script。
 3. 将新脚本命名为 HealthCollectible。
 4. 在 Hierarchy 中, 选择 CollectibleHealth 游戏对象。将 HealthCollectible 脚本从 Project 窗口拖放到 Inspector, 从而将这个脚本作为组件添加到游戏对象。
 5. 双击脚本文档以便在脚本编辑器中打开此文档。
 6. 删除 Start 和 Update 函数, 因为我不想在游戏开始时或在每一帧处理任何事务。
 7. 我希望脚本检测 Ruby 与可收集的生命值游戏对象发生碰撞的情况, 并向她提供一些生命值。为此, 请使用以下特殊函数名称:
- ✧ 为 Ruby 提供生命值

在 if 代码块内 (if 条件行后花括号内的代码), 我使用了先前编写的函数将 RubyController 的生命值更改 1。然后, 我调用了 Destroy (gameObject)。Destroy 是 Unity 的一个内置函数, 可销毁我作为参数传递给这个函数的任何对象; 在此示例中为 gameObject。这是将脚本附加到的游戏对象 (可收集的生命值包)。



```
Assets > Scripts > HealthCollectible.cs
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 void OnTriggerEnter2D(Collider2D other)
6 {
7     RubyController controller = other.GetComponent<RubyController>();
8
9     if (controller != null)
10    {
11        controller.ChangeHealth(1);
12        Destroy(gameObject);
13    }
14 }
```

调整 RubyController 脚本



```
Assets > Scripts > RubyController.cs
28
29 void FixedUpdate()
30 {
31     Vector2 position = rigidbody2d.position;
32     position.x = position.x + speed * horizontal * Time.deltaTime;
33     position.y = position.y + speed * vertical * Time.deltaTime;
34
35     rigidbody2d.MovePosition(position);
36 }
37
38 public void ChangeHealth(int amount)
39 {
40     currentHealth = Mathf.Clamp(currentHealth + amount, 0, maxHealth);
41     Debug.Log(currentHealth + "/" + maxHealth);
42 }
43
44 }
```

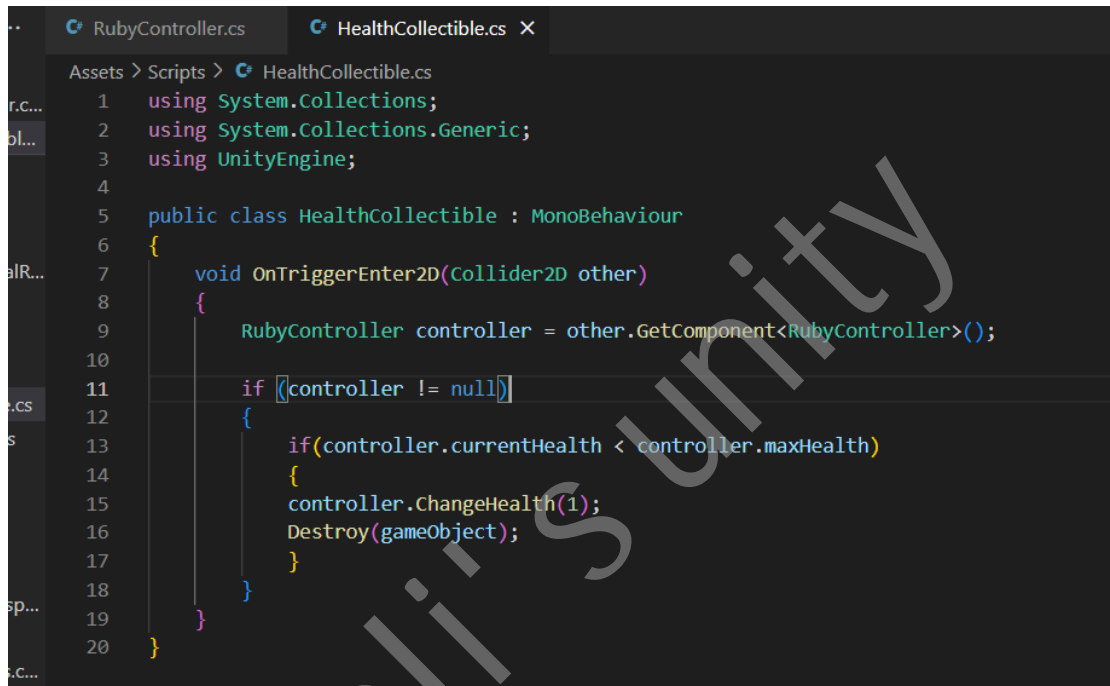
要允许 HealthCollectible 访问该函数, 请找到 RubyController 脚本并执行以下操作: 在 ChangeHealth 函数声明中的 “void” 之前添加 “public”。

在 Start 函数中，将 currentHealth 设置为 1，使我的角色最初具有较小的生命值。

检查 Ruby 是否需要生命值

现在，如果我在 Ruby 的生命值满血时拾取一个生命值可收集对象，脚本仍将销毁该可收集对象。这是因为根据我的设置，当角色进入触发器时，无论如何都要删除可收集对象。为了避免这种情况，我可以添加一项检查以便在销毁生命值可收集对象之前先查看 Ruby 是否需要生命值：

1. 打开 HealthCollectible 脚本。
2. 如下所示更改脚本：



```
Assets > Scripts > HealthCollectible.cs
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class HealthCollectible : MonoBehaviour
6  {
7      void OnTriggerEnter2D(Collider2D other)
8      {
9          RubyController controller = other.GetComponent<RubyController>();
10
11         if (controller != null)
12         {
13             if (controller.currentHealth < controller.maxHealth)
14             {
15                 controller.ChangeHealth(1);
16                 Destroy(gameObject);
17             }
18         }
19     }
20 }
```

在 RubyController 中定义一个属性

```
public int health { get { return currentHealth; } }
```

像变量一样的属性定义：

- ✧ 访问级别 (public)
- ✧ 类型 (int)
- ✧ 名称 (health)

但是，此处添加了两个 {} 代码块，而未使用 ; 来结束代码行。

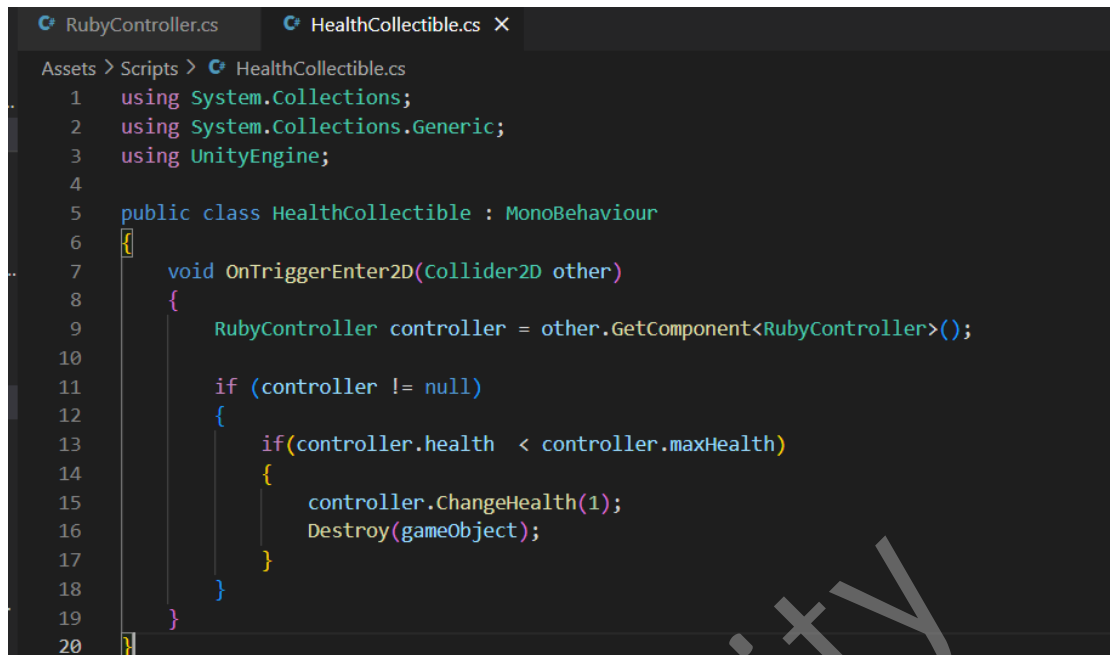
在第一个代码块中，我使用了 get 关键字来获取第二个代码块中的任何内容。

第二个代码块就像普通函数一样，因此只需返回 currentHealth 值。

编译器完全像函数一样处理此代码行，因此我可以在 get 代码块内的函数中编写所需的任何内容（例如声明变量、执行计算和调用其他函数）。

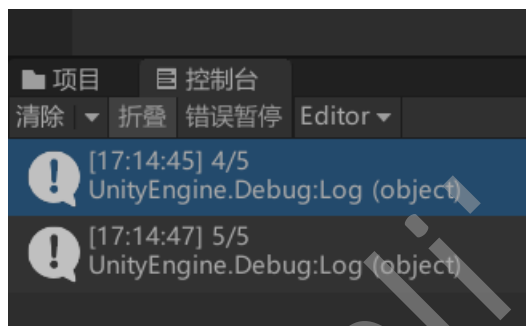
使用 HealthCollectible 中的属性

属性的用法很像变量，而不是像函数。此处，health 向我提供 currentHealth。但是，只有我读取 health 时，才有这样的作用。



```
RubyController.cs HealthCollectible.cs X
Assets > Scripts > HealthCollectible.cs
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class HealthCollectible : MonoBehaviour
6 {
7     void OnTriggerEnter2D(Collider2D other)
8     {
9         RubyController controller = other.GetComponent<RubyController>();
10
11         if (controller != null)
12         {
13             if (controller.health < controller.maxHealth)
14             {
15                 controller.ChangeHealth(1);
16                 Destroy(gameObject);
17             }
18         }
19     }
20 }
```

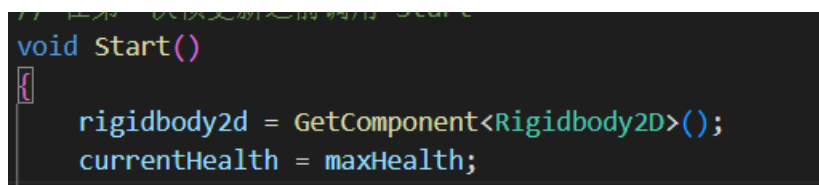
测试生命改变:



(七) 伤害区域和敌人

重置 Ruby 的生命值

1. 打开 RubyController 脚本。
2. 确认 Start 函数已将 currentHealth 设置为 maxHealth 而不是 1：
currentHealth = maxHealth;



```
void Start()
{
    rigidbody2d = GetComponent<Rigidbody2D>();
    currentHealth = maxHealth;
}
```

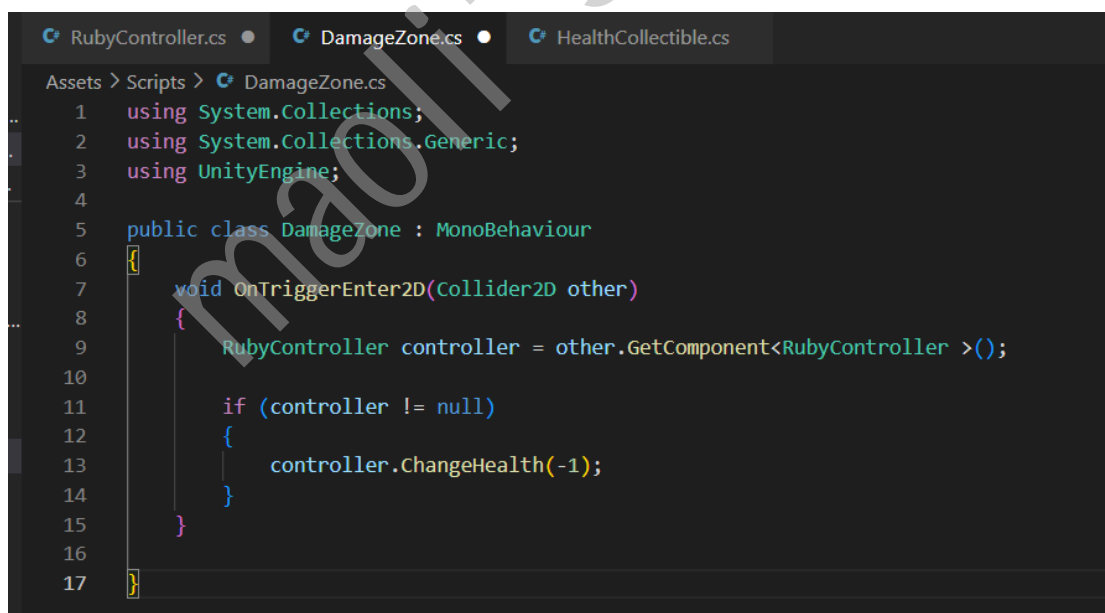
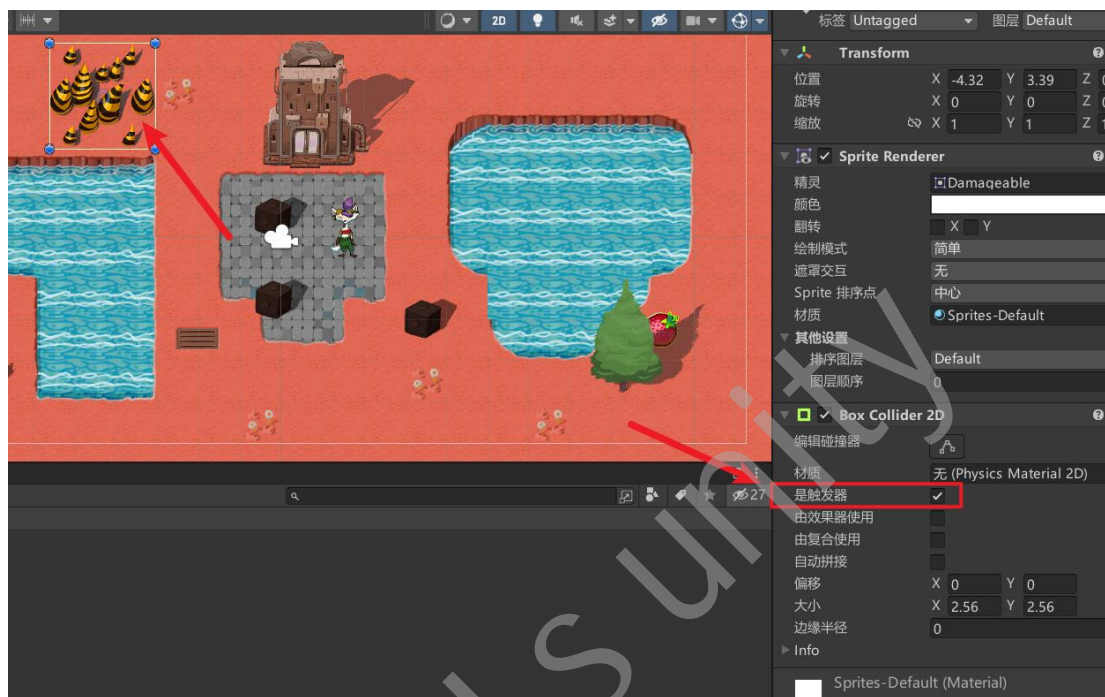
添加伤害区域

1. 从 Damageable 精灵创建一个新的游戏对象。我可以从以下方法中选择：
将精灵拖放到 Hierarchy 窗口中。
新建一个游戏对象，然后添加 Sprite Renderer 组件并分配该精灵。
2. 将 Box Collider 2D 组件添加到 Damageable 游戏对象，然后调整盒子的大小以适应

精灵，并在 Inspector 中启用 Is Trigger 复选框。

3. 创建一个名为 DamageZone 的新脚本。在 Project 窗口中，找到 Assets > Scripts 文档夹。右键单击，然后选择 Create > C# Script。

4. 在 Project 窗口中双击该脚本以在代码编辑器中打开脚本，然后将以下代码复制到新脚本中以替换其中已有的类。DamageZone 脚本与上一教程的 Collectable 脚本完全一样，只是这里的生命值变化为 -1，而且删除了对 Destroy 的调用：



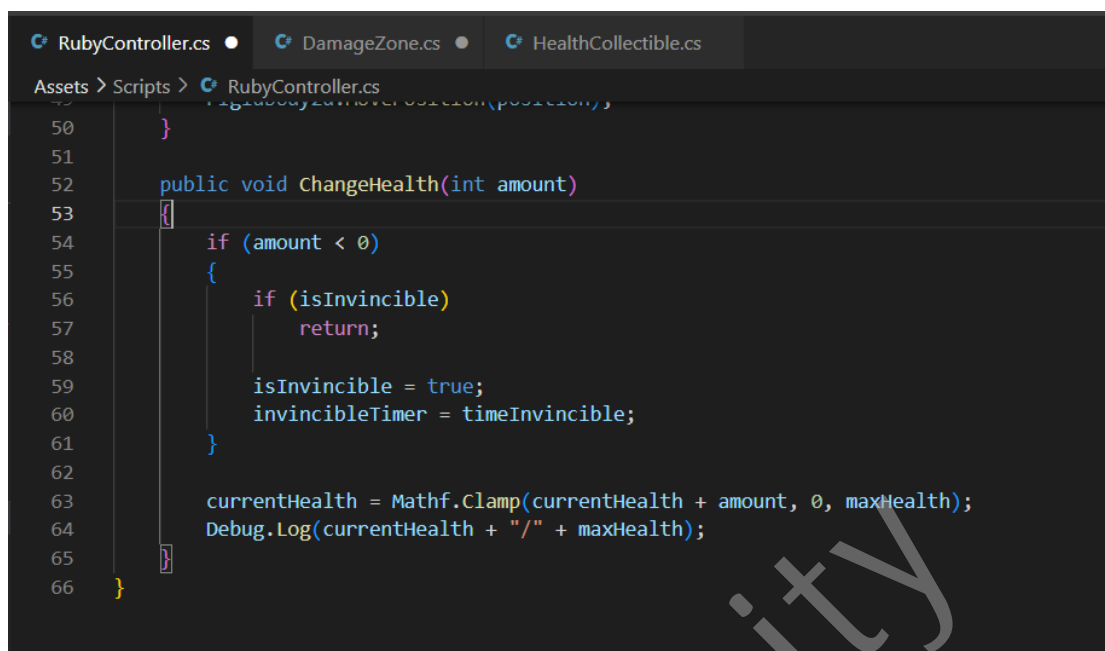
这很好，但只有在角色进入区域时才会伤害角色。如果角色停留在区域内，则不会再伤害角色。可以通过将函数名称从 OnTriggerEnter2D 更改为 OnTriggerStay2D 来解决此问题。刚体在触发器内的每一帧都会调用此函数，而不是在刚体刚进入时仅调用一次。

休眠模式

从不休眠

为了优化资源，物理系统在刚体停止移动时会停止计算刚体的碰撞；此时刚体进入“睡眠状态”。但在我这个情况中，我希望始终进行计算，因为即使在 Ruby 停止移动时也需要检

测她是否受到伤害，因此我要指示刚体永远不要进入睡眠状态。



添加三个新变量：

- ✧ 变量 1：一个名为 timeInvincible 的公共浮点变量。将此变量设置为 public 是因为我希望能够在 Inspector 中动态更改变量以调整该值。
- ✧ 变量 2：一个名为 isInvincible 的私有 bool 变量，用于存储当前是否处于无敌状态。bool (boolean 的缩写) 可让我们存储“true”或“false”。这在 if 语句中特别有用。
- ✧ 变量 3：一个名为 invincibleTimer 的私有浮点变量。此变量将存储 Ruby 在恢复到可受伤状态之前剩下的无敌状态时间。

修改一些函数：

- ✧ ChangeHealth 函数

在 ChangeHealth 函数中，我添加了一项检查以查看当前是否正在伤害角色（换言之，如果变化小于 0，则表示减小生命值）。如果是这样，我首先要检查 Ruby 是否已经处于无敌状态，如果是，那么将退出该函数，因为她现在无法受到伤害。否则，由于 Ruby 正受到伤害，我将 isInvincible bool 设置为 true 并将 invincibleTimer 变量设置为 timeInvincible，从而使 Ruby 处于无敌状态。

- ✧ Update 函数

在 Update 函数中，如果 Ruby 处于无敌状态，则从计时器减去 deltaTime。这实际上是在倒计时。当该时间小于或等于零时，计时器结束，Ruby 的无敌状态也结束，因此通过将 bool 重置为 false 来消除她的无敌状态。这样，下次调用 ChangeHealth 来伤害 Ruby 时，我就不会提前退出并再次伤害她、重置她的无敌状态等等。

让我们进入运行模式并测试我的脚本。如果我让 Ruby 停留在伤害区域，她应该每两秒才受到伤害，因为我将无敌时间设置为两秒。尝试在 Inspector 中使用不同的值，必要时更改时间。

有关图形的旁注

可以让 Sprite Renderer 平铺（而不是拉伸）精灵。因此，如果将伤害区域的大小调整到足以将精灵容纳两次，则 Sprite Renderer 会多次并排绘制精灵：



首先，确保游戏对象的缩放在 Transform 组件中设置为 1,1,1。
然后在 Sprite Renderer 组件中将 Draw Mode 设置为 Tiled，并将 Tile Mode 更改为 Adaptive。



在 Project 窗口中选择 Damageable 精灵，并将 Mesh Type 更改为 Full Rect。



敌人

从本教程顶部的资料中下载以下图像，将图像保存在计算机上，然后将图像导入 Unity 并放置在场景中。



来回移动

创建一个名为 `EnemyController` 的新脚本，并将这个脚本附加到敌人角色。现在，让我们编写一个脚本来使敌人循环上下移动。

我应该已经从前面的所有教程中充分学习到了此过程所需的知识，因此我可以自己尝试编写这个脚本，然后再查看答案。以下是一些重要的提醒信息：

- ✧ 我需要在脚本中通过变量获取 `Rigidbody2D`，并在此基础上使用 `MovePosition` 来移动游戏对象。务必在 `FixedUpdate` 中进行此过程。
- ✧ 别忘记，我可以通过 `public` 公开一个变量，以便在编辑器中调整此变量（例如速度）。

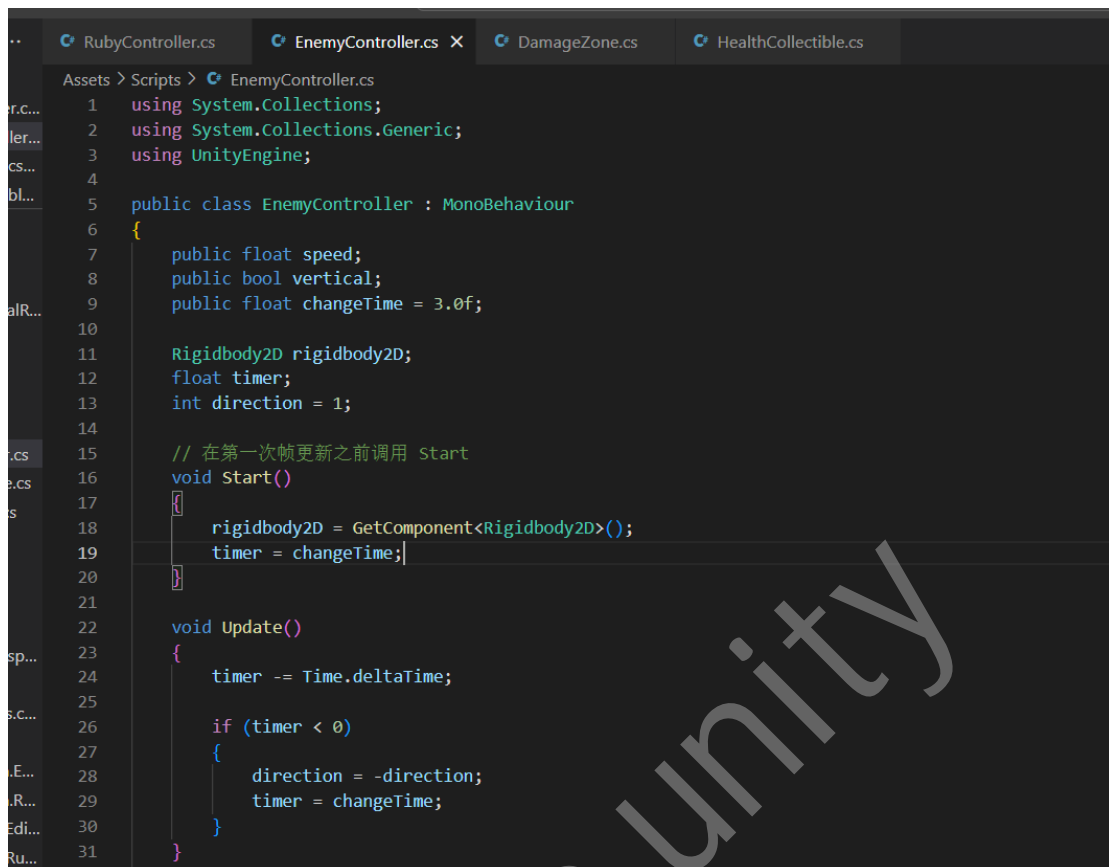
```
33 void FixedUpdate()
34 {
35     Vector2 position = rigidbody2D.position;
36
37     if (vertical)
38     {
39         position.y = position.y + Time.deltaTime * speed * direction;;
40     }
41     else
42     {
43         position.x = position.x + Time.deltaTime * speed * direction;;
44     }
45
46     rigidbody2D.MovePosition(position);
47 }
48
```

更改脚本，以便我可以在编辑器内以水平或垂直方向来回移动敌人。

- ✧ 对于方向，让我们使用名为 `vertical` 的公共 `bool` 变量。我可以在 `Update` 中进行测试以查看 `vertical` 是否为 `true`。如果为 `true`，则在我的世界中将敌人沿着 `y` 轴（而不是 `x` 轴）移动。
- ✧ 对于来回移动，我需要计时器（就像我用于无敌状态的计时器一样）。
- ✧ 只要我的计时器不为零，我就可以使敌人向一个方向前进，然后反转方向并重置计时器，依此无限执行这一循环过程。

为此，我需要创建更多变量：

1. 一个公共浮点变量 `changeTime`，表示我反转敌人方向之前的时间。
2. 一个私有浮点计时器，用于保存计时器的当前值。
3. 一个 `int` 变量，表示敌人的当前方向，值为 `1` 或 `-1`。



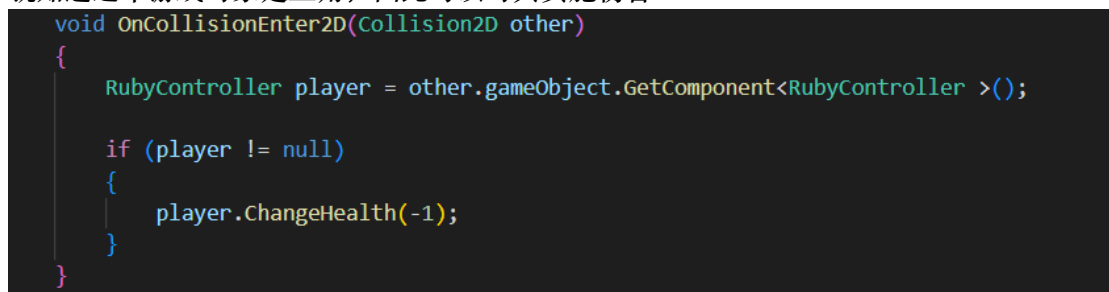
伤害

我得到了一个移动的敌人！现在，在 Ruby 与敌人碰撞时，可以让敌人伤害 Ruby，从而实现敌人与她的对抗。

与我的伤害区域不同，我不能使用触发器，因为我希望敌人碰撞体为“实心”并与物体实际碰撞。幸好，Unity 提供了第二组函数！

就像我用过的 OnTriggerEnter2D 一样，我也可以使用 OnCollisionEnter2D（这是刚体与某个对象碰撞时调用的函数）。在此示例中，我的敌人与世界或主角发生碰撞时，便会调用 OnCollisionEnter2D。就像我对伤害区域所做的那样，我也可以进行测试以查看敌人是否与我的主角发生了碰撞。

为此，我要检查与敌人碰撞的游戏对象是否具有 RubyController 脚本。如果有，那么我就知道这个游戏对象是主角，因此可以对其实施伤害：



五、 实验心得总结:

本次实验使用 Unity 引擎，通过 C#编写脚本我探索了以下功能：

- 对整个 Unity 编辑器的组织结构有了更多的了解
- 导入了第一个资源，即主角 Ruby 的精灵
- 编写了第一个脚本，用于在屏幕上移动 Ruby
- 通过轴处理了键盘输入
- 让角色的移动速度保持不变，无论每秒使用多少帧来渲染游戏都是如此
- 了解如何美化蓝色背景并开始装饰场景
- 使用瓦片在瓦片地图上进行绘制
- 从精灵创建瓦片
- 在单个图像文档中创建多个精灵
- 学习如何更改游戏对象的绘制顺序，以及如何从这些游戏对象创建预制件
- 探索了 Unity 中的物理系统的基础知识
- 添加了 Rigidbody 组件 来让物理系统处理对象
- 添加了碰撞体以使对象碰撞在一起
- 了解触发器的工作原理，以及如何检测刚体进入触发器以触发各种操作
- 用 C# 编写了自己的函数
- 使用私有和公共访问级别
- 编写了 if 语句，仅在测试结果为 true 时才执行某个操作
- 给角色所在的世界设置了一些挑战。学会如何使用伤害区域和移动的敌人来伤害角色。

对于这些内容的学习都使我对 Unity 开发有了更多了解和兴趣，今后我会继续探索相关的内容！