

- 根节点选择Area2D

类：  **Area2D**

继承：  CollisionObject2D <  Node2D <  CanvasItem <  Node <  Object

2D 空间中的一个区域，能够检测到其他 CollisionObject2D 的进入或退出。

描述



Area2D 是 2D 空间中的一个区域，由一个或多个 CollisionShape2D 或 CollisionPolygon2D 子节点定义，能够检测到其他 CollisionObject2D 进入或退出该区域，同时也会记录哪些碰撞对象尚未退出（即哪些对象与其存在重叠）。

这个节点也可以在局部修改或覆盖物理参数（重力、阻尼），将音频引导至自定义音频总线。

注意：使用 PhysicsServer2D 创建的区域和物体可能无法按预期与 Area2D 交互，并且可能无法正确发出信号或跟踪对象。

- 然后规定一个范围
 - 在Area2D下添加一个子节点CollisionShape2D（collision-碰撞）

类：  **CollisionShape2D**

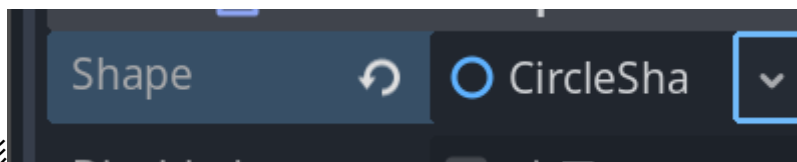
继承：  Node2D <  CanvasItem <  Node <  Object

向 CollisionObject2D 父级提供 Shape2D 的节点。

描述

向 CollisionObject2D 父级提供 Shape2D 并允许对其进行编辑的节点。这可以为 Area2D 提供检测形状或将 PhysicsBody2D 转变为实体对象。




•



- shape选择圆形

- 为小球添加图片
 - 在Area2D下添加sprite2D节点

类：  **Sprite2D**

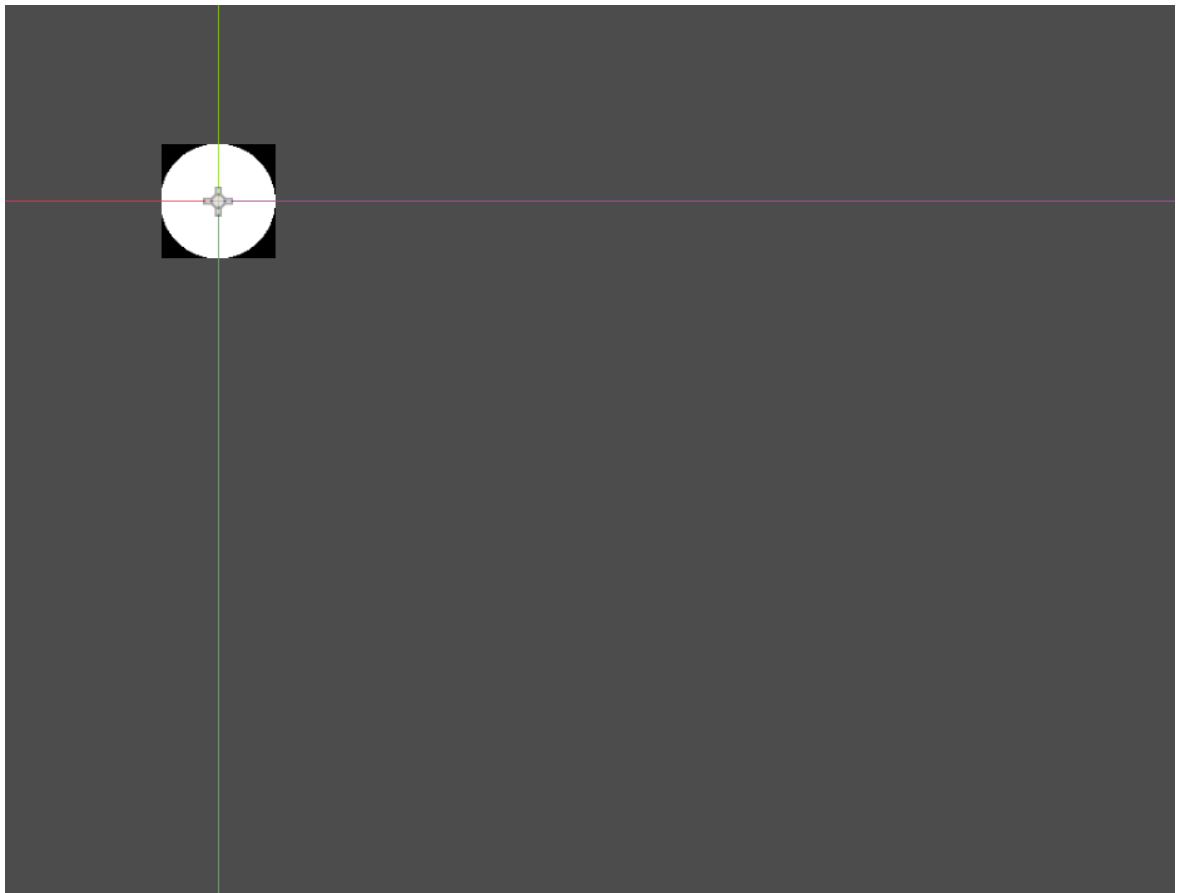
继承：  Node2D <  CanvasItem <  Node <  Object

通用精灵节点。

描述

显示 2D 纹理的节点。显示的纹理可以是较大图集纹理中的某个区域，也可以是精灵表动画中的某一帧。

- 然后将小球图片拽到texture上

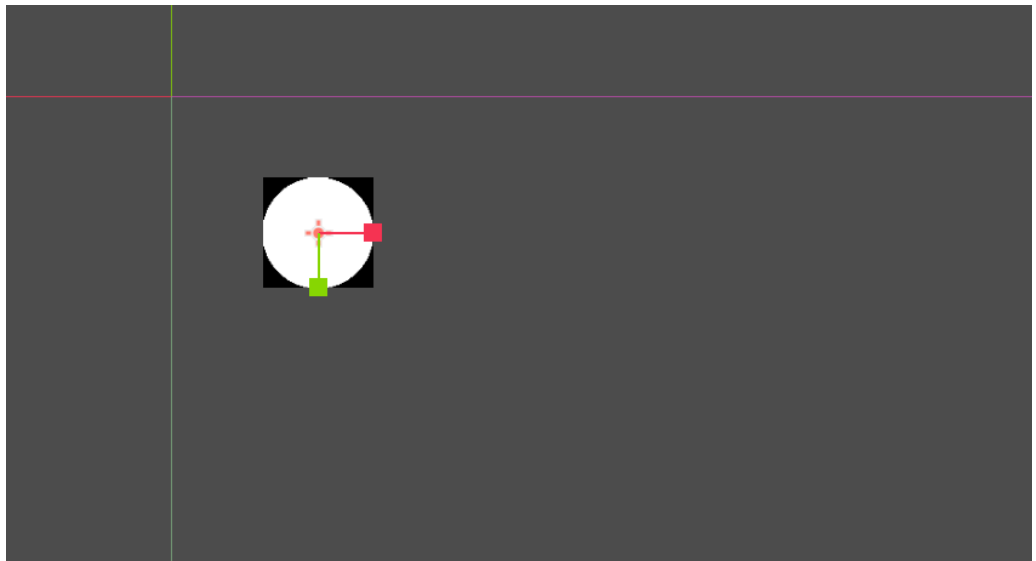


- 调整一下大小和位置，使他们重合
- 然后在文件区新建小球的文件夹，并将当前场景保存至小球文件夹
- 写根节点的脚本
 - 创建根节点（Area2D）的脚本
 - 先在_process函数下写出小球的运动

```
func _process(delta: float) -> void:  
    position = position + Vector2(1,0)
```

- 这里是二维向量的加法，水平方向一直向→运动
- 调试一下
 - 由于屏幕的左上角为坐标原点所以我们调整一下位置





- 然后f6运行, ok没问题
- 但我们对小球的运动做改变, 那么用全局变量定义一个增量

```

1  extends Area2D
2
3  var vec:Vector2=Vector2(1,0)
4  # Called when the node enters the scene tree for the first time.
5  func _ready() -> void:
6      pass # Replace with function body.
7
8
9  # Called every frame. 'delta' is the elapsed time since the previous frame.
10 func _process(delta: float) -> void:
11     position = position+vec
12

```

- 和上面的代码效果一样
- 然后我们需要小球能在飞出屏幕后重置位置
 - 需要定义重置位置的函数, 还要记录小球的位置

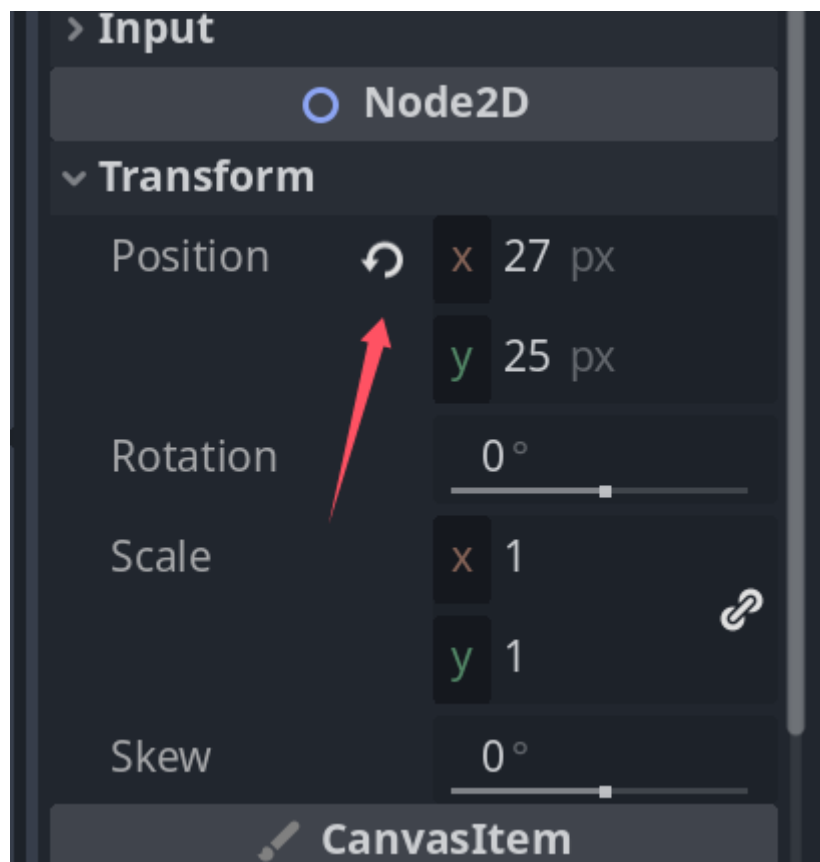
```

var vec:Vector2=Vector2(1,0)
var init_position # 记录小球初始位置
# Called when the node enters the scene tree for the first time.
▼ func _ready() -> void:
    >| init_position = position
    >|
    >| pass # Replace with function body.

# Called every frame. 'delta' is the elapsed time since the previous frame
▼ func _process(delta: float) -> void:
    >| # 实时判断小球是否出界并调整位置
    ▼ >| if position.x>500:
        >| >| rset()
        >| position = position+vec
        >|
    ▼ func rset():
        >| # 用于重置小球位置
        >| position=init_position

```

- ready函数是场景内所有节点加载完毕后立即触发的函数，那么可以在这里写一下初始信息的记录的代码
- process函数有一种循环的特性，所以可以写一下判断与循环体代码
- 全局变量挺重要的，要注意
- 调试一下，ok没问题



小球的初始位置

此处可以用来重置

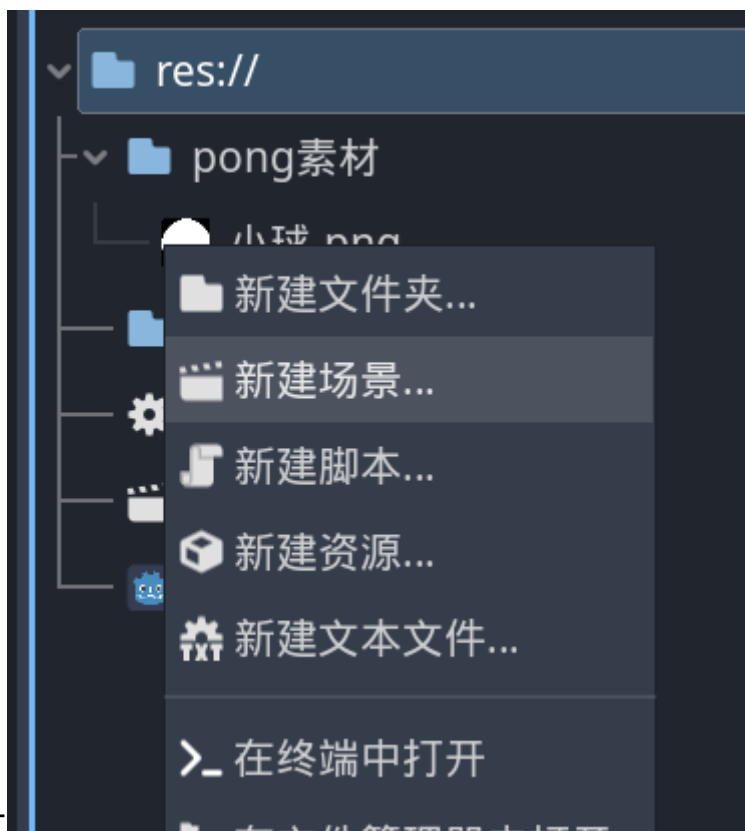
- 小球场景至此做完

地图场景

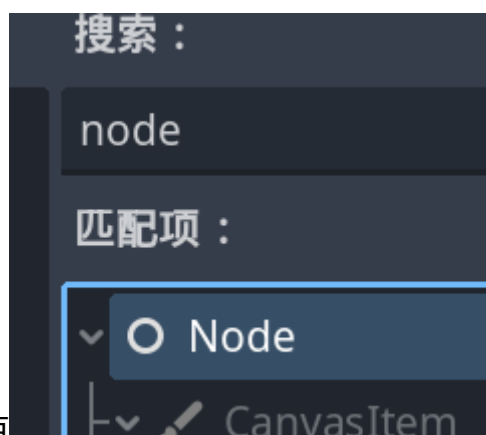
- 新建场景：




- 方法一







- 方法二



- 选择node作为根节点
- 添加ColorRect子节点作为地图的黑色背景版

类：  **ColorRect**

继承：  Control <  CanvasItem <  Node <  Object

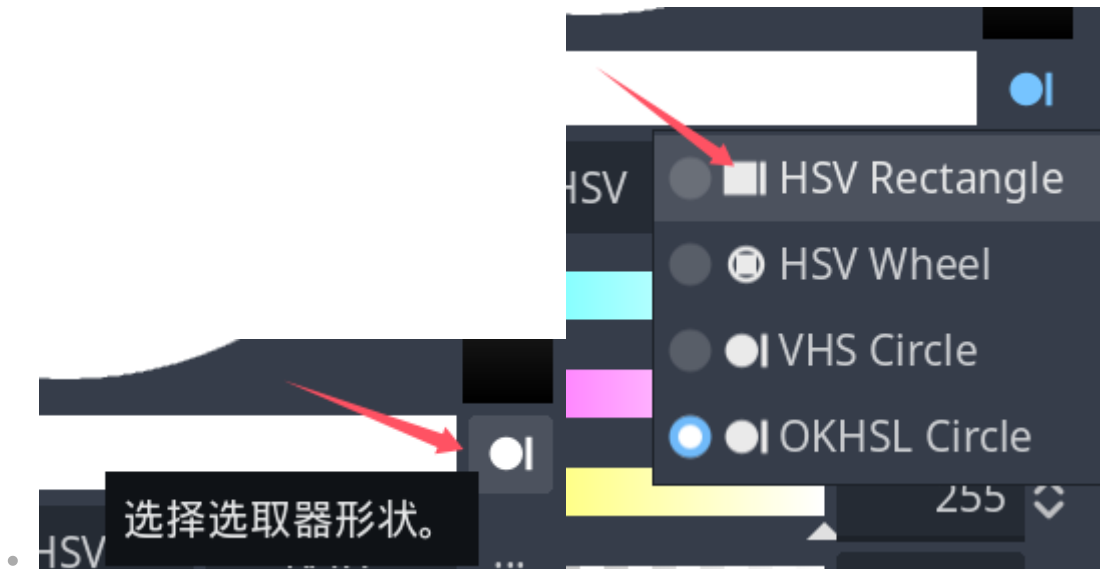
显示单色矩形的控件。

描述

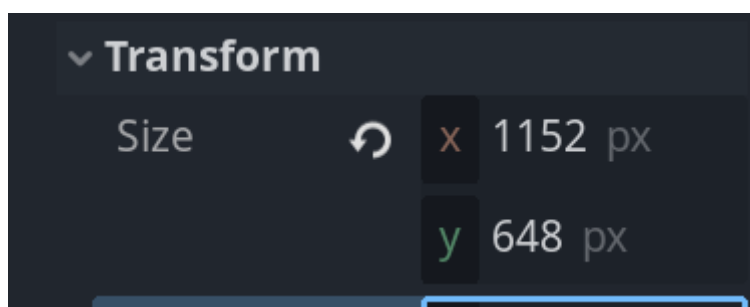
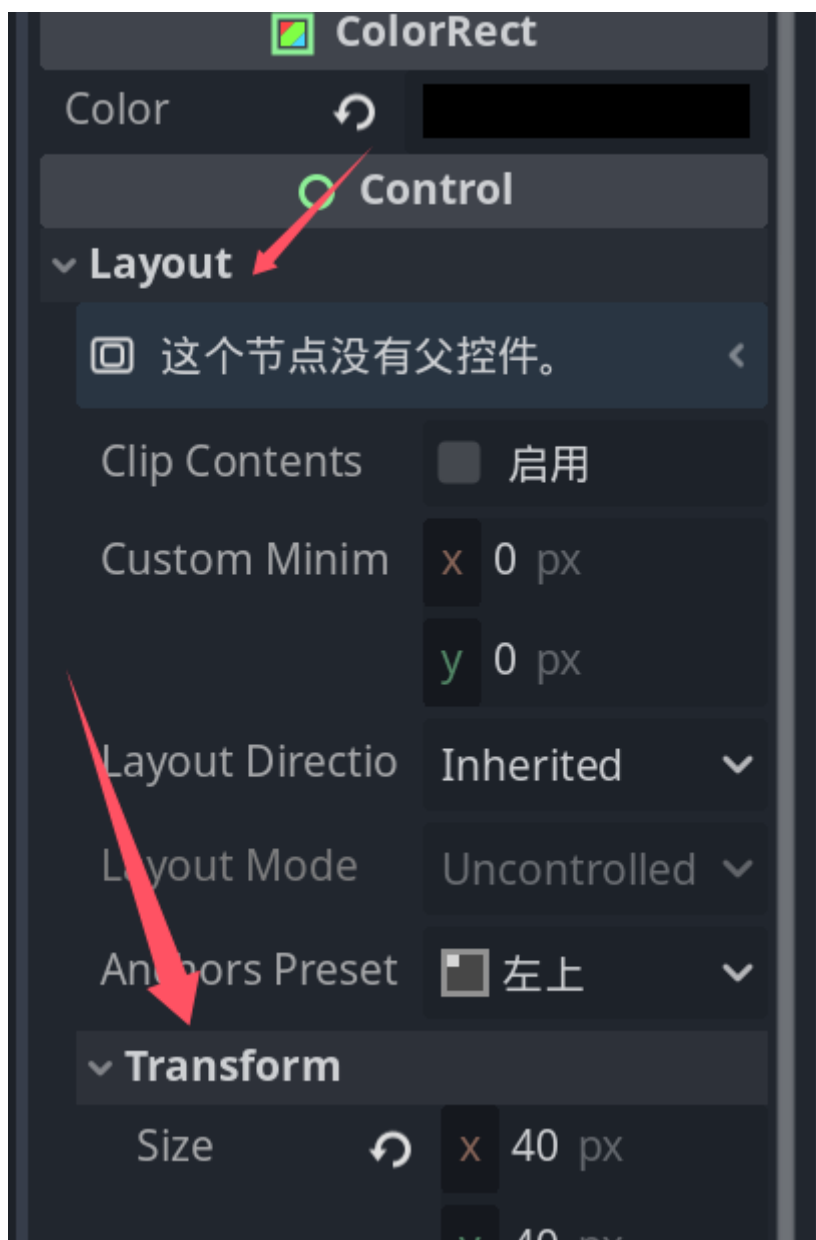
显示一个用纯色 color 填充的矩形。如果你需要单独显示边框，请考虑改用 Panel。

-

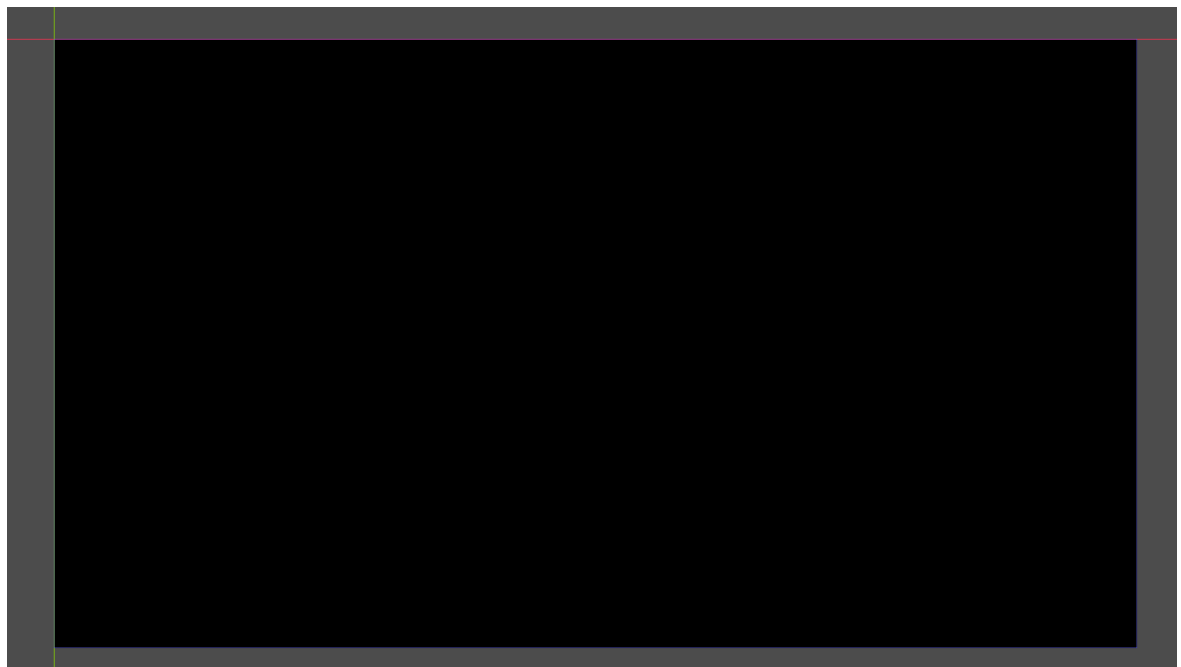
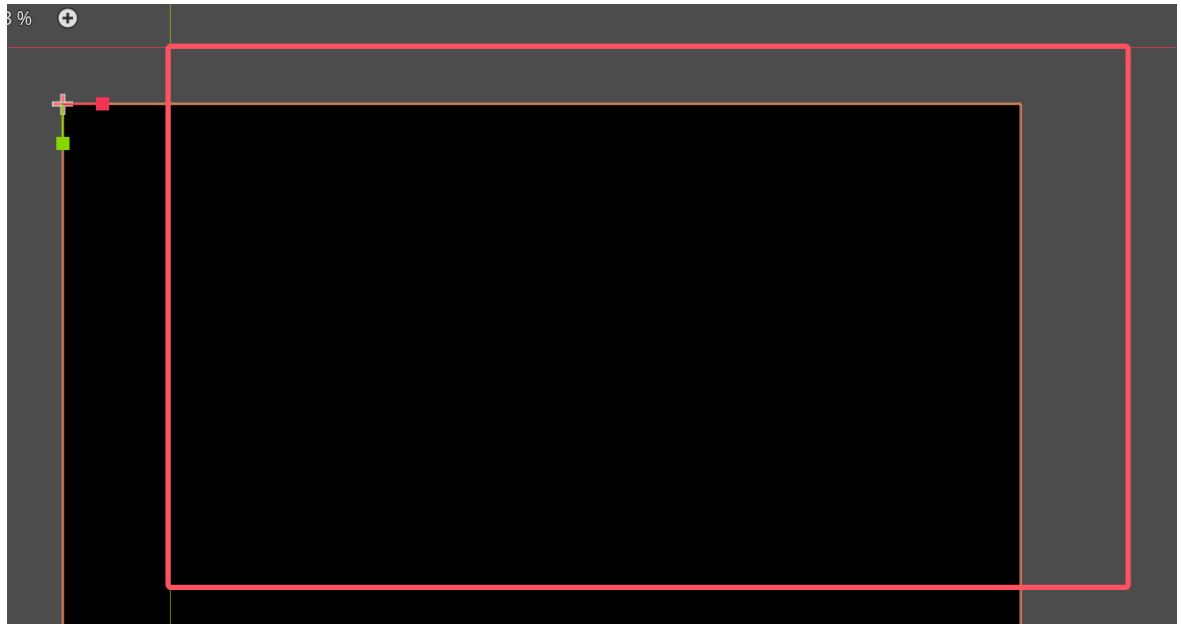
-  这里可以调颜色
- 选择HSV rectangle



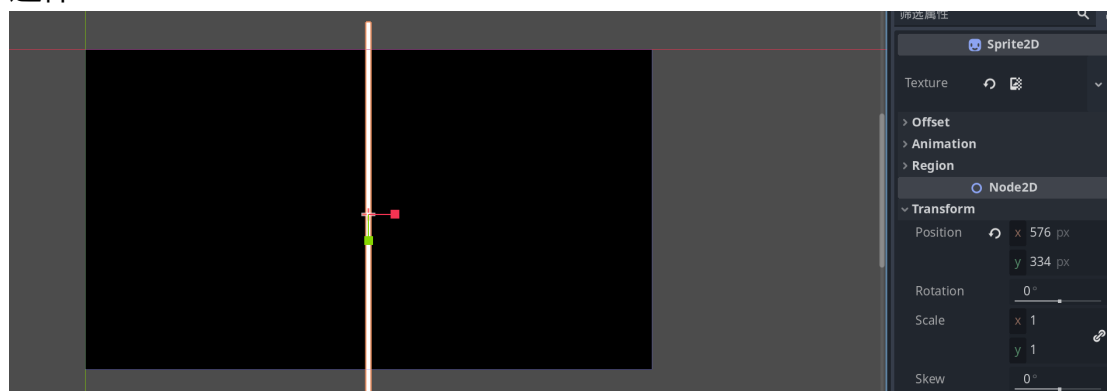
- 然后调成黑色就行
- 在layout找到transform改变长宽为1152: 648



- 这里我移开背景板，用红色标出的屏幕的大小

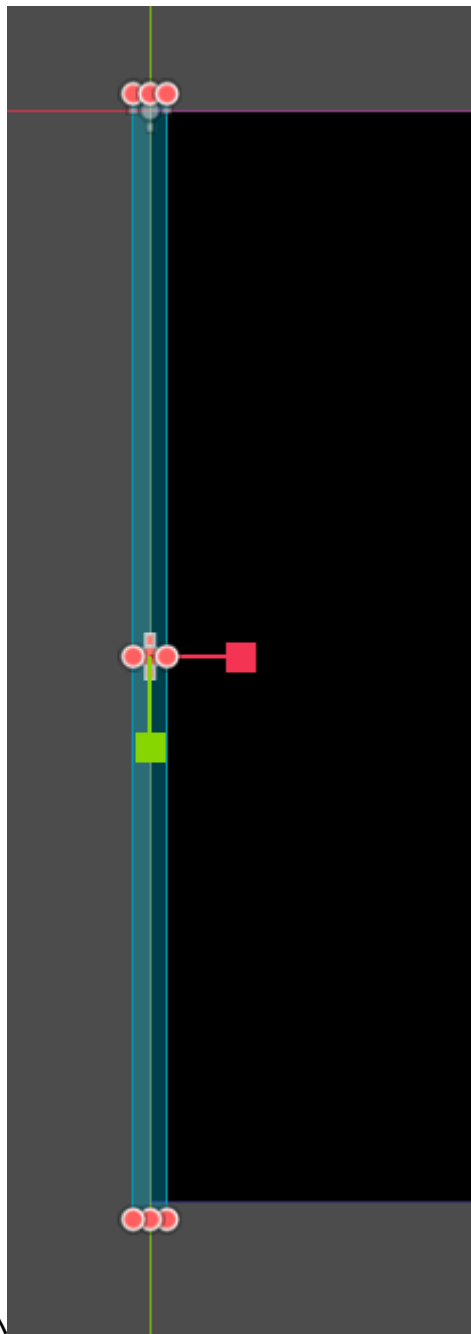
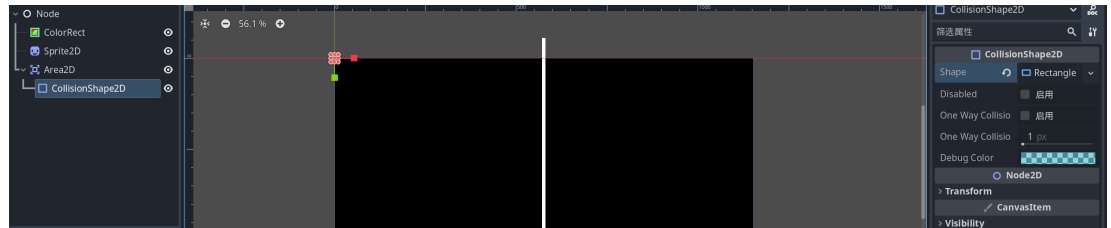


- 确认已经覆盖了整个屏幕
- 然后添加sprite2D节点显示竖直线
 - 这样



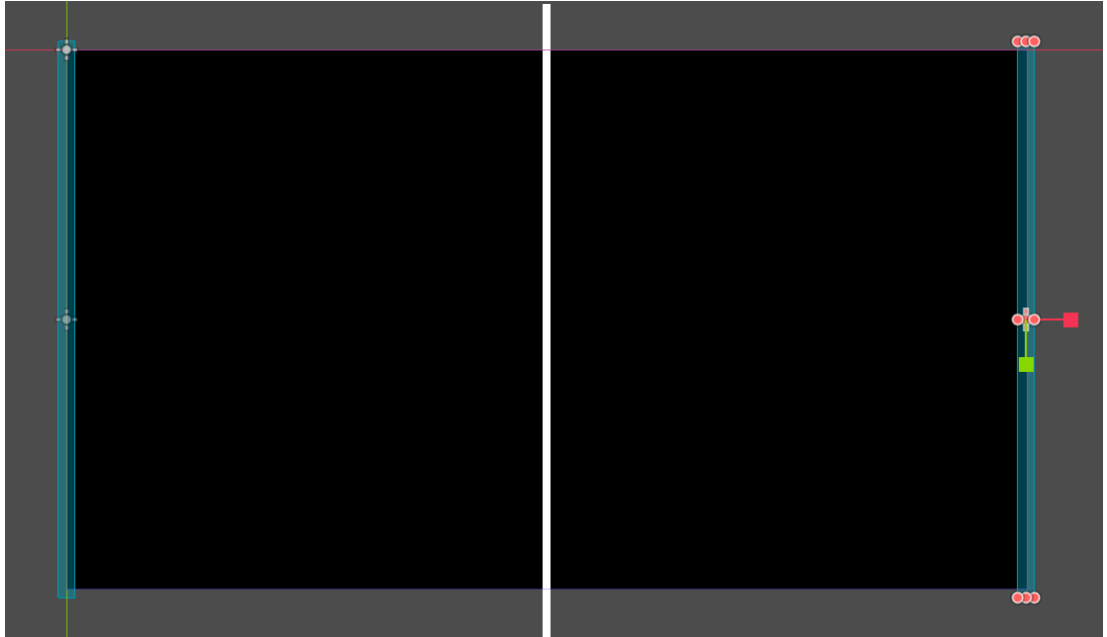


- 视频里是用缩放功能把线拉长了，如果素材长度不够的画就可以这么做，但我这里不用这么做，因为够长了
- 然后制作墙壁
 - 添加Area2D节点，然后再在这个节点下添加collisionshape2d节点，创建一个矩形



- 调整大小

- 然后ctrl+d复制这个Area2d节点拖到右边



- 墙壁就ok了
- 但是要需要有墙壁的“反弹”功能
- 那么编写墙壁的脚本
 - 这里需要高频地判断小球是否到达墙壁，并执行相应的子函数
 - 所以需要在process和physics process函数里写代码
 - 前者和后者由类似但本质不同的功能

```
func _process(delta: float) -> void:
    for i in get_overlapping_areas():
        pass
```

- 首先写出
 - 其中get overlapping areas 是

```
• Array[Area2D] get_overlapping_areas() const
```

返回相交的 **Area2D** 的列表。重叠区域的 `CollisionObject2D.collision_layer` 必须是这个区域 `CollisionObject2D.collision_mask` 的一部分，这样才能被检测到。

出于性能的考虑（所有碰撞都是一起处理的），这个列表会在物理迭代时进行一次修改，而不是在物体被移动后立即修改。可考虑改用信号。

返回的是相交的Area2d节点，那么我们先去小球那里写一下相应的代码

- 而且，这个for循环要做的事判断是否为小球的Area2d节点。
- 来到小球的脚本，写一段

```
func _ready() -> void:
    self.add_to_group('Ball')
    init_position = position
```

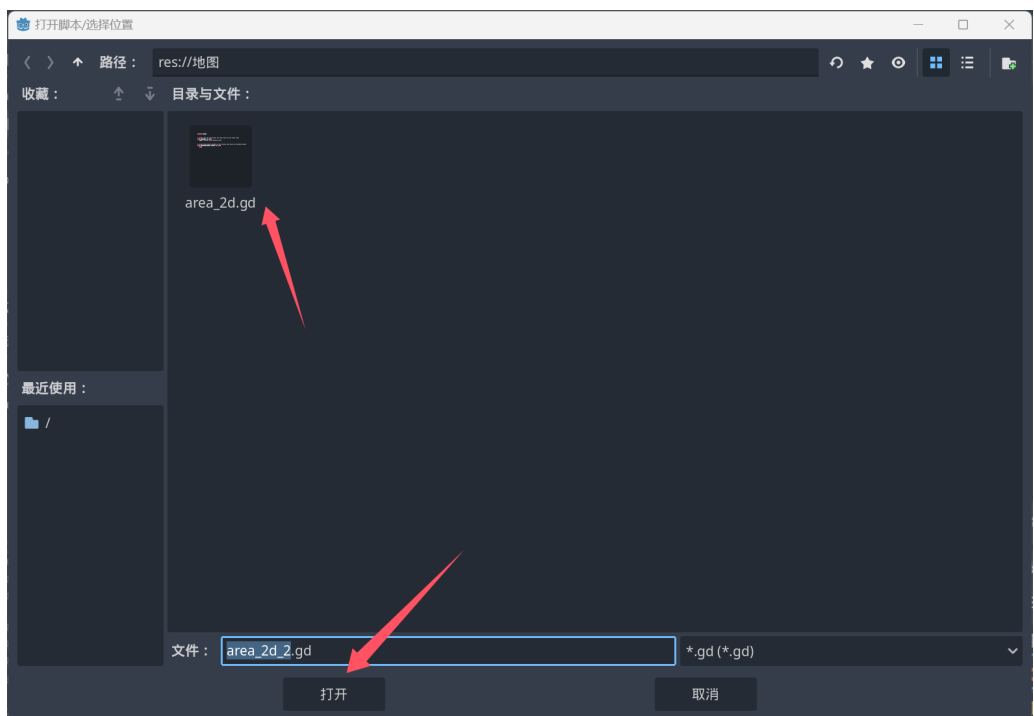
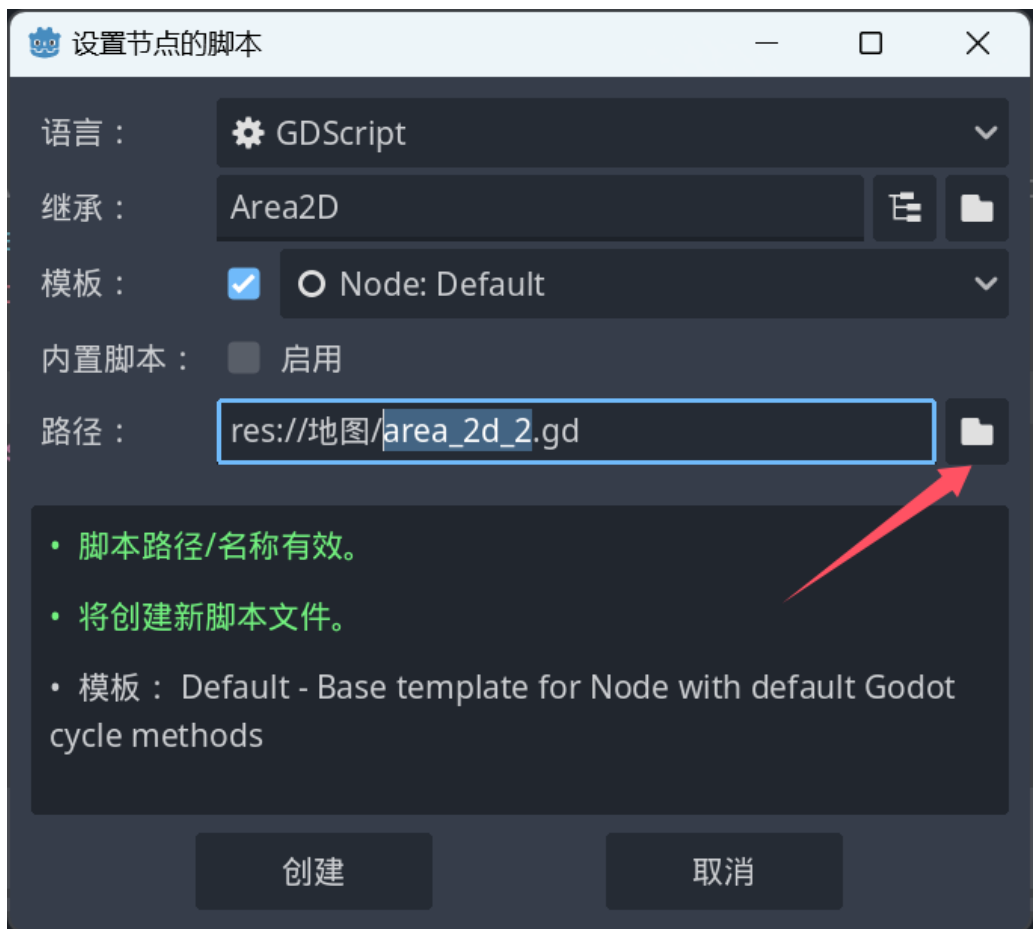
，这里是把小球自己加入一个名为Ball的group（如果这个group会先创建再加入）

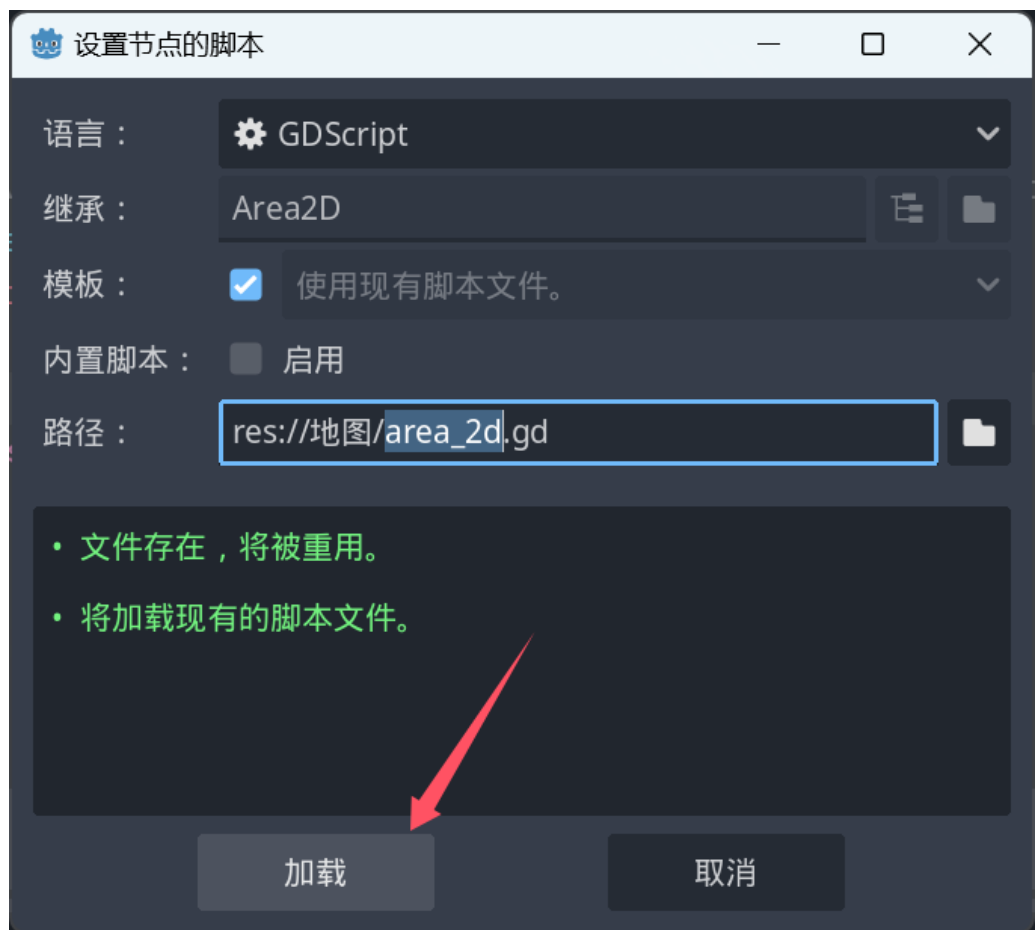
- 回到墙壁的脚本，写下

```
for i in get_overlapping_areas():  
>| if i.is_in_group("Ball"):  
>| >| i.reset()  
>| pass
```

这就是判断是否碰到墙壁，如果碰到就会重置位置的代码了

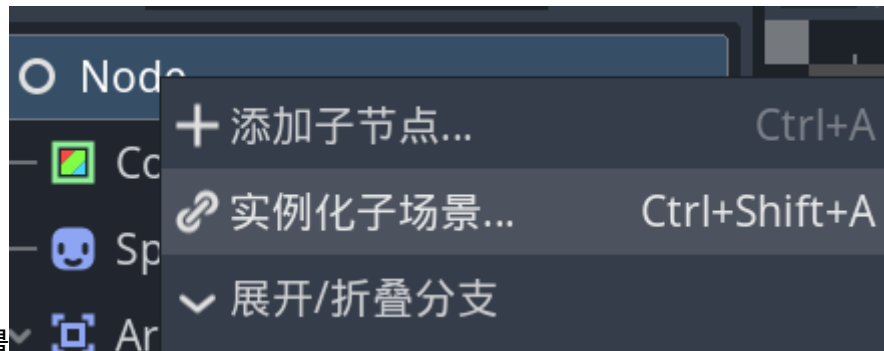
- 这里可以通过加载脚本的方式复制脚本到另一个相同作用的节点，也就是墙壁





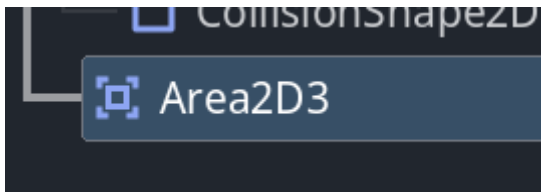
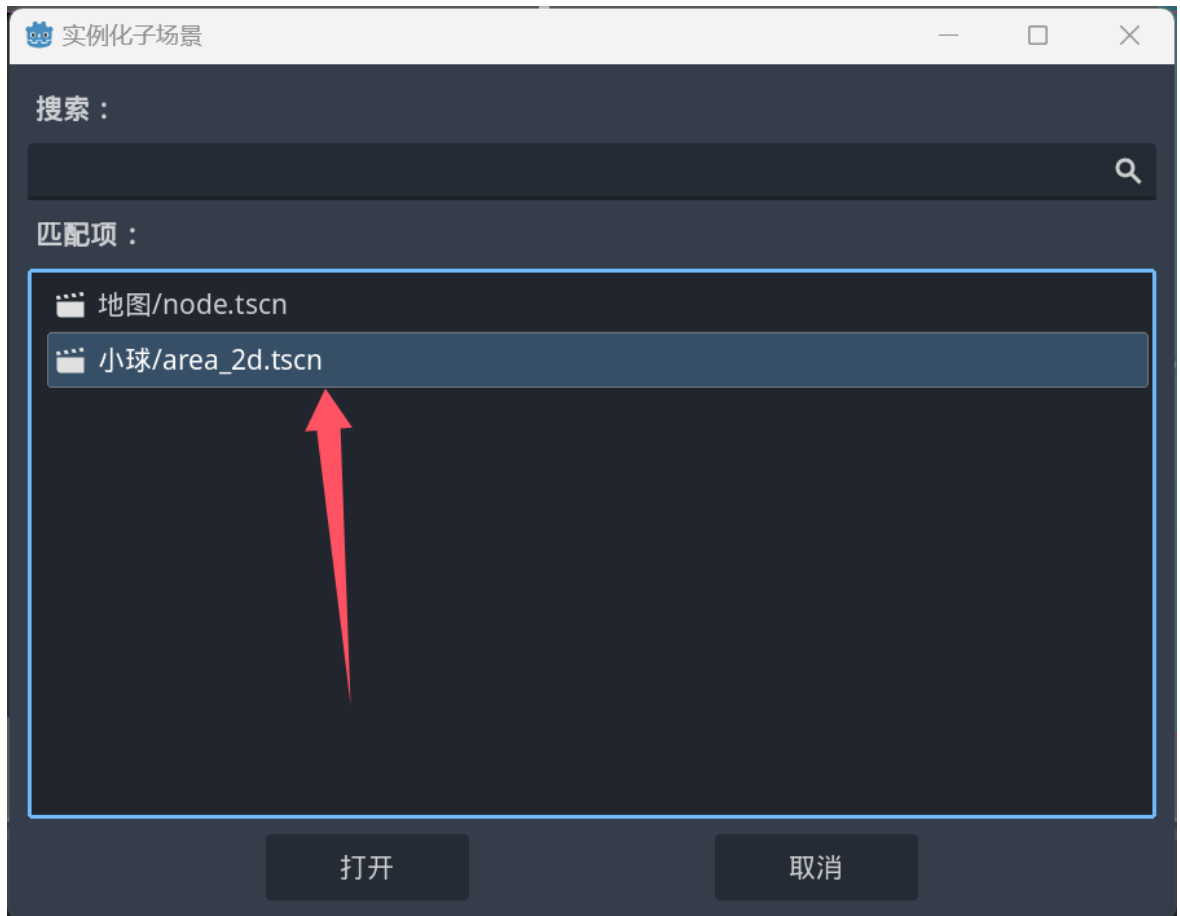
初步调试

- 将小球场景加入到地图场景中

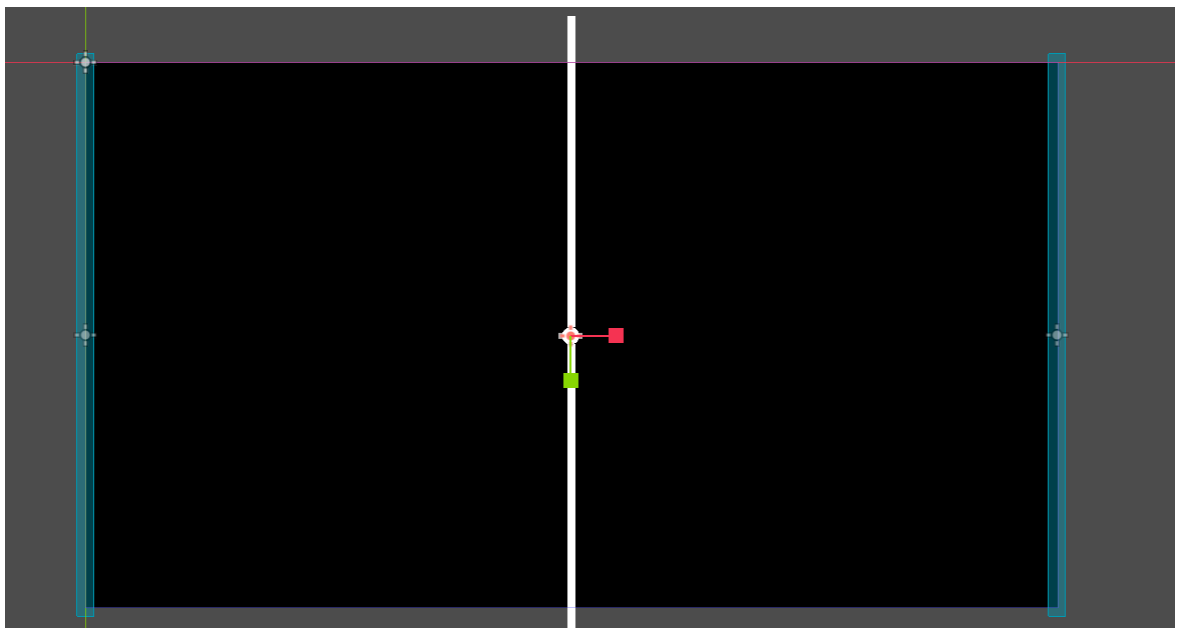


- 先实例化子场景

- 选择小球然后点打开



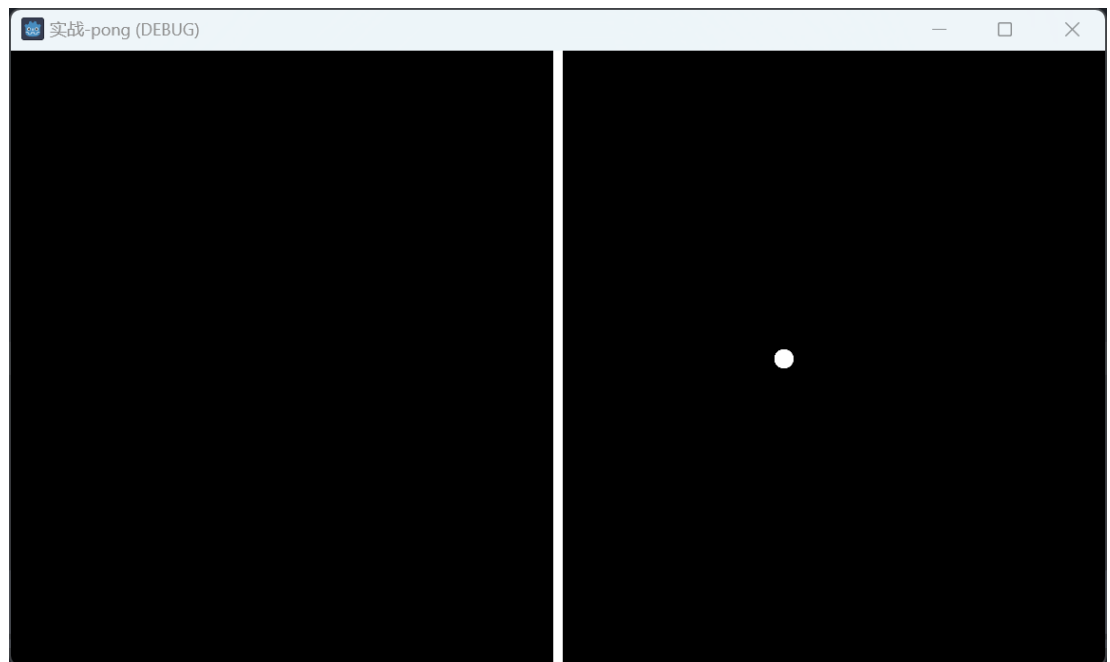
- 这个就是小球了，将它移至地图中央，记得



- 运行，不ok？小球不动！
 - 听取弹幕建议，注释了小球脚本里的这段代码

```
// called every frame: delta is the steps  
func _process(delta: float) -> void:  
>| # 实时判断小球是否出界并调整位置  
>| #if position.x>500:  
>| >| #self.reset()  
>| position = position+vec  
>|
```

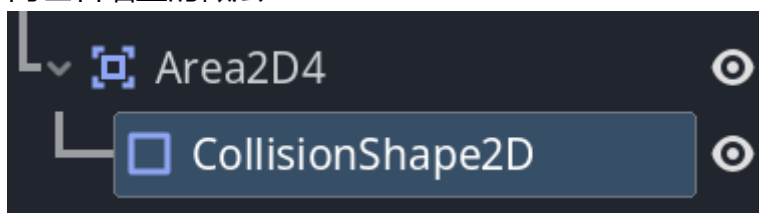
就可以动了

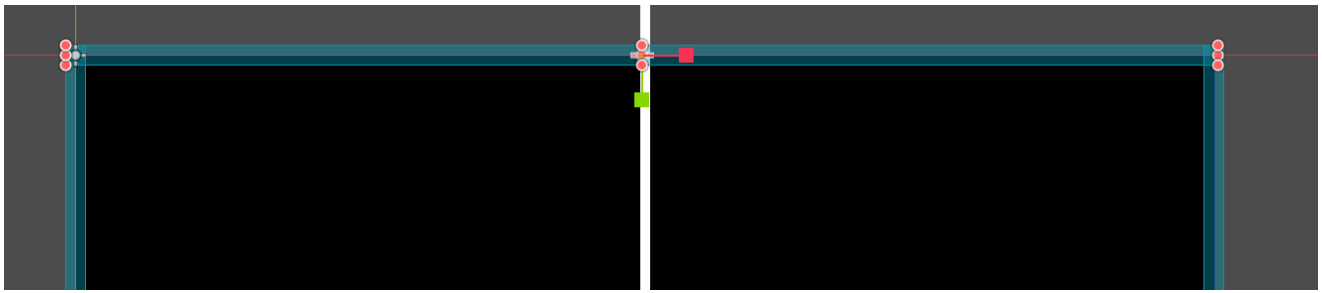


- 原因是小球的x位置已经大于500了，所以因为那串if代码导致一直重置在原点，造成了不动的现象
- 这下ok了

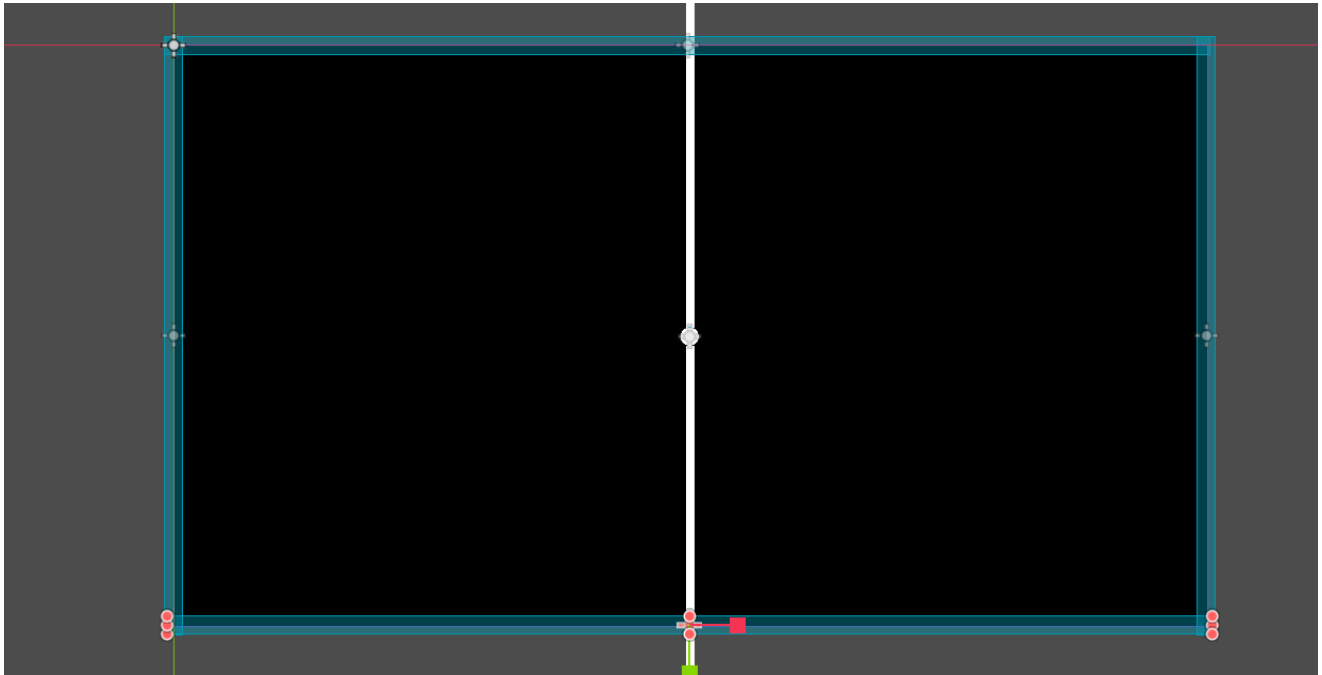
上下墙壁

同左右墙壁的做法





然后复制



- 然后编写脚本，在Area2D4节点编写（5也可以）
 - 这次要在

```
3  func _physics_process(delta: float) -> void:
4    >I  pass
```

编写代码

```
for i in get_overlapping_areas():
>I  if i.is_in_group("Ball"):
>I  >I  # 修改小球的运动方向
>I  >I  i.vec.y = 5 # 思路一：取正值
>I  >I  pass
```

-
- 也可以用全局变量然后作为参数修改


```

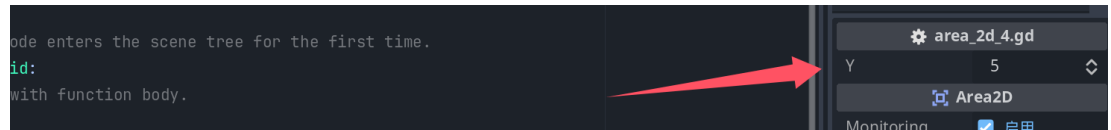
@export var y =5
# Called when the node enters the scene tree for the first time.
func _ready() -> void:
    pass # Replace with function body.

# Called every frame. 'delta' is the elapsed time since the previous frame.
func _process(delta: float) -> void:
    pass

func _physics_process(delta: float) -> void:
    for i in get_overlapping_areas():
        if i.is_in_group("Ball"):
            # 修改小球的运动方向
            i.vec.y = 5 # 思路一：取正值
            i.vec.y = y # 思路二：全局变量，也一样
    pass

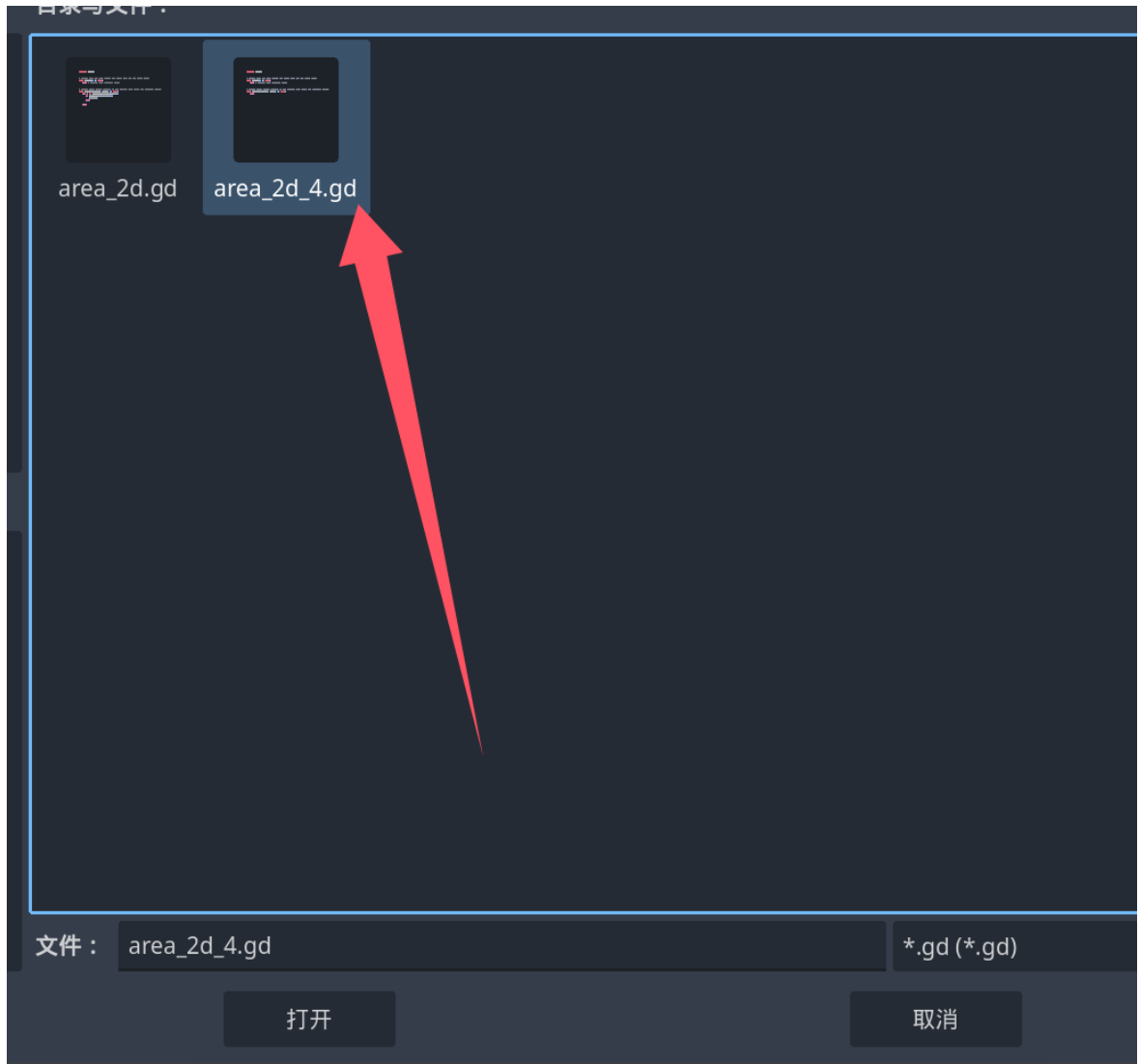
```

这里加上了@export, 会产生一个

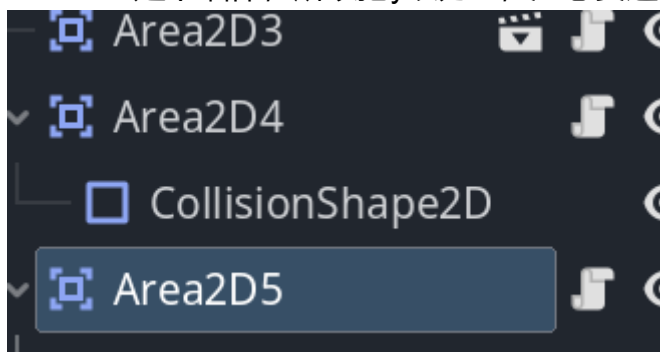


然后我们就能在右边更改这个y值了

- 然后在下边的墙壁加载脚本



- Area2D5是下墙体，所以把y改为-5，注意要选中



再改

```
vec:Vector2=Vector2(3,4)
```

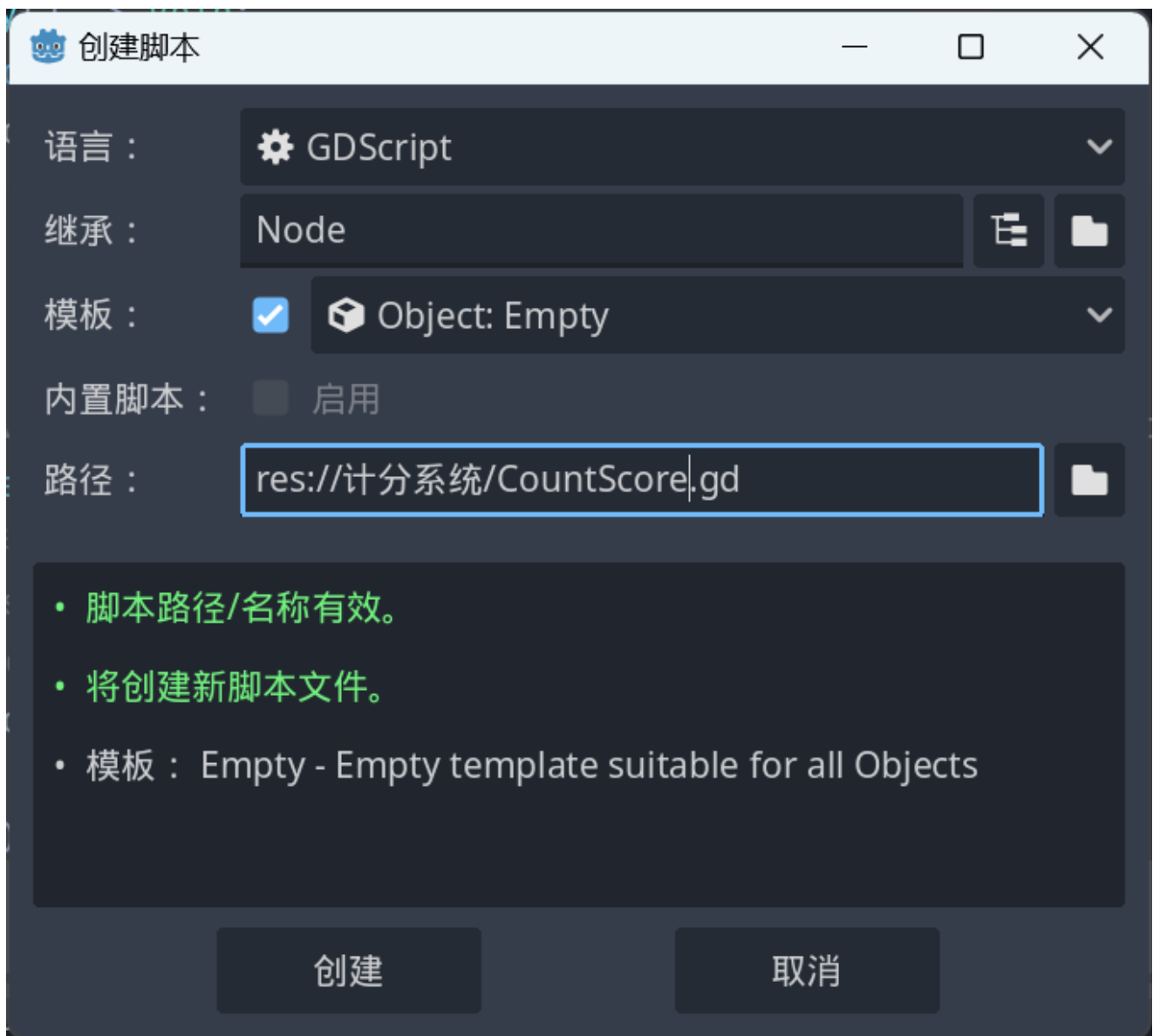
```
init position # 记录小球初始
```

- 最后把小球运动方向改成斜着的，
- 运行一下看看，ok了

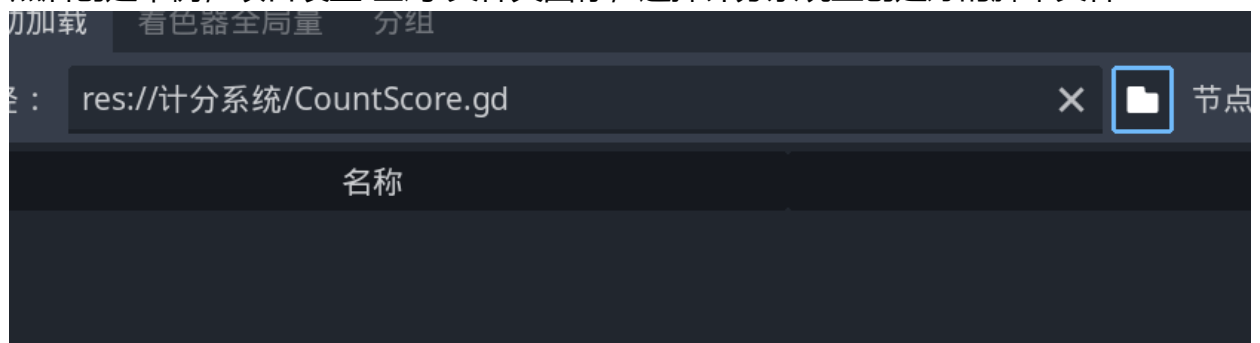
用单例制作记分系统

新建文件夹-计分系统

然后在文件夹里创建脚本



- 然后创建单例，项目设置-全局-文件夹图标，选择计分系统里创建好的脚本文件



- 然后 就可以了

- 在脚本里边定义玩家的分数变量

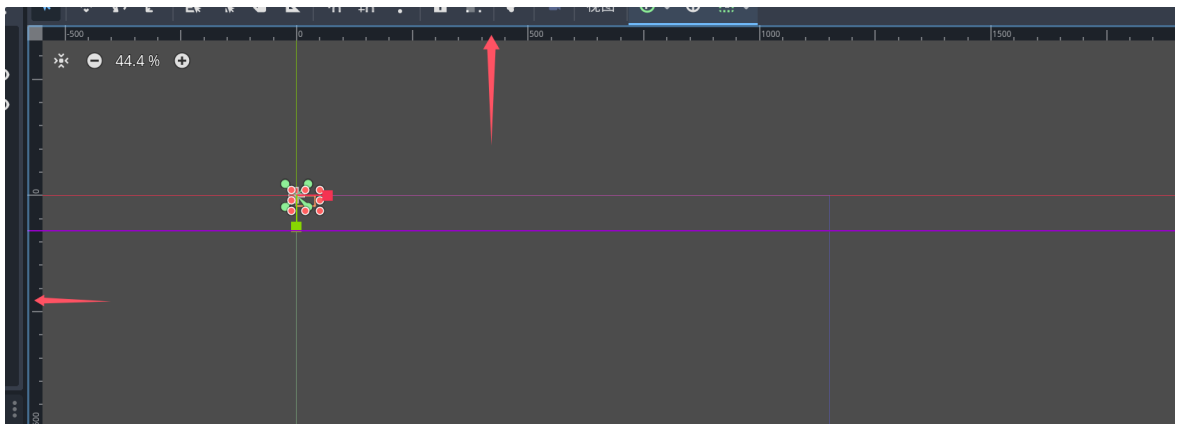
```
1 extends Node
2
3 var score1:int = 0
4 var score2:int = 0
5
```

- 然后在小球的process函数里写代码

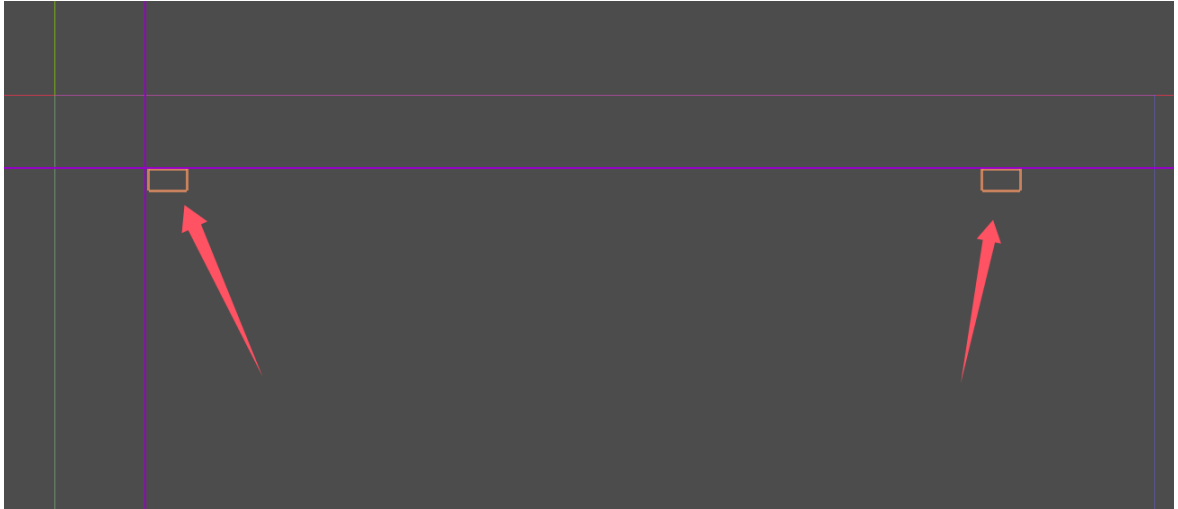
```
func rset():
    # 计分机制
    if vec.x>0:
        CountScore.score1=CountScore.score1+1
    else:
        CountScore.score2=CountScore.score2+1
```

表示小球向右飞出边界则玩家1加1分，反之向左飞出则玩家2加1分

- 然后创建一个显示得分的场景
 - 以node做根节点
 - 用label做子节点做文字显示
 - 这里可以通过拖拽这两个区域来创建参考线



- 我这里分别把文字放在这两边



- 新建文件夹保存这个文字场景
- 然后给两个label分别创建脚本，label用来显示玩家一分数，另一个玩家二

```
# Called every frame. 'delta' is the elapsed time since the previous frame.  
func _process(delta: float) -> void:  
    >I text=String.num_int64(CountScore.score1)  
    >I pass
```

```
# Called every frame. 'delta' is the elapsed time since the previous frame.  
func _process(delta: float) -> void:  
    >I text=String.num_int64(CountScore.score2)  
    >I pass
```

- 然后回到地图场景
- 把这个计分场景实例化到地图场景里
- 运行一下，欸，怎么一次加4分，原来是之前的墙壁检测函数写的有问题
 - 原因：范围检测是一个物理引擎的工作，而物理引擎刷新一次，检测的信息也会刷新一次，反之，物理引擎不刷新，检测的信息不刷新，而一般情况下屏幕刷新次数高于物理引擎，就会导致process多次被触发时，get-overlapping-areas会接受同一批的物理引擎处理的数据，导致rset被重复触发

- 解决办法就是把这段代码转移到physics-process函数里

```

10  func _process(delta: float) -> void:
11      >|
12      >| pass
13  func _physics_process(delta: float) -> void:
14      >| for i in get_overlapping_areas():
15          >| if i.is_in_group("Ball"):
16              >| >| i.rset()
17      >| pass
18

```

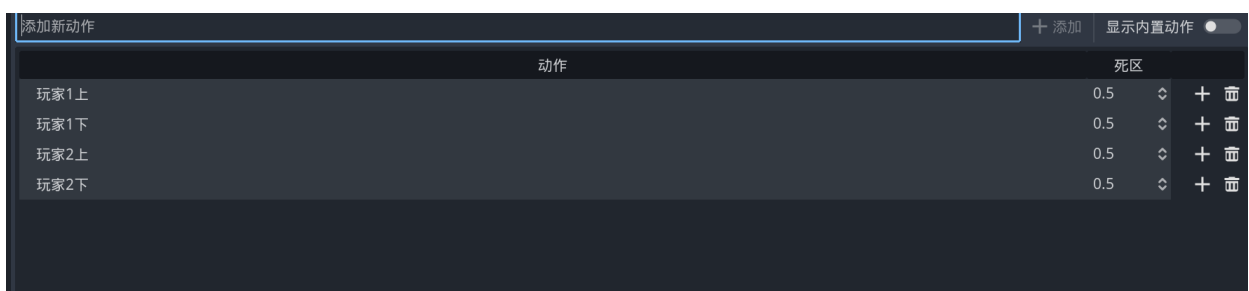
- 再运行一次，ok了

玩家的交互

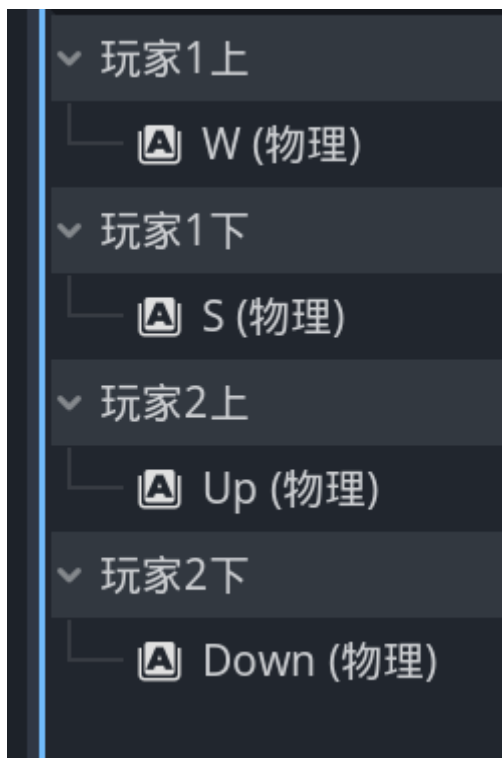
- 首先新建玩家文件夹



- 建立键位映射，项目-项目设置-键位映射，由于每个玩家仅需要进行上下移动操作，故一共需要四个键位的映射



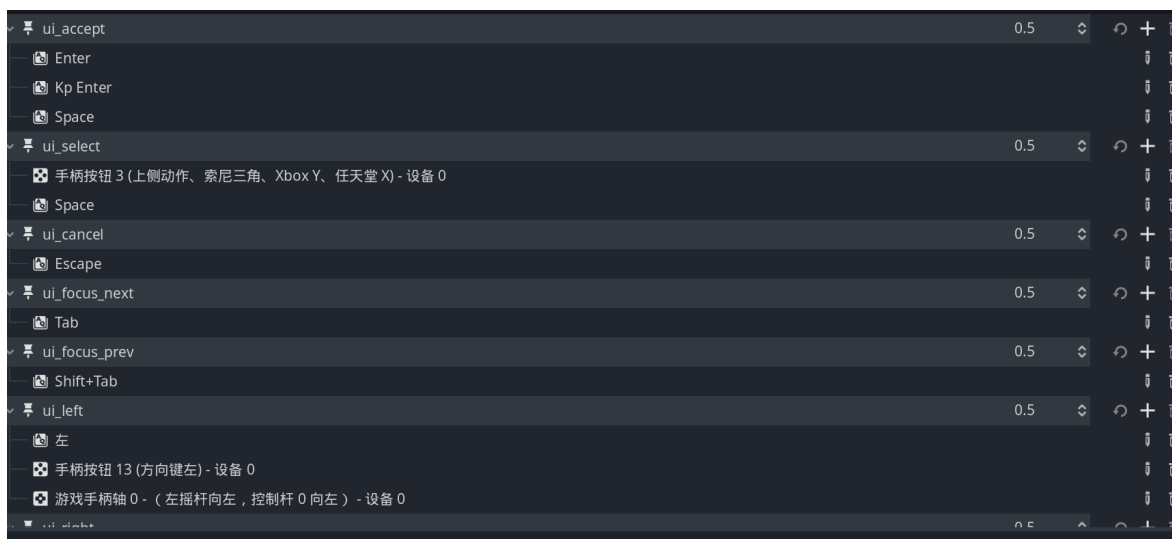
- 然后按右边的加号匹配相应的键位



- 挺快的，按一下键盘的键就出现对应的键了

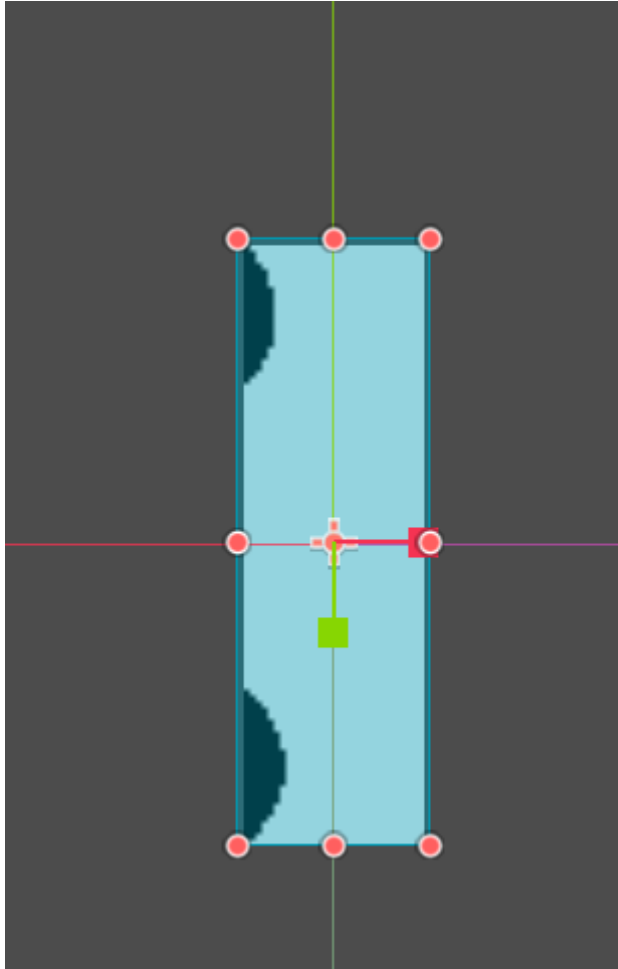


- 打开后可以看到被隐藏的内置映射

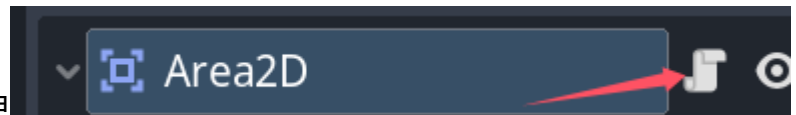


- 然后是创建玩家场景
 - 玩家1场景：
 - 依然使用Area2D作为根节点
 - 然后添加sprite2d和collisionshape2D节点
 - 给sprite2D添加球拍图片

- 然后给collision节点设置矩形区域并调整形状符合球拍



- 给area节点写脚本
 - 创建脚本的时候注意命名别重复，不然你不小心关闭后就不知道哪个是刚创建的脚本了

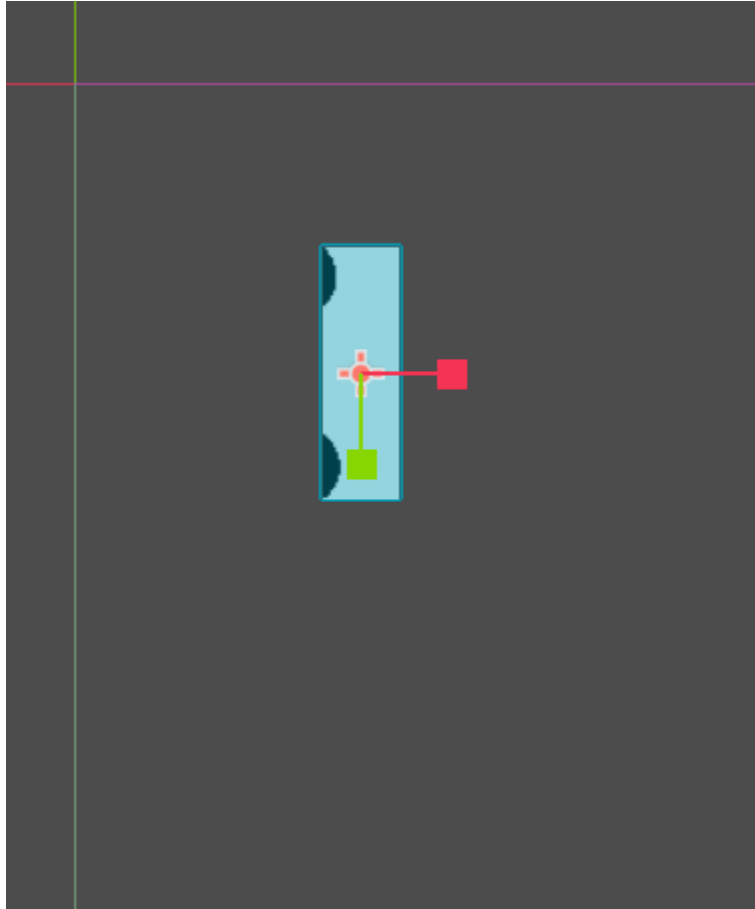


- 可以通过点这里
调出脚本窗口
- 然后

```
extends Area2D

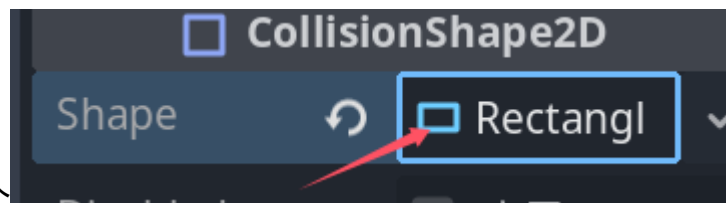
func _physics_process(delta: float) -> void:
    # 反弹小球的代码
    for i in get_overlapping_areas():
        if i.is_in_group("Ball"):
            i.vec.x = 5
    # 控制移动的代码
    var y1 = Input.get_action_strength('玩家1上')*5 # 这里是为了加快移动速度
    var y2 = Input.get_action_strength('玩家1下')*5
    # 位置改变的代码
    var y3 = position.y - y1 + y2 # 一种记录当前球拍的y位置的写法
    if y3 > 16:
        if y3 < 630:
            position.y = position.y - y1 + y2
```


- 保存然后调试一下，欸怎么不懂，原来是初始位置不太好，改一下位置



- ok了
- 改成这样也是可以的，我觉得

```
if y3>0:
>|   if y3<648:
>|   >|   position.y =position.y-y1+y2
```



- 可以通过点这个来看这个矩形区域长宽信息
- 然后来改上边这个if语句

```
if y3>48 && y3<600:
>|   position.y =position.y-y1+y2 # 这是移动小球的写法，想成向量的写法
>|   >|
```

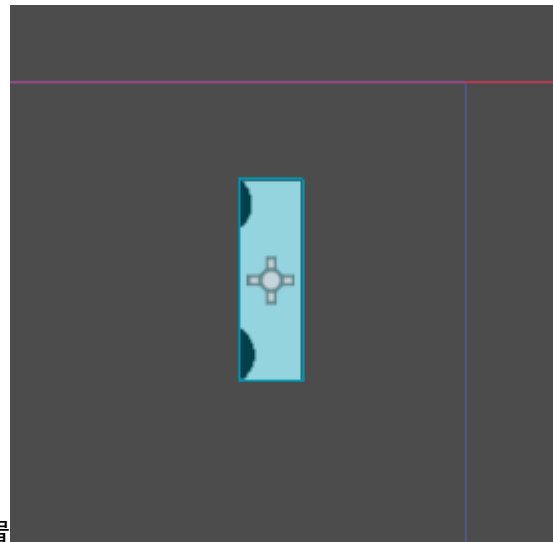
就差不多效果了

- 然后玩家2的节点和1一样，代码少部分不一样

```

func _physics_process(delta: float) -> void:
    >| # 反弹小球的代码
    >| for i in get_overlapping_areas():
    >| >| if i.is_in_group("Ball"):
    >| >| >| i.vec.x = -5
    >| # 控制移动的代码
    >| var y1 = Input.get_action_strength('玩家2上')*5 # 这里是
    >| var y2 = Input.get_action_strength('玩家2下')*5
    >| # 位置改变的代码
    >| var y3 = position.y - y1 + y2 # 一种记录当前球拍的y位置的写法
    >| if y3 > 48 && y3 < 600:
    >| >| position.y = position.y - y1 + y2 # 这是移动小球的写法，想成
    >| >| >|
    >|

```

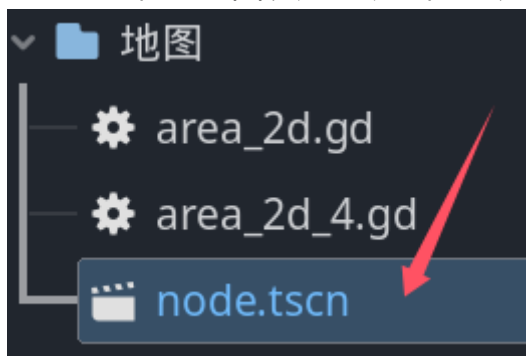


记得调图片，矩形大小，位置

- 调试看看，ok
- 最后把这两实例化到地图场景，调整位置后在调试一下就行了

完事大吉，但是导出

导出之前，把地图设为主场景，可以在文件区域单击这个地图节点文件



也可以点  然后-选择当前，就行了

导出：项目-导出-添加-选择windows-导出项目，选择导出的文件夹（最好是空的），然后命名游戏名称，然后保存