FICHA TÉCNICA

AUTOR

Equipe de Desenvolvimento Gillis Interactive

EDIÇÃO TÉCNICA

Coordenação de Desenvolvimento Gillis Interactive

REVISÃO & CONTROLE DE QUALIDADE

Equipe de Desenvolvimento Gillis Interactive

DESENVOLVEDOR DE GAMES COM GODOT 3.0
MÓDULO II

GODOT 3.0 / Gillis Interactive, 2021

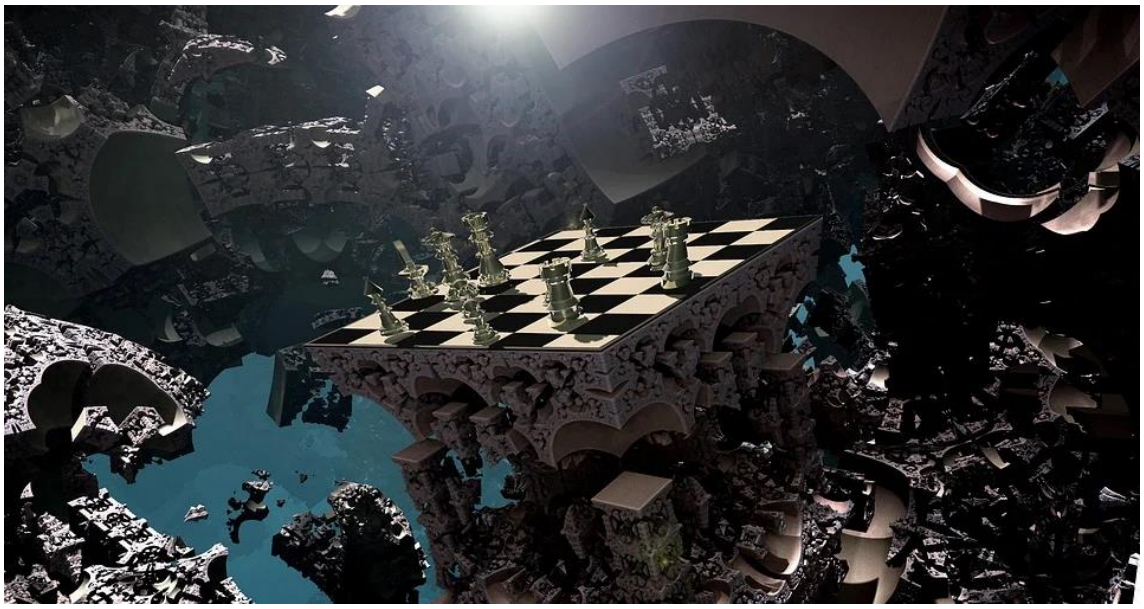
Edição: 1ª Idioma: português

Em conformidade com o Acordo Ortográfico da Língua Portuguesa.

GILLISINTERACTIVE
TECNOLOGIA & EDUCAÇÃO

Sumário

AULA 01



Configuração do projeto

Onde começar? Dependendo do jogo e de quão concreta é a sua ideia, a resposta pode ser muito diferente. No nosso caso, eu trapaceei um pouco ao fazer um protótipo do jogo e trabalhar algumas das ideias com antecedência. Ainda assim, divergiu um pouco da minha ideia inicial, assim como esta série - o tempo dirá.

Em um projeto maior, você pode começar com um documento de design, que pode ser tão simples quanto uma página de anotações ou tão complexo quanto um tratado de 500 páginas, apresentando todos os detalhes do mundo, enredo e mecânica do seu jogo. Não precisamos de nada tão elaborado aqui, então vamos apenas revisar o plano de jogo.

Plano de Jogo

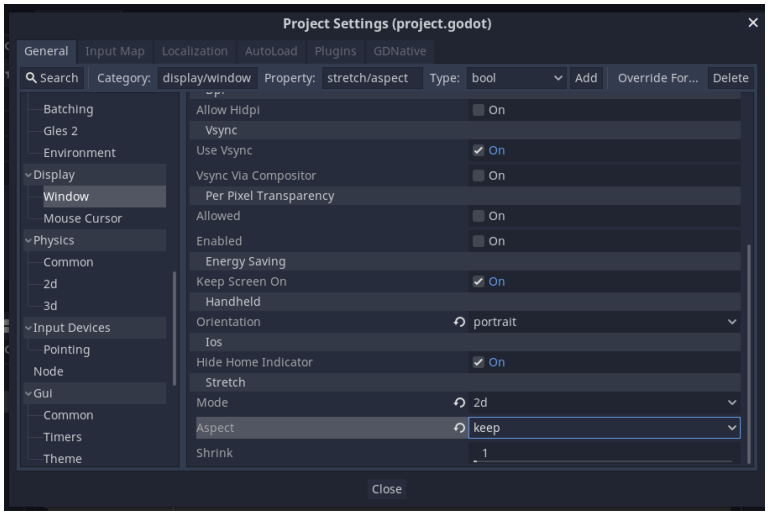
Neste jogo, o jogador controla um “personagem” que salta de círculo em círculo. O salto é iniciado por um clique ou toque, e se você não acertar outro círculo, você perde. A pontuação está relacionada a quanto tempo você sobrevive, e a dificuldade aumentará com o tempo com círculos que se movem, encolhem e / ou expiram.

A ideia são jogos rápidos e curtos com uma sensação de “topo de gama”. Tanto quanto possível, a arte permanecerá simples e limpa, com efeitos visuais e de áudio para adicionar apelo.

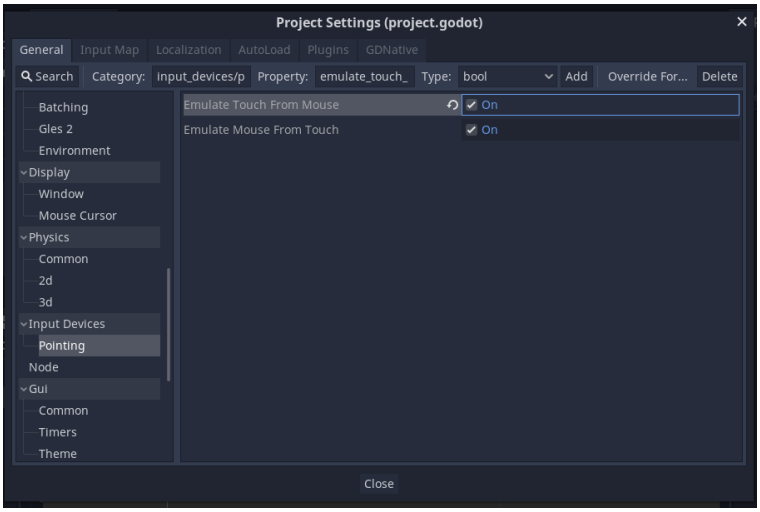
Começando

Vamos começar com as configurações do projeto. Precisamos definir o tamanho / comportamento de nossa tela. Queremos que este seja um jogo para celular, então ele precisará estar no modo retrato e ser capaz de se ajustar a tamanhos de telas variáveis, já que existem tantas resoluções de telefone disponíveis.

Abrimos as configurações do projeto e localizamos a seção *Exibição / janela* . Definimos o tamanho da tela para (480, 854), o *Handheld / Orientation* para “Portrait”, o *Stretch / Mode* para “2d” e o *Stretch / Aspect* para “Keep”.



Em seguida, em *Dispositivos de entrada / apontadores,* habilitamos “Emular toque do mouse”. Isso nos permitirá escrever o código usando apenas eventos de toque na tela, mas ainda jogar usando o mouse em plataformas de PC.



Organização do Projeto

Para manter as coisas organizadas, vamos fazer uma pasta para armazenar os objetos do jogo (objects) e uma para a IU (gui). Os recursos do jogo (imagens, áudio, etc.) irão para uma assetspasta.

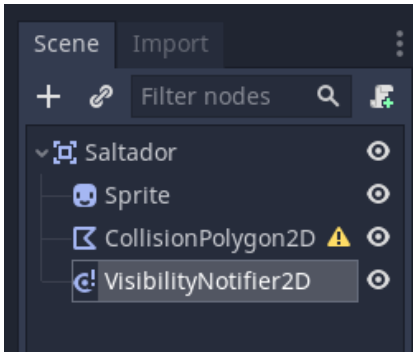
Assim que tivermos as pastas e os recursos configurados, estamos prontos para começar a codificar!

Objetos de jogo: temos dois objetos de jogo a fazer, que são o jogador (“saltador”) e o círculo.

Saltador

Para movimento e colisão, vamos usar a Area2D. Para ser justo, poderíamos usar KinematicBody2D aqui também e funcionária da mesma forma. No entanto, não precisamos realmente de colisão neste jogo, só precisamos saber quando o saltador entra em contato com um círculo. Vamos adicionar os seguintes nós:

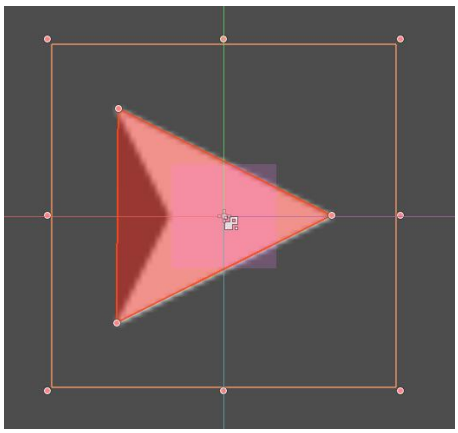
- Area2D ("Saltador"):
    - Sprite;
    - CollisionPolygon2D;
    - VisibilityNotifier2D.



Salvamos a cena (res://objects/) e arrastamos a imagem do círculo (res://assets/images/jumper.png) para a *Textura* do Sprite. Todas as imagens do jogo são totalmente brancas. Isso tornará mais fácil para nós colori-las dinamicamente mais tarde.

Desde que a arte é desenhada apontando para cima, definimos o Sprite's *Rotation* propriedade para 90.

Selecionamos o CollisionPolygon2D e adicionamos três pontos para cobrir a forma triangular do saltador.



ANOTAÇÕES

AULA 02

Codificação do movimento

Agora vamos adicionar um script ao corpo e começar a codificar seu comportamento:

Primeiro, os sinais e variáveis:

```
extends Area2D


var velocity = Vector2(100, 0)  # iniciando valor para teste


var jump_speed = 1000

var target = null  # se você estiver em um círculo
```

A seguir, detectaremos o toque na tela e, se estivermos em um círculo, chamaremos nosso método de salto:

```
func _unhandled_input(event):
    if target and event is InputEventScreenTouch and event.pressed:
        jump()
```

Saltar significa deixar um círculo e ir para a frente em velocidade de salto:

```
func jump():
    target = null
    velocity = transform.x * jump_speed
```

Nós detectaremos o acerto de um círculo com o sinal area_entered, então o conectamos. Se atingirmos um círculo, pararemos de seguir em frente.

```
func _on_Jumper_area_entered(area):
    target = area
    velocity = Vector2()
```

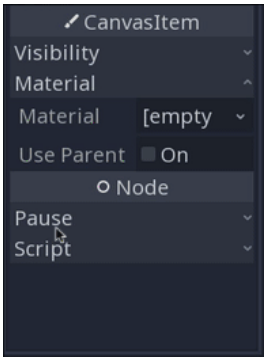
Se formos capturados por um círculo, queremos girar em torno dele. Vamos adicionar um pivô no círculo e combinar sua transformação para que nossa orientação esteja sempre voltada para fora. Caso contrário, avançamos em linha reta.

```
func _physics_process(delta):
```

```
if target:
    transform = target.orbit_position.global_transform
else:
    position += velocity * delta
```

Color Shader

Vamos usar um pequeno sombreador para personalizar a cor do Sprite. Selecionamos o Sprite na propriedade *Material* e adicionamos um novo ShaderMaterial. Clicamos nele e em *Shader* selecionamos “New Shader”, então clicamos nele. O painel do editor de sombreador será aberto na parte inferior.

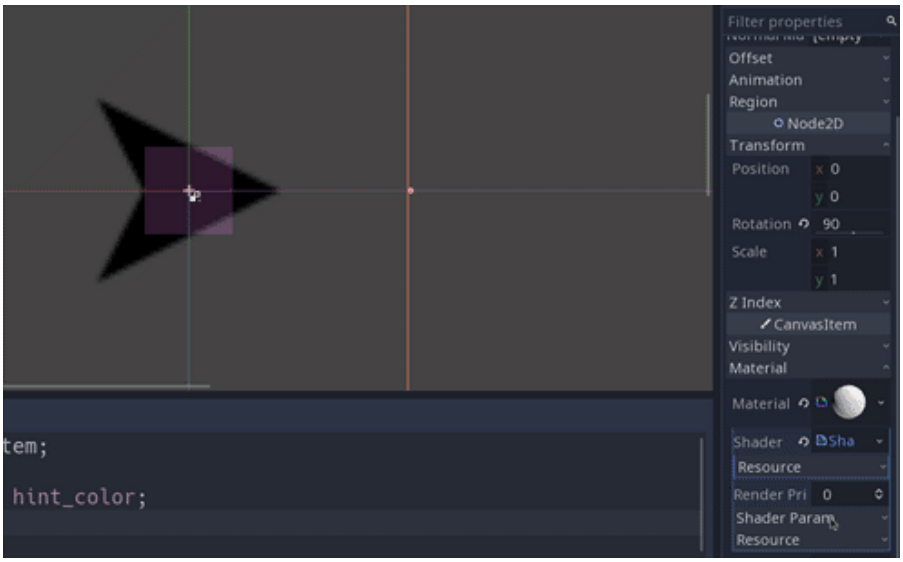


Aqui está o código para o nosso sombreador de cor. Ele usa uma variável para a cor, o que nos permite escolher um valor do Inspetor ou do nosso script de jogo. Em seguida, ele altera todos os pixels visíveis da textura para aquela cor, preservando o valor alfa (transparência).

```
shader_type canvas_item;
uniform vec4 color : hint_color;
void fragment() {
    COLOR.rgb = color.rgb;
```

```
    COLOR.a = texture(TEXTURE, UV).a;

}
```

Agora você verá uma seção *Shader Params* no Inspector, onde podemos definir um valor de cor:



Queremos usar esse mesmo shader em outro lugar, então na propriedade *Shader* escolhemos “Salvar” e salvamos como res://objects/color.shader.

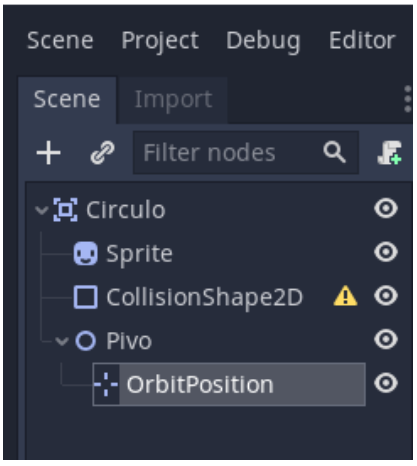
ANOTAÇÕES

AULA 03

Círculo

O segundo objeto do jogo é o círculo, que será instanciado várias vezes conforme o jogo avança. Eventualmente, iremos adicionar uma variedade de comportamentos, como mover, encolher, etc., mas para esta primeira interação, queremos apenas capturar o jogador.

Aqui está a configuração do nó inicial:



- Area2D ("Círculo")
    - Sprite
    - CollisionShape2D
    - Node2D ("Pivô")
        - Position2D (“OrbitPosition”)

O nó “Pivot” será usado para fazermos o jogador orbitar no círculo. A “OrbitPosition” será compensada por qualquer que seja o tamanho do círculo, e o jogador irá segui-lo.

Usamos res://assets/images/circle1_n.png como a textura do Sprite. Enquanto estamos aqui, adicionamos um ShaderMaterial e escolhemos “Carregar” para usar o que salvamos color.shader anteriormente.

Em seguida, adicionamos uma forma de círculo ao CollisionShape2D e anexamos um script ao nó raiz.

```
extends Area2D


onready var orbit_position = $Pivot/OrbitPosition
var radius = 100
var rotation_speed = PI


func _ready():
    init()


func init(_radius=radius):
    radius = _radius
    $CollisionShape2D.shape = $CollisionShape2D.shape.duplicate()
    $CollisionShape2D.shape.radius = radius
    var img_size = $Sprite.texture.get_size().x / 2
    $Sprite.scale = Vector2(1, 1) * radius / img_size
    orbit_position.position.x = radius + 25
```

```
func _process(delta):

    $Pivot.rotation += rotation_speed * delta
```

Na init()função, configuramos o tamanho do círculo, com base no dado *radius*. Precisamos dimensionar a forma de colisão, bem como dimensionar a textura para corresponder. Tente rodar a cena com diferentes valores de radius para testar.

ANOTAÇÕES

AULA 04

Cena Principal

Agora podemos testar a interação!

Criamos uma cena “Principal” usando uma instância Node2D e o Saltador e o Círculo nela. Organize-os de forma que o saltador atinja o círculo (a velocidade padrão do saltador é (100, 0)).

Tente correr. Você deve ver o saltador ser capturado pelo círculo e começar a orbitar. Clicar com o mouse deve enviar o saltador voando em qualquer direção para onde ele esteja apontando.

Círculos de Geração

Na parte anterior, criamos o objeto Saltador e Círculo, que compõem a maior parte do jogo. Agora precisamos adicionar a progressão: uma série contínua de círculos gerados, enquanto o jogador não errar.

Expandindo a cena principal

Vamos adicionar mais alguns nós ao Principal:

- **Position2D (“StartPosition”)**

Isso marcará a posição inicial do jogo. Coloque-o próximo ao centro da parte inferior da tela.

- **Camera2D**

A câmera seguirá o jogador conforme ele se move.

Vamos também configurar a câmera. Definimos seu *deslocamento* como (0, -200)- isso garantirá que possamos ver mais do mundo à nossa frente. Também defina *Atual* como “Ligado”.

Fazendo o script da cena principal

Remova as instâncias de saltador e círculo que criamos manualmente. Vamos adicioná-los no código daqui para frente.

Adicione o seguinte a saltador.gd:

```
signal captured
```

Iremos emitir este sinal quando o jumper atingir um círculo:

```
func _on_Jumper_area_entered(area):
    target = area
    velocity = Vector2.ZERO
    emit_signal("captured", area)
```

E vamos mudar a init()função no círculo para também aceitar uma posição:

```
func init(_position, _radius=radius):
```

```
    position = _position
```

Agora vamos adicionar um script à cena principal:

```
extends Node

var Circle = preload("res://objects/Circulo.tscn")

var Jumper = preload("res://objects/Saltador.tscn")



var player
```

Precisamos de referências a ambos os objetos para que possamos instanciá-los quando necessário.

```
func _ready():

    randomize()

    new_game()
```

Isso é temporário, mais tarde teremos uma IU com um botão Iniciar, para chamar a nova função do jogo.

```
func new_game():

    $Camera2D.position = $StartPosition.position

    player = Jumper.instance()

    player.position = $StartPosition.position

    add_child(player)

    player.connect("captured", self, "_on_Jumper_captured")
```

```
spawn_circle($StartPosition.position)
```

A função new_game() inicializa o jogo, gerando um jogador e um círculo na posição inicial e configurando a câmera.

```
func spawn_circle(_position=null):
    var c = Circle.instance()
    if !_position:
        var x = rand_range(-150, 150)
        var y = rand_range(-500, -400)
        c.position = player.target.position + Vector2(x, y)
    add_child(c)
    c.init(_position)
```

Aqui está nossa função spawn_circle(). Se for ultrapassada uma posição, ela a usará, caso contrário, escolhemos uma posição aleatória a alguma distância do alvo atual.

Esses são números temporários, assim que tivermos mais do jogo instalado e funcionando, veremos o quanto eles precisam ser ajustados.

```
func _on_Jumper_captured(object):
    $Camera2D.position = object.position
    call_deferred("spawn_circle")
```

Finalmente, precisamos da função que processa o sinal captured do saltador. Vamos mover a câmera para o novo círculo e gerar outro. Observe que, como essa função é chamada durante o processamento da física, obteremos um

erro se tentarmos adicionar à árvore de cena. Usar call_deferred() diz ao mecanismo para executar essa função assim que for seguro fazê-lo.

Experimente. Você deve ser capaz de pular de círculo em círculo, quantos você conseguiu?

Uma coisa chocante é que a câmera “se teletransporta” quando se move para o próximo círculo. Podemos melhorar isso habilitando a *suavização* na câmera. O *Smoothing / Speed* controla a rapidez com que a câmera interpola para a nova posição. Experimente algo entre 5 e 10.

Ajustes

Também é chocante que, quando atingimos um círculo, não começamos a girar no local em que atingimos. Adicionamos isto à função _on_Jumper_area_entered() do jumper:

```
target.get_node("Pivot").rotation = (position - target.position).angle()
```

Vamos também adicionar isto ao círculo init():

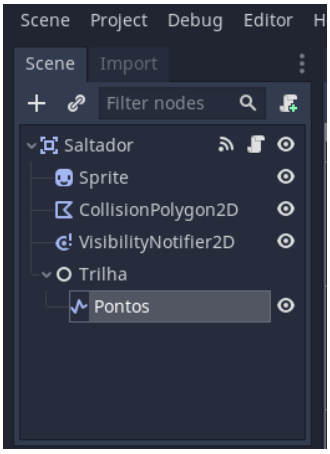
```
rotation_speed *= pow(-1, randi() % 2)
```

Isso muda aleatoriamente a velocidade de rotação para positiva ou negativa, portanto, nem sempre orbitaremos na mesma direção.

ANOTAÇÕES

AULA 05

Trilha

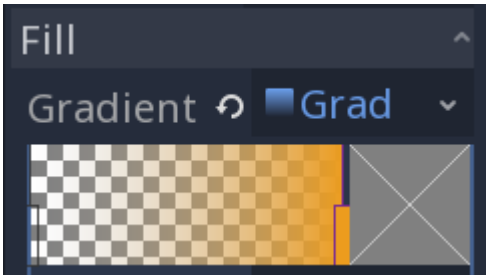


Adicionamos estes **nós** ao saltador:

- Node ("Trilha")
    - Line2D ("Pontos")

Vamos usar isso para fazer uma trilha que sai atrás do jogador. Mais tarde, vamos torná-lo mais atraente visualmente, mas, por enquanto, vamos ficar com um gradiente simples.

No *Preenchimento,* adicionamos um novo gradiente, então mova do transparente para uma cor de sua escolha:



Agora, no script do saltador, vamos adicionar o seguinte:

```
onready var trail = $Trail/Points


var trail_length = 25
```

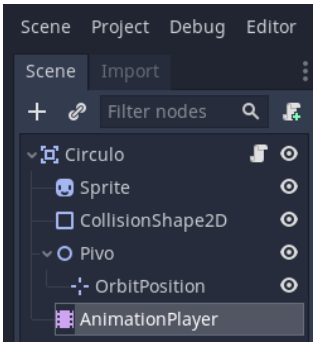
E então no _physics_process():

```
if trail.points.size() > trail_length:
    trail.remove_point(0)
trail.add_point(position)
```

Animações Circulares

Por fim, adicionaremos alguns recursos visuais aos círculos. Primeiro, adicionaremos um efeito quando o jogador pula e o círculo desaparece. Então, vamos adicionar um efeito de captura para quando atingirmos um círculo.

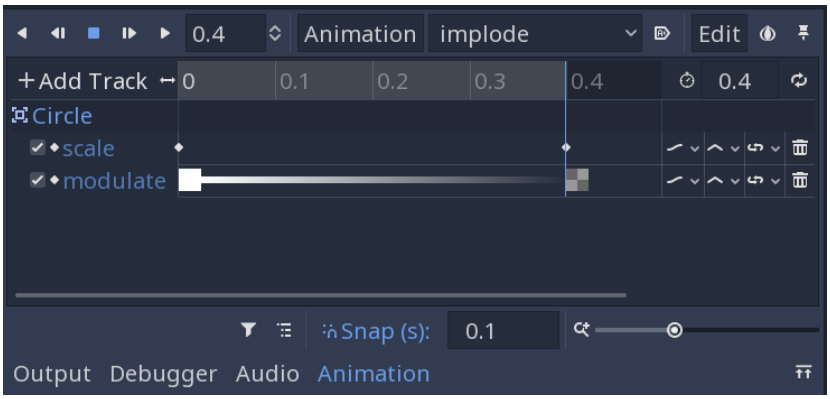
Para isso, adicionamos um nó AnimationPlayer ao Círculo, como na imagem ao lado:



Animação implode

Adicionamos uma nova animação chamada “implode”. Definimos o comprimento como 0,4 e duas propriedades do quadro-chave do nó Area2D raiz: *Dimensionar* em (1, 1) e modular em seu padrão ((1, 1, 1, 1)).

Em seguida, mova o controle deslizante totalmente até o final e digite os valores (0.1, 0.1) e (1, 1, 1, 0), esse é o valor “alfa” da cor.

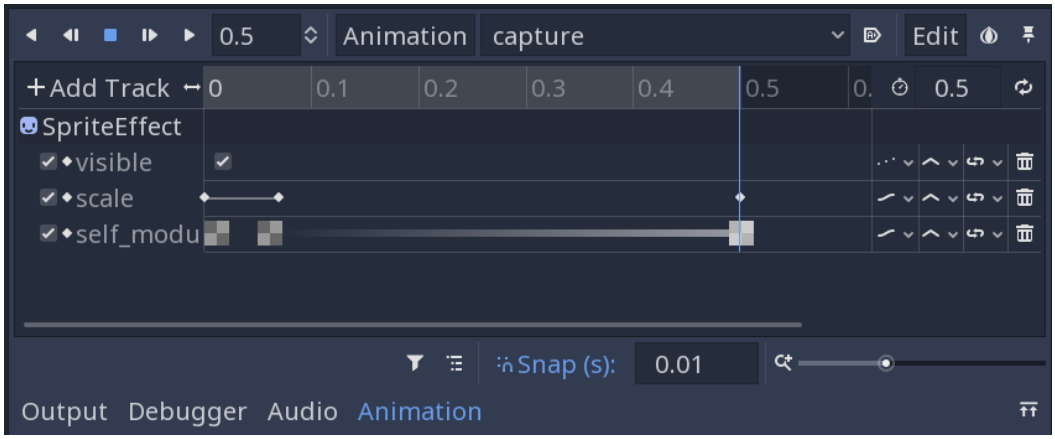


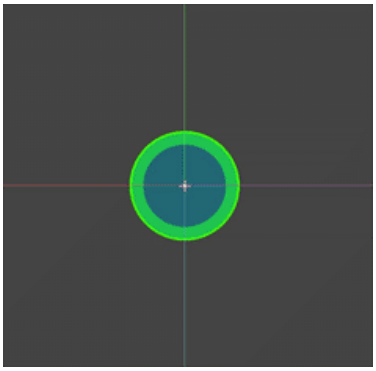
ANOTAÇÕES

AULA 06

Capturar animação

A animação de captura é um pouco mais complexa. Duplicamos o Sprite e chamamos o de SpriteEffect. Definimos sua propriedade Visible desativada. Vamos animar este segundo anel ampliando o círculo principal.





Aqui estão as funções a serem adicionadas ao script do círculo:

```
func capture():
    $AnimationPlayer.play("capture")


func implode():
    if !$AnimationPlayer.is_playing():
        $AnimationPlayer.play("implode")
    yield($AnimationPlayer, "animation_finished")
    queue_free()
```

Em Saltador.gd nossa função de salto torna-se:

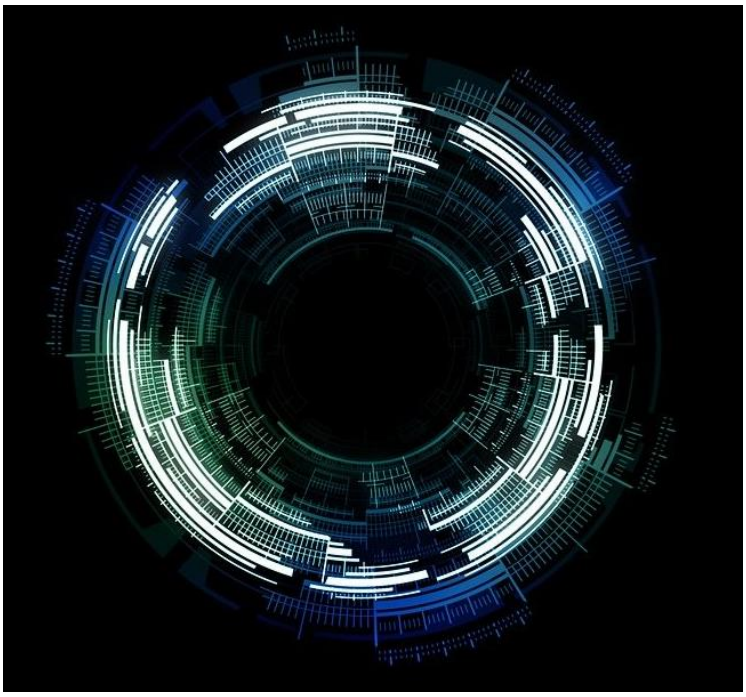
```
func jump():
    target.implode()
    target = null
    velocity = transform.x * jump_speed
```

E em Main, nosso método de captura é codificada assim:

```
func _on_Jumper_captured(object):
    $Camera2D.position = object.position
    object.capture()
    call_deferred("spawn_circle")
```

ANOTAÇÕES

AULA 07



Círculos limitados

No início, colocamos a jogabilidade básica em funcionamento. Agora vamos começar a adicionar alguns modos diferentes de círculos.

Modos de círculo

Eventualmente, teremos muitos modos diferentes, mas vamos começar com o modo “limitado”: o círculo só permite um determinado número de órbitas antes de desaparecer. Primeiro, vamos adicionar um nó Label para mostrar o número de órbitas restantes. Digite um número (1) no campo de texto para que possamos ver como fica.

Na seção *Fontes personalizadas*, adicionamos uma nova DynamicFont, carregamos os *dados* da *fonte* da pasta de ativos e defina o tamanho como 64. Para alinhar a etiqueta, no menu “Layout”, escolha “Centro”.

Adicione as seguintes novas variáveis no topo de Circle.gd:

```
enum MODES {STATIC, LIMITED}


var mode = MODES.STATIC
var num_orbits = 3  # Número de órbitas até que o círculo desapareça


var current_orbits = 0  # Número de órbitas que o jumper completou


var orbit_start = null  # Onde as órbitas começaram
```

Em seguida, precisamos definir o modo:

```
func set_mode(_mode):
    mode = _mode
    match mode:
        MODES.STATIC:
            $Label.hide()
        MODES.LIMITED:
            current_orbits = num_orbits
            $Label.text = str(orbits_left)
            $Label.show()
```

No momento, temos esses dois modos definidos, mas depois iremos adicionar mais.

Vamos também adicionar ao método init() uma maneira de passar um modo. O padrão deveria ser STATIC, mas vamos usar LIMITED agora para que possamos testar:

```
func init(_position, _radius=radius, _mode=MODES.LIMITED):
    set_mode(_mode)
```

O saltador está definindo a posição de rotação quando é capturado. Remova a linha Saltador.gd e coloque-a no método capture() do círculo :

```
func capture(target):
    jumper = target
    $AnimationPlayer.play("capture")
    $Pivot.rotation = (jumper.position - position).angle()
    orbit_start = $Pivot.rotation
```

Observe que agora estamos enviando uma referência ao jumper, portanto, adicione var jumper = null na parte superior e, no script Main.gd, atualize a chamada para ler object.capture(player).

Agora podemos verificar se o jumper completou o círculo e, em caso afirmativo, decrementar current_orbits:

```
func _process(delta):
    $Pivot.rotation += rotation_speed * delta
    if mode == MODES.LIMITED and jumper:
```

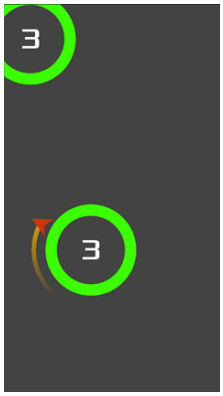
```
            check_orbits()


    func check_orbits():
        # Cheque se o saltador completou uma volta completa ao redor do círculo


        if abs($Pivot.rotation - orbit_start) > 2 * PI:
            current_orbits -= 1
            $Label.text = str(current_orbits)
            if orbits_left <= 0:
                jumper.die()
                jumper = null
                implode()
            orbit_start = $Pivot.rotation
```

Para que isso funcione, precisamos adicionar um die()método ao jumper:

```
    func die():
        target = null
        queue_free()


    func _on_VisibilityNotifier2D_screen_exited():
```

```
        if !target:

            die()
```

Também conectamos o VisibilityNotifier2Dsinal do jumper para que possamos remover o player quando ele sair da tela. Até aqui, temos esse resultado:



Efeito de círculo

A última coisa que faremos nesta parte é adicionar um efeito de "preenchimento" ao círculo para mostrar que as órbitas estão se esgotando. Para começar, usaremos algum código de desenho dos documentos oficiais da godot:

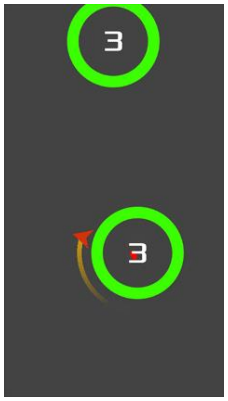
```
    func draw_circle_arc_poly(center, radius, angle_from, angle_to, color):

        var nb_points = 32

        var points_arc = PoolVector2Array()

        points_arc.push_back(center)

        var colors = PoolColorArray([color])


        for i in range(nb_points + 1):

            var angle_point = angle_from + i * (angle_to - angle_from) / nb_points - PI/2
```

```
            points_arc.push_back(center + Vector2(cos(angle_point), sin(angle_point)) * radius)

        draw_polygon(points_arc, colors)
```

Chamaremos esta função em _draw():

```
    func _draw():

        if jumper:

            var r = ((radius - 50) / num_orbits) * (1 + num_orbits - current_orbits)

            draw_circle_arc_poly(Vector2.ZERO, r, orbit_start + PI/2,

                    $Pivot.rotation + PI/2, Color(1, 0, 0))
```

Por último, adicionamos update() ao _physics_process para que seja convocado após cada chamada para check_orbits().



ANOTAÇÕES

AULA 08

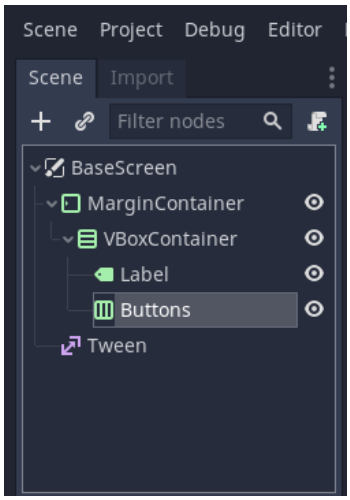
Menus

Agora que temos a jogabilidade básica, é hora de começar a trabalhar na IU. Precisaremos de telas de menu para o título, as configurações e o fim do jogo.

Telas de Menus

As três telas compartilharão um layout comum e algumas funcionalidades, portanto, começaremos com uma cena base da qual todas elas podem herdar. Na nova cena, comece com um CanvasLayer e nomeie como BaseScreen. Salve esta cena na pasta “UI”.

- CanvasLayer (“BaseScreen”)
    - MarginContainer
        - VBoxContainer
            - ✓ Label
            - ✓ HBoxContainer (“Botões”)
    - Tween



O MarginContainer vai garantir que nenhum dos nossos elementos de interface do usuário, fique muito perto da borda da tela. Defina todas as quatro propriedades de *Constantes personalizadas* como 20.

Em seguida, é um VBoxContainer para organizar os elementos principais. Defina suas *Constantes / Separação personalizadas* para 150.

O nó Label exibe o título da tela. Coloque “Título” em seu campo *Texto* e carregue o mesmo recurso de fonte que usamos para os círculos.

Finalmente, adicionamos um HBoxContainer denominado “Botões” que conterá os botões que adicionamos às telas. Definimos sua separação como 75. Em seguida, duplique o nó para que tenhamos outra linha de botões.

A face deve começar fora da tela, então defina o *deslocamento* no nó raiz para (500, 0).

ANOTAÇÕES

AULA 09

Adicionar script na cena

Vamos adicionar um script à cena com a seguinte codificação:

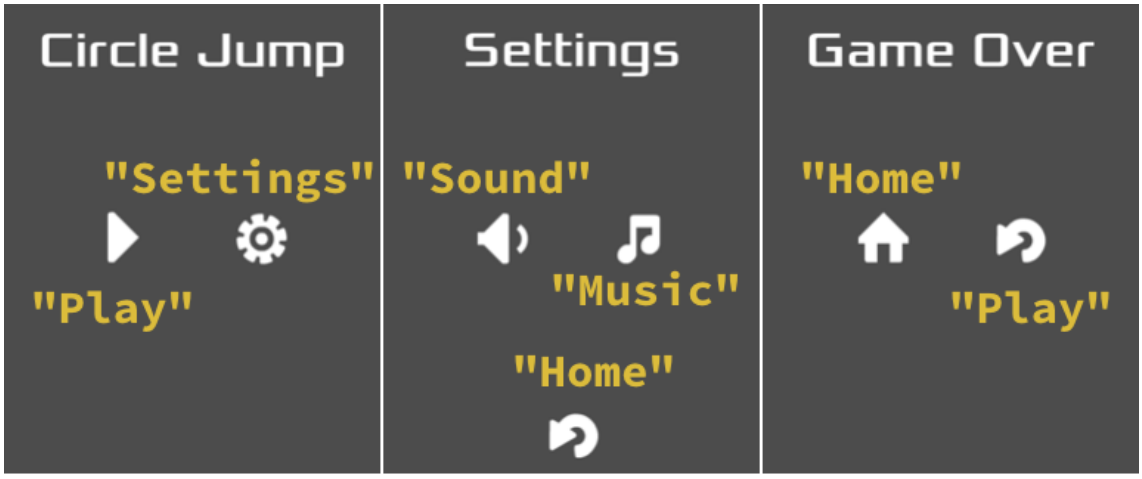
```
extends CanvasLayer


onready var tween = $Tween


func appear():
    tween.interpolate_property(self, "offset:x", 500, 0,
                0.5, Tween.TRANS_BACK, Tween.EASE_IN_OUT)
    tween.start()
```

```
    func disappear():
        tween.interpolate_property(self, "offset:x", 0, 500,
                0.4, Tween.TRANS_BACK, Tween.EASE_IN_OUT)
        tween.start()
```

Este script configura as animações que podemos chamar para fazer a tela aparecer e desaparecer.

Agora podemos fazer nossas três cenas herdadas. Para cada um, nomeie o nó raiz, altere o texto do rótulo e adicione TextureButtons aos recipientes “Botões”. Use as imagens da pasta de ativos para a textura *normal de* cada botão. Nomeie cada botão de acordo com sua função (“Reproduzir”, “Configurações”, etc.) e adicione-o ao grupo “botões”.

Aqui está a aparência das três cenas, usando os nomes dos botões indicados:



ANOTAÇÕES

AULA 10

Telas

Faremos mais uma cena com uma Node raiz chamada “Telas” e instalar as três telas nela. Adicione o seguinte script, que tratará das transições e estados da cena**:**

```
extends Node


signal start_game


var current_screen = null


func _ready():
    register_buttons()
    change_screen($TitleScreen)
```

```
func register_buttons():
    var buttons = get_tree().get_nodes_in_group("buttons")
    for button in buttons:
        button.connect("pressed", self, "_on_button_pressed", [button.name])


func _on_button_pressed(name):
    match name:
        "Home":
            change_screen($TitleScreen)
        "Play":
            change_screen(null)
            yield(get_tree().create_timer(0.5), "timeout")
            emit_signal("start_game")
        "Settings":
            change_screen($SettingsScreen)


func change_screen(new_screen):
    if current_screen:
        current_screen.disappear()
        yield(current_screen.tween, "tween_completed")
    current_screen = new_screen
```

```
        if new_screen:
            current_screen.appear()
            yield(current_screen.tween, "tween_completed")


    func game_over():
        change_screen($GameOverScreen)
```

Este script conecta todos os nossos botões vinculando o sinal pressed e passando o nome do botão como parâmetro. Isso permite que nosso _on_button_pressed() método decida o que cada botão deve fazer.

O método change_screen() lida com a transição para a tela selecionada, incluindo a opção null para quando não queremos exibir uma tela.

Execute-o para testar as transições de tela:

Criamos uma instância dessa cena em Main e, a seguir, conectamos seu sinal start_game à função new_game() em principal. Tente executar o jogo e você conseguirá iniciá-lo. A última parte será conectar a condição game over.

No Jumper, adicione um sinal chamado de die e emitimos esse sinal no método do notificador de visibilidade.

Adicionamos isto à função new_game():

```
player.connect("died", self, "_on_Jumper_died")
```

Em seguida, adicionamos esta nova função, que irá garantir que todos os círculos sejam removidos quando o jogador morrer.

```
func _on_Jumper_died():
    get_tree().call_group("circles", "implode")
    $Screens.game_over()
```

ANOTAÇÕES

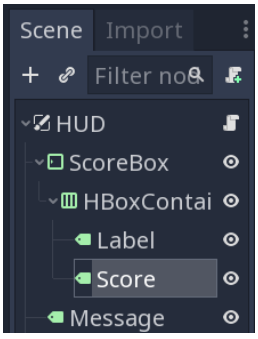
AULA 11

Pontuação e HUD

Na última parte, adicionamos UI na forma de menus para iniciar e configurar o jogo. Também precisamos de uma IU para exibir informações do jogo, como pontuação.

Cena HUD

Adicione uma nova cena com uma CanvasLayer raiz para ser nosso HUD. Dê a ele dois filhos: um MarginContainer denominado “ScoreBox” e um `Rótulo” denominado “Mensagem”. Sua árvore de cena deve ser semelhante a esta:



Definimos o layout de ScoreBox como “Bottom Wide” e as *Constantes personalizadas* como 20. Adicione um HBoxContainer filho e sob esses dois nós, Label. Nomeie o segundo rótulo como “Pontuação” e coloque 100 sua propriedade *Texto*. Defina o HBoxContainer's *alinhamento* com ‘End’.

Adicione o mesmo recurso DynamicFont a ambos os rótulos, mas escolha “Tornar único” no primeiro rótulo e defina seu tamanho para 32. Defina sua propriedade *Text* como “Score”. Em _Size Flags / Vertical, defina “Fill”. Seu layout deve ficar assim:

Agora, para o nó Message, carregue a fonte e defina *Text* como “Message” para que tenhamos algo para ver. Escolha também “Tornar único” no recurso de fonte (você verá o porquê na próxima seção). Defina *Align* and *Valign* para “Center” e *Clip Text* para “On”. Para layout, escolha “Center Wide”. Além disso, defina *Grow Direction* / *Vertical* para “Both”.

ANOTAÇÕES

AULA 12

Animação de mensagem

Esta mensagem mostrará informações durante o jogo (aumento de nível, bônus, etc.). Queremos que seja animado, apareça e depois desapareça. Para isso adicione um AnimationPlayer à cena.

Faremos duas animações: uma para definir os valores iniciais e outra para animar a exibição da mensagem. Adicione a primeira animação, “init” e clique no botão “Autoplay on Load”. Defina o comprimento para 0.1.

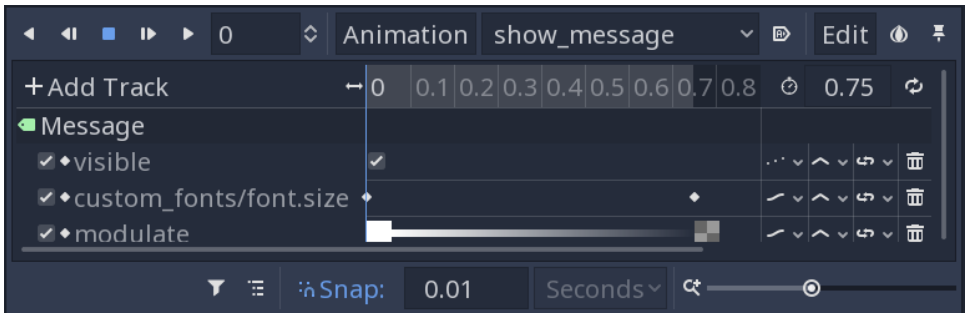
Adicione um quadro-chave de cada vez, use 0 para *Fonte / Tamanho* (64) e 1 para *Visível* definido como “Desligado”.

Adicione a segunda animação, “show_message”. Defina seu comprimento como 0.75 e *Visibilidade* do quadro-chave como “On”.

A seguir, definiremos o quadro-chave de *Font / Size* de 64 no momento 0 e 200 no final. Defina o *modo de atualização* da faixa como “Contínuo”.

Também queremos que ele desapareça à medida que cresce, portanto, faça o quadro-chave do valor modular alfa de 255 para 0.

Veja como devem ser as configurações de animação:



E a animação quando é reproduzida:

HUD Script

Agora vamos adicionar um script à cena, com métodos para atualizar as telas:

```
extends CanvasLayer


func show_message(text):
    $Message.text = text
    $AnimationPlayer.play("show_message")


func hide():
```

```
    $ScoreBox.hide()


func show():
    $ScoreBox.show()


func update_score(value):
    $ScoreBox/HBoxContainer/Score.text = str(value)
```

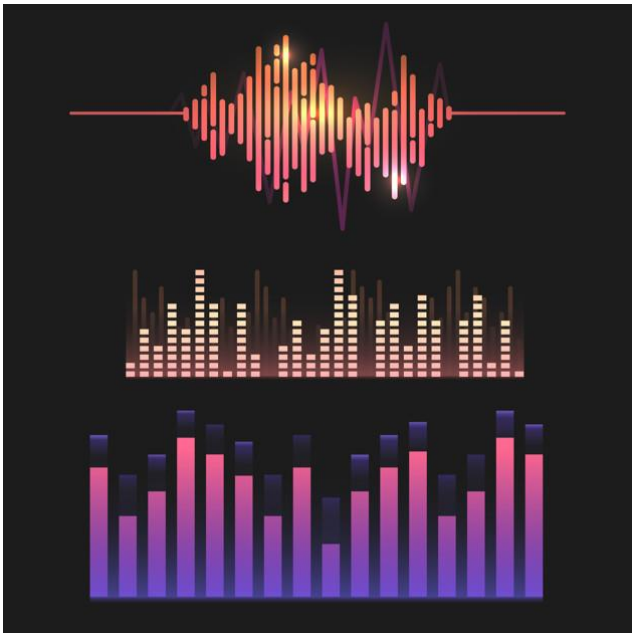
Instanciamos o HUD na cena principal e adicionamos $HUD.hide() às funções _ready() e _on_Jumper_died(). Em new_game() precisamos mostrar o HUD e exibir uma mensagem:

```
$HUD.show()
$HUD.show_message("Go!")
```

Para adicionar a pontuação, crie uma score variável e defina-a como 0 em new_game(). Em _on_Jumper_captured() incremento-o com 1. Certifique-se de ligar $HUD.update_score(score) após cada um deles.

ANOTAÇÕES

AULA 13



Sons e Cores

Configurações de Singleton

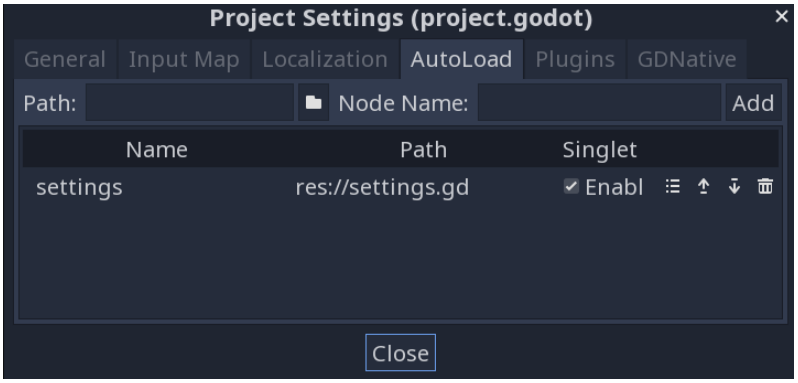
Primeiro, adicionaremos um novo script escolhendo *Arquivo -> Novo Script*. Dê um nome ao script settings.gd.

Neste script, colocaremos as configurações do jogo:

```
var enable_sound = true

var enable_music = true
```

```
var circles_per_level = 5
```

Adicionamos o script como um carregamento automático abrindo “Configurações do projeto” e selecionando a guia “Carregamentos automáticos”. Clique na pasta para carregar o script e clique em “Adicionar”.



Adicionando som

Para reproduzir sons, iremos adicionar vários nós AudioStreamPlayer a diferentes cenas.

- Primeiro, adicione um à cena Main e nomeie-o “Música”. Para seu uso de propriedade *Stream* res://assets/audio/Music_Light-Puzzles.ogg .
- Para a Screen scena, adicione outro chamado “Click”, que tocará quando tocarmos nos botões. Use menu_click.wavda pasta de ativos.
- Na Circle cena, adicione um reprodutor de áudio chamado “Beep” e use o arquivo de som 89.ogg.
- Finalmente, no Jumper, precisamos de dois efeitos sonoros: “Jump” e “Capture”. Use 70.ogg e 88.ogg, respectivamente.

Agora, para reproduzir os sons, podemos chamar seus métodos play(). Adicione isto a Main.new_game():

```
if settings.enable_music:
```

```
$Music.play()
```

Adicione também ao Main.on_Jumper_died():

```
if settings.enable_music:
    $Music.stop()
```

Além disso, adicione ao Screens.gd _on_button_pressed():

```
if settings.enable_sound:
    $Click.play()
```

No círculo, queremos reproduzir o som Beep quando um círculo limitado completa uma órbita completa. Isso está em check_orbits():

```
current_orbits -= 1
if settings.enable_sound:
    $Beep.play()
```

E em Jumper.gd, adicionamos os sons assim:

```
func jump():
    target.implode()
    target = null
    velocity = transform.x * jump_speed
    if settings.enable_sound:
```

```
            $Jump.play()


    func _on_Jumper_area_entered(area):
        target = area
        velocity = Vector2.ZERO
        emit_signal("captured", area)
        if settings.enable_sound:
            $Capture.play()
```

ANOTAÇÕES

AULA 14

Definições de Som

Agora que o som está funcionando, podemos conectar os botões na tela “Configurações” que podem alternar entre som e música.

A aparência do botão precisa ser alterada para corresponder ao estado atual de ativação / desativação da propriedade. Carregaremos as texturas primeiro para que possamos atribuí-las conforme necessário:

```
var sound_buttons = {true:
preload("res://assets/images/buttons/audioOn.png"),

false: preload("res://assets/images/buttons/audioOff.png")}

var music_buttons = {true:
preload("res://assets/images/buttons/musicOn.png"),

false:
preload("res://assets/images/buttons/musicOff.png")}
```

No momento, não estamos manipulando os botões quando eles estão pressionados. O problema é que estamos passando o nome do botão, o que não nos permite mudar sua textura. Em vez disso, vamos refatorar register_buttons() para passar uma referência ao próprio botão:

```
button.connect("pressed", self, "_on_button_pressed", [button])
```

Então podemos atualizar _on_button_pressed() assim:

```
func _on_button_pressed(button):
    if settings.enable_sound:
        $Click.play()
    match button.name:
        "Home":
            change_screen($TitleScreen)
        "Play":
            change_screen(null)
            yield(get_tree().create_timer(0.5), "timeout")
            emit_signal("start_game")
        "Settings":
            change_screen($SettingsScreen)
        "Sound":
            settings.enable_sound = !settings.enable_sound
            button.texture_normal = sound_buttons[settings.enable_sound]
```

```
        "Music":
            settings.enable_music = !settings.enable_music
            button.texture_normal = music_buttons[settings.enable_music]
```

Temas de cores

Também iremos adicionar uma maneira de ter diferentes esquemas de cores: talvez como uma opção de configuração, ou eles mudam conforme o jogador atinge níveis mais altos.

Armazenaremos os dados do esquema de cores em um dicionário, sendo as chaves o “nome” do esquema. Cada esquema de cores também será um dicionário, com as teclas indicando o componente do jogo que usará aquela cor.

Adicione isto a settings.gd:

```
var color_schemes = {
    "NEON1": {
        'background': Color8(0, 0, 0),
        'player_body': Color8(203, 255, 0),
        'player_trail': Color8(204, 0, 255),
        'circle_fill': Color8(255, 0, 110),
        'circle_static': Color8(0, 255, 102),
        'circle_limited': Color8(204, 0, 255)
    },
    "NEON2": {
        'background': Color8(0, 0, 0),
```

```
            'player_body': Color8(246, 255, 0),
            'player_trail': Color8(255, 255, 255),
            'circle_fill': Color8(255, 0, 110),
            'circle_static': Color8(151, 255, 48),
            'circle_limited': Color8(127, 0, 255)
        },
        "NEON3": {
            'background': Color8(0, 0, 0),
            'player_body': Color8(255, 0, 187),
            'player_trail': Color8(255, 148, 0),
            'circle_fill': Color8(255, 148, 0),
            'circle_static': Color8(170, 255, 0),
            'circle_limited': Color8(204, 0, 255)
        }
    }


    var theme = color_schemes["NEON1"]
```

Agora, em cada objeto, precisamos definir as cores com base nas propriedades de configurações.

Para o círculo, a cor é definida usando o recurso de material do sombreador. Como os recursos são compartilhados, isso significa que mudar a cor de um círculo mudaria todos eles. Vamos tornar o material de cada círculo único para evitar isso:

```
$Sprite.material = $Sprite.material.duplicate()

$SpriteEffect.material = $Sprite.material
```

A cor do círculo é determinada pelo modo que está usando, então set_mode() é onde escolheremos a cor:

```
func set_mode(_mode):
    mode = _mode
    var color
    match mode:
        MODES.STATIC:
            $Label.hide()
            color = settings.theme["circle_static"]
        MODES.LIMITED:
            current_orbits = num_orbits
            $Label.text = str(current_orbits)
            $Label.show()
            color = settings.theme["circle_limited"]
    $Sprite.material.set_shader_param("color", color)
```

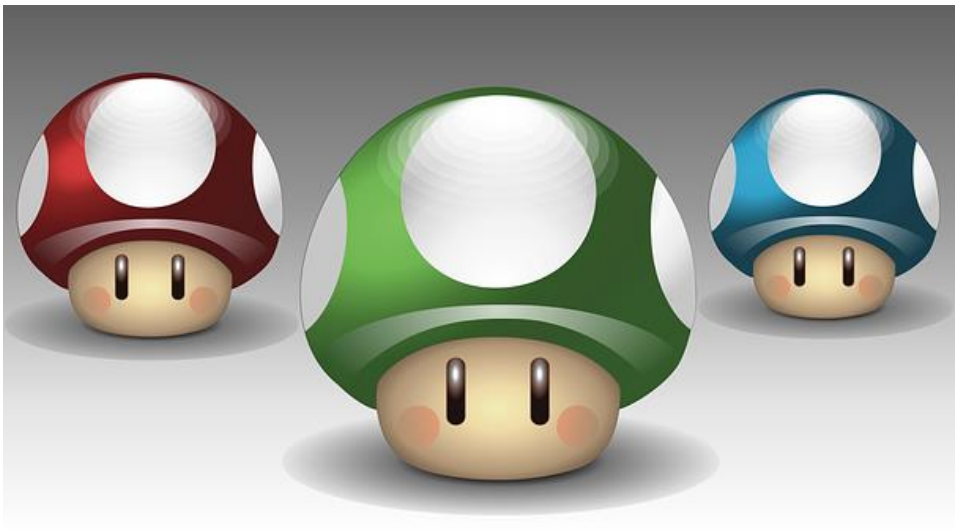
Então, na função _draw() em que estamos preenchendo o círculo limitado, substitua vermelho por settings.theme["circle_fill"].

Para o jogador, defina a cor em _ready():

```
    func _ready():

        $Sprite.material.set_shader_param("color",
settings.theme["player_body"])

        $Trail/Points.default_color = settings.theme["player_trail"]
```

ANOTAÇÕES

AULA 15



Corrigindo um bug

Nossa primeira tarefa é consertar um bug em nosso sistema de menus. Pressionar o botão “Iniciar” dará início a um novo jogo, mas como a tela está se movendo, ele pode ser pressionado novamente. Tente “enviar spam” para o botão Iniciar - o desastre acontecerá!

Podemos consertar isso desabilitando os botões enquanto a transição da tela está acontecendo. Como colocamos todos os botões em um grupo de “botões”, podemos fazer isso facilmente com call_group().

Aqui está o atualizado BaseScreen.gd:

```
extends CanvasLayer


onready var tween = $Tween
```

```
func appear():
    get_tree().call_group("buttons", "set_disabled", false)
    tween.interpolate_property(self, "offset:x", 500, 0,
                0.5, Tween.TRANS_BACK, Tween.EASE_IN_OUT)
    tween.start()

func disappear():
    get_tree().call_group("buttons", "set_disabled", true)
    tween.interpolate_property(self, "offset:x", 0, 500,
                0.5, Tween.TRANS_BACK, Tween.EASE_IN_OUT)
    tween.start()
```

Pontuação e nível

Conforme nossa pontuação aumenta, queremos que a dificuldade do jogo também aumente. Isso significa que, quando conseguirmos pontos, precisaremos verificar se ultrapassamos um determinado limite (circles_per_level). Podemos também ter outras coisas que nos dão pontos além de pular em um círculo. Para facilitar o manuseio, daremos à nossa variável de pontuação um método setget no script principal:

```
var score = 0 setget set_score
var level = 0
```

Depois de feito, atualize o new_game() para usar esse método:

```
func new_game():
    self.score = 0
    level = 1
```

Faça o mesmo com a alteração da pontuação em _on_Jumper_captured() e moveremos a atualização do HUD para o nosso novo método set_score():

```
func _on_Jumper_captured(object):
    $Camera2D.position = object.position
    object.capture(player)
    call_deferred("spawn_circle")
    self.score += 1


func set_score(value):
    score = value
    $HUD.update_score(score)
    if score > 0 and score % settings.circles_per_level == 0:
        level += 1
        $HUD.show_message("Level %s" % str(level))
```

Experimente o jogo e você verá uma mensagem “Nível 2” na tela quando chegar a cinco pontos.

Movendo Círculos

Parte da progressão de nível será o aumento da dificuldade. Uma maneira de fazer isso é movendo alguns círculos. Já temos vários tipos de círculo (estáticos e limitados), mas qualquer um deles deve ser capaz de se mover, então este não será um novo tipo de círculo. Em vez disso, será uma propriedade que qualquer círculo pode ter.

Abra a cena Circulo e adicione um nó Tween chamado “MoveTween”. Adicione isto ao topo do script do círculo:

```
onready var move_tween = $MoveTween


var move_range = 100  # Distância que o círculo se move.


var move_speed = 1.0  # A velocidade de movimento do círculo.
```

Se move_range for 0, teremos um círculo imóvel. Faremos o padrão 100 para que possamos testá-lo.

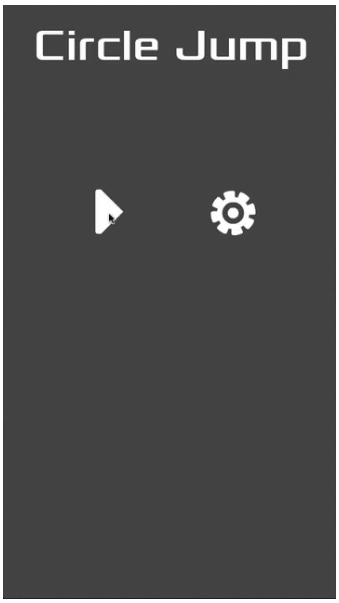
Para lidar com o movimento, vamos iniciar o MoveTween. Quando terminar, vamos reiniciá-lo na direção oposta, usando o tween_completedsinal.

Este é o código para iniciar o movimento. Conecte o sinal tween_completed a esta função:

```
func set_tween(object=null, key=null):
    if move_range == 0:
        return
    move_range *= -1
```

```
        move_tween.interpolate_property(self, "position:x",
                position.x, position.x + move_range,
                move_speed, Tween.TRANS_QUAD,
Tween.EASE_IN_OUT)
        move_tween.start()
```

Finalmente, adicionaremos set_tween() ao final da init()função e podemos experimentá-lo.



Salvando configurações

Adicionamos três propriedades de alternância no jogo - o que funciona bem - porém as configurações não são preservadas quando encerramos. Precisamos salvar essas configurações para que na próxima vez que você executar o jogo, elas persistam.

Primeiro, definiremos nosso arquivo de configurações em res://settings.gd:

```
var settings_file = "user://settings.save"
```

A seguir, adicionaremos funções de salvar / carregar para as três configurações do jogo que queremos salvar.

```
func save_settings():
    var f = File.new()
    f.open(settings_file, File.WRITE)
    f.store_var(enable_sound)
    f.store_var(enable_music)
    f.store_var(enable_ads)
    f.close()


func load_settings():
    var f = File.new()
    if f.file_exists(settings_file):
        f.open(settings_file, File.READ)
        enable_sound = f.get_var()
        enable_music = f.get_var()
        self.enable_ads = f.get_var()
        f.close()
```

Chamada load_settings() em _ready() e save_settings() no final set_enable_ads(). Além disso, Screens.gd precisamos salvar o estado

quando as configurações de som / música são alteradas, portanto, adicione settings.save_settings() a cada uma dessas partes da instrução match.

Outro problema que teremos é quando o jogo começa, os ícones no menu de configurações não refletem o estado que acabamos de carregar do arquivo salvo. Podemos definir aquele em register_buttons() que já está passando por todos os botões para conectar seus sinais:

```
for button in buttons:
    button.connect("pressed", self, "_on_button_pressed", [button])
    match button.name:
        "Ads":
            if settings.enable_ads:
                button.text = "Disable Ads"
            else:
                button.text = "Enable Ads"
        "Sound":
            button.texture_normal = sound_buttons[settings.enable_sound]
        "Music":
            button.texture_normal = music_buttons[settings.enable_music]
```

ANOTAÇÕES