

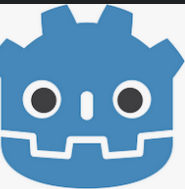


Godot C#: Plataforma Pixel 2D

Prof. Thiago Felski Pereira, MSc.

Visão Geral

- Ajustar a resolução do jogo.
- Criar objetos estáticos para o cenário.
- Entender o código do movimento do personagem.



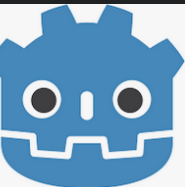
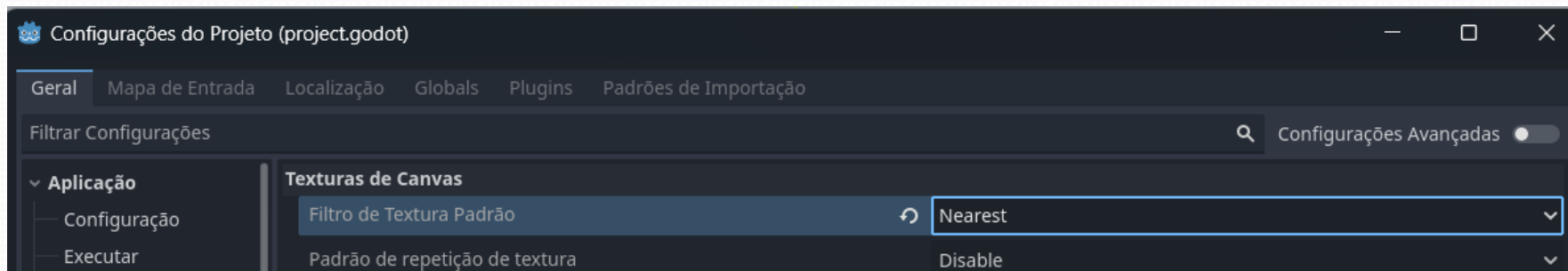
[2]



UNIVALI

Texturas

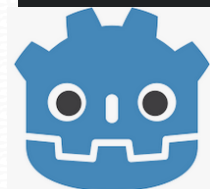
- As texturas de um jogo em PixelArte são parte da identidade do jogo, por isso é importante garantir que o jogo mantenha essas características.
- O Godot tenta aplicar filtros para melhorar a resolução dos objetos dos nossos jogos, mas isso acaba prejudicando algumas artes, como é o caso de jogos com pixelart.
- É possível desativar os filtros indo em
 - Menu -> Projeto -> Project Settings -> Renderização -> Texturas -> Filtro de Textura Padrão
 - e trocar de **Linear** para **Nearest**



3

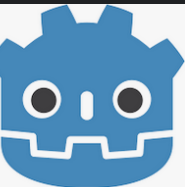
Tamanho da Janela de Jogo

- Editar o campo de Visão (Viewport)
 - Menu -> Projeto -> Project Settings -> Exibição -> Janela
 - Edite a Altura e Largura do Viewport para 320x180, isso fará com que a janela de jogo fique bem pequena.
 - Para testar isso crie uma cena com um Node2D, altere o nome para Mapa1 e execute sua aplicação (F6).
- Na mesma tela mude o
 - Esticar -> Modo para **canvas_items**, isso fará com que o jogo estique junto com a janela do jogo.
 - Para alterar o tamanho da janela, pois Viewport só ajusta o campo de visão. Ative (no canto superior direito as configurações avançadas. Em seguida, altere a **substituição da Largura/Altura** da janela para 1280x720



StaticBody2D

- É corpo físico que não pode ser movido por forças ou contatos externos, mas pode ser movido manualmente por outros meios, como código.
- Note que outros objetos podem colidir com ele, mas ele permanecerá estático independente da força que está sendo aplicada nele.
- Um corpo estático é ideal para criar elementos que ficarão imóveis no cenário como: chão, plataformas, etc.



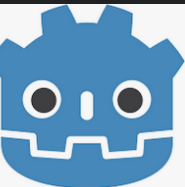
5



UNIVALI

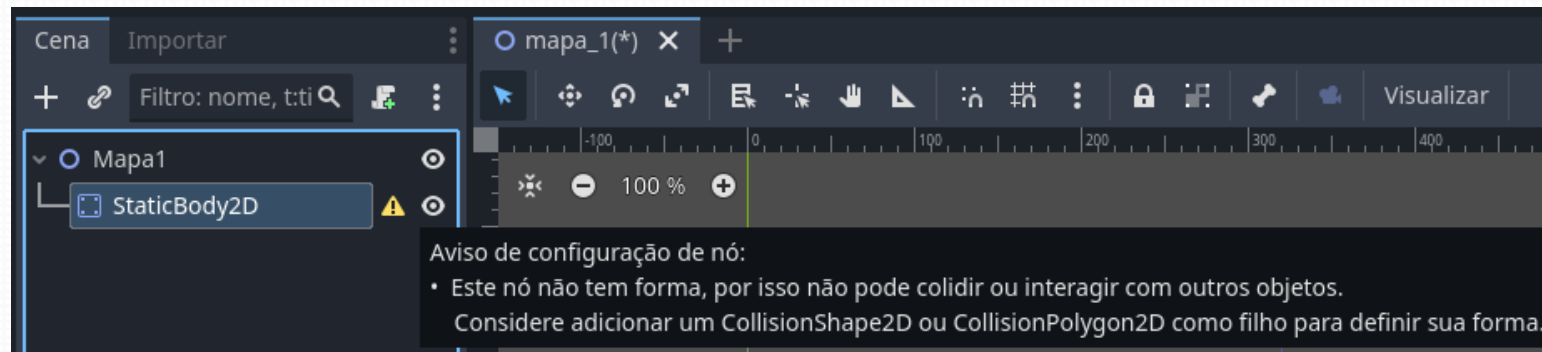
StaticBody2D


- É corpo físico que não pode ser movido por forças ou contatos externos, mas pode ser movido manualmente por outros meios, como código.
- Note que outros objetos podem colidir com ele, mas ele permanecerá estático independente da força que está sendo aplicada nele.
- Um corpo estático é ideal para criar elementos que ficarão imóveis no cenário como: chão, plataformas, etc.
- O Mapa1 será construído com um objeto estático que iremos desenhar com um polígono.



Mapa1

- Inclua um StaticBody2D no seu Mapa1. Observe que ao incluir esse nó na cena aparece um alerta dizendo que objetos físicos precisam de uma área de colisão.

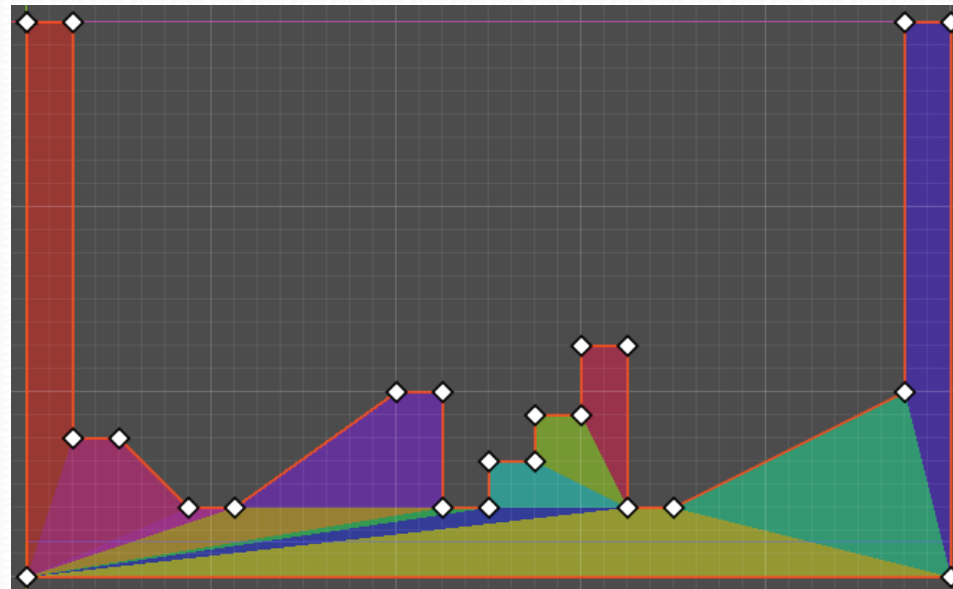


- Inclua um nó CollisionPolygon2D a esse StaticBody2D.
 - Ele é um pouco diferente do CollisionShape2D, mas irá facilitar a construção do Mapa1.
 - Note que essa forma incluiu 3 botões na tela .
 - Eles servirão para criar, editar e apagar pontos ao polígono.
 - Um polígono estará completo quando ele fechar um desenho, ou seja, quando você ligar o último ao primeiro ponto.



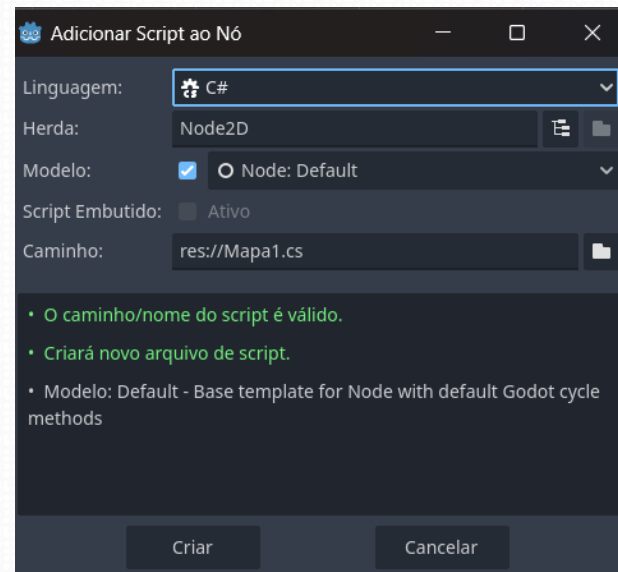
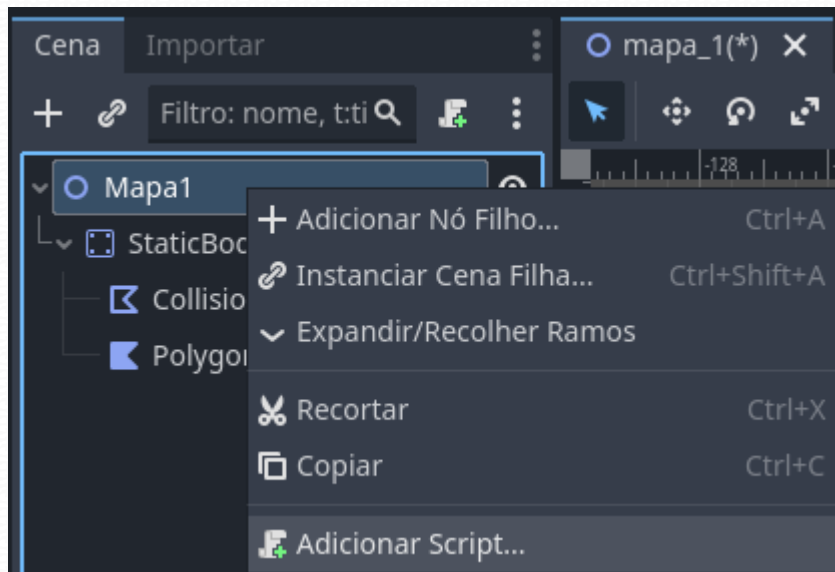
Mapa1

- Habilite o encaixe de grade, para criar linhas de apoio para o desenho do mapa.
- Desenhe seu Mapa1 criando pontos de polígono.
- Não se preocupe com a qualidade do desenho nesse momento, é só um ambiente de testes ainda. Ainda assim segue um exemplo de mapa.
- Note que ao executar a cena, ela não aparece, pois nossa forma não tem uma imagem.



Mapa1

- Adicione um nó Polygon2D ao StaticBody2D para desenharmos o Mapa que coincida com a área de colisão.
- Mas não iremos desenhar por cima da área de colisão, vamos aproveitar esse momento para verificar como acessar objetos na nossa cena por código (Script).
- Adicione um Script ao Mapa1.
 - Lembre-se de criar o código com Linguagem C#.



Mapa1

- Iremos fazer o Polygon2D receber a forma do CollisionPolygon2D em código.
- Código utilizando o **[Export]**
 - Crie variáveis, do tipo necessário, no seu código usando o prefixo [Export] e dê nomes a elas.

5 references

```
[Export] private CollisionPolygon2D colisao_poligono;
```

5 references

```
[Export] private Polygon2D poligono;
```

- Adicione na função _Ready() o comando para copiar o polígono da colisão no da imagem.

```
public override void _Ready() {  
    |   poligono.Polygon = colisao_poligono.Polygon;  
    |  
}
```

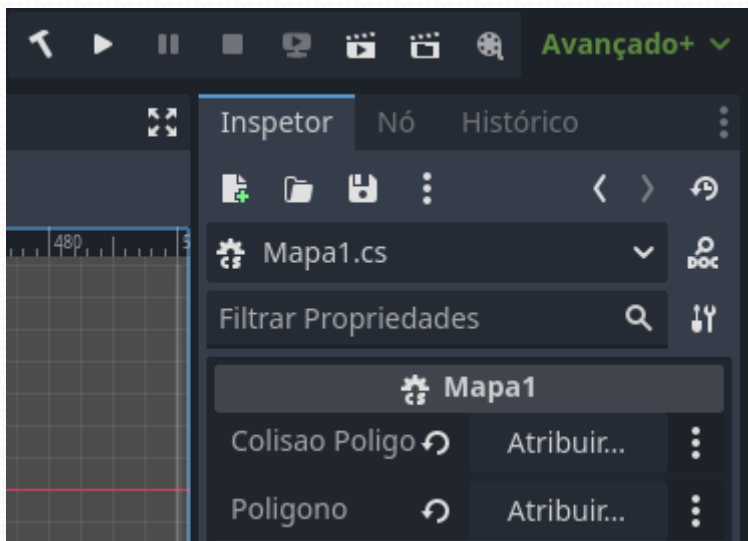


10

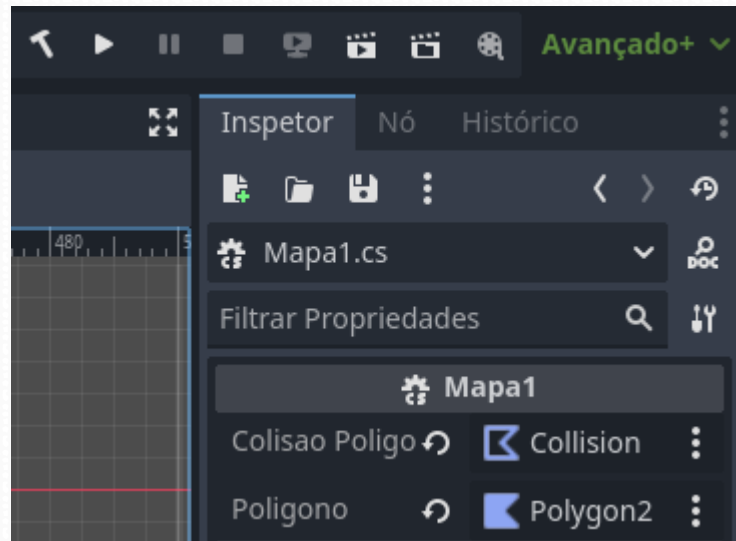
Mapa1

- Iremos fazer o Polygon2D receber a forma do CollisionPolygon2D em código.
- Código utilizando o **[Export]**
 - Compile seu código para as variáveis aparecerem na interface.
 - Utilize a ferramenta de martelo (visível no topo da imagem)
 - No Inspetor irão aparecer as variáveis que você quer exportar.
 - Basta arrastar os nós nas suas respectivas posições.

Antes



Depois



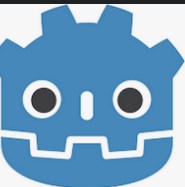
Mapa1

- Iremos fazer o Polygon2D receber a forma do CollisionPolygon2D em código.
- Código utilizando o **GetNode<>()**
 - Crie variáveis, do tipo necessário, no seu código usando o prefixo [Export] e dê nomes a elas.

```
5 references  
private CollisionPolygon2D colisao_poligono;  
5 references  
private Polygon2D poligono;
```

- Adicione na função _Ready() o comando para copiar o polígono da colisão no da imagem.

```
public override void _Ready() {  
    colisao_poligono = GetNode<CollisionPolygon2D>("CollisionPolygon2D");  
    poligono = GetNode<Polygon2D>("Polygon2D");  
    poligono.Polygon = colisao_poligono.Polygon;  
}
```



Mapa1

- Iremos fazer o Polygon2D receber a forma do CollisionPolygon2D em código.
- Código utilizando o **GetNode<>()**
- Desse modo não é necessário fazer o link na interface do Godot, pois isso já foi feito com o comando `GetNode<>()`
- O lado positivo é que não precisamos fazer isso na interface, sendo mais rápido para bons programadores.
- O lado negativo é que teremos que abrir o código para editar o `GetNode` cada vez que os nós mudarem.
- Por isso recomendo utilizar o comando `Export`.



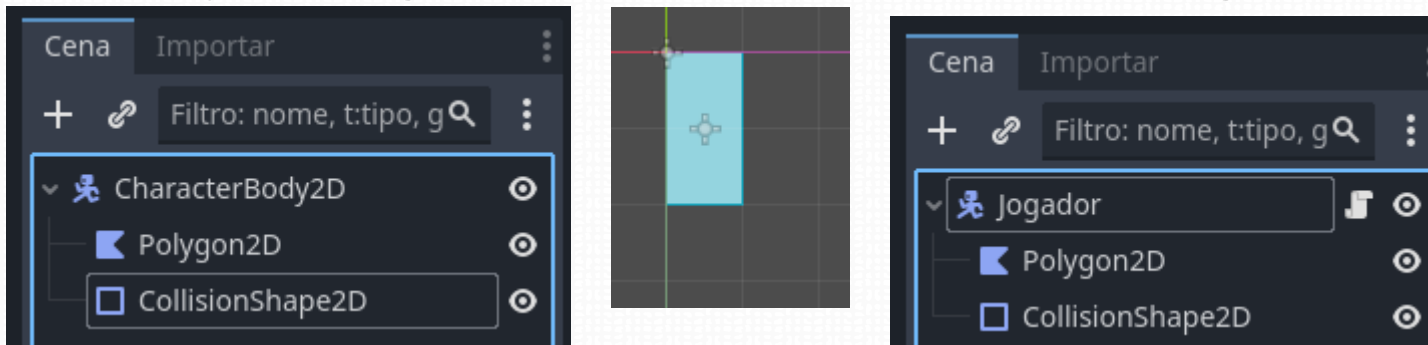
CharacterBody2D

- CharacterBody2D é uma classe especializada para corpos físicos que são controlados pelo usuário.
- Eles não são afetados pela física, mas afetam outros corpos físicos em seu caminho.
- Eles são usados principalmente para fornecer API de alto nível para mover objetos com detecção de parede e inclinação (método `moveAndSlide`), além da detecção geral de colisão (método `MoveAndCollide`).
- Isso o torna útil para corpos físicos altamente configuráveis que devem se mover de maneiras específicas e colidir com o mundo, como costuma acontecer com personagens controlados pelo usuário.



Jogador

- Crie um personagem, em uma nova cena, com a seguinte estrutura.

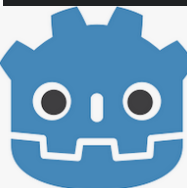


- Altere o nome da cena para Jogador.
- Adicione um Script em linguagem C# ao Jogador.
- Adicione desenhos um polígono de tamanho 2 para ser o jogador.
- Desenhe um forma de colisão retangular do tamanho do polígono.
- Salve a cena.



Mapa1 com Jogador

- Pode-se incluir o Jogador no Mapa1 de duas formas
 - 1. Com a cena Mapa1 selecionada, encontre no FileSystem o arquivo jogador.tscn e arraste o jogador na posição desejada no jogo.
 - 2. Clique com o botão direito no Mapa1 e selecione a opção instanciar cena filha, selecione o jogador, arraste o jogador da posição inicial para a desejada no mapa.
- Execute seu Jogo
 - Note que como estamos trabalhando com uma resolução pequena, temos a impressão que o personagem está extremamente rápido.
 - Se estivéssemos utilizando objetos de tamanho grande, essas mesmas configurações fariam o jogo parecer lento.



Jogador ajustes no Código

- Quando a resolução fica alta demais o jogo aparenta ser lento, mas quando diminuimos demais o tamanho o jogo aparenta ficar muito acelerado.
 - Podemos resolver isso mudando algumas variáveis.
 - Para isso o código do jogador foi facilitado
 - O `[Export]` foi utilizado em algumas variáveis para facilitar seus ajustes na ferramenta.
 - O código foi comentado para facilitar seu entendimento.



[17]

Jogador ajustes no Godot

- Esses passos são para garantir que o Jogador deslize pelas rampas e evitar que ele acelere ao deslizar por elas.
- Selecione a cena do Jogador e vá em
 - Inspetor -> CharacterBody2D -> Floor
- Propriedades de **Floor**
 - **Stop On Shope**: se ativo, o jogador não vai deslizar ao parar em rampas.
 - **Constant Speed**: se ativo, o jogador não vai acelerar ao descer rampas ou subi-las em câmera lenta.
 - **Max Angle**: ângulo máximo que o jogador consegue subir rampas.
 - **Snap Length**: quantidade de pixels que tem que ficam em contato para manter o personagem na rampa. Aumentar o valor diminui a probabilidade do personagem se desprender da rampa.



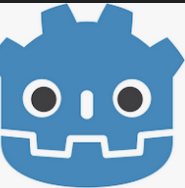
Atividade

- Faça os seguintes ajustes no Jogador.
 - Ajuste para deslizar pelas rampas.
 - Ajuste para subir e descer rampas a uma velocidade constante.
 - Programe o personagem para dar pulos duplos.
 - Crie uma variável para contar se o pulo duplo já foi utilizado.
 - Na função pulo faça pular se ainda não utilizou o pulo duplo ou se está no chão.
 - Restaure o pulo duplo quando o personagem estiver no chão.
 - Faça o personagem acelerar ao se movimentar, ao invés de ele se mover instantaneamente na velocidade máxima.
 - Use a função `Mathf.MoveToward()`
 - É interessante criar uma variável aceleração (`aceleracao`).
 - Multiplique a variável aceleração por delta.



XXX

- XXX



(20)



UNIVALI