Trabalho 6 – Colisões, Spawner e AI

1. Colisão

Temos entidades, temos tiros, mas nada disso serve pra alguma coisa se ninguém explode. Vamos providenciar! ♥

Primeiro, você vai precisar do header Collision.h providenciado no drive da disciplina. Usaremos a função IsColliding, uma implementação do SAT para dois Rects, para saber se dois GameObjects estão colidindo. Mas antes de fazermos isso, precisamos criar mais um componente, o Collider.

Ela serve para que você coloque colisão nos GameObjects que quiser, além de te permitir personalizar onde que será detectada a colisão.

Collider (herda de Component)

`+ Collider (associated : GameObject&,`
`  scale : Vec2 = {1, 1},`
`  offset : Vec2 = {0, 0})`
`+ box : Rect`
 	
`+ Update (dt : float) : void`
`+ Render () : void`
`+ SetScale (scale : Vec2)`
`+ SetOffset (offset : Vec2)`

`- scale : Vec2` `- offset : Vec2`

`> Collider::Collider (associated : GameObject&, scale : Vec2 = {1, 1}, offset : Vec2 = {0, 0})`

Inicialize todas as variáveis, mas pode deixar o box quieto.

`> Collider::Update (dt : float) : void`

A `box` é a caixa onde a colisão será testada. Ela é a `box` do `GameObject` modificado por uma escala e deslocado por um valor orientado à sua

rotação. Ou seja, sete a `box` atual como uma cópia da `box` de `associated` mas com sua largura e altura multiplicados por uma escala. Depois, faça com que o centro dela seja igual ao centro da `box` de `associated`, adicionado do atributo `offset` rotacionado pelo ângulo de `associated`.

```
> Collider::Render () : void
```

Vamos deixar vazio também, mas só por enquanto! :x

```
> Collider::SetScale (scale : Vec2) : void
> Collider::SetOffset (offset : Vec2) : void
```

Você já fez isso tantas vezes que tenho certeza que você sabe o que deve ser feito nessas funções.

Agora vamos usar isso. Vá no construtor de Character, Zombie e Bullet e adicione Collider a eles. Não se preocupe com scale e offset por enquanto.

No Update da sua State, após atualizar os objetos, percorra o vetor testando se cada objeto colide com outro. Mas faça isso somente com aqueles que tiverem o componente Collider.

Na hora de chamar IsColliding, use a box dos Colliders mas a rotação do GameObject. Garanta que cada par de objetos só é testado uma vez! Se houver colisão, notifique ambos os **GameObjects**.

Para que essa notificação seja possível, adicione o seguinte membro em GameObject:

```
+ NotifyCollision (other : GameObject&) : void
```

E o seguinte em Component:

```
+ NotifyCollision (other : GameObject&) : void, virtual
```

A implementação de GameObject é simplesmente percorrer o vetor de componentes notificando-os de que houve colisão.

Component deve ter uma implementação vazia, mas cada objeto que herdar de Component pode ter uma implementação, caso queira, dessa função. Ela define o comportamento que ele deve ter em relação a si mesmo quando seu GameObject colidir com outro GameObject. Por exemplo, se os

Zombies colidirem com uma Bullet, ele deve diminuir o próprio HP e a Bullet deve solicitar sua deleção.

O que nos leva ao nosso próximo problema: O comportamento de um objeto durante uma colisão depende de com quem ele está colidindo. Se uma Bullet colide com Characters ou Zombies, ela some. Se colide com outra Bullet, não. E é aí que entra a sacada de determinar qual GameObject estamos lidando dependendo de quais componentes ele possui. Se ele possui o componente Bullet, ele é um tiro, se não, ele é qualquer outra coisa menos um tiro.

Agora você pode escrever a função NotifyCollision dos objetos que agora possuem Collider. Crie um cooldown para o character tomar dano em contato com Zombies, para evitar que você tome dano todo frame enquanto estiver com overlap na hitbox. Também faça character tocar o som "Hit1.wav" quando toma dano e "Dead.wav" quando morre. Rodando o jogo, você deve reparar que há dois problemas: O primeiro é que as Bullets causam dano ao próprio objeto atirador ao se colidirem com ele, e o segundo é que se o seu personagem morre, o jogo crasha.

O segundo problema é mais fácil de resolver: O que provavelmente está causando o crash é o Update da câmera, que tenta encontrar seu foco, mas ele foi deletado. Se, ao ser notificado de uma colisão com uma Bullet, o personagem ficar com HP menor que zero, devemos chamar Unfollow() na câmera, ou seja, o trecho em Update que lidava com isso pode aproveitar a oportunidade e vir para cá.

Já o friendly fire é um pouco mais complicado de tratar. Usaremos uma solução não muito agradável, mas simples e efetiva. Adicione o seguinte membro à Bullet:

```
+ targetsPlayer : bool
```

O valor dessa flag deve ser dado no construtor de Bullet (sim, está enorme). Com ela, podemos saber se a Bullet acertou o objeto alvo ou não. Se o ponteiro de Character for igual a player, defina como false. Caso contrário, true. Por que isso se nosso personagem é jogável e único?

Veremos no futuro. Em Character, faça com que ele verifique o valor de targetsPlayer, e se true, tome o dano apenas se o ponteiro for igual a player, e caso false, apenas se for diferente.

Nossa detecção de colisões está pronta! Mas está meio estranha. As vezes a bullet atinge "à distância" dos alvos. Se tivéssemos como debugar isso...

Collider::Render ao resgate! No drive você pode encontrar um arquivo que contêm uma implementação para Collider::Render. Copie o conteúdo dela para dentro de sua classe e ajuste o nome das funções e variáveis caso tenha usado nomenclatura diferente da presente no arquivo. Se você compilar tudo novamente em modo debug, você terá uma surpresa! Essa ajuda visual é como informações de debug são criadas em jogos de forma a serem ainda mais úteis. Ajuste a escala e o offset dos colliders a gosto. E se você compilar em modo release, perceberá que essas caixas não estão mais presentes.

Essa é a utilidade dos Renders em componentes não visuais (diferentemente de Sprite) e elas podem ser controladas não só por diretivas de compilação como pelo próprio código. Se você quiser, você pode implementar formas de ativar e desativar essas informações com um simples pressionar de teclas.

Mas ok, tudo está ótimo e funcionando. Nossas entidades levam dano e morrem! Só precisamos agora de uma forma mais natural de criar inimigos do que o próprio jogador apertar spacebar. E de uma AI para eles poderem revidar.

2. WaveSpawner

Wave
+ Wave(zombies : int, cooldown : float) + zombies : int + cooldown : float

WaveSpawner (herda de Component)
+ WaveSpawner(associated : GameObject&) + Update (dt : float) : void + Render () : void
- zombieCounter : int - waves : vector<Wave> - zombieCooldownTimer : Timer - currentWave : int

> WaveSpawner::WaveSpawner (associated : GameObject&)

Crie algumas waves para popular o seu vetor, e inicialize as outras variáveis.

> WaveSpawner::Update (dt : float) : void

Aqui é onde a mágica acontece, comece chamando Update do seu timer. Se zombieCounter for menor do que a quantidade de zombies dessa wave, verifique se o timer está acima do cooldown. Caso sim, crie um Zombie apenas longe o bastante para não ser visível, em uma direção aleatória. Para poder gerar números (pseudo-)aleatórios, seede a função `rand()` no construtor de Game. Use a função `srand()` (`<cstdlib>`) com `time(NULL)` (`<ctime>`) como argumento. Incremente zombieCounter e reinicie o timer. Se zombieCounter já tinha passado da quantidade de zombies para essa wave, mude para a próxima wave, resete zombieCounter e, caso tenha passado da ultima wave, delete o objeto.

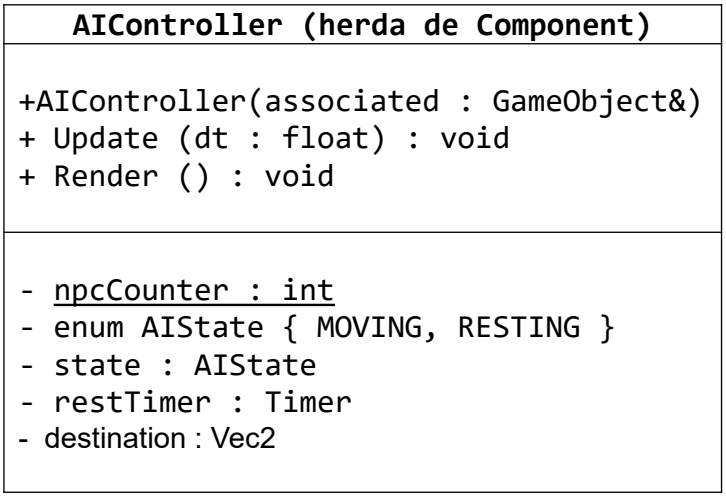
O restante das funções novas são triviais. Agora em Zombie::Update faça com que, caso Character::player seja diferente de nullptr, o Zombie ande em linha reta até o player, com velocidade constante.

Crie um WaveSpawner no seu State, e teste o programa. Já temos um

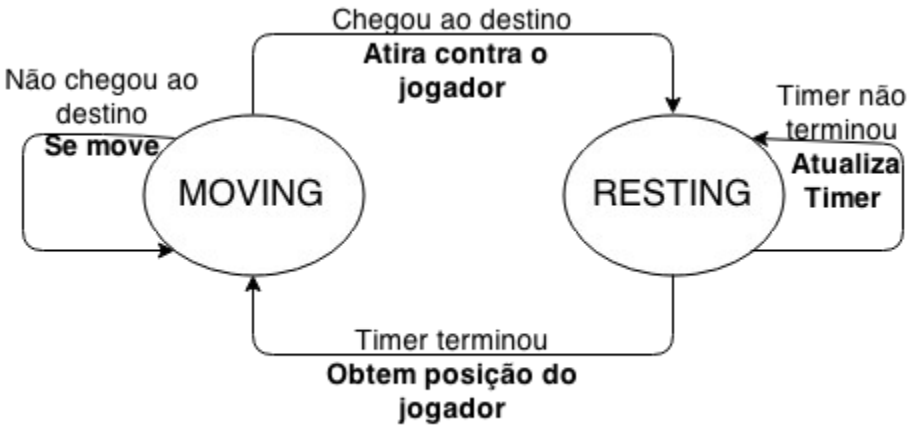
pequeno desafio para o jogador, mas seria ideal se as waves não começarem antes de acabar a atual. Crie em Zombie um contador static. No seu construtor incremente ele, e no destrutor decremente. Agora use esse valor para impedir a wave de mudar até não terem mais inimigos.

Nosso gameplay loop de wave survival está quase pronto, mas ainda falta algo. Em um apocalipse zumbi, não basta se preocupar com os mortos, também tem que competir com os vivos.

3. AIController



O enum nos dá os estados da máquina de estados que vamos implementar. Deve ser privada à classe.



O estado atual fica guardado no membro state, e deve ser, inicialmente, RESTING. Nesse estado, ele dá Update no restTimer, esperando passar o cooldown.

Assim que o cooldown termina, o NPC muda o seu estado para MOVING.

Se o estado do NPC é MOVING, a cada frame ele se move em direção ao jogador. Quando ele chega a essa posição, (ou perto o suficiente), ele obtém a nova posição do jogador e atira em sua direção. O Timer de cooldown é resetado, e o estado volta para RESTING.

Esse comportamento é implementado em AIController::Update. Se o jogador morre, o NPC não faz mais nada.

Faça seu Spawner criar alguns NPCs agora e divirta-se com o jogo. Para não terminarmos o trabalho muito cedo, vamos fazer alguns polimentos.

4. Polish

Primeiro, você deve ter percebido que Zombies mortos ainda dão dano em contato com o jogador e "comem" Bullets. Para resolver isso, use a função GameObjetct::RemoveComponent(), que pedimos para você fazer, mas nunca para usar ainda. Remova o colisor do seu Zombie assim que ele morrer. Falando no colisor, ele parece ter um delay. Isso acontece por que ele atualiza antes dos objetos que causam movimento. Crie seus colisores no start para resolver o problema.

Nossos Zombies também estão sem animação de andar pra esquerda.

Resolva isso. Caso queira fazer sua arma parecer mais uma shotgun(opcional), crie alguns projeteis a mais do lado do original. Só isso já basta por enquanto.

+ Extra (Conta como feature completa opcional na correção) : Nossas waves estão bem simples, apenas espaçando o spawn de cada inimigo igualmente. Se quiser criar waves com curva de dificuldade, o ideal seria as waves começarem mais devagar, e possivelmente depois criar vários inimigos ao mesmo tempo. Lembra de quando fizemos uma fila de comandos para o nosso character? Crie agora uma fila de comandos para o nosso wave spawner, podendo criar um novo Zombie, NPC ou esperar uma certa quantidade de tempo. Mude as waves para serem presets de filas, e crie umas waves mais interessantes com curva de dificuldade a gosto.