Trabalho 5 – Objetos em Movimento, Filas e Rotação

Nos construtores dos componentes que criaremos nesse trabalho nós iremos setar tudo relacionado ao próprio `GameObject`. Entretanto, haverão entidades que necessitarão da existência de outros `GameObjects` (e manter referências a elas). A criação deles não será em seus construtores, mas sim numa nova etapa de execução ao loop do jogo, o Start. E para se manter a referência de forma segura, teremos que deixar `unique_ptr` para trás e mudarmos para `shared_ptr` em alguns lugares.

A etapa Start acontece somente uma vez, que é quando a fase vai ser iniciada pela primeira vez. Vamos começar a fazer as alterações?

1. Criando Starts e mudando ponteiros

Em State vamos fazer as seguintes modificações e acréscimos.

State

`+ Start() : void`
`+ AddObject(go : GameObject*) : std::weak_ptr< GameObject >`
`+ GetObjectPtr(go : GameObject*) : std::weak_ptr< GameObject >`

`- started : bool` `- objectArray : std::vector< std::shared_ptr< GameObject > >`

Primeiro inicialize `started` com false no construtor.

Em `State::Start` você deve chamar `LoadAssets` e depois deve percorrer o `objectArray` chamando o `Start` de todos eles. Ao final, coloque `true` em `started`.

Em `State::AddObject`, ao invés de simplesmente colocar o `GameObject` passado no vetor, você vai criar um `std:shared_ptr< GameObject >` passando esse `GameObject*` como argumento de seu construtor. Depois faça um `push_back` desse `shared_ptr` em `objectArray`. Se `started` já tiver sido chamado, chame o start desse `GameObject`. E retorne um `std::weak_ptr < GameObject >` construído usando o `shared_ptr` criado.

Em `State::GetObjectPtr`, você vai percorrer o vetor de objetos que temos comparando o endereço armazenado em cada `std::shared_ptr` com o

passado como argumento. Crie e retorne um `std::weak_ptr` a partir do `std::shared_ptr` quando os endereços forem iguais. Retorne um `std:weak_ptr` vazio caso não encontre. Essa função é geralmente usada para se obter o `weak_ptr` de algum objeto que já temos o ponteiro puro dele e que já foi adicionado ao vetor de objetos.

Em `Game::Run`, chame o `Start` do `State` logo antes do `while`.

Mudanças em `GameObject`:

GameObject
`+ Start() : void`
`+ started : bool`

Em `GameObject`, você fará o mesmo que `State`. Inicializar `started` com `false` no construtor; no `Start` percorrer os componentes chamando o `Start` deles, setando `started`; e depois chamando o `Start` dos componentes adicionados em `AddComponent` quando `Start` já tiver sido chamado.

E, por último, em `Component`, adicione o método `+ Start() : virtual void` e deixe o corpo vazio em sua implementação.

2. Character: Personagem Jogável

Character (herda de Component)
+ Character (associated : GameObject&, sprite : std::string) + ~Character () + Start() : void + Update (dt : float) : void + Render () : void + Is (type : std::string) : bool + Command : class (ver abaixo) + Issue (task : Command) : void + player : Character*
- gun : std::weak_ptr< GameObject > - taskQueue : std::queue<Command> - speed : Vec2 - linearSpeed : float - hp : int - deathTimer : Timer

Gun (herda de Component)
+ Gun (GameObject& associated, Std::weak_ptr< GameObject > character) + Update (dt : float) : void + Render () : void + Is (type : std::string) : bool + Shoot (target : Vec2) : void
- shotSound : Sound - reloadSound : Sound - cooldown : int - cdTimer : Timer - character : std::weak_ptr< GameObject > - angle : float

Command (classe publica em Character)
+ Command (type : CommandType, x : float, y : float)
+ CommandType : enum (constantes MOVE e SHOOT) + type : ActionType + pos : Vec2

Finalmente, chegamos ao nosso protagonista. Essas classes farão um objeto composto por um par de objetos: Um é o personagem, que anda pelo mapa (Character), e o outro é a arma que ele usa para se defender (Gun). Também temos a classe Command, membro de character, que irá criar uma interface entre o personagem e as controladoras no futuro.

Uma peculiaridade: na classe Character, temos um mecanismo para encontrar o objeto de qualquer lugar. Mantemos um ponteiro da instância de um dos personagens principais para que os inimigos e o estado do jogo possam achá-los e reagir a eles mais facilmente.

```
> Character::Character (associated : GameObject&, sprite :
std::string)
```

Inicialize todas as variáveis. Além disso, adicione a SpriteRenderer, AnimationSetter e afins. Use a imagem encontrada no path dado nos argumentos para inicializar o SpriteRenderer. Os sprites para Character devem ter animação “idle”, “walking” e “dead”.

```
> Gun::Gun (associated : GameObject&,
                                character : std::weak_ptr<
                                                    GameObject >)
```

Inicialize todas as variáveis. Além disso, adicione a SpriteRenderer, AnimationSetter e afins. Use a imagem “Recursos/img/Gun.png”. Chame as animações de “idle” e “reloading”. Também inicialize shotSound e reloadSound com “Recursos/audio/Range.wav” e “Recursos/audio/PumpAction.mp3” respectivamente.

`> Character::~Character ()`

Caso seja o player, ~Character precisa setar a variável de instância como nullptr, para que outras entidades saibam que o objeto foi deletado.

`> Character::Start ()`

Crie a `Gun`, adicione ao estado atual do jogo e ao gun(weak ptr membro).

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

Há algumas etapas para o comportamento de Character: Primeiro, devemos executar ações pendentes. Checamos se há pelo menos uma ação na fila. Enquanto houver, checamos o tipo. Para ações de movimento, devemos calcular velocidades nos eixos x e y de forma que o Character se mova em linha reta na direção correta, e que o módulo da velocidade dele seja sempre constante. Caso a ação seja de tiro, apenas chame Shoot() de Gun. Depois de efetuada uma ação, tire ela da fila. Caso esteja vivo e tenha se movido, defina a animação para walking. Caso não esteja se movendo, idle. Se a vida dele ficar menor ou igual a 0 e já tiver passado a animação de morte por um tempo, deletamos o objeto.

Como estamos definindo a animação todo frame, o personagem irá ficar parado no primeiro frame de cada animação. Vá em AnimationSetter e inclua um novo membro std::string chamado current. Em SetAnimation, verifique se a animação atual é diferente da nova, e caso sim, mude a animação atual.

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

Antes, vamos verificar se o Character ainda existe. Porque, se ele foi deletado, devemos deixar de existir também. Em seguinte, vamos fazer com que o centro da box dele seja igual ao centro da box do corpo. Depois, temos que ajustar a arma. Reposicione ela alguma distância à frente do corpo, na direção apontada pelo angulo. Além disso, queremos que a arma passe uma certa quantidade de tempo parada depois de atirar. Também queremos uma animação de recarregar, sincronizada com o som. Use o timer e cooldown para isso.

`> Render () : void`

Deixe vazio para ambas as classes.

`> Is (type : std::string) : bool`

Você já entendeu como fazer essa não é mesmo?

`> Gun::Shoot (Vec2 target) : void`

Se cooldown for 0, calcule o ângulo para que a arma aponte para a posição do alvo. Depois, toque shotSound, e inicie o processo de cooldown. Iremos criar um projétil aqui também, porém ainda temos que fazer essa classe.

`> Character::Issue (Command task) : void`

Coloque a tarefa na fila.

Em State::State(), instancie Character em 1280,1280 (mais ou menos o centro do mapa), com o foco da câmera nele. Teste o jogo agora. Tente fazer alguma coisa. Não consegue. Nosso personagem é um verdadeiro zumbi.

3. PlayerController

PlayerController (herda de Component)
+ PlayerController (associated : GameObject&) + Start() : void + Update (dt : float) : void + Render () : void + Is (type : std::string) : bool

Vamos dar um pouco de vida ao nosso personagem. Como temos uma interface de comandos, só precisamos de um componente que possa dar esses comandos ao nosso amiguinho. Entra a PlayerController.

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

Verifique as teclas w, a, s e d do teclado. Quando estas forem

pressionadas, devemos mandar um comando de movimentação para o nosso Character. Na posição, mande um vetor de direção e sentido desejado. Quando o mouse for clicado, mande um comando de atirar com a posição do mouse. (Dica: use a função GetComponent())

O resto da classe é trivial de implementar. Vá para State() e adicione essa controladora ao nosso objeto. Você pode andar, sua arma finge que atira, já é um começo. Precisamos de um projétil.

4. Bullet: Projétil Genérico

Bullet (herda de Component)
+ Bullet (associated : GameObject&, angle : float, speed : float, damage : int, maxDistance : float) + Update (dt : float) : void + Render () : void + Is (type : std::string) : bool + GetDamage () : int
- speed : Vec2 - distanceLeft : float - damage : int

Bullet é um projétil que segue em linha reta após sua criação. Ele recebe vários parâmetros que a entidade atiradora determina, incluindo uma direção (angle), um módulo de velocidade (speed), o dano que ela vai causar e uma distância máxima a ser percorrida antes que o projétil "expire" (para que ele não ande pelo mundo infinitamente).

```
> Bullet (associated : GameObject&, angle : float, speed : float,
          maxDistance : float)
```

Inicialize Component. Depois crie e adicione a SpriteRenderer com "Recursos/img/Bullet.png". Daí, como Bullet tem a velocidade constante, calcule o vetor velocidade, pois este será usado em todo frame durante a

vida do objeto. Lembre-se também de setar a distância remanescente de acordo com o parâmetro dado.

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

Para cada Update, a Bullet deve se mover speed * dt, e devemos subtrair essa mesma distância da distância remanescente. E solicitar a deleção caso essa distância seja menor ou igual a zero.

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

Não faz nada.

```
> Is (type : std::string) : bool
```

Is retorna true se type for “Bullet”.

```
> GetDamage () : int
```

Retorna o dano que essa Bullet vai causar na colisão.

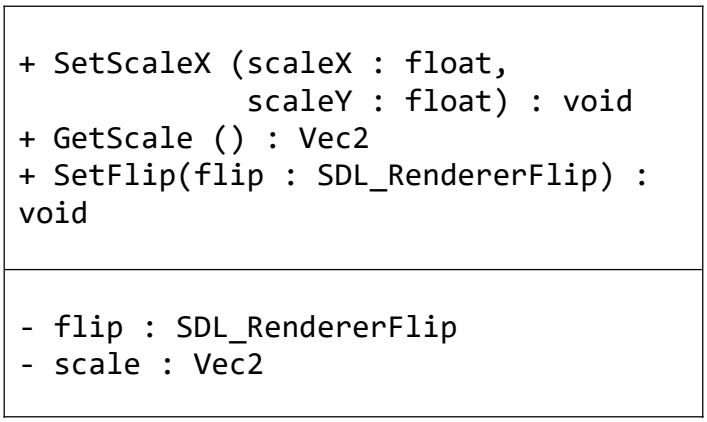
Temos um projétil pronto. Agora voltemos à Gun:

```
> Shoot (target : Vec2) : void
```

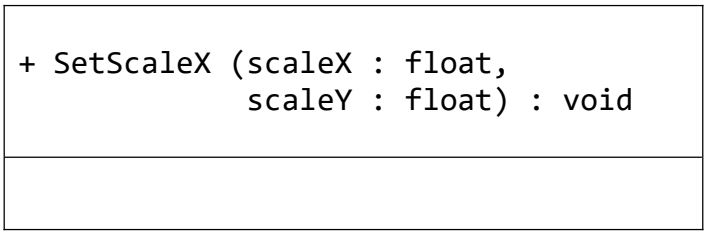
Precisamos construir uma Bullet e acrescentá-la ao vetor de objetos. Para isso, o primeiro passo é calcular a direção (ângulo) que queremos que o projétil siga. O resto dos argumentos são constantes arbitrárias. O projétil deve partir da posição do cano da arma. Se tudo estiver certo, a Bullet vai se mover na direção certa… Mas apontando para cima. Além disso, nosso personagem e arma estão sempre alinhados horizontalmente… hmm.

5. Sprites com Zoom e Rotação

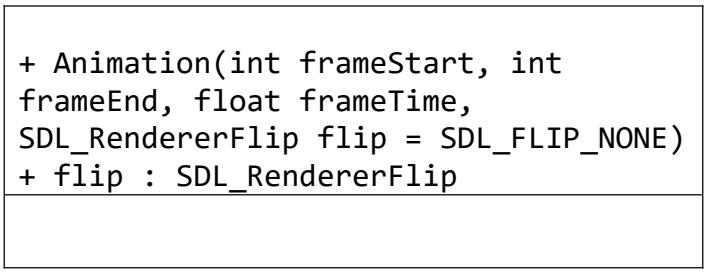
Fizemos um objeto que gira em torno de outro, mas algo ainda mais interessante é podermos girar objetos em torno de si mesmos. Além disso, é comum querermos manipular a escala de objetos in-game, sem alterar seu sprite, e espelhar nossas animações. Nossa engine não faz nada disso: vamos implementar! Adicione os seguintes membros em Sprite:



Os seguintes membros a SpriteRenderer:



E os seguintes membros a Animation:



Inicialize as escalas como 1 no construtor de Sprite, e flip como SDL_FLIP_NONE para não quebrar os Sprites já no programa. Além disso, ajuste Sprite::GetWidth() e Sprite::GetHeight() para retornarem a dimensão ajustada para a respectiva escala. Também acrescente float angle = 0 nos parâmetros de render, para o nosso SpriteRenderer passar o ângulo do seu objeto.

A função SetScale em Sprite é apenas um método Set para a escala. Mantenha a escala em dado eixo se o valor passado para ela for 0. A função SetFlip é igualmente trivial.

No SpriteRenderer não se esqueça de atualizar a box do GameObject associated. Para facilitar no futuro, mova a box dele de forma a manter o centro no mesmo lugar de antes da mudança de escala.

Adicione o flip em Animation, e faça as alterações necessárias no seu construtor. Também chame SetFlip em SetAnimation() no SpriteRenderer.

Agora adicione `+ angleDeg : double` ao GameObject e inicialize-o com 0 no construtor.

Agora precisamos ajustar a renderização. Um pequeno parêntese: Rotação e escala, na SDL 1.2, eram tarefas feitas por software por uma biblioteca um pouco temperamental, e davam brecha para vários bugs e memory leaks. Na SDL2, mudaremos três linhas do corpo de Sprite::Render para fazer o Sprite ser renderizado com zoom e/ou rotacionado.

Para o zoom, você deve ajustar para a escala as dimensões do retângulo de destino (quarto argumento da SDL_RenderCopy). O tamanho do Sprite será ajustado automaticamente para ocupar o novo retângulo.

Para a rotação e espelhamento, vamos substituir a SDL_RenderCopy pela SDL_RenderCopyEx. Ela recebe sete argumentos, sendo os quatro primeiros os mesmos da RenderCopy. Os três outros são:

- `angle : double` - Ângulo de rotação no **sentido horário** em **graus**.
- `center : SDL_Point*` - Determina o eixo em torno da qual a rotação ocorre. Se passarmos nullptr, a rotação ocorre em torno do centro do retângulo de destino, que é o que queremos.
- `flip : SDL_RendererFlip` - Inverte a imagem verticalmente (SDL_FLIP_VERTICAL), horizontalmente (SDL_FLIP_HORIZONTAL), ambos (bitwise or), ou não inverte (SDL_FLIP_NONE).

Pronto! Com isso, basta setar as escalas e rotações em seus objetos.

Para este trabalho, faça:

1. Bullet e Gun inclinadas no ângulo certo.

2. Animações de andar pra esquerda com Character.
3. Reoriente a arma caso ela esteja de cabeça pra baixo.

Ande pelo cenário, crie alguns Zombies, dê oi pra eles, e tente atirar. Nada acontece.

Quer saber? Resolvemos isso depois. Por hoje, é só.

+ Extra (+0,5 ponto) : Se você andar em cima de um Zombie perceberá que ele sempre aparece por cima do jogador. Isso ocorre por que estamos renderizando tudo na ordem de criação dos objetos. Uma forma de resolver isso é implementado um algoritmo de Sorting. Implemente um Z/Y sorting para a sua engine.