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`.

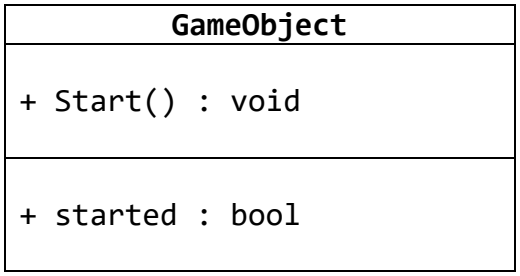
Em `State::AddObject`, ao invés de simplesmente colocar o `GameObject` passado no vetor, você vai criar um `std:shared_ptr< GameObject >` passando esse `GameObject*` como argumento de seu construtor. Depois faça um `push_back` desse `shared_ptr` em `objectArray`. Se `started` já tiver sido chamado, chame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



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.

Agora retire as Faces do seu trabalho. A partir daqui, introduziremos entidades definitivas do nosso jogo.

2. Alien: Seguindo os Cliques

Alien (herda de Component)
+ Alien (associated : GameObject&, nMinions : int) + ~Alien () + Start() : void + Update (dt : float) : void + Render () : void + Is (type : std::string) : bool
- Action : class (ver abaixo) - speed : Vec2 - hp : int - taskQueue : std::queue<Action> - minionArray : std::vector< std::weak_ptr<GameObject> >

Action (classe privada em Alien)
+ Action (type : ActionType, x : float, y : float)
+ ActionType : enum (constantes MOVE e SHOOT) + type : ActionType + pos : Vec2

Alien é um objeto controlado pelo mouse. Ele executa uma sequência de ações, uma a uma, contidas numa fila. Quando um dos botões do mouse é pressionado em determinado ponto da tela, o Alien cria uma ação e a enfileira. Além disso, ele mantêm um array de ponteiros para GameObjects (serão os Minions), os quais ele usará para atirar.

Uma Action, por sua vez, é dada por um tipo (mover ou atirar), e uma posição para onde o Alien deve se deslocar, ou onde ele deve atirar. O segundo caso, o Alien atirar, ficará pendente. Essa classe só é visível para Alien.

Ele deve herdar de Component pois ela implementa um comportamento específico desejado de um GameObject. GameObject foi projetado para **nunca** ser herdado, todo comportamento desejado de um GO deve ser implementado através de Componentes adicionados a ele. Essa é a característica fundamental desse padrão de projeto.

```
> Alien (associated : GameObject&, nMinions : int)
```

Adiciona um componente do tipo Sprite ao associated e inicializa as outras variáveis. Mover a adição dos componentes de dependência (e de GameObjects também, como veremos abaixo) para o construtor reduz o tamanho e simplifica o construtor do state mas possui, na nossa engine, uma desvantagem. Consegue descobrir qual é?

```
> Start () : void
```

Devemos popular o array de Minions com alguns destes objetos, espaçados igualmente. Enquanto não tiver certeza que o Alien funciona como desejado, não faça nada aqui.

```
> ~Alien ()
```

Devemos esvaziar o array com os minions.

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

Há algumas etapas para o comportamento de Alien: Primeiro, checamos se houve input que gere uma ação: clique do botão esquerdo do mouse para um tiro, ou direito para movimento. Se sim, enfileiramos uma ação com a posição do clique - lembre-se do ajuste da câmera.

Feito isso, devemos executar ações pendentes. Checamos se há pelo menos uma ação na fila. Se houver, checamos o tipo. Para ações de movimento, devemos calcular velocidades nos eixos x e y de forma que o Alien se mova em linha reta até aquele ponto, e que o módulo da velocidade dele seja sempre constante.

Isso tudo é caso o Alien não vá chegar ao destino no próximo frame. Se ele estiver a essa distância mínima da posição, devemos colocá-lo lá

imediatamente e dar a ação como realizada (tirar da fila). Há dois motivos para isso. Primeiro, se deixarmos o Alien andar na velocidade total, ele vai passar do ponto, e tentar voltar no frame seguinte, no qual ele vai passar de novo.

Ele provavelmente não vai se estabilizar, o que nos leva ao segundo motivo: Trabalhando com floats, nunca usamos '==’. É muito improvável que dois floats que calculamos sejam iguais. Em vez disso, usamos '>' e/ou '<' para estabelecer ranges aceitos. Novamente, o movimento se encerra se a distância for menor do que o que o Alien vai mover no frame.

Caso a ação seja de tiro... por enquanto, apenas tire a ação da fila. Precisamos implementar mais algumas coisas antes.

Devemos pedir para remover esse GameObject se a vida dele ficar menor ou igual a 0.

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

Vazio.

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

Retorna verdadeiro se o tipo passado é o mesmo desse componente ("Alien").

No construtor de State, crie um Alien (criar GO e adicionar componente Alien) em 512,300 (sete o centro da box para essa posição) e coloque-o no vetor de objetos. Teste se ele se move corretamente, independentemente da posição da câmera. Se sim, vamos ao...

3. Minion: Objeto em Órbita

Minion (herda de Component)
+ Minion (associated : GameObject&, alienCenter : std::weak_ptr< GameObject >, arcOffsetDeg : float = 0) + Update (dt : float) : void + Render () : void + Is (type : std::string) : bool + Shoot (target : Vec2) : void
- alienCenter : GameObject& - arc : float

Nem todo objeto do nosso jogo pode ter seu movimento controlado: Alguns são guiados por AI e tomam decisões em um contexto, e outros seguem regras simples indefinidamente, definidas no seu Update. Minion é o segundo caso, orbitando um GameObject dado até ser deletado.

Além do seu Sprite, Minion recebe um ponteiro para um objeto (o centro de sua órbita) e um ângulo (arc) que indica o arco da circunferência (a órbita si) já percorrida. Usando esses dois, podemos atualizar a box do Minion.

```
> Minion (associated : GameObject&,
          alienCenter : std::weak_ptr< GameObject >,
          arcOffsetDeg : float = 0)
```

Os argumentos do construtor (além do obrigatório por ser componente) são o objeto que devemos orbitar e o ângulo inicial em graus. Adicionamos o componente Sprite (igual fizemos para Alien) e calculamos o primeiro valor da box (explicamos como a seguir).

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

Como Minion anda em círculo, não em linha reta, não usamos uma velocidade com coordenadas x e y. Em vez disso, usamos uma velocidade

angular (radianos/segundo), e incrementamos o arco a cada frame. Defina uma constante para isso, de preferência, uma fração de pi.

Calcular a posição do Minion é uma tarefa bem mais fácil do que parece. Começamos assumindo que o Minion ficará na origem do mapa. Crie um vetor que represente a distância da origem que o Minion começará caso seu offset inicial seja zero (por exemplo {x = 200, y = 0}). Depois, o rotacionamos pelo seu arco atual (arc). Por último, somamos com a posição atual do centro de sua rotação, ou seja, o centro do box de alienCenter. Essa é a posição que **o centro do Minion** deve assumir.

Visto que `alienCenter` é um `std::weak_ptr`, devemos dar `lock()` nele para obtermos um `shared_ptr` que podemos usar para usarmos sua box. Se o que retornou for um `shared_ptr` for vazio, nosso centro não existe mais, então devemos pedir a deleção e encerrar o Update agora mesmo.

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

Deixe vazio.

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

O mesmo para o de seu pai, mas agora compare com o tipo desse componente.

Popule o vetor de Minions do Alien em Alien::Start (chame State::AddObject com o go criado e o retorno dessa função coloque em minionArray) e observe o movimento. Tente mover pelo cenário e veja se eles acompanham o Alien corretamente.

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

Shoot recebe uma posição e dispara um projétil naquela direção.

...espera, que projétil?

4. Bullet: Projétil Genérico

Bullet (herda de Component)
+ Bullet (associated : GameObject&, angle : float, speed : float, damage : int, maxDistance : float, sprite : std::string) + 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, um caminho para seu Sprite, 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, sprite : std::string)
```

Inicialize Component. Depois crie e adicione a sprite. 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 ao Minion:

`> Shoot (pos : 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 sprite é *img/minionbullet1.png*. O projétil deve partir da posição do próprio Minion.

De volta em Alien:

Se a ação a ser tratada é Shoot, escolha um Minion aleatório e mande-o disparar contra a posição dada (não se esqueça de dar `lock()`, estamos lidando com `weak_ptr`). Se tudo estiver certo, a Bullet vai se mover na direção certa... Mas apontando para a direita.

Qual o seu problema, dona Bullet?

5. Sprites com Zoom e Rotação

Fizemos um objeto que gira em torno de outro, Minion, 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. Nossa engine não faz nada disso: vamos implementar! Adicione os seguintes membros em Sprite:

+ SetScaleX (scaleX : float, scaleY : float) : void + GetScale () : Vec2
- scale : Vec2

Inicialize as escalas como 1 no construtor de Sprite e o ângulo como 0, 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.

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

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.

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, 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). Você pode implementar suporte à inversão de Sprites se quiser, mas por enquanto, use SDL_FLIP_NONE.

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

  Para este trabalho, faça:

1. Minions sempre com a parte de baixo do Sprite virada para o Alien;
2. Alien girando lentamente na direção contrária à translação dos Minions;
3. Minions com escala aleatória entre 1 e 1.5.
4. Tiro orientado na direção de seu movimento.

E por hoje, é só. Lembre-se de mover a câmera para testar o trabalho completo.

+ Extra (+0,5 ponto) : Faça com que o Minion mais próximo da posição onde o clique ocorreu seja escolhido para atirar.