Trabalho 6 – Animações, Movimento Acelerado e Colisão

Um grupo de alienigenas tenta invadir o planeta, mas perde o controle de suas naves e cai acidentalmente em um iceberg. Um par de pinguins deve enfrentar a ameaça do espaço!

1. Sprites Animados

Até agora, nossos Sprites só podem exibir imagens estáticas. Dificilmente um jogo se faz só com estas, no entanto: São necessárias animações para dar mais "vida" ao mundo do jogo. Alteraremos a classe Sprite para permitir o uso das mesmas.

Todos devem estar familiarizados com como uma animação funciona: Uma sequência de diferentes quadros é mostrada rapidamente, criando a ilusão de movimento. Em jogos 2D, fazemos exatamente a mesma coisa. Um exemplo é a sprite sheet a seguir, de Sonic The Hedgehog (Mega Drive).

A idéia é clipar e mostrar cada quadro da animação por um determinado tempo na tela. Quando chegamos ao fim da sheet, voltamos ao primeiro quadro e a animação se repete.

Precisamos saber antecipadamente quantos frames há na imagem e por quanto tempo cada frame deve ser mostrado. Pelo número de frames, sabemos a largura de cada quadro, que também nos dá o offset em x de um frame para outro.

Adicione os seguintes membros em Sprite:

```
+ Sprite (file : string,
          frameCount : int = 1,
          frameTime : float = 1)***
+ Update (dt : float) : void
+ SetFrame (frame : int) : void
+ SetFrameCount (frameCount : int) : void
+ SetFrameTime (frameTime : float) : void

- frameCount : int
- currentFrame : int
- timeElapsed : float
- frameTime : float
```

***Adapte o construtor pre-existente. Lembre-se de inicializar os parâmetros novos no construtor padrão também!

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

Update deve acumular os dts em timeElapsed. Se timeElapsed for maior que o tempo de um frame, passamos para o frame seguinte, setando o clip. Se o frame atual ultrapassar os limites da imagem, voltamos para o primeiro.

```
> SetFrame (frame : int) : void
```

Usado para escolher manualmente um frame. Deve setar o frame atual e o clip da imagem.

```
> SetFrameCount (frameCount : int) : void
> SetFrameTime (frameTime : float) : void
```

Setam os respectivos membros. Usadas para Sprites criados com o construtor padrão, ou, no caso de frameTime, para alterar a velocidade da animação.

Outras mudanças incluem:

```
> Open (file : string) : void
```

Open agora deve setar o clipe de acordo com o número de frames. Você pode também acrescentar os argumentos frameTime e frameCount nessa função (para o caso de Sprites criados sem imagem inicialmente), mas tendo as funções SetFrameTime e SetFrameCount, isso é opcional.

```
> GetWidth () : int
```

GetWidth agora deve retornar a largura de apenas um dos frames.

Se você tiver executado as mudanças corretamente, os Sprites pré-existentes no seu trabalho devem funcionar da mesma forma de antes. São, afinal, "animações" de um frame só e cujo update não é chamado. O primeiro objeto animável do nosso trabalho seria a Bullet.

No construtor de Bullet, adicione os parâmetros frameTime e frameCount, e use-os para criar o Sprite. Em Bullet::Update, atualize o Sprite. Em Minion, troque o Sprite passado para *img/minionBullet2.png*, que tem mais de um frame, e forneça os parâmetros adicionais.

Para ver se a animação está funcionando de forma adequada, pode ser uma boa idéia comentar o movimento da Bullet, para poder examiná-la com mais cuidado, e/ou aumentar a escala. Se tudo estiver bem, vamos aos...

2. Penguins: Movimento Acelerado

Penguins (herda de GameObject)
+ Penguins (x : float, y : float) + ~Penguins () + Update (dt : float) : void + Render () : void + IsDead () : bool + Shoot () : void + player : Penguins*
- bodySp : Sprite - cannonSp : Sprite - speed : Vec2 - linearSpeed : float - cannonAngle : float - hp : int

Finalmente, chegamos aos nossos protagonistas. Penguins é um objeto composto por um par de pinguins, e ele contem algumas peculiaridades em relação aos objetos que já criamos. Por exemplo, ele tem dois Sprites: Um é o pinguim de baixo, que desliza sobre o gelo (bodySp), e o outro é pinguim controlando o canhão montado em cima (cannonSp).

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

Controlaremos o movimento dos Penguins usando as teclas W e S para acelerar para a frente e para trás, e A e D para virar. Para guiar o canhão e atirar, usaremos o mouse.

`> Penguins (x : float, y : float)`

Inicialize todas as variáveis, inclusive aquelas herdadas de GameObject. Além disso, abra os dois Sprites, e inicialize a variável da instância com `this`. A box deve usar as dimensões do pinguim de baixo.

`> ~Penguins ()`

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

`> Update(dt : float) : void`

Quando apertamos W ou S, os Penguins não se movem diretamente. Em vez disso, aplicamos uma aceleração constante, que por sua vez implicará num aumento ou diminuição da velocidade. Deve haver limites positivos e negativos impostos para essa velocidade.

A direção para a qual o pinguim de baixo está apontado é alterada pelo pressionamento das teclas A ou D, que aplicam uma velocidade angular constante nos pinguins (a rotação não é acelerada). Sabendo essa direção, a velocidade linear, e o dt, você pode calcular a posição.

Ainda temos que ajustar o canhão. O canhão deve seguir a posição atual do mouse. Para saber o ângulo, use o centro da box, que, como veremos em seguida, coincidirá com o eixo do canhão, e forme uma reta. O canhão deve ter o mesmo ângulo dessa reta.

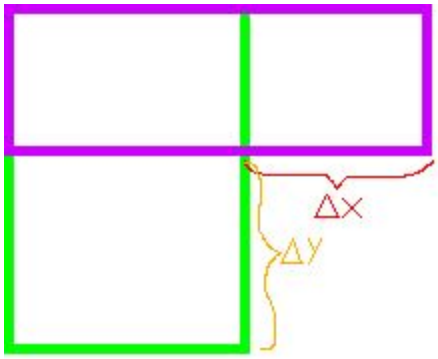
Finalmente, se o botão esquerdo do mouse for pressionado, devemos atirar.

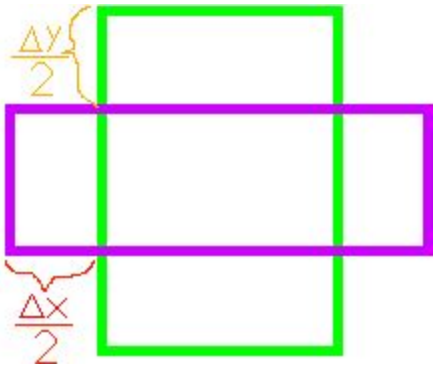
Obs.: É bem possível, usando rotação de vetores, não usar a variável linearSpeed. Ela só serve para simplificar o trabalho, especialmente considerando que os pinguins podem andar de ré.

`> Render() : void`

O pinguim de baixo deve ser renderizado como qualquer outro objeto: box, rotação e câmera. Já o canhão, como dito acima, deve ser posicionado de forma que o centro do seu sprite esteja na mesma posição do centro do corpo.

Isso é mais fácil do que parece. Como a rotação do Sprite é feita a partir do centro, você só precisa pensar em como deve ser posicionado o sprite do canhão na orientação padrão, e a SDL cuidará do resto.





`> IsDead() : bool`

Assim como Alien, os Penguins morrem se seu HP chegou a 0.

`> Shoot() : void`

Esta função é similar à de mesmo nome em Minion, com apenas duas diferenças. A primeira é que não precisamos de um ponto para calcular o ângulo do tiro: já fizemos isso em Update para obter o ângulo do canhão. A segunda diferença é que, como queremos que o tiro saia da ponta do canhão, precisamos estabelecer uma distância do centro dos Penguins onde será colocada a Bullet.

Em State, instancie Penguins em 704,640 (mais ou menos o centro do mapa), com o foco da câmera nele. Ande pelo cenário, dê oi pro Alien, e tente atirar nele. Nada acontece.

3. 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 que mostramos e explicamos em sala. Usaremos a função IsColliding, uma implementação do SAT para dois Rects, para saber se dois GameObjects estão colidindo. No Update da sua State, após atualizar os objetos, percorra o vetor testando se cada objeto colide com outro. Garanta que cada par de objetos só é testado uma vez! Se houver colisão, notifique ambos os objetos.

Para que essa notificação seja possível, adicione o seguinte membro

em GameObject:

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

Cada objeto que herdar de GameObject deve ter uma implementação dessa função. Ela define o comportamento que ele deve ter em relação a si mesmo quando colidir com outro objeto. Por exemplo, se os pinguins colidirem com uma Bullet, ele deve diminuir o próprio HP, e a Bullet deve se marcar como morta (para desaparecer).

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 um Penguins ou Alien, ela some. Se colide com outra Bullet, não. Precisamos de um mecanismo de identificação nos GameObjects. Adicione esta função à classe-mãe:

```
+ Is (type : string) : bool, virtual pura
```

Is recebe um identificador (no nosso caso, usaremos um nome de classe) e compara com o seu próprio. Assim, podemos testar se um objeto tem um tipo que nos interessa.

Essa função também tem a vantagem de que podemos usá-la para apontar objetos que herdam de uma mesma classe. Não há nenhum exemplo no nosso trabalho agora, mas suponha que Alien, Penguins e mais algumas classes herdassem de “Being”, e você quisesse tratar uma interação com um Being, independente de qual fosse, mas sem perder a capacidade de diferenciar suas classes individualmente.

Bastaria escrever sua Is da seguinte forma:

```
bool Penguins::Is(string type) {
      return (Being::Is(type) || type == “Penguins”);
}
```

Agora você pode escrever a função NotifyCollision de seus objetos. Rodando o jogo, você deve reparar que há dois problemas: O primeiro é que as Bullets causam dano ao próprio objeto atirador 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. Lembra que mantivemos uma variável de instância em Penguins? Se ao ser notificado de uma colisão com uma Bullet, os pinguins ficarem com HP menor que zero, devemos chamar Unfollow() na câmera.

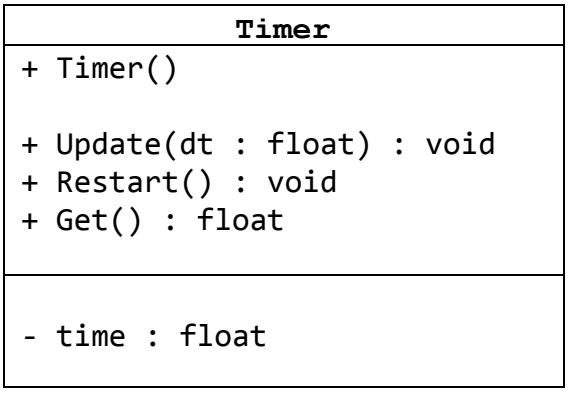
Já o friendly fire é um pouco mais complicado de tratar. Usaremos uma solução não muito agradável, mas simples e efetiva. Adicione o seguinte membro à Bullet:

```
+ 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. Note que para ler o valor da flag, você precisa castar other para Bullet& em NotifyCollision.

Nossa detecção de colisões está pronta! Nossas entidades levam dano e… somem, sem cerimônia. Me prometeram explosões!

4. Timer: Observando Intervalos de Tempo



Um componente muito simples, mas muito útil para um jogo é um contador de tempo. Timer acumula dts recebidos e, quando é pedido, retorna o tempo decorrido desde o início da contagem.

```
> Timer()
```

O timer é criado com o contador zerado.

`> Update(dt : float) : void`

Acumula os segundos em `time`.

`> Restart() : void`

Zera o contador.

`> Get() : float`

Retorna o tempo.

Como foi dito, um componente muito simples. A primeira coisa em que você deve aplicar o Timer são os Penguins: Use o Timer para impor um cooldown nos tiros deles. Não é necessário fazer o mesmo para o Alien, pois temos outros planos pra ele mais tarde.

Enfim, **explosões**.

5. Animation: Mostrando Animações Arbitrárias

Animation (herda de GameObject)

`+ Animation(x : float, y : float,`
`  rotation : float,`
`  sprite : string,`
`  frameCount: int, frameTime : float,`
`  ends : bool)`
 	
`+ Update(dt : float) : void`
`+ Render() : void`
 	
`+ IsDead() : bool`
 	
`+ NotifyCollision(other : GameObject*) : void`
`+ Is(type : string) : bool`

`- endTimer : Timer`
`- timeLimit : float`
`- oneTimeOnly : bool`
`- sp : Sprite`

É fácil perceber alguma semelhança entre Animation e Bullet: Ambas aceitam um Sprite arbitrário. No entanto, enquanto Bullet tem um significado para outras entidades, a única atribuição de Animation é executar sua animação, sem se mover ou reagir a outros objetos.

Essa animação pode ser executada indefinidamente (por exemplo, o fogo de uma tocha no seu cenário), ou pode ter um limite de tempo (geralmente uma execução completa). Nesse último caso, um Timer é empregado para designar a animação como "morta".

```
> Animation(x : float, y : float, rotation : float,
            sprite : Sprite, timeLimit : float, ends : bool)
```

Copia o Sprite, inicializa os membros herdados de GameObject, seta a flag ends e seta o timeLimit.

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

Atualiza a animação e o Timer.

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

Renderiza a animação na posição do objeto.

```
> IsDead() : bool
```

Se a animação nunca deve parar, ela nunca morre. Se ela está marcada para parar, devemos checar se o timer já atingiu o limite de tempo que calculamos.

```
> NotifyCollision(other : GameObject*) : void
```

Animation não deve reagir a uma colisão.

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

Is testará o tipo contra "Animation". Inclusive, se seu jogo final tiver muitas animações, uma maneira simples de evitar cálculos de colisão desnecessários é, antes de checar a colisão, checar se o objeto é uma Animation.

Quando os Penguins ou um Alien perderem HP, faça com que eles mesmos chequem se estão mortos. Se sim, crie na posição deles uma

Animation que use a sheet de explosão dada nos arquivos dos trabalhos (*img/aliendeath.png* e *img/penguindeath.png*). O timeLimit deve ser suficiente para a animação inteira ser mostrada: como você tem os parâmetros da animação (número e tempo de frames), é fácil calcular.

Nossas explosões bem que podiam ter som... Mas isso dá um pouco mais de trabalho. Vamos resolver outro problema: O Alien ainda está preso ao input, mas ele é, na verdade, um inimigo. Ele precisa de uma AI para funcionar como tal.

6. Mudanças em Alien

Teremos uma aula mais tarde sobre AI. É um campo complicado, normalmente, mas para esse caso, usaremos um padrão que se repete indefinidamente, evitando assim a tomada de decisões mais complexas.

Remova a classe Action e a fila de tarefas de Alien, e adicione os seguintes membros:

```
+ alienCount : int

- enum AlienState { MOVING, RESTING }
- state : AlienState
- restTimer : Timer
- destination : Vec2
```

A primeira é uma variável que diz quantas instâncias do Alien existem. Você deve incrementá-la no construtor e decrementá-la no destrutor. Já a enum nos dá os estados da máquina de estados que vamos implementar. Deve ser privada à classe, assim como era Action.

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 Alien obtem a posição atual do jogador e guarda em destination. Daí, calcula seu vetor velocidade, e muda o seu estado para MOVING.

Se o estado do Alien é MOVING, a cada frame ele se move em direção à posição da fila. Quando ele chega a essa posição, (ou perto o suficiente), ele obtem 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 Alien::Update, e certos trechos de código podem ser aproveitados. Porém, tome cuidado, alguns ajustes precisam ser feitos, especialmente em relação à câmera. Além disso, se o jogador morre, o Alien não faz mais nada.

Mas espera, já acabou? Pra que precisamos de alienCount, então? alienCount indica se ainda há inimigos no mapa. É a condição de vitória do jogo. Penguins::player ser nulo é a condição de derrota. Lembre-se disso, pois no próximo trabalho, terminaremos o nosso jogo, com telas de vitória e game over, além de sons e texto.