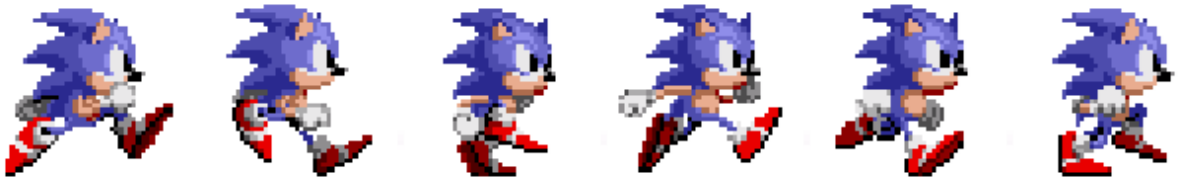
Trabalho 6 – Animações, Movimento Acelerado e Colisão

Um grupo de alienígenas 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 ideia 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 : std::string,
          frameCount : int = 1,
          frameTime : float = 1)***
+ SetFrame (frame : int) : void
+ SetFrameCount (frameCount : int) : void
+ SetFrameTime (frameTime : float) : void
```

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

***Adapte o construtor já existente. Lembre-se de inicializar os parâmetros novos no construtor 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. Para SetFrameCount, recomendamos resetar o frame inicial para 0 e recalcular a box do GameObject associated, visto que a largura será alterada (não se esqueça de considerar a escala).

Outra mudança é:

```
> 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. O primeiro objeto animável do nosso trabalho seria a Bullet.

No construtor de Bullet, adicione os parâmetros frameCount e frameTime, e use-os para criar o Sprite. Em Minion, troque o Sprite passado para Bullet 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 ideia 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

PenguinBody (herda de Component)
+ PenguinBody (GameObject& associated) + ~PenguinBody () + Start() : void + Update (dt : float) : void + Render () : void + Is (type : std::string) : bool + player : PenguinBody*
- pcannon : std::weak_ptr< GameObject > - speed : Vec2 - linearSpeed : float - angle : float - hp : int

PenguinCannon (herda de Component)
+ PenguinCannon (GameObject& associated, Std::weak_ptr< GameObject > penguinBody) + Update (dt : float) : void + Render () : void + Is (type : std::string) : bool + Shoot () : void
- pbody : std::weak_ptr< GameObject > - angle : float

Finalmente, chegamos aos nossos protagonistas. Os penguins farão um objeto composto por um par de pinguins: Um é o pinguim de baixo, que desliza sobre o gelo (PenguinBody), e o outro é pinguim controlando o canhão montado em cima (PenguinCannon).

Uma peculiaridade: na classe penguinBody, 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.

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.

```
> PenguinBody::PenguinBody (associated : GameObject&)
```

Inicialize todas as variáveis. Além disso, adicione a Sprite, e inicialize a variável da instância com `this`.

```
> PenguinCannon::PenguinCannon (associated : GameObject&,
                                penguinBody : std::weak_ptr<
                                              GameObject >)
```

Inicialize todas as variáveis. Além disso, adicione a Sprite.

```
> PenguinBody::~PenguinBody ()
```

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

```
> PenguinBody::Start ()
```

Crie o `PenguinCannon`, adicione ao estado atual do jogo e ao `pcannon`.

```
> PenguinBody::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.

Por último, solicite a deleção de si mesmo e do pinguim do canhão se hp for menor ou igual a zero.

Obs.: É bem possível, usando rotação de vetores, não usar a variável linearSpeed. Ela só serve para simplificar e muito o trabalho, especialmente considerando que os pinguins podem andar de ré. Principalmente se quiser implementar mecânicas como atrito.

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

Antes, vamos verificar se o PenguinBody ainda existe. Porque, se ele foi deletado, devemos deixar de existir também (assim como os minions em relação ao Alien). Em seguinte, vamos fazer com que o centro da box dele seja igual ao centro da box do corpo. Depois, 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. Não se esqueça de considerar a posição da câmera.

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

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

Deixe vazio para ambas as classes.

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

Você já entendeu como fazer essa não é mesmo?

```
> PenguinCannon::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 PenguinCannon onde será colocada a Bullet.

Em State::State(), instancie PenguinBody 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. 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`
`+ Is (type : std::string) : bool`
`+ 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::Is (type : std::string) : bool
> 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 Alien, Minion, Bullet, PenguinBody e PenguinCannon e adicione não só Sprite mas também 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

pinguins 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 Penguins ou Alien, 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.

Caso você tenha componentes que herdem de outros componentes (isso é algo muito mais avançado do que vamos utilizar nesse jogo), a função `Is` tem a vantagem de que podemos usá-la para apontar objetos que herdam de uma mesma classe. Suponha que Alien, Penguins e mais alguns componentes 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 Alien::Is(std::string type) {
      return (“Alien” == type || Being::Is(type));
}
```

Agora você pode escrever a função NotifyCollision dos objetos que agora possuem Collider. 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, os pinguins ficarem 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.

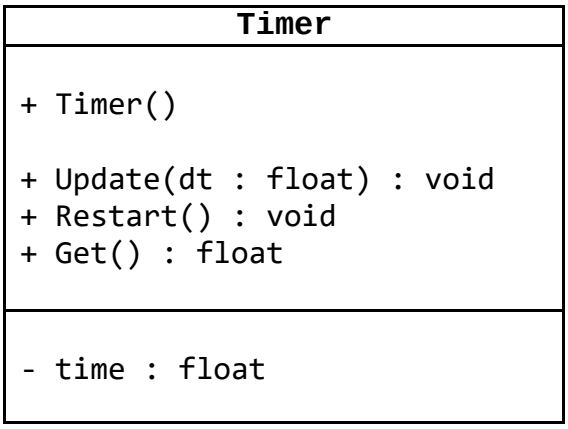
Nossa detecção de colisões está pronta! Mas está meio estranha. As vezes a bullet atinge “à distância” dos alvos. Se tivéssemos como debugar isso...

Collider::Render ao resgate! No moodle 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... somem, sem cerimônia. Me prometeram explosões! Eu quero minhas 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 é o PenguinCannon: Use o Timer para impor um cooldown nos tiros dele. Não é necessário fazer o mesmo para o Alien, pois temos outros planos pra ele mais tarde.

Enfim, **explosões**.

5. Mostrando Animações Arbitrárias

Animação nada mais é que uma Sprite que pode ter um prazo de validade. Então o que faremos será simplesmente acrescentar ao Sprite um argumento em seu construtor e dois atributos à classe.

A 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 da spritesheet). Nesse último caso, um Timer é empregado para designar a animação como “morta”.

Ao construtor, adicione um argumento `secondsToSelfDestruct : float = 0`. À classe, adicione os atributos `selfDestructCount : Timer` e `secondsToSelfDestruct : float`. No Update, cheque se secondsToSelfDestruct é maior que 0, se sim, ela tem prazo de validade. Então incremente o timer e se ele passar do prazo, solicite deleção.

Quando os Penguins ou um Alien perderem HP, faça com que eles mesmos chequem se estão mortos. Se sim, crie na posição deles um GameObject uma Sprite com prazo de validade 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. Também crie o componente Sound (com *audio/boom.wav*), dê play e adicione a esse GameObject.

Vamos resolver um 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

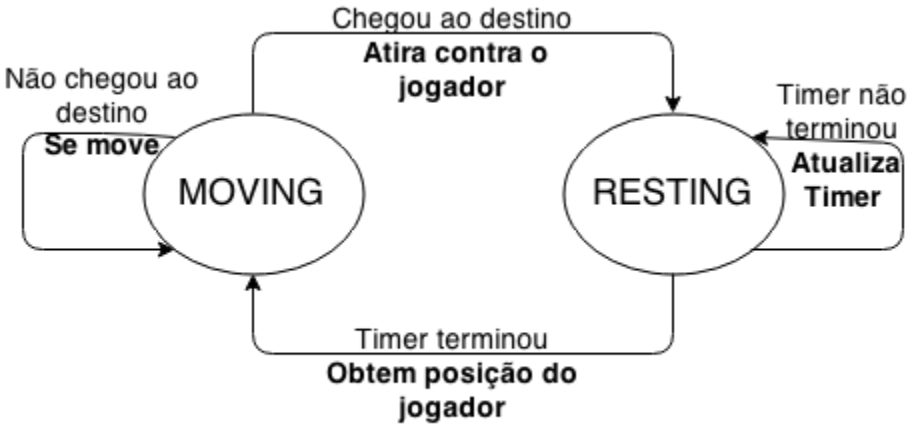
Teremos uma aula mais tarde sobre AI. É um campo complicado, normalmente, mas para esse caso, usaremos um padrão que se repete indefinidamente, evitando assim a tomada de decisões mais complexas.

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 obtém 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 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 Alien::Update, e certos trechos de código podem ser aproveitados. 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. PenguinBody::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 texto.