# Trabalho 2 – Geometria, Animações e Arrays de Objetos

## 1. Rect e Vec2

| Rect |
| --- |
| + x, y, w, h : float |

| Vec2 |
| --- |
| + x, y: float |

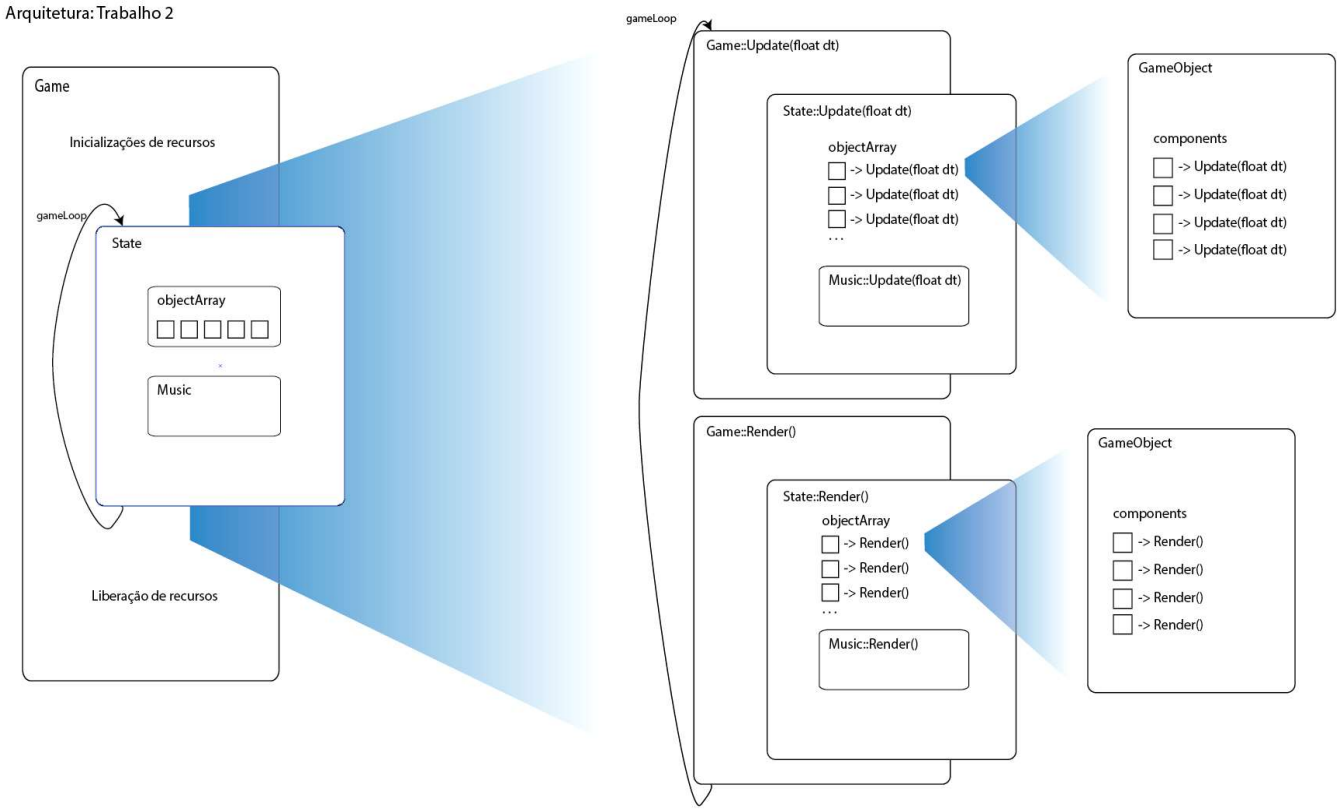
Nossas classes de geometria são bastante simples. `Vec2` expressa um vetor no R2, que pode tanto representar uma posição no espaço como uma grandeza. `Rect` expressa uma posição (canto superior esquerdo do retângulo) e dimensões. Exigiremos apenas esses membros para esse trabalho. No entanto, é fortemente recomendado que você desenvolva funções para suas classes, já que usaremos muitos cálculos geométricos ao longo do semestre. Algumas possibilidades:

- Construtores com inicialização em valores dados e/ou em zero
- Soma/subtração de vetores
- Multiplicação de vetor por escalar
- Magnitude
- Cálculo do vetor normalizado
    - Um vetor normalizado é um vetor unitário (de magnitude 1) com a mesma direção do vetor original
    - Matematicamente, podemos demonstrar que, ao dividir os componentes de um vetor pela magnitude dele, obteremos um vetor unitário.
- Distância entre um ponto e outro
    - Equivalente à magnitude da diferença entre dois vetores
- Inclinação de um vetor em relação ao eixo x
    - Atente para a diferença entre `atan()` e `atan2()`
- Inclinação da reta dada por dois pontos
    - A diferença entre dois vetores tem a mesma inclinação da reta - note que a ordem na subtração importa!
- Rotação em um determinado ângulo
    - O algoritmo para tal é baseado em matrizes de rotação:

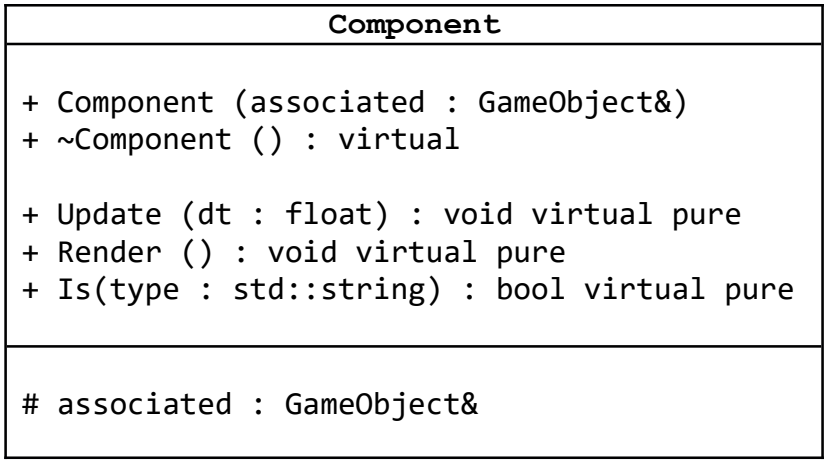
        - `x' = x * cosθ - y * sinθ`
        - `y' = y * cosθ + x * sinθ`
    - Note que, para um eixo y positivo para baixo, um ângulo positivo resulta numa rotação no sentido horário.
- Soma de `Rect` com `Vec2`
- Obter coordenadas do centro de um retângulo
- Distância entre o centro de dois `Rects`
- Saber se um ponto está dentro de um `Rect`
- Operadores de atribuição, soma, subtração
    - Isso não é necessário em momento algum, mas é uma feature interessante da linguagem C++ que está descrita na seção 14 do Apoio de C++ e será de grande ajuda.

Lembre-se trabalharemos com um eixo y que cresce para baixo, e que as funções de trigonometria da biblioteca padrão usam ângulos em radianos.

## 2. Component: Onde o jogo acontece



Observe que agora, State possui um objectArray, que será populado por GameObjects. Cada GameObject por sua vez, serve como uma caixa de Components. Ao invés de passar comportamentos semelhantes por meio de heranças, que podem se tornar longas e difíceis de manter, encapsulamos tais comportamentos em um componente, que serve como interface, trazendo maior flexibilidade ao código. O objeto que precisar daquele comportamento... adiciona o componente!

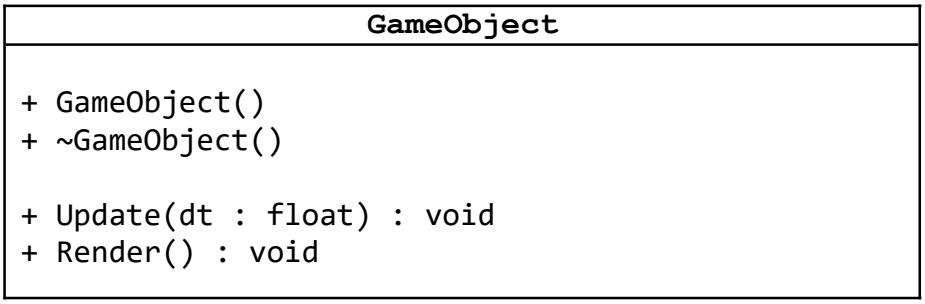


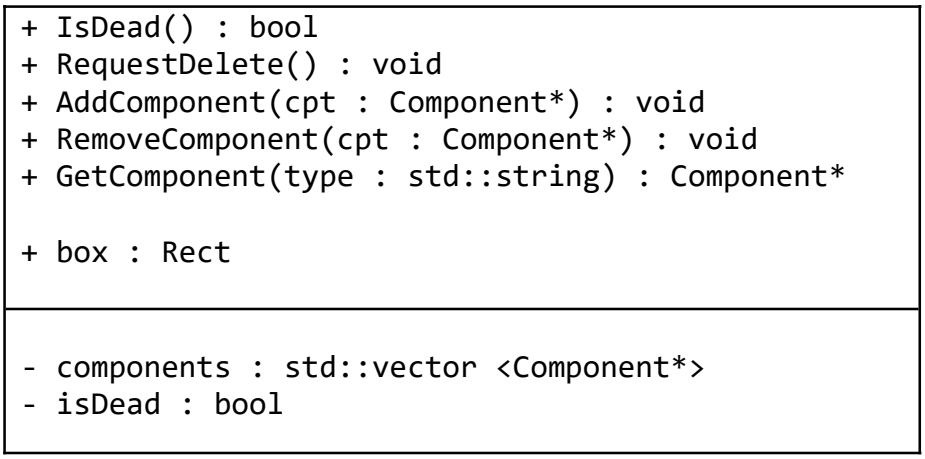

Essa é a classe que deve ser utilizada para adicionar lógica ao jogo utilizando herança. Todos os componentes do nosso jogo terão, no mínimo, as seguintes características:

- Referência ao `GameObject` que o contém.
- Função que atualiza estado do componente.
- Função para renderizar o que for necessário.
- Função para se determinar o seu tipo.

Como vocês podem observar, não temos como compilar o projeto no momento pois não criamos a classe `GameObject`. Isso não é motivo de pânico, o criaremos logo abaixo.

### 3. GameObject: Aquele que organiza os Componentes






`GameObject` é um agrupador de lógicas que estarão implementadas em seus componentes. Todo `GameObject` (GO para ficar mais curto) possui uma posição no jogo (`box`). Observe o tipo do atributo components. Esse array é uma estrutura de dados do tipo vector. Para os não-íntimos: `<vector>` é uma biblioteca padrão do C++. `std::vector`, o tipo definido nela, é um array que sabe se redimensionar sozinho caso seu tamanho máximo seja excedido.

A `<vector>` faz parte da chamada Standard Template Library, o conjunto de estruturas de dados pré-definidas em templates na linguagem, e uma das maiores vantagens de se usar C++ ao invés de C puro. Voltaremos a usar a STL mais vezes durante o curso.

`> GameObject ()`

Inicializa `isDead` com falso.

`> ~GameObject ()`

Percorre vetor de components dando delete em todos e depois dando `clear` no vetor. Dica: Percorra o vetor de trás para frente e chame `delete` usando o iterador do começo mais o index atual.

`> Update (dt : float)`

Percorre o vetor de componentes chamando o `Update(dt)` dos mesmos.

`> Render ()`

Percorre o vetor de componentes chamando o `Render` dos mesmos.

> IsDead ()

Retorna isDead.

> RequestDelete ()

Atribui verdadeiro a isDead.

> AddComponent (cpt : Component*)

Adiciona o componente ao vetor de componentes.

> RemoveComponent (cpt : Component*)

Remove o componente do vetor de componentes, isto é, se ele estiver lá.

> GetComponent (type : std::string)

Retorna um ponteiro para o componente do tipo solicitado que estiver adicionado nesse objeto. nullptr caso esse componente não exista.

Com o nosso GameObject pronto, podemos fazer as mudanças necessárias em state para armazenar e gerenciar todos os nossos objetos.

### 4. Mudanças em State

| **State (membros adicionais)** |
| --- |
| + ~State ()<br>+ AddObject (GameObject* go) : void |
| - objectArray : std::vector<std::unique_ptr<GameObject>> |

Para administrar os objetos instanciados no jogo, vamos manter um array de ponteiros para GOs. Perceba que não se trata de um vector de ponteiros, simplesmente. Estamos usando um outro template, contido em <memory>. É o std::unique_ptr. Essa classe recebe um ponteiro na sua instanciação, e se comporta como se fosse o próprio ponteiro. Sua importância está no fato de que, quando o unique pointer é apagado ou sai

do escopo, a área de memória para a qual o seu ponteiro aponta é automaticamente liberada.

Quando trabalhamos com containers de ponteiros, um erro muito comum é remover um ponteiro do array sem usar `delete` antes. Os `std::unique_ptrs`, introduzidos no C++11, resolvem esse problema com um overhead extremamente pequeno.

`> State ()`

Agora, bg é do tipo Sprite, porém queremos passar essa responsabilidade para um objeto. Criaremos um componente para cuidar da renderização de sprites mais à frente.

`> ~State ()`

Esvazia o array de objetos (`clear`).

`> Update ()`

No começo do método, percorra o vetor de GameObjects chamando o Update dos mesmos. No final, percorra o array de objetos testando se algum dos GameObjects morreu. Se sim, remova-a do array (`erase`). O loop de percorrimento do array precisa usar índices numéricos, já que iteradores se tornam inválidos caso um elemento seja adicionado ao vetor (o que vai acontecer em trabalhos futuros).

Sendo assim, para obter o iterador exigido como argumento de `vector::erase`, use o iterador de início (`vector::begin`) somado à posição do elemento.

`> Render ()`

Render deve percorrer o array chamando a função Render de todos os objetos nele. Aqui, não faz diferença usar iterador ou índice.

`> AddObject (GameObject* go)`

Essa função deve armazenar o ponteiro no vetor de objetos. Para isso, use a função emplace_back() de Vector, passando o ponteiro de GameObject dos parâmetros como argumento.

Com isso feito, podemos agora compilar o nosso projeto e ver que nada mudou ainda. Precisamos de componentes para podermos montar objetos com comportamentos desejáveis. Já temos como renderizar sprites como uma imagem estática, porém queremos ter como criar sprites animados, e associar eles a objetos do nosso jogo.

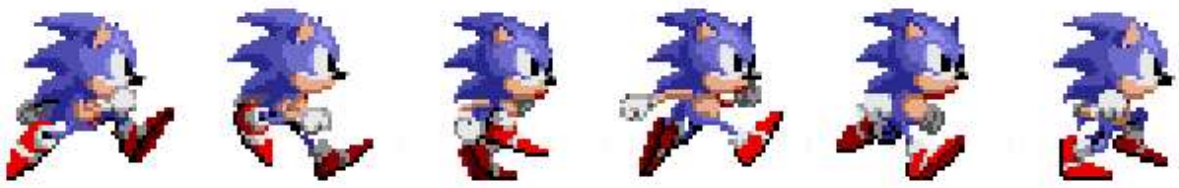
## 5. Animações

Para resolver o nosso problema, duas coisas serão necessárias. A primeira é adaptar os nossos sprites para ter animações, e a segunda é criar componentes que possam associar esses sprites a objetos e gerenciar as mudanças de animação.

### 5.1. 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 e altura de cada quadro, que também nos dá o offset em x e y de um frame para outro, respectivamente. Vamos adaptar nossa classe Sprite para funcionar como uma sprite sheet de duas dimensões.

Adicione os seguintes membros em Sprite:

```
+ Sprite (file : std::string,
          frameCountW : int = 1,
          frameCountH : int = 1,)***
+ Render (x : int, y : int,
          w : int, h : int)***
+ SetFrame (frame : int) : void
+ SetFrameCount (frameCountW : int,
           frameCountH : int) : void
```

```
- frameCountW : int
- frameCountH : int
```

***Adapte o construtor e render já existentes. Lembre-se de inicializar os parâmetros novos no construtor também!

`> SetFrame (frame : int) : void`

Deve setar o clip da imagem referente ao frame especificado. Para isso, chame SetClip() com x e y igual ao ponto de inicio do frame, w e h iguais à largura e altura de um frame respectivamente. Para encontrar o ponto de inicio do frame, divida o índice dele pela quantidade de frames em cada linha. O resultado te dará o número de linhas a pular, e o resto, o número de colunas. Multiplique esses números pela altura e largura de um frame para encontrar y e x respectivamente. Também é importante checar se esse clip está dentro do espaço da imagem original.

`> SetFrameCount (frameCountW : int, frameCountH : int) : void`

Seta os respectivos membros. Usada para Sprites criados com o construtor padrão.

Outra mudança é:

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

Devem retornar a largura e altura de apenas um dos frames.

### 5.2 Animation

| **Animation** |
| --- |
| + Animation (frameStart : int,<br>frameEnd : int,<br>frameTime : float) |
| + frameStart : int<br>+ frameEnd : int<br>+ frameTime : float |

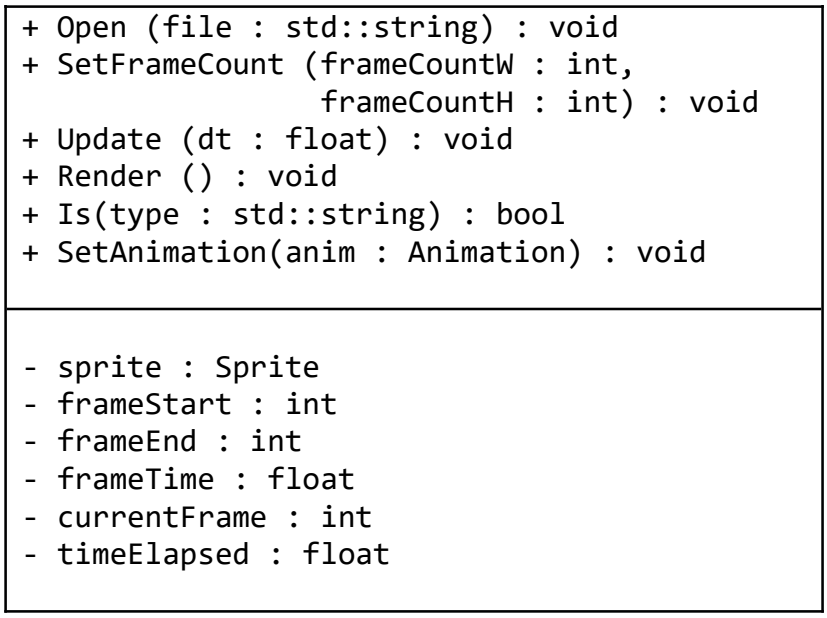
A classe Animation vai ser usada nos componentes para facilitar o gerenciamento de troca de animações. Ela nada mais é do que um conjunto de números que representa o primeiro e último frames da animação, e quanto tempo deve se passar em cada frame.

No construtor, apenas atribua os valores nos parâmetros aos respectivos atributos.

### 5.3 SpriteRenderer

Para cada objeto ter uma representação visual dentro do jogo, precisamos de um componente que gerencie a sua renderização. O componente SpriteRenderer vai servir de wrapper para `Sprite` e simplificar o processo de renderizar todos os sprites do nosso jogo.

| **SpriteRenderer (herda de Component)** |
| --- |
| + SpriteRenderer (associated : GameObject&)<br>+ SpriteRenderer (associated : GameObject&,<br>file : std::string,<br>frameCountW : int = 1,<br>frameCountH : int = 1) |




```
> SpriteRenderer (associated : GameObject&)
> SpriteRenderer (associated : GameObject&, file : std::string,
        frameCountW : int = 1, frameCountH : int = 1)
```

O seu construtor deve chamar o construtor de Component (https://stackoverflow.com/questions/6923722/how-do-i-call-the-base-class-constructor) em sua initializer list (http://en.cppreference.com/w/cpp/language/initializer_list). Aproveite e chame também o construtor de sprite. Inicialize frameStart, frameEnd, frameTime, currentFrame e timeElapsed como 0. No construtor que abre o arquivo do sprite, defina a altura e largura da box do GameObject associado com base no tamanho do sprite aberto.

```
> SetFrameCount (frameCountW : int, frameCountH : int) : void
> Open (file : std::string) : void
```

São apenas wrappers para as funções homônimas de seu sprite. Chame as funções passando os mesmos parâmetros. No caso de open, lembre de definir a altura e largura da box do GameObject associado.

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

Se frameTime for diferente de 0, incremente timeElapsed. Se timeElapsed for maior que frameTime, incremente currentFrame e subtraia frameTime de timeElapsed. Se currentFrame for maior que frameEnd, volte

para frameStart. Caso currentFrame mudar, chame SetFrame no sprite. Essa não é a forma ideal de implementar essa função, mas como não estamos calculando dt ainda, deixe assim por enquanto. Voltaremos aqui para utilizar dt e deixar as animações atreladas ao tempo.

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

Aqui basta chamar Render do sprite usando como parâmetros os valores da box do GameObject associado.

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

Essa função será extremamente semelhante em todos os componentes. Basta comparar a string dos parâmetros com uma string fixa, nesse caso “SpriteRenderer”, e retornar true quando forem iguais, false caso contrário.

```
> SetAnimation (anim : Animation) : void
```

Defina frameStart, frameEnd e frameTime de acordo com a animação nos parâmetros. Resete currentFrame para ser igual a frameStart e timeElapsed para 0. Por fim, chame SetFrame em sprite, passando currentFrame como argumento.

Vamos agora criar um objeto para ser o nosso background. Em state, Remova o membro bg. Primeiramente, devemos criar um ponteiro de GameObject. Crie agora um SpriteRenderer e adicione-o ao nosso objeto usando a função AddComponent. Coloque o ponteiro para esse GameObject criado no objectArray usando a função AddObject. Se tudo foi feito corretamente, pode compilar o projeto e ver a mesma imagem de antes, só que agora em formato de objeto e componente. Vamos criar algo agora que use as animações.

## 6. Zombie: Meu primeiro Componente de mecânicas

| Zombie (herda de Component) |
| --- |
| + Zombie (associated : GameObject&)<br><br>+ Damage (damage : int) : void<br><br>+ Update (dt : float) : void<br>+ Render () : void<br>+ Is (type : std::string) : bool |
| - hitpoints : int |

Zombie é um "inimigo" com uma determinada quantidade de HP.

`> Zombie (associated : GameObject&)`

Primeiro, deve construir construir a classe mãe Component na lista de inicialização e depois setar o valor inicial de `hitpoints`. (sugestão: 100 HP). Depois, crie um SpriteRenderer para seu objeto associado. Use a imagem "Enemy.png", com 3x2 frames. Depois de criado, inicie uma animação com os valores (0,3,10) usando a função SetAnimation. Não se esqueça de adicionar esse SpriteRenderer ao objeto com a função AddComponent.

`> Damage (damage : int)`

Deve reduzir os `hitpoints` na quantidade passada. E se ficar menor ou igual a zero, troque a animação para (5,5,0). Para isso, use a função GetComponent do seu objeto associado para obter o ponteiro de seu SpriteRenderer.

`> Update (dt : float)`

Por enquanto, Chame a função Damage com o valor de 1 todo frame. Esse comportamento será removido quando implementarmos os ataques.

`> Render ()`

Deixe vazio.

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

Retorne verdadeiro se o tipo for “Zombie”.

Com tudo isso feito, vá em State e crie um Zombie, da mesma forma que fez com o background. Mude a posição da box dele para 600,450. Compile seu projeto e teste ele. Se tudo estiver correto, verá um Zombie correndo sem se mover perto da linha do horizonte, que morrerá após algum tempo. Caso isso ocorra, parabéns! Porém, ter que definir a animação manualmente toda vez pode ser um pouco chato. Vamos criar um componente que ajuda a lembrar os detalhes de cada animação pra nós.

### 7. AnimationSetter

| **AnimationSetter (herda de Component)** |
|---|
| + AnimationSetter (associated : GameObject&)<br>+ Update (dt : float) : void<br>+ Render () : void<br>+ Is (type : std::string) : bool<br>+ SetAnimation (name : std::string) : void<br>+ AddAnimation (name : std::string, anim : Animation) : void |
| - animations : std::unordered_map<std::string, Animation> |

O ideal seria ter uma classe similar ao Animator da unity, capaz de criar transições elaboradas entre animações em momentos específicos com base em variáveis. Porém, no contexto da disciplina, basta ser capaz de trocar entre uma animação e outra. A classe AnimationSetter vai se encarregar de receber comandos de troca e passar os valores específicos para o nosso SpriteRenderer.

`> AnimationSetter (associated : GameObject&)`

Chame o construtor de component pela lista de inicialização.

`> Update (dt : float) : void`

`> Render () : void`

Deixe essas funções vazias.

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

Faça aqui como nos outros componentes.

`> SetAnimation (name : std::string) : void`

Busque no seu unordered_map por um par com chave = name. Se encontrar, chame SetAnimation do SpriteRenderer presente no mesmo objeto.

`> AddAnimation (name : std::string, anim : Animation) : void`

Busque no seu unordered_map por um par com chave = name. Se não encontrar, insira o par (name, anim) no seu mapa.

Agora vamos alterar o nosso Zombie mais uma vez. No construtor, crie um AnimationSetter para o nosso inimigo. Adicione os pares (“walking”,Animation(0,3,10)) e (“dead”,Animation(5,5,0)). Onde tinha chamado SetAnimation do SpriteRenderer, chame agora SetAnimation do AnimationSetter. Se tudo estiver correto, seu trabalho agora mostra um inimigo correndo no lugar por algum tempo, e depois morrendo, e você pode trocar essas animações por meio do nome delas no código. Com isso feito, no futuro será bem fácil de implementar sprites animados diversos trocando apenas alguns parâmetros e chamadas de função.