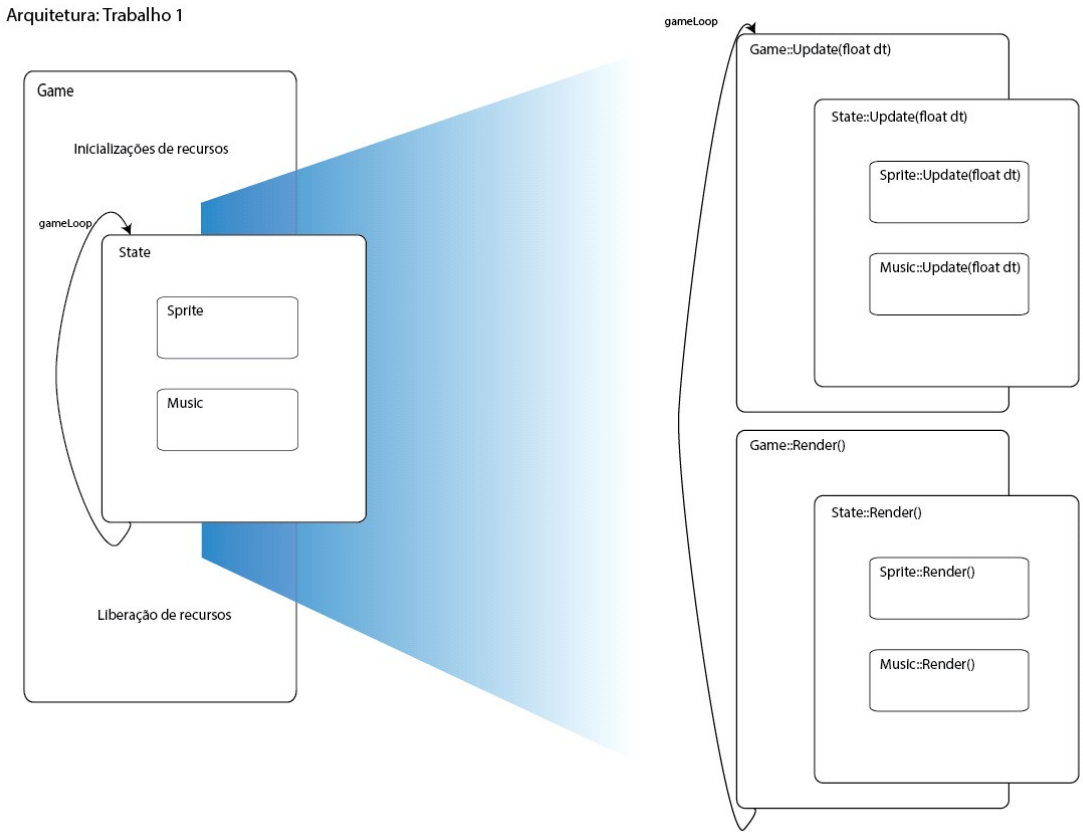
Trabalho 1 – Game Loop, Sprite e Music



A imagem acima representa a arquitetura que estamos prestes a montar para começar nossa engine. Os detalhes da implementação estão todos descritos abaixo. No momento, o importante é ter noção da responsabilidade de cada classe:

- Game é responsável por inicializar recursos, rodar o loop principal do jogo e liberar os recursos quando o jogo for encerrado;
- State é um "estado da tela de jogo". Podemos ter uma tela de apresentação, uma tela de NewGame, a tela do jogo em si, uma tela de fim de jogo, etc. Cada um desses seria um State. Começaremos implementando apenas um.
- Sprite irá tomar conta de uma imagem dentro do state;
- Music irá tomar conta de uma música no state.

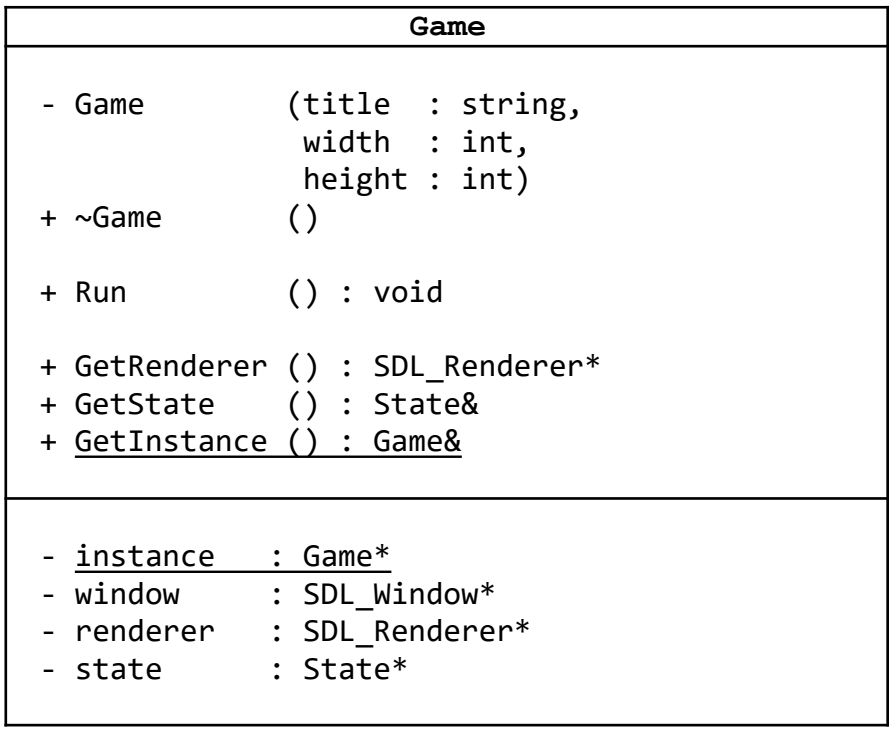
Para começarmos o trabalho, é recomendado criar a seguinte estrutura de

pastas no diretório do seu projeto:

- src: pasta para armazenar todos os arquivos .cpp compiláveis.
- include: pasta para armazenar todos os headers .h do seu projeto.
- Recursos: pasta com todos os assets que serão usados, disponível no Google Drive da disciplina.

Com essa estrutura, podemos começar a desenvolver o nosso jogo.

1. Game: A Estrutura Básica da Engine



Legenda: + membro public
membro protected
- membro private
membro static

A classe Game implementa as funções básicas da nossa engine, incluindo o main game loop, e a inicialização dos subsistemas (no caso, SDL, SDL_image e SDL_mixer) que precisaremos para outras classes funcionarem. Ao longo do semestre, faremos alterações na classe para incluir mais features.

Note que ela mantém registro da própria instância e que seu construtor é privado. Isso tem dois motivos: o primeiro é garantir que outras classes da engine tenham acesso ao renderizador e à janela. O segundo

propósito é impedir que haja múltiplas inicializações da biblioteca ou múltiplos jogos instanciados.

Crie os arquivos Game.h e Game.cpp. **TODOS OS ARQUIVOS CRIADOS DEVEM TER O MESMO NOME DE SUA CLASSE!** Essa é a forma convencional de nomear arquivos no contexto de programação orientada a objeto.

No header, usando o arquivo SDL_include.h fornecido, adicione "SDL.h". No .cpp, adicione "SDL_image.h" e "SDL_mixer.h". O primeiro é o cabeçalho para as funções principais da SDL, o segundo e terceiro são para funções que carregam imagens e áudio do disco rígido, respectivamente. Usaremos ambos nesse trabalho.

Implementando os métodos da classe:

`> GetInstance () : Game&`

Para os já familiarizados com padrões de projeto, podem ter observado que Game segue o padrão Singleton. Esse padrão é utilizado para impedir que uma classe possua mais de uma instância. Isso se consegue por conta do construtor privado, não sendo possível instanciar diretamente um objeto da classe, no caso Game. A única forma de utilizar a classe é por meio de seu método estático GetInstance(). Nesse método, a primeira coisa a se fazer é checar se já há uma instância dela rodando (`instance != nullptr`), se já existir, o retorne. Se não existir, instancie a primeira (e única!) instância de Game usando new.

Retorna `*instance` (que o compilador automaticamente resolverá como uma referência).

`> Game (title : string, width : int, height : int)`

Quando a classe é instanciada, a primeira coisa a se fazer é checar se já há uma instância dela rodando (`instance != nullptr`). Se já existir, há um problema na lógica do seu jogo. Se não existir, atribua *this* a instance. Isto é a base do padrão Singleton, e aparecerá em algumas outras classes ao longo do semestre.

Feito isso, devemos inicializar a biblioteca SDL e suas auxiliares antes de usar suas funções. Para tal, chamamos a função `SDL_Init(Uint32 flags)`,

com parâmetros correspondentes aos subsistemas da biblioteca que devem ser inicializados. As possibilidades de subsistemas são as seguintes:

```
SDL_INIT_TIMER          SDL_INIT_GAMECONTROLLER
SDL_INIT_AUDIO          SDL_INIT_EVENTS
SDL_INIT_VIDEO          SDL_INIT_EVERYTHING
SDL_INIT_JOYSTICK       SDL_INIT_NOPARACHUTE
SDL_INIT_HAPTIC
```

Mais informações sobre SDL_Init podem ser encontradas em http://wiki.libsdl.org/SDL_Init.

Pode-se usar vários parâmetros numa mesma chamada fazendo um *OU* lógico (*bitwise)* entre as flags. Para nossa disciplina, deverá bastar:

```
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER)
```

Uma observação: A `SDL_Init` retorna **diferente de zero quando falha**. Caso isso aconteça, deve-se abortar o programa com uma mensagem de erro. Convém usar SDL_GetError para saber a causa.

Quando funções da SDL falham, a função SDL_GetError pode ser usada para obter a mensagem do último erro ocorrido na biblioteca. Em situações de erro, pode ser interessante imprimir o retorno desta função num log.

Se a SDL foi inicializada corretamente, a próxima inicialização a se fazer é a da SDL_Image, via `IMG_Init`. As *flags* possíveis são `IMG_INIT_JPG`, `IMG_INIT_PNG` e `IMG_INIT_TIF`, e referem-se aos loaders de formatos de arquivos que você deseja usar no programa.

Diferentemente da SDL, a image não precisa ser inicializada explicitamente - o loader é carregado automaticamente quando é usado. No entanto, é boa prática tratar inicializações assim que possível, especialmente porque facilita o tratamento de erros.

A `IMG_Init` retorna um *bitmask* correspondente aos *loaders* que ela conseguiu carregar. Isso é, se ela conseguiu carregar os mencionados acima, seu retorno seria `(IMG_INIT_JPG | IMG_INIT_PNG | IMG_INIT_TIF)`. Se não carregar nenhum, o retorno é zero.

Após inicializarmos a biblioteca de imagens, temos que inicializar a biblioteca de sons, cuja interface está definida em SDL_mixer.h. Para Inicializarmos temos que chamar MIX_Init. Utilizando as seguintes funções:

```
int Mix_Init(int flags)
```

```
int Mix_OpenAudio(int frequency, Uint16 format, int channels, int chunksize)
int Mix_AllocateChannels(int numchans)


void Mix_CloseAudio()
void Mix_Quit()
```

A Mix_Init funciona como a IMG_Init: Recebe flags de inicialização e retorna as flags correspondentes aos subsistemas inicializados corretamente (no caso, decoders de diferentes formatos de música). Flags possíveis:

MIX_INIT_FLAC	**MIX_INIT_MP3**
MIX_INIT_OGG	**MIX_INIT_MOD**
MIX_INIT_FLUIDSYNTH	**MIX_INIT_MODPLUG**

O formato .wav está disponível por default, sem ser necessário inicializar. Assim como a IMG_Init, no entanto, se você não inicializar um decoder, ele é inicializado no primeiro uso. Quem faz a inicialização da API é a OpenAudio, e ela, sim, é obrigatória de se chamar.

Na função Mix_OpenAudio, channels é a quantidade de canais de saída de áudio que existem, ou seja, 2 para estéreo e 1 para mono. Use os valores MIX_DEFAULT_FREQUENCY, MIX_DEFAULT_FORMAT, MIX_DEFAULT_CHANNELS para os três primeiros argumentos, e 1024 para o chunksize. A função retorna 0 se for bem sucedida.

A quantidade de canais de áudio, diferentemente de canais de saída de áudio, serve para determinarmos quantos sons seremos capazes de tocar simultaneamente. Por default são 8. Nesse jogo não vamos precisar nem de 8, mas vamos mudar para 32 com Mix_AllocateChannels para caso o seu jogo final precise disso.

Obs.: Essa função só falha na sua execução se você não tiver muita memória disponível no seu computador **e** passar uma quantidade absurdamente grande de canais (muito grande *mesmo*). Caso seu jogo precise de centenas de canais, não se preocupe e vai fundo! Caso não precise, não há necessidade de desperdiçar memória alocando espaço que não vai ser usado.

Com a biblioteca pronta, podemos criar uma janela. Primeiro, deve-se criar a janela (membro *window*). A função para tal é

```
SDL_Window* SDL_CreateWindow(const char* title, int x, int y, int w, int h, Uint32 flags)
```

É possível posicionar a janela na tela usando `x` e `y`, e o macro `SDL_WINDOWPOS_CENTERED`, que pode ser usado para ambos os argumentos,

garante que ela será posicionada no centro da tela, independentemente da resolução da mesma.

`title`, `w` e `h` dizem respeito ao título e às dimensões da janela, e são os próprios argumentos do construtor de Game. Já *flags* são usadas para coisas como fullscreen. Nos trabalhos da disciplina, não usaremos nenhuma. Passe 0 nesse argumento. Caso queira saber que flags estão disponíveis, ou tenha dúvidas sobre a criação da janela, consulte http://wiki.libsdl.org/SDL_CreateWindow .

Em seguida, criamos um renderizador para esta janela. Use

```
SDL_Renderer* SDL_CreateRenderer(SDL_Window* window, int index, Uint32 flags)
```

onde `window` é a janela à qual o renderizador deve ser atrelado. O valor de `index` determina qual dos drivers de renderização queremos usar. Use o valor -1: Isso fará a SDL escolher o melhor renderizador para o ambiente e para as *flags* setadas. Para *flags*, por sua vez, podemos usar:

```
SDL_RENDERER_SOFTWARE        SDL_RENDERER_PRESENTVSYNC
SDL_RENDERER_ACCELERATED     SDL_RENDERER_TARGETTEXTURE
```

Use `SDL_RENDERER_ACCELERATED`, para requisitar o uso de OpenGL ou Direct3D.

Note que, se `SDL_CreateWindow` ou `SDL_CreateRenderer` falham, retornam nullptr. Se todas as inicializações até agora deram certo, terminamos essa etapa! A última coisa que o construtor deve fazer é instanciar um `State` para o nosso jogo, inicializando o membro `state`. Discutiremos a funcionalidade dessa classe na seção 2.

`> ~Game ()`

O destrutor desfaz as inicializações: deleta o estado, encerra a SDL_Music(Mix_CloseAudio e Mix_Quit) e a SDL_image (`IMG_Quit`), destrói o renderizador e a janela (`SDL_DestroyRenderer`, `SDL_DestroyWindow`), e, finalmente, encerra a SDL (`SDL_Quit`). A ordem importa! Faça na ordem inversa da inicialização, ou seja, quem foi inicializado primeiro será o último a ser destruído.

`> GetState () : State&`

Retorna `*state`. Isso será útil mais tarde, quando classes do jogo

queiram.

`> GetRenderer () : SDL_Renderer*`

Retorna o membro `renderer`. Será usada principalmente pelo componente Sprite.

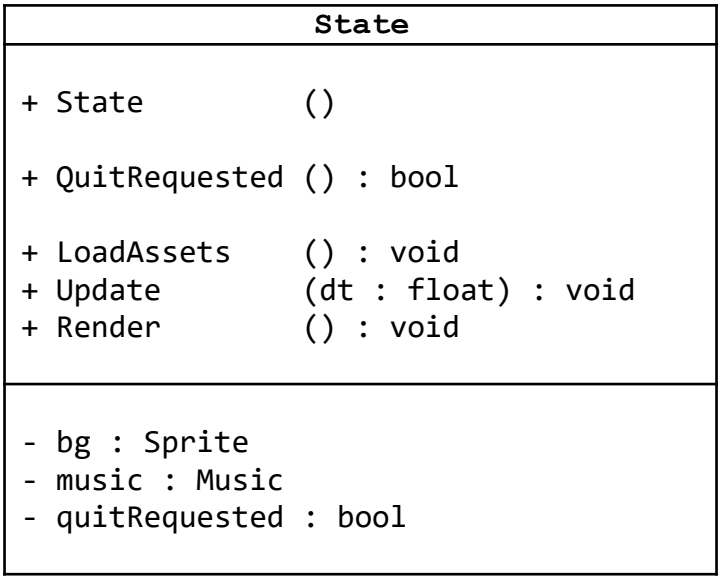
`> Run () : void`

Run é o que chamamos de Game Loop. Em um jogo, tudo ocorre dentro de um *loop*, sendo cada *frame* uma iteração, que, de forma simplificada, acontece assim:

1. Verifica, controla e carrega as telas de jogo;
2. Os dados de input são recebidos e processados;
3. Os objetos tem seus respectivos estados (posição, HP…) atualizados;
4. Os objetos são desenhados na tela.

Os passos 1 e 2 serão implementados em trabalhos futuros, quando criarmos um sistema separado para administrar inputs e tivermos estrutura para trocar as telas de jogo. Já os passos 3 e 4, assim como a interrupção do loop, dependem da classe State, que ainda não definimos. Logo, antes de terminar Run, vamos definir exatamente o que é o estado do jogo.

2. State: Lógica Específica



State é a responsável pela lógica específica do seu jogo. Game conhece apenas a manipulação do estado do jogo de maneira genérica, e vai chamar as funções de State sem saber, necessariamente, como ele se comporta.

State crescerá bastante ao longo do semestre, mas por enquanto, contem apenas os atributos bg, um Sprite (seção 3) a ser renderizado na tela; e quitRequested, um indicador de State para Game de que o jogo deve ser encerrado.

```
> State ()
```

O construtor de State inicializa `quitRequested` e instancia o Sprite, que será definido na seção 3. Consulte o Apoio de C++ se tiver dúvidas sobre como chamar construtores de membros.

```
> LoadAssets    () : void
```

O LoadAssets é uma método que cuida de pré-carregar os assets do state do jogo para que não haja problemas futuros como, por exemplo, o jogo tentar tocar a música antes dela terminar de ser carregada para a memória. Deixe para carregar imagens/fontes/músicas às suas variáveis aqui sempre que for possível.

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

Update trata da etapa 3 que discutimos em Game::Run. Até o fim do semestre, ela tratará da atualização do estado das entidades, testes de colisões e a checagem relativa ao encerramento do jogo. Como não temos entidades ainda, para esse trabalho, faremos só a última coisa.

Nesse trabalho, as condições de saída do programa são o usuário clicar no “X” da janela ou apertar Alt+F4. Para saber se isso ocorreu usaremos a função `SDL_QuitRequested()`. Se o retorno dela for `true`, sete a flag `quitRequested` para `true`.

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

Render, por sua vez, trata da etapa 4 de Game::Run, a renderização do estado do jogo. Isso inclui entidades, cenários, HUD, entre outros. Para esse trabalho, chame o render do fundo (bg) passando (0,0) como parâmetros.

`> QuitRequested () : bool`

QuitRequested retorna o valor da flag de mesmo nome na função, que será usado por Game para interromper o game loop.

Tendo estabelecido o funcionamento de State, voltemos à classe Game.

`> Run () : void`

Run é um simples loop, que funciona enquanto QuitRequested não retornar true. Dentro desse loop, chamamos Update (passe 0 como parâmetro por enquanto) e Render do estado. Em seguida, chamamos a função `SDL_RenderPresent`, que força o renderizador passado como argumento a atualizar a tela com as últimas renderizações feitas. Sem chamar essa função, a janela continuará vazia.

Por último, vamos impor um limite de frame rate ao jogo. O controle do frame rate costuma ser algo mais sofisticado, mas na nossa disciplina, nosso único interesse é impedir que o jogo use 100% da CPU sem necessidade.

Para isso, usa-se a função `SDL_Delay`, que atrasa o processamento do próximo frame em um dado número de milissegundos. Faça:

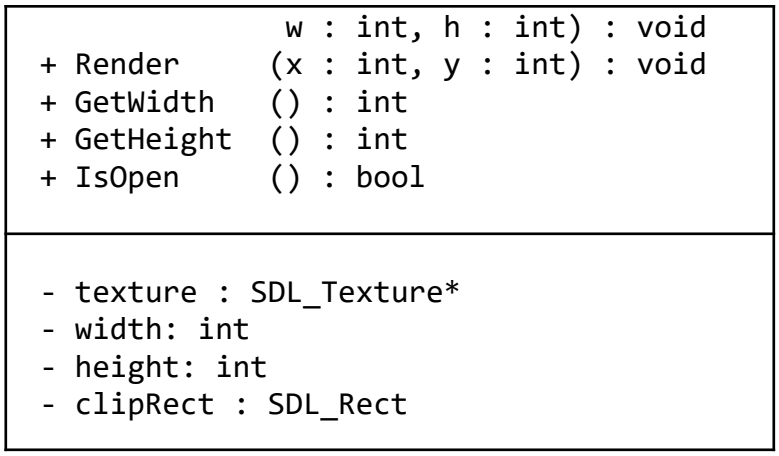
```
SDL_Delay(33);
```

Um frame a cada 33 milissegundos nos dará aproximadamente 30FPS. Note que isso pode cair caso haja muitas tarefas a serem executadas no frame, e que SDL_Delay pode exceder o número de ms pedidos por causa do agendamento do sistema operacional.

Assim, o main game loop está pronto! Apenas um probleminha... Não estamos renderizando nada ainda.

3. Sprite: Carregamento e Renderização de imagens

Sprite
+ Sprite ()
+ Sprite (file : string)
+ ~Sprite ()
+ Open (file : string) : void
+ SetClip (x : int, y : int,



A classe `Sprite` encapsula o carregamento e uso de SDL_Textures, o tipo da SDL que contém uma imagem carregada do disco pronta para ser renderizada num SDL_Renderer. Sprite tem quatro atributos:

- `texture`: A imagem em si
- `width, height`: As dimensões da imagem
- `clipRect` : O retângulo de *clipping* (determina uma parte específica da imagem para ser renderizada)

`> Sprite ()`

Seta texture como nullptr (imagem não carregada).

`> Sprite (file : string)`

Seta texture como nullptr. Em seguida, chama `Open`.

`> ~Sprite()`

Se houver imagem alocada, desaloca. Nunca use `delete` ou `free` em uma `SDL_Texture`. Use `SDL_DestroyTexture(SDL_Texture*)`.

`> Open(file : string) : void`

Carrega a imagem indicada pelo caminho *file*. Antes de carregar, deve-se checar se já há alguma imagem carregada em texture: Se sim, deve ser desalocada primeiro. Em seguida, carrega a textura usando

```
SDL_Texture* IMG_LoadTexture(SDL_Renderer* renderer, const char* path)
```

Onde `renderer` deve ser o renderizador de Game.

Trate o caso de IMG_LoadTexture retornar nullptr. É uma das

causas mais comuns de crashes nos trabalhos da disciplina. Novamente, o output de `SDL_GetError` sempre pode ajudar a descobrir o problema.

Com a textura carregada, precisamos descobrir suas dimensões. Use:

```
int SDL_QueryTexture(SDL_Texture* texture, Uint32* format, int* access, int* w, int* h)
```

Que obtém os parâmetros de texture e armazena-os nos espaços indicados pelos argumentos. O segundo e o terceiro parâmetros podem ser nullptr seguramente. Estamos interessados em `w` e `h`. Passe como argumento os endereços de `width` e `height`.

Por último, sete o clip com as dimensões da imagem, usando...

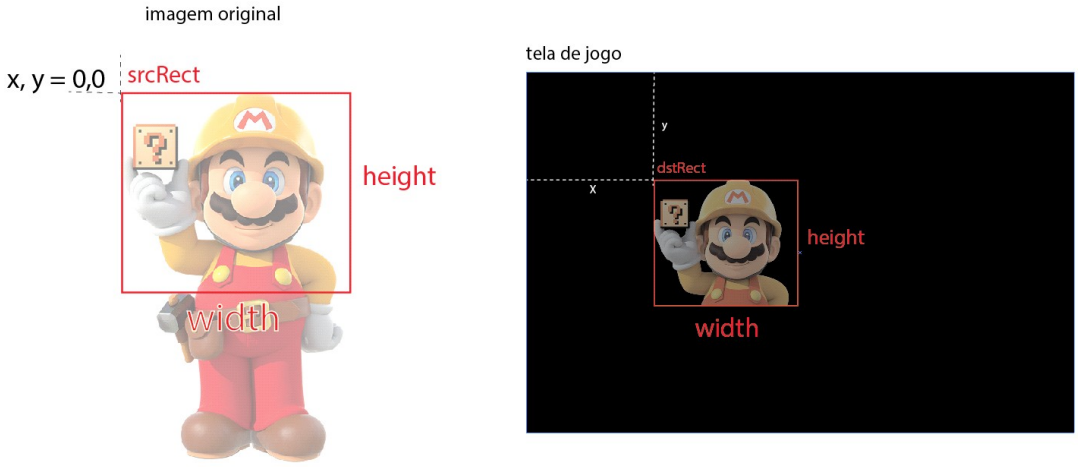
```
> SetClip(x : int, y : int, w : int, h : int) : void
```

Seta `clipRect` com os parâmetros dados.

```
> Render(x : int, y : int) : void
```

`Render` é um wrapper para `SDL_RenderCopy`, que recebe quatro argumentos.

- `SDL_Renderer* renderer`: O renderizador de Game.
- `SDL_Texture* texture`: A textura a ser renderizada;
- `SDL_Rect* srcrect`: O retângulo de clipagem. Especifica uma área da textura a ser "recortada" e renderizada.
- `SDL_Rect* dstrect`: O retângulo destino. Determina a posição na tela em que a textura deve ser renderizada (membros *x* e *y*). Se os membros *w* e *h* diferirem das dimensões do clip, causarão uma mudança na escala, contraindo ou expandindo a imagem para se adaptar a esses valores.



Já temos os três primeiros argumentos. Para dst, declare um SDL_Rect cujos membros `x` e `y` são os argumentos de Render, e `w` e `h`, os valores contidos no clipRect.

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

Retorna as dimensões da imagem.

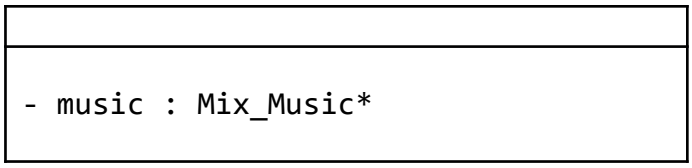
```
> IsOpen () : bool
```

Retorna *true* se `texture` estiver alocada.

E pronto! Podemos abrir imagens e mostrá-las na tela. Em State(), `bg` deve ser aberto com a imagem *img/Background.png*, presente no zip de resources para os trabalhos da disciplina, no Google Drive.

4. Music

Music
+ Music() + Music(file : std::string) + ~Music () + Play (times : int = -1) : void + Stop (msToStop : int = 1500) : void + Open (file : std::string) : void + IsOpen () : bool



Mix_Music e Mix_Chunk são tipos são definidos na SDL_mixer.h, e vamos usá-los agora. Semelhanças com a interface de Sprite são totalmente intencionais. Music "abre" um arquivo de música e o reproduz quando pedido.

```
> Music()
> Music(file : std::string)
```

O segundo construtor chama Open. O primeiro apenas inicializa music como nullptr para aguardar uma chamada a Open vinda de fora da função.

```
> Play (times : int = -1) : void
```

```
int Mix_PlayMusic(Mix_Music* music, int loops)
```

Mix_PlayMusic recebe uma música e quantas vezes ela deve ser tocada. Se loops for -1, a música repete infinitamente. Se loops for 0, a música não é tocada. Vale notar que a Mixer só suporta uma música sendo tocada por vez: Se outra música já estiver tocando, ela para.

Não se esqueça que Mix_Music pode ser nullptr.

```
> Stop (msToStop : int = 1500) : void
```

```
int Mix_FadeOutMusic(int ms)
```

Mix_FadeOutMusic para a música atual dando um efeito de fade, isto é, diminuindo gradualmente o volume até chegar em 0. O tempo para a música parar totalmente é passado como argumento da função, em milissegundos. Vamos deixar default como 1,5 segundos. Caso queria que pare imediatamente, basta passar 0 como argumento.

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

```
Mix_Music* Mix_LoadMUS(const char* file)
```

Carrega a música indicada no arquivo file. Lembre-se de tratar o caso em que nullptr é retornado.

```
> IsOpen () : bool
```

Checa se music é nula.

`> ~Music()`

Chama Stop e libera a música da memória.

```
void Mix_FreeMusic(Mix_Music* music)
```

Com a classe pronta, carregue o arquivo de música *audio/BGM.wav* no construtor do State. A música deve começar a tocar na criação do estado.

5. Main: Entry Point

A função `main` deve ser a mínima possível no seu jogo. Deve apenas criar um `Game`, executá-lo, e sair do programa. Lembrando que como `Game` é singleton, não devemos armazenar em uma variável do tipo `Game`, e sim em uma referência, que será retornada pelo método GetInstance. **A janela deve ter as dimensões 1200x900, e o título deve ser seu nome e matrícula.**

Importante: A SDL exige que a função main seja declarada com os argumentos argc e argv.

```
int main (int argc, char** argv)
```