

Trabalho 1 – Game Loop e Sprite

1. Classe SDLBase: Inicializando a biblioteca.

SDLBase
<pre>\$ screen : SDL_Surface*</pre>
<pre>\$ inicializaSDL() : int \$ getScreen() : SDL_Surface* \$ loadImage(arquivo : std::string) : SDL_Surface * \$ renderSurface(surface : SDL_Surface*, clip : SDL_Rect* = NULL, dst : SDL_Rect* = NULL) : void \$ atualizarTela() : void</pre>

A classe SDLBase será responsável por encapsular e facilitar algumas chamadas a funções da SDL e SDL_image, como inicialização, carregar imagens e atualizar a tela.

Crie a classe SDLBase (SDLBase.h e SDLBase.cpp).

No header, adicione os seguintes includes:

```
#include "SDL.h"
#include "SDL_image.h"
```

Não esqueça de incluir `<stdio.h>` ou `<iostream>` caso venha a usar funções de I/O e `<stdlib.h>` caso use a constante NULL.

A primeira biblioteca (SDL.h) contém as funções básicas de SDL e a segunda (SDL_image.h) possui funções para leitura de imagens.

Implementando os métodos da classe:

```
$ inicializaSDL() : int
```

É o método responsável por criar a Janela e uma surface para renderização.

- O primeiro passo a tomar é inicializar a biblioteca SDL. Para isso, utilizaremos a função `SDL_Init(Uint32 flags)`. Com ela, é possível inicializar diversos aspectos da biblioteca, os quais devem ser especificados nos parâmetros. Os parâmetros possíveis são:

SDL_INIT_AUDIO – inicializa o áudio
SDL_INIT_CDROM – inicializa o CD-ROM
SDL_INIT_EVENTTHREAD – inicializa a manipulação de threads
SDL_INIT EVERYTHING – inicializa tudo
SDL_INIT_JOYSTICK – inicializa joysticks
SDL_INIT_TIMER – inicializa o timer
SDL_INIT_VIDEO – inicializa o vídeo

Eles devem ser colocados como parâmetros separados por | (“ou” lógico).

Para inicializar áudio, vídeo e timer, por exemplo, deve usar:

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

- b) Como além de criar uma tela nós também iremos manipulá-la em seguida, temos de criar uma variável do tipo ponteiro para essa **superfície** da **tela**. A variável que a tela usa é `SDL_Surface` (que saberemos, posteriormente, o significado). Portanto, crie na classe um ponteiro estático para a surface, por exemplo:

```
static SDL_Surface *screen;
```

Para inicializar a tela, utilizaremos a função:

```
SDL_SetVideoMode(int width, int height, int bpp, Uint32 flags).
```

O retorno da função é um ponteiro para `SDL_Surface` (a tela). Os parâmetros são os seguintes:

- int width – a largura da tela
- int height – a altura da tela
- int bpp – a taxa de bits por pixel a ser utilizada. Pode ser 8, 15, 16, 24 e 32. Quanto maior, mais lento.
- Uint32 flags – as flags da tela. Podem ser vários, separados por “|”. Os modos mais importantes são:
 - SDL_SWSURFACE – a tela é armazenada na memória do sistema;
 - SDL_HWSURFACE – a tela é armazenada na memória de vídeo;
 - SDL_FULLSCREEN – a tela fica em tela cheia (cuidado! Se não houver mecanismo para sair do jogo terá de usar Alt+TAB).
 - SDL_DOUBLEBUF – Ativa a renderização no front e no backbuffer.

Assim, para inicializar uma tela de 800x600 pixels, 32 bpp e tela armazenada na memória de software, use:

```
screen = SDL_SetVideoMode(800, 600, 32, SDL_SWSURFACE);
```

- c) Por fim, é possível mudar o nome da janela utilizando a função:

```
SDL_WM_SetCaption(const char* title, const char* icon);
```

Essa função recebe 2 argumentos:

- title é a string que será mostrada no título da janela;

- icon é a string que será mostrada quando a janela estiver minimizada.

Para mostrar o título Trabalho 1 – <matricula>, por exemplo, use:

```
SDL_WM_SetCaption("Trabalho 1 - <matricula>", "Trabalho 1 - <matricula>");
```

```
$ getScreen() : SDL_Surface*
```

Retorna a tela (pode ser necessário acessá-la diretamente)

```
$ loadImage( arquivo : std::string) : SDL_Surface *
```

- a) Primeiramente precisamos entender o conceito de *surfaces* (superfícies). *Surfaces* são superfícies de imagens de um certo tamanho e que contém uma série de pixels a serem ou não mostrados. Uma tela é uma *surface*, pois contém uma série de pixels a serem mostrados. *Surfaces* podem ser “coladas” umas em cima das outras, como películas.

As imagens que aparecerão na tela – como backgrounds, sprites de personagens, elementos de display – todos são *surfaces* que serão coladas na *surface* da tela, que por sua vez será mostrada. Portanto, uma imagem, na visão do SDL, é uma variável do tipo `SDL_Surface`.

Para carregar uma imagem na tela, utilizaremos a função:

```
IMG_Load(const char* file),
```

que está contida na biblioteca `SDL_image.h`. A função retorna um ponteiro para uma `SDL_Surface` e o parâmetro é o nome do arquivo da imagem, que podem ser do formato BMP, GIF, JPG, LBM, PCX, PNG, PNM, TGA e TIF.

Dessa forma, para carregar a imagem contida no arquivo passado como parâmetro, utiliza-se o seguinte código:

```
SDL_Surface *surface = 0;
surface = IMG_Load(arquivo.c_str());
```

- b) Nessa hora, a imagem contida no arquivo já está carregada na *surface*. Porém, não necessariamente essa *surface* está com seu formato otimizado para exibição na tela, e, nesse caso, exibir essas imagens pode demorar até 10 vezes mais do que se essas *surfaces* estivessem no formato da tela. Para resolver isso, utilizaremos as funções:

```
SDL_Surface * SDL_DisplayFormatAlpha(SDL_Surface* surface)
SDL_Surface * SDL_DisplayFormat(SDL_Surface* surface)
```

Essas funções recebem como argumento uma `SDL_Surface` e retornam essa mesma *surface* no formato da tela, ou seja, otimizado para exibição. A 1ª faz isso mantendo o canal alpha (transparência) da imagem (se houver).

E como saber se a *surface* tem canal alpha? Para isso, vamos acessar o parâmetro `format` da *surface*:

`surface->format->Amask`

Se esse parâmetro for diferente de 0, a surface tem transparência (e canal alpha). Senão, a surface é opaca.

Precisaremos de uma `SDL_Surface` auxiliar para fazer a conversão. Logo: Teste se a surface carregada tem alpha.

Utilize a conversão correta na surface e coloque o resultado na surface auxiliar.

Libere a memória da surface original, não otimizada, através da função:

```
SDL_FreeSurface(SDL_Surface* surface)
```

Por fim, retorne a surface carregada e convertida.

```
$ renderSurface( surface : SDL_Surface*, clip : SDL_Rect* = NULL,  
                 dst : SDL_Rect* = NULL) : void
```

Para fazer com que uma surface seja mostrada na tela, você deve “colar” (blit) a superfície da imagem na tela. Para isso, utilizaremos a função `SDL_BlitSurface(SDL_Surface* src, SDL_Rect* srcrect, SDL_Surface* dst, SDL_Rect* dstrect)`;

Os parâmetros são:

- `SDL_Surface *src` – a superfície que será colada,
- `SDL_Rect *srcrect` – o retângulo da superfície a ser colado. Caso seja `NULL`, será colada a superfície inteira,
- `SDL_Surface *dst` – a superfície destino (no caso, a tela, ou seja, `SDL_Surface *screen`),
- `SDL_Rect *dstrect` – retângulo onde a superfície será colada. Caso seja `NULL`, será colada na posição 0, 0.

Nessa função, existe um novo tipo de variável – a `SDL_Rect`. Trata-se de uma struct que representa um retângulo, ou seja, sua posição x, y, largura (width) e altura (height). É importante para determinar onde a superfície será colada (ou seja, sua posição na tela). Para, por exemplo, criar um retângulo na posição (120, 410), de largura 110 pixels e altura 300 pixels, bastaria fazer:

```
SDL_Rect retangulo;  
retangulo.x = 120; retangulo.y = 410;  
retangulo.w = 110; retangulo.h = 300;
```

Lembrando que, como o retângulo representará a posição da imagem, é possível obter a altura e largura de uma surface acessando seus parâmetros `width` e `height` (por exemplo, `fundo->w` e `fundo->h` retornam a largura e altura da superfície, respectivamente).

Para este método, basta repassar a tela e os parâmetros recebidos para a função de blit:

```
SDL_BlitSurface(surface, clip, screen, dst);
```

Repare que, usando os valores padrão NULL para os argumentos dos SDL_Rects* e pelo fato de que a SDLBase é a (única) dona da SDL_Surface * screen, simplificamos a renderização, de modo que, para mostrar uma imagem na posição (0,0), por exemplo, basta fazer:

```
SDLBase::renderSurface(surface);
```

```
$ atualizarTela() : void
```

É necessário “atualizar” as modificações na tela utilizando a função SDL_Flip(SDL_Surface* screen). Para isso, basta usar:

```
SDL_Flip(screen);
```

2. Classe Sprite: Mostrando imagens na tela

Sprite
<pre>- surface : SDL_Surface* - clipRect : SDL_Rect</pre>
<pre>+ Sprite(arquivo : string) + ~Sprite() + load(arquivo : string) : void + clip(x : int, y : int, w : int, h : int) : void + getClip() : SDL_Rect + getWidth() : int + getHeight() : int + render(x : int, y : int) : void</pre>

A classe Sprite usará a SDLBase para carregar e exibir as imagens. Não esqueça de incluir a SDLBase.h

A classe “sprite” possui 2 atributos:

- surface – contém a imagem propriamente dita
- clipRect – contém o seu retângulo de clip, que define qual parte da imagem será mostrada.

Os métodos da classe Sprite são os seguintes:

- ```
+ Sprite(arquivo: string) : void
```
- Construtor da classe. Recebe como parâmetro o caminho para o arquivo da imagem.
  - Inicializa o valor da surface como NULL.
  - Em seguida, ela carrega a imagem utilizando o método load.

```

+ ~Sprite() : void
 • Destrutor da classe. Confere se ja tem uma surface carregada (se a surface não é nula). Se tiver alguma, descarrega (usando SDL_FreeSurface(surface));
+ load(arquivo : string) : void
 • Confere se ja tem uma surface carregada (se a surface não é nula). Se tiver alguma, descarrega (usando SDL_FreeSurface(surface));
 • Carrega a surface utilizando SDLBase::loadImage(arquivo);
 • Define x e y do clipRect como 0;
 • Define w e h do clipRect como surface->w e surface->h;
+ clip(x : int, y : int, w : int, h : int) : void
 • Define os valores retângulo de clipping (clipRect)
+ getClip() : SDL_Rect
 • Retorna o retângulo de clipping
+ render(x : int, y : int) : void
 • Declara um SDL_Rect dst com os valores x, y, 0, 0.
 • Chama o método SDLBase::renderSurface(surface, &clipRect, &dst).
 OBS.: A função SDL_BlitSurface zera os valores x e y de um SDL_Rect caso eles sejam negativos. Se for necessário manter um clipRect com valores negativos será necessário copiá-lo para um SDL_Rect auxiliar antes de passar para a SDL.
+ getWidth() : int
 • Retorna o valor surface->w;
+ getHeight() : int
 • Retorna o valor surface->h;

```

### 3. Classe GameManager: Game Loop

| GameManager                                     |
|-------------------------------------------------|
| - bg : Sprite*                                  |
| + GameManager()<br>+ ~GameManager()<br>+ run(); |

A classe GameManager controlará o fluxo e toda lógica específica do jogo. Tem como atributos privados tudo que será mostrado / usado no jogo. Logo, a classe deve incluir os headers de todos os objetos usados.

Para este trabalho, o GameManager terá apenas um atributo:

- bg – Será a Sprite utilizada para mostrar o fundo (ou BackGround) do jogo.

Os métodos da classe GameManager são os seguintes:

```

+ GameManager() : void
 • Construtor da classe, deve inicializar todos os objetos do GameManager.
 • No caso, carregar o bg.

```

+ ~GameManager() : void

- Destrutor da classe. Deve destruir todos os objetos e desalocar toda a memória utilizada pelo GameManager.
- No caso, delete(bg);

+ run() : void

Em um jogo, tudo ocorre dentro de um loop, sendo que cada *frame* significa uma iteração desse loop. Faremos um loop que termina apenas quando for requisitado. Para isso, usaremos a função `SDL_QuitRequested()`, que retorna true caso algum evento de fechar a janela tenha acontecido (como clicar no "X", alt+F4, kill, etc).

Como o objetivo no momento não é aprofundar muito, segue um conceito genérico: dentro do loop de jogo temos basicamente três partes:

1. Os dados de input são recebidos e processados.
2. Os objetos na tela têm suas posições/física alteradas (chamaremos de UPDATE);
3. Os objetos são desenhados na tela (chamaremos de RENDER).

Lembrando que qualquer carregamento (seja de imagem, som, video, etc) deve ser feito ANTES do loop – dentro do loop não vamos carregar objetos. Vale observar o que foi aprendido nas ultimas aulas teóricas a respeito do loop de jogo.

Ao final de cada loop, devemos atualizar todas as modificações na tela. Dessa forma, o loop do jogo funcionará da seguinte forma:

```
while(!SDL_QuitRequested())
{
 /* Captura de Input */

 /* Atualização dos objetos */

 /* Todos os comandos de renderização */
 bg->render(0,0);

 /* Atualizar a tela */
 SDLBase::atualizarTela();
}
```

#### 4. Entry Point: A função main

A função main deve ser utilizada o mínimo possível em um jogo, apenas como ponto de entrada. Ou seja, basicamente, ela deve construir o GameManager, rodar o jogo e destruí-lo.

IMPORTANTE: A SDL exige que a função main seja declarada com os parâmetros argc e argv:

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