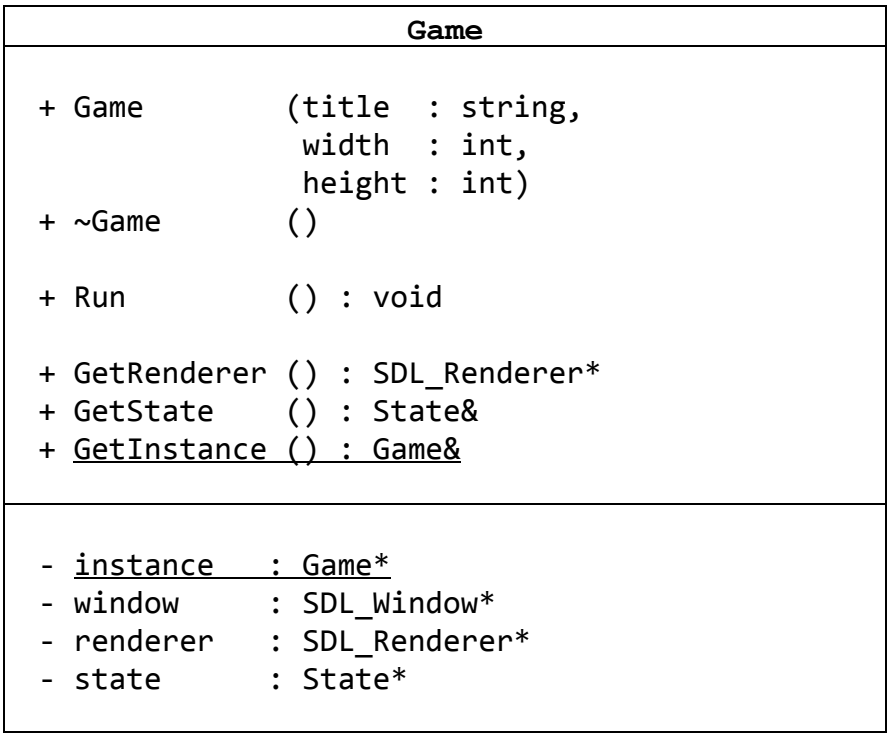
Trabalho 1 – Game Loop e Sprite

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 e SDL_image) que precisaremos para outras classes funcionarem. Ao longo do semestre, faremos alterações na classe para incluir mais features.

Note que ela mantem registro da própria instância. 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. No header, adicione os includes "SDL.h" e "SDL_image.h”. O primeiro é o cabeçalho para as funções principais da SDL, o segundo, para funções que carregam imagens do disco rígido. Usaremos ambos nesse trabalho.

Implementando os métodos da classe:

`> 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 != NULL`). 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. Convem 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.

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 NULL. 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_image (`IMG_Quit`), destrói o renderizador e a janela (`SDL_DestroyRenderer`, `SDL_DestroyWindow`), e, finalmente, encerra a SDL (`SDL_Quit`).

`> GetInstance () : Game*`

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

`> GetState () : State&`

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

`> GetRenderer () : SDL_Renderer*`

Retorna o membro `renderer`. Será usada principalmente pela classe Sprite.

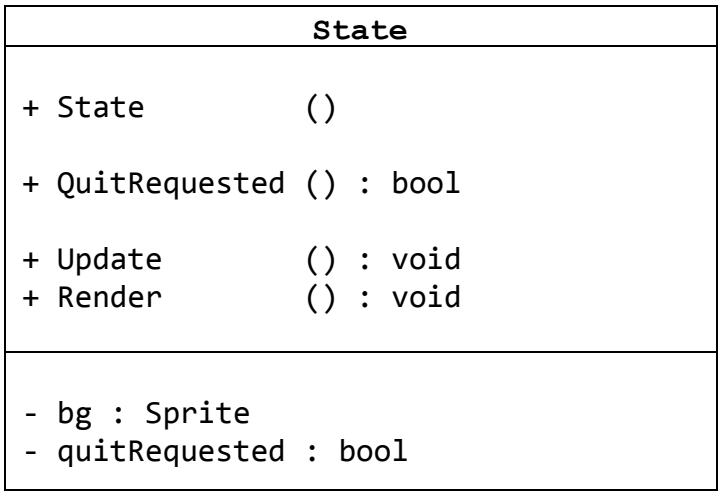
`> Run () : void`

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. Os dados de input são recebidos e processados;
2. Os objetos tem seus respectivos estados (posição, HP…) atualizados;
3. Os objetos são desenhados na tela.

O passo 1 será implementado em trabalhos futuros, quando criarmos um sistema separado para administrar inputs. Já os passos 2 e 3, 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.

`> Update (dt : float) : void`

Update trata da etapa 2 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 3 de Game::Run, a renderização do estado do jogo. Isso inclui entidades, cenários, HUD, entre outros. Para esse trabalho, renderize o fundo (bg) de forma a preencher a tela.

`> 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 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, w : int, h : int) : void   + Render (x : int, y : int) : void   + GetWidth () : int   + GetHeight () : int   + IsOpen () : bool
- texture : SDL_Texture*   - width: int   - height: int   - clipRect : SDL_Rect

A classe `Sprite` encapsula o carregamento e uso de SDL_Textures, o tipo da SDL que contem uma imagem carregada do dis 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 NULL (imagem não carregada).

`> Sprite (file : string)`

Seta texture como NULL. 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 NULL. É uma das causas mais comuns de crashes nos trabalhos da disciplina. Novamente, `SDL_GetError` 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 NULL 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. Não use as dimensões completas da imagem, pois causará problemas em trabalhos futuros.

`> 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/ocean.jpg*, presente no zip de resources para os trabalhos da disciplina, no Moodle.

4. 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. **A janela deve ter as dimensões 1024x600, 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)
```