Introdução ao Desenvolvimento de Jogos – Turma A
Professora: Carla Denise Castanho (carlacastanho@cic.unb.br)
Monitores: Davi "Doom" Diniz (khenstot@gmail.com)
Lenonardo Guilherme (leonardo.guilherme@gmail.com)
Luiggi Monteiro Reffatti (lmreffatti@gmail.com)

Trabalho 4 – Texto, Audio, Timer, StateManager e States

1. Texto: Escrevendo na tela

Text
- font : TTF_Font* - textSurface : SDL_Surface* - fontFile : string - text : string - style : int - ptsize : int - color : SDL_Color - box : SDL_Rect
+ Text(fontFile : string, text : string, ptsize : int = 10, style : int = 0, color : SDL_Color = 0, x : int = 0, y : int = 0) + ~Text() + render(camera : int = 0, cameraY : int = 0) : void + setPos(x : int, y : int) : void + setText (text : string) : void + setColor(color : SDL_Color) : void + setStyle (style : int) : void + setSize(ptsize : int) : void

Uma das formas de escrever algo na tela é usando a biblioteca SDL_ttf. Através de arquivos de fonte no formato *.ttf (TrueType font), é possível renderizar uma surface contendo um texto e blitá-la na tela.

```
#include <SDL/SDL_ttf.h>
```

Para usá-la, basta usar as seguintes funções:

int TTF_Init() - inicializa a biblioteca
void TTF_Quit() - finaliza a biblioteca
TTF_Font ***TTF_OpenFont**(**const char** *file, **int** ptsize) **–** abre uma fonte *.ttf. ptsize normalmente é 16.
void TTF_SetFontStyle(TTF_Font *font, **int** style) – determina o estilo de uma fonte. Pode ser:
TTF_STYLE_BOLD, *TTF_STYLE_ITALIC*, TTF_STYLE_UNDERLINE, TTF_STYLE_NORMAL
caso queira colocar mais de um estilo, separá-los com o "ou" lógico: "|".

`SDL_Surface *TTF_RenderText_Solid(TTF_Font *font, const char *text, SDL_Color fg)` – retorna uma surface com uma fonte *font de um texto *text na cor fg rederizada.

Atenção: a variável SDL_Color é representada por uma struct contendo valores r, g e b, que são respectivamente vermelho, verde e azul. Se {r, g, b} = {0, 0, 0}, temos a cor preta.

Obs.: existem outras duas funções que fazem o mesmo:
TTF_RenderText_Shaded()
TTF_RenderText_Blended()
A diferença entre elas é o modo de renderização. Aplica-se anti-aliasing e a fonte fica mais suave. Recomendo utilizá-las.

`int TTF_SizeText(TTF_Font *font, const char *text, int *w, int *h)`- determina o tamanho da fonte. Um texto deve ser fornecido.

Modificações na SDLBase

- Adicione a SDL_ttf na lista de includes (não esqueça de linká-la / adicionar no linker).
- No construtor, Logo após inicializar a tela, inicialize a biblioteca. Não esqueça de finalizá-la no destrutor.
- Aproveitando que você está no construtor da SDLBase, altere o Caption da janela para trabalho 4.

A abstração Texto deve ser semelhante à abstração Sprite. Significa que, quando você quiser escrever algo na tela, ao invés de aplicar todos os comandos da SDL_ttf, utilizará um objeto da classe Text para isso.

Os atributos da classe são:

- `font : TTF_Font*`
  A fonte utilizada pelo texto.
- `textSurface : SDL_Surface*`
  A surface que terá o texto.
- `fontFile : string`
  O arquivo de onde foi carregada a fonte.
- `text : string`
  O texto que sera mostrado na tela.
- `style : int`
  O estilo aplicado no texto
- `ptsize : int`
  O tamanho, em pt, do texto
- `color : SDL_Color`
  A cor do texto
- `box : SDL_Rect`

A caixa do texto (posição x,y e tamanho w,h)

Note que se parece com a classe Sprite. A diferença é que a surface da Sprite é gerada com base em uma imagem, e a surface da Text é gerada com base no texto a ser apresentado, no arquivo da fonte, na cor e no seu style, para depois ser desenhada na tela.

Os métodos da classe são:

```
+ Text(fontFile : string, text : string, ptsize : int = 10, style : int = 0, color : SDL_Color = 0, x : int = 0, y : int = 0)
```

- Recebe o arquivo da fonte fontFile, o seu texto, posições x e y, o style, as cores red, green e blue, e o tamanho.Sendo assim, ele:
- atribui todos os valores passados por parâmetros aos atributos correspondentes da classe
- abre a fonte (usando TTF_OpenFont) com base no nome do arquivo passado
- inicializa a variavel SDL_color com base nos atributos,
- cria a surface do texto usando TTF_RenderText_Blended (ou Solid ou Shaded);
- determina os valores da rect;

```
+ ~Text()
```

- usa TTF_CloseFont para liberar o espaço da variavel font;
- libera a surface com SDL_FreeSurface

```
+ render(camera : int = 0, cameraY : int = 0) : void
```

- manda desenhar a surface na tela;

```
+ setPos(x : int, y : int)    : void
```

- Altera os valores de x e y de box

```
+ setText (text : string)     : void
```

- libera o espaço da textSurface atual;
- troca o valor do atributo text
- renderiza a surface novamente com TTF_RenderText;
- atualiza os valores w e h do rect;

```
+ setStyle (style : int)      : void
```

- libera o espaço da surface atual;
- troca o valor da style usando TTF_SetFontStyle;
- renderiza a surface novamente;
- atualiza os valores w e h do rect;

Alguns outros métodos possíveis são ChangeColor e ChangeSize. Deduza o funcionamento destes métodos e implemente-os.

```
+ setColor(color : SDL_Color) : void
+ setSize(ptsize : int)       : void
```

2. Áudio: Música e efeitos sonoros (sfx)

Audio
- fileName : string - sound : Mix_Chunk* - music : Mix_Music* - channel : int
+ Audio(fileName : string, type : int) + ~Audio(); + Play(n : int) : void + Stop() : void

Agora vamos dar um pouco mais de vida ao “jogo” e colocar música e efeitos sonoros! Para isso, utilizaremos a biblioteca SDL_mixer.

Modificações na SDLBase

- `#include <SDL/SDL_mixer.h>`.
- Não esqueça de adicionar a SDL_mixer ao linker.
- No construtor, Logo após inicializar a TTF, inicialize a biblioteca. Para isso, use:
  `int Mix_OpenAudio(int frequency, Uint16 format, int channels, int chunksize)`
  onde:
  frequency – frequência da música. Usamos MIX_DEFAULT_FREQUENCY.
  format – formato da música. Usamos MIX_DEFAULT_FORMAT.
  channels – numero de canais disponíveis. Usamos MIX_DEFAULT_CHANNELS.
  chunksize – bytes utilizados para output. Usamos 1024.

  Obs: é importante que o chunksize esteja em 1024 para evitar um atraso na execução do som.

  Os atributos da classe são:

`- fileName : string`
Nome do arquivo de audio que foi carregado
`- sound : Mix_Chunk*`
Efeito sonoro. Deve ser um .wav
`- music : Mix_Music*`
Música. Entre os formatos aceitos, está o .mp3 e o .ogg. Recomendamos o uso de .ogg
`- channel : int`
Caso seja um efeito sonoro, o canal usado para tocar o som é armazenado aqui.

Os métodos da classe são:

`+ Audio(fileName : string, type : int)`

- Recebe o nome do arquivo de audio e o seu tipo (pode ser chunk para tipo = 0, e music para tipo = 1). Carrega em *som* ou *musica*, dependendo do valor de *tipo*. A variável não carregada deve ser NULL.
  Para carregar efeitos sonoros e músicas, utilizamos:
  - `Mix_Music *Mix_LoadMUS(char *file),`
    sendo *file o nome do arquivo da música
  - `Mix_Chunk *Mix_LoadWAV(char *file);`
    sendo *file o nome do arquivo do som.

- Obs.: Uma abordagem mais interessante seria detectar a extensão do arquivo na string e escolher o tipo!

`+ ~Audio()`

- Liberam o espaço das variáveis Mix_Music e Mix_Chunk, usando as funções: Mix_FreeMusic() e Mix_FreeChunk(), respectivamente.

`+ Play(n : int) : void`

- Toca o som ou a música n vezes.
  Para tocar uma música, basta usar :
  `int Mix_PlayMusic(Mix_Music *music, int loops)`
  onde: music – música a ser tocada
  loops – numero de loops. -1 para tocar infinitamente.

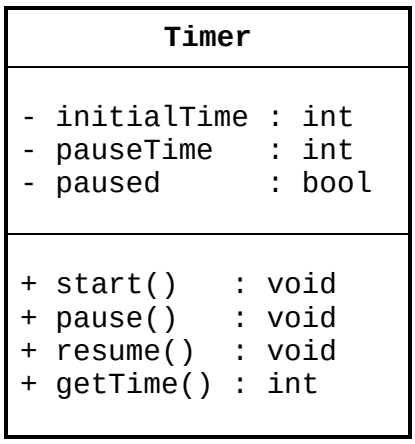
- Para tocar um som, basta usar:
  `int Mix_PlayChannel(int channel, Mix_Chunk *chunk, int loops)`
  onde:
  channel - canal onde o som será tocado. -1 para tocar em um canal livre.
  chunk – o som a ser tocado.
  Loops – numero de vezes a ser tocado. -1 para tocar infinitamente. Obs.: 0 toca uma vez.
  IMPORTANTE: O método retorna o número do canal alocado para tocar o som. Guarde esse número na variável channel.

`+ Stop() : void`

- Para o som ou a música. Para isso, usamos:
  - Mix_FadeOutMusic(int fade)
    Onde fade é o tempo de fade out da música.
  - Mix_HaltChannel(int channel)
    Onde channel é o número do canal que está tocando.

Atenção: esta é só uma sugestão para a abstração de Audio que junta som e música. Caso queira fazer duas abstrações, uma **Music** e outra **Sound**, sinta-se livre para isso.

3. Classe Timer: Cooldown!



A função do Timer é estabelecer um contador de tempo que pode ser inicializado e pausado, pois isso tem diversas aplicações nos jogos.

Os atributos da classe são:

`- initialTime : int`
Armazena o tempo de quando o Timer foi iniciado
`- pauseTime : int`
Armazena o tempo de quando o timer foi pausado
`- paused : bool`
Indica se o timer está parado.

Os métodos da classe são:

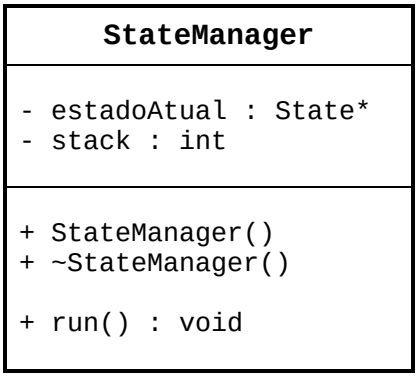
`+ start() : void`
Define o valor da initialTime como SDL_GetTicks(), marcando o tempo em que o método start() foi chamado. Além disso, define paused como false.

`+ pause() : void`
Define o valor pauseTime como SDL_GetTicks(), marcando o tempo em que o método stop() foi chamado. Além disso, atribui true a paused.

`+ resume() : void`
Calcula a quanto tempo o timer está pausado (subtraindo pauseTime de initialTime). Então, pega o tempo atual e subtrai dele este tempo. Este valor (tempo atual – tempo pausado) deve ser atribuído ao initiaTime. Além disso, atribuímos false ao atributo pause.

`+ getTime() : int`
o método getTime() depende do valor booleano de pause:
- Se for verdadeiro, retorna a diferença entre pauseTime e inicialTime
- Se for falso, retorna a diferença entre o SDL_GetTicks() atual e o initialTime.

4. Classe StateManager: Gerenciador de cenas do jogo



A idéia por trás da State Manager é fazer o controle das várias cenas diferentes de um jogo. Por exemplo: Menu inicial, Fase 1, Fase 2, Fase 3, Tela de encerramento (vitória e/ou derrota).

Os atributos da classe são:

`- estadoAtual : State*`

Armazena o estado atual do jogo.

`- stack : int`

Armazena valores que devem ser passados de um estado a outro.

Os métodos da classe são:

`+ StateManager()`

O constructor deve inicializar tudo que é necessário para o jogo funcionar. Ou seja:

- Inicializar a SDL (através da SDLBase),
- Calcular o primeiro dt.
- Inicializar estado atual com o primeiro estado do jogo – e também carregá-lo o com o método load().
- Inicializar quaisquer outras bibliotecas necessárias para o jogo (como chamar a função srand()).
- Dar o primeiro update na InputManager.
- Inicializar a stack vazia.

`+ ~StateManager()`

O Destrutor deve finalizar tudo que precise ser finalizado. Ou seja

- Finalizar a SDL (através da SDLBase – crie o método caso ainda não esteja criado),
- Descarrgar e destruir o estado atual.
- Finalizar quaisquer outras bibliotecas necessárias para o jogo

`+ run() : void`

No método Run() temos, basicamente, o game loop. A cada loop será calculado um novo valor de timer e dt, e serão chamados os métodos update() e render() do estadoAtual, atualizaTela() e calculado o dt do frame.

A forma mais simples de se fazer um gerenciador de estados é usando o valor de retorno do update() e do unload() de cada estado. Ou seja, criar um mecanismo onde, por exemplo, ao retornar 0, nada ocorre, ao retornar 1 ele vai para outro estado específico (como o *options*, por exemplo), ao retornar 2 ele muda para outro estado específico (como o jogo propriamente dito, por exemplo), e assim em diante.

Assim, no método Run() do Game Manager, deve-se fazer um switch do valor de estadoAtual->Update(), para que ações sejam tomadas de acordo com o seu valor.

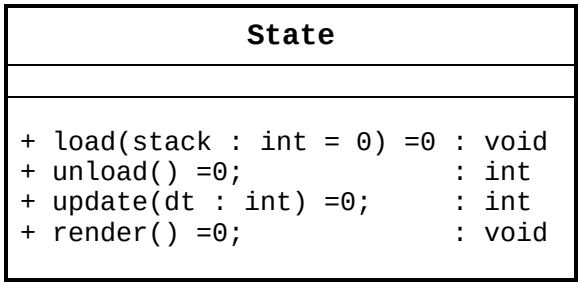
Exemplo:

```
while(quit == false )
{
	/* Calculo de Timer e dt */

	/* Atualização da InputManager – deve ser chamado somente aqui */

	// STATE MANAGER
	switch(estadoAtual->update(dt))
	{
		case STATE:
			// fazer nada
		break;
		case STATEMENU:
			stack = estadoAtual->unload();
			estadoAtual = new StateMenu();
			estadoAtual->load(stack);
		break;
		case STATEGAME:
			stack = estadoAtual->unload();
			estadoAtual = new StateGame();
			estadoAtual->load(stack);
		break;
		case STATEQUIT:
			quit = true;
		break;
	}

	/* Atualização dos objetos na tela */

	/* Delay */
}
```

Assim, se dentro do Update() do SplashState, por exemplo, houver uma linha “return 1” para algum evento específico (ao apertar ENTER, por exemplo), o estadoAtual será trocado para o LevelState. No próximo loop, serão os métodos Update() e o Render() do LevelState que estarão sendo executados – assim, o estado será trocado.

Vale lembrar que este método é extremamente parecido com o run do GameManager. Você pode se basear nele na hora de criar este.

5. Classe State: Menu, Jogo, Tela de fim...



A classe State é uma classe abstrata, ou seja, é uma classe cujos métodos não foram implementados, mas que deverão ser implementados por classes filhas. Isso porque, no GameManager, existe uma instância da classe *State*, mas podem existir diversos tipos de estados que agem da mesma forma.

Mas afinal, o que é um estado? Imagine um jogo. Primeiro aparece a tela de título (splash state) ela é um estado do jogo, com características próprias. Essas características envolvem, normalmente, uma imagem de fundo, uma imagem com o título de jogo e um menu de escolha de *New Game*, *Load Game* ou *Options*. Ao escolher o menu *Options*, por exemplo, é apresentado um outro estado, no qual há um menu com uma série de opções a serem escolhidas. Se escolher *New Game*, provavelmente aparecerá a primeira fase, com uma série de elementos que a caracterizam – como aspectos da jogabilidade, física, cenários, AIs, etc. Observe que os estados se diferem consideravelmente, mas todos eles basicamente contém uma série de entidades e pedaços de código.

Um *State* contém, basicamente, quatro métodos:

`+ load(stack : int = 0) =0 : void`

Método a ser chamado para carregar o estado – todos os objetos necessários para o estado serão criados, instanciados e inicializados aqui e somente aqui.

Caso seja necessário algum parâmetro para inicializer o estado deve ser passado na stack.

`+ unload() =0; : int`

Método a ser chamado ao descarregar o estado – todos os objetos serão removidos para liberar memória.

Caso este estado quera passar algum dado para o próximo estado, deve retornálo aqui, colocando na stack.

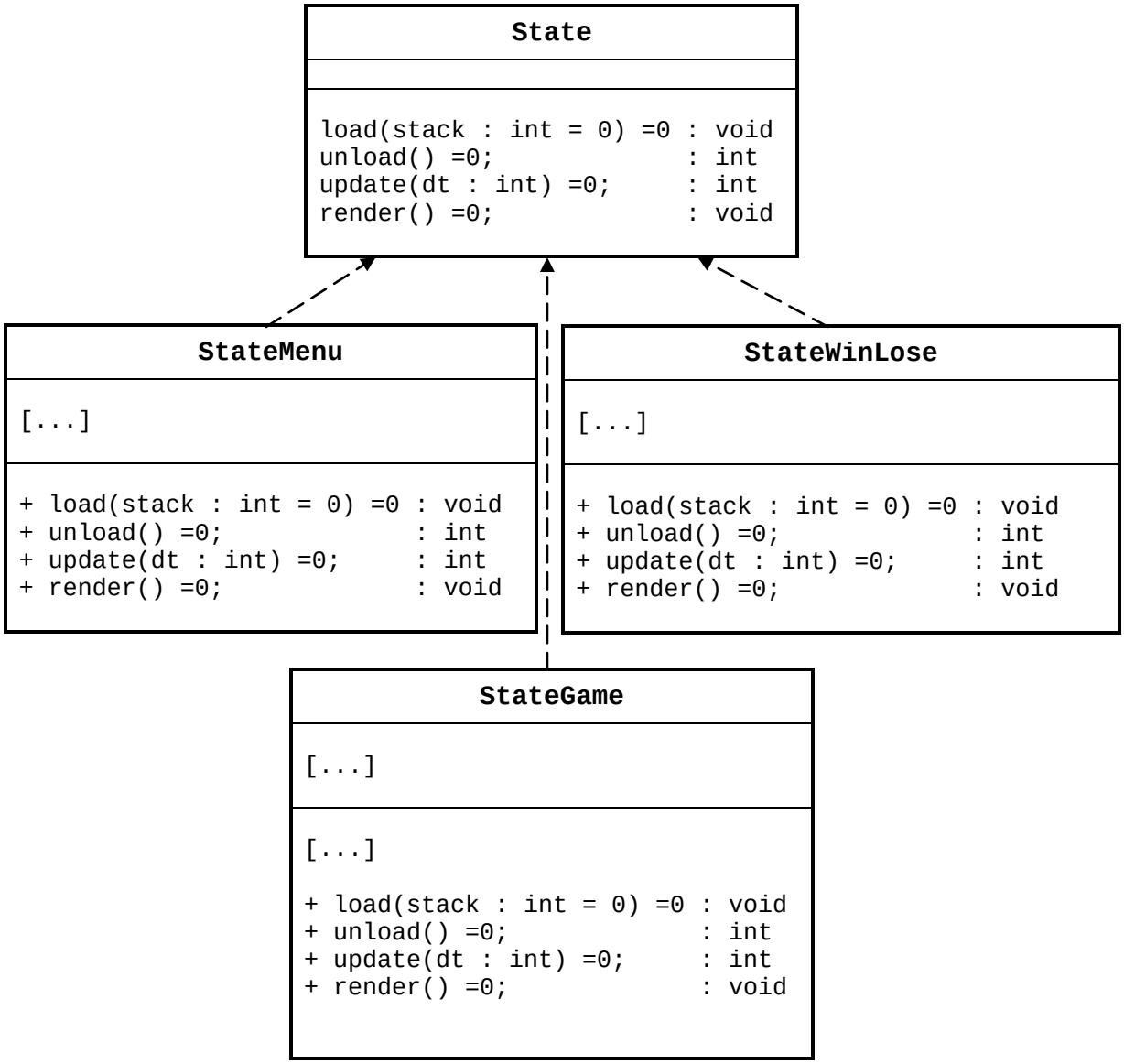
`+ update(dt : int) =0; : int`

Método contendo toda a parte do código envolvendo jogabilidade, física, mecânicas e etc. O parâmetro é o dt, já que tudo é calculado com base no dt. Retorna um inteiro a ser utilizado no gerenciador de estados

`+ render() =0; : void`

Método que conterá todas as funções de renderização de objetos na tela.

7. Demonstração da Game Engine



Agora que a Game Engine está pronta, altere o jogo da Nave para que use tudo que foi implementado neste trabalho. Para isso:

- Vamos adicionar Música, Texto e Timer ao GameManager:
  - Defina 2 variáveis do tipo Audio: uma música e um sfx.
  - No construtor da classe, carregue a musica com uma música e o sfx com um som de explosão.
  - Agora coloque a música para tocar com repeat, no construtor mesmo.
  - Quando a ship colidir com os planetas ou com o UFO, toque o sfx.
  - Adicione uma variável (ou mais de uma) do tipo TEXT, e mostre na tela as instruções (comandos disponíveis) de como jogar.
  - Além disso, adicione um cooldown à criação de planetas. Ou seja, crie uma variável do tipo timer e, usando-a, só permita que um novo planeta seja criado pelo UFO a cada X milisegundos.

- Transforme a antiga gameManager em um stateGame (filho da State). Para isso será necessário refatorar a classe. Comece renomeando-a para stateGame. Então:
    - Toda a inicialização que é feita no construtor do GameManager deve ser feita no load do StateGame.
    - Adicione uma variável ao state game, chamda returnState. Essa variável dever ser atualizada e retornada todo frame, no update.
    - Inicialize a stateReturn com 0 (ou o valor que signifique “não trocar de estado”).
    - Remova completamente o método Run (certifique-se que o StateManager implementa suas tarefas corretamente)
    - Atualize o returnState de acordo com o input (comando para voltar para o menu e comando para sair do jogo).
    - Em checkCollision, atualize o returnState se o jogo acabou (use um valor específico para a tela de game over).
    - No método unload() retorne valores diferentes se foi a Nave ou o UFO quem ganhou.

- Na main, inicialize e rode a stateManager (e não a antiga Game Manager)

- Crie um StateMenu (filho da State), que tenha no mínimo:
    - Um fundo, toque uma música e tenha um texto indicando qual comando deve ser dado para iniciar o jogo

- Crie um StateWinLose (filho da State), que tenha no mínimo:
    - Um fundo, uma música e um texto (diferentes para os casos de vitória e “derrota”).
    - Para definir qual estado exibir use o valor recebido na stack, retornado pelo unload do stadeGame.
    - Deve ter também um texto indicando qual comando deve ser utilizado para jogar novamente.

IMPORTANTE: No State, o stack foi definido como inteiro somente para facilitar a programação e explicação. Dependendo de como seus estados se comunicam, pode ser necessário passar muitos parâmetros entre os estados.
Por isso, vocês são livres para criar a estrutura que precisarem para usar de stack. Eu sugiro uma struct bem definida, ou uma union, ou ainda um void* (castado para uma struct* sempre que necessário).

Agora vocês tem a Game Engine 100% pronta. E o que é código específico fica bem definido e separado da Engine, ou seja, os States. Se vocês quiserem limpar sua Engine para começar a trabalhar no jogo final, basta remover os states e criar seus próprios. Easy as a pie!

Happy Gaming e Happy developing!