

Trabalho 7 – Game States, Audio e Texto

1. Refatoração - Game States

Quantos jogos que você conhece te largam direto na primeira fase, sem uma tela de título ou uma tela inicial? Provavelmente, não muitos. Em geral, os jogos possuem mais do que a lógica do gameplay normal: possuem telas de título, de Game Over, de menus, mini-games...

É possível fazer isso numa classe só - *possível*, não *recomendável*. O resultado seria um código enorme, cheio de ifs, cujos trechos não tem muita relação entre si. Como organizar isso, então? A verdade é que você já conhece a solução: O conceito de Game State, que é o que implementamos em State.

Originalmente, apresentamos State como uma forma de separar a lógica específica do jogo da lógica da engine, uma classe com interface genérica que a engine pudesse manipular. Essa mesma característica permite que Game troque o State em execução, para permitir comportamentos diferentes.

Assim, um State passa a ser uma parte do jogo com lógica própria. Imagine uma tela de título de um jogo hipotético: Ela mostra um background, um logo, opções como New Game, Continue e Options, e um cursor. Não estamos preocupados com a colisão de entidades, por exemplo; talvez nem haja entidades. Apenas movemos o cursor para cima e para baixo, esperando uma escolha. Esse é o primeiro estado.

Aperte New Game. Você assistirá a uma cutscene inicial. Este é um estado que mostra na tela determinadas imagens e trechos de texto predefinidos. Quando a cutscene acabar, você estará no estado do jogo. O jogo tem preocupações com a atualização e renderização de entidades quaisquer, e tem condições de vitória e derrota.

Com essa distinção em mente, vamos examinar que mudanças são necessárias na nossa engine. Atualmente, Game tem um ponteiro pra State. Mudaremos isso para uma pilha de ponteiros para State. Acontece que nossa classe State atual tem um comportamento bem definido. Precisaremos transformá-la numa interface para essa mudança na arquitetura.

Vamos resolver isso primeiro, para depois vermos como Game vai ficar. Renomeie sua classe State atual para StageState. Trataremos das alterações a serem feitas nela na seção 1c.

1a. State

State
<pre>+ State() + ~State() : virtual + Update(dt : float) : void, virtual pura + Render() : void, virtual pura + Pause () : void, virtual pura + Resume () : void, virtual pura + AddObject(object : GameObject*) : void, virtual + PopRequested() : bool + QuitRequested() : bool # UpdateArray(dt : float) : void, virtual # RenderArray() : void, virtual</pre>
<pre># popRequested : bool # quitRequested : bool # objectArray : std::vector<std::unique_ptr<GameObject>></pre>

State implementa uma interface básica de comportamento de um estado (Update e Render, como antes), além de uma interface de comunicação de com a engine (PopRequested, para a deleção do estado, QuitRequested para o encerramento do jogo).

> State()

Apenas inicializa os membros (as flags).

> AddObject(object : GameObject*) : void, virtual

Copie a função de StageState para cá.

- > Pause () : void, virtual pura
- > Resume () : void, virtual pura

Essas funções são chamadas por Game quando há uma mudança na pilha de estados. Pause especifica o comportamento de um estado quando um outro é empilhado por cima dele, e Resume, quando o estado acima é desempilhado.

Isso pode ser usado para várias coisas. Para controlar sons e música, que implementaremos em instantes; para chamar rotinas de cleanup ou recarregamento de recursos; para desfazer e refazer bindings com classes que usam o padrão Observer; entre outros.

- > PopRequested() : bool
- > QuitRequested() : bool

Retornam a flag correspondente. Essas funções são usadas por Game para controlar a troca de estados ou o fim do jogo.

- > UpdateArray(dt : float) : void, virtual
- > RenderArray() : void, virtual

Aqui, colocaremos os loops de atualização e renderização de objetos no vetor que tínhamos na antiga Game. Esses loops não costumam mudar muito, então nos convem já tê-los prontos. Note que as funções são virtuais, caso seja necessário mudá-las.

Com a nova interface estabelecida, vamos às mudanças em Game.

1b. Game

Game
+ Game(title : string, width : int, height : int) + ~Game() + <u>GetInstance()</u> : Game& + GetRenderer() : SDL_Renderer* + GetCurrentState() : State& + Push(state : State*) : void + Run() : void + GetDeltaTime() : float - CalculateDeltaTime() : void
- frameStart : int - dt : float - <u>instance</u> : Game* - storedState : State* - window : SDL_Window* - renderer : SDL_Renderer* - stateStack : std::stack<std::unique_ptr<State>>

Game implementa uma pilha de estados. Apenas o estado do topo é atualizado e renderizado, e em todo frame, checamos se ele quer ser desempilhado. Temos também uma função Push, que será usada por outros estados para empilhar um novo.

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

Em vez de inicializar state (que não existe mais), inicialize storedState como nullptr.

> ~Game()

Se existe um storedState não nulo, delete-o. Esvazie a pilha de estados. De resto, é o mesmo destrutor de antes.

```
> GetInstance() : Game&  
> GetRenderer() : SDL_Renderer*  
> GetCurrentState() : State&
```

As duas primeiras são as mesmas funções que já temos. A terceira função deve retornar uma referência para o estado no topo da pilha.

```
> Push(state : State*) : void
```

Push serve para colocar um estado no topo da pilha. No entanto, o estado não é colocado diretamente: Como a maioria dos pushes ocorrerão na chamada ao Update do estado atual (mid-frame), Push deve apenas reservar esse ponteiro para ser empilhado logo antes do frame seguinte. Isso é responsabilidade de...

```
> Run() : void
```

Run é o nosso main game loop, mas agora com a pilha de estados. Antes que o loop comece, você deve buscar o estado inicial do jogo: Este será Push-ado pela main e estará em storedState. Coloque-o na pilha, e sete o ponteiro de volta para nullptr. Se não houver um estado inicial, encerre o jogo.

O loop agora tem duas condições pra ser interrompido. A primeira é a pilha estar vazia, e a segunda, QuitRequested do estado atual. No corpo, execute as mesmas tarefas de antes, mas antes de chamar SDL_Delay, vamos gerenciar a pilha.

Primeiro, precisamos saber se o estado atual quer ser deletado - se sim, o desempilhamos e damos Resume no estado agora no topo da pilha. Depois, se storedState não for nulo, no topo da pilha é pausado, e empilhamos o estado guardado. Note que ambas as coisas podem acontecer num frame, e por isso, a ordem das operações é importante.

Na saída, lembre-se de limpar os Resources, como antes. O ideal seria fazer isso

```
> GetDeltaTime() : float  
> CalculateDeltaTime() : void
```

Mesmo cálculo e retorno do dt feito antes.

Voltemos agora à StageState, antiga State, para adaptar o jogo que tínhamos antes à arquitetura nova.

1c. StageState

StageState (herda de State)
<div>+ StageState() + ~StageState() + Update() : void + Render() : void + Pause() : void + Resume() : void</div>
<div>- bg : Sprite - tileSet : TileSet - tileMap : TileMap</div>

Não há muito o que dizer. StageState é a mesma classe de antes, com algumas funções e membros retirados, já que agora são herdados de State. Você deve se basear no seu código de antes, atentando para o fato de que, agora, podemos usar UpdateArray e RenderArray. Pause e Resume podem ser vazias.

Com StageState pronto, vá até a sua função main e a altere para ter um Game, dar push num StageState e executar o jogo. Se tudo estiver bem, o jogo vai funcionar do mesmo jeito de antes. Teste antes de continuar. Vamos agora aproveitar a arquitetura nova para fazer uma tela de título.

1d. TitleState

TitleState (herda de State)
<pre>+ TitleState() + Update() : void + Render() : void + Pause() : void + Resume() : void</pre>
<pre>- bg : Sprite</pre>

> TitleState()

Inicializa bg.

> Update(dt : float) : void

Deve responder a eventos de janela e a tecla ESC, casos em que devemos sair do jogo. Também devemos checar se a barra de espaço é pressionada, se sim, devemos criar um novo StageState e dar Push.

> Render() : void

Por enquanto, só renderize o bg.

Altere StageState para voltar à tela de título quando ESC for pressionada (e não sair do jogo). Execute o programa, alternando entre os estados.

1e. Mudanças em Resources (Extra)

A troca de estados estando implementada, é uma boa hora para lembrar que o nosso gerenciamento de recursos é um tanto falho. Afinal, se trocamos de fase várias vezes, passando por vários tilesets, vários sprites, assim que voltarmos pra tela de título, ainda teremos tudo na memória.

Não faz sentido manter na memória objetos que não serão mais usados, mas saber disso é o grande problema. Como mencionamos anteriormente, há uma série de fatores que podem ser levados em consideração. Usaremos o mais simples de todos, a contagem de referências

do `std::shared_ptr`.

Essa classe, assim como sua irmãzinha `unique_ptr`, está em `<memory>`. Ela permite que cópias do ponteiro sejam feitas, mas mantém, internamente, uma contagem do número de instâncias de `shared_ptr` com aquele ponteiro. Se a contagem de instâncias chegar a zero (isto é, a última cópia daquele ponteiro está sendo destruída), o objeto é destruído também, usando `delete`.

Essa última parte é um problema para nós, já que os tipos da SDL não são tratados com `delete`, e sim com `SDL_DestroyTexture`. Voltaremos a isso num instante.

```
std::unordered_map<std::string, std::shared_ptr<SDL_Texture>>
    imageTable;
```

`Resources::imageTable` deve passar a ser uma tabela de `shared_ptr`s para `SDL_Texture`, tornando-se um forte candidato para a variável com o tipo mais legal do trabalho. A função `Resources::GetImage` deve retornar um `shared_ptr`, e `Sprite` deve guardar um também (em vez do ponteiro simples para `SDL_Texture*`).

Em `GetImage`, caso a imagem já esteja alocada, o procedimento é o mesmo. Caso não esteja, depois de chamar `IMG_LoadTexture` e checar se a textura é nula, criamos um `shared_ptr` (alocação estática!) com a textura, inserimos na tabela, e retornamos.

Em `ClearImages`, procuramos `shared_ptr`s cujo número de instâncias é igual a 1 (função `unique`). Removemos essas entradas da tabela, destruindo a última instância e causando a desalocação da textura. Ah, mas ainda não resolvemos a questão do `delete`, não é?

Além do primeiro argumento da construção de `shared_ptr`, podemos passar um functor que especifique como o ponteiro deve ser deletado. Mas nesse caso, melhor ainda do que definir uma classe seria usar uma função lambda. Consulte o apoio de C++, seção 13, caso não conheça esses componentes da linguagem.

O formato de lambda aceito pelo construtor de `shared_ptr` é uma função que recebe um ponteiro simples do tipo apontado pelo `shared_ptr`. Com `Resources` implementado dessa maneira, a função `ClearImages` passa a ser chamada sempre que um estado é pop-ado, para liberar recursos que ele não vai mais usar.

Uma observação importante sobre o fechamento do jogo: Garanta

que, antes de liberar os recursos ao sair do loop de `Game::Run`, a pilha de estados está vazia. Se o loop for quebrado por `quit`, e não `pop`, isso não vai ser verdade, e a limpeza de recursos não afetará os recursos de estados ainda alocados.

Enfim, essa mesma adaptação na classe (uso de `shared_ptr`) deverá ser feita para gerenciar `Mix_Music*` e `Mix_Chunk*`. O que são eles? Excelente pergunta!

2. Music

Music
+ Music() + Music(file : string) + Play (times : int) : void + Stop () : void + Open (file : string) : void + IsOpen () : bool
- music : Mix_Music*

`Mix_Music` e `Mix_Chunk` são tipos são definidos na `SDL_mixer.h`, e vamos usá-los agora. Vá até a `Game` e adicione chamadas às funções de inicialização e encerramento da biblioteca. São elas:

```
int Mix_Init(int flags)
int Mix_OpenAudio(int frequency, Uint16 format, int channels, int chunksize)

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.

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.

Semelhanças com a interface de Sprite são totalmente intencionais. Music “abre” um arquivo de música e o reproduz quando pedido. “Abre”, já que delegamos a tarefa de administrar recursos para Resources. Adicione os seguintes membros àquela classe.

```
- musicTable : std::unordered_map<std::string, Mix_Music*>
```

Uma tabela assim como a de imagens. Se você implementou o extra, use shared_ptr<Mix_Music>.

```
+ GetMusic (file : string) : Mix_Music*
```

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

Assim como GetImage, procura o arquivo na tabela, se não encontrar, carrega, e, se o carregamento for bem sucedido, insere na tabela.

```
+ ClearMusic () : void
```

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

Libera as músicas da tabela. Se você implementou o extra, GetImage deve fornecer o deletor com Mix_FreeMusic, e esta função vai apenas procurar os shared_ptrs únicos.

De volta a Music.

```
> Music()  
> Music(file : 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) : void
```

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

Mix_PlayMusic recebe uma música e quantas vezes ela deve ser repetida *depois da primeira execução*. Se loops for -1, a música repete

infinitamente. Vale notar que a Mixer só suporta uma música sendo tocada por vez: Se outra música já estiver tocando, ela para.

> Stop () : 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.

> Open (file : string) : void

Pede a abertura do recurso para Resources. Se bem sucedido, guarda o ponteiro.

> IsOpen () : bool

Checa se music é nula.

Com a classe pronta, acrescente um membro Music em StageState e carregue o arquivo de música *audio/stageState.ogg*. A música deve começar a tocar na criação do estado.

3. Sound

Sound
+ Sound() + Sound(file : string) + Play (times : int) : void + Stop () : void + Open (file : string) : void + IsOpen () : bool
- chunk : Mix_Chunk* - channel : int

Sound é quase a mesma classe de Music, mesmo na implementação. As diferenças estão nas funções da Mixer usadas, e no fato de que, diferente

das músicas, existem 32 canais diferentes para reproduzir sons, e temos que manter registrado em qual canal o chunk está tocando para podermos pará-lo se necessário.

Primeiro, adicione em Resources membros GetSound, ClearSounds, e soundTable, exatamente como os de music, trocando apenas as funções da Mixer usadas.

```
Mix_Chunk* Mix_LoadWAV(char* file)
void Mix_FreeChunk(Mix_Chunk* chunk)
```

> Play (times : int) : void

```
int Mix_PlayChannel(int channel, Mix_Chunk* chunk, int loops)
```

A Mixer permite que você administre os channels manualmente, mas é preferível que você deixe ela fazer isso para você. Peça o channel -1, ela escolherá um primeiro canal vazio e retornará o número dele para você.

Novamente, loops indica quantas vezes o som deve se repetir. Passe o valor apropriado para a função.

> Stop () : void

```
int Mix_HaltChannel(int channel)
```

Mix_HaltChannel para um canal específico, ou, se receber -1, para todos. Use o número que Mix_PlayChannel te retornou.

> Open (file : string) : void

Pede a abertura do arquivo para Resources.

Sons prontos! Quando Penguins e o Alien emitirem uma animação de explosão, faça com que eles reproduzam o som de explosão (*audio/boom.wav*).

...Espero que não tenha cansado de alterar Resources. Temos mais uma mudança a fazer.

4. Text

Text
<pre>+ TextStyle : enum (SOLID, SHADED, BLENDED) + Text (fontFile : string, fontSize : int, style : TextStyle, text : string, color : SDL_Color, x : int = 0, y : int = 0) + ~Text() + Render(cameraX : int = 0, cameraY : int = 0) : void + SetPos(x : int, y : int, centerX : bool = false, centerY : bool = false) : void + SetText(text : string) : void + SetColor(color : SDL_Color) : void + SetStyle(style : TextStyle) : void + SetFontSize(fontSize : int) : void - RemakeTexture() : void</pre>
<pre>- font : TTF_Font* - texture : SDL_Texture* - text : string - style : TextStyle - fontSize : int - color : SDL_Color - box : Rect</pre>

Você precisará da `SDL_ttf.h` para implementar essa classe, que por sua vez, precisa ser inicializada em Game usando `TTF_Init()` e encerrada por `TTF_Quit()`. Nenhuma das funções recebe argumentos e a `Init` retorna 0 se for bem sucedida.

Text cria uma textura contendo um texto arbitrário, em uma fonte carregada de um arquivo. A cor do texto é dada por uma `SDL_Color`, uma struct contendo membros `r`, `g` e `b`. Como ela não tem um construtor além do padrão, pode ser conveniente criar um índice de cores mais usadas no seu código, ou uma função que retorne uma `SDL_Color` automaticamente para certos parâmetros.

Precisaremos dos membros `GetFont`, `ClearFonts` e `fontTable` em `Resources`. As funções da TTF são as seguintes:

```
TTF_Font* TTF_OpenFont (const char* file, int ptsize)
void TTF_CloseFont (TTF_Font* font)
```

Note que uma instância de uma fonte não depende só do arquivo, mas também do tamanho da fonte. Assim, não é possível usar `fontFile` como o a chave da tabela diretamente, mas isso é fácil de resolver: Concatene `fontFile` com o tamanho para criar uma chave única.

```
> Text(fontFile : string, fontSize : int, style : TextStyle,
      text : string, color : SDL_Color, x : int = 0, y : int = 0)
```

Inicialize todos os membros da classe, excluindo `box.w` e `h`, e incluindo `texture` (`nullptr`).

Com todos os membros inicializados, e se a fonte tiver sido carregada corretamente, chame `RemakeTexture` para criar a textura.

```
> ~Text()
```

Se a textura existir, destrua-a.

```
> Render(int cameraX = 0, int cameraY = 0) : void
```

Renderiza a textura (se existir) na posição dada por `box`, ajustada para a posição da câmera dada. Use nossa velha amiga `RenderCopy`.

```
int SDL_RenderCopy(SDL_Renderer* renderer, SDL_Texture* texture,
                  const SDL_Rect* srcrect, const SDL_Rect* dstrect)
```

```
> SetPos(x : int, y : int, centerX : bool = false,
      centerY : bool = false) : void
```

Altera `box.x` e `box.y` para as coordenadas dadas. Se a flag `centerX` for

true, centralize o texto em x em vez de setar diretamente. O mesmo vale para centerY.

```
> SetText(text : string) : void
> SetColor(color : SDL_Color) : void
> SetStyle(style : TextStyle) : void
> SetFontSize(fontSize : int) : void
```

Cada uma destas funções altera um atributo e chama `RemakeTexture` para aplicar a mudança.

```
> RemakeTexture() : void
```

Se já houver uma textura criada, destrua-a.

RemakeTexture usa uma das três funções a seguir para renderizar o texto.

```
SDL_Surface* TTF_RenderText_Solid(TTF_Font* font, const char* text, SDL_Color
fg)
SDL_Surface* TTF_RenderText_Shaded(TTF_Font* font, const char* text,
                                   SDL_Color fg, SDL_Color bg)
SDL_Surface* TTF_RenderText_Blended(TTF_Font* font, const char* text,
                                   SDL_Color fg)
```

Qual usar depende do estilo do texto. Solid é o texto renderizado diretamente, sem nenhum tipo de tratamento visual. O texto Shaded é renderizado em um retângulo da cor dada em bg. Usaremos preto sempre na nossa implementação, mas se você quiser, pode implementar métodos para mudar essa cor.

Finalmente, Blended cria um texto com as bordas suavizadas (usando o canal alpha), para que ele se adapte melhor à região da tela onde for renderizado. É mais custoso de se renderizar que os outros dois, mas faz uma diferença perceptível.

Mas essas funções tem uma coisa diferente: Elas não retornam uma `SDL_Texture*`, mas sim um ponteiro para estrutura da `SDL 1.2`, `SDL_Surface`. Não podemos usá-la no nosso contexto atual, então use as seguintes funções para obter uma textura e deletar a surface temporária.

[illegible]

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

Lembre-se também de setar os membros w e h do box, que podem ter sido alterados com a mudança. Você pode obtê-los direto da surface, ou usar `SDL_QueryTexture`.

Agora que temos texto, vá até a tela de título e coloque um texto orientando o usuário a pressionar espaço para continuar. No entanto, só ter o texto ali é desinteressante, logo, usando uma flag e um Timer, faça com que o texto alterne entre ser mostrado ou não.

Quase pronto~ O que falta no nosso jogo é simplesmente, acabar. Não importa o que aconteça, ficamos em `StageState` enquanto não voltarmos para o menu. Vamos resolver!

5. (ou 1f.) EndState

EndState (herda de State)
+ EndState(stateData : StateData)
+ Update() : void
+ Render() : void
+ Pause() : void
+ Resume() : void
- bg : Sprite
- music : Music
- instruction : Text

`EndState` é uma tela que diz para o jogador “YOU WIN” ou “YOU LOSE”.

> `EndState(stateData : StateData)`

...Espera, o que raios é uma `StateData`?

StateData
+ playerVictory : bool

StateData é um mecanismo de comunicação entre estados. Nem sempre os dados de um estado são restritos a eles, muito pelo contrário, é comum que eles compartilhem informações durante a transição de um para o outro.

A classe (que não precisa ter esse nome, nem ser a mesma na construção de todos os estados) é feita **especificamente** para o seu jogo conter todas as informações que queiramos passar. Stats, pontuação, entre outras. No caso do nosso jogo, EndState só está interessado em se o jogador ganhou ou não.

```
> EndState(stateData : StateData) : void
```

Se o jogador ganhou, o bg e a música são aqueles de vitória. Se perdeu, bg e música de derrota. Em todo caso, a instruction apenas diz para o jogador pressionar ESC para ir ao menu, ou espaço para tentar de novo.

```
> Update() : void
```

EndState responde a eventos da janela (pede quit), à tecla ESC (pede pop) e à barra de espaço (pede pop e põe um StageState na pilha).

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

Renderiza o bg e os textos.

Pause e Resume são vazias.

Agora vá até o StageState. Lembra das condições de vitória e derrota que descrevemos no último trabalho? Assim que forem atendidas, faça com que o StageState peça pop (afinal, o jogo acabou) e empilhe uma EndState com os dados apropriados.

Para terminar, distribua aleatoriamente um número razoável de Aliens pelo mapa. Para impedir que eles ajam em sincronia, dê updates também aleatórios neles para avançar o timer de cooldown.

Os pinguins continuam andando pra fora da borda do cenário... Hm. A maneira normal de resolver isso é acrescentando metadados ao nosso tileset, para saber se estamos fora do mapa, ou tentando atravessar um tile intransponível.

6. Tilemap

...Sabe do que mais? Deixa pra lá.

O mapa do nosso jogo é um retângulo 1408x1280. Se o pinguim passar desses limites, impeça o seu movimento. Não é o jeito bonito de fazer, mas vamos logo pro jogo final, sim? Muita coisa a ser feita por lá, ainda.

Olhando as classes que desenvolvemos até agora, você deve saber diferenciar o que é engine e o que é lógica específica. Pegue as primeiras, adicione ao projeto do seu jogo final, e vamos ao trabalho.

Happy Development!

+ Extra (+1,0 ponto) : Implemente as mudanças em Resources descritas na seção 1e. Elas devem ser feitas para todos os tipos de recursos, não só para `SDL_Textures`.