Trabalho 7 – Game States 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…

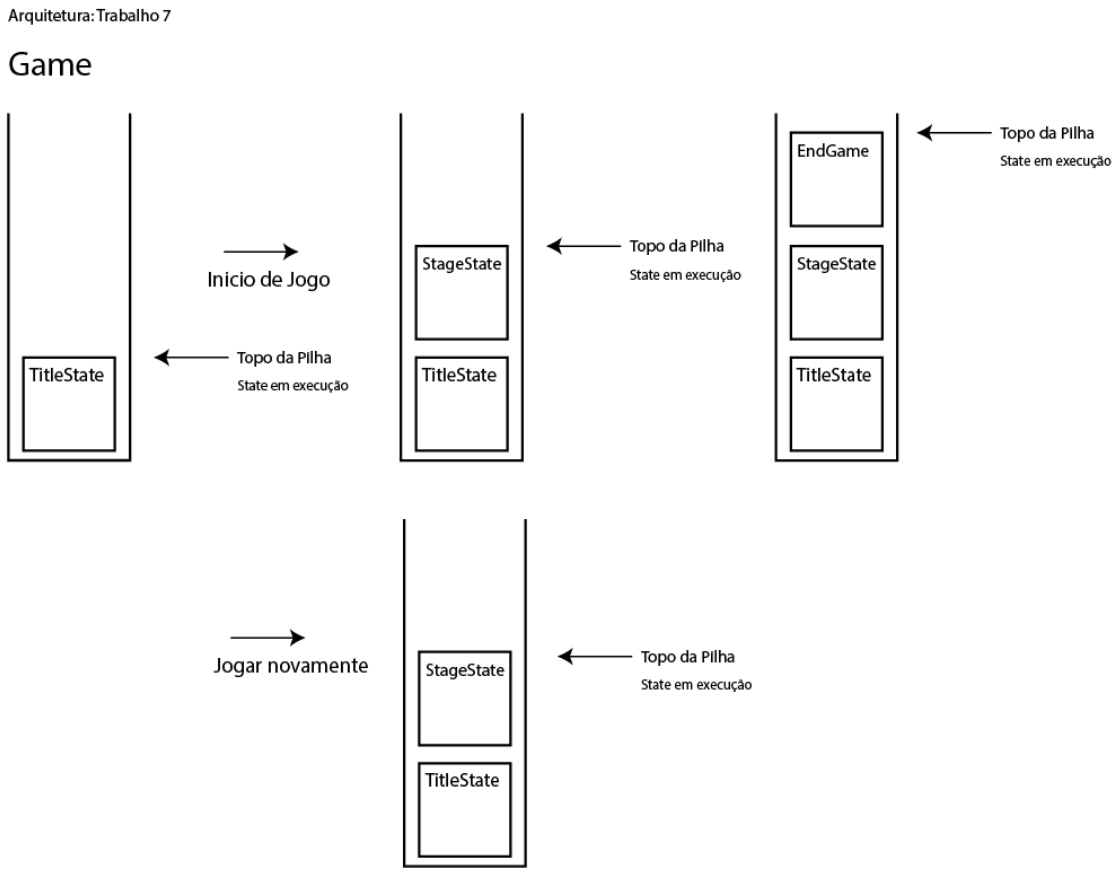
É possível fazer isso numa classe só - *possível*, não *recomendável*. O resultado seria um código enorme, cheio de ifs, cujos trechos não tem muita relação entre si. Como organizar isso, então? A verdade é que você já conhece a solução: O conceito de Game State, que é o que implementamos em State.

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.



(Manter ou não o estado na pilha depende de alguns fatores: se vale a pena ou não deixar o state em memória para ser carregado mais rápido, a lógica de transição de telas, por exemplo acessar menu, que pode te levar a uma tela de progressos salvos, a um minigame, ou a uma tela de opções, etc etc.)

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
+ State () + ~State () : virtual + LoadAssets () : void, virtual pura + Update (dt : float) : void, virtual pura + Render () : void, virtual pura + Start () : void, virtual pura + Pause () : void, virtual pura + Resume () : void, virtual pura + AddObject (object : GameObject*) : std::weak_ptr< GameObject >, virtual + GetObjectPtr (object : GameObject*) : std::weak_ptr< GameObject >, virtual + PopRequested () : bool + QuitRequested () : bool # StartArray () : void # UpdateArray (dt : float) : void, virtual # RenderArray () : void, virtual
# popRequested : bool # quitRequested : bool # started : bool # objectArray : std::vector<std::shared_ptr<GameObject>>

State implementa uma interface básica de comportamento de um estado (Start, 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).

```
> ~State ()
```

Esvazia o vetor de objetos.

```
> AddObject (object : GameObject*) : std::weak_ptr< GameObject >,
virtual
> GetObjectPtr (object : GameObject*) : std::weak_ptr< GameObject >,
virtual
```

Copie as funções de StageState para cá.

```
> Start () : void, virtual pura
> 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. Resume, quando o estado acima é desempilhado. E Start para quando ele é criado e adicionado na pilha.

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

```
> Update () : void, virtual pura
> Render () : void, virtual pura
```

Já essas funções são chamadas por Game todo frame. Elas são puras pois você pode querer fazer alguma computação/renderização antes ou depois das dos objetos e, dessa forma, isso se torna possível.

```
> 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.

```
> StartArray () : void, virtual
> UpdateArray (dt : float) : void, virtual
> RenderArray () : void, virtual
```

Aqui, percorreremos os objetos no vetor fazendo a inicialização, atualização e renderização deles. Esses loops não costumam mudar muito, então nos convém 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 : std::string, width : int, height : int) + ~Game () + GetInstance () : Game& + GetRenderer () : SDL_Renderer* + GetCurrentState () : State& + Push (state : State*) : void + Run () : void + GetDeltaTime () : float - CalculateDeltaTime () : void
- frameStart : int - dt : float - instance : 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 : std::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. Limpe os Resources. 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 armazenar 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 ouver 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 do loop, vamos primeiro gerenciar a pilha e depois executar as mesmas tarefas de antes.

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

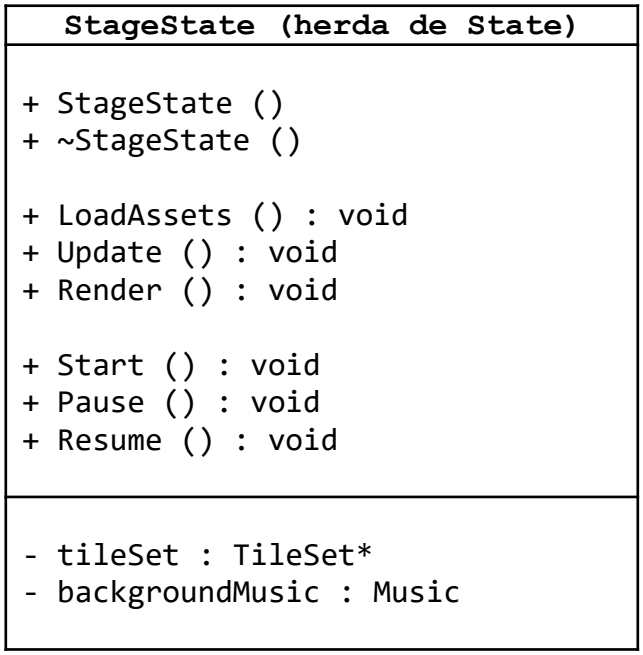
Na saída, lembre-se de limpar os Resources, como antes.

```
> 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.

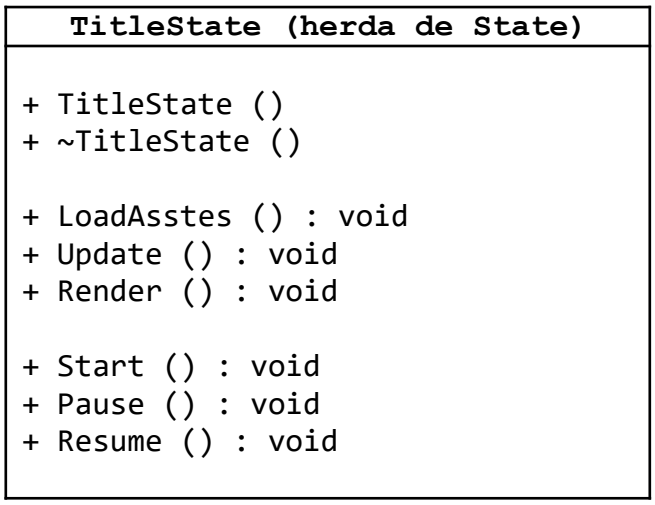
1c. StageState



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 StartArray, 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()`

Inicializa adiciona um GameObject com a imagem “img/Title.png”.

`> 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.

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 (Opcional, não vale nota)

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 mantem, 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_ptrs 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_ptrs 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.

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

2. Text

Text (herda de Component)
+ TextStyle : enum (SOLID, SHADED, BLENDED) + Text (associated : GameObject&, fontFile : std::string, fontSize : int, style : TextStyle, text : std::string, color : SDL_Color) + ~Text () + Update (dt : float) : void + Render () : void + SetText (text : std::string) : void + SetColor (color : SDL_Color) : void + SetStyle (style : TextStyle) : void + SetFontFile (fontFile : std::string) : void + SetFontSize (fontSize : int) : void - RemakeTexture () : void
- font : TTF_Font* - texture : SDL_Texture* - text : std::string - style : TextStyle - fontFile : std::string - fontSize : int - color : SDL_Color

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, b e a. 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. Algo que pode ajudar é usar Aggregate Initialization. Pesquise sobre isso, caso queira.

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 a chave da tabela diretamente, mas isso é fácil de resolver: Concatene fontFile com o tamanho para criar uma chave única.

```
> Text (associated : GameObject&, fontFile : string, fontSize : int,
        style : TextStyle, text : string, color : SDL_Color)
```

Inicialize todos os membros da classe e incluindo texture (nullptr).

Com todos os membros inicializados, chame RemakeTexture para criar a textura.

```
> ~Text ()
```

Se a textura existir, destrua-a.

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

Por enquanto é vazia.

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

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

```
int    SDL_RenderCopyEx(SDL_Renderer*    renderer,    SDL_Texture*    texture,
                    const SDL_Rect* srcrect, const SDL_Rect* dstrect,
                     double angle, SDL_Point* center, SDL_RendererFlip flip)
```

ClipRect é de {0,0} até largura e altura da box de associated. DestRect é a posição da box de associated subtraída pela câmera e largura igual do ClipRect.

```
> 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. Obtenha a nova fonte e só continue se conseguir carregá-la.

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.

```
SDL_Texture* SDL_CreateTextureFromSurface(SDL_Renderer* renderer,
                                          SDL_Surface* surface)
void SDL_FreeSurface(SDL_Surface* surface)
```

Se tiver conseguido gerar uma textura, lembre-se 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, faça com que o texto alterne entre ser mostrado ou não. Você pode fazer como quiser, desde uma flag e um Timer ou dois floats e um Timer para mais controle. Modifique o construtor caso julgue necessário.

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!

3. (ou 1f.) EndState

EndState (herda de State)
+ EndState () + ~EndState () + LoadAssets () : void + Update () : void + Render () : void + Start () : void + Pause () : void + Resume () : void
- backgroundMusic : Music

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

```
> EndState ()
```

Vamos usar GameData para determinar qual tela de final será carregada... Espera, o que raios é uma GameData?

GameData
+ playerVictory : bool

GameData é um mecanismo de armazenamento de informações genéricas sobre o jogo. 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. Como por exemplo a pontuação, até qual fase o jogador chegou, quanto tempo ele sobreviveu ou se ele ganhou ou perdeu.

A classe (que não precisa ter esse nome) é feita **especificamente** para o seu jogo e contêm 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 () : void
```

Se o jogador ganhou, crie um bg e inicialize a música usando arquivos de vitória. Se perdeu, o bg e a música são os de derrota. Em todo caso, a instruction apenas diz para o jogador pressionar ESC para sair do jogo, ou espaço para jogar de novo.

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

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

Agora vá até o StageState. As condições de vitória e derrota serão acabarem as waves, ou o jogador morrer, respectivamente. Assim que forem atendidas, faça com que o StageState peça pop (afinal, o jogo acabou) e empilhe uma EndState com os dados apropriados setados.

Os Characters 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.

4. Tilemap

…Sabe do que mais? Deixa pra lá.

O mapa do nosso jogo é um retângulo 1280x1536 começando em 640x512. Se o Character 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.

Acredito que nesse ponto você já deva saber para que serve LoadAssets e todos os lugares aos quais ela deva ser chamada. Se não tiver feito ainda, faça agora.

Happy Development!

+ Extra (opcional, não vale nota): Implemente as mudanças em Resources descritas na seção 1e. Elas devem ser feitas para todos os tipos de recursos, não só para SDL_Textures. Isso deve afetar Mix_Music em Music, Mix_Chunk em Sound e TTF_Font em Text.