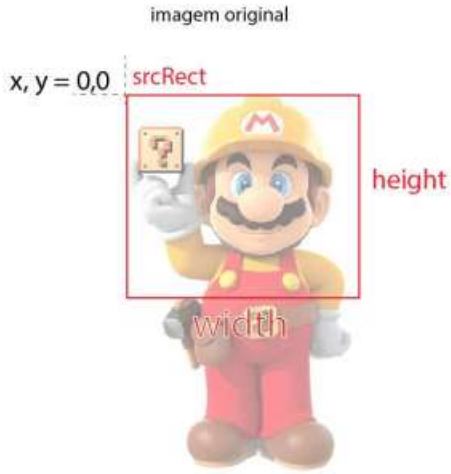
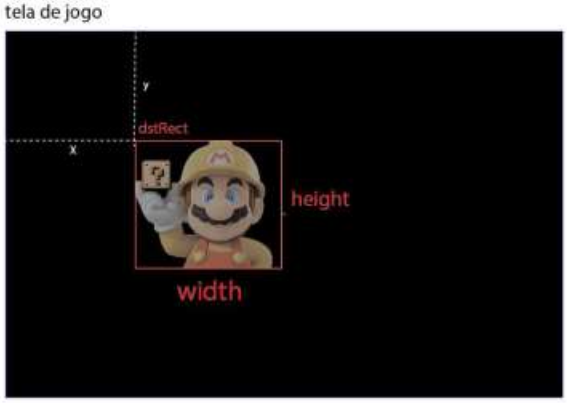
Trabalho 3 – Tile Set, Tile Map e Resource Management

Arquitetura: Trabalho 3





TileSet

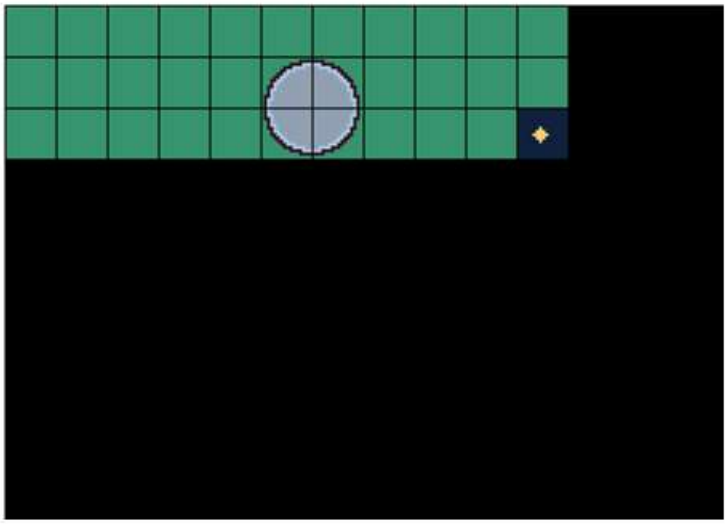
TileMap

colunas

linhas — 3 11

0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	14	16	0	0	0	0
0	0	0	0	0	28	30	0	0	0	13

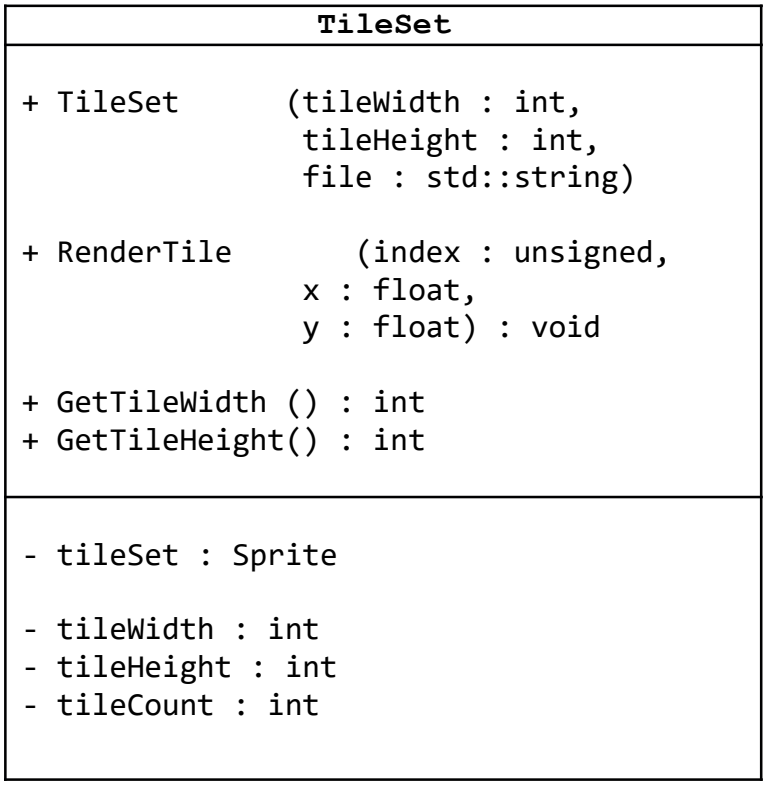
Tela de Jogo



A imagem acima refresca um pouco a memória sobre a renderização da imagem no Trabalho1. Lembramos que, para renderizar, utilizávamos uma

função que recebia como parâmetros um SDL_Rect referente ao retângulo que vamos utilizar da imagem original (srcRect) e outro SDL_Rect indicando onde e quais dimensões esse "recorte" teria na tela de jogo (dstRect). Vamos utilizar essa noção para conseguir recortar o nosso TileSet, criando recortes de imagens indexados, que serão mapeados na tela por meio do TileMap.
Confundir TileSet e TileMap no primeiro contato é bem comum. Traduzindo talvez facilite:
- TileSet: conjunto de azulejos (apenas a imagem que agrupa todos os tiles);
- TIleMap: mapa de azulejos (mapeamento de como distribuir os azulejos na tela);

1. TileSet: Tiles Para Nossos Mapas



Uma classe de tileset é responsável por armazenar os tiles utilizados na renderização do TileMap. Internamente, os tiles fazem parte de um grande Sprite (*img/Tileset.png*). Quando queremos renderizar um deles, recortamos usando o clip do Sprite.

```
> TileSet (tileWidth : int, tileHeight : int, file : std::string)
```

Seta as dimensões dos tiles e abre o Sprite. Se a abertura for bem sucedida, descobre, pelo tamanho do sprite e dos tiles, quantas colunas e quantas linhas o tileset tem. Use isso para definir a quantidade de frames do sprite (quantidade de tiles).

```
> RenderTile (index : unsigned, x : float, y : float) : void
```

Cheque se o índice é válido para o número de tiles que temos, isto é, está entre 0 e o número de tiles - 1. Se sim, sete o frame desejado no sprite, e renderize na posição dada.

```
> GetTileWidth () : int
> GetTileHeight() : int
```

Retornam as dimensões dos tiles. Estas serão usadas por…

2. TileMap: Mapeando Tiles em Posições

TileMap (herda de Component)

`+ TileMap (associated : GameObject&, file : std::string, tileSet : TileSet*)`
 	
`+ Load (file : std::string) : void`
 	
`+ SetTileSet (tileSet : TileSet*) : void`
 	
`+ At (x : int, y : int, z : int = 0) : int&`
 	
`+ Render () : void`
 	
`+ RenderLayer (layer : int) : void`
 	
`+ GetWidth () : int`
`+ GetHeight () : int`
`+ GetDepth () : int`

```
- tileMatrix : std::vector<int>
- tileSet    : std::unique_ptr<TileSet>
- mapWidth   : int
- mapHeight  : int
- mapDepth   : int
```

TileMap simula uma matriz tridimensional, representando nosso mapa e suas diversas camadas. Essa matriz contém, em cada posição, um índice de tile no TileSet.

Os atributos são um vector de inteiros, um ponteiro para o TileSet em uso, e as dimensões do mapa (largura, altura e número de camadas).

`> TileMap (associated : GameObject&, file : std::string, tileSet : TileSet*)`

Chama Load com a string passada e seta o tileset.

`> Load (file : std::string) : void`

Load deve carregar um arquivo de mapa, no formato dado pelo arquivo *map/map.txt*, presente no zip de resources no Google Drive da disciplina. Os primeiros três números são as dimensões do mapa: largura, altura e profundidade. Em seguida, vêm os tiles, que devem ser carregados em ordem para a matriz de tiles.

Note que, para o arquivo que usamos na disciplina, tiles vazios são representados por -1, que é o padrão do editor de tilemaps open source TileD.

`> SetTileSet (tileSet : TileSet*) : void`

Troca o tileSet em uso.

`> At (x : int, y : int, z : int = 0) : int&`

`At` é um método acessor. Ele retorna uma referência ao elemento `[x][y][z]` de tileMatrix. Acontece que tileMatrix é um vetor, portanto, você precisa calcular qual o índice real do elemento `[x][y][z]` no vetor (encapsular essa conta é um dos propósitos da função).

`> RenderLayer (layer : int) : void`

Renderiza uma camada do mapa, tile a tile. Note que há um ajuste a se fazer: Deve-se considerar o tamanho de cada tile (use os membros `GetTileWidth()` e `GetTileHeight()` de TileSet).

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

Renderiza as camadas do mapa. Dica: utilize o RenderLayer e use o box do GameObject que o contém para definir a posição do mapa no jogo.

```
> GetWidth  ()  : int
> GetHeight ()  : int
> GetDepth  ()  : int
```

Retornam as dimensões do mapa.

3. Mudanças

Temos os tiles, temos o mapa. Faremos agora algumas alterações em State para renderizá-lo.

Acrescente no construtor um GameObject com o TileMap(map/map.txt), lembrando de construir o Tileset(img/Tileset.png) para enviar ao TileMap. As dimensões das subimagens do tileset são 64x64. Coloque o box desse GameObject em (0,0).

Se tudo der certo, você vai ver o mesmo mapa que mostramos em sala na janela. Além disso, seu Zombie deve continuar funcionando como estava antes. Vamos incrementá-lo um pouco.

4. Sound

Sound
+ Sound() + Sound(file : std::string) + ~Sound() + Play (times : int = 1) : void + Stop () : void + Open (file : std::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 vários canais diferentes para reproduzir sons, e temos que manter registrado em qual canal o chunk está tocando para podermos pará-lo se necessário.

```
> Sound()
```

Atribui nullptr a chunk.

```
> Sound(file : std::string)
```

Chama o outro construtor(esse é um dos usos da initializer list) e depois chama Open passando a string recebida.

```
> Play (times : int = 1) : 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. E é por isso que alocamos 32 canais no começo. Se quiséssemos colocar um som e não tivesse algum canal livre, ele não tocaria.

Dessa vez, loops indica quantas vezes o som deve ser repetido, ou seja, loops = 1 faz tocar duas vezes. Já nosso argumento é quantas vezes deve ser executado, ou seja, times = 1 deve tocar somente uma vez. 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. Pare o canal no qual o chunk correspondente está tocando, mas só faça isso se chunk for diferente de nullptr.

> Open (file : std::string) : void

```
Mix_Chunk* Mix_LoadWAV(char* file)
```

Abre o arquivo chamando o MIX_Load. Lembre-se de tratar o caso em que a abertura do arquivo falha!

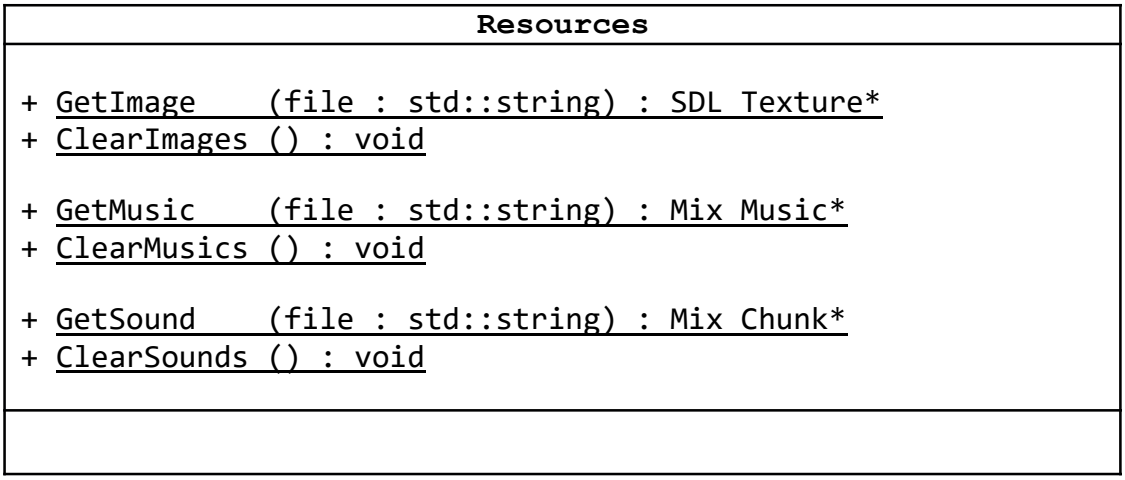
> ~Sound()

```
void Mix_FreeChunk(Mix_Chunk* chunk)
```

Se chunk for diferente de nullptr, chame Halt e depois desaloque o som usando o Mix_FreeChunk.

Agora que temos sons, adicione um membro do tipo Sound à classe Zombie, chamado deathSound. Inicialize ele com o arquivo "Recursos/audio/Dead.wav" no construtor, e quando o HP dele chegar em 0 na função damage, chame Play(1). Se tudo for de acordo com o planejado, nosso Zombie fará um barulho assim que morrer. Sucesso! Porém agora temos muitos assets para gerenciar, e se criarmos mais instâncias de Zombie no futuro, todas elas terão que possuir a própria cópia do sprite e do som. Vamos corrigir isso.

5. Resources: Gerenciando nossas texturas



```
- imageTable : std::unordered_map<std::string, SDL_Texture*>
- musicTable : std::unordered_map<std::string, Mix_Music*>
- soundTable : std::unordered_map<std::string, Mix_Chunk*>
```

TileSet tem um mecanismo de reuso de Sprites próprio, mas ainda temos o problema, vindo do trabalho passado, de Zombies estarem realocando o mesmo sprite multiplas vezes na memória. Precisamos de um mecanismo que mantenha registradas SDL_Textures que já estão alocadas, e permita que vários objetos as compartilhem.

Uma estrutura ideal para isso é uma tabela de hash, mas são difíceis de se implementar… onde vamos achar uma?

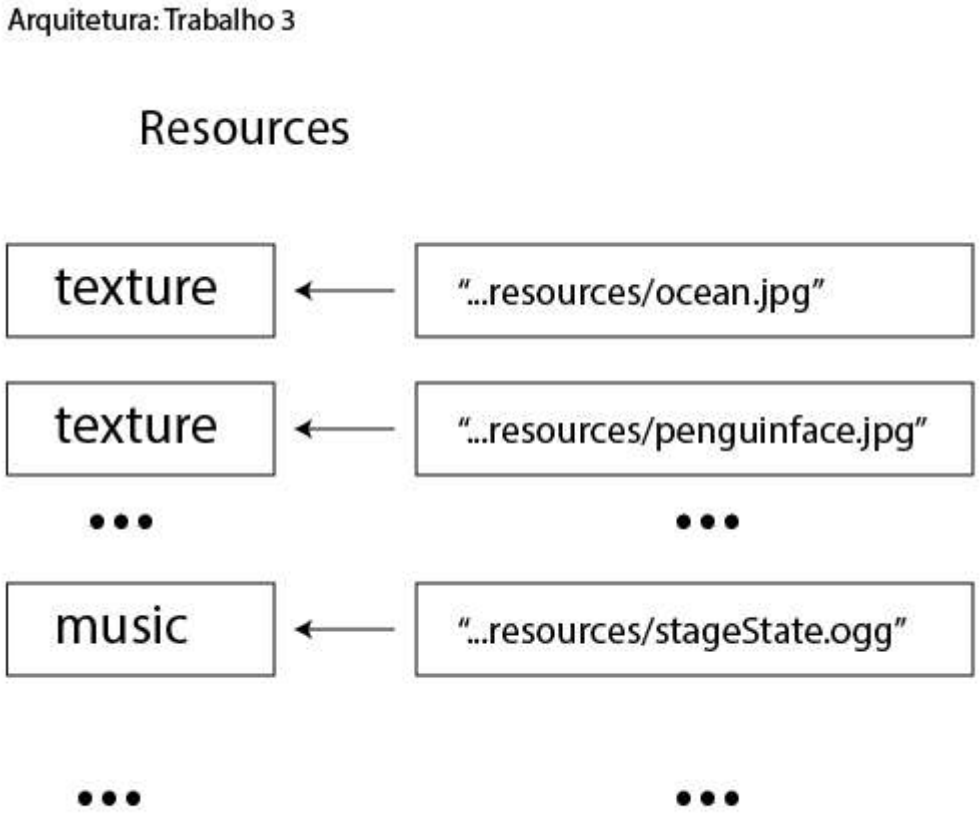
```
#include <unordered_map>
```

C++11 acrescentou à STL um template novo bastante interessante. O unordered_map é uma tabela de hash, e recebe dois parâmetros:

1. Um tipo a ser usado como chave para a tabela
2. Um tipo de conteúdo apontado pela chave

A chave passa por uma operação de hashing, e o conteúdo é encontrado de forma bastante eficiente. O membro imageTable é associado uma string (o caminho de um arquivo) a um ponteiro de textura.

Toda vez que carregarmos uma nova textura, iremos guardá-la nessa tabela, para que não precisemos carregá-la de novo depois. Isso implica em mudanças em Sprite.



> `Sprite::~Sprite ()`

O destrutor não deve mais destruir a textura em uso. Resources tratará de alocações e desalocações daqui pra frente.

> `Sprite::Open (file : std::string) : void`

Open, da mesma forma, não deve mais destruir a textura se ela já estiver alocada, mas além disso, em vez de chamar IMG_LoadTexture, ela chamará Resources::GetImage.

Este método, por sua vez…

> `Resources::GetImage (file : std::string) : SDL_Texture*`

Primeiro, cheque se a imagem já existe na tabela de assets (`find`). Se sim, obtenha o ponteiro gravado lá e retorne.

Se ela não existe, carregue, da mesma forma que fazia em Sprite. Se a imagem foi carregada com sucesso, insira o par caminho e ponteiro na tabela e retorne o ponteiro.

Com isso, teremos a garantia de que uma mesma imagem nunca será carregada mais de uma vez. Mas as texturas ainda não são desalocadas.

> Resources::ClearImages () : void

Percorre a tabela de imagens destruindo textura por textura. Ao final, esvazia a tabela. Inclua uma chamada a esse método após o main game loop, em Game::Run.

O mesmo deve ser feito para Music/Mix_Music e para Sound/Mix_Chunk.

Você deve imaginar que um jogo liberar memória só na saída não é apropriado. Uma olhada rápida em diretórios de instalação dos mesmos mostra GBs e mais GBs de recursos, e seria inviável manter tudo em memória ao mesmo tempo.

De fato, o gerenciamento de recursos nesses jogos conta com algoritmos mais elaborados. Saber quantos objetos estão usando aquele recurso, se alguém pode precisar em breve, há quanto tempo ele está ou não em uso, quanta memória o jogo ainda pode usar, se há como alocar novos recursos na memory pool, todas essas informações são relevantes.

A maior parte disso está fora do escopo da disciplina, e é normalmente desnecessário para os trabalhos finais. Mais tarde, apresentaremos uma maneira de contar usuários de um recurso usando std::shared_ptr, mas por enquanto, não se preocupe com isso.

Com tudo pronto, tente criar no seu State mais alguns inimigos em posições diferentes e veja-os morrendo algum tempo depois.