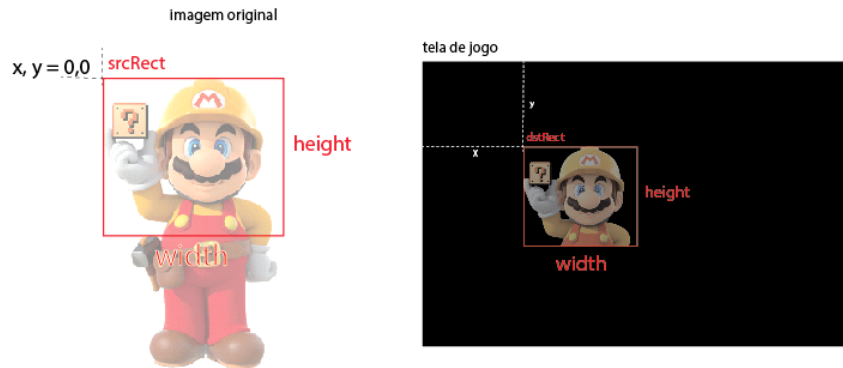
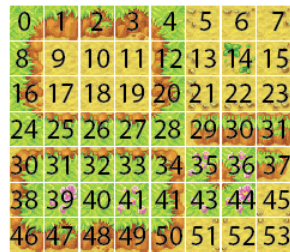


Trabalho 3 – Tile Set, Tile Map e Resource Management

Arquitetura: Trabalho 3



TileSet

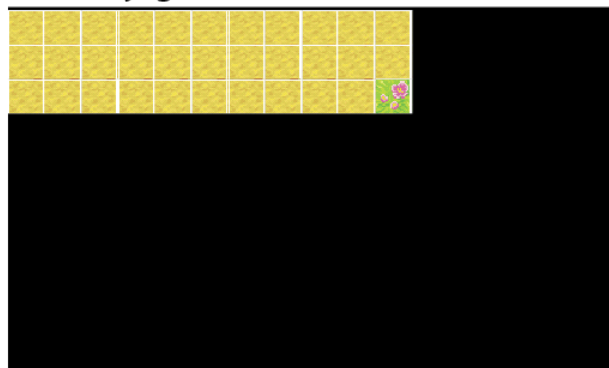


TileMap

linhas — 2 11
colunas —

	9	9	9	9	9	9	9	9	9	9	9	9
	9	9	9	9	9	9	9	9	9	9	9	9
	9	9	9	9	9	9	<u>9</u>	9	9	9	9	39

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

TileSet	
+ TileSet	(tileWidth : int, tileHeight : int, file : std::string)
+ RenderTile	(index : unsigned, x : float, y : float) : void
+ GetTileWidth ()	: int
+ GetTileHeight()	: int
- tileSet	: Sprite
- rows	: int
- columns	: int
- tileWidth	: int
- tileHeight	: int

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.

```
> 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, calcule e sete o clip desejado no sprite, e renderize na posição dada.

Mas como faremos para fazer o Sprite renderizar com base nos argumentos x e y ao invés de associated.box? Bem, será necessário criar uma sobrecarga de Sprite::Render que receba esses argumentos. É um boa para evitar duplicação de código organizado fazer o Sprite::Render que não recebe argumentos chamar o Sprite::Render que recebe esses argumentos utilizando os valores do associated.

```
> 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, cameraX : int = 0, cameraY : int = 0) : void
+ GetWidth	() : int
+ GetHeight	() : int
+ GetDepth	() : int

```
- tileMatrix : std::vector<int>
- tileSet    : 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/tileMap.txt*, presente no zip de resources no Moodle. 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 0, que é o padrão do editor de tilemaps open source TileD. Para o nosso código, é mais conveniente que eles sejam representados por -1, e o primeiro tile do tileset por 0. Portanto, subtraia um de cada índice lido do arquivo.

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

Renderiza uma camada do mapa, tile a tile. Note que há dois ajustes a se fazer:

- Deve-se compensar o deslocamento da câmera
- Deve-se considerar o tamanho de cada tile (use os membros `GetTileWidth()` e `GetTileHeight()` de `TileSet`)

Ainda não temos câmera. Você pode deixar para implementar o primeiro ponto naquela ocasião, mas já faça a função recebendo esses argumentos.

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

Renderiza as camadas do mapa. Dica: utilize o `RenderLayer` e o box do `GameObject` que o contém.

```
> 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/tileMap.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). Importante: não se esqueça de desalocar o `TileSet` na hora certa!

Se tudo der certo, você vai ver o mesmo mapa que mostramos em sala na janela. Além disso, suas faces devem continuar funcionando como estavam antes. Se você lembra do que falamos no último trabalho, isso é uma coisa ruim.

4. Resources: Gerenciando nossas texturas

Resources
<u>+ <code>GetImage (file : std::string) : SDL Texture*</code></u> <u>+ <code>ClearImages () : void</code></u>
<u>+ <code>GetMusic (file : std::string) : Mix Music*</code></u> <u>+ <code>ClearMusics () : void</code></u>
<u>+ <code>GetSound (file : std::string) : Mix Chunk*</code></u> <u>+ <code>ClearSounds () : void</code></u>
 <u>- <code>imageTable : std::unordered map<std::string, SDL Texture*></code></u> <u>- <code>musicTable : std::unordered map<std::string, Mix Music*></code></u> <u>- <code>soundTable : std::unordered map<std::string, Mix Chunk*></code></u>

TileSet tem um mecanismo de reuso de Sprites próprio, mas ainda temos o problema, vindo do trabalho passado, de Faces estarem realocando o mesmo sprite dezenas de 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:

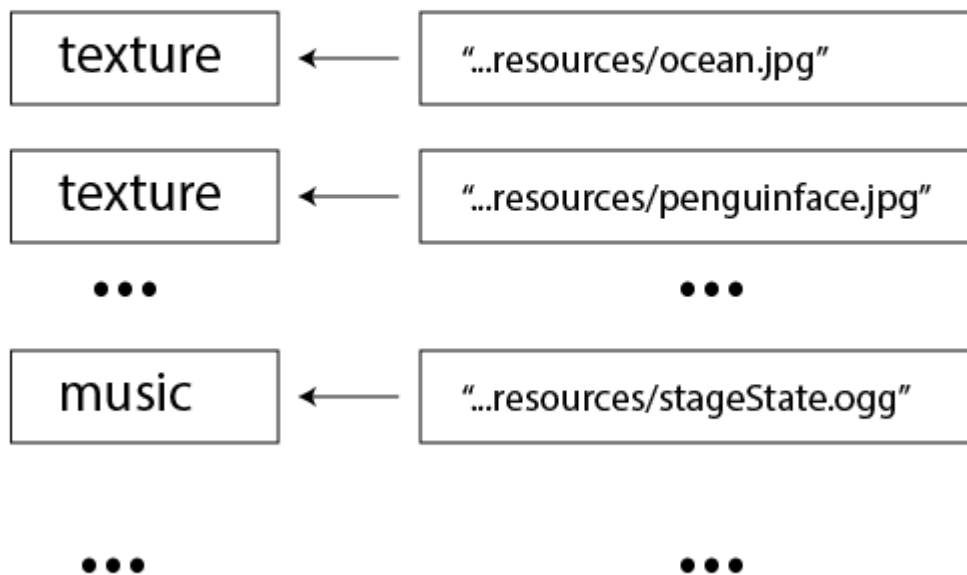
1. Um tipo a ser usado como chave para a tabela
2. Um tipo de conteúdo apontado pela chave

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.

Arquitetura: Trabalho 3

Resources



> `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.