Trabalho 3 – Tile Set, Tile Map e Resource Management

1. TileSet: Tiles Para Nossos Mapas

TileSet
+ TileSet (tileWidth : int, tileHeight : int, file : string) + Render (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 : 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.

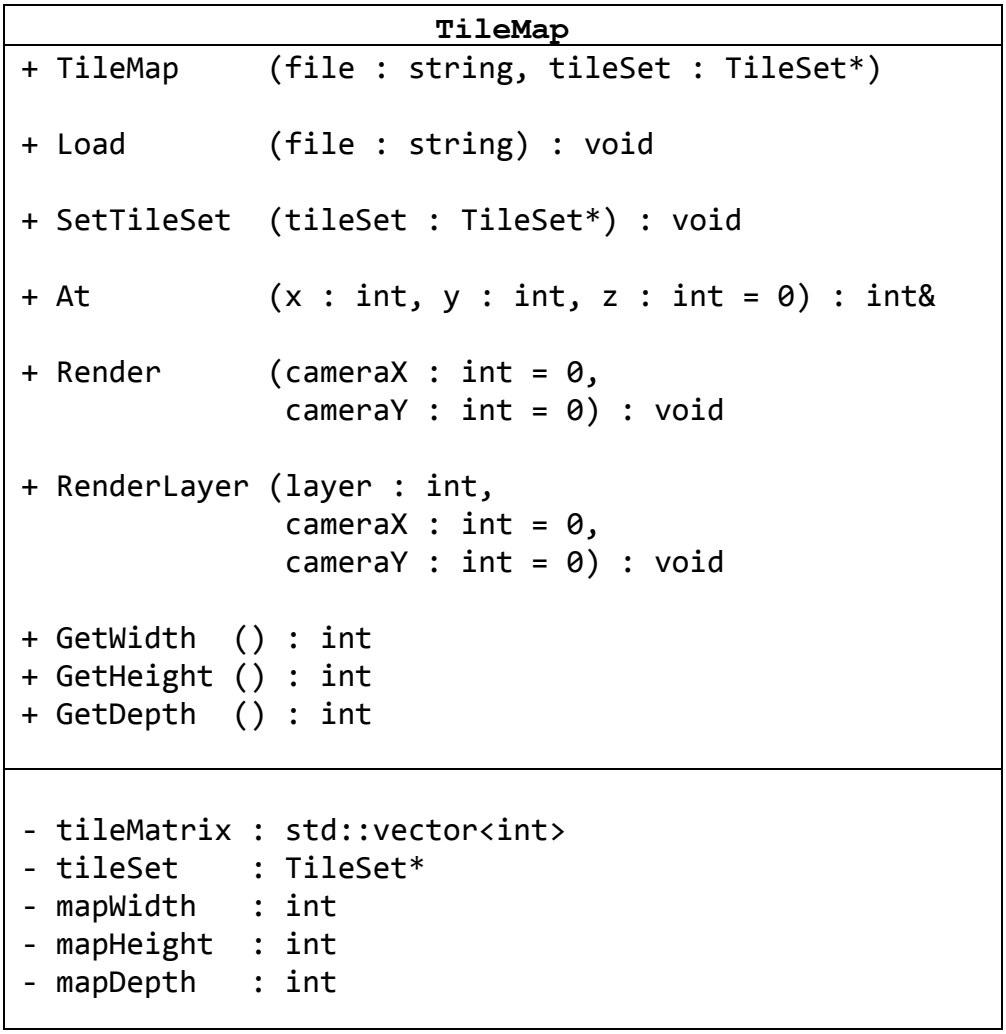
`> Render (index : unsigned, x : float, y : float) : void`

Cheque se o índice é válido, para o número de tiles que temos. Se sim, calcule e sete o clip 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 simula uma matriz tridimensional, representando nosso mapa e suas diversas camadas. Essa matriz contem, 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 (file : string, tileSet : TileSet*)
```

Chama Load com a string passada e seta o tileset.

```
> Load (file : 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.

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

Renderiza todas as camadas do mapa.

```
> 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.

a. Acrescente os membros tileSet (TileSet) e tileMap (TileMap).
b. No construtor, inicialize os dois. Os tiles são 64x64.
c. Em Render, renderize o TileMap na posição 0,0 da tela (considere 0, 0 como a posição da câmera, já que ainda não temos uma ainda). No nosso jogo, queremos que a layer 0 apareça por baixo dos objetos, e a layer 1, por cima.

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
+ GetImage (file : string) : SDL_Texture* + ClearImages () : void
- imageTable : std::unordered_map<std::string, SDL_Texture*>

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 compartilhem-as.

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 associa 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 : 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 : 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 (`emplace`), 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.

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.