Trabalho 4 – Input, Temporização e Câmera

1. InputManager: Abstraindo os SDL_Events

InputManager
+ Update () : void + KeyPress (key : int) : bool + KeyRelease (key : int) : bool + IsKeyDown (key : int) : bool + MousePress (button : int) : bool + MouseRelease (button : int) : bool + IsMouseDown (button : int) : bool + GetMouseX () : int + GetMouseY () : int + QuitRequested () : bool + GetInstance () : InputManager& - InputManager () - ~InputManager ()
- mouseState : bool[6] - mouseUpdate : int[6] - keyState : ler abaixo - keyUpdate : ler abaixo - quitRequested : bool - updateCounter : int - mouseX : int - mouseY : int

Ao invés de usar uma função como a State::Input, é preferível que haja uma parte na sua engine que leia o estado do input do jogo a cada frame, e que possa ser consultado por interessados. InputManager cumpre essa

função: Ele é um singleton que lê a pilha de eventos sempre que seu método Update é chamado (no início de cada frame).

Cada mudança no estado de uma tecla ou botão do mouse é registrada no array e na tabela contidos na classe. Como a SDL indexa botões do mouse de 1 a 5, não é necessário nada além de um array simples para fazermos acesso direto.

No entanto, para as teclas, os valores de SDL_Keycodes (a enum da SDL que se refere às teclas) estão um pouco mais espalhadas. Teclas cujos valores tem um caracter correspondente tem keycodes no range 0x0 até 0x7F, igual ao valor na tabela ASCII.

Teclas que não tem valor de caracter, como Caps Lock, teclas F, Shift, Alt, etc., estão no range 0x40000000 até 0x4000011A. Ou seja, ainda é possível colocar todas as teclas num array, mas é necessário fazer um ajuste para teclas no range superior (subtrair 0x3FFFFF81, para que os blocos fiquem alinhados).

Se essa solução não soar agradável (entendo perfeitamente), você pode usar a solução que a documentação da SDL propõe, isto é, usar uma tabela de hash. Dessa forma, nenhum ajuste precisa ser feito, basta acessar direto o valor na tabela. O código feito corretamente dessa forma fica mais limpo.

O problema é que você precisa conhecer como a unordered_map funciona para garantir que o programa é bug-free. Especificamente, como o par emplace/find e o operator[] funcionam.

Dependendo da sua escolha, os membros serão:

```
- keyState : bool[416]   (0x7F + 0x11A, os tamanhos dos blocos)
- keyUpdate : int[416]
```

ou:

```
- keyState : std::unordered_map<int, bool>
- keyUpdate : std::unordered_map<int, int>
```

Agora, você pode estar se perguntando por que manter dois vetores para guardar o estado dos botões. De verdade, apenas `keyState` e `mouseState`

guardam estado - true se o botão está pressionado, false caso contrário. Os vetores de Update, no entanto, guardam uma informação importante - quando o último pressionamento ou soltura ocorreu.

Quando falamos de controles em jogos, há uma grande diferença entre uma tecla *ser pressionada* e *estar pressionada*. O primeiro se trata de um único frame, o segundo é verdadeiro em qualquer frame desde o pressionamento até a soltura. Algumas ações no jogo só são executadas no frame de pressionamento, e o seu InputManager precisa saber distinguir.

`> GetInstance () : InputManager&`

Usaremos um pequeno truque para administrar a nossa instância singleton. Ao invés de dar um new InputManager na primeira vez que a função é chamada, declare dentro dela uma variável InputManager estática. Você pode então simplesmente retorná-la. Isso se chama *Meyers' Singleton*.

`> InputManager ()`

O construtor deve inicializar os arrays de estado e update do mouse. Se você estiver usando vetores para as teclas, faça o mesmo para eles. Já as tabelas de hash não precisam (nem devem) ser inicializadas. Lembre-se também das outras variáveis da classe (updateCounter, quitRequested e mouse)

`> Update () : void`

Update faz o processamento de eventos, de forma similar a como fizemos em Game::Input. Fazemos uso da função SDL_PollEvent, uma função que recebe um ponteiro para uma variável do tipo SDL_Event. Se houver um evento ainda a ser processado, ela retorna true e o grava na variável de evento. Se não, retorna false.

SDL_Event é uma union. Assim, antes de tentar ler um evento, devemos identificar seu tipo usando o membro `event.type`. Há muitos tipos possíveis, por exemplo, para lidar com roda do mouse ou touch screens. Se te interessarem, consulte a documentação da SDL. No momento, estamos interessados apenas nos seguintes tipos:

SDL_KEYDOWN	Pressionamento de tecla
SDL_KEYUP	Uma tecla foi solta
SDL_MOUSEBUTTONDOWN	Pressionamento de botão do mouse
SDL_MOUSEBUTTONUP	Botão do mouse foi solto
SDL_QUIT	Clique no X, Alt+F4, etc.

QUIT deve ser tratado simplesmente setando a flag quitRequested. MOUSEBUTTONDOWN e UP devem setar o botão correspondente no array de mouse buttons como true ou false. Para saber qual foi o botão pressionado, acesse `event.button.button`.

Para os eventos de key, podemos saber qual o keycode via `event.key.keysym.sym`. Se você optou por usar vetores, lembre-se de fazer o ajuste em teclas no range maior. Se você optou pelo unordered_map, note que o operator[] *insere automaticamente elementos que não existem*. Última dica que eu dou... :p

Lembre-se também que, para todo evento, seja botão ou tecla, deve ocorrer o registro do valor do contador de updates do frame atual para aquela tecla.

Além disso, você já deve ter percebido, brincando com as faces, que se segurarmos a tecla, após um tempo o programa começa a receber vários eventos SDL_KEYDOWN. Para saber se o evento é repetido, cheque o valor da flag `event.key.repeat`. Se ela for igual a 1, não registre nada sobre esse evento. Com isso, terminamos o loop.

Antes de começar o loop, em frame, temos três outras tarefas. Primeiro, obter as coordenadas atuais do mouse (SDL_GetMouseState). Segundo resetar a flag de quit. Isso pode parecer estranho, mas pense que se um evento de quit não for tratado por State no frame em que ocorreu, é porque State já tratou e resolveu não fechar o jogo (ainda), ou não se interessou.

Terceiro, devemos implementar o contador de updates. Ele será usado em

```
> KeyPress   (key : int) : bool
> KeyRelease (key : int) : bool
> IsKeyDown  (key : int) : bool
> MousePress   (button : int) : bool
> MouseRelease (button : int) : bool
> IsMouseDown  (button : int) : bool
```

___Press e ___Release estão interessadas no pressionamento ocorrido naquele frame, e só devem retornar true nesse caso. Use os vetores ___Update e o updateCounter para saber. Is___Down retorna se o botão/tecla está pressionado, independente de quando isso ocorreu.

Para quem usou vetor: Lembre-se do ajuste. Para quem usou tabela de

hash: Nenhuma destas funções precisa de mais de uma linha.

```
> GetMouseX () : int
> GetMouseY () : int
```

Retornam as coordenadas atuais do cursor.

```
> QuitRequested (): bool
```

Retorna `quitRequested`.

Com a classe pronta, adicione também os #defines a seguir no header da classe. Eles servem para ocultar os nomes da SDL do seu programa e evitar um acoplamento tão forte.

```
#define LEFT_ARROW_KEY    SDLK_LEFT
#define RIGHT_ARROW_KEY   SDLK_RIGHT
#define UP_ARROW_KEY      SDLK_UP
#define DOWN_ARROW_KEY    SDLK_DOWN
#define ESCAPE_KEY        SDLK_ESCAPE
#define LEFT_MOUSE_BUTTON SDL_BUTTON_LEFT
```

Agora, vamos integrar o InputManager ao programa. Acrescente uma chamada ao método Update em Game::Run(), logo antes da chamada ao update do estado. Feito isso, remova State::Input() e adicione os seguintes comportamentos a State::Update():

- Setar a flag de quit de State se ESC for pressionado ou se o InputManager apontar evento de Quit;
- Se a barra de espaço for pressionada, criar uma Face da mesma forma de antes.

Feito isso, em Face::Update, faça ela checar por pressionamentos do botão do mouse. Caso haja um, e o cursor esteja dentro do box dela, aplique o mesmo dano aleatório de antes. Agora abra o seu jogo, e delicie-se ao ver que, após esse trabalho todo, temos quase a mesma coisa que tínhamos ao final do trabalho passado!

...Isso não é legal, né. Não, não é legal. Vamos fazer umas coisas mexerem na tela, pra animar. Para isso, precisamos implementar...

2. Temporização: Medindo Intervalos Entre Frames

É essencial, por mais simples que seja, que uma simulação de física consiga dizer quanto tempo passou desde a última atualização. Por exemplo, para um objeto atualizar sua posição...

$$X = X_0 + v_0 * \Delta t + \frac{a * \Delta t^2}{2}$$

...Ele precisa saber quanto tempo se passou desde a última atualização, e é responsabilidade da engine informá-lo disso. Passar o valor é só uma questão de chamar Update(dt), mas antes, precisamos calculá-lo.

Adicione os seguintes membros em Game:

```
- frameStart : int
- dt : float
- CalculateDeltaTime () : void
+ GetDeltaTime () : float
```

```
> CalculateDeltaTime () : void
```

CalculateDeltaTime atribui valor a dt, e deve ser chamada no início de cada iteração do main game loop. Para calcular intervalos de tempo, podemos usar a seguinte função, que nos diz quantos milissegundos se passaram desde a inicialização da biblioteca.

```
Uint32 SDL_GetTicks ()
```

No início de cada frame, `frameStart` guarda o início do frame anterior. Atualizamos `frameStart` e usamos o valor antigo e o novo para calcular dt em milissegundos. Não é o ideal, pois milissegundos são intervalos pequenos demais para se trabalhar de forma intuitiva - converta para segundos, tomando cuidado com erros de arredondamento.

O cálculo do dt deve ser a primeira coisa no main game loop. Lembre-se de inicializar `dt` e `frameStart` na classe.

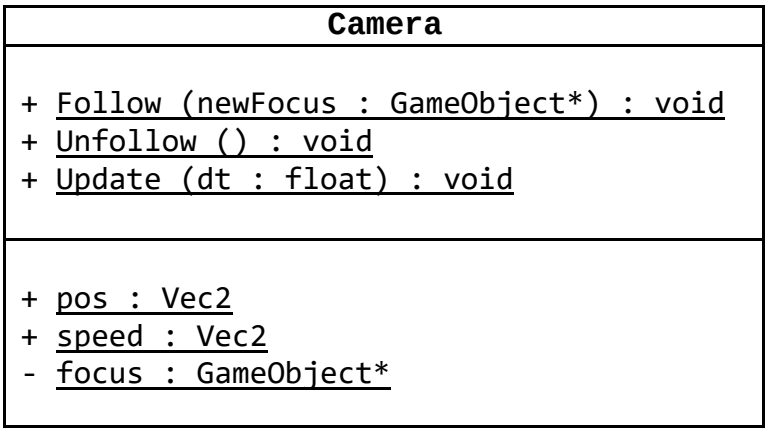
```
> GetDeltaTime () : float
```

GetDeltaTime retorna dt para entidades interessadas, incluindo State. Com esse valor em mãos, você pode atualizar os objetos do seu vetor de GameObjects. Faça isso e verá... A mesma coisa.

Suas faces lá, paradonas, olhando pra você... você olha pra elas... elas olham pra você... Não chegamos a lugar nenhum. E é nesse momento de indignação que você grita:

"*CANSEI! Agora vou fazer TUDO mexer de uma vez!*" ╭ಠ益ಠ╮

3. Camera: Movendo a Cena



Independente da perspectiva - primeira pessoa, terceira pessoa, top-down - jogos tentam criar a ilusão de que há um observador na cena, e que vemos o jogo pelos olhos dele.

Para alcançar essa ilusão, manteremos a posição desse observador hipotético guardada em algum lugar, e comunicamos ou deixamos acessível aos objetos da cena. Assim, cada pode se renderizar e se atualizar levando em consideração o deslocamento da câmera.

Arquitetura: Trabalho 4

Tela de Jogo

Observe que existe um movimento relativo entre câmera e "objetos do mundo". Dizer "a câmera está indo para a direita" é equivalente a dizer "todos os objetos do mundo estão indo para a esquerda". O resultado que aparece na tela em ambas situações é o mesmo.

(Momento filosófico..) Às vezes, dependendo do jogo, pode ser mais conveniente manter os valores da câmera fixos e os objetos do mundo se moverem. Em outros casos, manter os objetos fixos e a câmera mover.

`> Follow (newFocus : GameObject*)`

Seta um novo ‘foco’ para a câmera, que é um GameObject a ser seguido por ela.

`> Unfollow () : void`

Atribui nullptr ao foco.

`> Update (dt : float) : void`

Se a câmera tiver um foco, faremos com que ele fique centralizado na tela. Nesse caso, o movimento independe de dt, depende apenas do tamanho da tela.

Se não houver um foco, devemos responder ao input: Setamos a velocidade da câmera de acordo com dt e com as teclas pressionadas, e somamos à posição.

Perceba que, apesar de simples, se codificada dessa forma a câmera pode operar de três maneiras: chamando Update com foco, chamando sem foco, e setando as coordenadas diretamente em State, sem chamar Update. Usaremos a câmera sem foco, por enquanto. Em State::Update, chame o update da câmera, e em State::Render, passe as coordenadas da câmera para o TileMap, e teste se ele se move corretamente.

Se estiver, bom! Mas ainda estamos com as Faces numa posição fixa na tela. Faça três adaptações: Primeiro, faça a renderização deles levar em consideração a posição da câmera. Segundo, em Update, quando for checar se a Face foi clicada, compense o deslocamento da câmera nas coordenadas do mouse - O InputManager nos dá as coordenadas da tela, não do mundo. Terceiro, em State::AddObject, também leve em consideração a câmera na hora de posicionar as Faces.

Obs: Quando usarmos um foco, tome cuidado com o objeto que você escolher: Se ele for deletado antes de a câmera parar de segui-lo, o programa vai crashar.

4. CameraFollower

CameraFollower (herda de Component)
+ CameraFollower (go : GameObject&) + Update (dt : float) : void + Render () : void + Is (type : std::string) : bool

Se você mover a câmera para além do tilemap, pode-se observar um efeito indesejado. A imagem fica repetida sobre um pedaço de si mesma. Isso acontece porquê não estamos limpando o render anterior, na verdade estamos sempre renderizando por cima do frame anterior. Nós queremos que seja sempre por cima de uma imagem, a ocean.jpg, lembra?

Mas nós já a adicionamos. Onde ela está? Bom, como Sprite é componente e fica num gameObject, este fica fixo no mapa. Isso é desejável para objetos, mas não para imagens de fundo e elementos de UI. Para resolver isso, basta criarmos o componente CameraFollower. A única coisa que precisa fazer é no Update fazer com que a posição de seu gameObject associado seja igual à posição da câmera.

Adicione esse componente ao gameObject que contêm a Sprite de fundo e voilà!

5. TileMap: Parallax Scrolling (extra)

(+1,0 ponto): Implemente Parallax Scrolling no seu jogo (com a renderização das camadas na hora certa). Ver explicação em sala.