Introdução ao Desenvolvimento de Jogos – Turma A
Professora: Carla Denise Castanho (carlacastanho@cic.unb.br)
Monitores: Gustavo Arcanjo (gustavo.arcanjo@gmail.com)
Leonardo Guilherme (leonardo.guilherme@gmail.com)
Lucas Carvalho (lucasnvcarvalho@gmail.com)
Lucas Nunes (l.nunes.202@gmail.com)
Matheus Pimenta (matheuscscp@gmail.com)
Murilo Sousa (murlsousa@gmail.com)

Trabalho 2 – Movimento, Herança e Input

1. Classe InputManager: Coletando e Gerenciando Entrada.

InputManager
- mouse, *teclado : Uint8 - keyDown[N_KEYS], keyUp[N_KEYS] : Uint8 - quitGame : bool - mouseDown[N_MOUSE_BUTTONS], mouseUp[N_MOUSE_BUTTONS] : Uint8 - mouseX, mouseY : int - *instance : InputManager
+ getInstance() : InputManager* + Update() : void + isKeyDown(key : int) : bool + isKeyUp(key : int) : bool + isKeyPressed(key int) : bool + isMouseDown(button int) : bool + isMouseUp(button int) : bool + isMousePressed(button : Uint8) : bool + mousePosX() : int + mousePosY() : int + isMouseInside(*rect : SDL_Rect) : bool + QuitGame(): bool

A classe InputManager será responsável por verificar os estados das teclas e do mouse. A classe deve ser implementada utilizando o padrão singleton (Este padrão garante a existência de apenas uma instância de uma classe, mantendo um ponto global de acesso ao seu objeto). Para referência, consultar: http://pt.wikipedia.org/wiki/Singleton#Em_C.2B.2B

Importante!
A InputManager apresentada aqui foi construída de forma a ficar simples e didática.
Ela pode ser otimizada em quase todos os lugares, de inúmeras formas.
Sugerindo algumas:

- Os vetores não precisam ser zerados todos os frames.
- Os estados não precisam ser atualizados todos os frames.

- Os vetores Up e Down poderiam ser substituídos por outra estrutura.
- Poderia ser usado um padrão listener, ou connect.

Sinta-se livre para implementar a classe como achar melhor, se você quiser **melhorar** a performance da Classe pode ignorar completamente essa especificação.

Sobre os atributos da classe:

- `mouse, *teclado : Uint8`
  Guardam o Estado do mouse e do teclado.
- `mouseX, mouseY : int`
  Guardam as posições do mouse.
- `keyDown[N_KEYS], keyUp[N_KEYS] : Uint8`
  Guardam as teclas apertadas/soltas do teclado.
- `mouseDown[N_MOUSE_BUTTONS], mouseUp[N_MOUSE_BUTTONS] : Uint8`
  Guardam as teclas apertadas/soltas do mouse.
- `quitGame : bool`
  Guarda o evento de fim de jogo.
- `*instance : InputManager`
  Guarda uma referência da classe (singleton).
  O construtor da classe deve chamar o método Update().

Sobre os métodos da classe:

`+ getInstance() : InputManager*`
Retorna uma referência para a classe (singleton).

`+ Update() : void`
Para verificar eventos, usamos uma variável do tipo da struct SDL_Event e a função SDL_PollEvent(event). A SDL_PollEvent verifica todos os eventos de input que estiverem sendo recebidos, um por vez, e retorna false quando não houverem mais. Assim, para obter todos eventos que estiverem sendo recebidos, usa-se o seguinte:

```
while(SDL_PollEvent(event)
{
      /* verifica tipo do evento e toma a ação */
      switch (event.type)
      case SDL_KEYDOWN:
            keyDown[event.key.keysym.sym] = true;
}
```

Dentro do loop, verifica-se qual o tipo do evento (acessando event.type). Os tipos de evento que nos interessam a princípio são:

SDL_KEYDOWN – verificar se uma tecla foi pressionada.
SDL_KEYUP – verificar se uma tecla foi solta.
SDL_MOUSEMOTION – verifica se o mouse se moveu
SDL_MOUSEBUTTONDOWN – verifica se um botão do mouse foi pressionado
SDL_MOUSEBUTTONUP – verifica se um botão do mouse foi solto.

SDL_QUIT – verifica o fim do jogo (se a janela foi fechada, sinal de TERM, etc)

Após verificar qual o tipo do evento, podemos:

- acessar event.key(.keysym.sym) para saber qual tecla foi pressionada, caso o evento seja do tipo SDL_KEYDOWN ou SDL_KEYUP; ou
- acessar event.button para saber qual botão do mouse foi pressionado, caso o evento seja do tipo SDL_MOUSEBUTTONDOWN ou SDL_MOUSEBUTTONUP;

Importante: A SDL armazena todos os eventos em uma pilha de eventos. Quando usamos o método SDL_PollEvent(event) esses eventos são retirados da pilha. Ou seja, podemos fazer o Poll de um evento apenas uma vez. Para que possamos checar se um evento qualquer aconteceu a qualquer momento no frame, descarregaremos esses eventos para vetores próprios.

Sabendo disso, implemente o Update da seguinte forma:

1. Zerar os vetores mouseDown, mouseUp, keyDown, keyUp
2. Atualizar o estado do mouse e do teclado (usando as funções SDL_GetMouseState() e SDL_GetKeyState();
3. Descarregar os eventos de Input da SDL para os nossos vetores (keyUp, keyDown, mouseUp e mouseDown). Basta definir a posição da tecla ou botão apertado como true.
4. Checar pelo quitGame e armazená-lo

```
+ isKeyDown(key : int) : bool
+ isKeyUp(key : int) : bool
+ isMouseDown(button int) : bool
+ isMouseUp(button int) : bool
+ isKeyPressed(key int) : bool
+ isMousePressed(button : Uint8) : bool
```

Note que há uma diferença entre perceber se um botão **foi** pressionado ou **está sendo** pressionado. No primeiro caso, retornamos true apenas no frame único em que o botão foi pressionado. Já no segundo caso, retornamos true enquanto o botão estiver sendo pressionado.
Dessa forma, temos os métodos isKeyDown e isMouseDown, que verificam se uma tecla ou um mouse foram pressionados, respectivamente – ou seja, retornam true apenas no momento em que a tecla/mouse tiverem sido pressionados em um frame.

Já os métodos isKeyPressed ou isMousePressed verificam se alguma tecla ou mouse estão sendo pressionados, respectivamente – ou seja, retornam true enquanto a tecla/mouse estiverem sendo pressionados, durante todos os frames.
Além disso, isKeyUp, assim como isMouseUp, é um método que retorna true no momento em que uma tecla é solta.

Dica: para detectar se um botão está sendo pressionado, basta obter o valor no vetor keyState. Para detectar se um botão foi pressionado, o modo mais fácil é por verificação de eventos.

```
+ mousePosX() : int
+ mousePosY() : int
```

Os métodos mousePosX e mousePosY retornam as posições x e y do mouse.

`+ isMouseInside(*rect : SDL_Rect) : bool`

O método isMouseInside verifica se o mouse está dentro de um retângulo e retorna true nesse caso.

`+ QuitGame(): bool`

Retorna o valor contido em guitGame.

Alterações no Game Manager:

Alterar todo o método processEvents para utilizar a InputManager.
Dentro de processEvents o jogo deve:

1. Chamar o Update da InputManager.
2. Usando a InputManager:
   - atualizar a câmera;
   - criar os planetas;
   - checar pelo fim de jogo.

Após essas mudanças o jogo deve continuar funcionando corretamente.

2. Temporização:

Precisamos adicionar ao game loop uma temporização. Isso porque os objetos do jogo, para terem suas posições atualizadas de acordo com a física, precisam da variação de tempo entre o frame anterior e o atual, que é o Δt das fórmulas de física, como na fórmula da atualização do espaço em um movimento uniforme:

$X = X_0 + v\Delta t$

Sendo assim, para obter o tempo, utilizamos a função
`Uint32 SDL_GetTicks ( void ).`

Ela retorna o número de milissegundos desde a inicialização da biblioteca. No entanto, como dito, o que nos interessa não é isso, mas sim a diferença de tempo entre um frame e outro, que é nosso ΔT = TFRAME_ATUAL - TFRAME_ANTERIOR.

Calcule, para cada iteração do loop, o valor de timer e o valor de dt (Δt). Esse cálculo deve ser a 1ª coisa feita no main game loop (método run).

Dica: Use 3 variáveis:
`int dt, frameStart e frameEnd;`
frameStart para guardar o tempo ao iniciar o frame, frameEnd para guardar o do frame anterior, antes que o timer do frame atual seja calculado e dt para conter o cálculo do tempo do frame.

Mande imprimir no terminal o valor de dt, a cada frame, apenas para teste. Verifique. Essa é a taxa de milisegundos por frame variável do seu jogo!

Por fim, para reduzir o gasto desnecessário de processamento do jogo, adicionamos um delay ao final do loop. Esse delay serve para limitar o número de

frames processados pelo computador por segundo, caso o frame atual tenha sido processado muito rápido.

Como o limite de percepção da visão humana é de 30 quadros por segundo, fixaremos o valor do fps do jogo em 30. Para isso, basta testar se o dt atual foi menor que 1000/30 (ou seja, se gastou menos que 1s/30 para processar o frame.), e, caso tenha sido, completar o tempo que falta com um `SDL_Delay(1000/30 - dt)`.

3. Classe GameObject: O pai de todos os objetos.

GameObject
+ x : float + y : float
+ GameObject(x : float, y : float) : GameObject + update(dt : int) = 0 : int + render(camera : float, cameraY : float) = 0 : void

A fim de representar todos os objetos do jogo usaremos uma superclasse GameObject. Essa classe possui apenas posição e métodos de atualização e renderização, comuns a todos os objetos de jogo.

Sobre os atributos da classe:

`- x, y : float`

Guardam a posição do objeto

Sobre os métodos da classe:

`+ GameObject(x : float, y : float) : GameObject`

O construtor da classe. Inicializa as posições x e y de acordo com os parâmetros do construtor.

`+ update(dt : int) = 0 : int`

Método virtual puro, deve ser implementado nos filhos.

`+ render(camera : float, cameraY : float) = 0 : void`

Método virtual puro, deve ser implementado nos filhos.

4. Classe Planet: Brincando de herança.

Planet : GameObject
- sprite : Sprite* + hitPoints : int
+ Planet(sprite : Sprite*, x : float, y : float, hitPoints : int) + update(dt : int) = 0 : int + render(cameraX = 0.0 : float, cameraY = 0.0 : float) : void

Treinando o uso de herança, faremos aqui uma superclasse Planet (filha de GameObject), de onde serão derivados todos os planetas usados nos trabalhos (isso inclui nosso velho amigo planeta vermelho, a Terra, a lua e futuramente os planetas azul e verde).

Sobre os atributos da classe:

`- sprite : Sprite*`
Guardam um ponteiro para a Sprite a ser usada na renderização do planeta
`- hitPoints : int`
Guarda a vida do planeta

Sobre os métodos da classe:

`+ Planet(sprite : *Sprite,  x : float, y : float, hitPoints : int) : float`
O construtor da classe. Deve chamar o construtor da classe pai (passando os valores de x e y) e inicializar o valor de hitPoints e da sprite.

`+ update(dt : int) = 0 : int`
Método continua sendo virtual puro, deve ser implementado nos filhos.

`+ render(cameraX = 0.0 : float, cameraY = 0.0 : float) = 0 : void`
Se a sprite nao for nula, mostra na tela (levando em consideração os valores da câmera).

5. PlanetRed: O fim da linha

PlanetRed : Planet
+ PlanetRed(sprite : Sprite*, x : float, y : float, hitPoints = 1: int) + update(dt : int) : int

Chegamos ao fim da cadeia de herança. Vamos reimplementar o planeta vermelho (que deve ser usado no vetor de planetas, como antes), como filho da classe Planet.

Sobre os métodos da classe:

`+ Planet(sprite : *Sprite,  x : float, y : float, hitPoints : int) : float`

O construtor da classe. Deve chamar o construtor da classe pai (passando todos os valores)

`+ update(dt : int): int`

Deve usar a classe InputManager para testar se está sendo clicado, e caso esteja deve aplicar o dano em si mesmo (O.o).
Dica: usar os métodos InputManager::getInstance->isMouseDown() e InputManager::getInstance->isMouseInside()

Alterações no Game Manager:

Alterar o vetor de planetas para ser um vetor de **Planet** (e **não** de PlanetRed!).
Alterar o método addPlanet para criar um novo PlanetRed.

Criar um novo método na classe GameManager, update(dt : int). Todo o update do jogo deve acontecer dentro desse método. Dentro desse novo método, atualize a posição da camera, chame o método update(dt : int) de todos os planetas no vetor (onde eles calcularão seu dano) e em seguida chame o método checkPlanets.

Criar um novo método na classe GameManager: render(cameraX : float, cameraY : float). Toda a renderização do jogo deve acontecer dentro desse método. Ou seja, mova todas as chamadas de renderização que estavam "soltas" no run para este novo método: A renderização do background, do tileMap e dos planetas.

Após essas mudanças o jogo deve continuar funcionando corretamente.

6. Earth: Movimento Controlado

Earth : Planet
- vx, vy : float
+ Earth(sprite : Sprite*, x : float, y : float, hitPoints = 1: int) + update(dt : int) : int

Agora vamos fazer um planeta que se movimenta de acordo com o input do teclado.

Sobre os atributos da classe:

`- vx, vy : float`

Guardam a velocidade nos eixos x e y do objeto.

Sobre os métodos da classe:

`+ Earth(sprite : *Sprite,  x : float, y : float, hitPoints = 1: int) : float`

O construtor da classe. Deve chamar o construtor da classe pai (passando todos os valores)

`+ update(dt : int): int`

Deve usar a classe InputManager para se movimentar, usando as teclas **ASWD**. Para isso: testar se cada uma dessas teclas está pressionada e atualizar sua velocidade XY, em função do tempo dt.

Dica: Zere a velocidade, e então verifique se o jogador está apertando D (use usar o método InputManager::getInstance->isKeyPressed()) e faça vx = c*dt. Se o personagem estiver andando para trás, faça vx = -c*dt, sendo c uma constante que desejar (c = 0.3 é um bom número). Faça o mesmo com o eixo y. Depois incremente os valores de x e y de acordo com a velocidade.

Alterações no Game Manager:

Criar uma Terra no jogo.
Para isso:

- Inclua a classe Earth.
- Declare a earth e sua sprite.
- Inicialize a earth e sua sprite.
- Adicione-os ao destrutor.
- Adicione o update da Earth ao update da GameManager.
- Adicione o render da Earth ao render da GameManager.

7. Moon: Movimento Não-Controlado

Moon : Planet
- radius : float - angle : float - center : Planet*
+ Moon(sprite : *Sprite, hitPoints = 1: int, center : Planet*) + update(dt : int) : int

Por fim, vamos fazer uma lua que se movimenta de acordo com regras estabelecidas no update, sem interferência do usuário. No caso, a lua deve girar em volta de um planeta (orbitar).

Sobre os métodos da classe:

`+ Moon(sprite : *Sprite, hitPoints = 1: int, center : Planet) : float`

O construtor da classe. Deve chamar o construtor da classe pai (passando a sprite, hitpoints e a posição XY do planeta centro). Alé disso, deve inicializar o raio (de acordo com o raio do planeta centro) e o ângulo inicial com 0.

`+ update(dt : int): int`

Atualizar o ângulo de acordo com o dt, de modo que o movimento fique suave. Então atualizar a posição da lua de acordo com o ângulo e a posição do planeta centro.

Dica: Ao atualizar o ângulo, a mesma fórmula que usamos no movimento linear vale para o movimento angular: angulo = c*dt (onde 0.2 é um bom valor para c). Use a função seno para calcular o deslocamento no eixo x e a função cosseno para calcular o deslocamento no eixo y. Some este valor à posição do centro do planeta, levando em consideração o raio da lua e do planeta.

Alterações no Game Manager:

Criar uma lua orbitando a Terra no jogo.
Para isso:

- Inclua a classe Moon.
- Declare a moon e sua sprite.
- Inicialize a moon e sua sprite.
- Adicione-os ao destrutor.
- Adicione o update da moon ao update da GameManager.
- Adicione o render da moon ao render da GameManager.

8. Classe TileSet : Armazenando as Sprites dos Tiles

TileSet
- tileWidth, tileHeight : int - lines, columns: int - tileSet : Sprite* - vTiles : std::vector<Sprite *>* - destRect : SDL_Rect*
+ Tileset(filePath : std::string, lines : int, columns : int); + Tileset(tileWidth : int, tileHeight : int); + addTile(filePath : std::string) : void + render(index : int, posX : float, posY : float) : void + usingSingleFile() : bool + getTileWidth() : int + getTileHeight() : int

A classe TileSet será responsável por armazenar todas as Sprites (Tiles) utilizadas na renderização do TileMap. Ela permite 2 formas de armazenamento desses Tiles: Como um TileSet (imagem única com vários tiles) ou como um vetor de tiles (cada Tile está armazenado em uma imagem separada).

Atributos da classe:

- `tileWidth, tileHeight` : Armazenam a largura e a altura dos tiles.
- `tileSet` : Armazena um arquivo único contendo todo o tileset.
- `lines, columns` : Armazenam o número de linhas e colunas do tileSet (no caso de utilizar arquivo único)
- `destRect` : Armazena o retangulo que sera usado para clipar os tiles do tileSet (no caso de utilizar arquivo único)
- `vTiles` : Armazena os Tiles caso sejam carregados a partir de arquivos separados

Implementando os métodos da classe:

`+ Tileset(tileWidth: int, tileHeight: int, filePath: std::string)`

- Construtor da classe que inicializará o TileSet com uma única imagem de tileset.
- Inicializa atributos da classe: useSingleFile como true, tileWidth e tileHeight com os parametros passados, a vTiles como NULL.
- A Sprite tileset deve ser carregada a partir do `filePath`.
- O número de lines e columns deve ser calculado de acordo com o tamanho da imagem do tileSet carregado e dos valores tileWidth e tileHeight passados.

`+ Tileset(filePath: std::string, columns: int, lines: int)`

- Construtor Opcional: Criar um construtor quase igual ao anterior, mas ao inves de calcular o número de linhas e colunas a partir do tamanho dos tiles, fazer o inverso.

`+ Tileset(tileWidth : int, tileHeight : int);`

- Construtor da classe que será usado para inicializar um tileset vazio. Seus tiles serão adicionados posteriormente usando o método addTile.
- Inicializa todos os atributos da classe: useSingleFile como false, lines e columns como 0, tileWidth e tileHeight com os parametros passados, a Sprite tileSet como NULL e cria um novo vTiles.

`+ addTile(filePath : std::string) : void`

- Testar se o mapa foi inicializado para usar tileSet (arquivo único) ou não. Caso não, carrega o tile e coloca no vTiles.

`+ render(index : int, posX : float, posY : float) : void`

- Testar se o mapa foi inicializado para usar tileSet (arquivo único) ou não.
- Caso sim, deve-se calcular a posição do retângulo de clipping: O index deve ser mapeado para linha e coluna do tileSet, multiplicado pelos respectivos valores de altura e largura, passado para o tileSet através do método clip, e só então deve ser dado o render(posX, posY);
- `Caso não`, renderizar o tile da posição index do vTiles.

`+ usingSingleFile() : bool`

- Retorna True se o Tileset foi inicializado com arquivo único (se tileSet != NULL) ou false se foi inicializado para utilizar um vetor de Tiles (se tileSet == NULL).

`+ getTileWidth() : int`

- Retorna a largura dos tiles.

`+ getTileHeight() : int`

- Retorna a altura dos tiles.

9. Classe TileMap: O Mapa

TileMap
- tileMatrix : std::vector<std::vector <std::vector <int> > > - tileset : Tileset* - mapWidth, mapHeight, mapLayers : int
+ Tilemap(mapWidth : int, mapHeight : int, tileSize : int, layers = 1: int, tileSet = NULL : Tileset *) + Tilemap(mapa : std::string, tileSet = NULL: tileSet *) + load(mapPath : std::string) : void + setTileset(tileset : Tileset*) : void + at(x : int, y : int, z : int = 0) : int& + render(cameraX = 0.0 : float, cameraY = 0.0 : float) : void + width() : int + height() : int + layers() : int

A classe TileMap será responsável por armazenar todas as informações do mapa, o que no caso mais simples são os índices dos tiles para renderização.

Atributos da classe:

- `tileMatrix`: Matriz para armazenar as informações de cada tile.
- `tileSet` : Ponteiro para o Tileset que será usado na renderização.
- `width, height, layers` : Armazenam o número de linhas, colunas e camadas do

Implementando os métodos da classe:

`+ Tilemap(mapWidth : int, mapHeight : int, tileSize : int, layers = 1: int, tileSet = NULL : Tileset *)`

- Inicializa as variáveis do TileMap: width, height, layers e tileset.
- Redimensiona a tileMatriz para as dimensões corretas e a inicializa com -1.

`+ Tilemap(mapa : std::string, tileSet = NULL: tileSet *)`

- Define o tileset e inicializa width, height e layers com 0.
- Chama o método load(mapa) para carregar o tileMap do arquivo.

`+ load(mapPath : std::string) : void`

- Carrega o mapa a partir de um arquivo. O formato desse mapa pode e deve ser definido arbitrariamente.
- Sugestão: Usar um formato que pode ser gerado por um editor de TileMap, como por exemplo o Tiled. Se esse formato se mostrar muito complicado, usar uma simplificação.
- Width, height e layers devem estar definidos no arquivo do tilemap.

`+ setTileset(tileset : Tileset*) : void`

- Redefine o tileset usado na renderização.

`+ at(x : int, y : int, z : int = 0) : int&`

- Retorna uma referência para o conteúdo do tileMap na posição x,y,z. Para facilitar o acesso a matrizes 2D, é definido 0 como padrão em z. Deve funciona como um acessor, igual ao da classe vector.

`+ render(cameraX = 0.0 : float, cameraY = 0.0 : float) : void`

- Checa se o tileSet não é nulo. Caso não seja:
- Itera em toda a matriz, checando se o índice armazenado é < 0. Caso seja, renderiza aquele tile.
- Para renderizar, é necessário passar a posição correta daquele tile na tela (posição do tile * dimensão do tile) e levar em consideração a posição da câmera.

`+ renderLayer(cameraX : float, cameraY : float, layer : int) : void`

- Opcional: Renderiza apenas uma camada do mapa. Istó será útil para aplicar a técnica de parallax no tilemap, que valerá pontos extras no Trabalho 2.

`+ width() : int`

- Retorna a largura do tilemap (número de linhas).

`+ height() : int`

- Retorna a altura do tilemap (número de colunas).

`+ layers() : int`

- Retorna o número de camadas do tilemap.

Alterações no Game Manager:

Para o trabalho 2, o Tilemap deve:

- Carregar e renderizar o TileSet a partir de um arquivo único de imagem;
- Carregar e renderizar um TileMap de 2 camadas, a partir de um arquivo.