Trabalho 2 – Geometria e Arrays de Objetos

1. Rect e Point

Rect
+ x, y, w, h : float

Vec2
+ x, y: float

Nossas classes de geometria são bastante simples. Vec2 expressa um vetor no R2, que pode tanto representar uma posição no espaço como uma grandeza. Rect expressa uma posição (canto superior esquerdo do retângulo) e dimensões. Exigiremos apenas esses membros para esse trabalho. No entanto, é fortemente recomendado que você desenvolva funções para suas classes, já que usaremos muitos cálculos geométricos ao longo do semestre. Algumas possibilidades:

- Construtores com inicialização em valores dados e/ou em zero
- Soma/subtração de vetores
- Multiplicação de vetor por escalar
- Magnitude
- Cálculo do vetor normalizado
  - Um vetor normalizado é um vetor unitário (de magnitude 1) com a mesma direção do vetor original
  - Matematicamente, podemos demonstrar que, ao dividir os componentes de um vetor pela magnitude dele, obteremos um vetor unitário.
- Distância entre um ponto e outro
  - Equivalente à magnitude da diferença entre dois vetores
- Inclinação de um vetor em relação ao eixo x
  - Atente para a diferença entre atan() e atan2()
- Inclinação da reta dada por dois pontos
  - A diferença entre dois vetores tem a mesma inclinação da reta - note que a ordem na subtração importa!
- Rotação em um determinado ângulo
  - O algoritmo para tal é baseado em matrizes de rotação:

        - x' = x * cosθ - y * sinθ
        - y' = y * cosθ + x * sinθ
      - Note que, para um eixo y positivo para baixo, um ângulo positivo resulta numa rotação no sentido horário.
- Soma de Rect com Vec2
- Obter coordenadas do centro de um retângulo
- Distância entre o centro de dois Rects
- Saber se um ponto está dentro de um Rect
- Operadores de atribuição, soma, subtração
  - Isso não é necessário em momento algum, mas é uma feature interessante da linguagem C++ que está descrita na seção 14 do Apoio de C++.

Lembre-se trabalharemos com um eixo y que cresce para baixo, e que as funções de trigonometria da biblioteca padrão usam ângulos em radianos.

2. GameObject: Ancestral dos Objetos

GameObject
+ ~GameObject() : virtual + Update(dt : float) : void, virtual pura + Render() : void, virtual pura + IsDead() : bool, virtual pura + box : Rect

Todos os objetos do nosso jogo terão, no mínimo, as características de GameObject. São essas:

1. Posição e dimensões (box)
2. Função para atualizar o estado de acordo com o tempo decorrido (Update)
3. Função que reuna todas as renderizações necessárias (Render)
4. Função que comunique que o objeto precisa ser deletado (IsDead)

Todas as funções serão definidas em classes filhas, onde também tratamos de inicializar box. Uma dessas classes é...

3. Face: Meu Primeiro Objeto

Face (herda de GameObject)
+ Face (x : float, y : float) + Damage (damage : int) : void + Update (dt : float) : void + Render () : void + IsDead () : bool
- hitpoints : int - sp : Sprite

Face é um "inimigo" com uma determinada quantidade de HP. (sugestão: 30 HP)

```
> Face (x : float, y : float)
```

Primeiro, deve abrir o Sprite (*img/penguinface.png*). Em seguida, deve inicializar box baseado nos valores passados. Lembre-se que box.x e box.y são o canto superior esquerdo do objeto, e se atribuirmos diretamente o x e o y passados, Face será posicionada com o canto superior esquerdo nessas coordenadas, o que nem sempre é conveniente.

Para ficar mais intuitivo, calcule valores para box.x e box.y tal que o objeto fique centralizado nas coordenadas x e y recebidas. As dimensões podem ser obtidas do Sprite carregado.

```
> Damage (damage : int)
```

Deve reduzir os hitpoints na quantidade passada.

```
> Update (dt : float)
```

Por enquanto, não temos temporização no programa. Deixe Update vazia.

`> Render ()`

Renderiza sp na posição atual.

`> IsDead ()`

Face está morto quando seus hitpoints estão menores ou iguais a zero.

4. Mudanças em State

State (membros adicionais)

`+ ~State ()`
 	
`- Input () : void`
`- AddObject (mouseX : float, mouseY : float) : void`

`- objectArray : std::vector<std::unique_ptr<GameObject>>`

Para administrar os objetos instanciados no jogo, vamos manter um array de ponteiros para GOs. Esse array é uma estrutura de dados do tipo vector. Para os não-íntimos: `<vector>` é uma biblioteca padrão do C++. `std::vector`, o tipo definido nela, é um array que sabe se redimensionar sozinho caso seu tamanho máximo seja excedido.

A `<vector>` faz parte da chamada Standard Template Library, o conjunto de estruturas de dados pré-definidas em templates na linguagem, e uma das maiores vantagens de se usar C++ ao invés de C puro. Voltaremos a usar a STL mais vezes durante o curso.

Perceba que não se trata de um vector de ponteiros, simplesmente. Estamos usando um outro template, contido em `<memory>`. É o `std::unique_ptr`. Essa classe recebe um ponteiro na sua instanciação, e se comporta como se fosse o próprio ponteiro. Sua importância está no fato de que, quando o unique pointer é apagado ou sai do escopo, a área de memória para a qual o seu ponteiro aponta é automaticamente liberada.

Quando trabalhamos com containers de ponteiros, um erro muito comum é remover um ponteiro do array sem usar `delete` antes. Os `std::unique_ptrs`, introduzidos no C++11, resolvem esse problema com um

overhead extremamente pequeno.

`> ~State()`

Esvazia o array de objetos (`clear`).

`> Input ()`

O corpo dessa função está disponível no Moodle. Podem ser necessários alguns ajustes nele para se adequar aos nomes de variáveis ou funções do seu código. Além disso, você pode tirar a chamada à SDL_QuitRequested em Update(), já que Input cuida de eventos de SDL_QUIT para nós.

`> Update ()`

No começo do método, chame Input(). Depois, percorra o array de objetos testando se alguma das Faces morreu. Se sim, remova-a do array (`erase`). O loop de percorrimento do array precisa usar índices numéricos, já que iteradores se tornam inválidos caso um elemento seja adicionado ao vetor (o que vai acontecer em trabalhos futuros).

Sendo assim, para obter o iterador exigido como argumento de `vector::erase`, use o iterador de início (`vector::begin`) somado à posição do elemento.

`> Render ()`

Render deve percorrer o array chamando a função Render de todos os objetos nele. Aqui, não faz diferença usar iterador ou índice.

`> AddObject (mouseX : int, mouseY : int)`

AddObject é chamada por Input e recebe a posição atual do cursor. Para esse trabalho, queremos que o Face seja instanciado a 200 pixels dessa posição, num ângulo aleatório. Coloque o ponteiro para a Face criada no objectArray. Use `emplace_back`, do C++11, ao invés de a tradicional `push_back`, para que o `unique_ptr` seja construido já dentro do vetor.

Para poder gerar números (pseudo-)aleatórios, seede a função `rand()` no construtor de Game. Use a função `srand()` (`<cstdlib>`) com `time()` (`<ctime>`) como argumento.

5. Problemas

Você pode ter percebido dois problemas particularmente graves com os componentes especificados aqui. São eles:

- **A imagem do Sprite de Face é alocado novamente sempre que uma Face é criada.** Se você segurar uma tecla com o programa já pronto, verá que o consumo de memória cresce muito, já que há várias cópias da mesma imagem nela. O correto, num jogo de verdade, é manter um índice de recursos em algum lugar. Trataremos esse problema no próximo trabalho.

- **A captura de input é feito no meio de código específico.** Como dissemos no trabalho 1, o que se espera é que a captura de Input seja feita por uma classe separada, controlada por Game. Os objetos "interessados" buscam os eventos que importam para eles, não é State quem deve notificá-los. Também mudaremos isso em breve.