Trabalho 2 – Geometria e Arrays de Objetos

1. Rect e Vec2

Rect
+ x, y, w, h : float

Vec2
+ x, y: float

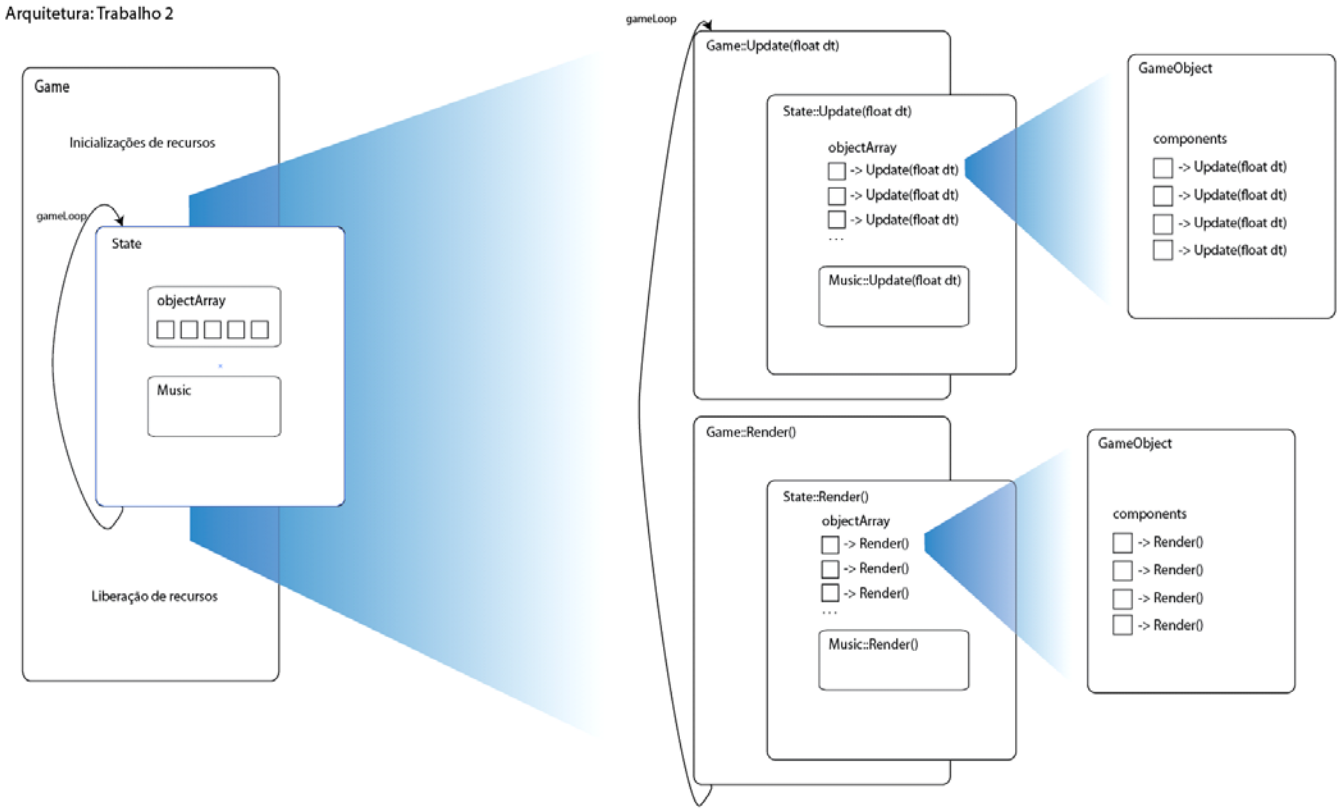
Nossas classes de geometria são bastante simples. `Vec2` expressa um vetor no R2, que pode tanto representar uma posição no espaço como uma grandeza. `Rect` expressa uma posição (canto superior esquerdo do retângulo) e dimensões. Exigiremos apenas esses membros para esse trabalho. No entanto, é fortemente 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:

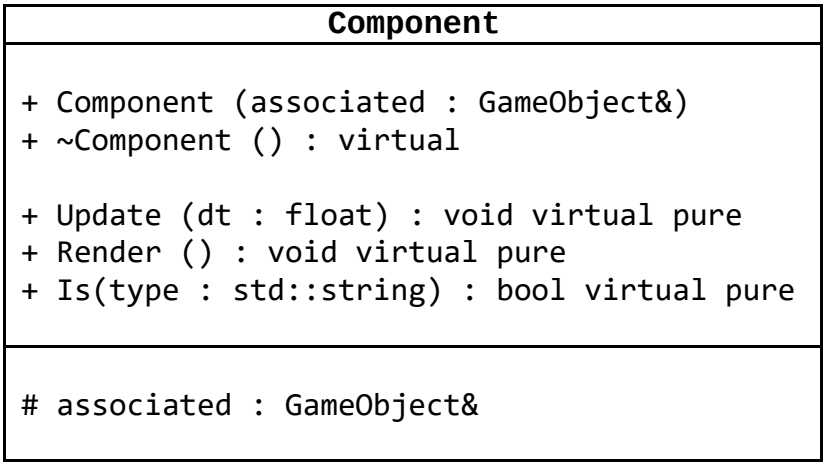
        - `x’ = x * cosθ - y * sinθ`
        - `y’ = y * cosθ + x * sinθ`
    - Note que, para um eixo y positivo para baixo, um ângulo positivo resulta numa rotação no sentido horário.
- Soma de `Rect` com `Vec2`
- Obter coordenadas do centro de um retângulo
- Distância entre o centro de dois `Rects`
- Saber 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++ e será de grande ajuda.

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. Component: Onde o jogo acontece



Observe que agora, State possui um objectArray, que será populado por GameObjects. Cada GameObject por sua vez, serve como uma caixa de Components. Ao invés de passar comportamentos semelhantes por meio de heranças, que podem se tornar longas e difíceis de manter, encapsulamos tais comportamentos em um componente, que serve como interface, trazendo maior flexibilidade ao código. O objeto que precisar daquele comportamento... adiciona o componente!



Essa é a classe que deve ser utilizada para adicionar lógica ao jogo utilizando herança. Todos os componentes do nosso jogo terão, no mínimo, as seguintes características:

- Referência ao `GameObject` que o contém.
- Função que atualiza estado do componente.
- Função para renderizar o que for necessário.
- Função para se determinar o seu tipo.

Como vocês podem observar, não temos como compilar o projeto no momento pois não criamos a classe `GameObject`. Isso não é motivo de pânico, o criaremos logo abaixo.

2.1 A Sprite na verdade é um componente!

Para isso acontecer não é necessário mudar muita coisa, é quase só colocar o `Sprite` para herdar de `Component`. As pequenas mudanças necessárias são que o Render do Sprite não precisa mais receber onde vai renderizar em seus argumentos, agora ele vai usar a box do `GameObject` que o contém (`associated`) recebido em seu construtor. Lembrando que, como ele

herda de `Component`, `Update` e `Is` devem ser implementadas, mesmo que `Update` com um corpo vazio. Não se esqueça de setar a largura e altura da box do `GameObject` que o contém (`associated`) baseado no carregado pela Sprite em seu construtor.

Lembre-se também que, quando colocamos a primeira imagem do jogo (o background), era um simples Sprite. Precisamos arrumá-lo também (assim que construirmos a próxima classe, GameObject).

3. GameObject: Aquele que organiza os Componentes

GameObject
+ GameObject() + ~GameObject() + Update(dt : float) : void + Render() : void + IsDead() : bool + RequestDelete() : void + AddComponent(cpt : Component*) : void + RemoveComponent(cpt : Component*) : void + GetComponent(type : std::string) : Component* + box : Rect
- components : std::vector <Component*> - isDead : bool

`GameObject` é um agrupador de lógicas que estarão implementadas em seus componentes. Todo `GameObject` (GO para ficar mais curto) possui uma posição no jogo (`box`). Observe o tipo do atributo components. 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.

> GameObject ()

Inicializa isDead com falso.

> ~GameObject ()

Percorre vetor de components dando delete em todos e depois dando clear no vetor. Dica: Percorra o vetor de trás para frente e chame delete usando o iterador do começo mais o index atual.

> Update (dt : float)

Percorre o vetor de componentes chamando o Update(dt) dos mesmos.

> Render ()

Percorre o vetor de componentes chamando o Render dos mesmos.

> IsDead ()

Retorna isDead.

> RequestDelete ()

Atribui verdadeiro a isDead.

> AddComponent (cpt : Component*)

Adiciona o componente ao vetor de componentes.

> RemoveComponent (cpt : Component*)

Remove o componente do vetor de componentes, isto é, se ele estiver lá.

> GetComponent (type : std::string)

Retorna um ponteiro para o componente do tipo solicitado que estiver adicionado nesse objeto. nullptr caso esse componente não exista.

4. Sound

Sound (herda de Component)
+ Sound(associated : GameObject&) + Sound(associated : GameObject&, file : std::string) + Play (times : int = 1) : void + Stop (msToStop : int = 500) : void + Open (file : std::string) : void + IsOpen () : bool + Update(dt : float) : void + Render() : void + Is(type : std::string) : void
- chunk : Mix_Chunk* - channel : int

Sound é quase a mesma classe de Music, mesmo na implementação. As diferenças estão nas funções da Mixer usadas, e no fato de que, diferente das músicas, existem vários canais diferentes para reproduzir sons, e temos que manter registrado em qual canal o chunk está tocando para podermos pará-lo se necessário.

Primeiro, adicione em Resources membros GetSound, ClearSounds, e soundTable, exatamente como os de music, trocando apenas as funções da Mixer usadas.

```
Mix_Chunk* Mix_LoadWAV(char* file)
void Mix_FreeChunk(Mix_Chunk* chunk)
```

> Play (times : int = 1) : void

```
int Mix_PlayChannel(int channel, Mix_Chunk* chunk, int loops)
```

A Mixer permite que você administre os channels manualmente, mas é preferível que você deixe ela fazer isso para você. Peça o channel -1, ela escolherá um primeiro canal vazio e retornará o número dele para você. E é por isso que alocamos 32 canais no começo. Se quiséssemos colocar um som

e não tivesse algum canal livre, ele não tocaria.

Dessa vez, loops indica quantas vezes o som deve ser repetido, ou seja, loops = 1 faz tocar duas vezes. Já nosso argumento é quantas vezes deve ser executado, ou seja, times = 1 deve tocar somente uma vez. Passe o valor apropriado para a função.

> Stop (msToStop : int = 500) : void

```
int Mix_HaltChannel(int channel)
```

Mix_HaltChannel para um canal específico, ou, se receber -1, para todos. Use o número que Mix_PlayChannel te retornou. Vamos colocar 0,5 segundos para tempo de fade out, só para não ficar abrupto.

> Open (file : std::string) : void

Pede a abertura do arquivo para Resources.

5. Face: Meu primeiro Componente de mecânicas

Face (herda de Component)
+ Face (associated : GameObject&) + Damage (damage : int) : void + Update (dt : float) : void + Render () : void + Is (type : std::string) : bool
- hitpoints : int

Face é um “inimigo” com uma determinada quantidade de HP.

> Face (associated : GameObject&)

Primeiro, deve construir sua classe mãe e depois setar o valor inicial de hitpoints. (sugestão: 30 HP)

> Damage (damage : int)

Deve reduzir os hitpoints na quantidade passada. E se ficar menor ou

igual a zero, chame o RequestDelete do GO que o contém (associated), e dê play no componente Sound de seu associated, se houver um.

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

Deixe vazio.

```
> Is (type : std::string)
```

Retorne verdadeiro se o tipo for “Face”.

6. Mudanças em State

State (membros adicionais)
+ ~State () - Input () : void - AddObject (mouseX : int , mouseY : int) : void
- objectArray : std::vector<std::unique_ptr<GameObject>>

Para administrar os objetos instanciados no jogo, vamos manter um array de ponteiros para GOs. 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 ()
```

Agora, bg é do tipo Sprite, que agora é componente. Quando você terminar de implementar a função AddObject, vai saber melhor como voltar

aqui e consertar isso.

Você vai basicamente repetir o que foi feito lá, com exceção da adição do componente Face.

`> ~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.

ATENÇÃO: O dano só é aplicado se o GameObject tiver um componente do tipo face, caso contrário, continue procurando.

`> Update ()`

No começo do método, chame Input(). Depois, percorra o vetor de GameObjects chamando o Update dos mesmos. No final, percorra o array de objetos testando se algum dos GameObjects 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)`

Primeiramente, devemos criar um GameObject que conterá as informações do nosso primeiro inimigo. Crie e adicione um componente Sprite com a imagem *img/penguinface.png.* Atribua uma posição (na box do GO) para o GameObject com base nos argumentos.

Lembrando que é bom compensar o tamanho da sprite, caso contrário

a imagem não ficará centrada nas coordenadas passadas. Você pode usar as informações do tamanho da imagem, contida na Sprite e colocadas na largura e altura da box de GameObject, para realizar o cálculo.

Depois disso, adicionemos a esse GameObject o Componente Sound usando *audio/boom.wav* e, por último, o que o define: Face.

Coloque o ponteiro para esse GameObject criado no objectArray. Use `emplace_back`, do C++11, ao invés de a tradicional `push_back`, para que o `unique_ptr` seja construído já dentro do vetor.

AddObject é chamada por Input e recebe a posição atual do cursor. Para esse trabalho, ajuste Input para que esse objeto seja instanciado a 200 pixels dessa posição, num ângulo aleatório. Para poder gerar números (pseudo-)aleatórios, seede a função `rand()` no construtor de Game. Use a função `srand()` (`<cstdlib>`) com `time(NULL)` (`<ctime>`) como argumento.

Agora que você implementou `State` usando unique pointers para manter um registro de GOs, você pode retornar à classe `GameObject` e modificá-la para, ao invés de usar ponteiros puros, usar `unique_ptr<Component>`. As funções que você terá que modificar são `~GameObject`, `AddComponent`, `RemoveComponent` e `GetComponent`. Além da declaração do vetor `components`.

7. Problemas

Você pode ter percebido dois problemas particularmente graves 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.