DOCUMENTAÇÃO / RELATÓRIO
TRABALHO FINAL
CIÊNCIA DA COMPUTAÇÃO
UFMG
PDS 2 / 2019.1

GRUPO 1:

- Laís Guimarães Lima Gomes;
- Lucas Nascimento Marinho;
- Luiz Henrique Romanhol Ferreira;
- Victor Vieira Brito Amaral Pessoa.

1. Introdução:

1.1 O jogo:

Tales of Nothing é um jogo ao estilo RPG, no qual o jogador percorrerá salas que funcionam como pontos específicos da floresta na qual a aventura se passa. Cada sala é interligada com uma ou duas outras salas, de modo que após concluir os acontecimentos da sala em que se está, o jogador seguirá para uma dessas duas salas. A escolha da sala para a qual o jogador prosseguirá varia com base em respostas e decisões feitas pelo próprio jogador na sala em que está, possibilitando desta forma, diferentes caminhos e finais para a aventura.

As salas se dividem em duas categorias principais, as salas de combate e as salas de interação, em que respectivamente o jogador pode enfrentar monstros em um sistema de batalha baseada em turnos e interagir com personagens ou objetos que encontrar pela aventura.

Ao iniciar o jogo, o jogador pode escolher entre quatro classes de personagem, sendo elas: Warrior, Mage, Ranger e Lucky. Cada uma tem seu próprio estilo de combate e possuem valores de atributos distintos.

1.2 O código:

O código do jogo é pautado no tipo abstrato Inicializer. Essa classe armazena ponteiros que concatenam as mais diferentes partes do jogo, de modo a permitir o funcionamento em conjunto das mesmas. Assim, um ponteiro de Player é definido e repassado para as salas de combate e interação.

Além disso, a classe Inicializer cria objetos de todas as outras possíveis salas, de batalha ou interação, e define ,em um boost variant, qual será a próxima sala se o jogador escolher direita ou esquerda, responder sim ou não, ganhar a batalha ou fugir.

2. Implementação:

2.1 Jogador:

O jogador possui a classe principal Player, com as funções genéricas de suas subclasses e a suas subclasses Lucky, Mage, Ranger e Warrior, sendo que cada uma dessas classes possuem uma sobrescrita dos métodos level_up e p_damage. Com isso, o jogador consegue tomar decisões sobre o que fazer em batalhas, subir de level e dar diferentes ataques nos inimigos.

Métodos mais importantes:

- ➔ p_decision : Durante uma batalha, o jogador pode escolher entre 6 opções, sendo as 3 primeiras ataques diferentes, adquiridos quando sobe de level, a 4º se refere a opção de desviar (dodge), a 5º a utilizar porção e a 6º a fugir. Esse método retorna qual foi a escolha do jogador.
- ➔ status: Printa na tela os atributos do jogador em seu estado atual,
- ➔ level_up: Aumenta o level do jogador e lhe concede melhorias nos atributos, como vida e habilidade de desvio, dependendo de qual level em que ele se encontra. Nesse método, temos um override em cada subclasse, de modo que ataque novos adquiridos variam dependendo da subclasse.
- ➔ p_damage : Durante uma batalha, ao escolher alguma das opções de ataque disponíveis, dependendo de qual classe do RPG você está utilizando (Lucky, Mage, Ranger, Warrior), os efeitos dos ataques e a forma de calcular dano variam, de modo a criar uma variabilidade entre as classes disponíveis.
- ➔ p_potion_heal : Durante uma batalha, ao escolher a opção de curar (Potion), caso o jogador ainda tenha porções em seu inventário, ele utilizará uma delas para poder recuperar vida, aumentando sua vida e diminuindo em um o seu número de porções.

2.2 Inimigos:

Os inimigos têm como classe principal Enemy, em que possui as funções genéricas de suas subclasses e as subclasses Boss, Gnome, Pumpkin, Skeleton, Slime, Wolf.Cada uma das subclasses possuem uma sobrescrita do método e_damage. Desse modo, o ataque dos inimigos variam dependendo de qual inimigo o jogador está enfrentando.

Métodos mais importantes:

- ➔ status : Printa na tela os atributos do inimigo em seu estado atual.
- ➔ e_damage : Durante uma batalha, os inimigos utilizam 3 possíveis ataques, sendo que esses ataques também variam de acordo com o inimigo enfrentado. Nesse método, cada subclasse possui, então, uma subscrita para diferenciar seu

funcionamento, porém todas retornam no final das contas o dano a ser aplicado ao jogador.

➔ p_fail: Durante uma batalha, caso o jogador falhe em desviar de um ataque inimigo, ao usar a opção de desviar, porção ou fugir, o inimigo lhe dá um ataque de oportunidade e o valor desse dano baseia nos atributos dodge_fail, potion_fail e run_fail, respectivamente. Esse método calcula o valor desse dano, usando um desses atributos como parâmetro, que é passado pelo método run_battle dependendo da situação e retorna o valor do dano final.

2.3 Dados:

Durante o jogo é necessário utilizar dados para calcular dano do jogador ou inimigo, além disso, para verificar se o jogador ou inimigo acertaram seus ataques ou se o jogador conseguiu fazer algumas ações, como desviar ou fugir do inimigo.

Métodos mais importantes:

➔ roll_dice : Recebe como parâmetro o número de faces do dado que é para ser lançado, sendo que esse valor pode ser 2/4/6/8/10/12/16/20, com isso, o método calcula um valor aleatório entre 1 e o valor máximo de faces e o retorna, para que outras funções, como p_damage ou e_damage, possam utilizar.

2.4 Sala de Batalha:

As salas de batalha funcionam com base na classe Battle_room, que contém métodos que permitem que batalha onde o jogador e um inimigo controlado pelo computador, de modo que ambos intercalam turnos para realização de suas ações.

Os métodos mais importantes dessa classe são:

- define_attacking : Durante a batalha o jogador pode escolher entre três tipos de ataque, os três só estarão disponíveis depois do usuário atingir um determinado level, e esta função configura qual o ataque principal escolhido pelo jogador.
- player_atacking : Determina se player consegue atacar o inimigo comparando se a jogada de um dado de 20 faces obtém um número maior que a armadura do inimigo. Se o jogador tiver sucesso o dano corresponde é aplicado sobre o inimigo, e sua vida é reduzida.
- enemy_attacking : Caso o jogador não tenha fugido da batalha ou tenha se esquivado do ataque do inimigo, esse método determina se o inimigo consegue atacar o jogador comparando se a jogada de um dado de 20 faces obtém um número maior que a armadura do jogador. Se o inimigo tiver sucesso o dano correspondente é aplicado ao jogador, e sua vida é reduzida.
- stop_battle : Responsável por checar se a vida do jogador ou no inimigo antigiu um valor menor ou igual a zero. Caso um desses acontecimentos tenha ocorrido, o vencedor (entidade que ainda contém vida) é configurado como vencedor usando um objeto de classe de enumeração chamada Winner, e a batalha é configurado para terminar usando a variável _battle_on .
- run_battle : Utilizando as três funções descritas acima esse método simula uma batalha, chamando um método da classe Player chamado p_decision o jogador pode escolher como irá proceder no jogo entre seis opções (Ataque 1, Ataque 2, Ataque 3, Esquivar-se, Usar Poção e Fugir da batalha). Dentre as opções , se

player tentar se esquivar, terá sucesso se sua armadura for maior do que o resultado da jogador de um dado de 20 faces, caso falhe receberá um dano respectivo ao inimigo que está enfrentando. Se escolher usar um poção, ele vai tentar se esquivar usando a mesma lógica já descrita e sua vida terá recuperação de até 5 pontos (de modo que não ultrapasse seus pontos de vida máximo). Por fim, se o jogador escolher fugir, terá sucesso caso o resultado da jogada de um dado de 20 faces for maior que a variável e_taunt do inimigo. Depois da ação do jogador o inimigo irá atacar. É checado se o jogo terminou, e o processo de repete até que um dos envolvidos morra.

2.5 Salas de interação:

As salas de interação tem como premissa a existência de apenas uma classe de sala (Basic_room) que contenha todo o necessário para a existência de qualquer tipo de sala, visto que o tipo dos objetos instanciados serão desta classe. As demais salas de interação são classes derivadas da classe inicial, que por meio do construtor e de sobrescritas (overrides) modelam a classe base de modo a formar distintas possibilidades de salas. Dessa forma, aplica-se o princípio de substituição de Liskov, o que permite uma maior facilidade no manuseio das salas ao se gerar o mapa do jogo.

Existem doze classes derivadas de salas personalizadas, sendo elas:

- Abism_death_room: sala em que o são exibidas mensagens informando que o personagem escorregou para dentro de um abismo e morre. Ao final aciona um método que altera uma variável booleana que por sua vez indica que o jogo deve ser encerrado.
- Dmg_npc: sala em que dependendo de qual versão é sinalizada no parâmetro do construtor, varia entre ser acertado por uma pedra e cair em uma armadilha. Em ambos os casos strings são exibidas informando o ocorrido e pontos de vida do personagem são descontados por meio de uma chamada de um método da classe Player.
- History_room: sala em que dependendo do valor de uma variável estática que é incrementada cada vez que uma History_room é implementada, exibe um texto que conta trechos da história do jogo.
- Left_right: sala em que o jogador se encontra em uma encruzilhada e deve apenas escolher qual caminho pretende seguir.
- Life_elixir: sala em que o jogador encontra um elixir mágico que restaura seus pontos de vida até o limite permitido e informa a situação atual do personagem. Para tal, funções da classe Player são chamadas.
- Life_fairy: sala em que o jogador encontra uma fada que restaura cinco de seus pontos de vida sem que o limite permitido seja ultrapassado e informa a situação atual do personagem. Para tal, funções da classe Player são chamadas.
- Life_potion: sala em que o jogador encontra uma poção que restaura cinco de seus pontos de vida sem que o limite permitido seja ultrapassado e informa a situação atual do personagem caso o jogador opte por usar o item, ou adicionar a poção a mochila do personagem. Para tal, funções da classe Player são chamadas.
- Mimic_death_room: sala em que o são exibidas mensagens informando que o personagem encontrou e foi atacado pelo monstro mimico, um ser que enquanto

repousa assume a aparência de um baú de tesouro, mas que ao sentir uma oportunidade de atacar, revela presas e mata aqueles que tentaram abri-lo.. Ao final aciona um método que altera uma variável booleana que por sua vez indica que o jogo deve ser encerrado.

- Pitfall_random: sala em que uma das quatro possíveis classes do jogo é sorteada e caso a classe do jogador seja a mesma o direciona para uma das rotas. Caso contrário o direciona para a rota oposta. Para tal, funções da classe Player são chamadas.
- Quiz_room: sala em que informa-se por meio do parâmetro do construtor qual a versão da sala a ser utilizada. Uma vez decidido, uma pergunta é exibida e duas possíveis respostas surgem para que o jogador opte por uma delas. Caso o comando para a resposta seja inválido, a resposta pe requerida novamente. A resposta dada pelo jogador influencia em como o jogo termina ou qual rota o jogador seguirá.
- Tip_room: sala em que uma dica para algo que virá posteriormente no jogo será informada. Essa dica varia com base no valor passado no parâmetro do construtor.
- Xp_room: sala em que o jogador encontra uma fada que lhe proporciona trezentos pontos de experiência e informa a situação atual do personagem uma vez que isso pode gerar um aumento no nível do personagem. Para tal, funções da classe Player são chamadas.

Dentre várias outras variáveis, “id” (identificador da sala) permite a distinção de objetos repetidos da mesma sala que se situam em locais diferentes do mapa. Já “player”, é um ponteiro utilizado como referência para as salas a respeito das informações do personagem, permitindo interações que podem conferir e/ou alterar pontos de vida do personagem, a classe, entre outros.

Dentro desse ramo do código duas funções se destacam. “choosing_way” é uma função que nos casos mais simples recebe um comando do teclado informando para qual lado o jogador optou por ir. Caso o comando seja inválido, uma exceção será lançada e o comando será requerido novamente. Em casos de sobrescrita do método, a escolha pode ser pré-definida ou mesmo envolver consequências distintas. Já o método “room_interaction” tem como funcionalidade ser sobrescrito em todas as ocasiões, visto que ele coordena o funcionamento da sala, realizando ações como chamada de outros métodos e análise/alteração de valores, sendo em outras palavras o ordenamento dos acontecimentos da sala em questão.

2.6 Sala Genérica:

Essa entidade é uma representação genérica de uma sala. Tem como base a classe Room. A classe contém um ponteiro para um objeto do tipo Battle_room e um ponteiro para um objeto do tipo Basic_room , mas na instanciação de um objeto da classe somente um desses tipos de sala é instanciado e o outro se torna um nullptr. Tal resultado é obtido através de um overload de construtores, de modo que dependendo dos parâmetros recebidos na instanciação de um objeto da classe Room pode as ser armazenada uma batalha ou uma interação.

Um método muito importante dessa classe é o get_room, que funciona utilizando o tipo de dado chamado variant, que faz parte da biblioteca boost. Esse tipo de dado foi usado

para que o retorno dessa função pudesse ser do tipo Basic_room ou Battle_room de acordo com o tipo armazenado na instanciação do objeto da classe Room. Um atributo do tipo variant pode ter dois tipos, mas um deles sempre será nulo, logo é solução ideal para obter qual sala está armazenada na classe. Para que esse atributo seja acessado basta usar uma função get da biblioteca boost que irá retornar o ponteiro armazenado dentro do atributo do tipo variant.

2.7 Evento:

Essa entidade representa um evento do jogo, sendo assim é a unidade básica de funcionamento. A classe pai é a classe Event. Essa classe contém um ponteiro para um objeto da classe Room, logo armazena uma sala de batalha ou uma sala de interação. Seus métodos mais importantes são game_start e pick_player. O primeiro imprime na tela uma introdução e um tutorial para o jogador sobre as principais funcionalidades do jogo. E o segundo permite que o jogador escolha qual das quatro possíveis classes deseja ser (Mage, Ranger, Warrior ou Lucky).

O destrutor da classe pai está configurado como virtual para que evite eventuais problemas nas classes filhas.

A classe filha Event_interaction configura o seu objeto do tipo Room como uma Basic_room e um identificador que é recebido como parâmetro pelo construtor da classe, esse número indica qual posição da sala no mapa do jogo (o identificador das classes de interação varia entre 0 e 27). Um detalhe importante é que como a classe Basic_room e suas filhas foram criadas respeitando o princípio de Liskov, logo os objetos da classe Basic_room são inicializados usando os construtores das classes filhas de modo a criar salas específicas que podem ser agrupadas pois são um objeto do tipo Basic_room.

A classe filha Event_battle configura o seu objeto do tipo Room como uma Battle_room, inicializa o inimigo que será enfrentado nessa batalha e um identificador é recebido pelo construtor da classe, esse número indica qual a posição da sala no mapa do jogo (o identificador das classes de batalha varia entre 28 e 38).

2.8 Inicializador:

A inicialização das principais entidades do jogo é realizada pelo classe Inicializer. Seus atributos mais importantes são _main_event, um ponteiro para objeto da classe Event responsável por armazenar informações importantes do jogo (como o ponteiro para o jogador, uma variável bool que indica se jogo ainda não acabou, e variável bool que indica se o jogador ganhou o jogo), e o vetor de ponteiros para objetos da classe Event, que armazena todas os 38 objetos do tipo Event que armazenam as 38 salas que o jogador pode entrar ao longo do jogo.

Seus principais métodos estão listados abaixo:

- inicialize_objects : Responsável por inicializar os objetos de Event usando construtores da classe Event_interaction e Event_battle, o que é possível pois o contrato das classes filha respeita o princípio de Liskov. O player de _main_event é passado para a sala (de batalha ou de interação) armazenada dentro de cada objeto de Event. E esse objetos são armazenados no vetor _choices.

- inicialize_next_turn : Configura as possibilidades de salas da esquerda e da direita para cada um dos elementos do vector _choices.
- configure_next : Primeiramente o métodos verifica se o parâmetro id_room recebido é um número válido para o identificador da sala, e então de acordo com o identificador determina se será necessário chamar o método choosing_way da classe Basic_room. Se não for necessário, o método configura a variável _next_room de _main_event como primeiro identificador do atributo _next_turn da sala armazenada no vector choices com identificador igual a id_room. Caso contrário, o método chossing_way é chamado e caso retorne 1 configura _next_room como o segundo (equivalente a ir para direita) identificador do atributo _next_room da sala armazenada no vector choices com identificador equivalente a id_rom. Senão, _next_room é configurado como o primeiro identificador (equivalente a ir para esquerda). Outro detalhe importante é que método checa se o identificador que será configurado como _next_room é igual a 50 pois 50 é o identificador dos objetos Event que armazenam um nullptr, indicando fim de jogo.
- configure_current : Configura o atributo _current_room de _main_event usando o identificador id_room recebido que o tipo corresponde a sala que contém esse identificador
- run_room : Responsável por checar se o _current_room de _main_event , ou seja , a sala que o jogador está atualmente é do tipo batalha ou de interação e chamar um dos dois métodos abaixo de acordo com o tipo de sala.
- running_battle : Chama a função run_battle da sala de batalha que o jogador está nesse momento e realiza a checagem se o jogador morreu na batalha o jogo termina (variável _game_on de _main_event é igualada a falso). Configura o _next_room com base no fato se o jogador fugiu da batalha ou não.
- running_interaction : Chama a função room_interaction da sala de interação que o jogador está nesse momento e checa se jogador morreu dentro da sala de interação.
- game_end : Responsável por checar se alguma das condições necessárias para o jogo terminar foi atingida e imprimir mensagem na tela correspondente ao fato do jogador ter ganhado ou perdido o jogo. No caso a vida do jogador pode ter chegado a 0, ter chegado a uma das salas cujo identificador indica que não tem mais sala depois.

2.9 Uso da Parte Gráfica:

A parte gráfica do jogo é considerada como uma parte extra, pois não atende aos critérios mínimos de qualidade.
Ela foi criada usando a biblioteca Allegro 5, em C++. A ideia principal foi tornar possível que outras partes do código pudessem chamar funções que desenhariam na tela o cenário e moveriam a dinâmicia de jogo conforme o desejado.

Sobre as funções declaradas como public, que cumprem essa função, há:

- cout : A função recebe quatro parâmetros, uma para cada linha impressa na tela, e é capaz de deixar a mensagem desejada no rodapé da tela, sem interromper a execução do jogo, até que jogador aperte uma tecla pré-definida.

- choose: A função, além de imprimir uma escolha a ser feita na tela, retorna essa escolha a partir de uma variável numérica no teclado, valor que pode ser armazenado numa variável pra uso futuro.
- choose_path: A função imprime o texto desejado na tela e não permite que o programa siga a execução adiante, além do loop de andar e respirar do jogo, até que o jogador decida, numa encruzilhada, qual caminho tomar. Para isso foi definido no mapa alguns caracteres que representam escolhas. Uma vez tomada a decisão, o jogador não pode voltar atrás, porque há barreiras no jogo que impedem esse movimento.
- wait_passage: Para algumas partes do jogo, foi definido que o jogador se depararia com algumas criaturas, a função wait_passage espera que o jogador chegue a um caminho reto, onde seria conveniente começar os procedimentos para tal encontro.
- spawn: Para que as criaturas apareçam no mapa é necessário utilizar essa função, ela coloca um personagem num local fixo e não visível do mapa, mas bem a frente do caminho do jogador, para dar a impressão de que o personagem em questão já estava lá.
- clean_text: Permite que o texto na tela seja limpo facilmente e espera a entrada de teclas para prosseguir adiante.
- despawn: Executa uma animação de despawn do personagem escolhido, retornando o personagem, ao final, ao lugar de início da função spawn, para que esse seja utilizado quantas vezes for necessário.

2.9 Sobre a Implementação da Tela :

A ideia que norteia a execução se desenha sobre uma matriz de caracteres que recebe seus valores de um mapa escrito num documento .txt. Esses caracteres representam elementos do TAD Bloco, e cada um é composto de muitos elementos. Para simplificar a explicação, considera-se por hora, que sejam retângulos de cores diferentes.

Cada bloco em uma posição x e y é criado como um retângulo que começa na posição (x,y)*tamanho_do_bloco e termina ((x+1),(y+1))*tamanho_do_bloco da tela, dessa forma cria-se uma tela com os elementos existentes no bloco de notas, representados por retângulos de cores diferentes.

A partir daí, duas coisas são feitas para deixar o jogo mais atrativo. A primeira é que implementa-se o movimentar da tela, na segunda, se adicionam detalhes aos blocos.Para que os blocos se movimentam, são consideradas variáveis _map_displacement_x e uma _map_displacement_y, ambas float, essas variáveis são consideradas quando se imprime coisas da tela. Para adicionar os detalhes, considera-se que existem vários retângulos para cada bloco, e cada um tem um valor horizontal ou vertical que especifica onde, dentro da lógica do retângulo maior, devem criados outros retângulos menores, e de quais cores.

Uma vez que isso é implementado, ainda não há diferença de iluminação para elementos na parte superior ou inferior da tela, uma função se encarrega disso, rgb_with_luminosity, a função pega a posição do objeto na tela e, com base nisso, retorna o valor do rgb com a

luminosidade, que considera mais a luminosidade e menos o rgb original conforme x e y estão mais perto da beirada superior esquerda da tela.

Considerando que até esse ponto existem blocos na tela que aparecem, tem detalhes, se movem conforme o necessário e apresentam o mínimo de sentido no quesito iluminação, ainda não existem personagens.

Os personagens usam a lógica de coordenadas em relação à tela, de forma que são idealmente livres para se movimentar pelo mapa sem ocupar alguma posição ou ofuscarem um bloco. Apesar disso, character é uma TAD filha de bloco, já que usam um raciocínio parecido, na verdade a mesma função, para serem escritos na tela. Os personagens que se mantêm estáticos em relação ao mapa consideram o movimento do personagem principal, e compensam esse movimento para se manterem na posição adequada.

Sobre as folhas das árvores, precisam ter tamanhos aleatórios, e por isso não podem seguir o mesmo raciocínio que os elementos anteriores. Por isso há um vetor do tamanho da largura do mapa que é inicializado com valores aleatórios dedicados ao tamanho de cada árvore nessa coordenada x do mapa. As árvores desaparecem conforme saem da tela e são replantados em outro lugar, isso para dar a impressão de que o cenário é sempre diferente. Por causa disso, as árvores precisam ser desenhadas por métodos diferentes dos anteriores.

3. Conclusão:

A implementação da base do jogo (Dados, Personagem e Inimigos) não tiveram grandes dificuldades, devido o fato de ser mais uma base e não há uma utilização de bibliotecas complexas como o caso da inicializer, utilizando a boost.

A decisão de usar a classe boost foi devido a necessidade de preservar o contrato da classe Event e suas classes filhas, de modo que Inicializer pudesse ter um vetor de objetos de Event inicializados como Event_interaction ou Event_battle, ou seja, para preservar o princípio de Liskov. Um dos desafios foi conseguir incluir a biblioteca na compilação do make file, uma vez que poderia ser instalada em diferentes diretórios no computador, isso foi resolvido incluindo a biblioteca no diretório third_parties do Git Hub, motivo pelo qual os commits de Laís Guimarães estão tão altos, uma vez que foi responsável por incluir a biblioteca no Git Hub.

Porém, a principal dificuldade encontrada foi a colaboração do grupo, em que dois membros, Luiz Henrique e Victor Vieira, sempre acabavam gerando algum problema ou atraso no projeto, sendo não fazendo a sua parte por não saber como fazer, porém sem ao menos pesquisar sobre o básico de C++, pois apenas isso era necessário; códigos e testes entregues repletos de erros, por simplesmente não testar ou compilar, por não saber como compilar no terminal, novamente, sem ao mesmo procurar saber como fazer isso. Além de ignorar qualquer regra pré definida de boas práticas ou padrões de código.
Ademais, os testes da basic_room e suas subclasses foram removidos devido ao fato de seu criador, Victor Vieira, não ter compilado os teste e nenhum deles sequer compilavam, o mesmo foi notificado, porém não corrigiu.
A parte extra não teve testes criados pelo responsável, Luiz Henrique.

4. Bibliografia:

- http://www.cplusplus.com/reference/
- https://www.boost.org/
- https://stackoverflow.com/