Universidade Federal de Minas Gerais

Programação e Desenvolvimento de Software II

André Luiz Abranches Moreira

Igor Guimarães

João Matos

Mateus Daniel Simonetti

Mateus Henrique Ferreira Magalhães

Vando Carlos Diniz Reis

Trabalho Prático de Programação e Desenvolvimento de Software II

Documentação

Belo Horizonte

2019

Introdução

Nosso grupo decidiu fazer como trabalho prático um jogo de RPG inspirado no clássico Zelda NES. A interface gráfica que decidimos usar foi a biblioteca Allegro, pois é intuitiva e também já era bem conhecida pelo grupo em geral.

A maior parte do desenvolvimento do projeto foi feito no ambiente de desenvolvimento Code::Blocks, mas nosso grupo também criou uma makefile devido ao trabalho que existe para instalar Allegro no Code::Blocks. Mais detalhes sobre a compilação e pastas do jogo estão no README localizado na página do arquivo no GitHub.

Visão Geral

O código tem cerca de duas mil linhas, possui 12 classes distintas, e cerca de 60 funções. O código gira em torno principalmente de 3 classes, a classe “Game” que carrega dados do jogo e Allegro, “Instance” que é responsável por realizar as transições e “estados” do jogo, e a classe “Entity” que por si só não faz nada, mas é a classe pai de qualquer objeto dentro do jogo.

Como mencionado antes, o jogo funciona em torno de estados, estacando múltiplas funções na pilha, embora aparente que quando você sai de um “menu” e vai para outro, na verdade o código apenas lançou outra função, e o outro menu continuará a ser executado assim que essa função terminar. Isso pode causar vários problemas se não houver cuidado, pois pode causar *glitches* estranhos, mas nosso grupo fez o melhor para acabar com esses problemas.

A biblioteca Allegro também traz consigo varias características própias, em geral ela trabalha com eventos, e utilizamos desses eventos (como passar de tempo e apertar uma tecla) para fazer o jogo rodar.

A Abstração

O código é divido em muitas partes diferentes, em geral, criar ou recriar entidades, zonas, paredes ou qualquer outra coisa do jogo se tornou exponencialmente mais fácil conforme o desenvolvimento foi progredindo graças à classe Entity.

O Encapsulamento

O encapsulamento do código também é bem alto, especialmente para as funções de hitbox, e utilizando uma biblioteca como Allegro, realmente torna os métodos bem mais encapsulados.

A Herança

A herança do código está centrada na classe Entity e seus filhos, Entity carrega praticamente todos os dados que uma entidade precisa e seus filhos têm em certos casos seus próprios dados específicos, como a classe Player e Boss

O Polimorfismo

O polimorfismo ajudou esse jogo de uma maneira significativa, utilizando dele e do método virtual “Update” na classe Entity, todas suas classes filhos tem seus próprios Updates que permite criar um vetor e dar update em todas as entidades muito mais rápido e fácil do que se tivéssemos que separar essas funções

A Modularidade

O código possui apenas um header que a Header.h, essa também garante incluir toda as outras bibliotecas que vamos precisar. Mas as implementações das classes estão bem divididas, sendo que cada uma tem seu próprio .cpp (com exceção da Entity que estão todas juntas)

O Tratamento de Exceções

A biblioteca Allegro já traz muitos tratamentos de exceções consigo, mas o código também possui varias maneiras para garantir que caso entre lixo, ele vai tentar reparar. A maior função que garante isso é a função Check(Game*, Instance*), mas muitos métodos possuem suas próprias lógicas para evitar produzir ou receber lixo, outra função que protege o sistema de lixo é a função InitCheck() que garante que o jogo foi inicializado corretamente.

void UpdateGame(Game*, Instance*);

Uma das funções mais importantes do código (se não a mais), ao receber os ponteiros de Game e Instance, ela está sempre sendo frequentemente chamada e refrescando o jogo para dar a maior fluidez possível. Devido a ela não se encaixar especificamente em nenhuma das classes principais, eh uma função void.

Ela primeiramente atualiza os cooldowns das flechas e dos ataques do jogador (para evitar spam) e verifica se os valores dos cooldowns estão dentro dos parâmetros.

Após isso ela redesenha a tela, o jogador e o mapa. E então percorre o vetor das entidades da zona que o jogador se encontra para atualiza-los e se ainda forem válidos desenha-los no mapa.

Por fim ela atualiza as informações do jogador (vida, flechas e “rupees”) e também chama a função Check antes de terminar tudo.

void Check(Game*, Instance*);

Essa função serve como barreira, e garante que algumas variáveis não vão obter valores impossíveis, serve como uma medida de segurança para evitar bugs que podem crashar o jogo.

Struct Double

Essa estrutura é muito simples, e foi criada apenas para facilitar a passagem de algumas variáveis, sua função é apenas carregar 4 variáveis:

- int i: Ajuda a salvar a posição i da matriz de zonas localizada na classe Game.

- int j: Ajuda a salvar a posição j da matriz de zonas localizada na classe Game.

- float x: Compõe o vetor velocidade de algumas entidades.

- float y: Compõe o vetor velocidade de algumas entidades junto a x.

Class Hitbox

Uma das classes mais importantes do jogo, ela que permite calcular as interações de duas entidades diferentes, pois usamos as hitbox dentro delas para saber como elas estão em relação uma à outra. Ela possui:

- float xi: x de um determinado ponto inicial.
- float yi: y de um determinado ponto inicial.
- float xf: x de um determinado ponto final.
- float yf: y de um determinado ponto final.

Com esses dois pontos, criamos uma caixa que representa a área que uma entidade ou parede ocupa. E suas funções são:

- void BoxAddX(float): adiciona no xi e no xf um valor, para evitar que eles se dessincronizem e destorçam a caixa.
- void BoxAddY(float): idem de BoxAddX só que para os pontos yi e yf.
- bool Border(): retorna true para caso a hitbox esteja no limite do mapa.
- bool Collision(Hitbox): compara duas hitboxes e retorna true caso elas se intersectem.

Class Entity

A classe pai da maioria das entidades, ela visa especialmente deixar explicito o método update para ser usado por seus filhos e já declarar as variáveis que outras entidades vão precisar.

- Hitbox box: a hitbox da entidade.

- int health: valor de int para representar vida, usando principalmente pelo Player e o Boss.

- int type: um valor de int que serve para determinar certas interações entre tipos diferentes de entidade.

- Double v: o vetor velocidade da entidade, ou no caso do player a posição dele na matriz que será passada para Game.

- int dir: usado para registrar a direção de algumas entidades para ter uma animação correspondente.

- int cd, cd_2: cooldowns de algumas ações da entidade, para tornar o jogo mais justo.

- ALLEGRO_BITMAP* sprite: é a imagem que corresponde a entidade em jogo.

- virtual Update(Game*, Instance*): função a ser sobescrita pelos seus filhos que vai atualizar a entidade.

Class Drop: public Entity

Essa classe representa os drops que os inmigos deixam quando você os derrota. É uma classe razoavelmente pequena e simples.

- void Update(Game*, Instance*): quando essa entidade é atualizada ela checa se o player entrou em contato com ela, e se entrou ela vai dar um bônus para o player, dependendo do tipo dela.

- Drop(Hitbox): cria o drop quando o inimigo é derrotado e recebe por parâmetro a hitbox dele para o sistema saber onde que o drop tem que ser posto.

Class Projectile: public Entity

Essa classe é o que representa todos os projéteis, no jogo existem dois tipos de projéteis, as flechas que são disparadas pelo jogador e as “bolas” que são disparadas pelo inimigo.

- void Update(Game*, Instance*): esse update ele vai atualizar a posição do projétil, se ele colidir com alguma parede ele será destruído. Após isso vai verificar se ele colidiu com alguma entidade, se for uma flecha e a entidade for inimiga, ele desabilitara a unidade, criara um drop e desabilitará a flecha. Se for um projetil inimigo, ele fará o jogador perder vida e então destruirá o projétil.

- void Arrow(Game*): essa função criará o projetil para ser uma flecha.

- void Ball(Game*, int): similar a função Arrow, mas essa também tem um int para mudar o sprite do projétil isso possibilita reutilizar essa função para criar um projetil similar só que com imagem diferente.

Class Player: public Entity

Como é de se esperar, essa é a maior e mais importante entidade e possui uma quantidade bem alta de funções devido as possibilidades do que o jogador pode fazer.

- char name[9]: um vetor de char que armazena o nome escolhido pelo jogador e escreve ele na tela de load dos saves.

- int registered: int que diz se o save foi ou não registrado, utilizamos disso para decidir quais saves mostrar na tela de load dos saves.

- int save: int que diz a que save o player corresponde, e é utilizado quando o precisarmos salvar os dados do jogador de volta no save.txt dele.

- int progress: esse só possuí dois valores possíveis, 0 representa quando o jogador ainda não conquistou o arco, 1 (ou maior) quando ele conquistar. Mesmo fazendo pouco, ele ainda pode ser expandido a qualquer momento.

- vector<ALLEGRO_BITMAP*> att_sprite: um vetor com os sprites de ataque do jogador, para tornar mais fácil de animar os ataques.

- int r, a: contador de rupees e flechas respectivamente.

- void Move_(Game*): MoveL, MoveR, MoveU e MoveD, move o jogador para a esquerda, direita, cima e baixo respectivamente, e também muda a direção dele, se o jogador colidir com alguma parede, o movimento é desfeito.

- void Attack(Game*, Instance*): essa função consiste em realizar a animação de ataque do jogador, ela faz isso mudando o sprite do jogador e criando novas hitboxes para poder fazer o ataque funcionar como esperado.

- void SwordCollision(Game*, Hitbox) Essa função checa se a hitbox da espada enviada acertou alguma unidade e realiza o efeito correto de ataque.

- void Save(Game*): salva as mudanças que ocorram ao jogador no seu arquivo certo.

- void Load(string): carrega todas as informações do jogador do .txt.

Class Bow: public Entity

O arco é muito similar a um drop, mas ele é um pouco mais especial porque não pode ser aleatório e ele muda alguns arquivos do jogado quando leva update.

- void Update(Game*, Instance*) : no seu update o arco vai alterar a hud e o mapa do jogo se o player tiver com seu progresso = 1 ou ele entrar em contato com a hitbox do arco. Se ele ele só entrar com a hitbox do arco e não estiver ainda com seu progresso = 1, vai ser realizado uma animação.

- Bow(): cria o arco na posição específica dele.

Class Shooter: public Entity

O atirador é um inimigo que vai ficar andando em uma direção, eventualmente ele vai parar, atirar, e andar de novo. Ele utiliza dos cooldowns e do projétil “bola” para isso

- void Update(Game*, Instance*): seu cooldown 1 (cd) verifica se ele já pode andar em alguma direção aleatória, o cooldown 2 (cd_2) verifica se ele pode atirar, no fim de cada update ele diminui o valor dos dois cooldowns, quando ele puder ele vai atirar na direção do jogador esperar um pouco e voltar a andar.

- Shooter(int, int): cria um shooter na posição que foi enviada.

Class Runner: public Entity

O corredor é um inimigo que quando o jogador se aproxima, ele começará a perseguir o jogador, ao entrar em contato o corredor vai ser jogado para trás e dará dano no jogador.

- Update(Game*, Instance*) : o corredor só começa a perseguir o jogador quando o jogador chega muito perto, então ele muda o seu cooldown 1 e fica “ativo”, quando o corredor persegue o jogador, ele fica constantemente atualizando seu vetor de velocidade e ele ignora paredes, ao acertar, ele tem um é jogado para longe, para dar uma chance pro jogador contra-atacar.

- Runner(int, int): cria um runner na posição que foi enviada.

Class Boss: public Entity

O boss é o objetivo principal do jogo, embora ele não seja tão difícil devido a forma que o jogo funciona, ele ainda é um desafio. Ele possui “estados” o estado 0 é o intervalo que ele tem entre os ataques, os outros estados (1, 2 e 3) são o estados que ele vai ficar executando um ataque em especifico de 3 que ele possui, ao levar 10 de dano e for derrotado, será mostrado a tela de vitória e o jogo estará concluído.

- int state: é a variável que controla o estado de ataque do boss.

- int hit: essa variável controla se o player já levou dano do ataque, para evitar que ele dê mais dano no player do que deveria.

- void Update(Game*, Instance*): essa função primeiramente checará a vida do boss e se for menor que 0, desativará a entidade. Quando o cooldown 1 for igual a 0, ele vai aumentar o cooldoown 2 e ativará um novo estado (que não seja 0) e vai permanecer nele até o cooldown 2 chegar a 0 o boss vai para o estado 0, onde ele vai permanecer um tempo reduzindo o cooldown 1 para 0 e repetindo o ciclo. Cada estado é um ataque diferente.

- Boss(): essa função vai criar o boss na posição dele

Class Zone

A classe de zona carrega as informações especificas de uma zona, como os inimigos nela e as paredes, além de ter funções para auxiliar a interação com essas informações.

- vector<Hitbox> wall: esse vetor carrega uma série de hitboxes que representam as paredes que as entidades tem que respeitar

- vector<Entity> e: o vetor de entidades carrega todas as entidades que estão naquela zona

- void Move_(Game*, Instance*): MoveR, MoveL, MoveU, MoveD, similar ao Move_Player, só que esse faz as transições da zona

- void BossZone(Game*, Instance*): realiza a animação para a sala do boss.

- void Load(string file_path): vai verificar o arquivo no path indicado e carregar todas as paredes encontradas nele

- bool WallCheck(Hitbox): essa função vai verificar todas as paredes da zona e retornar true se houver alguma colisão com a box indicada

Class Game

A classe game carrega muitos dados que fazem o jogo funcionar, por esse motivo no código ela está quase sempre sendo enviada por referencia, devido a sua necessidade e utilidade.

- ALLEGRO_DISPLAY* d: é a janela do jogo.
- ALLEGRO_EVENT_QUEUE* q: é a fila de eventos do Allegro, o que permite checar os inputs do jogador.
- ALLEGRO_TIMER* t: é um timer que faz o jogo rodar numa frequência correta caso não haja outro input do usuário.
- ALLEGRO_EVENT ev: os eventos que acontecem no sistema
- Player* player: o jogador principal e seus dados
- Zone zone[3][7]: uma matriz com as zonas, isso ajuda a saber facilmente em que zonas estamos sem precisar carregar cada uma individualmente
- bool playing: quando essa se torna falsa, o usuário não pode mais jogar e o jogo fecha, voltando para o Menu, porém ele pode ser inicializado novamente sem necessidade de reiniciar a aplicação.
- bool running: booleana que dita se a aplicação deve continuar sendo executada ou não.
- Double c: salva a posição na matriz de zonas.
- InitCheck(): checa se o jogo foi inicializado corretamente usando as rotinas Allegro.
- LoadZones(): carrega todas as paredes do jogo e posiciona todos os inimigos.
- Game() : Inicializa Allegro e registra todas as variáveis da classe.
- ~Game(): Libera todas as imagens e memorias alocadas e desinstala Allegro.

Class Instance

A classe instance é a maior classe do jogo, ela que realiza a transição de estados de jogo e também carrega a maioria das imagens e efeitos visuais além do jogo em si.

- ALLEGRO_BITMAP*:
  - strt_background[3]: um vetor de backgrounds para poder fazer a animação na tela inicial
  - load_background: o background da tela de load
  - register_background: o background da tela de registro
  - elimination_background: o background da tela de eliminação
  - defeat_background: background da tela de derrota
  - victory_background: background da tela de vitória
  - map_background: o mapa do jogo
  - hud: a hud do jogo
  - pause_screen: tela de pause;
  - heart: sprite do coração cheio, também usado como ponteiro nas seleções
  - empty_heart: sprite do coração vazio
- ALLEGRO_FONT*
  - f: fonte de 24 pixeis
  - f_1: fonte de 48 pixeis
  - f_2: fonte de 72 pixeis
- void StartMenu(Game*): Essa função inicia o menu inicial, que tem uma animação e espera até o jogador apertar start (enter).
- void FileMenu(Game*): nesse menu você pode ver todos seus saves e selecionar com qual você quer jogar, também pode deletar ou criar novos saves dependendo da situação
- EliminationMenu(Game*): permite que um save seja “deletado” por fazer o registro dele ser igual a 0 no arquivo
- void MainGame(Game*): inicia o jogo principal, é um estado relativamente bem simples e com bastantes métodos envolvidos
- void PauseScreen(Game*): salva e pausa o jogo, o usuário tem então a opção de continuar ou voltar para o menu principal
- void Victory(Game*): salva o jogo e espera o jogador apertar enter, ao apertar retorna o usuário para o menu principal
- void Defeat(Game*): similar ao Victory(Game*) mas não salva as mudanças do jogador
- void Register(Game*, string, int): registrará o player no save enviado por string, o int é utilizado para o programa saber onde desenhar o nome
- void DisplayHealth(int, int, int): Mostra uma quantidade de vida usando os sprites de coração na posição enviada

- void DisplayNum(int, int, int): similar ao DisplayHealth, mas esse mostra apenas números de até 3 dígitos
- Instance(): carrega todas as imagens e fontes do jogo
- ~Instance(): livra a memória usado pelas imagens e fontes