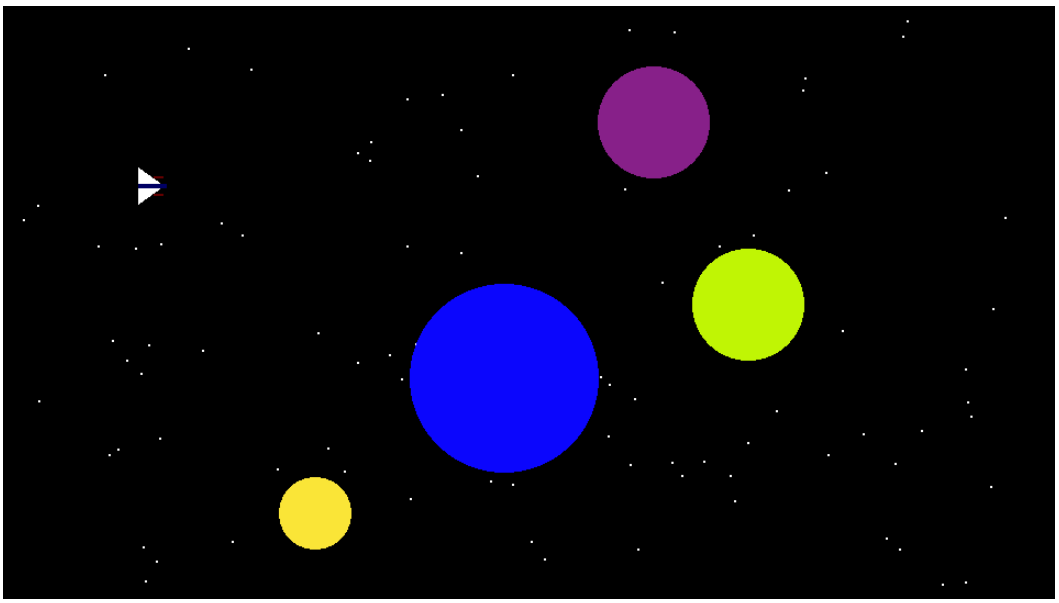
UFMG UNIVERSIDADE FEDERAL DE MINAS GERAIS

ICEx Instituto de Ciências Exatas

R-type



(Documentação)

Aluna: Maria Eugênia Reis da Fonseca
Professor: Pedro O.S. Vaz de Melo

Belo Horizonte
Fevereiro de 2022

1 - Introdução

O trabalho desenvolvido neste Trabalho Prático foi inspirado no jogo R-Type, um vídeo game arcade lançado em 1987 pela produtora Irem. Na aventura, a humanidade luta contra a raça alienígena chamada Bydo, cujo principal objetivo é destruir a raça humana. O jogador é então responsável por pilotar a nave “R-9” através do espaço para derrotar seus inimigos. Para tal, sua nave é equipada com armas que disparam raios de diferentes intensidades e aniquilam os adversários no impacto. Na versão simplificada desenvolvida neste TP os projéteis disparados pela nave jogadora são círculos que assumem diferentes tamanhos à medida que o player segura a tecla espaço, e, diferentemente da versão original, o jogo possui um nível único que só termina quando o jogador colide com um inimigo (círculo) ou com um bloco retangular.

2 - Jogabilidade

Os controles de movimento do jogo são bem simples. Para controlar a nave, o jogador deve utilizar as seguintes teclas:

W - move a espaçonave para cima;
S - move a espaçonave para baixo;
A - move a espaçonave para a esquerda/para trás;
D - move a espaçonave para a direita/para a frente;
Tecla Espaço - dispara os projéteis circulares e incrementa o raio dos disparos conforme o jogador segura a tecla.

3 - Allegro

O jogo foi desenvolvido através da engine Allegro 5 que oferece funções para gráficos, sons, dispositivos de entrada dentre outros, para as linguagens C e C + +. Para desenvolver o jogo utilizando a biblioteca foi necessário fazer uso de algumas funções básicas que garantem a integração adequada de diferentes módulos.

3.1 - Rotinas de Inicialização

A função `al_init()` garante que a biblioteca Allegro seja inicializada e, por isso, deve ser chamada antes de outras funções da engine. Outras funções de inicialização também são essenciais para garantir a funcionalidade, como a `al_create_display` que cria a janela onde o jogo será exibido - retornando um tipo

`ALLEGRO_DISPLAY` - e a função `al_install_keyboard`, que inicializa o teclado para receber o input do jogador.

A utilização das funções pode variar de acordo com a forma que se pretende jogar, também sendo possível receber input de outras fontes, como mouse, por exemplo.

3.2 - Objetos

Os objetos do jogo são os principais elementos que aparecem na tela do jogador, tendo sido definidos através do comando `typedef struct`, cada qual com diferentes campos. As entidades definidas foram: I) `Nave`, estrutura que representa o jogador; II) `Bloco`, que representa os obstáculos retangulares a serem evitados e formas transformadas; III) `Ponto`, utilizado para definir as coordenadas centrais de Inimigos e Balas; IV) `Inimigo`, que representa os inimigos circulares; V) `Balas`, estrutura utilizada para gerar os projéteis disparados pelo jogador.

3.3 - Procedimentos de criação de objetos

Para cada estrutura mencionada acima existem diferentes procedimentos:

`Nave`

`void initNave(Nave *nave)` é o procedimento responsável por definir os campos da variável *nave. Desta forma, as coordenadas (x,y), direção, cor e velocidade da nave são definidos aqui;

`void desenhaNave(Nave nave)` é o procedimento que desenha a estrutura, ou seja, faz com que a nave apareça na tela com as cores e formas estabelecidas;

`void atualizaNave(Nave *nave)` permite a devida atualização do objeto em tempo de execução do jogo.

`Bloco`

`void initBloco(Bloco *bloco)` é o procedimento responsável por definir os campos da variável *`bloco`. Desta forma, aqui são definidas as coordenadas (x,y), a largura, altura, cor e o campo “ativo” que controla se o objeto está ativo ou não na tela;

`void desenhaBloco(Bloco bloco)` é o procedimento que desenha a entidade com os campos previamente estabelecidos;

`void atualizaBloco(Bloco *bloco)` permite a atualização do objeto em tempo de execução do jogo.

Inimigo

`void initInimigo(Inimigo inimigo[ ], int tamanho)` procedimento responsável inicializar o array de `NUM_INIMIGOS` e configurar os valores para os campos “ativo” e vel (velocidade) dos inimigos;

`void inimigoRandom(Inimigo *iniRandom)`, parecida com as funções de definição dos objetos acima, esse procedimento “seta” os campos de um inimigo e foi separada da anterior para facilitar a manipulação;

`void soltaInimigo(Inimigo inimigo[ ], Bloco ret, int tamanho)` é o procedimento que controla a liberação de inimigos na tela sob determinadas condições. Aqui é feito o controle de colisão entre os próprios inimigos ao serem iniciados na tela, e entre os inimigos e um bloco que esteja ativo, não permitindo que essas entidades sejam criadas caso haja sobreposição de objetos;

`void desenhaInimigo(Inimigo[ ], int tamanho)` é o procedimento que desenha um inimigo na tela de acordo com os campos configurados previamente;

`void atualizaInimigo(Inimigo inimigo [ ], int tamanho)` permite a atualização dos inimigos em tempo de execução do jogo.

Balas

`void initBalas(Balas balas[ ], int tamanho)` responsável por inicializar o array de `NUM_BALAS` e definir valores para os campos raio, velocidade e “atirou”;

`void atiraBalas(Balas balas[ ], int tamanho, int carregandoTiro)` é o procedimento que controla o disparo de uma bala na tela sob determinadas condições; ela é chamada quando o jogador solta a tecla espaço dentro do loop principal;

`void atualizaBalasAtivas(Balas balas[ ], int tamanho)` é o procedimento responsável por atualizar as balas que estão ativas na tela, desativando-as quando ultrapassam o limite da tela e restabelecendo o valor mínimo do raio;

`void desenhaBalasAtivas(Balas balas[ ], int tamanho)` procedimento que implementa o desenho das balas ativas na tela do jogador;

`void posicionaBalasCarregando(Balas balas[ ], Nave nave, int tamanho, int carregandoTiro)` esse procedimento é responsável por posicionar e desenhar a bala na tela, diante da nave jogadora. Aqui a bala ainda não foi disparada e não está percorrendo (“atravessando”) a tela;

`void atualizaBalasCarregando(Balas balas[ ], int tamanho, int carregandoTiro)` é o procedimento que atualiza o tamanho do projétil antes de ser disparado, ou seja, enquanto a bala está sendo carregada seu raio é incrementado até atingir o raio máximo (`RAIOESPECIAL`), quando esse valor é obtido o tiro para de ser carregado/incrementado.

O cenário do jogo, apesar de não ter sido definido como uma estrutura também possui suas próprias funções:

`void initGlobais( )` é responsável por inicializar a variável referente à cor da tela do jogo;
`void desenhaCenario( )` é responsável por “pintar” a tela com a cor definida na função acima;
`void estrelas( )` é responsável por desenhar um array de estrelas no cenário, dividindo-o em três planos possíveis para criar um efeito multidimensional.

3.4 - Procedimentos de Transformações

Para lidar com os objetos criados e controlar as colisões dentro do jogo foram implementados procedimentos de transformações, que basicamente alteram as formas geométricas dos objetos dependendo das entidades envolvidas na colisão:

`void trNaveEmBloco(Bloco *ret, Nave nave)` é o procedimento que faz com que a nave jogadora assuma propriedades de um `Bloco`;

`void trInimigoEmBloco(Bloco *ret, Inimigo *c)` é o procedimento que faz com que um inimigo assuma propriedades de um `Bloco`;

`void trInimigosEmBloco(Bloco ret[ ], int tamanho, Inimigo c[ ])` é o procedimento que transforma um array de `Inimigos` em um array com propriedades de `Blocos`;

`void trBalaEmBloco(Bloco *ret, Balas *bala)` é o procedimento que faz com que uma `Bala` assuma propriedades de um `Bloco`;

`void trBalasEmBloco(Bloco ret[ ], int tamanho, Balas bala[ ])` é o procedimento que transforma um array de `Balas` em um array com propriedades de `Blocos`.

3.5 - Funções de colisões

Para detectar as colisões entre os objetos do jogo foram implementadas funções de colisões:

`int colisaoBlocos(Bloco a, Bloco b)` é a função que retorna a colisão entre dois blocos ativos;

`float distancia(Ponto p1, Ponto p2)` é a função que calcula a distância euclidiana entre dois pontos, necessária para as funções de colisão entre círculos;

`int colisaoInimigos(Inimigo *c1, Inimigo *c2)` é a função que detecta a colisão entre dois inimigos, que são círculos;

`int colisaoInimigosAtivos(int i, Inimigo inimigo[ ])` é a função responsável por detectar a colisão entre inimigos ativos na tela, que podem possuir velocidades diferentes; essa função é chamada no procedimento `atualizaInimigo` já mencionado;

`int colisaoNaveBloco(Nave nave, Bloco bloco)` é a função que retorna a colisão entre a nave jogadora e um `Bloco`; para tal, ela primeiramente chama o procedimento `trNaveEmBloco` e, em seguida, a função `colisaoBlocos`;

`int colisaoNaveInimigos(Nave nave, Inimigo inimigo[ ])` é a função responsável por retornar a colisão entre a nave jogadora e um array de inimigos; para tal, ela inicialmente chama os procedimentos `trNaveEmBloco` e `trInimigosEmBloco` e, em seguida, invoca a função de detecção entre blocos `colisaoBlocos`;

`int colisaoBalaBloco(Balas bala[ ], Bloco bloco)` é a função que detecta a colisão entre as balas disparadas pelo jogador e um bloco (obstáculo) ativo na tela; para isso as balas são transformadas em blocos pela função `trBalasEmBloco` e posteriormente se detecta a colisão entre os blocos com a função `colisaoBlocos`;

`int colisaoBalaInimigos(Balas balas[ ], int bTamanho, Inimigo inimigo[ ], int iTamanho)` é a função que: detecta a colisão entre as balas disparadas pelo jogador e os inimigos ativos na tela; determina se as balas são desativadas junto dos inimigos ou não, dependendo do raio que possuem ao serem disparadas ; e estabelece o aumento da pontuação ao se eliminar um inimigo.

3.6 - Função main

Na porção inicial da função main é feita a declaração de variáveis locais importantes para o funcionamento do jogo - como `ALLEGRO_DISPLAY *display`, `ALLEGRO_EVENT_QUEUE *event_queue` e `ALLEGRO_TIMER *timer` - e também se faz uso das rotinas de inicialização mencionadas na seção 3.1 deste documento. Posteriormente é feito o registro de sources - onde se registra na fila eventos de tela, de tempo e de teclado - e a inicialização dos objetos já definidos. Antes de se iniciar o loop principal são definidas três variáveis: `int playing` = 1, que é utilizada como condição de funcionamento do jogo; `int pontuação` = 0, usada para acumular os pontos do jogador ao destruir os inimigos; e `int carregandoTiro`, utilizada para definir o estado de carregamento de tiro, sendo chamada posteriormente dentro dos eventos de teclado referentes à tecla espaço.

Funções implementadas para cada objeto nas seções 3.3, 3.4 e 3.5 também são chamadas dentro do loop para garantir a atualização das entidades sob o evento do temporizador. À variável `playing` atribui-se agora os valores de não colisão entre a nave jogadora e um bloco, ou de não colisão entre a nave jogadora e um inimigo, de forma que qualquer colisão entre essas entidades leva ao encerramento do programa.

Ainda dentro de `while (playing)` é feita a verificação dos eventos de teclado, como o pressionamento das teclas responsáveis pela jogabilidade (seção 2), valendo-se destacar os eventos para os casos `ALLEGRO_KEY_SPACE` onde, caso a tecla espaço seja pressionada (`ALLEGRO_EVENT_KEY_DOWN`), a variável `carregandoTiro` é setada para 1 e se faz a chamada das funções `posicionaBalasCarregando` e `atualizaBalasCarregando.` Caso a tecla seja liberada (`ALLEGRO_EVENT_KEY_UP`) , a variável `carregandoTiro` é atribuida à 0, e chama-se a função `atiraBalas`, que libera os projéteis na tela. Finalmente, ao final da função principal acontece o armazenamento de um novo recorde (caso ocorra) e os procedimentos de fim de jogo.