UNIVERSIDADE FEDERAL DE MINAS GERAIS

PROGRAMAÇÃO E DESENVOLVIMENTO DE SOFTWARE 1

Documentação – Trabalho Prático
“R-type”

Henrique Lucas Gomes Rezende

Belo Horizonte
2022

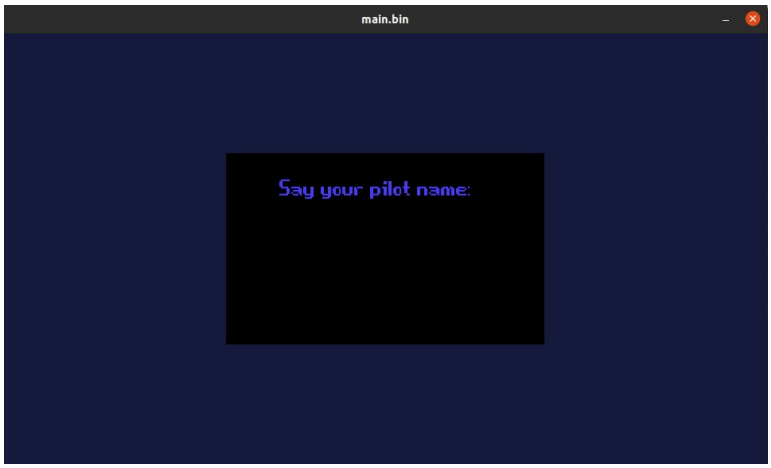
- ## Introdução

Esta documentação tem o objetivo de explicar a implementação e o funcionamento do Trabalho Prático, que visa a aplicação e o desenvolvimento de conhecimentos aprendidos na disciplina PDS 1. Neste trabalho, o aluno deve desenvolver um jogo semelhante ao clássico R-type utilizando a linguagem C e a biblioteca Allegro.

- ## Descrição do jogo

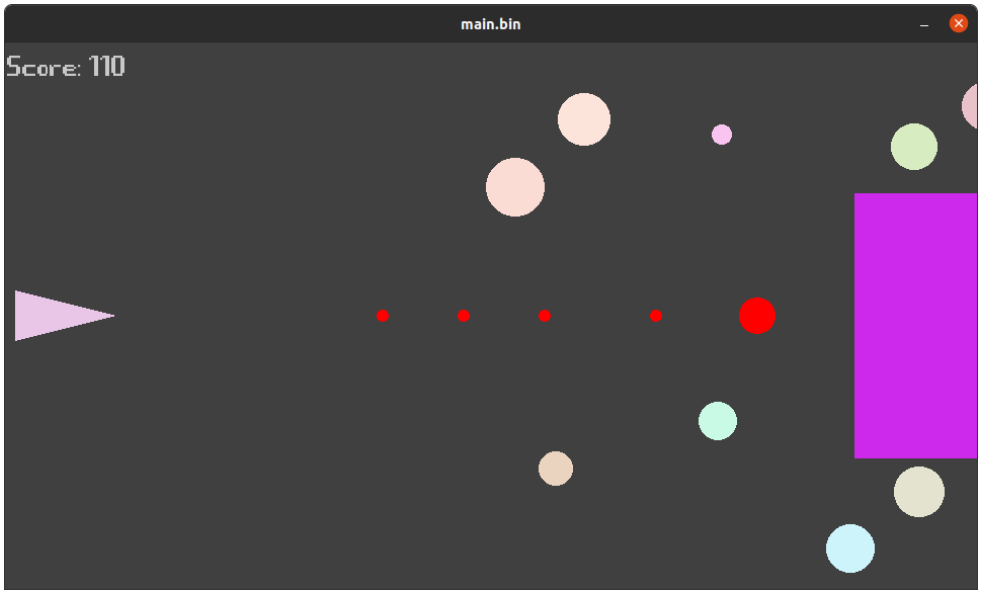
O jogo é composto de quatro telas, sendo duas delas apenas informativas e outras duas interativas.

  - ### Tela Inicial



É uma tela interativa, na qual será requisitado do jogador que digite um nome de até 15 caracteres e conclua com um Enter. Ao concluir com o Enter o jogador irá automaticamente para a Tela jogável.

  - ### Tela jogável



É uma tela interativa, na qual o jogador controla uma nave que tem por objetivo desviar de blocos e inimigos, sendo possível destruir tais inimigos com tiros variados o que acarretar no aumento de seu score. Ao ser destruído por um inimigo ou bloco, o jogo irá automaticamente para a Tela de parabéns ou Tela final.

1. Nave: representada pelo triângulo a esquerda, controlada via W (subida), S (descida), A (esquerda) e D (direita). Com o uso do Espaço é possível realizar tiros básicos (apenas um clique), médios (clicar e segurar brevemente) e carregados (clicar e segurar até o limite do carregamento).
2. Inimigos: representados pelas esferas de cores e tamanhos variados, são capazes de destruir a nave ao se chocarem com a mesma. São destruídos ao se chocarem com: tiros, blocos e outros inimigos. Realizam apenas um movimento direita → esquerda em velocidades variadas.
3. Score: representado pela numeração no canto superior esquerdo, é acrescida quando um inimigo é destruído por um tiro da nave. Quanto maior for o tamanho do inimigo maior será a pontuação acrescida, variando de 15 a 40 pontos.
4. Obstáculos: representados por retângulos de tamanhos e cores variadas, são capazes de destruir inimigos, tiros e a própria nave. Se movimentam da direita para esquerda em um velocidade fixa.
5. Tiros: representados por esferas vermelhas de tamanhos fixos e variados. Tiros básicos possuem tamanho fixo e são destruídos ao se chocarem com qualquer objeto. Tiros médios seguem mesmo padrão dos básicos variando apenas em tamanho. Tiros carregados possuem o tamanho limite fixo e somente são destruídos ao chegarem ao fim do mapa ou ao encontrarem um bloco.

- **Tela de parabéns**

É uma tela informativa que mostra ao jogador sua quebra de recorde pessoal, assim como, a sua pontuação atual. Para prosseguir basta clicar no Enter indo, automaticamente, para a Tela final.

- **Tela Final**

É uma tela informativa que mostra a lista ordenada de recordes de cada jogador. Para encerrar o jogo basta clicar no Enter.

• Descrição do código

O código é composto do arquivo main.c, como arquivo principal, e das demais bibliotecas particulares: bloco.c, inimigos.c, nave.c, pontua.c e tiro.c.

- **Structs**

1. Bloco: armazena as informações do bloco atual, como: posição x e y de seu ponto principal, altura largura e cor.
2. Inimigo: armazena informações de um inimigo, como: posições x e y, raio, velocidade e cor.
3. Tiro: armazena informações de um tiro, como: posições x e y, raio e cor.
4. Nave: armazena informações da nave, como: altura, largura, posição x e y de seu ponto principal, direções x e y de movimentação, velocidade e cor.
5. Scores: armazena informações do score do jogador atual, como: nome, pontos e um valor booleano que indica sua quebra/ou não de recorde.

O **main.c**, como próprio nome indica, é o arquivo principal do jogo. Através dele ocorrem as trocas de informações, chamadas de funções, controle de jogabilidade, etc. Descrição linha a linha:

1 a 13: importações de bibliotecas particulares e as do Allegro;

15 a 27: declarações globais de constantes, variáveis Allegro, além de structs particulares;

29 a 120: funções utilizadas no int main;

125 a 127: declaração de variáveis Allegro;

129 a 193: configurações gerais do Allegro;

195: variáveis globais iniciadas;

199 a 202: variáveis de contabilização declaradas e iniciadas;

206 a 212: declaração e inicialização de variáveis booleanas de controle ;

215 a 383: incia-se declarando a variável ‘ev’ que armazenará a fila de eventos bem como chama-se a função al_wait_forevent() que aguardaram por um evento e o armazenará em ev. Caso ev seja do tipo timer: a variável ‘cont’ sofrer um controle de tamanho pelo if; a ultima posição do nome do jogador será redefinida; o cenário será desenhado bem como a tela inicial completa. Caso ev seja do tipo close display: as variáveis playing, registry e final receberão valor false, indicando, respectivamente que o jogo não deve ser iniciado, nada deve ser registrado e a tela final não deve ser mostrada. Caso ev seja do tipo key down: a letra alfabética teclada será registrada no nome do jogador e cont será acrescida em um 1 algarismo, o mesmo valendo ao teclar espaço; ao teclar backspace, cont será decrescida em um valor caso seja maior que 0 e a posição final do nome será reposicionado; ao teclar enter será eliminado do final do nome, caso haja, o espaço final, além disso, será analisado se o nome registrado não é nulo, caso não seja a variável save receberá valor true sinalizando que o registro está permitido, por fim, a variável registry receberá valor false, simbolizando que o registro está encerrado.

386 a 493:incia-se declarando a variável ‘ev’ que armazenará a fila de eventos bem como chama-se a função al_wait_forevent() que aguardaram por um evento e o armazenará em ev. Caso ev seja do tipo timer: as funções de desenho para o cenário, score e bloco serão chamadas; a função atualizaBloco() será chamada e retornara um valor booleano indicando se a nave se chocou; um if analisará o valor de playing, antes que ele seja alterado posteriormente, para definir se houve ou não choque; a função desenhaNave() será chamada, bem como, a função atualizaNave() que retornara um valor booleano indicando se a nave se colidiu com algum inimigo; um if fará o controle do tamanho do tiro a ser disparado, caso a soma esteja permitida ela será feita até o valor de 30 e a função de desenha carregamento será chamada, caso a soma não seja permitida a chamada da função atiraTiro() será feita; por fim as funções atualizaTiro(), desenhaInimigo(), e atualizaInimigo() serão chamadas e tela jogável será atualizada por completo. Caso ev seja do tipo close display: as variáveis playing, final e fim_rec receberão valor false, indicando, respectivamente, que a tela jogável deve ser fechada, a tela final não deve aparecer e nem a tela de parabéns. Caso ev seja do tipo key down: pressionando w dir_y da nave será decrescido em 1 algarismo; pressionando s dir_y da nave será acrescido em 1 algarismo; pressionando a dir_x da nave será decrescido em 1 algarismo; pressionando d dir_x da nave será acrescido em 1 algarismo; caso espaço seja pressionado o valor de soma será alterado para true. Caso ev seja do tipo key up: não pressionando w dir_y da nave será acrescido em 1 algarismo; não pressionando s dir_y da nave será decrescido em 1 algarismo; não pressionando a dir_x da nave será acrescido em 1 algarismo; não pressionando d dir_x da nave será decrescido em 1 algarismo; caso espaço seja pressionado o valor de soma será alterado para falso, bem como, o valor de permission para true.

496 a 499: Caso save esteja permitido, a função registraPontos() será chamada e, em seguida, consultaSave() que atualizará a lista de jogadores e recordes, bem como, retornará a quantidade de pilotos registrados.

502 a 524: Caso a tela de parabéns esteja permitida e o jogador tenha quebrado seu recorde a tela de parabéns será mostrada. Incia-se declarando a variável 'ev' que armazenará a fila de eventos bem como chama-se a função al_wait_forevent() que aguardaram por um evento e o armazenará em ev. A função telaParabens() será chamada e tela geral será atualizada. Caso ev seja do tipo close display: o valor de fim_rec será atualizado para false e final receberá valor false, sinalizando, respectivamente, que a tela final não deve ser mostrada e que a tela de parabéns deve ser fechada. Caso ev seja do tipo key down: pressionando enter o valor de fim_rec será alterado para false, sinalizando que a tela de parabéns deve ser fechada.

527 a 528: Chama a função telaFinal() e logo em seguida atualiza a tela por completo.

531 a 549: Serve como uma esperá para o enter do usuário. Incia-se declarando a variável 'ev' que armazenará a fila de eventos, bem como, chama-se a função al_wait_forevent() que aguardaram por um evento e o armazenará em ev. Caso ev seja do tipo display close: final receberá valor false e a tela final será fechada. Caso ev seja do tipo key down: pressionando enter o valor de final será alterado para false, sinalizando que a tela final deve ser fechada.

551 a 552: As funções al_destroy_display() e al_destroy_event_queue() serão chamadas e, respectivamente, destruirão o display e alista de eventos antes criados.

- ### Funções

  - **initGlobais():** responsável por iniciar as variáveis globais bem como chamar as funções que assim o fazem.
  - **desenhaCenario():** responsável por desenhar a cor de fundo do cenário
  - **desenhaScore():** responsável por desenhar o número do score no canto superior esquerdo da tela
  - **telaInicial():** responsável por definir a tela inicial de inserção de nome. Desenha uma nova cor de fundo bem como um retângulo ao centro, onde será escrita a mensagem "Say your pilot name:" e logo abaixo o nome digitado pelo jogador.
  - **telaParabens():** responsável por desenhar a tela de parabéns para o jogador. Desenha uma nova cor de fundo bem como um retângulo ao centro, onde será escrita a mensagem "Congratulations NOME_JOGADOR" e logo abaixo o recorde do mesmo.
  - **telaFinal():** responsável por desenhar a tela final com a lista de todos os jogadores e seus recordes. Desenha a cor de fundo como preta, posteriormente, desenha os nome no lado esquerdo seguido de "…", por fim, desenha os respectivos recordes de cada um no lado direito.

O **bloco.c** é uma biblioteca destinada a controlar funções referentes aos blocos, como: criação, inicialização, movimentação, colisão, destruição e recriação.

- ### Funções

- **initBloco():** responsável por iniciar o ponteiro do tipo Bloco com os valores inicias e randômicos de um novo bloco aleatório seguindo as regras de limite de tamanho e localização;
- **atualizaBloco():** responsável por atualizar a posição do bloco e identificar a colisão com um bloco, inimigo ou nave. Primeiramente, atualiza a posição x do bloco, dando a impressão de movimentação. Em segundo lugar, analisa a colisão com a nave, tiros e inimigos. No caso da Nave são considerados seus y's dos pontos base e o x de seu ponto principal (bico), enquanto do bloco serão utilizados os valores x e y de seus pontos principais (pontos geradores de um retângulo). Para concluir uma colisão, analisasse a se a nave se encontra entre a parte frontal e traseira do bloco, posteriormente se ela se encontra acima ou abaixo e por fim se um de seus três pontos ultrapassa os limites superior, inferior ou frontal do bloco. Caso ela esteja antes do bloco não há como se ter colisão, caso esteja depois, analisasse se ela está entre a parte superior e inferior do bloco e por fim se a distância entre o bico da nave e a traseira do bloco seja menor que a largura nave. No caso dos inimigos e tiros, permanece a mesma logica para a posição dos blocos, tanto para os tiros, quanto para os inimigos, são utilizadas as lógicas de linhas, na qual usasse a posição dos y's e x's superiores e inferiores como linhas para a comparação. A diferença na detecção de colisões se dá pelo fato de os tiros se colidirem apenas frontalmente e lateralmente com o bloco, enquanto a nave e os inimigos se colidem com as quatro direções. Como efeito de uma colisão: a nave é destruída retornando um false; o tiro é destruído assim como o inimigo. Caso a parte traseira do bloco tenha passado o inicio da tela, um novo bloco será criado.
- **desenhaBloco():** responsável por desenhar um bloco de acordo com o bloco enviado como parâmetro ;

O **inimigo.c** é uma biblioteca destinada a controlar funções referentes aos inimigos, como: criação, inicialização, destruição, movimentação e detecção de colisão entre inimigos.

- ### Funções

  - **crialnimigo():** responsável por criar um novo inimigo iniciando-o com os valores inicias e randômicos de um novo inimigo aleatório seguindo as regras de limite de tamanho, localização e velocidade e, posteriormente, adicioná-lo ao ponteiro do tipo Inimigo na posição passada por referência;
  - **initlnimigo():** responsável inicializar cada uma das posições do ponteiro com um novo inimigo aleatório;
  - **destroilnimigo():** responsável por destruir um inimigo sobrescrevendo um novo inimigo aleatório na posição do antigo.
  - **atualizalnimigo():** responsável por atualizar a posição dos inimigos e detectar colisões entre inimigos. Primeiramente, atualiza a posição x do inimigo, dando a impressão de movimentação. Em segundo lugar, detecta se o inimigo chegou ao final da tela, caso sim, destrói esse inimigo. Para a detecção de colisões utilizasse a mesma lógica de linhas apresenta no caso de detecção de colisões do bloco. Comparasse as posições de um inimigo com todos os demais da lista de inimigos em busca de uma colisão, faz se isso até que toda a lista seja detectada. Ao detectar uma colisão, o menor inimigo nela envolvido será eliminado.

- **desenhaInimigo():** responsável por desenhar todos os inimigos da lista de inimigos;

A **nave.c** é uma biblioteca destinada a controlar funções referentes a nave, como: inicialização, desenho, movimentação e colisão com inimigos.

- ## Funções

  - **initNave():** responsável por iniciar o ponteiro do tipo Nave com os valores inicias e randômicos de uma nova Nave aleatória seguindo as regras de limite de tamanho, localização, direção e velocidade;
  - **desenhaNave():** responsável por desenhar a nave segundo a variável tipo /nave passada como parâmetro;
  - **atualizaNave():** responsável por atualizar a posição da nave e detectar colisões com inimigos utilizando os ponteiros referentes a nave e a lista de inimigos. Atualiza a posição da nave segundo os valores de dir_y e dir_x multiplicados por sua velocidade, levando em consideração os limites x e y da tela. Para a detecção de colisões, utilizasse da mesma lógica de pontos da nave utilizado na detecção de colisões com o bloco, além de utilizar a mesma lógica de pontos principais x e y utilizada na detecção de colisões com o bloco. Percorre-se toda a lista de inimigos em busca de uma colisão, caso encontrada retornas-se false;

O **pontua.c** é uma biblioteca destinada a controlar as funções referentes ao registro e consulta das informações dos pilotos e seus recordes.

- ## Funções

  - **initPiloto():** responsável por iniciar o ponteiro do tipo Scores com os valores inicias de um novo piloto seguindo de default;
  - **consultaSave():** responsável por consultar as informações do arquivo de recordes armazenando no vetor, passado por referência, o nome e a pontuação de cada piloto registrado, além de retornar o valor de pilotos registrados. Para realizar a consulta, abre-se o arquivo em questão e realiza-se a leitura da primeira linha, que contem a quantidade de pilotos registrados, posteriormente, é feita a leitura linha por linha capturando e armazenando as informações necessárias, por fim o arquivo é fechado e a quantidade retornada;
  - **comparaNome():** responsável por comparar duas strings, caracter por caracter, averiguando se as duas são iguais. Caso sejam retorna true, caso não, false;
  - **procuraPiloto():** responsável por procurar pelo piloto requisitado, salvar sua posição e retornar true caso seja encontrado, caso não encontrado, retornar falso e salvar a última posição da lista. Para realizar a consulta, abre-se o arquivo em questão e realiza-se a leitura da primeira linha, que contem a quantidade de pilotos registrados, caso a quantidade seja 0, é salvo o número 0 e retorna-se false. Posteriormente, é feita a leitura linha por linha capturando e armazenando as informações necessárias e comparando os nomes encontrados, caso algum deles seja igual ao procurado, o valor da posição é salvo e retorna-se true. Caso nenhum nome seja igual, o arquivo é fechado, a ultima posição disponível na lista é armazenada e é retornado false;

- **registraPiloto():** responsável por registrar o ponteiro do tipo Scores que contem a as informações do novo piloto na posição indicada pelo primeiro int enviado, incrementando o valor do segundo int enviado, ao valor de tamanho da lista. Realiza a leitura do arquivo semelhantemente a consultaSave. Caso o tamanho da lista esteja no máximo, o novo piloto será registrado na última posição. Caso o piloto já possua um recorde anterior, é feita a comparação da pontuação atual e antiga escolhendo e registrando a maior na lista. É feita a busca na lista por um piloto com menor pontuação que o piloto a ser registrado, caso encontrado, a lista será reorganizada para que o novo piloto esteja disposto abaixo de uma pontuação maior que ele e acima de uma pontuação menor que ele. Por fim, o arquivo e fechado e novamente aberto no modo escrita, o tamanho é registrado na primeira linha, seguido do registro de todos os pilotos com suas respectivas pontuações.
- **registraPontos():** responsável por controlar o registro de novos e recorrentes pilotos, utilizando das funções acima descritas.

O **tiro.c** é uma biblioteca destinada a controlar as funções referentes aos tiros, como: a criação, inicialização, destruição, movimentação e colisão entre tiros e inimigos.

Funções

- **initTiros():** responsável por iniciar o ponteiro do tipo Tiro com os valores inicias de um tiro nulo seguindo as regras de default;
- **destroiTiro():** responsável por destruir um tiro o sobrescrevendo. Para destruir um tiro percorre-se a lista de tiros a partir da posição do tiro que se quer eliminar copiando o próximo tiro da lista para a posição atual, fazendo com que o tiro eliminado seja sobrescrito pelos demais tiros. Por fim a quantidade de tiros é decrementada.
- **controleTiros():** responsável por identificar se o tiro, que se encontra na lista, referente a posição enviada como parâmetro chegou ao fim da tela ou se é um tiro nulo que representa o final da lista. Compara se o tiro em questão é nulo ou não, caso seja retorna 100, caso não, destrói o tiro e repassa a posição enviada;
- **atiraTiro():** responsável por registrar um novo tiro na lista com base nas posições da nave. O tiro será registrado caso haja permissão e se o seu raio maior ou igual a 4. Suas posições são baseadas nas posições da nave e seu tamanho conforme passado por referência. Por fim, a quantidade de trios é incrementada, o tamanho do tiro é redefinido e a permissão é revogada.
- **desenhaCarregador():** responsável por desenhar o carregamento do tiro com base no tamanho enviado e na posição da nave;
- **atualizaTiro():** responsável por atualizar a posição dos tiros, detectar inimigos atingidos e eliminar tiros. Percorre-se a lista de tiros até que se encontre um tiro nulo simbolizando o fim da lista, durante o percurso atualiza-se a posição de cada um dos tiros de acordo com a velocidade, além de desenhar cada tiro na tela e procurar por alguma colisão com inimigo. Para detectar inimigos atingidos, é utilizado a lógica de colisão entre inimigos, na qual um dos inimigos é substituído por um tiro. Percorre-se toda a lista de inimigos fazendo as devidas comparações, caso seja detectada uma colisão, o inimigo é destruído e se o tiro for menor que que um tiro carregado (19) ele é destruído. Por fim, caso um dos tiros tenha x>960, é

averiguado se ele se trata de um tiro que passou os limites tela e deve ser eliminado, ou se trata de um tiro nulo indicando o fim da lista.