



1 Instruções

1.1 Introdução

Project Conflict é um jogo multijogador criado para ser jogado por duas pessoas. Cada jogador possui um número limitado de pontos de vida, e quando os mesmos se esgotarem o jogador inimigo vencerá. O objetivo é acertar tiros de plasma no tanque inimigo até que os mesmos sejam depletados, lhe trazendo a vitória.

1.2 Menus

Ao iniciar o programa, você é apresentado ao menu principal. Nesse menu, você pode apertar **ESC** para ir ao menu de seleção de músicas ou **ENTER** para ir diretamente ao jogo. No menu de seleção de músicas, você pode clicar nos nomes das opções para as selecionar e apertar **ESC** novamente para voltar ao menu principal. Assim que você pressionar **ENTER** você será levado para a primeira tela de seleção de tanques. O jogador 1 deve então escolher entre três tanques diferentes, cada um com os próprios valores para vida, recarga de energia e velocidade. Assim que o primeiro player escolher, será a vez do segundo. E por fim, você será levado à tela de seleção de layout do mapa, podendo escolher qual layout de obstáculos lhe preferir. Quando a escolha for feita, o jogo se iniciará.

1.3 O Jogo

Quando iniciado, o jogo apresentará os dois players em cantos extremos da tela. Cada player possui três valores importantes: os Pontos de Vida determinam quantos tiros o jogador ainda pode levar sem ser eliminado; a Recarga de Energia determina quanto tempo demorará até o próximo tiro de plasma ficar pronto para ser disparado; a velocidade determina quão



rapidamente o jogador se move pelo cenário. Os dois primeiros valores são sempre mostrados durante a partida por meio de duas barras coloridas, como exemplificado na imagem à esquerda.

Cada tanque possui seus valores próprios e um *playstyle* correspondente. Caso um player seja atingido vezes o suficiente para ser eliminado, aparecerá a tela de vitória do player inimigo, e o número de vezes que o mesmo já venceu. Ao se apertar **ESC** durante o jogo, todos os processos pararão e a tela ficará com uma mensagem de “PAUSADO”. Ao pressionar a tecla novamente, o jogo será descongelado.

1.4 Controles

W, A, S, D: Movimentos do tanque do player 1.

↑, ↓, →, ←: Movimentos do tanque do player 2.

Espaço: Tiro do player 1.

ENTER: Tiro do player 2.

ESC: Pause/Continue.

2 O Código

Linhas 1 à 11: Includes das bibliotecas utilizadas.

Linhas 15 à 19: Struct dos obstáculos do cenário, com valores para X, Y, largura e altura.

Linhas 21 à 37: Struct dos tanques dos jogadores, com diversas variáveis destinadas ao controle de estados dos players, além dos valores de X, Y, velocidade, ângulo e componentes dos eixos.

Linhas 39 à 46: Struct dos projéteis, ou tiros, com os valores X, Y, ângulo, e variáveis de controle como id para a identificação do tiro, hit para verificação de colisão com um player, viajando para determinar quando o tiro está sendo usado e cooldown para determinar o tempo necessário para a recarga do mesmo.

Linhas 48 à 103: Declaração de todas as variáveis ALLEGRO que serão utilizadas no projeto, incluindo: `display`; `event_queue`; `timer`; `bitmaps`; `samples` e `sample_instances`; fontes; `mouse_state`.

Linhas 105 à 117: Declarações das constantes globais de controle.

Linhas 119 à 144: Declarações das variáveis globais de controle.

Linhas 146 à 161: Declarações dos objetos *struct* globais, como os tanques, projéteis e obstáculos, além da declaração da variável `FILE` destinada ao histórico de vitórias.

Linhas 163 à 207: Inicialização prévia de todas as funções do código. Sendo um código altamente modular, muitas funções foram criadas com o propósito de organizar a leitura e interpretação do mesmo.

Linhas 210 à 222: Função **Main()**. Essa função é extremamente pequena por conta dos esforços de resumir todas as rotinas em funções concisas.

//Inicializações e etc

Linhas 227 à 254: **`void initAllegro()`**: Faz todas as inicializações básicas do allegro.

Linhas 256 à 265: **`void initEventQueue()`**: Inicia a `event_queue` e registra as fontes de eventos do programa.

Linhas 267 à 283: **`void initTanks()`**: Inicia todos os valores de ambos os tanques, exceto aqueles que serão iniciados apenas pelas escolhas dos players.

Linhas 285 à 345: **`void loadMemory()`**: Carrega todos os `bitmaps`, fontes, `samples` e o arquivo do histórico de vitórias. Lê os valores do arquivo e o fecha. Começa a tocar a música base do menu do jogo.

Linhas 347 à 386: `void loadObstacles()`: Define todos os valores dos obstáculos, alterando-os de acordo com o layout escolhido.

Linhas 388 à 428: `void destroy()`: Destrói todas as variáveis ALLEGRO criadas no código.

Linhas 430 à 453: `void initMusicas()`: Define a variável ALLEGRO_SAMPLE_INSTANCE dependendo do valor de `id_musica` e a inicia.

//Renderizações

Linhas 456 à 463: `void desenhaTudo()`: Resume todas as funções de desenho em uma só.

Linhas 465 à 481: `void desenhaCenario()`: Desenha o background do jogo dependendo do valor de `id_mapa`.

Linhas 483 à 501: `void desenhaTanques()`: Desenha ambos os tanques, alterando quais *sprites* a serem usados caso os tanques levem dano.

Linhas 503 à 515: `void desenhaObstaculos()`: Desenha os bitmaps correspondentes ao valor de `id_mapa`, mudando os obstáculos de acordo.

Linhas 517 à 529: `void desenhaTiro(Projatil *tiro)`: Desenha os projéteis de cada tanque e verifica a colisão dos mesmos com obstáculos e ao sair da tela.

Linhas 531 à 582: `void desenhaTudo()`: Desenha as barras de vida e energia de ambos os tanques. Tive que criar renderizações específicas para cada número com o objetivo de os centralizar, visto que dígitos diferentes possuem tamanhos diferentes.

Linhas 584 à 614: `void desenhaMenu2(ALLEGRO_MOUSE_STATE mouse, int id)`: Desenha os menus interativos, recebendo as posições do mouse e id de qual menu desenhar.

//Seções interativas

Linhas 617 à 690: `void menu()`: Verifica qual menus renderizar, além de checar todos os cliques e interações com os mesmos.

Linhas 692 à 722: `void jogo()`: Loop principal do jogo. Chama várias das funções de inicialização e determina quais eventos renderizar por vez. Também limita os eventos no caso do jogo estar pausado e fecha as seções do jogo no loop caso a tela final tenha sido alcançada. Por último, reescreve os valores das vitórias atualizadas no arquivo de histórico.

Linhas 724 à 730: `void telaFinal()`: Desenha a tela final dependendo de qual player venceu, além de seu número de vitórias.

Linhas 732 à 778: `void events()`: Processa os eventos fora do jogo, apenas iniciando os outros eventos quando o game iniciar, chamando outra função.

Linhas 780 à 861: `void ingameEvents(ALLEGRO_EVENT evento)`: Processa os eventos dentro do jogo, como colisão e etc. É chamada pela `events()`; a partir do momento em que o jogo começa.

Linhas 863 à 914: `void escolheTanque(Tanque *player)`: Define o sprite e valores específicos dos tanques de ambos os players dependendo das suas escolhas, armazenadas em `player.id_tanque`.

//Gameplay

Linhas 917 à 936: `void moveTanque(Tanque *player)`: Atualiza a posição atual do player recebido baseado nos estados `player.dir`, `player.esq`, `player.forward` e `player.back`, realizando os cálculos das componentes vetoriais do movimento do player e os aplicando.

Linhas 938 à 952: `void criaTiro(Projatil *tiro)`: Define os valores iniciais da trajetória do tiro logo após a chamada da função.

Linhas 954 à 969: `void atira()`: Atualiza a posição do tiro baseado nos valores `projectile_speed` e `tiro.angle`, caso o mesmo esteja viajando.

Linhas 971 à 976: `void cooldowns()`: Decrementa os `cooldowns` do jogo toda vez que é chamada.

Linhas 978 à 981: `void checaVida()`: Verifica os pontos de vida dos jogadores e atualiza as variáveis `vencedor` e `finish` de acordo.

//Colisões

Linhas 984 à 1000: `void colideTudo()`: Assim como a `desenhaTudo()`, essa função resume todas as colisões em uma função só.

Linhas 1002 à 1009: `void colideTela(Tanque *player)`: Verifica se o player recebido saiu da tela, e ajusta sua posição de acordo.

Linhas 1011 à 1022:

`bool colisaoCirculoRetangulo(Obstaculo *bloco, float player_x, float player_y, int raio)`: Verifica a colisão entre os lados de um círculo e um retângulo, ignorando as quinas do retângulo.

Linhas 1024 à 1042: `void colisaoLados(Obstaculo *bloco, Tanque *player, int raio)`: Aplica a movimentação subsequente de uma colisão com os lados, verificando no processo com que lado do obstáculo o player colidiu, e alternando entre tipos de respostas de acordo.

Linhas 1044 à 1052: `bool colisaoQuinas(int quina_x, int quina_y, Tanque *player, int raio)`: Verifica a colisão entre um ponto e o player recebido, sendo utilizado apenas para a colisão das quinas dos blocos.

Linhas 1054 à 1062:

`bool colisaoCirculos(float x1, float x2, float y1, float y2, int r1, int r2, int isplayer)`: Verifica a colisão entre objetos circulares. O código é extremamente semelhante ao da função anterior. É usada na colisão entre players e entre player e projétil.

Linhas 1064 à 1080: `void colideObstaculos(Obstaculo *bloco, Tanque *player1, Tanque *player2)`: Aplica a movimentação completa pós colisão dos players com os obstáculos, chamando as funções `colisaoLados()`; e `colideQuinas()`.

Linhas 1082 à 1085: `void colideQuinas(int quina_x, int quina_y, Tanque *player, int raio)`: Aplica a movimentação direcionada às colisões dos players com as quinas dos obstáculos, impedindo-os de se prenderem nas mesmas.

Linhas 1087 à 1098: `void colidePlayers()`: Usa a função `colisaoCirculos()`; para verificar a colisão entre os dois players e responde ajustando a posição de ambos basendo-se na posição um do outro.

Linhas 1100 à 1103: `void colideTiro(Tanque *player, Projétil *tiro)`: Chama a função `colisaoCirculos()`; para verificar a colisão entre tanques e projéteis, e atualiza os valores de `tiro.viajando` e `hit` de acordo.

3 Créditos

Todas as imagens, músicas e uma das fontes utilizadas no projeto foram retiradas do jogo OMORI. A produção do Project Conflict não possui nenhuma intenção comercial.

Arte do tank do Newgrounds [aqui](#).