Scientia cum Industria

Modelo de submissão de artigos

Desenvolvimento de um Jogo em Assembly x86 Baseado no Jogo K-Star Patrol

Nome dos autores: Gabriel Tomazzoni Mazzarotto, Gabriela Procopio Bento Pereira

Resumo

Este trabalho é um relatório sobre o desenvolvimento em Assembly da arquitetura 8086 baseado no jogo lançado pela CBS Software, o K-Star Patrol, seguindo algumas simplificações no jogo original.

1. JOGO

1.1 Menu Principal

Ao iniciar a execução do jogo, é mostrado o menu principal, contendo o título escrito em 2 linhas, escrito com o auxílio de um gerador de texto em ASCII, patorjk.com/software/taag. Abaixo disso a nave aliada e a nave inimiga se alternam, atravessando a tela, a nave aliada andando da esquerda para a direita e a nave inimiga da direita para a esquerda, abaixo disso tem o texto "Jogar" e "Sair" dentro de caixas, também escritos utilizando "Strings", o texto selecionado está na cor vermelha e o outro em branco, também foi implementado uma música, que fica tocando indefinidamente dentro do menu, a música também se repete dentro do jogo. Abaixo, a imagem 1, mostra o menu do jogo desenvolvido.

Figura 1. Imagem do menu do jogo

Ao Selecionar a opção "Sair", a execução do jogo é encerrada, selecionar a opção "Jogar" o jogo em si é iniciado.

1.2 Funcionamento do Jogo

Ao iniciar o jogo, a tela exibe alguns elementos, eles são: A nave principal, 8 naves aliadas, o terreno, e uma barra de status contendo a pontuação e o tempo do setor, além de naves inimigas que surgem na tela. A Figura 2 mostra a tela inicial do jogo.

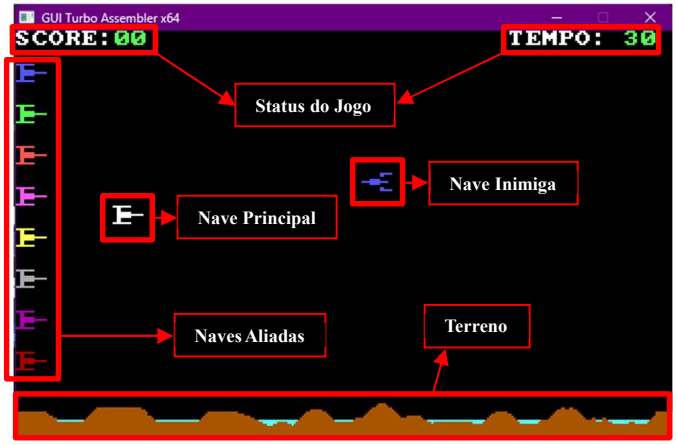

Figura 2. Tela inicial do jogo

O jogo é dividido em 3 setores, cada um dos setores possui a duração de 60 segundos, o objetivo do jogador é controlar a "nave principal" de forma a sobreviver durante a duração do tempo dos 3 setores, desviando das "naves inimigas" que andam da direita da tela para a esquerda, aparecendo em posições aleatórias na tela do jogo e também de forma aleatória se movendo para cima ou para baixo, ao atingir a nave principal, ambas as naves são destruídas e a nave aliada mais acima na tela se torna a nave principal. As naves inimigas também podem atingir as naves aliadas, caso a colisão ocorra, ambas as naves são destruídas. Caso todas as naves aliadas sejam destruídas, a tela de "game over" é mostrada e o jogo volta ao menu principal, como mostra Figura 3.

Figura 3. Tela de Game Over

A nave principal pode se mover apenas verticalmente ao clicar nas setas “cima” e “baixo” do teclado, além da movimentação vertical, a nave principal pode disparar um tiro ao clicar na barra de espaço, que surge no pixel mais a frente da nave principal e percorre a tela em linha reta, até chegar ao final dela ou então ao atingir uma nave inimiga, ao atingir, ambos os elementos são destruídos.

Ao chegar ao final de um setor, a tela de transição do próximo setor é mostrada, conforme Figura 4 e algumas alterações no jogo são realizadas:

- A cor do terreno é alterada (Setor 1: Marrom; Setor 2: Cinza; Setor 3: Roxo);
- O tempo entre o aparecimento de naves inimigas é diminuído;
- A velocidade de reprodução da música aumenta;
- A pontuação é alterada de acordo com algumas regras:
    - Setor 1 para 2:
        - São somados 1000 pontos por nave aliada ainda viva;
        - São subtraídos 10 pontos por nave inimiga que chega ao final da tela;
    - Setor 2 para 3:
        - São somados 2000 pontos por nave aliada ainda viva;
        - São subtraídos 20 pontos por nave inimiga que chega ao final da tela;
    - Final do setor 3:
        - São subtraídos 30 pontos por nave inimiga que chega ao final da tela;

Figura 4. Tela de troca de Setor

Caso o jogador consiga chegar até o final da duração dos 3 setores, ele vence o jogo, a tela de vencedor aparece, conforme Figura 5 e então o jogo retorna ao menu principal.

1.3 Música e sons de efeito

1.3.1 Música Principal

A música principal do jogo é composta de uma sequência de 36 notas que se repetem indefinidamente, a música principal do jogo é tocada no menu principal e durante toda a duração do jogo, a cada setor sua cadência é aumentada, fazendo com que as notas sejam emitidas em um intervalo de tempo menor.

Figura 5. Tela de Vencedor

A música possui uma nota principal, mais grave que fica se repetindo, e entre essa nota “base” existe algumas variações compostas de notas mais agudas. A música foi inspirada em outros jogos de arcade 8/16 bits com temática espacial.

1.3.2 Música entre setores

Quando o jogador chega ao fim de um setor, durante a tela de transição é tocado uma música, essa composta de 11 notas tocadas em uma sequência. Essa música começa em um tom mais agudo, desce para um tom mais grave e sobe novamente, passando a ideia de transição.

1.3.3 Música Game Over

Essa é composta de uma sequência de 7 notas que decrescem em seu tom, dando uma impressão de “queda”.

1.3.4 Música Tela Vencedor

Já a música que toca ao vencer o jogo é composta de 11 notas, sendo as primeiras mais graves e crescendo o tom sequencialmente, dando a impressão de subida, chegada.

1.3.5 Sons de efeito

Alguns sons de efeito são emitidos durante o jogo:

- Ao alterar as opções no menu principal;
- Quando um tiro é disparado;
- Quando ocorre uma colisão entre o tiro e a nave inimiga;
- Quando ocorre colisão entre a nave inimiga e a principal;
- Quando ocorre colisão entre a nave inimiga e as naves aliadas;

2. SOLUÇÃO

2.1 Algoritmo

O algoritmo do jogo seguiu uma lógica sequencial, que pode ser descrita nas seguintes etapas, cada etapa é compreendida por várias rotinas que serão posteriormente aprofundadas:

Etapa 1 – Inicialização

- Configuração de Variáveis e Parâmetros Iniciais: Definir posições iniciais de aliados, inimigos, sons, músicas, limites de tela e outras variáveis de controle;

- Carregamento de Música e Sons: Inicializar vetores de música para o menu, vitória, e game over, com frequências e tamanhos de cada sequência de som.
- Configuração do temporizador e dos delays.

Etapa 2 - Loop do Menu Principal

- Exibir Menu Inicial: Escreve o título do jogo e apresenta opções de "Iniciar" e "Sair";
- Desenha e movimenta as naves;
- Identifica se ocorreu a seleção de alguma das opções: Caso o jogador selecione sair, a aplicação é encerrada, caso escolha "Jogar" o laço principal do jogo é iniciado;
- Tocar Música de Fundo do Menu: Reproduz a música de fundo para o menu.

Etapa 3 - Loop Principal do Jogo

- Desenha tela inicial: Naves Aliadas, Terreno e Status;
- Entrada do Jogador: Ler controles de movimento e tiro, armazenando ações.
- Desenha novo inimigo: Verifica se é necessário desenhar novo inimigo e gera em posição aleatória;
- Atualização de Posições e Movimentação: Atualiza a posição do aliado, inimigo, e do tiro conforme os valores de delay configurados.
- Verificação de Colisão: Detectar colisões entre o tiro e o inimigo ou entre o jogador e obstáculos. E verifica se a condição de derrota foi atingida, nesse caso chamando a tela de "Game Over";
- Verificação de timer: verifica se o valor do timer deve ser decrementado;
- Configurar flags: altera flags para determinar as condições dos eventos (vitória ou alteração do setor);
- Fim do timer de setor: Verifica se o timer do setor chegou ao final e chama a transição para novo setor;
- Condição de Vitória: Verifica se o timer terminou e se o jogador sobreviveu aos 3 setores, assim sendo a tela de vitória é exibida;

Etapa 4 - Finalização e Reinicialização

- Todas as variáveis, flags e vetores são resetados para o seu estado inicial
- Retorna ao laço do menu principal

Ao Final do documento se encontra a Figura 6 que apresenta o fluxograma completo do algoritmo do jogo conforme as etapas apresentadas anteriormente.

O código do jogo possuí 2057 linhas excluindo as linhas vazias e linhas que possuem apenas comentários. Ocupando 83338 Bytes.

2.2 Memória

Para o funcionamento do algoritmo, várias informações são salvas na memória, tais como as "sprites" utilizadas durante o jogo, posições, vetores de controle de posição das naves, vetores para salvar a sequência de notas das músicas e sons reproduzidos no jogo, flags de controle, vetores de flags, sequências de caracteres para a escrita dos textos, variáveis para controle dos tempos do jogo etc.

A seguir será apresentado o detalhamento de algumas das principais variáveis e blocos de memória utilizados.

2.2.1 Sprites

As Sprites são sequências de bytes na memória de valores numéricos que representam as cores que devem ser impressas na tela a fim de "desenhar" determinado elemento do jogo, as cores são representadas por números de 0 a 15, quando determinado byte é movido para uma posição no segmento de memória de vídeo, o pixel referente aquela posição será exibido com a cor representada pelo número, dessa forma os elementos podem ser "desenhados" ao se mover as sequências de bytes para as posições desejadas no segmento de vídeo.

A seguir são apresentados as Sprites utilizadas para representar os elementos do jogo todas elas representam elementos de 9 pixels de altura e 15 pixels de largura, totalizando uma sequência de 135 bytes:

- SpriteTiro: utilizado para desenhar o tiro que é disparado pela nave, com os valores 15(branco) e 0 (preto), conforme mostra a Figura 7.

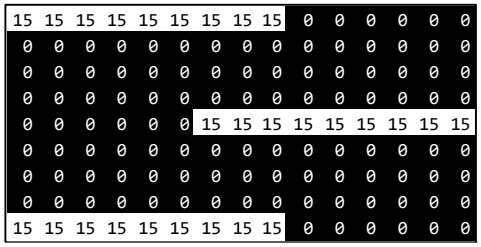

Figura 7. Representação do bloco de memória SpriteTiro.

- SpriteNave: utilizado para desenhar as naves aliadas e nave principal com os valores 15(branco) e 0 (preto), conforme mostra a Figura 8

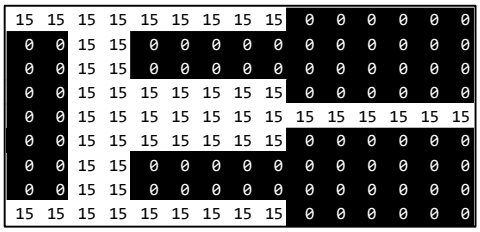

Figura 8. Representação do bloco de memória SpriteNave.

- SpriteInimiga: utilizado para desenhar as naves inimigas com os valores 9(azul claro) e 0 (preto), conforme mostra a Figura 9.

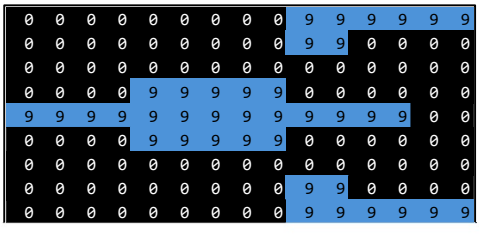

Figura 9. Representação do bloco de memória SpriteInimiga.

- explosao1: utilizado para desenhar primeiro frame da animação de explosão com os valores 14 (amarelo) e 0 (preto), conforme mostra a Figura 10.

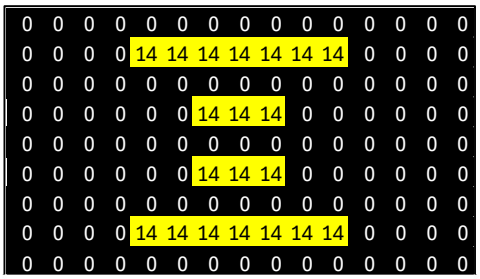

Figura 10. Representação do bloco de memória explosao1.

- Explosao2: utilizado para desenhar segundo frame da animação de explosão com os valores 12 (vermelho claro) e 0 (preto), conforme mostra a Figura 11.

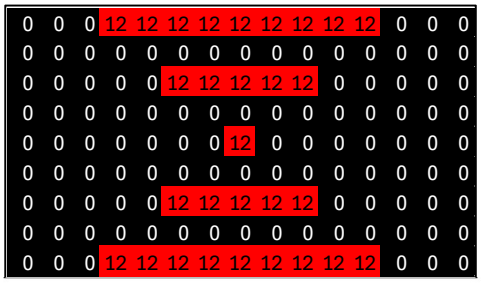

Figura 11. Representação do bloco de memória explosao2

Além das Sprites dos principais elementos do jogo, também é utilizado um bloco de memória que se refere ao desenho do terreno do jogo, este representando um elemento com 20 pixels de altura e 480 pixels de largura, totalizando 9600 bytes de valores 0 (preto), 1(azul) e 2 (verde), estes valores são substituídos durante a execução do jogo, já que a cor do terreno é alterada conforme o setor em que o jogo se encontra. A Figura 12 representa a sprite do terreno do jogo.

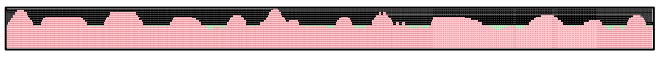

Figura 12. Representação do bloco de memória SpriteTerreno

2.2.2 Variáveis de Texto

Alguns blocos de memória são utilizados para salvar os textos que são impressos na tela, essas variáveis são: Titulo1, Titulo2, TituloSetor1, TituloSetor2, TituloSetor3, TextoGameOver1, TextoGameOver2, TextoVencedor, textoTimer.

2.2.3 Variáveis de controle

Além das Sprites a memória é utilizada para salvar o valor de diversas variáveis e flags de controle para o funcionamento do jogo, elas são:

- **DelaySetorH** (word), **DelaySetorL** (word): Usado para salvar os 16 bits mais altos e os 16 bits mais baixos para a interrupção de delay entre os setores;
- **DelayMovimentoH** (word) , **DelayMovimentoL** (word): Usados para salvar os 16 bits mais altos e os 16 bits mais baixos para a interrupção de delay entre os ciclos de movimentação do jogo;
- **DivTimer** (byte): Variável de controle para indicar quantas voltas do laço principal são necessárias para que 1 segundo se passe na execução do jogo, inicialmente setado em 20, em combinação com a variável ContTimer;
- **ContTimer** (byte): Usado para contar o número de vezes que ocorre uma volta no laço principal do jogo, a variável é incrementada a cada volta, toda vez que essa variável chegar a 20, 1 segundo se passou na execução do jogo, visto que o delay de movimentação inicialmente é setado em 50 milissegundos, dessa forma:

$$50ms\ x\ 20 = 1000ms = 1s$$

- **rangeAleatorio** (word), **numAleatorio** (word) e **deslocAleatorio** (word), **RotacaoAleatorio** (byte): Variáveis utilizadas para a geração de números randômicos;
- **LimiteCima** (word), **LimiteBaixo** (word): Utilizadas para determinar o limite superior e inferior do movimento da nave principal;
- **PosicaoAliadaInicial** (word): Contém a posição inicial na memória de
- **PosicaoNavePrincipal** (word): Utilizado nas procs de movimentação da nave principal;
- **PosicaoAliadaComparacao** (word), **PosicaoInimigaComparacao** (word): Utilizada nas procs de identificação de colisão entre naves aliadas e inimigas;
- **FlagAliadaFrente** (byte): Flag para identificar se a operação realizada é referente a nave principal ou não;
- **DeslocamentoTiro** (word): Salva o valor do deslocamento do tiro, é incrementado de acordo com a quantidade de pixels que o tiro é movido, o valor é utilizado para determinar a colisão entre o tiro e as naves inimigas;
- **PosicaoTiro** (word): Salva a posição atual do elemento tiro na memória de vídeo;
- **FlagTiro** (byte): Flag utilizada na indicação se existe um tiro ativo no jogo no momento;
- **FlagInimigo** (byte): Flag de controle utilizada na proc de identificação da nave inimiga;
- **FlagApagaAliada** (byte): Flag para controlar se é necessário ou não apagar a nave na proc que desenha as naves aliadas;
- **DeslocamentoInimigo** (word): Valor inicial do deslocamento das naves inimigas.
- **DirecaoInimigo** (byte): Utilizada para identificar se o inimigo vai se mover diagonalmente para cima ou para baixo;

- **PosicaoInimiga** (word): Utilizada para salvar o valor da posição da nave inimiga para as procs de deslocamento;
- **PosicaoInimigaColuna** (word) e PosicaoInimigaLinha (word): Utilizadas nas procs de identificação de colisão da nave inimiga;
- **CorInimiga** (byte) e **CorAliada** (byte): Salva as cores iniciais das naves inimigas e nave principal
- **contNavesInimigasAtravessaram** (word): Salva a quantidade de naves inimigas que atravessaram a tela durante a execução do setor;
- **contNavesInimigas** (word): salva a quantidade de naves inimigas existentes;
- **contDeslocamentoHorizontalInimigo** (word): Utilizada para salvar o valor do deslocamento horizontal do inimigo a fim de verificar se ele deve ou não se movimentar diagonalmente;
- vetorPosicaoInimigos, vetorPosicaoInimigaLinha, vetorDirecaoInimigos, vetorContDeslocamentoHorizontalInimigo, vetorDeslocamentoInimigos e vetorFlagInimigos (word): Vetores utilizados para salvar as informações de cada um dos inimigos simultaneamente;
- **DeslocamentoTerreno** (word): Utilizado para salvar a quantidade de pixels que foi deslocada no movimento do terreno;
- **PosicaoElemento** (word): Utilizado na proc de impressão dos elementos;
- **PosicaoVazio** (word): Utilizado na proc de apagar os elementos;
- **CorRemap** (byte): Utilizado na proc de remapear as cores de determinados sprites;
- **Jogar** (byte): Flag utilizada para controlar qual opção está selecionada no menu;
- **Jogando** (byte): Flag utilizada para controlar se o jogo está em andamento ou está no menu principal.
- **corJogar** (byte): Salva a cor inicial do texto “Jogar”
- **corSair** (byte): Salva a cor inicial do texto “Sair”
- **corTerreno** (byte): Salva a cor inicial do terreno;
- **CorAliadaAtual** (byte): Salva a cor atual da nave principal;
- **contNaves** (word): Contador de naves aliadas ainda vivas;
- **vetorPosNaves** (word), **vetorFlagNaves** (byte), **vetorCorNaves** (byte), **vetorCorOriginalNaves** (byte): Vetores utilizados no controle das naves aliadas;
- **ColidiuAliada** (byte): Flag para indicar se ocorreu ou não uma colisão com nave aliada;
- **setorAtual** (byte): Salva qual o setor atual do jogo.
- **Score**(word): Salva o valor da pontuação;
- **Timer** (byte): Salva o valor do timer;
- **TempoSetor** (byte): Variável que controla o tempo de duração dos setores;
- **contTimerInimigoLimite** (byte): Segue a mesma lógica da variável DivTimer, usada para controlar quantas naves inimigas devem aparecer por segundo, valor é alterado dependendo do setor:
    - Setor 1: 40, ou seja, 1 nave inimiga a cada 2 segundos;
    - Setor 2: 30, ou seja, 1 nave inimiga a cada 1,5 segundos
    - Setor 3: 20, para que 1 nave inimiga surja por segundo;
- **contTimerInimigo** (byte): Contador usado em combinação com a variável contTimerInimigoLimite;
- **contNavesInimigasVivas** (byte): Usado para salvar o valor de naves inimigas atualmente em jogo;
- **maxNavesInimigas** (byte): Número máximo de naves inimigas que podem estar vivas simultaneamente, valor é alterado dependendo de qual setor o jogo se encontra;

2.2.4 Variáveis para música

Para a reprodução da música e sons durante o jogo, as sequências de notas são salvas em blocos de memória, estes blocos são identificados por:

- **vetorMusicaSetor**;
- **vetorMusicaVencedor**;
- **vetorMusicaGameOver**;
- **vetorMusicaMenu**;
- **notaMusicaMenu;**

Além das sequências de notas, a memória é utilizada para salvar variáveis de controle para a reprodução da música, a seguir uma explicação de cada uma delas:

- **velocidadeMusicaMenu**: Utilizado para determinar a cada quantos ciclos do laço principal uma nota deve ser emitida, quanto menor o valor, mais rápida se torna a música, esse valor é alterado de acordo com qual setor o jogo se encontra, assim tornado a música mais “rápida” conforme a passagem dos setores;
  contadorMusicaMenu: Utilizado em combinação com velocidadeMusicaMenu para o controle da emissão das notas das músicas;

2.3 Rotinas

Para o funcionamento do jogo conforme esperado, várias rotinas foram desenvolvidas, a seguir serão citadas as principais rotinas e será descrito o funcionamento delas.

2.3.1 Rotinas de Movimentação:

- **desenha_elemento:** Principal rotina utilizada para desenhar os elementos na tela, recebe como parâmetros de entrada a variável PosicaoElemento e a Sprite a ser desenha fica salva em SI. O segmento de dados é

configurado para acessar os dados do programa, e o segmento de vídeo ES é configurado para acessar a memória de vídeo em 0A000H. Após isso a posição do elemento é passado para o ponteiro de destino DI, a quantidade de linhas da sprite é armazenada em dx (9 linhas) e o tamanho de cada linha (15 pixels) é armazenado em CX. Após isso a instrução rep movsb é utilizada para copiar 15 bytes da variável apontada por SI para a memória de vídeo, após isso DI é ajustado para a próxima linha da memória de vídeo, ou seja, somando 305, para que aponte para o início da próxima linha, DX é decrementado e se repete o desenho da linha 9 vezes.

- **apaga_elemento:** Recebe a posição do elemento a ser apagado em AX, transfere o valor de AX para a variável PosicaoElemento, aponta SI para SpriteVazio e chama a rotina desenha_elemento.
- **desenha_tiro:** Transfere o conteúdo de PosicaoTiro para PosicaoElemento e aponta o ponteiro SI para o início de SpriteTiro, então chama a proc desenha_elemento para desenha o tiro na posição desejada.
- **desenha_nave_inimiga:** Transfere o conteúdo de PosicaoInimiga para PosicaoElemento e aponta o ponteiro SI para o início de SpriteInimiga, então chama a proc desenha_elemento para desenha o tiro na posição desejada.
- **remapeia_cor:** Proc utilizada para alterar as cores da sprite da nave principal e das aliadas, recebe a posição da sprite em SI então percorre os 135 bytes da sprite testando se o bit é diferente de 0, caso seja, ele substituído pelo conteúdo da variável CorRemap.
- **desenha_nave:** Transfere o conteúdo de PosicaoAliada para PosicaoElemento, também transfere o conteúdo de CorAliada para CorRemap, define o ponteiro SI para o início de SpriteNave, então chama a proc remapeia_cor e posteriormente chama a proc desenha_elemento a fim de desenha uma nave aliada.
- **movimentaAliada:** Essa rotina é responsável por identificar as entradas do teclado vindas pelo jogador, ou seja, movimentação da nave aliada para cima e para baixo ou o clique na barra de espaço para o tiro. Primeiramente é identificado se ocorre alguma entrada por parte do jogador, utilizando a interrupção 16H, essas entradas podendo ser: Seta para cima, Seta para baixo ou barra de espaço, dependendo de qual tecla foi pressionada, determinada ação será realizada**:**
  - **Seta para cima:** chama a rotina apaga_elemento passando como parâmetro a posição atual da nave, verifica se o limite superior foi atingido pela nave principal, caso não, a posição da nave aliada é diminuída em 1280, ou seja, 4 linhas para cima, após isso chama a rotina desenha_nave para redesenhar a nave na posição atualizada.
  - **Seta para baixo:** chama a rotina apaga_elemento passando como parâmetro a posição atual da nave, verifica se o limite inferior foi atingido pela nave principal, caso não, a posição da nave aliada é aumentada em 1280, ou seja, 4 linhas para baixo, após isso chama a rotina desenha_nave para redesenhar a nave na posição atualizada.
  - **Barra de espaço:** Verifica se já existe um tiro na tela, através da flag FlagTiro, se sim, segue para o final da rotina, caso não, FlagTiro é incrementada, a posição inicial do tiro é calculada (15 pixels a mais que a posição da nave principal), chama a rotina desenha_tiro.
- **movimentaTiro:** Rotina responsável por realizar a movimentação do tiro disparado pela nave principal, primeiramente é verificado se FlagTiro possui o valor 0, caso sim, realiza um jump para o fim da proc, caso FlagTiro seja diferente de 0, significa que existe um tiro a ser movimentado, portanto a proc segue, o valor de PosicaoTiro é movido para AX, então a proc apaga_elemento é chamada, uma vez feito isso o valor da variável DeslocamentoTiro é comparada com 240, caso seja igual ou maior, o tiro chegou até a parede dessa forma FlagTiro e DeslocamentoTiro são setados em 0. Caso contrário o tiro deve ser movimentado, dessa forma PosicaoTiro é incrementada em 8 juntamente com DeslocamentoTiro e então a rotina desenha_tiro é chamada a fim de desenhar o tiro 8 pixels a frente.
- **desenhaAleatoriaInimiga:** Rotina responsável por desenhar aleatoriamente um inimigo na tela, através da rotina numeroAleatorio que será explicada mais à frente no texto, um número aleatório é gerado para o valor de coluna e da linha em que o inimigo será desenhado, através desses valores o valor de PosicaoInimiga é calculado, através da equação:

$$PosicaoInimiga = Linha * 320 + Coluna$$

- **movimentoInimigo:** Rotina responsável por movimentar uma nave inimiga, o início da rotina se dá pela checagem se a nave atingiu o limite esquerdo da tela, através da variável DeslocamentoInimigo, caso tenha atingido o limite, a nave é apagada, a Flag do inimigo é setada em 0, o número de naves inimigas vivas é decrementado e o contador de naves que atravessaram é incrementado. Após essa checagem o movimento horizontal do inimigo é realizado, decrementando em 5 a posição do inimigo e chamando a rotina desenha_nave_inimiga. A cada 5 movimentações horizontais é realizada a movimentação vertical da nave inimiga, nesse caso um número aleatório é gerado para decidir se a nave vai subir ou descer, é testado se a nave já chegou no limite superior ou inferior, incrementando ou decrementando a PosicaoInimiga em 1600 (5 linhas) e depois chamando pulando para o trecho de movimento horizontal.
- **movimentoTodosInimigos:** Rotina que gerencia o movimento de todos os inimigos no jogo, percorrendo um laço que itera sobre os vetores de informações dos inimigos: vetorFlagInimigos, vetorPosicaoInimigos, vetorDeslocamentoInimigos, vetorDirecaoInimigos, vetorContDeslocamentoHorizontalInimigo, vetorPosicaoInimigaLinha. Cada iteração verifica se o inimigo está ativo, atualiza suas informações de posição, direção e deslocamento chamando a rotina movimentoInimigo, após isso, grava os dados atualizados nos vetores.

- **desenha_elemento_terreno:** Utilizada para desenhar o elemento do terreno na tela, pixel a pixel, recebe PosicaoElemento que indica a posição inicial do desenho e percorre uma área da memória apontada por SI que contém o desenho do terreno, os pixels são transferidos 1 a 1 dessa área de memória para a memória de vídeo através da utilização de rep movsb, 320 bytes são transferidos por linhas para 20 linhas, a cada linha SI é incrementado em 160 visto que a SpriteTerreno contém 480 bytes de largura para que a impressão de movimento do terreno ocorra.
- **desenhaTerreno:** Rotina que configura a posição inicial do desenho do terreno e o deslocamento inicial dentro da SpriteTerreno, após isso chama desenha_elemento_terreno.
- **movimentaTerreno:** Responsável por atualizar a posição do terreno no jogo, cada vez que é chamada incrementa o valor de DeslocamentoTerreno em 3, após isso verifica se o valor de deslocamento chegou em 480, limite do terreno, caso sim, seta DeslocamentoTerreno em 0, fazendo com que o desenho seja desenhando desde o início da Sprite, caso contrário o terreno é desenhado com o deslocamento de 3 pixels, o que dá a impressão de movimento.

2.3.2 Rotinas de Verificação de colisão:

Para a verificação da colisão as rotinas utilizadas foram baseadas no *Separating Axis Theorem* (teorema dos eixos de separação), segundo Chong (2012), o *Separating Axis Theorem* essencialmente afirma que, se for possível traçar uma linha entre 2 polígonos, eles não colidem, caso contrário sim. Sabendo disso para testar as colisões primeiro são testados os cenários nos quais os elementos não colidem, no caso da aresta inferior do primeiro elemento estar em uma linha acima da linha da aresta superior do segundo elemento, ou a aresta superior do primeiro elemento estar abaixo da aresta inferior do segundo elemento. Tendo passado nestes testes do posicionamento vertical, o elemento certamente não colide, caso contrário, é realizado o teste do posicionamento horizontal, ou seja, é testado se a aresta mais a direita do primeiro elemento está à frente da aresta mais à esquerda do segundo elemento, e a aresta mais a direita do segundo elemento não está posicionada atrás da aresta mais a esquerda do primeiro elemento, a colisão ocorre, caso contrário não.

- **verificaColisao:** Rotina utilizada na verificação de colisão entre naves aliadas e inimigas, recebe como parâmetros a posição da nave aliada através de DX e da nave inimiga através de CX, baseado no que foi citado anteriormente o algoritmo verifica a colisão ou não entre os dois elementos baseados nas posições iniciais, caso ocorra a colisão, a flag ColidiuAliada é setada em 1 e contNavesInimigasVivas é decrementado.
- **verificaColisaoNaves:** Carrega as variáveis PosicaoAliada e PosicaoInimiga em DX e CX, depois a rotina verificaColisao é chamada, caso a colisão seja identificada, a rotina segue desenhando a explosão no local da colisão, ambas as naves são apagadas, os flags e contadores são atualizados, após é chamada a rotina que desenha a próxima nave aliada, caso não exista mais naves para ser posicionada a frente, a função de Game Over é chamada, indicando o fim do jogo.
- **verificaColisaoNavesTodas:** Essa rotina percorre um vetor de posições de cada uma das naves inimigas ativas (vetorPosicaoInimigos) e verifica colisões com naves aliadas, atualizando os valores dos vetores de posições e flags dos inimigos.
- **verificaColisaoTiro:** Essa rotina funciona da mesma forma que a rotina verificaColisaoNaves, porém o primeiro elemento é o tiro e o segundo elemento é a nave inimiga, ao final caso ocorra a colisão, a proc que desenha explosão é chamada, o tiro é apagado, e o FlagInimigo setado em 0;
- **verificaColisaoTiroTodas:** A rotina verifica se algum dos tiros disparados pelo jogador colide com qualquer nave inimiga. Se houver colisão, a rotina trata a colisão, atualiza o estado do jogo e incrementa a pontuação do jogador.

2.3.3 Rotinas de Controle de Jogo:

Essa seção trata sobre a principais rotinas de Controle do jogo, elas são:

- **numeroAleatorio:** Essa rotina é a responsável por gerar os números aleatórios utilizados em outras rotinas do programa. A geração do número aleatório é baseada na manipulação matemática de um número fornecido pela interrupção INT 1Ah, 00h - Read System-Timer Time Counter, conforme Filatov (2001) a interrupção retorna o horário atual do dia, no formato de contador de *clock*, a parte alta do contador é armazenada em CX e a parte baixa em DX. Esse número retornado é incrementado de acordo com o *clock* do sistema, ou seja, o número em si já é um número pseudoaleatório, visto que cada vez que a função é chamada ela fornece um valor diferente. Apesar disso ele é um número sequencial, o que não é desejado para o funcionamento adequando do jogo, para contornar isso, manipulações matemáticas são realizadas com o objetivo de tornar o número gerado pela rotina não-sequencial.
  Primeiramente o número retornado pela interrupção é limitada a um valor entre 1 e 7 e armazenado na variável RotacaoAleatorio, após isso a interrupção 1Ah é chamada novamente, a parte alta desse novo valor é rotacionada X vezes a esquerda de acordo com o valor de RotacaoAleatorio, o valor então é "misturado" com AX que contém a parte baixa do retorno da primeira interrupção através do comando XOR DX, AX e então o valor é subtraído por um valor arbitrário, nesse caso foi utilizado: 0FCF1h. A valor que possuímos agora é então divido pelo valor contido na variável rangeAleatorio e incrementado com o valor de deslocAleatorio que irá determinar a faixa em que o número aleatório retornado deve estar, por exemplo: para um número entre 1 e 10, rangeAleatorio deve ser 10 e deslocAleatorio deve ser 1.
  Essa Rotina foi criado tendo por base o método Gerador Linear Congruente, segundo Bolte (2007), Um gerador linear congruente (LCG) gera números

pseudorrandômicos começando com um valor chamado semente e aplicando repetidamente uma relação de recorrência para criar uma sequência.

- **delay:** Rotina utilizada para gerar um intervalo de tempo durante a execução do código, ela chama a Interrupção int 15h passando o parâmetro 86h, segundo Filatov (2001), essa interrupção faz o sistema aguardar por um número especificado de microssegundos antes de retornar o controle ao "chamador", para a utilização da função a parte alta do valor em microssegundos deve estar em CX e a parte baixa em DX.

- **ESC_UINT16:** A rotina ESC_UINT16 converte um número inteiro sem sinal de 16 bits em uma string de caracteres ASCII, armazenando o resultado na área da memória apontado por DI

- **escreveTexto:** Rotina utilizada para escrever uma String na tela, a rotina utiliza a interrupção 10h com o parâmetro 1300h, essa interrupção escreve na tela uma String armazenada na memória apontada por ES:BP, CX precisa conter o tamanho da String, DH e DL são respectivamente as coordenadas de coluna e da linha da tela na qual a String será escrita.

2.3.4 Rotinas para a reprodução da música:

As rotinas para a reprodução das músicas e sons do jogo, são rotinas que percorrem uma sequência de "notas" representadas por frequências chamando 3 sub-rotinas específicas para a reprodução em si das notas, são elas:

- **play_sound:** A rotina configura o timer do sistema para gerar uma frequência específica e ativa o PC Speaker para tocar um som. Recebe o valor referente a frequência a ser tocada em BX, configura o modo de "onda quadrada" no timer, envia a parte baixa e alta do valor de controle para o timer e então ativa o speaker.
- **stop_sound:** A rotina desativa o PC Speaker, interrompendo qualquer som que esteja sendo tocado. Ela lê a porta do speaker, realiza uma operação AND com o valor 0FCh (11111100) e al, assim zerando os bits 0 e 1 de al e então escreve al novamente na porta do speaker.

2.4 Instruções 8086:

Algumas instruções que não foram vistas em sala de aula foram utilizadas no desenvolvimento do programa, todas elas referentes aos controles de portas de entrada e saída do sistema para a reprodução da música, instruções do tipo IN e OUT, utilizadas para controlar tanto o PC Speaker quanto o timer.

OUT: segundo Jurgens, O comando transfere um byte em AL ou um word em AX para um endereço de porta de hardware especificado.

IN: Também segundo Jurgens, um byte ou word é lido de uma "porta" e colocado em AL ou AX, respectivamente.

3. Conclusões

Durante o desenvolvimento do projeto, diversas dificuldades foram encontradas, especialmente relacionadas às limitações da programação em Assembly para a arquitetura 8086. Principalmente as questões relacionadas ao controle de movimentação, colisão e geração de números aleatórios. A programação em Assembly traz a necessidade de compreender de forma aprofundada o funcionamento do hardware como por exemplo para as configurações da memória de vídeo ou então para as portas de entrada e saída a fim de conseguir reproduzir som. Apesar das dificuldades, o trabalho proporcionou um grande aprendizado, as limitações do Assembly e a melhor compreensão da interação entre software e hardware trouxeram uma perspectiva diferente quanto a programação em geral, além de demonstrar a importância das otimizações de recursos nos sistemas com limitações de capacidade. Fora isso, o trabalho proporcionou o contato com soluções de problemas já consolidados como por exemplo a detecção de colisão utilizando o método Separating Axis Theorem e a geração de números aleatórios.

Por fim, o projeto trouxe um grande aprendizado de forma prática, demonstrando a importância de diversos conceitos para o desenvolvimento de um programa robusto e funcional, mesmo com as limitações advindas da programação Assembly.

4. Bibliografia

[1] **CHONG, Kah Shiu.** Collision Detection Using the Separating Axis Theorem. 2012. Disponível em: https://code.tutsplus.com/collision-detection-using-the-separating-axis-theorem--gamedev-169. Acesso em: 07 dez. 2024.

[2] **FILATOV, Vitaly**. INT 1Ah, 00h (0): Read System-Timer Time Counter. 2001. Disponível em: http://vitaly_filatov.tripod.com/ng/asm/asm_029.1.html. Acesso em: 7 dez. 2024.

[3] **FILATOV, Vitaly**. INT 15h, 86h (134): Wait. 2001. Disponível em: http://vitaly_filatov.tripod.com/ng/asm/asm_026.13.html. Acesso em: 7 dez. 2024.

[4] **BOLTE, Joe**. Linear Congruential Generators. 2007. Wolfram Demonstrations Project. Disponível em: https://demonstrations.wolfram.com/LinearCongruentialGenerators/. Acesso em: 7 dez. 2024.

[5] **JURGENS, David**. HelpPC Quick Reference Utility. Disponível em: https://helppc.netcore2k.net/. Acesso em: 7 dez. 2024.

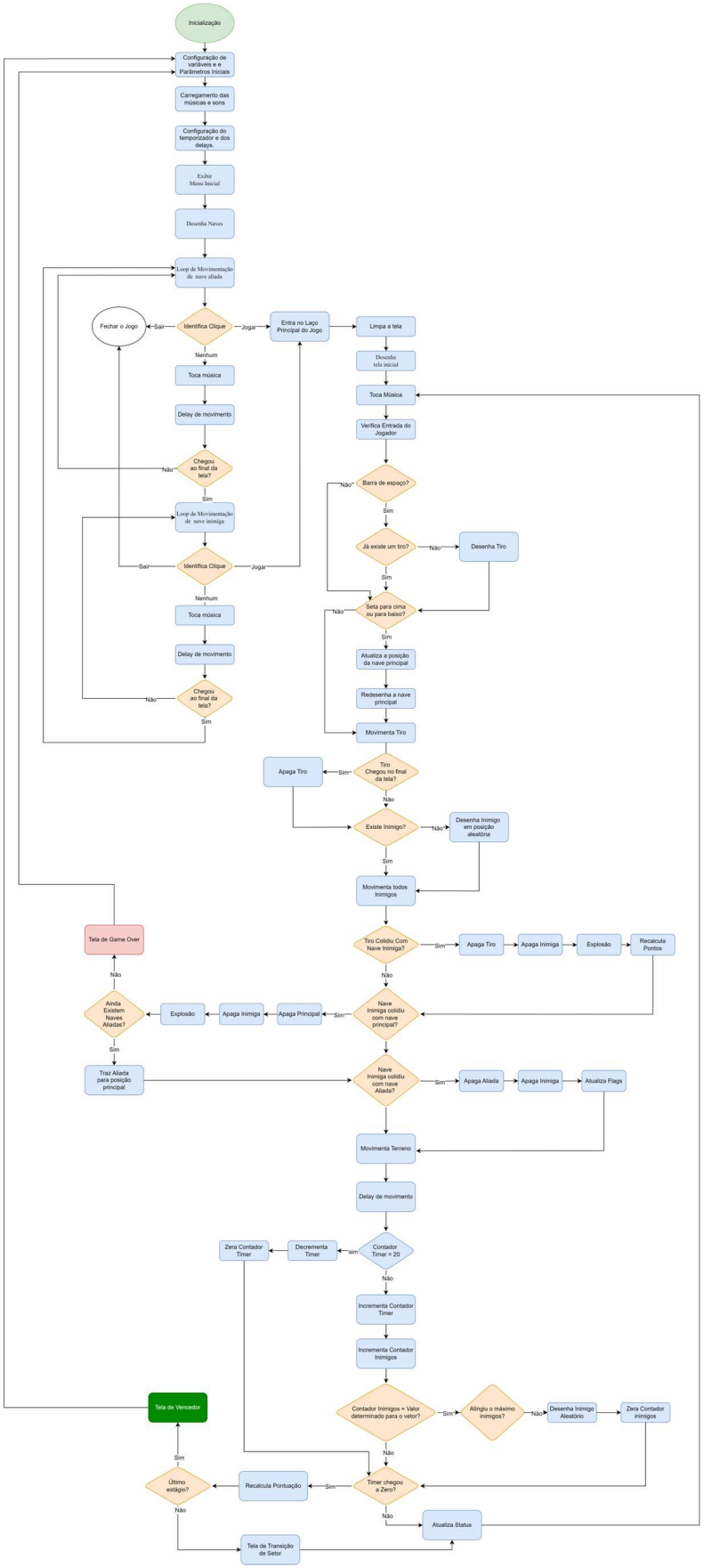

Figura 6. Fluxograma completo do jogo