Asteroid Way

Henrique Molinari Ambrosi e Bruno Falcade Paese

Resumo

Com o tempo, a programação vem se distanciando cada vez mais da linguagem de máquina para se aproximar da linguagem humana. Com isso, embora o código se torne mais fácil de implementar, por baixo dos panos, ele realiza processos que o programador nem faz ideia. O intuito desse trabalho é, usar o ASSEMBLY, uma linguagem extremamente próxima à linguagem de máquina para poder entender como a programação funciona realmente por baixo dos panos. Para isso, desenvolveu-se um jogo de nave simples. Como resultado, pôde-se observar como as principais funções das linguagens funcionam, bem como seus prós e contras.

Palavras-chave

jogo, naves, assembly, programação

1. Introdução

Este relatório contém uma breve explicação do jogo desenvolvido durante a disciplina de Arquitetura de Computadores. O Asteroid Way, é um jogo na qual o jogador controla uma nave que deve evitar asteroides que vêm em sua direção, uma vez que eles podem danificar a nave em caso de colisão. Há também pickups os quais proporcionam benefícios ao jogador. O jogo foi desenvolvido em ASSEMBLY para a arquitetura x86 utilizando o Turbo Assembler Gui [1] como montador e IDE (*Integrated Development Environment*).

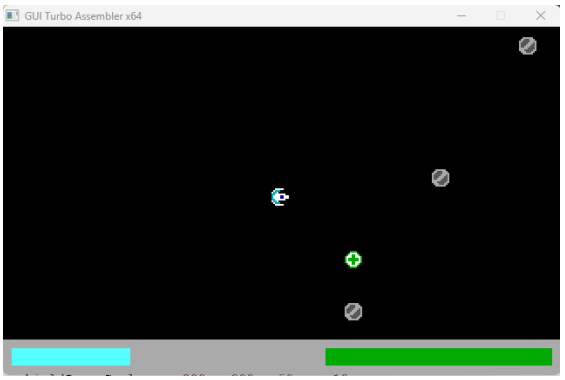
Há quatro telas no jogo, um menu principal, o qual contém as opções de jogar ou sair do jogo. A tela do Jogo em si, a qual contém o jogador, asteroides, e *pick ups* de vida e de escudo, além da UI (*User Interface*) do jogo. Há também a tela de vitória e de derrota, as quais contém textos indicando se o usuário venceu ou não o jogo.

Figura 1 - Tela inicial do jogo

Da forma em que o jogo foi implementado, a condição de vitória é sobreviver por um determinado número de fases, cuja dificuldade é incrementada por meio da redução do tempo entre o aparecimento dos asteroides e o aumento da velocidade de movimento deles. Os asteroides possuem uma chance de aparecerem a cada ciclo do loop principal do jogo.

A tela principal consiste em duas partes principais, a interface do usuário e a área do jogo. Na primeira, é mostrado o nível de saúde da nave (em verde) e o temporizador da fase (em azul). Este último indica o tempo restante para passar para a próxima fase. Na área do jogo está a nave, e no lado direito aparecem os asteroides e *pickups* como vida e escudo, os quais ao chegarem no lado esquerdo da tela, desaparecem para dar a ilusão da nave estar voando na direção contrária.

Figura 2 - Tela principal do jogo

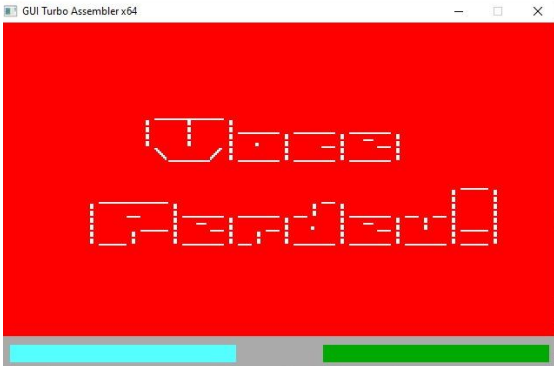


Ao final das fases é mostrada uma tela de vitória, indicando ao jogador que houve um sucesso na missão. Essa tela possui uma animação, na qual a tela é preenchida com amarelo e o texto ”Parabens!”, como o exemplo na figura 3.

Figura 3 - Tela de vitória do jogo

Em caso de morte, é mostrado uma tela similar à de sucesso, com uma animação na cor vermelha e um texto “Voce perdeu!”.

Figura 4 - Tela de morte



2. Solução

2.1. Algoritmo

O programa é dividido em três procedimentos principais, além do início do jogo. Na fase inicial, algumas configurações são feitas, como definir registradores e o modo de vídeo. Um menu inicial é criado, usando um procedimento para escrever texto na tela, e sprites (imagens) são exibidos para a nave, asteroide e itens de jogo.

Há um loop de controle no menu inicial que espera a escolha entre "Jogar" ou "Sair". Se "Jogar" for escolhido, o jogo principal começa. Neste jogo, um temporizador é atualizado para controlar a duração de uma fase. Quando a fase termina, os dados são atualizados para avançar para a próxima fase.

O programa também lê a entrada do teclado, atualizando a posição do jogador. Se a tecla de espaço for pressionada, o código para atirar é executado.

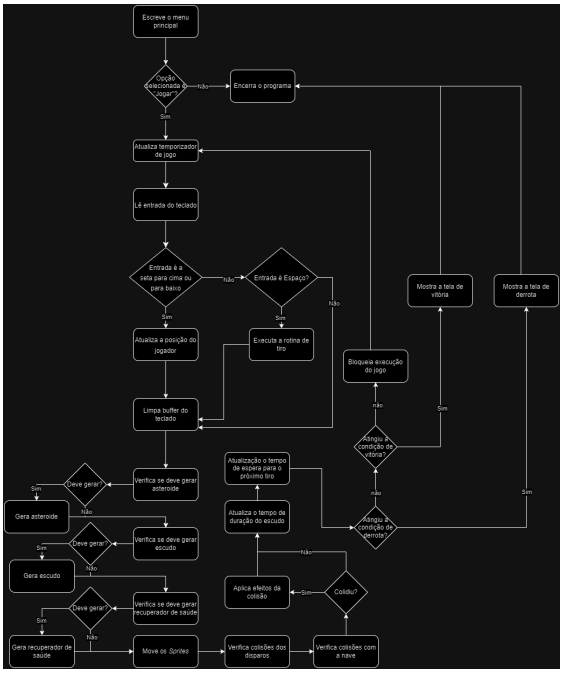
Para aparecerem os asteroides e *pickups*, é utilizado um contador incrementado a cada ciclo do laço principal. Quando esse contador chega a um ciclo coincidente com o ciclo de geração de um asteroide ou escudo, é executada a função para escrever um *sprite* no lado direito da tela. Para definir a linha em que o *sprite* aparecerá, também é utilizado um número “aleatório” gerado a partir do tempo atual e da posição do jogador. Após 200 ciclos o contador é reiniciado.

A movimentação dos sprites acontece através da *procedure* “MOVE_SPRITES”. Na abordagem utilizada, um pixel com o valor 255 é utilizado como referência para identificar o início dos asteroides e pickups. A tela é então percorrida, e quando esse pixel é detectado, uma área de 10 por 10 endereços é movida um pixel para a esquerda, resultando na movimentação desses elementos.

Há dois temporizadores adicionais, utilizados para delimitar o tempo em que o escudo está ativo e o tempo entre os tiros. Para o escudo a duração desse temporizador é de cinco segundos e para o tiro é de um segundo. Diferentemente do tempo das fases, esses temporizadores não tem uma representação visual, apenas são gerenciados em memória.

Por fim, é bloqueada a execução do programa por 50ms para que seja possível ver a situação do jogo. Abaixo, na Figura 5, segue o diagrama simplificado do fluxo principal do jogo.

Figura 5 - Diagrama do fluxo principal do jogo



Para realizar esse fluxo, o código resultante utilizou 1730 linhas de código, as quais também incluíram comentários com a explicação

do código e ocupou 39.934 bytes para o arquivo de extensão .asm e 14,779 bytes para o de extensão .exe.

2.2. Memória

Em termos de memória, o jogo pode ser dividido em dois tipos, a memória de dados e a de código. No primeiro tipo, estão localizadas as configurações e os dados necessários para o programa funcionar. Esses dados foram separados por seções demarcadas por comentários para facilitar a localização de cada variável.

A primeira seção conta com os códigos das teclas que vão servir como comandos para os jogadores. Ela utiliza somente constantes e detém *Scan codes* e códigos ASCII.

A segunda seção é responsável por guardar os textos do jogo, os quais incluem tanto artes em ASCII, geradas utilizando uma **f**erramenta de geração disponível na web [2], como sequências de caracteres.

A terceira seção inclui os *sprites* do jogo. nela é importante reparar que, com exceção do *sprite* da nave, todos os outros *sprites* foram feitos sem utilizar o código de cor zero, o qual representa o preto e indica que o espaço está vazio. Para fazer isso utilizou-se valores alternativos de preto (255 e 254). Isso é importante para detectar a área do *sprite* e facilitar as *procedures* de colisão. Além disso, todo *sprite* que se move para a esquerda foi marcado com o primeiro pixel sendo 255, uma vez que esses *sprites* não são salvos em nenhum outro lugar além da própria tela do jogador. Esse código de pixel é utilizado para identificar os *sprites* na hora de movê-los para a esquerda.

A quarta seção indica configurações de UI (*User Interface*) e locais de jogo contendo principalmente o pixel de início das barras inferiores do jogo, a cor delas, localização do jogador e um vetor de localizações de cada tiro disparado.

A quinta e última seção guarda configurações de jogo. É ela que é alterada quando um jogador passa de nível para configurar o próximo, além de determinar algumas regras do jogo. Em seu corpo, as principais variáveis são o tempo das fases, tempo de cada ciclo do jogo, tempo de geração de asteroides e escudos, nível, velocidade dos elementos que se movem para a esquerda (escudo, asteroide e recuperador de vida), quantidade de recuperadores de vida na fase, vida do jogador, duração do escudo e tempo de disparo.

Em termos gerais, o programa ocupa uma memória de dados estimada em 2,45 KB e uma memória de código estimada em 4,38 KB. Considerando também a pilha com seu tamanho fixo de 512 bytes configurado no código, tem-se uma estimativa de tamanho total em memória de 7,34 KB. É importante destacar que os números acima são estimativas, o primeiro calculado a partir das variáveis do próprio arquivo de forma braçal e o segundo, contado a partir da geração de um arquivo hexadecimal com o auxílio do bloco de notas e a ferramenta de contagem de palavras.

2.3. Rotinas

Já em questão de código, o jogo é composto de diversas *procedures*, cada uma com uma responsabilidade como a leitura do teclado, a colisão e a movimentação dos elementos.

A primeira rotina a se destacar é a “READ KEYBOARD INPUT", a qual, como o nome sugere, é a responsável por ler o *buffer* do teclado por meio de uma interrupção. Ela também é a responsável por traduzir a tecla pressionada em um comando no jogo e realizar esse comando. A ação dos comandos disponíveis preveem a movimentação do jogador para cima e para baixo alterando sua posição na memória e reescrevendo a nave no local adequado e a geração de um disparo que salva sua posição em um vetor na memória e desenha o disparo na tela. Por fim, essa função também limpa o *buffer* para evitar que o jogador fique preso em um comando, uma vez que a leitura do *buffer* é muito mais rápida que um ciclo do jogo.

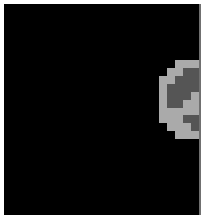
A segunda função a ser citada é a “GAME_TIMER”, cuja função é controlar o temporizador da fase. Para isso, ela decrementa a barra a largura da barra de tempo da UI em uma unidade de decremento por ciclo. Essa unidade é definida na memória com o rótulo de “timeBarScaleDecrement“. Além disso, ao zerar o temporizador, a *procedure* é responsável por chamar a próxima fase.

Já para a movimentação dos outros elementos da tela que não sejam controlados pelo jogador, é utilizada a *procedure* “MOVE_SPRITES”, a qual conta com dois *loops* principais. O primeiro é utilizado para a movimentação dos disparos, em que o programa itera sobre o vetor de disparos movendo todos para a direita com o dobro do número de pixels de movimento de um asteroide (o que causa a sensação de o dobro da velocidade), além disso,

antes de mover o disparo, ele faz o cálculo para identificar se o disparo colidirá com algum outro elemento, assim removendo o disparo e, caso a colisão seja com um asteroide, removendo o asteroide (Esse esquema de colisão é realizado pela *procedure* "CHECK_BULLETS_COLISION"). O segundo loop para os outros elementos, os quais vão todos se mover para a esquerda na mesma velocidade definida no rótulo de memória "asteroidSpeed". Para o segundo loop é executada uma varredura completa na tela em busca de pixels com o valor 255 (os quais demarcam o início do *sprite*), com isso o programa vai em busca do primeiro pixel da quarta linha para descobrir qual é o *sprite* em questão, utilizando a *procedure* "GET_SPRITE", e por fim, remove o *sprite* redesenhando ele no local adequado. Além disso, em ambos os *loop*, a *procedure* realiza cálculos para determinar se o *sprite* deve ser movido ou simplesmente removido (caso chegue no limite da tela).

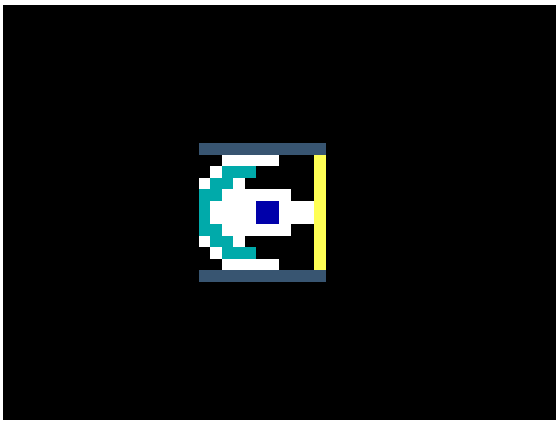
Para a geração dos asteroides e *pickups* no limite direito da tela é utilizada a *procedure* "SPAWN_SPRITE_END_SCREEN". Ela funciona de forma a receber o endereço de um *sprite*, gerar um número aleatório entre 0 e 170 e desenhar o *sprite* na tela. Para geração de números aleatórios utilizou-se a parte baixa do relógio do sistema mais a posição do jogador. A geração ocorre na *procedure* "GENERATE_RANDOM_NUMBER", a qual também recebe como parâmetro um valor máximo que pode ser retornado e o *sprite* que será desenhado usando tal número gerado, o que garante que *sprites* não sejam gerados na mesma posição. Já para a remoção e escrita do *sprite* são usadas as *procedures* "REMOVE_SPRITE" e "PRINT_SPRITE" respectivamente. É interessante destacar que a última faz um cálculo para desenhar o *sprite* mesmo se ele estiver parcialmente fora da tela. O que dá o efeito de vir da esquerda quando é gerado. A Figura 6 exemplifica isso.

Figura 6 - *Sprite* gerado parcialmente na extremidade esquerda da tela



Já para detectar a colisão do jogador com outros objetos que se movem para a esquerda, a *procedure* utilizada é a "HANDLE_PLAYER_COLLISION", a qual passa percorrendo os limites da nave de forma a verificar primeiramente a parte superior, depois a inferior e por fim a frontal. Essa ordem é importante, pois o tratamento das três colisões passa por *procedures* diferentes para detectar o ponto de início do objeto, o primeiro pixel. Na colisão superior, após ser detectado um pixel diferente de zero, o programa navega pelo *sprite* até encontrar seu limite na parte superior, depois navega para a esquerda até encontrar o primeiro pixel demarcado com o código de cor 255. Na colisão central, basta o programa identificar a colisão e navegar para a esquerda até encontrar o 255. Por fim, na colisão inferior, o programa navega até o limite esquerdo do *sprite*, navega até o limite inferior e por fim decrementa dez linhas (navega 10 linhas para cima) chegando à origem do *sprite* (isso somente é possível pelo fato de todos os objetos terem o mesmo tamanho). A Figura 7 exemplifica a área de colisão da nave.

Figura 7 - Visualização do detector de colisões da nave (em cinza e amarelo)



Após obter o pixel de origem, as três *procedures* convergem de volta na mesma que as chamou para determinar qual objeto colidiu com a nave. Para isso, é chamada a *procedure* que navega até o primeiro pixel da quarta linha (a *procedure*, utilizada é a "GET_SPRITE"). Esse pixel foi escolhido estrategicamente por conter uma cor diferente para cada *sprite*. Após isso, já foram determinados o ponto inicial do *sprite* e qual o objeto que o *sprite* representa, então basta apagar o *sprite* e aplicar o efeito de sua colisão, processo feito nessa mesma *procedure*.

Embora essas sejam as *procedures* principais do código, existem diversas outras que foram utilizadas de forma a facilitar processos que

ocorreriam frequentemente. Além disso, também foram utilizadas algumas para facilitar o encontro de *bugs* e erros no comportamento do código. Um exemplo é a "PRINT_PIXEL", a qual simplesmente desenhava um pixel na tela na posição do registrador "DI" e chamava a interrupção para travar a execução do programa por determinado tempo.

3. Conclusões

Por fim, obteve-se sucesso na criação do jogo. Embora ele não seja divertido de jogar de fato, ele implementa todas as funcionalidades propostas. Assim, como a ideia do projeto era ter uma experiência com uma linguagem mais próxima à da máquina e entender os processos realizados pelo computador para alcançar os comandos de alto nível utilizados nas linguagens atuais, é possível afirmar que o projeto rendeu experiências bastante positivas e alguns aprendizados.

No processo de criar o jogo, a mentalidade inicial era utilizar a menor quantidade de memória de dados possível. Com o passar do tempo, essa mentalidade mudou para uma utilização mais frequente da memória, uma vez que em alguns casos ela poderia ter sido mais utilizada em prol de reduzir a quantidade de comandos assim reduzindo tanto a quantidade de instruções executadas pela CPU, como a quantidade de memória utilizada, uma vez que o código do programa também é armazenado em memória.

Além disso, outra concepção que mudou do começo do projeto para o final foi a dificuldade de trabalhar com uma linguagem de baixo nível. No começo, a ideia parecia extremamente complexa. Porém, após um certo tempo de projeto, percebeu-se que a lógica do algoritmo era a mesma, além de ser possível entender exatamente o que o programa fazia por não ter tantos processos por baixo dos panos. Fato é que o programador tem muito mais liberdade para fazer o que precisa.

Quanto às principais dificuldades enfrentadas, têm-se como efeito de causa dos principais aprendizados. O primeiro problema foi encontrado logo no começo, já que tinha-se conhecimento do que precisava ser feito, mas pouco de como fazê-lo. Nessa etapa, a pesquisa na documentação [4] foi muito importante e o andamento do projeto dava passos lentos. Porém com a prática, o desenvolvimento começou a acelerar e a curva de produtividade teve um aumento expressivo.

O segundo problema principal foi o fato de não termos armazenado os asteroides e *pickups* em memória além da tela. Isso começou a ter impacto no momento de codificar as colisões, uma vez que existia um conflito entre a experiência visual do usuário e o funcionamento correto das mecânicas do jogo. E embora isso tenha sido vencido com algumas ideias como alterar os *sprites* de forma a guardar informações neles, aumentou significativamente a quantidade de código e a complexidade de algumas operações. Frente a isso, a mecânica do tiro foi feita de forma a salvar sua posição na memória e não enfrentar os mesmos problemas.

4. Referências

[1] Gui Turbo Assembler. Guitasm8086. Disponível em: https://sourceforge.net/projects/guitasm8086/. Acesso em: 03 de dez. de 2023.

[2] Patorjk. Ferramenta de Estilos de Texto. Disponível em: http://patorjk.com/software/taag. Acesso em: 03 de dez. de 2023.

[3] Portugal-a-Programar. Número Aleatório em Assembly. Disponível em: https://www.portugal-a-programar.pt/forums/topic/69426-numero-aleatorio-em-assembly/. Acesso em: 03 de dez. de 2023.

[4] Agner Fog. x86 Instruction Set Reference. Disponível em: https://c9x.me/x86/. Acesso em: 03 de dez. de 2023.