Breakout: Desenvolvimento em assembly 8086

Gregor Yannis Wey* e Leonardo Martelli Oliveira*

Resumo

Breakout é um jogo desenvolvido pela Atari em 1976, sucedendo o popular Pong, propondo um jogo para apenas um jogador. A mecânica do jogo consiste em manter a bola sendo rebatida por uma raquete, com o objetivo de quebrar todos os blocos. No seguinte trabalho será apresentado o desenvolvimento deste jogo em *assembly* para máquinas com processador 8086.

Palavras-chave
Breakout, assembly, 8086

I. Jogo

O jogo pode ser dividido em três partes, a tela inicial, tela de jogo e tela final. No primeiro momento, a tela inicial é mostrada como a figura 1, através das setas do teclado cima e baixo, o jogador pode escolher Jogar ou Sair.

Fig. 1: Tela inicial.

O jogo é simples: o jogador tem o controle de uma raquete, movimentando-a com as setas direita e esquerda, assim possibilitando rebater e direcionar a bola para tentar destruir as quatro camadas de blocos, como pode ser visto na figura 2. O bloco é destruído pelo simples toque da bola. Ao destruir todos os blocos, as camadas dos mesmos e a velocidade são restadas, mantendo pontuação e número de vidas atuais.

Caso a bola não é rebatida, irá atingir o limite inferior da tela, assim diminuindo a vida em uma unidade. Ao zerar as vidas, o jogador perde e o jogo acaba. A pontuação e vidas restantes são mostradas na parte superior da tela: pontuação alinhada à esquerda e a quantidade de vidas à direita.

Ao acabar o jogo, a tela final é mostrada, como na figura 3. Nela é contida uma mensagem de fim de jogo e a pontuação atingida.

II. Solução

A. Algoritmo

Ao todo, o projeto teve 1321 linhas, muitas delas vazias com propósito de organização e fácil leitura e interpretação do código. O tamanho do arquivo totalizou 17.6 KB (18,099 bytes) , comparado a jogos atuais é algo mínimo. O fluxograma do jogo pode ser observado na imagem 4.

* Universidade de Caxias do Sul, Caxias do Sul, RS, gywey@ucs.br; lmoliveira18@ucs.br.

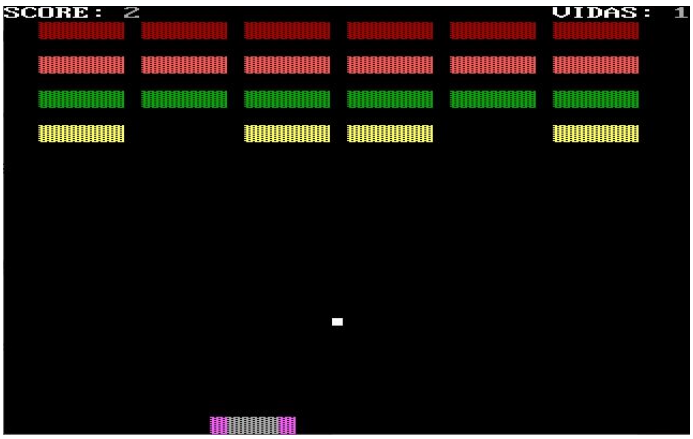

Fig. 2: Tela de jogo.

Fig. 3: Tela de game over.

O menu é apresentado com as opções que podem ser escolhidas. Caso o jogador escolha sair, o programa é encerrado. Ao escolher a opção de Jogar, o jogo é iniciado.

Ao fim do jogo, a terceira tela é mostrada. As três principais partes do projeto serão explicadas separadamente: tela inicial, jogo e a tela de fim de jogo.

O fluxo da tela inicial é demonstrado na figura 5. Após o modo de vídeo padrão de 25 linhas por 40 colunas ser instanciado, é escrito o menu com opções, vide a figura 1.

O *input* do teclado é testado, as setas para baixo e para cima podem ser pressionadas para posicionar o cursor onde for desejado. No final é realizado uma comparação se a tecla

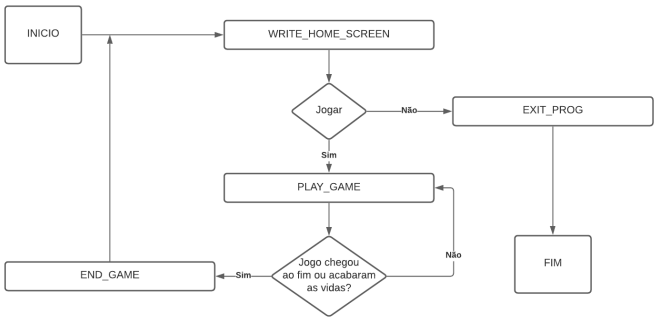

Fig. 4: Fluxograma do jogo.

da seta para direita foi pressionada, caso não o algoritmo volta para o teste de *input* inicial. Se a tecla foi pressionada, é realizado um novo teste, para verificar em qual posição o cursor se encontra. Caso esteja sobre a opção Jogar, o procedimento PLAY-GAME é chamado. Caso contrário, é chamado EXIT-GAME, assim saindo do jogo.

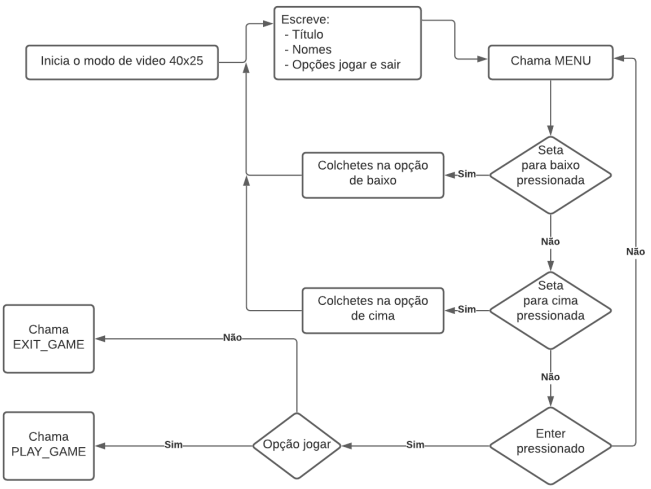

Fig. 5: Fluxograma da tela inicial.

O laço principal do jogo, figura 6, inicializa o modo vídeo padrão, com o objetivo de limpar a tela. Os itens são inicializados: raquete, bola, pontuação, vidas. As quatro fileiras de blocos e o cabeçalho são mostrados na tela, como na figura 2. É lido então se há *input* nas teclas direcionais direita e esquerda, a fim de mover a raquete.

O calculo da velocidade da bola é realizado baseado na quantidade de blocos quebrados. Após é comparado se a quantidade de passos (iterações do laço principal) é menor que a velocidade, caso sim, o laço é continuado. É chamada a função AWAIT, acionando a interrupção de espera (15h) durante 750 milissegundos. A seguir é incrementado os passos.

Caso a quantidade de passos é maior que a velocidade, a próxima posição da bola é calculada, assim acontecendo uma movimentação da bola. É verificado os limites e colisões da bola neste cálculo. Se a bola atinge o limite inferior, é subtraída uma vida e a bola é ”posta” em cima da raquete novamente.

É realizado um teste para determinar se o jogo deve ser terminado: caso a quantidade vidas atingir zero, o mesmo é terminado por intermédio da rotina END-GAME, figura 7. Após o jogo terminar a tela final é apresentada através da

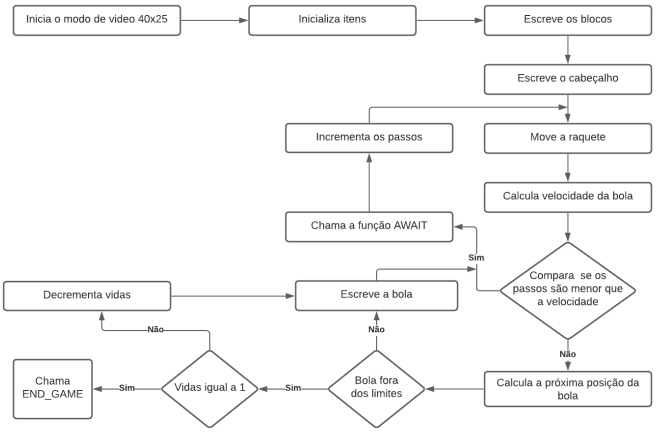

Fig. 6: Fluxograma do laço principal.

rotina WRITE-GAMEOVER-SCREEN, nela a *string* de fim de jogo e *score* são escritas junto com a pontuação. O programa aguarda uma tecla ser pressionada, para então voltar à rotina da tela principal.

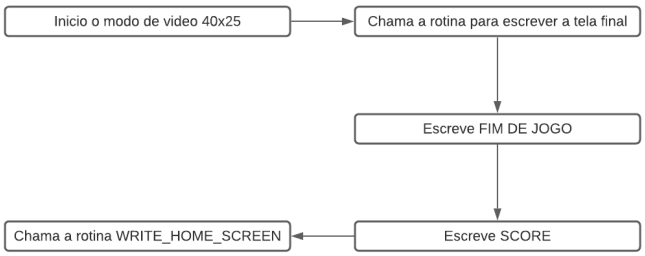

Fig. 7: Fluxograma do fim de jogo.

B. Memória

O modelo de memória escolhido foi o *small*. O espaço reservado para o segmento de pilhas é de 16KB. No segmento de dados foram definidas algumas constantes, como as cores e seus códigos. Algumas dessas cores - Vermelho, Vermelho claro, Verde e Amarelo - formam um vetor, organizado sequencialmente, utilizado para definir as cores dos blocos do jogo.

Foram definidas algumas *strings* como por exemplo o título *Breakout*, Fim de Jogo, os nomes dos autores e outros termos que são mostrados no menu, cabeçalho e fim de jogo.

Entre os dados declarados, estão as velocidades da bola em hexadecimal, em ordem decrescente. Algumas variáveis como a quantidade de vida, pontuação, a direção da bola, blocos destruídos, coordenadas dos itens são declarados também.

Ao total os dados utilizam 513 bytes, sendo a maior parte as *strings* definidas.

C. Rotinas

- Laço principal

A principal rotina do programa é a chamada PLAY-GAME. Nela acontece o laço principal do jogo.

Antes do inicio da iteração, é chamada a *proc* de inicialização do modo de video, a fim de limpar a tela. Após são inicializados os itens: blocos, valores do cabeçalho, posição da bola e raquete (aleatórias).

Dentro do laço principal, valida-se caso quebrou todos blocos - se sim, os blocos são reescritos; faz a movimentação da raquete de acordo com o *input*; calcula-se a velocidade da bola (iterações realizadas neste laço, até que seja rescrita e recalculada a posição nova da bola); verifica-se o contador - armazenado em CX - atingiu a velocidade calculada, se sim as operações de calculo e escrita da bola são realizadas. Após chama-se a *proc* de espera (meio milissegundo) e incrementado o contador do laço.

O laço é terminado caso se perca o jogo, ao perder as três vidas.

- Calculo de deslocamento na memória de video

A rotina CALCULATE-VIDEO-POS, recebe em DX as coordenadas de X e Y na tela, respectivamente, e retorna no mesmo registrador o deslocamento dentro do espaço de memória reservado para o modo de video.

- Calculo de velocidade

As velocidades são guardadas em um vetor BALL-SPEEDS, para definir a velocidade é utilizado o valor armazenado no BLOCKS-CLEARED como parâmetro ou a variável de *flag* que aramazena a informação se algum bloco vermelho já foi destruído.

O algoritmo faz uma comparação inicial com a variável *flag* IS-KICKED-RED, caso seja verdadeiro (1), AX recebe 24, que é o número máximo de blocos que podem ser destruídos. Logo valor de velocidade máxima é setado.

Caso contrário acontece a recuperação do dado de quantos blocos já foram destruídos. Assim realizada a divisão inteira de blocos destruídos por quatro, uma vez que a cada quatro blocos a velocidade deve aumentar (no contexto do projeto, o número de iterações para movimentação da bola diminui). O resultado desta divisão inteira, é multiplicado por dois, visto que BALL-SPEEDS tem os itens do vetor de tamanho *word*. O produto então é o deslocamento necessário em BALL-SPEEDS para chegar a velocidade desejada. A proc retorna em AX o valor da velocidade.

- Impressão na tela

A impressão tem várias rotinas que atendem diferentes necessidades. Para números no formato BCD decimal existe WRITE-NUMBER, a qual utiliza a rotina WRITE-CHAR. Esta que através da interrupção 21h e com AH = 02h, permite escrever carácteres.

WRITE-STRING utiliza a interrupção 10h com AH = 13h, esta que exige como parâmetros o tamanho da *string* em CX; o inicio da *string* em BP, a coluna e linha em DL e DH, respectivamente; e o atributo da cor em BL.

A escrita da bola e raquete, utiliza a rotina WRITE-WITHOUT-INT-WITH-INFO. Recebe os parâmetros das coordenadas em DL (equivalente ao X cartesiano) e DH (equivalente ao Y cartesiano) que será escrito; tamanho da *string* em CX e em AX o início da mesma, ou seja, o *offset* do vetor que define a *string*. Nesse que contém os dados *char* e cor, respectivamente, para cada carácter que deseja ser escrito. A rotina WRITE-WITHOUT-INT-WITH-INFO escreve sem utilizar interrupções, com um REP MOVSB é possível copiar de um endereço de memória diretamente para outro endereço de memória.

Os blocos são escritos pela rotina WRITE-BLOCK. SI recebe o *offset* do vetor das cores e AL o código do caractere que constitui os blocos. Ela que utiliza dois *loops*. O *loop* externo é para o controle das quatro linhas, setando a cor que será escrita e a posição na memória de video em que será armazenado os blocos. No *loop* interno, utiliza-se REP STOSW para escrever os blocos, uma vez que AX é constituído por AH com a cor e AL com o carácter a ser escrito, estes dados vão direto na memória. Ao sair deste *loop* interno SI é incrementado para a próxima cor no vetor e DI recebe o deslocamento a ser realizado para escrever na próxima linha.

- Detecção de colisão

Sempre é testado se a próxima coordenada da bola irá colidir com um bloco, limite ou raquete. Para isso a rotina GET-KICKED é utilizada. Nela, a próxima posição é comparada com o *char* correspondente aos blocos, que constitui também a raquete. Caso aquela posição da memória de video não armazena o carácter do bloco, o programa continua. Caso seja o carácter, é testado o byte referente a cor, em AH, com magenta da raquete, se negativo é movido 1 para AL sinalizando que um bloco foi encontrado e a direção da bola não será diferente da normal (isto é, não irá voltar pelo trajeto que estava vindo). AH é comparado com cada uma das cores dos blocos, cada uma das cores tem a sua função interna, atualizando o *score* de acordo com a pontuação pré estabelecida. A *proc* CLEAR-BLOCK então é chamada.

- Apagar blocos

Mencionada anteriormente, a rotina CLEAR-BLOCK é encarregada de apagar o bloco colidido. Por meio das instruções de divisão, multiplicação e soma, é calculado qual dos blocos deve ser apagado. CALCULATE-VIDEO-POS calcula o deslocamento dentro da memória de vídeo. Com AX setado em zero, por meio de REP STOSW, é realizado o processo de apagar caractere por caractere diretamente na memória. Outra *proc* para apagar algum item da tela, é a rotina CLEAR. Nela é passado por parâmetro em DX a posição da tela a ser apagada. Utilizasse a mesma instrução anterior para apagar, via STOSW.

- Movimentação da bola

A movimentação é feita através da rotina principal CALCULATE-NEXT-BALL-POSITION, que utiliza BX para verificar as coordenadas futuras da bola. A direção da bola, armazenada na memória, é movida em CL. A bola então é apagada da tela.

A direção é definida sendo um dos quatro números: 1 - subindo para a direita, 2 - descendo para a direita, 3 - descendo para a esquerda e 4 = subindo para a esquerda. A direção que está em CL é comparada sequencialmente com cada um desses números, chamando uma das quatro rotinas de direção: GOING-RIGHT-DOWN, GOING-RIGHT-UP, GOING-LEFT-DOWN e GOING-LEFT-DOWN.

As rotinas estas que contêm maneiras similares de funcionamento. Primeiramente é comparado se a próxima posição vertical da coordenada é igual ao limite superior ou inferior da tela, caso seja no limite inferior, é decrementada a vida em uma unidade e reiniciada a posição da bola em cima da raquete.

Caso bata no limite superior, a bola irá se direcionar para baixo, seguindo sua direção horizontal. O mesmo acontece ao colidir com algum bloco.

Caso não haja colisão, será verificada a próxima posição diagonal, seguindo mesmo comportamento descrito anteriormente. Nesse caso, é verificado os limites laterais, caso colida, sua direção vertical é mantida, invertendo a horizontal.

Após é verificado sua próxima posição horizontal. Caso essa colida com algum limite ou bloco, igualmente sua direção vertical é mantida e inverte-se a horizontal.

O comportamento mais distinto é ao ser retornado em AL o valor dois, ou seja, colidiu com a ponta da raquete. Com esse comportamento, suas duas direções - horizontal e vertical - são invertidas.

- Movimentação da raquete

A raquete utiliza a rotina MOVE-RACKET. A interrupção 16h com AH = 1 permite obter o estado do *buffer* do teclado, caso não haja nada, o jogo continua. Porém se houver algo no *buffer*, AH = 0 permite a leitura da tecla pressionada.

A coordenada inicial é passada para BL e a final para BH, AH é comparado com o código correspondente da tecla esquerda e da tecla direita, cada uma das comparações leva a sua função correspondente.

Na função da tecla direita BL é comparado com o limite direito da raquete , evitando da raquete ultrapassar o limite lateral direito.

Na função da tecla esquerda é comparado se ao decrementar BL a *flag* negativa é acionada , assim evitando a raquete de ultrapassar o limite lateral esquerdo.

Caso os limites não são atingidos em ambas, a raquete tem sua coordenada atualizada e o carácter remanescente da movimentação é apagado.

- Números Aleatórios

O mecanismo de é inspirada em um exemplo visto em um fórum [1], a interrupção 1Ah, com AH = 00h, permite ler o tempo real do sistema, AL recebendo as horas, CX a parte alta do relógio e DX a parte baixa. Implementamos o número máximo do valor a ser gerado (o valor mínimo é implícito a zero) por BX, dos registradores usados pela interrupção, acabamos usando apenas o DX. BX é dividido pela parte baixa do relógio, gerando o número aleatório.

III. Conclusões

De forma geral foi um trabalho muito interessante de ser feito. Mesmo que inúmeras adversidades foram encontradas durante o desenvolvimento, conseguimos finalizá-lo com sucesso e superamos as expectativas iniciais do grupo.

Devido ao tamanho do projeto, no inicio tínhamos a impressão de que seria um trabalho muito difícil, no além da pouca familiaridade com *assembly*, no entanto acabou sendo um aprendizado gigantesco. Muitas vezes o código fazia sentido total no teste de mesa mas não funcionava ao rodar, foram horas e mais horas apenas para corrigir erros causados pela falta de experiência, atenção e prática. Como por exemplo o cálculo de velocidade de acordo com x blocos destruídos, foram cerca de três dias apenas para solucionar a dificuldade.

É possível dizer que o projeto foi um sucesso, e sem dúvidas ensinou muito sobre *assembly* e ótimas práticas de programação. Conhecer como um computador funciona por trás do alto nível é uma experiência sobrestimada.

Somos gratos pela oportunidade apresentada e por todo conhecimento adquirido durante todo o processo do trabalho. Iremos levar para o resto de nossa carreira tudo que aprendemos.

IV. Referências

[1] Thread: Assembly random numbers. codeguru.com , 2007. Disponível em https://forums.codeguru.com/showthread.php?430236-Assembly-random-numbers. Acessado em 13/12/2020.