TRON

Foi-nos pedido para implementar uma versão do jogo *TRON* em linguagem *Assembly* do *P3* e sugerida uma metodologia de trabalho para a realização deste projeto, a qual seguimos. Começámos por refletir sobre a lógica que servia de base ao jogo para a realização dos fluxogramas nos quais nos baseámos para a codificação do programa. Vamos agora abordar os tópicos descritos pelo enunciado do projeto e clarificar a forma como os implementámos.

- ## Início do Jogo:

  Para escrever duas mensagens, centradas, na janela de texto do *P3* procedemos à codificação de uma rotina que recebe como 'argumentos' a posição onde se ia centrar a mensagem e um apontador para o primeiro caracter. Usámos o porto de controlo da janela de texto para escrever nas posições necessárias (toda a escrita na janela de texto no projeto foi controlada pelo porto de controlo da janela de texto, visto que assim que é inicializado a sua utilização é obrigatória).

  Após uma planificação do projeto decidimos que era vantajoso inicializar o *LCD* antes do jogo começar. Foram escritas duas rotinas, uma que escreve no *LCD* as mensagens que são fixas, pois aqueles caracteres nunca são alterados ao longo do programa, e outra que trata de escrever/atualizar os valores presentes em memória correspondentes aos pontos de ambos os jogadores e tempo máximo de jogo.

- ## Espaço de Jogo:

  Para efetuar a escrita de caracteres na janela de texto do P3 é necessário mover para o porto de controlo da janela de texto um valor cujo octeto mais significativo representa a linha da janela de texto e o octeto menos significativo representa a coluna. Partindo desta representação e da lógica que seria necessária implementar para detetar colisões decidimos que a melhor maneira de proceder à codificação desta lógica seria criar como que uma representação do campo de jogo, transposta para memória.

  Com isto decidimos que era necessário reservar algum espaço em memória. Para nos facilitar a escrita neste espaço em memória era preciso converter uma coordenada do tipo ecrã para uma coordenada que apontasse para a posição na matriz em memória que representava esta coordenada do ecrã. Procedemos à codificação de uma rotina que realiza este trabalho, a rotina *converteEcraMatriz*.

- ## Representação da partícula:

Para representar a partícula no ecrã procedemos à escrita do caracter adotado para cada partícula na posição respetiva, seja essa a posição em que começam ou alguma das posições pelas quais a partícula passa durante o seu movimento (explicado no ponto seguinte). Para escolher a posição onde escrevemos a partícula usamos, novamente, o porto de controlo da janela de texto, já abordado num dos tópicos anteriores.

- ## Movimento da partícula:

Visto que o movimento da partícula apenas pode ser alterado de duas formas para cada uma das partículas, ou muda de direção à esquerda ou à direita, e esta alteração é feita por uma rotina de serviço a uma interrupção (respetiva ao botão que muda a direção de cada partícula) achámos que a melhor maneira de abordar o movimento da partícula seria manter um registo para cada partícula que guardasse a direção para onde vai mover-se de seguida e essas rotinas alteram o valor desses registos. Temos que esta direção toma o valor 0 se a partícula se for mover para a direita, 1 se se for mover para cima, 2 para a esquerda e 3 para baixo. Daqui saí que as rotinas que alteram a direção da partícula funcionam incrementando ou decrementado o valor destes registos (com especial atenção ao valor 3, que volta a 0 caso a partícula mude de direção à esquerda e ao valor 0, que muda para 3 caso a partícula mude de direção à direita). De seguida aplicamos esta direção à posição atual da partícula com a rotina *aplicaVetor* e caso não hajam colisões da parte desta partícula procedemos a desenhá-la no ecrã e na posição respetiva que temos reservada em memória. Para a escrever no ecrã usamos a coordenada que sai da rotina *aplicaVetor*, já na memória é necessário converter esta coordenada, da forma abordada em cima, para um endereço de memória, no qual vamos registar que a partícula lá esteve para efeitos de detetar colisões. Caso hajam colisões, detetadas através das posições em memória procedemos às rotinas de fim de jogo, que detetam quem perdeu, escrevem no ecrã e dão pontos a quem ganhou (em caso de empate damos pontos a ambos os jogadores).

- ## Níveis de jogo:

Todos os níveis requeridos foram implementados, com recurso ao temporizador do *P3* recorrendo a *delays* consecutivos de uma décima de segundo até completar os tempos necessário para cada dificuldade. Para determinar quanto *delay* precisávamos para cada uma das dificuldades usamos o valor que ia ser escrito no display de 7 segmentos, que representa os segundos que passaram desde o início do jogo. O *delay* é chamado pela rotina *delayDificuldade*, que avalia automaticamente a dificuldade em que está o jogo e qual será o *delay* a chamar.

- ## Fim do jogo:

A parte mais difícil do fim do jogo é detetar uma colisão que o termina, a escrita das mensagens centradas é feita como no início do jogo, o mesmo acontece com a interrupção que recomeça o jogo, isto já está implementado no princípio do jogo. Quanto às colisões usamos a matriz que foi reservada em memória, na qual foi registada se um jogador, ou seja uma partícula, lá passou, daí detetamos as colisões e caso as hajam acaba o jogo. Seguimos depois para uma série de condições para detetar quem ganhou, ou empate caso aconteça, e dar os pontos devidos aos jogadores.

- ## Outros aspetos importantes:

Durante a implementação das funcionalidades que serviam de base ao jogo reparámos em alguns problemas nos métodos que usámos para as implementar, passamos agora a expor estes problemas e o método como foram implementados para resolver os problemas.

Na primeira versão da nossa implementação do temporizador, com o *timer* do *P3* tentámos usar como intervalo para as interrupções do *timer* o valor que era necessário à dificuldade em que o jogo se encontrava e usámos uma lógica rebuscada para escrever os segundos no *Display de 7 segmentos*, eventualmente corrigimos isto para evitar termos o valor dos segundos escritos em tempo real.

A versão inicial do nosso programa, que permitia jogar uma partida do jogo, ao carregar no I1 durante o jogo deixava esta interrupção pendente até que esse jogo acabasse e logo de seguida iniciava a próxima partida. Resolvemos isto ao verificar se estava a decorrer um jogo na rotina de interrupção deste interruptor. Durante o período de espera entre jogos havia um problema parecido com este, que foi resolvido de forma similar. Entre jogos era possível alterar a direção inicial de um dos jogadores, resolvemos isto da mesma maneira que o problema anterior.

Quanto a funcionalidades extras, implementámos a pausa da forma que aparecia no enunciado do projeto, ou seja, com o interruptor 7 e ainda uma mensagem que aparece, assim que uma partida termina, e diz qual foi o jogador que ganhou (ou se foi empate se for o caso).