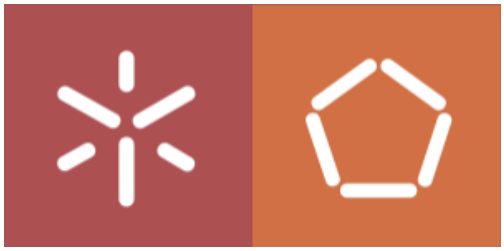
Universidade do Minho



Mestrado Integrado em Engenharia Física

Linguagens para Computação Numérica

Trabalho Prático: A física por detrás do lançamento de bolas de basquetebol

Realizado por:

Leander Reascos A89154

Magda Duarte A89144

Sara Coelho Ribeiro A89150

Índice

Introdução

O presente relatório realizou-se no seguimento do trabalho prático para a cadeira Linguagens para Computação Numérica. O problema proposto, cujo modo como foi abordado será exposto neste relatório, consiste na realização de um jogo interativo de lançamentos de *basketball* que permita a competição entre um ou mais jogadores. Tem como objetivo a realização de um lançamento que permita o encesto da bola. Teríamos que cada jogador possui 5 oportunidades por ronda, sendo estas 3 no seu total, para encestar de uma posição gerada aleatoriamente. Consoante o sucesso ou insucesso do lançamento, gerar-se-ia uma nova posição, nomeadamente, o sucesso do lance conduz a uma posição mais distante do cesto, enquanto que o insucesso de 2 lances seguidos leva ao sorteamento de uma posição mais perto deste. Conseguido o lançamento, de acordo com a zona de onde este foi feito, o jogador recebe 1,2 ou 3 pontos, estando, portanto, o campo dividido em 3 zonas distintas.

Definição de objetivos

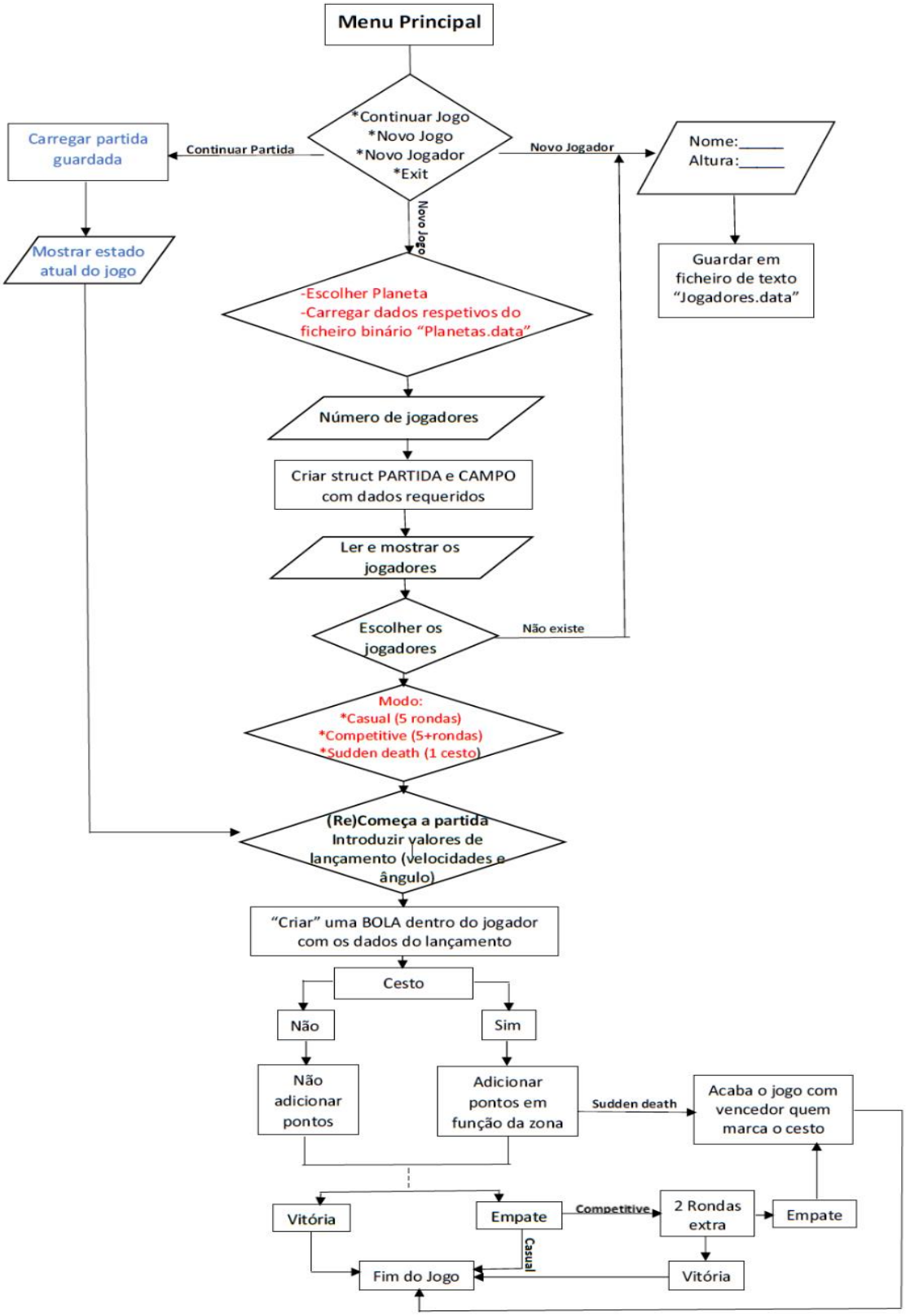
Face ao enunciado base apresentado e, simultaneamente, aos critérios de bonificação propusemo-nos a atingir os seguintes objetivos:

- Codificação das bases do jogo, ou seja, dos algoritmos requeridos no enunciado (Menu, NovoJogo, GerarArea, GerarPosicao, Print, AlturaSalto, Lançamento, ClassificaçãoFinal);
- Criação de uma funcionalidade que permita guardar a partida acabada/inacabada de forma a poder recarregar e prosseguir a mesma numa fase futura;
- Criação de diversos cenários de jogo, vários planetas: disponibilização de campos de jogo variantes conforme o planeta escolhido. Consoante a escolha, mudam as dimensões do campo, a ação da gravidade sobre o lance e a altura alcançada pelo salto. Mudam, assim, também as velocidades máximas permitidas;
- Disponibilização de três modos de jogo: Casual, Competitive, Sudden Death:
    1. **Casual:** permitirá um jogo standard cujo decorrer se divide em três rondas com 5 tentativas e cujo ambiente permite o jogo terminar em empate;
    2. **Competitive:** neste modo de jogo entramos em conta com o critério de bonificação que, basicamente, não permite o caso de empate. Terá as três rondas standard e, em caso de igualdade de pontuação, ao fim destas, decorrem duas rondas extra, ao fim das quais, se o empate ainda se verificar, dá-se a entrada na ronda Sudden Death que decide o vencedor somente pelo primeiro que marcar;
    3. **Sudden Death**: este modo, já incluído no anterior em “caso excecional”, garante, tal como antes, a vitória ao fim do primeiro lançamento bem-sucedido.

- Impressão/*Display* dos lançamentos em tempo real. A consola apresenta uma parábola que representa a trajetória da bola consoante os valores de ângulo e velocidades de lançamento e do salto. Disponibilização também da visualização do lançamento através da funcionalidade gnuplot;
- Disponibilização de um campo equiparado a um campo de basketball num contexto verídico. Desta escolha advém um sorteamento de posições que tem de ter conta as diversas formas de cada área. A partir do cesto, as áreas serão denominadas, por um critério de proximidade, 1, 2 e 3;
- Rondas com níveis de dificuldade adaptados- sorteamento da nova posição numa área que surja em conformidade com a performance do jogador nas rondas anteriores;
- Criação de obstáculos a partir das áreas 2 e 3 – permitimos a existência de obstáculos estáticos (semelhantes a um muro) e cinéticos (semelhantes a um drone) que se dividem ainda em duas subcategorias conforme o tipo de movimento; temos *drones* com um movimento dito mais aleatório, e *drones* com um movimento harmónico simples, ou seja, dotados de um movimento sequencial de cima para baixo;

Métodos e material desenvolvido

Inicialmente, fizemos um diagrama de fluxo de trabalho, onde esquematizámos como o jogo correria e, assim, tivemos uma melhor perceção do que seria necessário fazer.



Utilizámos, como controle de versões e auxiliar de interação entre o grupo, o site GitHub, onde se propuseram as tarefas por fazer, se dividiram as mesmas e através do qual se uniu o código.

Feita uma primeira divisão das tarefas, as quais se focaram na criação do menu principal, e na criação das structs base do jogo e funções a elas associadas como as de introdução de dados, e uma vez concluídas, cada elemento do grupo, com base no fluxo elaborado, encarregava-se de novas tarefas.

De um modo geral, dividimos o "problema grande" em pequenos afazeres. Em particular, desenvolvemos bibliotecas temáticas, o que facilitou a organização das diferentes funções e evitou a acumulação de um único código extenso e confuso.

Numa fase final, concluída a estrutura básica do jogo que permitia o decorrer *standard* do mesmo, prosseguimos para o melhoramento desta primeira versão, aplicando melhoramentos tanto visuais (ao nível de cores de cenários, do *layout* do jogo, e de elementos multimédia) como funcionais, de forma a otimizá-lo.

Fizemos uso da totalidade da matéria lecionada ao longo do semestre: matrizes, estruturas e ficheiros e afins. Em termos de metodologia aplicada à memória utilizada pelo jogo, tentou-se maximizar o uso de memória dinâmica, ou seja, o uso de *pointers* rentabilizando a memória RAM.

É ainda de referir que em vista ao melhoramento estético do jogo foram introduzidas funções (como a de gerar música, ou a que permite a atualização da partida numa página web) que recorrem a outras linguagens que não o C. Em termos percentuais podemos reduzir a utilização das diferentes linguagens da seguinte forma:

- C: 84%
- JavaScript: 5,9%
- Python: 4,3%
- HTML:3,1%
- CSS: 2,7%

Relativamente ao **material desenvolvido** são de salientar (além das múltiplas funções essenciais):

Ficheiros

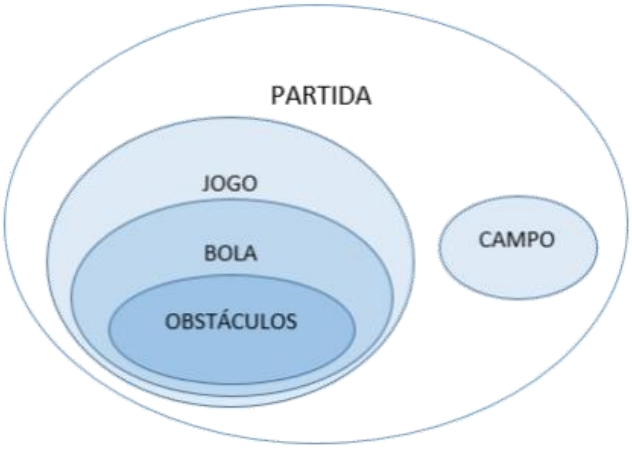
Criados para o armazenamento e consequente layout de informação indispensável ao decorrer do jogo:

- Ficheiros de texto cujo print disponibiliza as páginas de interface com o jogador (página inicial/logo, menus, campo, modos, instruções de jogo, classificações *record*);
- Dois ficheiros de texto que guardam os jogadores e as suas alturas- Jogadores.data;
- Um ficheiro binário que guarda os diferentes valores respeitantes aos diferentes planetas. Em cada posição estão um planeta e os respetivos valores da

gravidade, largura e comprimento do campo - planets.data- e outro ficheiro de texto contendo os cabeçalhos decorativos de cada planeta- planetsVisual;

- Um ficheiro de texto para o qual é exportado o resumo da partida- nome dado pelo jogador.

○ Estruturas principais



- **Obstaculo**

```
11
12  typedef struct Obstaculo{
13          float xdist;
14          float ymax;///altura do muro(1); altura inicial(2); amplitude do moviemtno*2(3)
15          float vd; //velocidade do drone
16          float ax; //aceleracao segundo x do drone
17          float ay;//aceleracao segundo y do drone
18          float teta; ////angulo que a velocidade inicial faz com a horizontal ou frequencia angular no movimento harmonico
19          //POSIÃtOES
20          float x;
21          float y;
22  }OBSTACULO;
23
```

Armazena os dados relativamente ao obstáculo, nomeadamente, altura, velocidades e acelerações dado o caráter estático ou cinético deste. Essa natureza é definida por uma variável na próxima estrutura.

- **Bola**

```
24  typedef struct bola{
25      float h0;  // Pos em y inicinal
26      float v; // Velocidade inicial
27      float ang; //angulo de lanzamiento
28      // Pos actual no tempo
29      float x;
30      float y;
31      //Matriz de la parabola
32      char **parabola;
33      //datosExtra
34      float hMax;
35      float distCesto;
36      float g;
37      int cesto;
38      int bonus;
39      int before;
40      //Obstaculo
41      int aux;
42      OBSTACULO obstaculo;
43      //settings
44      int animacion;
45      int gnu;
46      float vAni;
47  }BOLA;
```

A struct BOLA guarda dados que caracterizam a trajetória da bola: as posições inicial e atual, a parábola da trajetória, a altura máxima, a distância ao cesto e a gravidade que atua no planeta selecionado. A matriz parábola, tal como o nome insinua, guarda o "*plot*" do lançamento. Abriga ainda variáveis auxiliares ao conhecimento da validação (ou não) do cesto, à atribuição de pontos bónus e uma outra relativa ao tipo de obstáculo. Nesta struct inclui-se, igualmente, uma struct OBSTACULO.

- **Jogador**

```
17  typedef struct jogador{
18      //Base de datos
19      char *name;
20      float altura;
21      //posiÃ§Ã£o
22      float posX;
23      float posY;
24      int zona;
25      //Launch
26      BOLA bola;
27      //Rondas
28      int numeroRondas;
29      int *pontosRondas;
30      int total;
31      int *fallos;
32  }PLAYER;
```

Esta struct caracteriza o jogador. Para isso usa uma string para o nome, floats para a altura e posição do jogador, inteiros para a zona em que este se encontra (1,2 ou 3) e o número de rondas e ainda um apontador para o array dos pontos das rondas. Nesta struct encontra-se ainda uma struct BOLA guardando os valores de lançamento introduzidos por este jogador.

- **Campo**

```
4   typedef struct campoDeJuego
5   {
6       char nome[8];
7       char file[100]; //nome do ficheiro
8       //Matriz de carateres do campo de jogo
9       char **campoDeJuego;
10      int lin;
11      //dimensiones em metros do campo
12      float ancho;
13      float largo;
14      //dimenÃ§Ãµes em carateres do campo (vem do ficheiro)
15      int aChar;
16      int lChar;
17      //gravidade
18      float g;
19  }CAMPO;
20
```

A struct CAMPO caracteriza o campo de jogo que é imprimido na consola. É descrito através de dados relativos ao planeta tais como o nome e o valor de gravidade e as suas dimensões, tanto em metros como em caracteres.

- **Partida**

```
34  typedef struct partida{
35      int modoDeJogo;
36      int njogadores;
37      int nRondas;
38      int nPlaneta;
39      PLAYER *players;
40      CAMPO  campo;
41      int tentativas;
42
43      //settings
44      SETTINGS settings;
45
46  }PARTIDA;
47
```

A struct PARTIDA tem um inteiro que é característico de cada modo de jogo e mais 4 que definem o número de jogadores, o número da ronda, o número característico de cada planeta e o número de tentativas. Nesta estrutura está ainda uma estrutura do tipo CAMPO, que, tal como foi indicado antes, caracteriza o campo de jogo.

Funções de maior importância

Na função main, além de termos a sequencialização das funções criadas que permitem o decorrer do jogo, temos o menu principal requerido que disponibiliza as diferentes opções:

A função main encontra-se em “lcn.c”, que tem ainda duas funções, sendo de salientar a que permite a leitura dos jogadores selecionados.

Estando o programa dividido por bibliotecas, podemos, agora, salientar, em cada uma destas, algumas das funções mais importantes:

○ main.h

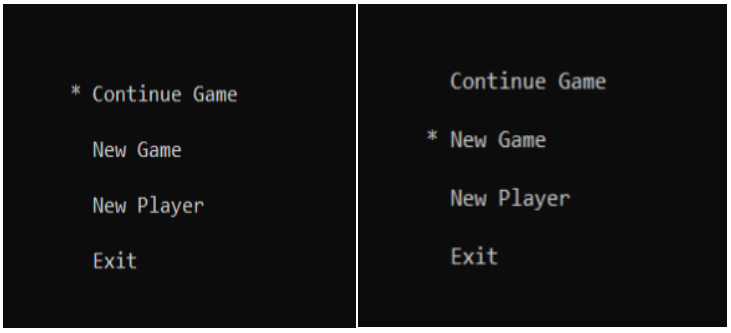
- **mainSelect**: função que possibilita o *interface* do jogador com a consola a partir de comandos usuais (*keys* do teclado). É utilizada para os diversos menus de seleção disponibilizados ao jogador ao longo da inicialização do jogo.
  Funciona com base no char devolvido pelo *keyboard* (getch()) e a partir de um “ciclo viciado” (while (1)), abandonado apenas quando premida a tecla ENTER.

```
60  while(1){
61    printMenu(menu,i,opcion);
62    char cTecla;
63    cTecla = getch();
64    if(cTecla == 0)
65      cTecla = getch();
66    else
67    switch(cTecla)
68    {case 13://ENTER
69        freeMemoryMain(menu,i);
70        return opcion+1;
71        break;
72      case 9://TAB
73        break;
```

Trabalha ainda com a ajuda de funções como printHeader, transversal ao programa todo, responsável por fazer o print do logótipo do jogo, e printMenu, que, recebendo a matriz menu (previamente inicializada com o tamanho adequado para aquele que se pretende imprimir lido a partir do respetivo ficheiro), faz a colocação devido do caracter '*' na matriz, cuja posição depende do char devolvido pelo utilizador. Este caracter funciona, portanto, como um cursor. A cada interação do utilizador com a consola, que não ordene a escolha de uma das opções do menu (através do ENTER), corresponde um novo print do menu tendo em conta o char opção devolvido, ou seja, o posicionamento de '*'.

```
47  while(fgets(buffer,sizeof(buffer),fp)){
48    if(i > 0) {
49      menu = (char **) realloc(menu, (i+1)*sizeof(char *));
50      if(menu == NULL) exit(-1); }
51    menu[i] = (char *)malloc((2+strlen(buffer))*sizeof(char));
52    menu[i][0] = ' ';
53    menu[i][1] = ' ';
54    strcpy((menu[i]+2),buffer);
55    i++;            }
56
```

```
23  void printMenu(char **m, int n, int pos){
24    int i;
25    system("cls");
26    printHeader();
27    printf("\n\n");
28    m[pos][0] = '*';
29    for(i=0; i<n; i++){
30      if(i!=pos){
31        m[i][0] = ' ';
32      }
33      printf("\t\t\t\t\t%s\n",m[i]);
34    }
35
36  }
```



- settings.h
  - **segundoMenu, selectNumber, select, boleanMenu:** funções de interação com o jogador que lhe permitem personalizar vários aspetos do jogo.

- novojogador.h
  - **novoJogador**: função que permite a inserção por parte do utilizador de uma "nova personagem", um novo jogador para além daqueles que possibilitamos inicialmente (importados do ficheiro "Base_de_datos/Jogadores.data").

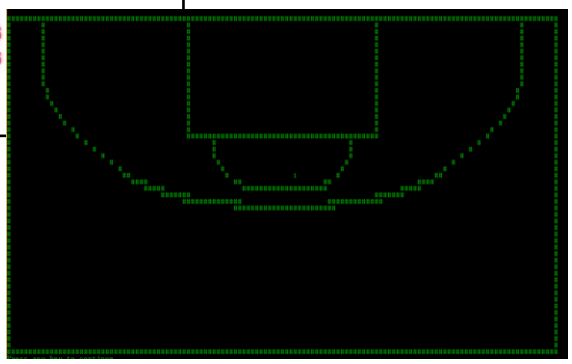
- campoDeJogo.h
  - **addJugador:** função que adiciona o jogador ao campo, ou seja, coloca o caracter

que o identifica na matriz do campo.

- **printCampo:** função que imprime o campo e os jogadores, a partir de números identificadores da ordem pela qual jogam, nas devidas posições.

```c
void addJugador(float x, float y, char n, CAMPO *campo){
    int posX,posY;
    posX = (int) (x*campo->lChar/campo->largo);
    posY = (int) (y*campo->aChar/campo->ancho);
    campo->campoDeJuego[posY][posX]=n;
}

void printCampo(CAMPO *campo){
    int i;
    for(i=0;i<campo->lin;i++){
        printf("%s",campo->campoDeJuego[i]);
    }
}
```



○ valoreslaunch.h

- **readTecla:** função que lê a tecla pressionada e cria uma variável que vai ser utilizada para controlar as variáveis do jogo e a música.

- **distCesto:** função que determina a distância ao cesto.

```c
int readTecla(){
    char cTecla;
    int var = -1;
    do{
        cTecla = getch();
        if(cTecla == 0)
          cTecla = getch();
        else
        switch(cTecla)
        {
          case 13://ENTER
            var = 0;
            break;

          case 9://TAB
            var = 9;
            break;

          case 72://ARRIBA
            var = 72;
            break;

          case 80://ABAJO
            var = 80;
            break;

          case 75://IZQUIERDA
            var = 75;
            break;

          case 27://ESC
            var = 27;
            break;
          case 77://DERECHA
            var = 77;
            break;
          case 'p'://Pause music
            Musica('p');
            break;
          case 'n':
            Musica('n');
            break;
          case 'b':
            Musica('b');
            break;
          case 's':
            Musica('s');
              break;
          default:
            break;
        }
    }
    while(var == -1);
    return var;
}
```

○ launch.h

- **gerarObstaculo**: função que tal como o nome indica, conforme bola->aux, que transporta o inteiro ditador da existência e do tipo de obstáculo (cinético ou estático), constrói a estrutura do obstáculo, ou seja, preenche os respetivos "argumentos".

- **verificaCesto**: função responsável pela verificação do lance ao longo do *plot* do lançamento. Verifica, num "instante", se este foi bem-sucedido ou não. Entra em conta com um critério de validação que considera cesto todos os lançamentos que concluam num movimento tal que: nos instantes em que, pela primeira vez, a bola esteja nos arredores do cesto (enX=1) chegue com uma altura superior ao cesto (beY=1) e, nos instantes em que transponha os limites do "diâmetro do cesto", obedeça às 4 fronteiras, ou seja, aos limites do x e do y. Ambos critérios devem ser validados para que ocorra cesto. O primeiro resulta em bola->before=1 e o segundo em enX=1 e enY=1, concluindo em bola->cesto=1.

- **parabola**: função que implementa o código necessário à impressão da imagem do lançamento em tempo real.

  Começa por "inicializar", através da função newBola, alguns argumentos da estrutura bola, isto é, anulá-los, nomeadamente, as variáveis cesto e before, ditadoras da validação do lançamento quando ambas apresentam o valor 1.

  Dá-se a estruturação base da matriz bola->parabola, ou seja, a introdução de caracteres definidores do campo, em si "estático", e dos obstáculos, caso seja o caso (bola->aux=1, obstáculo do tipo estático).
  No caso cinético (bola->aux>1), a sua inserção recorre à atualização de uma variável "definidora do tempo".

```
206     if(bola->aux == 1){
207
216
217         for(i=sizeLinhas-1; i>posYobs; i--){
218             bola->parabola[i][posXobs]='|';
219         fclose(obs);
220         }
221     }
```

  Inicializando essa tal variável, dentro de um ciclo do-while:

  1. Executa-se a função launch_time que, por sua vez, além de incorporar a verificaCesto, atualiza a posição da bola e do *drone* no caso de este ser móvel (bola->aux>1), através das típicas equações x(t) e y(t) do movimento de um projétil (bola e obstáculo caso bola->aux=2) ou das de um movimento harmónico (bola->aux=3).
     É verificada a ocorrência de colisão entre a bola e o obstáculo quando a posição da bola revela uma certa proximidade à do obstáculo, mais concretamente, em que bola->x pertence a [bola->obstaculo->x-0.3; bola->obstaculo->x+0.3].
     O "tempo" cresce com um incremento de 0,01 segundos sendo a posição da bola e do obstáculo (bola->aux>1) atualizados. É também nesses

instantes implementada a função que procede à verificação do cesto.

```
292                 t += 0.01;
293             }while(bola->y > 0 && pasoObstaculo);
294
295             if(bola->aux > 1 || obs != NULL){
296                 fclose(obs);
297             }
```

```
258             float t = 0;
259             float dt = 0;
```

2. A função, na matriz bola->parabola, guarda o caracter '@' na posição da bola nesse instante, imprimindo, logo "de seguida", a matriz obtida por essa transformação. Já na posição do obstáculo no caso móvel anteriormente referido (bola->aux>1) é colocado '#' de acordo com a atualização da sua posição.
   No seio desta função, verificamos ainda a recorrência à função printMatriz que executa o papel que o nome da própria insinua.

```
257         if(bola->x>=0 &&bola->y>0 && x<=sizeColumnas && y>=0){
258             bola->parabola[y][x]='@';
259             printMatriz(bola->parabola,sizeLinhas,sizeColumnas);
260         }
```

Ainda inerente ao plot do lançamento temos a função printParabola, que, fazendo uso da funcionalidade gnuplot, guarda vários dados, em ficheiro, relativos ao campo, ao lançamento e aos obstáculos, trabalhando, assim, para um plot que resulta numa melhor visualização gráfica, mas não em tempo real, do lançamento.

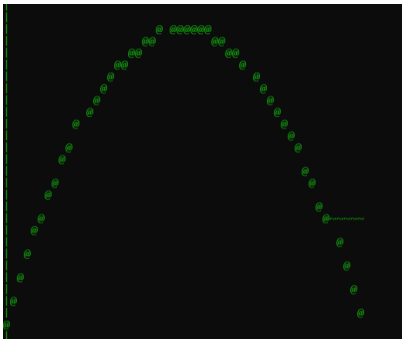
```
143 void printParabola(BOLA *bola){
144     FILE *gp;
145     gp = popen(GNUPLOT, "w");
146     if (gp == NULL) {
147         printf("Erro ao abrir pipe para o GNU plot.\n"
148             "Instale gnuplot, dependencia necesaria para o funcionamento completo do programa\n");
149         exit(0);
150     }
151     fprintf(gp, "set yrange[0:%f]\n", bola->hMax);
152     fprintf(gp, "set xrange[0:%f]\n", bola->distCesto);
153     fprintf(gp, "set object circle at first %f,%f radius char %f \n",bola->distCesto-0.012,hCesto,diametroCesto);
154     fprintf(gp, "plot x\n");
155     if(bola->aux > 0)
156         fprintf(gp, "plot 'obstaculo','%s'\n",GRA
157     else
158         fprintf(gp, "plot '%s'\n",GRAPH);
159
```

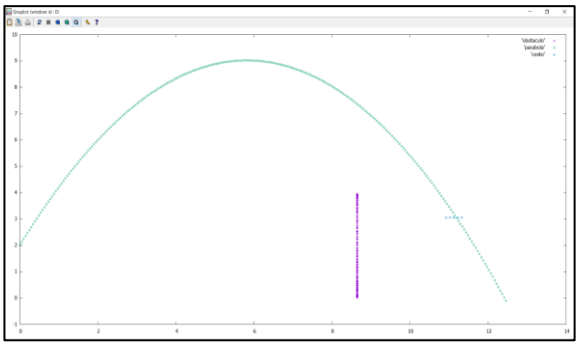
```
211         while(ay <= bola->obstaculo.y){
212             fprintf(obs, "%.3f \t %.3f \n", bola->obstaculo.x,ay);
213             ay+=0.03;
214         }
```

```
239         fprintf(fp, "%.3f \t %.3f \n", bola->x,bola->y);
240
```

```
244     if(bola->aux > 1){
245         y = (int)(sizeLinhas-bola->obstaculo.y*carateresPorMetro);
246         x = (int)(bola->obstaculo.x*carateresPorMetro);
247         fprintf(obs, "%.3f \t %.3f \n", bola->obstaculo.x,bola->obstaculo.y);
```

O ciclo é interrompido consoante a ultrapassagem por parte da bola dos limites impostos ao campo ou a ocorrência de choque entre o obstáculo e a bola.





- partida.h
  - **veRondas:** função que conta os pontos dos jogadores e que, em caso de empate no modo *competitive*, acrescenta duas rondas, assim como procede ao aumento de memória no caso *SuddenDeath* quando houver a necessidade mais tentativas.

  - **gerarPosicoes:** função que gera aleatoriamente as posições dos jogadores em cada área (1,2 ou 3). As áreas são definidas matemática e implicitamente nesta função tendo em conta, dada a dinamização do campo conforme o planeta, para a determinação das dimensões do campo, as "razões das medidas no campo terrestre". Como já foi antes explícito, através desta função disponibilizamos um campo a duas dimensões com zonas definidas como na realidade (por parábolas e semicircunferências).

  - **posRandomRonda :** função que assegura as mesmas características iniciais para todos os jogadores: o bónus, a zona e a posição na zona.

```
269 void posRandomRonda(PARTIDA *partida, int n){
270     int i;
271     readCampo(&partida->campo);
272     gerarPosicoes(&partida->players[0],&partida->campo,n);
273     for(i=0; i<partida->njogadores; i++){
274         partida->players[i].bola.bonus = 0;
275         partida->players[i].zona = n;
276         partida->players[i].posX = partida->players[0].posX;
277         partida->players[i].posY = partida->players[0].posY;
278         addJugador(partida->players[i].posX,partida->players[i].posY,'I',&partida->campo);
279     }
280 }
```

  - **adaptaZona:** função que adapta as zonas tendo em conta a zona em que o jogador está e se marcou ou não cesto. No primeiro caso, "aumenta" a zona se o jogador acertar no cesto caso este não esteja na zona 3; no segundo caso, "diminui" a zona se o jogador falhar duas vezes e não estiver na área 1; no caso 3 o jogador acertou no cesto mas já se encontra na zona 3, por isso, apenas é deslocado para uma posição mais distante do cesto e no caso 4 o jogador falhou 2 vezes mas já se encontra na zona 1, daí que apenas se atualize a sua posição

para mais perto do cesto (os dois últimos casos ocorrem apenas quando a posição original não está já nos limites do campo).

```
456         switch(n){
457
458             case 1:                                                    //////////////////////ZO
459                 gerarPosicoes(player,campo,player->zona+1);
460                 player->zona++;
461             break;
462
463             case 2:                                                  //////////////////////ZONA MAIS PERT
464                 gerarPosicoes(player,campo,player->zona-1);
465                 player->zona--;
466             break;
467
468             case 3:
469                 gerarPosicoes(player,campo,player->zona);
470                 distatual1=distCesto(player->posX,player->posY,campo->largo,campo->ancho);
471                 if(!(distoriginal>=MAXIMO-MAXIMO*0.002)){
472                     while (distatual1<=distoriginal ){
473                     gerarPosicoes(player,campo,player->zona);
474                     distatual1=distCesto(player->posX,player->posY,campo->largo,campo->ancho);
475                     }
476                 }
477                 break;
478
479             case 4:
480                 gerarPosicoes(player,campo,player->zona);
481                 distatual2=distCesto(player->posX,player->posY,campo->largo,campo->ancho);
482                 if(!(distoriginal<=MINIMO+0.1)){
483                     while (distatual2>=distoriginal  && distatual2!=MINIMO){
484                         gerarPosicoes(player,campo,player->zona);
485                         distatual2=distCesto(player->posX,player->posY,campo->largo,campo->ancho);
486                     }
487                 }
488                 break;
489         }
```

- **runRonda:** função que permite o decorrer da tentativa. Inclui a função printPartida que imprime na consola as informações da ronda atual e, para além disso, faz o requerimento dos valores necessários para o lançamento, nomeadamente as velocidades de salto e lançamento e o ângulo de lançamento. Permite o lançamento e a sua verificação.
- **Informacao**: função que imprime a informação do jogo num ficheiro de texto com o nome indicado pelo utilizador e no website utilizando a função fprintf.

- **ConfirmRonda:** função que, tendo em conta a variável bola.cesto e o número de vezes que o jogador falhou o cesto (bola.f), o reposiciona utilizando a função adaptaZona.

- **playGame:** função que encadeia as funções necessárias para o decorrer do jogo e que faz a diferenciação entre o decorrer dos modos.

Conclusões

Objetivos atingidos

Tendo em vista os objetivos com que nos comprometemos no início do trabalho, podemos concluir que apenas um não foi alcançado: a funcionalidade que permitiria o carregamento de um jogo e a sua continuação a partir do "momento em que foi abandonado".

Fora esse, foram cumpridos os seguintes:

- Disponibilização dos vários cenários de jogo (planetas);
- **Impressão dos lançamentos em tempo real;**
- Disponibilização dos 3 modos de jogo com as características enunciadas anteriormente (**inclusive um modo que em caso de empate gera 2 rondas extra, e em caso de empate neste entra-se em Sudden Death**);
- Disponibilização de um campo dito "real";
- **Incremento a dobrar da pontuação em caso de lançamentos com sucesso sucessivos**;
- **Decorrer de um jogo de nível de dificuldade personalizado e progressivo (determinação aleatória de posições por zona e determinação da primeira posição de acordo com a ronda - ronda 1 na zona 1, ronda 2 na zona 2 e ronda 3 para a frente na zona 3)**;
- **Desenvolvimento dos diferentes tipos de obstáculos aleatórios e do algoritmo que permite verificar a ocorrência de colisão entre a bola e este** (apenas disponibilizados nas zonas 2 e 3, na 2 estáticos e na 3 cinéticos).

Para além destes já estabelecidos no início do trabalho, com vista ao melhoramento, foi ainda possível alcançar os que se seguem:

- Disponibilização de um menu permitindo configurar diferentes *settings* do jogo, entre eles a música, os vídeos, o *plot* do lançamento, a velocidade da animação correspondente e o número de rondas e tentativas;
- Interação do jogador com a consola por comandos mais práticos como a utilização, portanto, de comandos comuns do teclado (setas, letras…) para introdução de valores necessários ao jogo bem como para a seleção de opções nos menus;
- Disponibilização de uma *playlist* e apresentação de vídeos comemorativos ou jocosos conforme os lançamentos bem-sucedidos ou falhados (em phyton);
- Dinamização do número de rondas e tentativas por ronda;
- Disponibilização de uma lista dos detentores de recordes no jogo, atualizada conforme os resultados dos novos jogadores superem ou não os já registados.
- Disponibilização numa página web dos dados do último jogo (em java e html).

Dificuldades apresentadas

Ao longo da execução do trabalho, além das dificuldades que surgiram na tentativa de construir a funcionalidade de Carregar Partida, enfrentámos outras mais pequenas que acabámos por ultrapassar:

- Perante a função de verificação de cesto, numa fase em que trabalhávamos com o tempo do sistema, deparámo-nos com uma dificuldade na mesma que advinha do facto de esta estar construída em torno das funcionalidades do decorrer do tempo. Na altura, esta validação decorria no tempo usado para a construção da parábola, desse modo, este tempo sofria de "saltos" que ora avançavam o instante em que era cesto, não o contabilizando, ora não, conforme os ditos "saltos temporais" com origem no próprio tempo de execução do código intermediário, tal como o gasto no print da parábola.
  Este problema foi solucionado implementando a função de validação de cesto de um modo diferente, não recebendo o tempo t inicializado em primeiro lugar mas um tempo (t-dt) correspondente ao tempo real do lançamento cuja correspondência se deve ao desconto ponderado do tempo extra consumido ao longo da execução do programa de 0.03 segundos. No entanto procedemos ainda à elaboração de uma versão mais entendível que consiste no incremento "manual" do tempo em 0,1 segundos tal como foi explorado anteriormente.

Quanto ao objetivo que acabámos por abandonar, que diz respeito à disponibilização da funcionalidade de Carregar Partida, revelou dificuldades ao nível do guardar dos dados. O maior problema residiu na gravação em ficheiros das structs que criámos, dado o facto de estas por si só terem já structs nos seus "argumentos". A gravação em ficheiros dessas revela-se, assim, embora possível, com nível de dificuldade acrescido pelo simples facto de terem interiormente outras structs.

Discussão dos resultados obtidos

Após a realização deste trabalho podemos concluir que os resultados foram muito favoráveis, visto que conseguimos realizar todos os requisitos que nos foram propostos. Em relação às características que seriam bonificadas, todas foram implementadas no nosso jogo. Para além disso, decidimos acrescentar alguns "extras" que considerámos relevantes e/ou divertidos no contexto do jogo, tais como: os diferentes cenários de jogo (planetas), a impressão do campo na consola, uma interação mais visual do jogador com os valores do lançamento, a disponibilização de alteração de características do jogo através dos *settings* e, ainda, a utilização de música e vídeos para tornar o jogo mais apelativo.

○ Possíveis melhorias

No final do trabalho verificámos que existem alguns aspetos que poderiam ser melhorados. Um desses aspetos é a impressão do jogador no campo da consola, isto porque o campo imprimido na consola tem dimensões em caracteres (no eixo dos x 129 caracteres e no eixo dos y 52 caracteres) e as posições geradas têm dimensões em metros que se alteram em função do planeta (na Terra são: no eixo dos x 15m e no eixo dos y 14 m). Deste aspeto resulta a ilusão de que o jogador não calhou na zona de jogo correta.

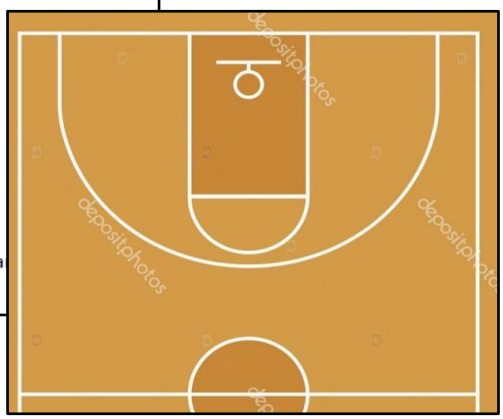
```
4
5   typedef struct campoDeJuego
6   {
7       char nome[8];
8       char file[100]; //nome do ficheiro
9       //Matriz de carateres do campo de jogo
10      char **campoDeJuego;
11      int lin;
12      //dimensiones em metros do campo
13      float ancho;
14      float largo;
15      //dimenÃ§Ãµes em carateres do campo (vem
16      int aChar;
17      int lChar;
18      //gravidade
19      float g;
20  }CAMPO;
21
```

```
22  void addJugador(float x, float y, char n, CAMPO *campo){
23      int posX,posY;
24      posX = (int) (x*campo->lChar/campo->largo);
25      posY = (int) (y*campo->aChar/campo->ancho);
26      campo->campoDeJuego[posY][posX]=n;
27  }
28
```

```
37  void printCampo(CAMPO *campo){
38      int i;
39      for(i=0;i<campo->lin;i++){
40          printf("%s",campo->campoDeJuego[i]);
41      }
42  }
```

Para além disso, a imperfeita correspondência entre o campo imprimido e o definido matematicamente (na imagem), faz com que pareça que os jogadores não estão na área pretendida.

```
154 void gerarPosicoes(PLAYER* jogador,CAMPO* campo,int n){
155
156     //LIMITES ZONA 1
157     float xmin1=(5.05/15)*(campo->largo);
158     float xmax1=(campo->largo)-xmin1;
159
160     float rquadrado=pow((2.45/15)*(campo->largo),2);
161
162
163     //LIMITES ZONA 2
164
165     float xmin2=0.9/15*(campo->largo);
166     float xmax2=(campo->largo)-xmin2;
167     float a=-8.95/14*(campo->ancho)/pow((xmin2-(campo->la
168
169
```



Segue o fluxo atualizado: