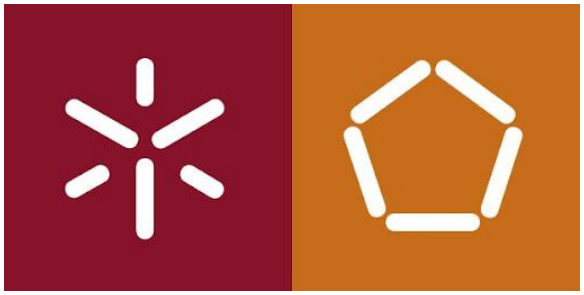

Universidade do Minho

Relatório do Projeto Fase 3

Ano letivo 2021/2022

Maio 2022

Licenciatura em Engenharia Informática

Unidade Curricular de Computação Gráfica

Ana Gonçalves a93259

Bruno Pereira a93298

Francisco Toldy a93226

João Delgado a93240

Índice

Introdução 3
Patches de Bezier 4
VBO's 8
XML Parser 9
Curvas de CatmullRom 11
Rotações com Duração 14
Testes com os Ficheiros Providenciados 15
Protótipo de Sistema Solar 16
    Planetas 16
    Lua 18
    Cometa 18
Funcionalidade Extra 21
Conclusão 22

Introdução

Neste ficheiro vamos apresentar o resultado de desenvolvimento da fase 3 do projeto, que consistia em 3 grandes requisitos. O primeiro trata-se da criação de figuras a partir de patches de bezier, fornecidos a partir de um ficheiro com uma sintaxe bem definida . Seguidamente, a utilização de VBO's, isto é, todos os pontos vão ser enviados para a placa gráfica, o que previne o envio desses mesmos pontos para a placa gráfica a cada renderização ou um de cada vez. Finalmente, o desenvolvimento de movimentos por curvas de Catmull-Rom.

Logo, neste ficheiro vai ser apresentada a solução para cada um destes pontos, seguido de uma demonstração dos resultados sobre os ficheiros testes. Finalmente vai ser apresentado o avanço no modelo do sistema solar e a conclusão.

Patches de Bezier

A função responsável pela conversão dos ficheiros do tipo patch para ficheiros .3d é a função “writeBezier”. No entanto, de modo a obter esse mesmo ficheiro baseado em patches de Bezier é necessário primeiro fazer o parse do ficheiro do patch respetivo. Essa função, “parsePatch”, é chamada na função “writeBezier”. A função “parsePatch”, ao receber o path para o ficheiro patch, vai conseguir armazenar o nº de patches, os índices dos patches, o número de pontos de controlo e os pontos de controlo.

```
void parsePatch(patchFile) {
    string lineRead
    ifstream patchFileStream
    const char delimiter is ","

    if (file is open) {
        while (get one line)) {
            if (is first Line) {
                numberOfPatches is equal to the first number in the first line
            }
            else if (is line with indexes) {
                add patch index to patches_indeces
            }
            else if (is line with number of control points) {
                number of controlo points is equal to the number in that line
            }
            else if (is line with control points) {
                add control point of that line to control_points
            }
            go to next line
        }
        close file
    }
    else show "Unable to open file"
}
```

Figura 1: Pseudocódigo da função "ParsePatch"

Após a chamada de “parsePatch” é chamada a função “writeSurface” que ao receber a tesselagem e o nome do ficheiro destino passados como argumentos, vai escrever nesse mesmo ficheiro os pontos dos triângulos da figura desejada. Com este objetivo, é precalculado cada uma das matrizes que vão permitir obter os valores das coordenadas em cada um dos eixos que vai permitir concluir o cálculo de cada uma das coordenadas dos pontos com base nas diferentes posições na grelha.

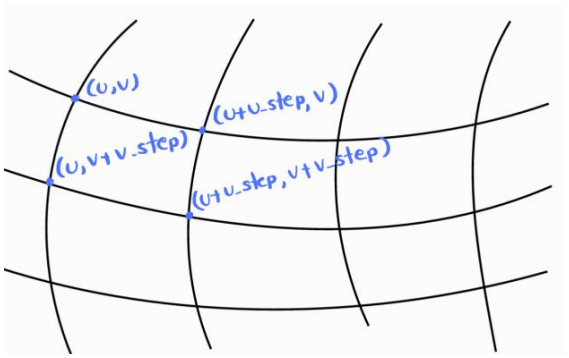

Figura 2: Ilustração da grelha para o desenho de patches

```
void writeSurface(tesselation, fileName){

    u_step is equal to the difference value between the u points (1/tesselation)
    v_step is equal to the difference value between the v points (1/tesselation)

    open file stream

    for(each patch){
        calculate the precalculated x, y and z matrixes

        for(each u value){
            for(each v value){

                ponto1 equals to point in the grid with u and v values
                ponto2 equals to point in the grid with next u and v values
                ponto3 equals to point in the grid with u and next v values
                ponto2 equals to point in the grid with next u and next v values

                write to file ponto1
                write to file ponto3
                write to file ponto4

                write to file ponto1
                write to file ponto4
                write to file ponto2
            }
        }
    }
    close file
}
```

Figura 3: Pseudocódigo da função "writeSurface"

No entanto, esta função (”writeSurface”), chama outras 2 funções muito importantes, “precalculate” e “getSurfacePoint”. Começando pela primeira função, “precalculate” que vai precalular as operações que são constantes num mesmo patch em relação a cada coordenada, que consiste simplesmente na obtenção de 3 matrizes precalculadas, cada uma resultante da multiplicação da matriz de bezier pela matriz de pontos de controlo sobre um dado eixo e também pela inversa da matriz de bezier, como representado na seguinte fórmula:

$$p(u,v) = [u^3 \quad u^2 \quad u \quad 1] M \begin{bmatrix} P_{00} & P_{10} & P_{20} & P_{30} \\ P_{01} & P_{11} & P_{21} & P_{31} \\ P_{02} & P_{12} & P_{22} & P_{32} \\ P_{03} & P_{13} & P_{23} & P_{33} \end{bmatrix} M^{-1} \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$$

```
void preCalculate(currentPatch) {

    aux[4][4] is auxiliar matrix in between operations
    controlPoints[4][4] is the matrix with one of the coordenates of the control points

    m[4][4] is the fixed bezier matrix

    mTransposed[4][4] is the matrix that will be occupeded by one transposed matrix
    put in mTransposed the transposed of the fixed bezier matrix

    get the matrix with x coordenates of the control points
    multiplicate fixed bezier matrix with matrix with x coordenates of control points resulting in the aux matrix
    multiplicate the fixed bezier transposed matrix with mTransposte resulting in the final constant matrix of x coordenates

    get the matrix with y coordenates of the control points
    multiplicate fixed bezier matrix with matrix with y coordenates of control points resulting in the aux matrix
    multiplicate the fixed bezier transposed matrix with mTransposte resulting in the final constant matrix of y coordenates

    get the matrix with z coordenates of the control points
    multiplicate fixed bezier matrix with matrix with z coordenates of control points resulting in the aux matrix
    multiplicate the fixed bezier transposed matrix with mTransposte resulting in the final constant matrix of z coordenates

}
```

Figura 4: Pseudocódigo da função "Precalculate"

Já com as 3 matrizes constantes para cada patch calculadas, a função “getSurfacePoint” ao receber os valores de u e v vai devolver as 3 coordenadas do ponto na grelha, em que cada um deles é obtido a partir da multiplicação do vetor u pela matriz precalculada sobre um dado eixo e finalmente pelo vetor coluna v.

```
float* getSurfacePoint(float u, float v){

    values is going to be filled with the final 3 coordenates to return

    for(each value) value is equal to 0

    uVector[4] is the vector u
    vVector[4][1] is the vector v

    aux[4] is going to be used as auxiliar in between operations

    aux is equal to the multiplication of the u vector with the precalculated matrix with x coordenates
    first value is equal to the multiplication of the aux vector with the column vector

    aux is equal to the multiplication of the u vector with the precalculated matrix with y coordenates
    second value is equal to the multiplication of the aux vector with the column vector

    aux is equal to the multiplication of the u vector with the precalculated matrix with z coordenates
    third value is equal to the multiplication of the aux vector with the column vector
```

Figura 5: Pseudocódigo da função "getSurfacePoint"

No entanto para conseguir obter cada uma das matrizes dos pontos de controlo de cada uma das variáveis (utilizado na função “preCalculate”) é chamada a função “getControlPointsMatrix”. Esta última função, com base no patch atual e a variável desejada vai aceder às coordenadas dos pontos de controlo obtidos a partir do parser e devolver a matriz da variável input de cada um dos pontos de controlo:

```
void getControlPointsMatrix(currentPatch, coord, matrix[4][4]) {
    aux = 0;
    for (each line) {
        for (each column) {
            index is equal to the value in patches_indices according to the current patch and number of indexes already read
            matrix[i][j] is equal to the value in control_points in the current index according to the variable desired
            increment control points read
        }
    }
}
```

Figura 6: Pseudocódigo da função "getControlPointsMatrix"

VBO’s

A implementação dos VBO’s implicou fazer alterações ao programa Engine. Até à fase anterior, o parser tinha a capacidade de armazenar os pontos de cada ficheiro lido num vetor de vetores . Esse vetor de vetores seria processado pela função drawPrimitive.

Para implementar VBO’s o grupo começou por preparar os dados, para tal, começou-se por fazer alterações na função parser. O método parser percorre o ficheiro de um dado modelo, inserindo cada uma das coordenadas dos pontos lidos num vetor de floats tmp. Depois, inserimos o método glBindBuffer, glVertexPointer. O método glBindBuffer marca o VBO correto como ativo, neste caso o VBO ativo será o último buffer no array de buffers. Este índice é explicado pelo facto de numa dada execução do parser, o modelo que está a ser lido, logicamente, é último modelo a ser lido.

Acrescentámos ainda o método glBufferData , com o cálculo do tamanho do buffer efetuado ao multiplicar o tamanho de um float pelo tamanho do array tmp onde são colocadas as coordenadas de cada ponto lido. Este glBufferData terá como objetivo colocar os pontos do modelo lido nesta execução do parser, e armazenados em tmp no array de buffers global.

Os buffers preenchidos em cada uma das execuções necessárias do método parser estão apontados por uma variável global.

Importante referir que a cada leitura de um modelo, a variável filesRead aumenta, aumentando também o número de buffers no array de buffers global.

De seguida, teríamos que preparar o desenho. Para tal, na função drawPrimitive, removemos o ciclo existente na 2ª fase para desenho de cada um dos pontos lidos, componente da solução apresentada na segunda fase. Substituímos esse ciclo pela função glBindBuffer, glVertexPointer e glDrawArrays.

```
void drawPrimitive(int fileIndex){
    glColor3f(1.0f, 1.0f, 1.0f);

    glBindBuffer(GL_ARRAY_BUFFER, buffers[fileIndex]);
    glVertexPointer(3, GL_FLOAT, 0, 0);
    glDrawArrays(GL_TRIANGLES, 0, sizes.at(fileIndex));
}
```

Figura 7: Função "drawPrimitive"

XML Parser

Até esta fase do projeto, cada uma das transformações era única. No entanto, com a possibilidade de ocorrer translações e rotações com especificações diferentes foi necessário realizar alterações ao parser xml de modo a obter todos os valores necessários para realizar essas mesmas operações.

Relativamente ao translate, a existência de uma translação sobre uma curva obrigou a uma adaptação do parser quando reconhecia um elemento com o id de “Translate”, e que se existisse um atributo nesse elemento com o nome “Time” vai de seguida obter o valor do atributo “Align” e os pontos de controlo que se seguem.

Se esse elemento “Time” não existir vai tratar essa translação como uma translação normal.

Com a seguinte alteração feita ao parser, vai ser possível aceder a um vetor com arrays de 3 valores, isto é, um ponto, um vetor com o número de pontos por curva que vai ser útil para calcular os pontos que representam uma dada curva, um vector com os tempos de translações e um vetor com os aligns de translações.

```
else{
          float value of attribute time is added to translateTimes
          char* value of attribute align is added to translateAlign
          add the value corresponding to translation within a curve (7.0f)
          l_point is the tinyXml element of the first control point

          start counting number of control points

          while(exists control points){
              translate_x is the float  value of the attribute x
              translate_y is the float  value of the attribute y
              translate_z is the float  value of the attribute z

              pt will agregate 3 coordenates for 1 point

              add to pt translate_x
              add to pt translate_y
              add to pt translate_z

              increment number of control points
              l_point is the next tinyXml point element
          }
      add number of points to nrPointsPerCurve
      increment number of curves
      }
```

Figura 8: excerto de pseudocódigo da função "readTransformations"

A segunda e última alteração feita no xml_parser foi relativa à rotação com diferentes especificações também, que obrigou a repetir o processo desenvolvido para o translate anteriormente apresentado. Logo, para distinguir entre uma rotação normal e uma rotação com as especificações novas é verificado se existe o atributo “Time” no elemento Rotation.

Se esse atributo existir, vai guardá-lo num vetor de tempos de rotações, como também os atributos x, y e z iguais aos de uma rotação normal.

```
else{
      float value of attribute time is added to rotateTimes
      rotate_x is the float value of the attribute x
      rotate_y is the float value of the attribute x
      rotate_z is the float value of the attribute x
      add the value corresponding to rotation within timestamp (8.0f)

      add to rotatePoint vector rotate_x
      add to rotatePoint vector rotate_y
      add to rotatePoint vector rotate_z
  }
```

Figura 9: Excerto de pseudocódigo da função "readTransformations":

Assim sendo, com as alterações realizadas no parser, conseguimos verificar que na renderização de cada uma das operações (translação, rotação, escala, pushMatrix, popMatrix e modelo) foram adicionadas 2 operações: translação sobre uma curva e rotação com uma duração variável.

Logo, a função alterada para acomodar estas alterações é a função “Draw” , chamada no renderScene. Em semelhança as outras operações, existirão os seguintes casos para ter em consideração: Curvas de CatmullRom e Rotações com duração.

Curvas de CatmullRom

Para a criação desta transformação temos vários processos a realizar, sendo o primeiro o desenho efetivo da curva a realizar, obtido a partir da função “renderCatmullRomCurve”.

A função “renderCatmullRomCurve” consiste simplesmente em utilizar um nível de tesselagem (o escolhido foi 100) para obter cada um dos pontos da curva e desenhar um loop que percorre esses mesmos pontos.

```
void renderCatmullRomCurve() {

    pos[4] will be occupied by each of the curve's point
    deriv[4]; will be occupied by each of the curve's points derivative but will not be used at this point

    vector x that will contain all components of coordenates

    for (100 times) {
        assign to pos the value of a point in current tesselation value
        add coordenate x to x
        add coordenate y to x
        add coordenate z to x
    }

    bind buffers
    add data pf x to buffers

    bind buffers
    define sintax of buffers as an array
    draw array as loops

}
```

Figura 10: Pseudocódigo da função "renderCatmullRomCurve"

A segunda e última alteração feita no xml_parser foi relativa à rotação com diferentes especificações também, que obrigou a repetir o processo desenvolvido para o translate anteriormente apresentado. Logo, para distinguir entre uma rotação normal e uma rotação com as especificações novas é verificado se existe o atributo “Time” no elemento Rotation.

Se esse atributo existir, vai guardá-lo num vetor de tempos de rotações, como também os atributos x, y e z iguais aos de uma rotação normal.

```
void getGlobalCatmullRomPoint(globalTime, pos, deriv) {

    n_points is the number of groups of 3 coordenates

    t is the real global t
    index is the segment
    t is now the position in the segment

    int indices[n_points];

    indices[0] is the first point index
    indices[1] is the second point index
    indices[2] is the third point index
    indices[3] is the fourth point index

    p0[3] is equal to the array in control points at index equal to the sum of the current point index and indices[0]
    p1[3] is equal to the array in control points at index equal to the sum of the current point index and indices[1]
    p2[3] is equal to the array in control points at index equal to the sum of the current point index and indices[2]
    p3[3] is equal to the array in control points at index equal to the sum of the current point index and indices[3]
```

Figura 11: Pseudocódigo da função "getGlobalCatmullRomPoint"

Logo, esta função vai-se basear na função que calcula a posição no instante com base em 4 pontos de controlo, “getCatmullRomCurve”. Cada um dos pontos é obtido a partir da seguinte formula e no pseudocódigo:

$$P(u,v) = [t^3 \quad t^2 \quad t \quad 1] \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

$$P'(u,v) = [3t^2 \quad 2t \quad 1 \quad 0] \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

```
void getCatmullRomPoint(t, p0, p1, p2, p3, pos, deriv) {

    float m[4][4] is the catmullRom matrix

    a[4] is a auxiliar vector in between operations
    tVector[4] is the t vetor
    tVectorL[4] is the t vector derivatives

    for (each coordenate component){
        ponto[4] is composed by the same component of each control point

        a is the resulting vector of the multiplication of the m matrix and vector ponto

        pos[i] is the multiplication of the tVector and the resulting vector a
        deriv[i] is the multiplication of the tVectorL and the resulting vector a
    }
}
```

Figura 12: Pseudocódigo da função "getCatmullRomPoint"

Obtendo as coordenadas do ponto de translação agora temos de verificar se é necessário alinhar o modelo desenhado à curva que percorre. Para realizar este processo é necessário calcular a matriz de rotação, realizar a rotação seguida da translação e finalmente atualizar o valor de y e de t.

```
case 7:
  n_points is the number of points in the current curve
  add to current_index the number of points of current curve

  if(align is true) {
        for (each component) {
            copy deriv's value toxi (only for better interpreting)
        }

        normalize x vector
        normalize y vector
        normalize y_prev vector
        normalize z vector

        calculate zi by crossproduct of x and previous y
        calculate y by crossproduct of x and z

        rotMatrix[4][4] will be the final rotation matrix

        assign to rotMatrix the rotation matrix

        translate according to x vector
        rotate according to rotMatrix

        for (each component) {
            previous y is current y
        }

        planetsPos[countCurves] is the current position of the current curve

        update t_prev_trans as the current
    }
  increment number of curves
```

Figura 13: Excerto de pseudócodigo da função "draw"

Para realizar esta transformação é primeiro necessário obter a matriz de rotação com base nos 3 vetores calculados através da função *"buildRotMatrix" de acordo com a seguinte matriz:*

$$M = \begin{bmatrix} X_x & Y_x & Z_x & 0 \\ X_y & Y_y & Z_y & 0 \\ X_z & Y_z & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotações com Duração

De modo a obter uma rotação especificando o tempo que demora a realizar uma rotação completa vai ser necessário identificar o valor correspondente da sua transformação (a equipa escolheu 8.0f).

Quando identificado ele vai calcular o angulo de rotação que têm de realizar em cada renderização e adicioná-lo ao angulo total de rotação.

```
case 8:
    planetsAngle[countRotate] is the current angle of the current planet

    update t_prev_rot as the current
    rotate along y axis with value

    break;
}
```

Figura 14: Excerto da função "draw"

Testes com os Ficheiros Providenciados

Teste_3_1:

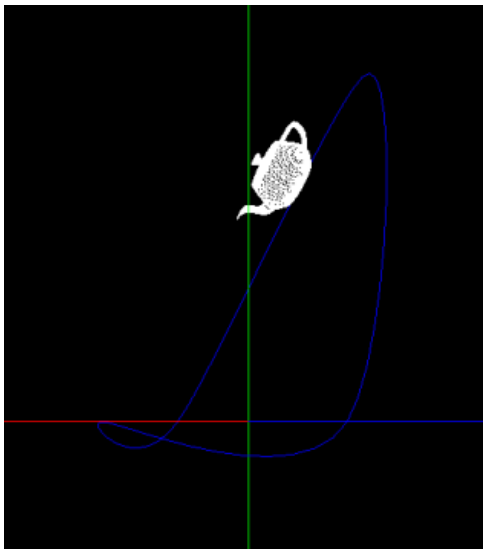

Figura 15: Resultado do Teste_3_1

Teste_3_2:

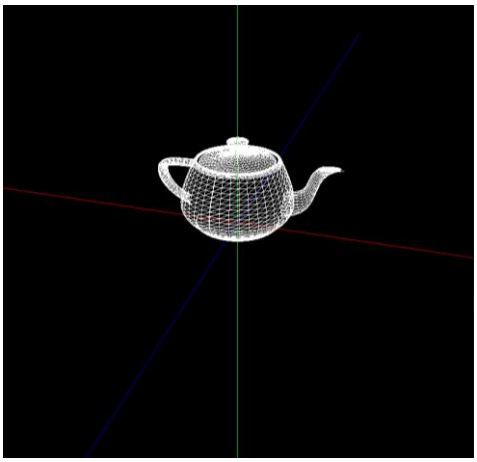

Figura 16: Resultado do Teste_3_2

Protótipo de Sistema Solar

Planetas

A demo requerida deveria ser capaz de implementar um sistema solar dinâmico, complementado com um cometa. Para tal, seria necessário alterar o ficheiro xml gerado na fase anterior, com algumas alterações. Primeiro, foi tomada a decisão de recentrar o sol no centro dos eixos. Segundo, em cada group dedicado a um planeta, seria necessário implementar uma curva.

Numa fase inicial, foram criadas curvas com 4 pontos de controlo, no entanto, logo após o primeiro teste foi notado que não bastaria, sendo necessário 8 pontos. Para conseguirmos as coordenadas desses 8 pontos foi elaborado um script em python :

```
raio = 736
pontos = 8

step = (2*math.pi)/pontos
x = 0
z = 0
i = 0
angulo = 0

while (i<pontos) :
    x = raio*math.cos(angulo) -raio
    z = raio*math.sin(angulo)
    angulo +=step
    i+=1
    print("<point ",i," x = \"" ,math.floor(x) ,"\" y = \"0\" ", " z=\"" , math.floor(z),"\" />")
```

Figura 17: Scrip em python para calculo dos pontos de controlo das órbitas

Este script seria capaz de gerar as coordenadas dos pontos já no formato necessário para acrescentar ao ficheiro xml. O valor de x e z seriam calculados com coordenadas polares, sendo que à coordenada x deveria ser sempre subtraído o raio da órbita, de forma centrar a órbita no sol. O raio da órbita foi calculado manualmente antes de cada execução do script python, somando as translações simples feitas até o momento do desenho desse planeta. A razão para ser feita essa subtração é o facto de, após as translações feitas no ficheiro até esse momento, o centro dos eixos deixou de estar na sua posição inicial , o Sol.

A nível do tempo de translação, à semelhança da fase anterior, o ponto de partida foi uma tentativa de manter o sistema o mais realista possível, pegando nos valores proporcionais de translação de cada planeta , vistos na tabela seguinte :

Planeta	Tempo de Translação
Mercúrio	0.241
Vénus	0.614
Terra	1
Marte	1.88
Júpiter	11.9
Saturno	29.4
Úrano	83.7
Neptuno	163.7

Multiplicamos cada um dos valores por 10 e declarar esses valores como os tempos de translação no ficheiro XML obtendo o seguinte:

Planeta	Tempo de Translação
Mercúrio	2.41
Vénus	6.14
Terra	10
Marte	18.8
Júpiter	119
Saturno	294
Úrano	837
Neptuno	1637

Assim, o código XML para um dado planeta teria o seguinte formato :

```
<group><!-- mercury-->
            <transform>
                <translate time = "2.4" align="True" > <!-- O campo align diz se o objecto deve ser orientado na curva -->
                    <point x = "0" y = "0" z = "0" /> x
                    <point x = "-80" y= "0" z ="195" />
                    <point x = "-276" y = "0" z = "276" /> x
                    <point x = "-471" y = "0" z="195 "/>
                    <point x = "-552" y = "0" z = "0" /> x
                    <point x = "-471" y = "0" z = "-195" />
                    <point x = "-276" y = "0" z = "-276" /> x
                    <point x = "-81" y = "0" z = "-195" />
                </translate>
                <scale x="0.33" y="0.33" z="0.33" />
                <rotate time="5" x="0" y="1" z="0" />
            </transform>
            <models>
                <model file="sphere.3d" /><!-- generator sphere 7.8 10 10 sphere.3d -->
            </models>
        </group>
```

Figura 18: Ficheiro xml exemplo de um dos planetas

Lua

A lua, na sua qualidade de satélite natural da Terra, necessitou de cuidado acrescido, sendo necessário lembrar que esta deveria acompanhar a Terra na sua translação, além de percorrer o seu percurso de translação habitual. Para tal, invés de ter a Lua no seu próprio group, separado da Terra, incorporámos o seu group no da Terra. Assim, após a inclusão do modelo sphere.3d relativo à Terra, é feita uma translação de valor de 10 ao longo do eixo x e criado um group para acomodar as transformações geométricas e o modelo sphere para a Lua.

Cometa

Depois dos planetas, seria necessário colocar um cometa no sistema solar, para tal, a primeira decisão a tomar seria a localização do cometa no sistema solar. Foi tomada a decisão de posicionar a ponta mais distante da órbita entre Júpiter e Saturno. De seguida foi necessário calcular a órbita. Para tal, foi tido como ponto de partida uma órbita com o mesmo formato das calculadas anteriormente. Depois, foi feito um esforço para descentrar essa mesma órbita tendo em atenção evitar colisões com o Sol e alterando os valores de z para tornar a órbita mais próxima de uma elipse. Por fim foram calculados um total de 12 pontos de controlo para a orbita do cometa.

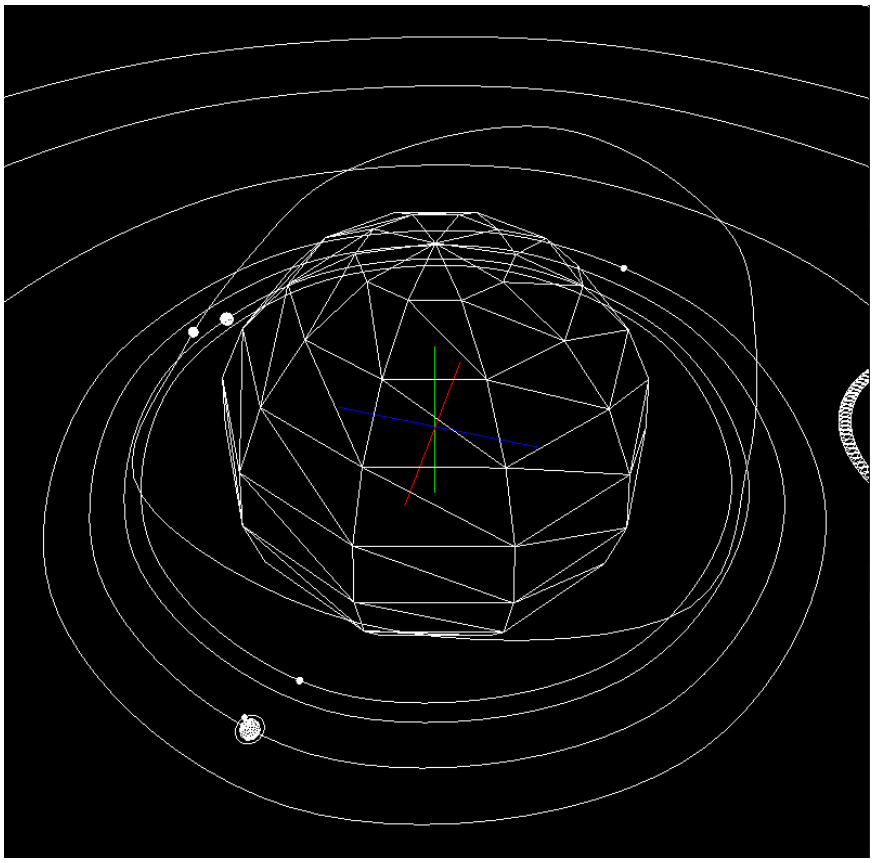

Figura 19: Resultado final .1

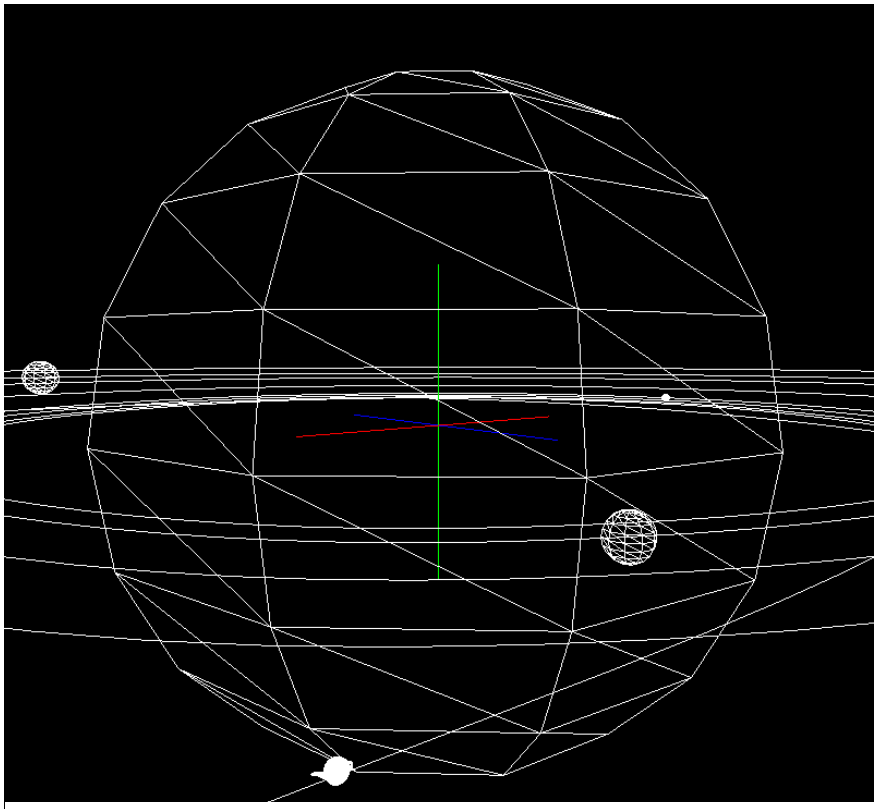

Figura 20: Resultado final .2

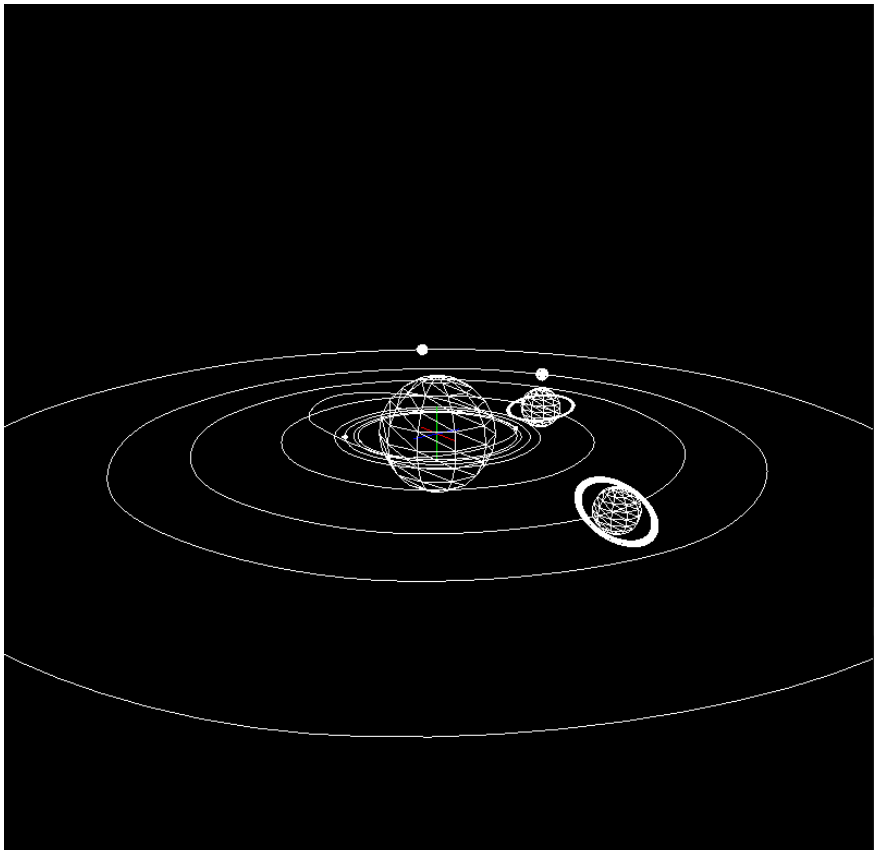

Figura 21: Resultado final .3

Funcionalidade Extra

Para propósitos de demonstração e melhor visibilidade das translações completas dos planetas mais distantes, foi acrescentada a possibilidade de controlar as velocidades de translação. Essa funcionalidade foi implementando ao associar 2 teclas do teclado, a tecla L e R, sendo que estas alteram o valor de uma variável global denominada por time_multiplier. A variável time_multiplier é utilizada no cálculo da posição do planeta.

Por último, foi ainda adicionada a capacidade de mostrar ou esconder as curvas das órbitas, premindo a tecla S.

```
void processKeys(unsigned char c, int xx, int yy) {

    switch(c){
        case 'l':
            time_multiplier is multiplied by 2
            break;
        case 'r':
            time_multiplier is divided by 2
            break;
        case 's':
            if showCurve is false, showCurve becomes true
            else{
                showCurve becomes true
            }
    }
}
```

Figura 22: Pseudocódigo da função "processKeys"

Conclusão

Terminada a segunda fase deste trabalho, foram atingidos todos os objetivos propostos pelos docentes. Os resultados obtidos estão maioritariamente em concordância com os resultados indicados no ficheiro de testes fornecido aos alunos e o código apresentado é legível e compreensível com o auxílio do pseudocódigo. Além disso, encontramo-nos bastante satisfeitos com o avanço no desenvolvimento do sistema solar e ansiosos para os progressos que se realizarão nas próximas fases.