

Computação Gráfica
Trabalho Prático
Fase 2 - Transformações Geométricas

Luís Almeida
A84180

João Pedro Antunes
A86813

Fernando Lobo
A87988

Diogo Monteiro
A71452

4 de abril de 2021

Resumo

O presente documento descreve a resolução do enunciado adotada pelo grupo. Nesta fase do trabalho prático foi nos proposta a implementação de transformações geométricas na nossa aplicação engine. Para este efeito alteramos a estrutura dos nossos ficheiros XML, dando-nos assim a possibilidade de especificar quais as transformações geométricas a aplicar aos modelos de forma a criar cenas mais complexas.

Conteúdo

1	Introdução	3
2	Análise e Especificação	4
2.1	<i>Parsing XML</i>	4
2.1.1	Classe <i>Group</i>	5
2.1.2	Classe <i>Transform</i>	5
2.1.3	Classe <i>Translate</i>	5
2.1.4	Classe <i>Rotate</i>	5
2.1.5	Classe <i>Scale</i>	5
2.1.6	Classe <i>Scene</i>	6
3	Concepção/desenho da Resolução	7
3.1	Classe <i>Transform</i>	7
3.2	Classe <i>Scene</i>	8
3.3	<i>Parsing XML</i>	9
4	Extras	11
4.1	<i>Settings</i>	11
4.1.1	Câmera	11
4.1.2	<i>Background</i>	11
4.2	<i>XML Extended Parsing</i>	12
4.2.1	<i>Evaluate</i> de expressões	12
4.2.2	Atribuição de Variáveis	12
	set	12
	pi	13
	cos,sin,tan,sqrt	13
	rand	13
	Alguns exemplos de atribuições	13
	Implementação	13
4.2.3	<i>Flow</i> operators	14
	If..Then..Else	14
	While	14
	Alguns exemplos de <i>Flow</i>	14
	Implementação dos operadores	14
4.3	<i>Shape Gen</i> extras	15

4.3.1	<i>Torus</i>	15
4.4	Testes realizados e Resultados	16
5	Conclusão	21

Capítulo 1

Introdução

Nesta fase do trabalho prático é necessária a realização de transformações geométricas por parte do engine a modelos existentes. Começamos por descrever o processo de *parsing* do ficheiro *XML* onde vão estar especificadas (além dos ficheiros que contêm os modelos) as transformações geométricas a aplicar a cada um deles. Especificamos também as alterações feitas ao processo de *render* da cena especificada, que agora terá de lidar com múltiplos grupos e transformações geométricas. No fim apresentamos os extras que decidimos incorporar nesta fase do trabalho prático.

Capítulo 2

Análise e Especificação

2.1 *Parsing* XML

Os ficheiros XML vão agora apresentar uma estrutura hierárquica entre grupos de modelos. Estas hierarquias vão-nos permitir aplicar diferentes transformações geométricas consoante o nível do grupo.

Por exemplo, temos o seguinte ficheiro XML que aplica uma translação e uma rotação a uma esfera:

```
<scene>
  <group>
    <translate X=5 Y=0 Z=2 />
    <rotate angle=45 axisX=0 axisY=1 axisZ=0 />
    <models>
      <model file="sphere.3d" />
    </models>
  </group>
</scene>
```

Figura 2.1: Exemplo de transformações aplicadas a um grupo

No entanto, podemos ter um nível filho de um certo grupo, que vai "herdar" as transformações geométricas aplicadas ao pai, assim como terá transformações geométricas que serão aplicadas a ele e aos seus filhos dos níveis inferiores na hierarquia. De seguida apresentamos uma situação onde é aplicada uma translação a um grupo "pai" e uma outra translação a um grupo "filho", que ilustra esta ideia enunciada:

```

<scene>
  <group>
    <translate X=1 />
    <models>
      <model file="sphere.3d" />
    </models>
    <group>
      <translate Y=1 />
      <models>
        <model file="cone.3d" />
      </models>
    </group>
  </group>
</scene>

```

Figura 2.2: Exemplo de hierarquia entre grupos com diferentes transformações geométricas

Ora, para extrairmos a informação que nos é relevante, podemos pensar em iterar o ficheiro XML "ativando" diferentes níveis de profundidade. Assim, ser-nos-á possível manter esta hierarquia entre os diferentes grupos.

2.1.1 Classe *Group*

Para armazenarmos a informação relativa a um determinado grupo criamos a classe *Group*. Os atributos relevantes de um grupo são as transformações geométricas que lhe vão ser aplicadas, os modelos que estão presentes no mesmo e os filhos do grupo na hierarquia.

2.1.2 Classe *Transform*

Esta será uma classe abstrata que define uma transformação geométrica. Todas elas terão valores para X, Y, Z , pelo que podemos incluir um vetor que armazene estes valores na definição da classe *Transform*. Usar esta classe também nos permite armazenar na mesma estrutura de dados diferentes transformações geométricas.

2.1.3 Classe *Translate*

A classe *Translate* irá implementar a classe abstrata *Transform*, armazenando o vetor de translação.

2.1.4 Classe *Rotate*

A classe *Rotate* também implementa a classe *Transform*, no entanto, além dos valores X, Y e Z que são as coordenadas do eixo de rotação, é preciso indicar também o ângulo da rotação.

2.1.5 Classe *Scale*

Por fim, a classe *Scale* é a última classe que implementa *Transform*. Armazenamos o vetor de escala.

2.1.6 Classe *Scene*

Para armazenarmos as informações relativas à cena que está a ser desenhada criamos uma classe *Scene* que englobará todos os grupos, modelos, câmara e modo de desenho da cena. Também nesta classe vamos ter presente o número de triângulos que desenharam a cena, bem como um método que calcula o número de FPS.

Tendo estas classes definidas, podemos implementar o processo de *parsing* definido anteriormente de uma forma simples. Basta criarmos uma função que itere um ficheiro *XML*, e sempre que encontre um grupo chama uma função auxiliar *parseGroup*. Esta função por sua vez chama as funções auxiliares *parseTranslate*, *parseRotate*, *parseScale* que fazem o *parsing* das transformações geométricas relativas ao grupo. Chama também a função *parseModels* para fazer o *parsing* dos nomes dos ficheiros que contêm a informação relativas aos modelos do grupo, assim como se chama a si própria recursivamente caso encontre um filho.

Capítulo 3

Concepção/desenho da Resolução

3.1 Classe *Transform*

Nesta secção vamos apresentar a definição da classe abstrata que representa uma transformação geométrica bem como as definições das classes que representam cada uma das diferentes transformações geométricas.

```
class Transform{
    public:
        std::vector<float> axis;

        Transform(std::vector<float> vals){
            this->axis.assign(vals.begin(),vals.end());
        }

        virtual void applyTransform() = 0;
};

class Rotate : public Transform{
    public:
        float degrees;

        Rotate(std::vector<float> vals, float angle):Transform(vals){
            this->degrees = angle;
        }

        void applyTransform(){
            glRotatef(degrees,axis[0],axis[1],axis[2]);
        }
};

class Translate : public Transform{
    public:
        Translate(std::vector<float> vals):Transform(vals){
        }

        void applyTransform(){
            glTranslatef(axis[0],axis[1],axis[2]);
        }
};

class Scale : public Transform{
    public:
        Scale(std::vector<float> vals):Transform(vals){
        }

        void applyTransform(){
            glScalef(axis[0],axis[1],axis[2]);
        }
};
```

Note-se que o método *applyTransform* é um método *virtual* pois invoca a função que efetivamente realiza a transformação geométrica indicada, pelo que não pode ser generalizado.

3.2 Classe *Scene*

A nossa classe *Scene* começa por implementar as ideias referidas no capítulo anterior. O método que irá desenhar a cena é *drawGroups*, que itera o *vector* de grupos da cena e os desenha.

```
class Scene{
    public:

        int lastRefresh = glutGet(GLUT_ELAPSED_TIME);
        float frames = 0;

        // camera Object
        Camera cam;

        // Models
        std::vector<Group>*groups = new std::vector<Group>() ;
        std::map<std::string,Model*>*modelTable = new std::map<std::string,Model
            *>();

        // - Engine Runtime Options
        // Mode of polygon (fill / line)
        int polyMode = GL_FILL;
        // Origin Axis
        int oAxisDr = 100;
        // cameraCenter Axis
        int ccAxisDr = 0;

        //- Engine Starter Options
        std::vector<float> background{0.2f,0.2f,0.3f,0.2f};

        void drawGroups(){
            for (Group g: (*groups)){
                g.makeGroup();
            }
        }

        float getFPS(){
            float fps = -1;
            frames ++;
            int time = glutGet(GLUT_ELAPSED_TIME);
            if ((time-lastRefresh)>1000){
                fps = ((frames*1000)/(time-lastRefresh));
                lastRefresh = time;
                frames = 0;
            }
            return fps;
        }
    }
```

O método *drawGroups* chama o método *makeGroup* para cada um dos grupos que pertencem à cena. Este último método serve para tirarmos partido das funções *pushMatrix* e *popMatrix* do OpenGL. Começamos por fazer um *pushMatrix*, aplicamos as transformações relativas ao grupo usando o método *applyTransforms*, percorremos o *vector* de grupos "filhos" do grupo atual invocando recursivamente o método *makeGroup* (tirando partido de ainda termos aplicadas as transformações geométricas do grupo "pai"). No fim deste processo simplesmente fazemos um *popMatrix* para voltarmos ao sistema de coordenadas com que o método foi chamado.

```

void makeGroup(){
    glPushMatrix();

    applyTransforms();
    for (std::pair<std::vector<float>,Model*> m: models){
        glColor3f(m.first[0],m.first[1],m.first[2]);
        if (flagTRI) m.second->drawT();
        else m.second->drawVBO();
    }

    for (Group grp: child){
        grp.makeGroup();
    }

    glPopMatrix();
}

void applyTransforms(){
    for (Transform*t: trans){
        t->applyTransform();
    }
}

```

3.3 Parsing XML

Ora, começamos então a definir a função *parseFromTag* que vai estar responsável por este processo. Passamos como argumento o *vector* de grupos da cena com que estamos a trabalhar para que os grupos do ficheiro *XML* sejam armazenados nesta.

```

void parseFromTag(std::vector<Group>*gps, XMLElement* base){
    if (base){
        XMLElement* iterator;
        for(iterator = base->FirstChildElement(); iterator != NULL
            ; iterator = iterator->NextSiblingElement()){
            std::string tagName(iterator->Value());
            if (!tagName.compare("group")){
                parseGroup(gps, iterator);
            }
        }
    }
}

```

A função *parseGroup* adota o processo descrito anteriormente, armazenando o grupo que está a ser lido no *vector* de grupos "filho" do grupo recebido como argumento.

```

void parseGroup(std::vector<Group>*gps, XMLElement* base){
    Group g;
    XMLElement* iterator;
    for(iterator = base->FirstChildElement(); iterator != NULL;
        iterator = iterator->NextSiblingElement()){
        std::string tagName(iterator->Value());
        if (!tagName.compare("translate")){
            parseTranslate(g, iterator);
        }

        else if (!tagName.compare("rotate")){
            parseRotate(g, iterator);
        }

        else if (!tagName.compare("scale")){
            parseScale(g, iterator);
        }
    }
}

```

```

    }

    else if (!tagName.compare("models")){
        parseModel(g, iterator);
    }

    else if (!tagName.compare("group")){
        parseGroup(&(g.child), iterator);
    }

}

(*gps).push_back(g);
}

```

As funções *parseTranslate*, *parseRotate*, *parseScale* são simples e adotam o processo descrito no capítulo anterior.

```

void parseTranslate(Group &parent, XMLElement* base){
    std::vector<float> v;
    v.push_back(getParsAtt(base, "X"));
    v.push_back(getParsAtt(base, "Y"));
    v.push_back(getParsAtt(base, "Z"));
    parent.trans.push_back(new Translate(v));
}

void parseRotate(Group &parent, XMLElement* base){
    std::vector<float> v;
    v.push_back(getParsAtt(base, "axisX"));
    v.push_back(getParsAtt(base, "axisY"));
    v.push_back(getParsAtt(base, "axisZ"));
    parent.trans.push_back(new Rotate(v, getParsAtt(base, "angle")));
}

void parseScale(Group &parent, XMLElement* base){
    std::vector<float> v;
    v.push_back(getParsAtt(base, "X", 1));
    v.push_back(getParsAtt(base, "Y", 1));
    v.push_back(getParsAtt(base, "Z", 1));
    parent.trans.push_back(new Scale(v));
}

```

A função *parseModel* tenta procurar o modelo do grupo que está a ser lido na *modelTable* da cena, e se este não existir adiciona-o. Por fim, adiciona o endereço de memória do modelo à lista de modelos do grupo. Este método adicionará também o número de triângulos que constituem um determinado modelo ao número total de triângulos da cena.

```

void parseModel(Group &parent, XMLElement* base){
    XMLElement* iterator;
    for(iterator = base->FirstChildElement(); iterator != NULL;
        iterator = iterator->NextSiblingElement()){
        std::string fileName(iterator->Attribute("file"));

        if ( (*(currentScene->modelTable)).find(fileName) == (*(
            currentScene->modelTable)).end()){
            (*(currentScene->modelTable))[fileName] = new
                Model(fileName);
        }
        tris += (int)((*(currentScene->modelTable))[fileName])->
            facesList.size();
        parent.models.push_back( std::pair<std::vector<float>,
            Model*>(color, (*(currentScene->modelTable))[fileName])
        );
    }
}

```

Capítulo 4

Extras

4.1 *Settings*

Com a intenção de tornar mais acessível a manipulação da *scene*, recorreu-se à definição de uma nova *label*, "settings", que possui as preferências relativas à posição de câmera e cor de *background*.

4.1.1 Câmera

A presença de uma *label* "camera", permite que seja possível posicionar a câmera na *scene*, com as coordenadas e a orientação pretendidas. Para isso é lido a posição do centro da câmera (*label* "center"), que contém três atributos, os quais, correspondem às coordenadas do ponto para o qual a câmera está orientada e é ainda obtida a posição, definida por coordenadas polares, da câmera, relativamente ao centro, como atributos da *label* "position".

```
<camera>
  <center X="0" Y="0" Z="0"/>
  <position A="3*pi/4" B="pi/6" R="400"/>
</camera>
```

Figura 4.1: Definições da câmera

4.1.2 *Background*

Quanto ao *background*, são retirados os valores de *red*, *green*, *blue* e *alpha*, que serão depois usados para definir a cor do *background* da *scene*.

```
<background R="0.1" G="0.1" B="0.1" A="0.7"/>
```

Figura 4.2: Definições de *background*

4.2 XML Extended Parsing

Juntamente com o parser básico requisitado para realizar transformações em arquivos da *Engine*, o grupo dedicou tempo especial para desenvolver extensões ao arquivo XML requisitado a fim de facilitar a criação de cenas para demonstração da engine. As extensões desenvolvidas tratam-se de comandos semelhantes a uma linguagem de programação, tais como *loops*, *If's*, etc. Os comandos são interpretados na leitura do XML, aplicando as devidas alterações.

4.2.1 Evaluate de expressões

Para permitirmos o uso de variáveis e cálculos mais complexos no arquivo, foi tomada a decisão de que todas as expressões associadas a valores no XML passariam por um processo avaliação interno, nos quais as variáveis seriam substituídas por suas devidas correspondências numéricas.

Assim, foi criada uma classe a ser colocada no parser base de XML, esta classe faria a avaliação de expressões com um *Hash Table* interna para variáveis, e com um método de tratamento de strings que dada uma expressão retornaria seu respectivo valor numérico em *float*.

Para a realização do parsing, a classe de avaliação baseia-se, basicamente, em diversos métodos semelhantes ao seguinte:

```
float getNextExpr(){
    float res = getNextTerm();
    char check = peekChr();
    while (check == '-' || check == '+'){
        if(popChr() == '+'){
            res += getNextTerm();
        }
        else{
            res -= getNextTerm();
        }
        check = peekChr();
    }
    return res;
}
```

Que realizam o parsing da expressão em diferentes "camadas", até atingirem um valor atômico ou da *Hash Table*. A avaliação de expressões substituirá, portanto, o método de "*FloatAttribute*" do *tinyXML2*, passaremos a ler os atributos como strings e avaliá-las internamente com o objeto apresentado. No pior dos casos, estaremos avaliando um único valor de float como uma expressão. No caso da avaliação de um erro a expressão será avaliada como 0, para não prejudicar a avaliação restante do arquivo.

4.2.2 Atribuição de Variáveis

Para a utilização de variáveis durante uma especificação XML, o utilizador poderá utilizar-se de variáveis do tipo *float* atribuídas durante o ficheiro XML. As variáveis podem ser atribuídas através de 7 *tags* diferentes, *set*, *pi*, *rand*, *cos*, *sin*, *tan*, *sqr*. Sendo todas avaliadas a partir do atributo "*tgt*" como a variável a ser atribuída e, nas que se aplicam, "*exp*" como a expressão a ser calculada para a atribuição.

set

A *tag* mais simples, representa uma simples atribuição da expressão "*exp*" na variável "*tgt*", equivalente, em uma linguagem comum de programação a `tgt = exp`.

pi

Representa a atribuição do valor de π à variável "tgt", correspondente a `tgt = M_PI` em C++.

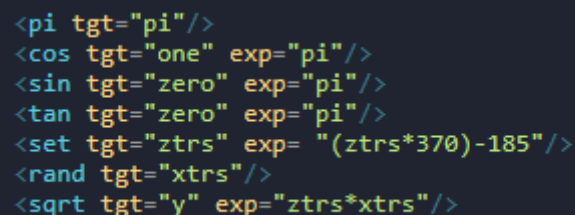
cos,sin,tan,sqrt

Estas *tags* possuem todas um funcionamento semelhante, nestas, não é feita uma atribuição simples de uma expressão a uma variável, mas é antes realizado um cálculo sobre a expressão passada, o cálculo realizado é o correspondente ao nome da tag, i.e., `tgt = cos(exp)`.

rand

Tag que atribui a uma variável um valor *Float* aleatório, normalizado (entre 1 e 0).

Alguns exemplos de atribuições



```
<pi tgt="pi"/>
<cos tgt="one" exp="pi"/>
<sin tgt="zero" exp="pi"/>
<tan tgt="zero" exp="pi"/>
<set tgt="ztrs" exp=" (ztrs*370)-185"/>
<rand tgt="xtrs"/>
<sqrt tgt="y" exp="ztrs*xtrs"/>
```

Figura 4.3: Exemplo atribuições no XML

Implementação

A seguir um excerto de código utilizado nestas operações, demonstrando o uso da tabela de hash e da função de avaliação mencionada anteriormente.

```
void setParsAtt(XMLElement*base, int opt){
    float evaluation;

    const char* target = base->Attribute("tgt");
    if (!target || !target[0]) return;
    std::string varTgt(target);

    const char* value = base->Attribute("exp");
    if (!(value)) value = "";

    pars.setExpr(value);
    evaluation = pars.evalOpts(opt);
    if (pars.error) printf("Warning! Something went wrong parsing XML
        !\n");

    pars.hash[varTgt] = evaluation;
}
```

4.2.3 Flow operators

If..Then..Else

Tags utilizadas para simular o comportamento de um condicional na criação da cena descrita no XML. Inicialmente cria-se a *tag* relativa à guarda do condicional, nela é especificada uma expressão com o atributo "*exp*" que será avaliado para a obtenção da condição.

Internamente, deverão estar especificados dois conjuntos de operação marcados com *Then* ou *Else*, no caso da não existência de uma destas tags, por omissão considera-se uma ação nula, como em muitas outras linguagens. Para emular o comportamento, o parser verifica a avaliação da expressão, e em seguida chama-se recursivamente na tag correspondente ao resultado da condição.

While

Simula o comportamento de um *while loop*, possui avaliação semelhante à *tag "if"*, tomando em consideração uma expressão descrita no atributo "*exp*". Para emular este comportamento, o parser de XML é chamado recursivamente na tag até que a condição torne-se falsa.

Alguns exemplos de Flow

```
<if exp= "dist > (radiusTree*radiusTree)">
  <then>
    <!-- Do something -->
  </then>
  <else>
    <!-- Do other things -->
  </else>
</if>

<while exp="var > 0">
  <!-- loop things -->
</while>
```

Figura 4.4: Exemplos no XML

Implementação dos operadores

A seguir um trecho onde é possível notar a avaliação e controle de fluxo descrito pelos operadores demonstrados acima, seguindo as especificações de aplicação também referidas.

```
void parseFlowStm(std::vector<Group>*gps, XMLElement*base, std::string
tagName){
    float cond = getParsAtt(base,"exp");

    if (!tagName.compare("if")){
        XMLElement*iterator;
        if (cond){
            for(iterator = base->FirstChildElement(); iterator
                != NULL && strcmp(iterator->Value(),"then");
                iterator = iterator->NextSiblingElement());
        }
        else{
            for(iterator = base->FirstChildElement(); iterator
                != NULL && strcmp(iterator->Value(),"else");
                iterator = iterator->NextSiblingElement());
        }
    }
}
```



```

        }
        parseFromTag(gps, iterator);
    }

    if (!tagName.compare("while")){
        while (cond){
            parseFromTag(gps, base);
            cond = getParsAtt(base, "exp");
        }
    }
}

```

4.3 *Shape Gen extras*

Nesta fase optou-se também pela criação de um *shape*, *Torus*, diferente dos já sugeridos.

4.3.1 *Torus*

A geração deste sólido é bastante semelhante à dos já fornecidos, no entanto, a única diferença está na definição das coordenadas de cada vértice. Tendo em conta o número de *slices* é possível, através de um ângulo, α , contido no plano *XZ*, determinar a orientação de um vetor, contido também nesse plano. Com isto e juntamente com as *stacks* e um ângulo β (definido através de dois vetores, um com a mesma orientação que o vetor gerado anteriormente e um vetor perpendicular ao eixo do *Y*, ambos normalizados) é possível calcular as coordenadas dos vértices correspondente a cada *stack* e a cada *slice*.

```

void shape(){
    float alpha = (2*M_PI)/slices;
    float beta = (2*M_PI)/(stacks-1);
    int vtxPts = 1;

    std::vector<int> row;

    for (int st = 0; st < stacks+1; st++){
        row.clear();
        for (int sl = 0; sl < slices+1; sl++){
            if (vtxPts < this->vex){
                std::vector<float> aux;
                aux.push_back(extr*(sin(alpha*sl)) + intr*sin(alpha*sl)*(cos(beta*st)));
                aux.push_back(intr*(sin(beta*st)));
                aux.push_back(extr*cos(alpha*sl) + intr*cos(alpha*sl)*cos(beta*st));

                (this->vertex).push_back(aux);
                row.push_back(vtxPts++);
            }

            else{
                row.push_back((vtxPts % this->vex)+1);
                vtxPts++;
            }
        }
        (this->planar).push_back(row);
    }
}

```

4.4 Testes realizados e Resultados

De seguida apresentamos o ficheiro *XML* que gera um sistema solar especificado por grupos hierárquicos, assim como a cena gerada em *openGL*:

```
<scene>
  <pi tgt="pi"/>

  <!-- settings -->

  <settings>
    <camera>
      <center X="116.5" Y="0" Z="0"/>
      <position A="pi/2" B="0" R="100"/>
    </camera>
    <background R="0.1" G="0.1" B="0.1" A="0.7"/>
  </settings>

  <set tgt="merRot" exp="130"/>
  <set tgt="venRot" exp="40"/>
  <set tgt="earRot" exp="15"/>
  <set tgt="marRot" exp="86"/>
  <set tgt="jupRot" exp="234"/>
  <set tgt="satRot" exp="93"/>
  <set tgt="urnRot" exp="69"/>
  <set tgt="nepRot" exp="420"/>
  <set tgt="pluRot" exp="42"/>

  <group>

    <!-- sun -->
    <group>
      <scale X="0.6" Y="0.6" Z="0.6" />
      <models>
        <model file="sphere.3d" R="1.0" G="1.0" B="0.0" />
      </models>

      <!-- Mercury -->
      <group>
        <rotate angle="merRot" axisY="1"/>
        <translate X="28.05"/>
        <scale X="0.03505" Y="0.03505" Z="0.03505" />
        <models>
          <model file="sphere.3d" R="0.86" G="0.86" B="0.86" />
        </models>
      </group>

      <!-- Venus -->
      <group>
        <rotate angle="venRot" axisY="1"/>
        <translate X="52.13"/>
        <scale X="0.087" Y="0.087" Z="0.087" />
        <models>
          <model file="sphere.3d" R="1.0" G="0.64" B="0.0" />
        </models>
      </group>

      <!-- Earth -->
      <group>
        <rotate angle="earRot" axisY="1"/>
        <translate X="71.94"/>
        <scale X="0.091" Y="0.091" Z="0.091" />
```

```

<models>
  <model file="sphere.3d" R="0.12" G="0.56" B="1.0"
    />
</models>
<group>
  <rand tgt="degree"/>
  <set tgt="rot" exp="degree*360" />
  <rotate angle="rot" axisY="1" axisX="1" axisZ="0"
    />
  <translate X="60.34"/>
  <scale X="0.27" Y="0.27" Z="0.27" />
  <models>
    <model file="sphere.3d" R="0.86" G="0.86"
      B="0.86" />
  </models>
</group>
</group>

<!-- Mars -->
<group>
  <rotate angle="marRot" axisY="1"/>
  <translate X="116.5"/>
  <scale X="0.049" Y="0.049" Z="0.049" />
  <models>
    <model file="sphere.3d" R="1.0" G="0.12" B="0.0"
      />
  </models>
  <group>
    <rand tgt="deg"/>
    <set tgt="rot" exp="deg*360" />
    <rotate angle="rot" axisY="1" axisX="1" axisZ="0"
      />
    <translate X="70.34"/>
    <scale X="0.2" Y="0.2" Z="0.2" />
    <models>
      <model file="sphere.3d" R="0.86" G="0.86"
        B="0.86" />
    </models>
  </group>
  <group>
    <rand tgt="degs"/>
    <set tgt="rotat" exp="degs*360" />
    <rotate angle="rotat" axisY="1" axisX="1" axisZ="0"
      " />
    <translate X="60.34"/>
    <scale X="0.27" Y="0.27" Z="0.27" />
    <models>
      <model file="sphere.3d" R="0.86" G="0.86"
        B="0.86" />
    </models>
  </group>
</group>

<!-- Inner Ring -->
<group>
  <set tgt="radiusRocks" exp="150" />
  <set tgt="radiusRocksMax" exp="160" />
  <set tgt="rockNum" exp="700" />

  <while exp="rockNum">0">
    <rand tgt="ang"/>
    <rand tgt="xtrs"/>

```

```

<set tgt="xtrs" exp= "(xtrs*radiusRocksMax)+
radiusRocks"/>

<set tgt= "rockNum" exp="rockNum-1" />
<group>
  <rotate angle="ang*360" axisY="1"/>
  <translate X="xtrs"/>
  <set tgt="scl" exp="0.5"/>
  <scale X="scl" Y="scl" Z="scl" />
  <set tgt="minRads" exp="5"/>
  <rand tgt="zscl"/>
  <group>
    <scale X="scl" Y="scl" Z="scl" />
    <models>
      <model file="asteroid.3d"
        R="0.4" G="0.4" B="0.4"
        />
    </models>
  </group>
</group>
</while>
</group>

<!-- Jupiter -->
<group>
  <rotate angle="jupRot" axisY="1"/>
  <set tgt="moonNum" exp="9" />
  <translate X="373"/>
  <scale X="0.2" Y="0.2" Z="0.2" />
  <models>
    <model file="sphere.3d" R="0.74" G="0.56" B="0.56"
    />
  </models>
  <group>
    <set tgt="radiusRocks" exp="50" />
    <set tgt="rockNum" exp="9" />
    <while exp="rockNum<0">
      <rand tgt="ztrs"/>
      <rand tgt="xtrs"/>
      <set tgt="ztrs" exp= "(ztrs*370)-185"/>
      <set tgt="xtrs" exp= "(xtrs*370)-185"/>
      <set tgt="dist" exp="(xtrs*xtrs)+_+(ztrs*
        ztrs)" />
      <set tgt="dist" exp="dist*_6" />
      <if exp= "dist<_+(radiusRocks*radiusRocks)
        ">
        <then>
          <if exp= "dist<_+((
            radiusRocks*
            radiusRocks)*1.2)">
            <then>
              <set tgt= "rockNum" exp="rockNum-1" />
            </then>
          </if>
        </then>
      </if>
    </while>
    <group>
      <rand tgt="degs"/>
      <set tgt="rotat" exp="degs*360" />
    </group>
    <rotate angle="rotat" axisY="1" axisX="1" axisZ="0" />
    <translate X="xtrs*1.5" Z="ztrs"/>
    <set tgt="scl" exp="0.08"/>
    <scale X="scl*1.3" Y="scl*1.3" Z="scl*1.3" />
    <set tgt="minRads" exp="5"/>
    <group>
      <rand tgt="zscl"/>
    </group>
  </group>
</group>

```

```

                                <scale X="0.2" Y=
                                    "0.2" Z="0.2"
                                />
                                <models>
<model file="sphere.3d" R="0.4" G="0.4" B="0.4"/>
                                </models>
                                </group>
                                </group>
                                </then>
                                </if>
</then>
</if>
</while>
</group>
</group>

<!-- Saturn -->
<group>
    <rotate angle="satRot" axisY="1"/>
    <translate X="532"/>
    <scale X="0.1836" Y="0.1836" Z="0.1836" />
    <models>
        <model file="sphere.3d" R="0.74" G="0.73" B="0.42"
            />
    </models>
    <group>
        <rotate angle="90*xtrs" axisY="0" axisX="1" axisZ=
            "0" />
        <scale X="3.5" Y="0.1" Z="3.5" />
        <models>
            <model file="torus.3d" R="0.96" G="0.87" B
                ="0.7" />
        </models>
    </group>
</group>

<!-- Uranus -->
<group>
    <rotate angle="urnRot" axisY="1"/>
    <translate X="687.33"/>
    <scale X="0.1365" Y="0.1365" Z="0.1365" />
    <models>
        <model file="sphere.3d" R="0.0" G="0.36" B="1.0"
            />
    </models>
    <group>
        <rotate angle="180*xtrs" axisY="0" axisX="1" axisZ
            ="0" />
        <scale X="3.0" Y="0.1" Z="3.0" />
        <models>
            <model file="torus.3d" R="0.69" G="0.88" B
                ="0.9" />
        </models>
    </group>
</group>

<!-- Neptune -->
<group>
    <rotate angle="nepRot" axisY="1"/>
    <translate X="776.03"/>
    <scale X="0.1354" Y="0.1354" Z="0.1354" />
    <models>
        <model file="sphere.3d" R="0.0" G="0.0" B="0.8" />
    </models>

```

```

</group>

<!-- Pluto -->
<group>
  <rotate angle="pluRot" axisY="1"/>
  <translate X="915.19"/>
  <scale X="0.017" Y="0.017" Z="0.017" />
  <models>
    <model file="sphere.3d" R="0.8" G="0.8" B="0.8" />
  </models>
</group>
</group>
</group>
</scene>

```

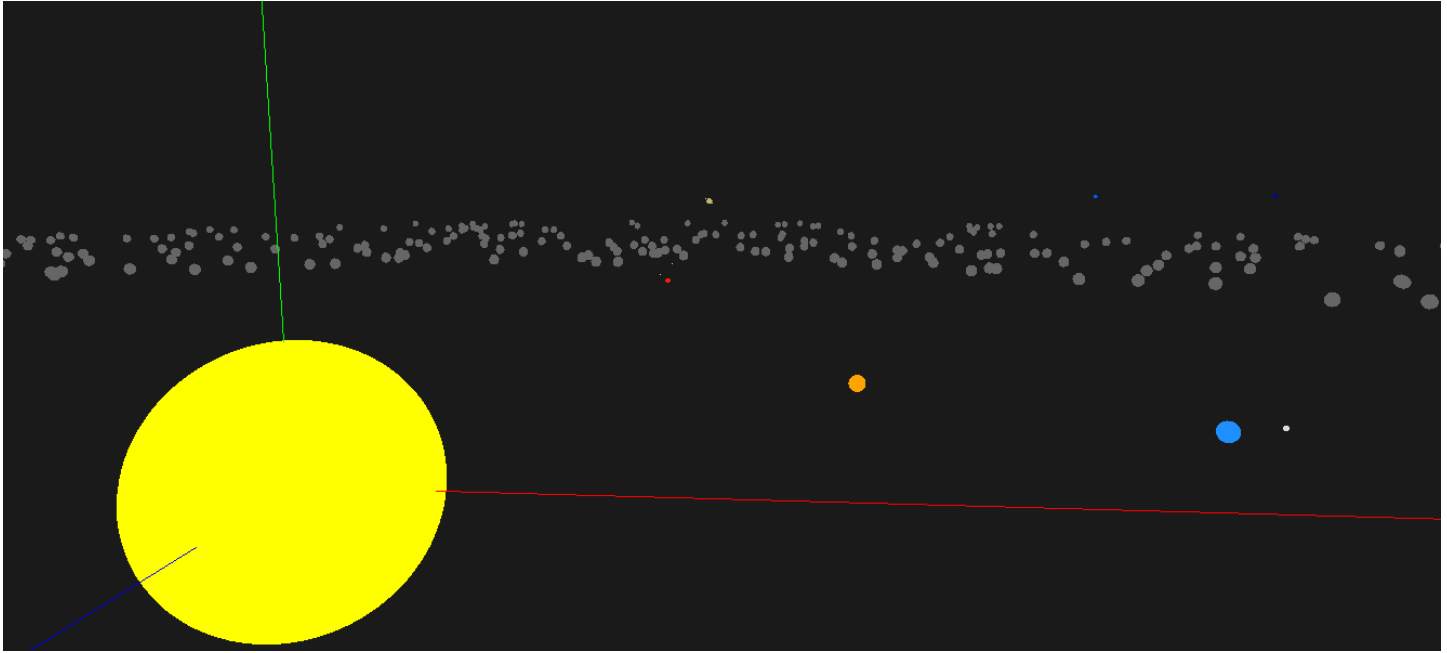


Figura 4.5: Cena gerada pelo XML, sistema solar com cintura de asteróides

Capítulo 5

Conclusão

Foram cumpridos todos os objetivos desta fase, tanto a extensão do ficheiro *XML*, como a introdução de transformações geométricas na aplicação *engine* e a cena *demo* do sistema solar escrita em grupos hierárquicos. Consideramos também que o projeto é facilmente extensível para novas etiquetas *XML*, como por exemplo transformações geométricas personalizadas, bastando para isto criar uma nova função de *parsing* para a nova etiqueta, e a correspondente transformação geométrica na *engine*. Os extras adicionados ao *parsing* do *XML* permitem-nos também especificar, com bastante simplicidade, cenas complexas.