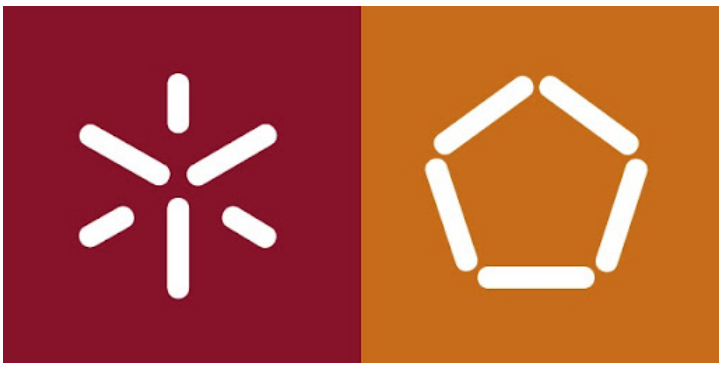


Universidade do Minho

Unidade Curricular:

Computação Gráfica

Relatório Fase 2 - Projeto Prático

Elementos do grupo:

A93253: David Alexandre Ferreira Duarte
A89518: Ema Isabel Quintãos Dias
A94166: Samuel de Almeida Simões Lira

Introdução

Para a fase 2 do projeto foi solicitado uma continuação do *engine* do programa. No caso, pretende-se que os ficheiros *XML* lidos contenham grupos onde são discriminadas as transformações associadas aos ficheiros discriminados nos *models*. Existem três tipos de transformações que devem ser consideradas: translação, rotação e escala. Além disso, é referido, pela equipa docente, que cada *group* pode conter *subgroups* aninhados cujas transformações deverão ser herdadas. No entanto, na construção de *groups* não aninhados as transformações associadas ao primeiro não deverão ser transmitidas aos seguintes e assim consecutivamente.

Classes

Para atingir os objetivos, foi necessário criar uma classe para cada tipo de transformação existente e uma classe *group* que identifica os parâmetros que são necessários armazenar para a sua futura construção. De seguida iremos especificar cada uma destas.

Class Translate

No desenvolvimento desta classe sabemos que para realizar uma translação é necessário três parâmetros, três coordenadas, que serão associados mais tarde à função *gl_Translatef*. Desse modo são guardadas três *floats*, x, y e z.

Class Rotate

Para esta, utiliza-se o mesmo raciocínio da classe anterior, adicionando um variável que representa o ângulo e novamente três *floats* x, y e z, utilizados para a função *gl_Rotatef*.

Class Scale

Novamente, para se poder utilizar a função *gl_Scalef* são guardados três parâmetros x, y e z.

Class Group

Sabendo que, cada *group* pode ter associado a si várias transformações (translações, rotações e escalas) foram criados três parâmetros: *Translate* t*, *Rotate* r*, *Scale* s*, utilizando como recurso as classes anteriormente referidas. Além disso, sabe-se que cada *group* pode ter agregado vários *subgroups* e, portanto, é armazenado um *Vetor <Group>* para os seus filhos, assim como o número destes. Por fim, são criados dois vetores um de triângulos relativos aos ficheiros .3d dos *models* do *group*, uma vez que é mais eficiente guardar os triângulos e não o nome dos ficheiros, e também um vetor que contém o nome das transformações guardadas por ordem que foram lidas nesse mesmo *group*. Isto pois foi lecionado nas aulas teóricas que a ordem pela qual as transformações são realizadas alteram o resultado final.

Raciocínio para Leitura dos *Groups*

Foi criado como variável global um vetor com o nome *groupbrothers*, de maneira a armazenar os principais *groups* irmãos escritos no ficheiro *XML* assim como, uma variável *int iBrother* representando um contador dos mesmos. Na função *engine* após a leitura do <*world*> invoca-se a função *readCamera* para o filho do mesmo, função esta que é responsável por completar os parâmetros necessários às funções *gluLookAt* e *gluPerspective*.c De seguida, atendendo que o *group* é um irmão de <*camera*>, no ficheiro *XML*, é feito um ciclo *for* responsável por iterar todos os irmãos do mesmo onde se completa o vetor *groupbrothers* com invocação da função *readGroup(group)* onde cada *group* é preenchido por *group = group->NextSiblingElement("group")*.

Função *readGroup*

Esta função é responsável por preencher os campos da classe *Group*, de forma recursiva, uma vez que é a forma possível para criar cada filho depois adicionado ao vetor, que por sua vez é transmitido como parâmetro da função.

Inicialmente é criado um ciclo que permite iterar os elementos do ficheiro XML que correspondem às transformações, sejam elas translações, rotações ou escalas. De forma a permitir a possibilidade de leitura dos ficheiros da primeira fase do projeto, cria-se ainda a condição de não existirem transformações no ficheiro. Assim, a equipa estabeleceu que caso terminem ou não existam transformações o ciclo termina e prossegue-se com a leitura dos ficheiros incluídos nos modelos.

Para armazenar nos vetores das transformações os valores lidos, é criada a leitura do nome do tipo da mesma, e com condições verifica-se a que tipo corresponde, armazenando-se no array correspondente.

No final, itera-se os ficheiros e invoca-se a função que lê os mesmos e devolve um vetor de triângulos, de seguida foi criado um ciclo recursivo para cada *subgroup*, adicionando os mesmos ao vetor de filhos e invocando esta mesma função para cada um deles.

Raciocínio para o Render Scene

Para o render scene desenhar os *groups* necessários, respeitando a hierarquia pedida, é realizado um raciocínio recursivo em todo semelhante com aquele que foi realizado para preencher as classes criadas.

Assim, começa-se pela iteração recursiva dos elementos do vetor de *groups* irmãos, onde para cada um é realizado um *glPushMatrix()* e no final um *glPopMatrix()*, isto porque para cada irmão as transformações não são aplicadas hierarquicamente. Para cada um, invoca-se a função *draw*, responsável por desenhar cada *group*, apresentando as suas transformações.

Função *draw*

A função *draw* recebe como parâmetro um grupo e acede ao seu campo que contém os nomes das transformações que ser-lhe-ão aplicadas. Para cada uma destas transformações acede ao array na posição indicada e retira os campos de cada, aplicando a mesma com os respectivos valores guardados. No final de aplicar as funções correspondentes às transformações, itera o vetor dos triângulos desse grupo.

Por fim, é criado um ciclo que acede ao campo do array de filhos do grupo, invocando-se recursivamente a função *draw* para cada um dos filhos. É de reparar que é usado um *glPushMatrix()* e um *glPopMatrix(),* para cada filho, pois os filhos entre si correspondem a irmãos, e portanto, devem apenas incluir as transformações correspondentes aos seus pais e não conter as dos seus irmãos anteriormente lidos.

Informações adicionais

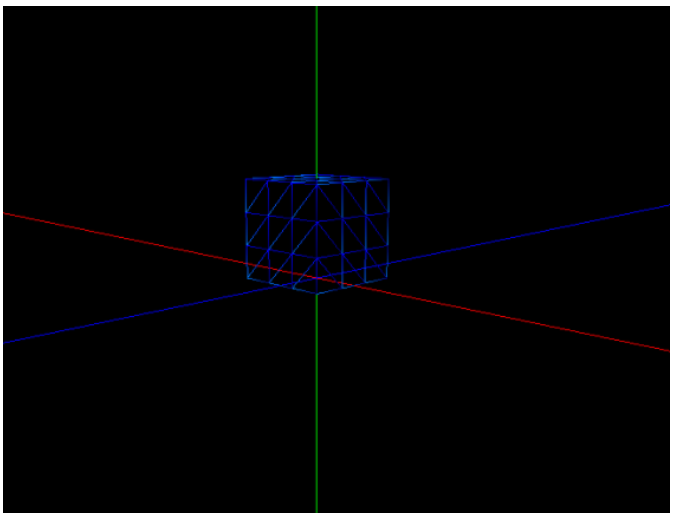
É importante referir que a equipa incluiu, nesta fase, a possibilidade de mover a figura através dos rato, no entanto, atendendo que se pede que a figura seja visualizada através de uma posição da câmera refletida no ficheiro *xml* lido, foi necessário utilizar uma função que confere os ângulos alfa e beta, que são alterados com o movimento realizado pelo rato. Estes valores são calculados com base nas coordenadas esféricas, pois com as mesmas é possível obter os ângulos através *arcsin e arctan*. De realçar que o raio é calculado com base na distância da posição da câmera e do ponto de visualização, ambos retirados da leitura do ficheiro *xml*.

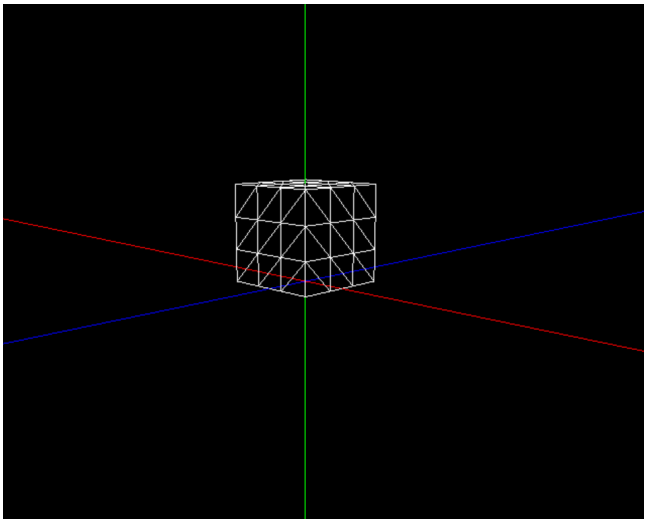
Resultados obtidos

A equipa antes de criar o ficheiro *xml* que deveria representar o sistema solar, testou o projeto com os ficheiros disponibilizados pela equipa docente, obtendo os seguintes resultados:

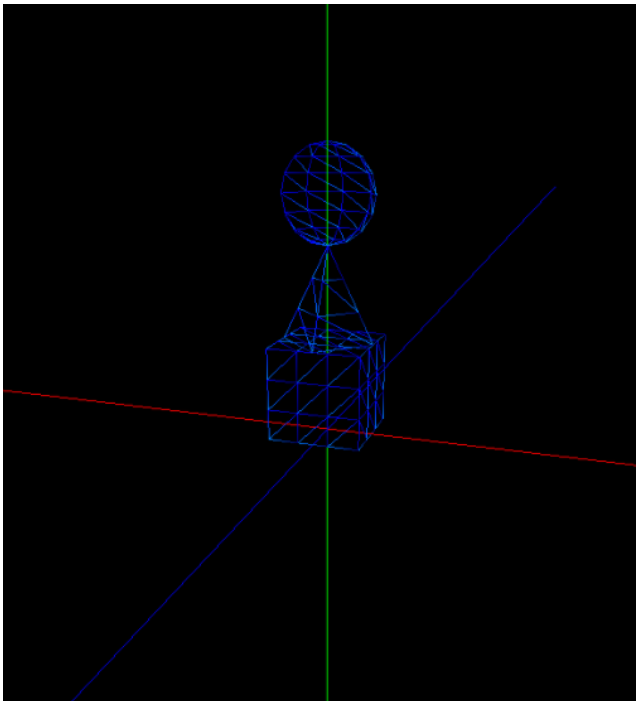
Do lado esquerdo, com o desenho na cor azul podemos encontrar os resultados da equipa face ao ficheiro teste que se pode visualizar do lado direito com a imagem da figura na cor branca.

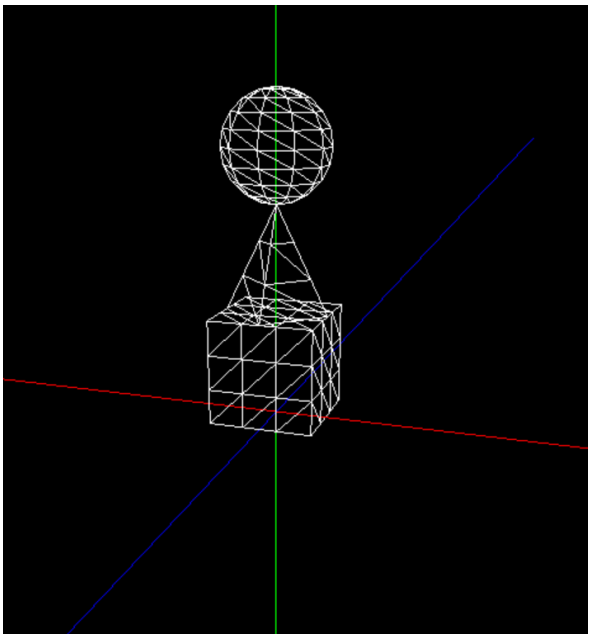
test_2_1.xml:



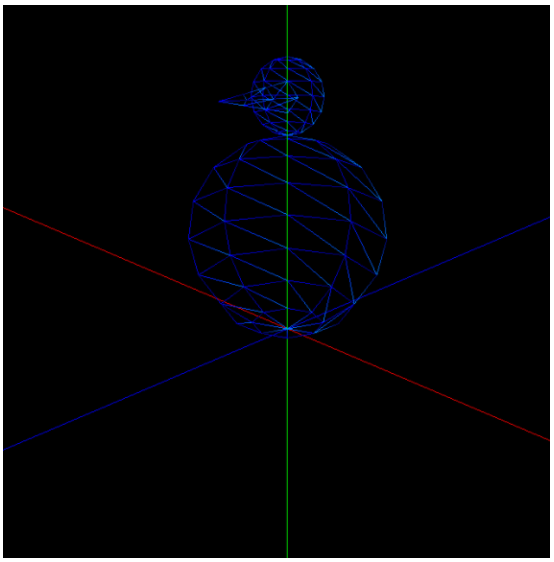


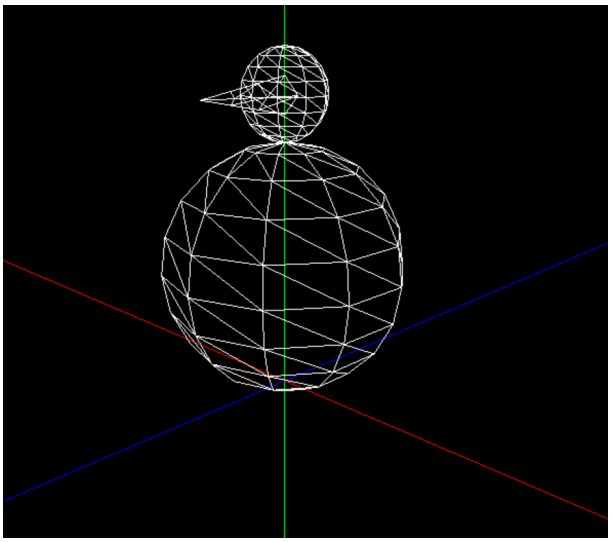
test_2_2.xml:



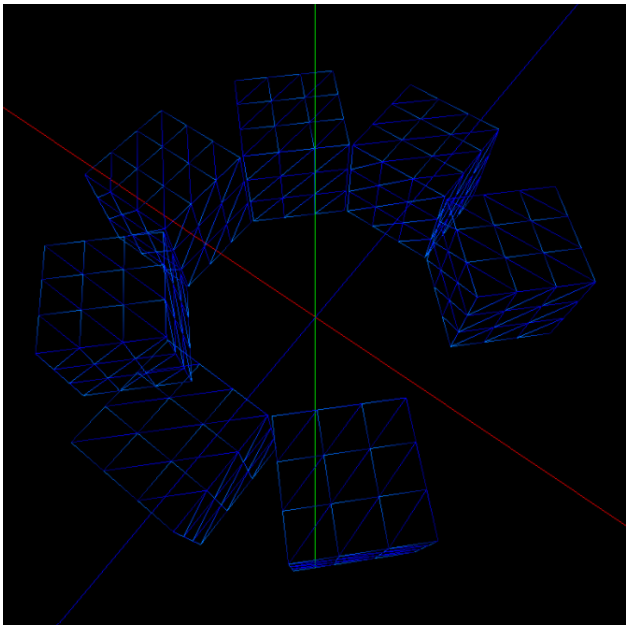


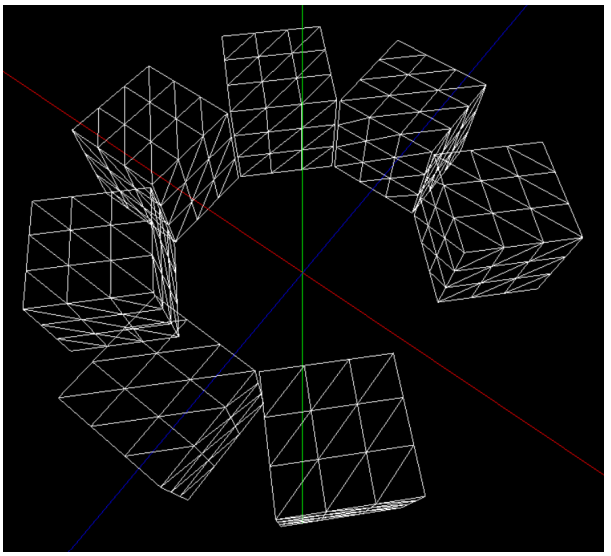
test_2_3.xml:





test_2_4.xml:





Sistema Solar

O ficheiro *xml* que modela o sistema solar, é constituído de forma hierárquica, com o sol por elemento pai, os planetas como seus elementos filhos, e por sua vez os planetas têm como elementos filhos as respectivas luas, devido ao imenso número de luas o grupo limitou-se a representar apenas um reduzido número variando de acordo com o número real de luas por planeta, e no caso de saturno também existem esferas representando os meteoros que formam os famosos anéis que rodeiam o planeta. Todos estes corpos celestes foram desenhados a partir do ficheiro *sphere.3d* variando de tamanho com diferentes escalas, e utilizando as transformações *translate* e *rotate* para os diferenciar em posição. Para as imagens seguintes, a equipa utilizou como recurso o teclado para rodar de forma mais eficiente em torno do eixo y e afastar-se da câmara. No entanto, também é possível modelar a imagem com recurso ao rato.