Computação gráfica - Fase 3

Diogo Pires[a93239], Gonçalo Soares[a93286], Marco Costa[a93283], and Rita Teixeira,[a89494]

Universidade do Minho

1 Introdução

Este projeto foi realizado no âmbito da Unidade Curricular de Computação Gráfica. O principal objetivo deste trabalho é a continuação das fases anteriores, a partir das quais conseguimos obter vários modelos e aplicar-lhes transformações. Nesta nova fase, os objetivos foram:

- Gerar novos modelos baseados em *patches* de Bezier;
- Utilizar VBO's;
- Implementação de Catmull- Rom cubic curves.
- Modo multi-view.

. Sendo assim, ao longo deste relatório iremos descrever os passos tomados que levaram à nossa resolução e também iremos demonstrar tais resultados.

2 Generator

Para esta fase, foi implementada a transformação de um ficheiro do tipo *patch* em um ficheiro do tipo *3d*, que, posteriormente, vai ser lido no *Engine*.

O ficheiro *patch* contém a descrição da superfície de *Bezier* desejada. Já o ficheiro *3d*, como vimos em fases anteriores, contém um conjunto de triângulos a serem desenhados pelo *Engine*.

Assim sendo, o primeiro passo é o tratamento do ficheiro *patch*. Depois da extração dos pontos de controlos de cada *patch* definido no ficheiro, passamos à parte do seu processamento. Assim, para calcular um dado ponto num instante u e v foi utilizada a seguinte formula:

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

Fig. 1. Formula utilizada para o calculo do ponto da superfície *Bezier* num dado instante u e v

Optamos por pre-calcular a multiplicação das matrizes que, em principio, não se alterariam. Ou seja, $M * P * M^T$, onde M é a Matriz de *Bezier*, M^T a sua transposta e P a matriz que contém os 16 pontos de controlo.

Para facilitar a multiplicação decidimos calcular separadamente as coordenadas do ponto final. Logo vamos ter 3 matrizes pre-calculadas para cada campo da coordenada.

Segue-se, agora, o cálculo de todos os pontos da superfície para a posterior definição de triângulos. Para este cálculo é utilizada a tesselação passada como argumento como o número de divisões por cada *patch*.

Para o cálculo dos triângulos simplesmente iteramos cada conjunto de quatro pontos consecutivos (em duas linhas e colunas consecutivas) formando sempre dois triângulos para definir este "quadrado".

3 Engine

A respeito do *engine*, o objetivo é o de realizar transformações de *translate* e *rotate* dos elementos. Estas animações serão realizadas baseadas em *Catmull-Rom cubic curves*, sendo que também serão considerados VBOs.

Assim sendo, alteramos o rendering dos modelos de maneira a utilizar VBOs e também adicionamos translações de *Catmull-Rom* e rotações dinâmicas.

3.1 VBOs

Para alterarmos a renderização dos modelos de forma a utilizarem buffers de vértices, tivemos que mudar a estrutura da classe *Model*. Assim, removemos a classe *Triangle* que usavamos para guardar os vários triângulos e substituímos por um vetor de *floats* que contém todos os vértices de cada triângulo, sequencialmente.

Com isto, para renderizar o modelo, precisamos apenas de "dizer" à placa gráfica para desenhar o *buffer* de vértices todo de uma só vez. Isto, obviamente, é muito mais eficiente do que a implementação anterior em que enviavamos à placa gráfica um vértice de cada vez.

3.2 Animações

Para implementarmos as animações, foi necessário alterar as classes *Translate* e *Rotate*. Para este efeito, em ambas as classes, tivemos que adicionar algumas variáveis:

- **bool dynamic** - Define-se uma translação/rotação como dinâmica (animação) ou estática. Desta forma, é possível distinguir se a transformação é relativa a uma animação, no caso dos planetas a rodarem, ou se descreve uma translação para formar uma cena estática, como a da pirâmide.
- **milliseconds full_time** - Define o tempo total da animação
- **int start** - Define o instante de tempo na qual a animação começou.

Rotação A rotação não se mostrou um grande desafio, não sendo necessário adicionar nenhuma variável, para além das referidas acima. Deste modo, só temos que calcular o tempo decorrido com a seguinte formula:

$$cur\_time = (glutGet(GLUT\_ELAPSED\_TIME) - start)\%full\_time$$

De seguida, usamos este valor para calcular a rotação atual:

$$cur\_rotation = (cur\_time * 360.0f)/(full\_time)$$

3.3 Translação

Para calcularmos a curva de Catmull-Rom necessária para as transações dinâmicas tivemos que recorrer ao uso de matrizes.

$$p(t) = [t^3 \quad t^2 \quad t \quad 1] \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_o \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

Fig. 2. Formulas para o cálculo da posição num instante de tempo t

Esta equação, dado um instante de tempo t e quatro pontos de controlo, retorna a posição nesse instante. Mais ainda, se substituirmos a matriz do tempo pela sua derivada, obtemos a tangente da curva nesse instante, que é necessária para alinharmos a direção do objeto com a curva.

Apesar disto, esta equação apenas calcula a curva entre os pontos P_1 e P_2, para além de que só faz uso de quatro pontos de controlo, o que não gera uma curva muito interessante. Por causa disto, para desenharmos a curva inteira, temos de conseguir iterar por todos os pontos de controlo, de forma a conseguirmos desenhar todos os segmentos constituintes da curva.

```
void Translate::get_global_catmull_rom_point(float gt, float *
    pos, float *deriv) {
    int n_points = ctrl_points.size();

    float t = gt * n_points;
    int index = floor(t);
    t = t - index;
```

```
    int indices[4];
    indices[0] = (index + n_points-1) % n_points;
    indices[1] = (indices[0]+1) % n_points;
    indices[2] = (indices[1]+1) % n_points;
    indices[3] = (indices[2]+1) % n_points;

    auto p0 = ctrl_points[indices[0]];
    auto p1 = ctrl_points[indices[1]];
    auto p2 = ctrl_points[indices[2]];
    auto p3 = ctrl_points[indices[3]];

    get_catmull_rom_point(t, p0, p1, p2, p3, pos, deriv);
}
```

Esta função recebe o tempo real decorrido e decide quais os pontos de controlo, assim como o tempo t dentro desse segmento que deve enviar para a função que calcula a posição e a derivada.

De seguida, a translação pode ser aplicada imediatamente. Porém, para obtermos a matriz de rotação, necessitamos de mais alguns cálculos:

$$\begin{aligned} \overrightarrow{X_i} &= p'(t) \\ \overrightarrow{Z_i} &= X_i \times \vec{Y}_{i-1} \\ \overrightarrow{Y_i} &= \overrightarrow{Z_i} \times \overrightarrow{X_i} \end{aligned} \qquad M = \begin{bmatrix} X_x & Y_x & Z_x & 0 \\ X_y & Y_y & Z_y & 0 \\ X_z & Y_z & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig. 3. Cálculo da matriz de rotação

Com esta matriz podemos finalmente aplica-la à matriz do OpenGL através da função *glMultMatrixf*.

Variáveis adicionais De forma a acomodarmos as alterações requeridas por esta fase, bem como outras que achamos interessantes adicionar, as seguintes variáveis foram criadas na classe *Translate*:

- **vector<tuple<float,float,float>> ctrl_points** - Lista de todos pontos de controlo da curva;
- **bool align** - Define se o modelo deve, ou não, ser rodado de forma a ficar alinhado com a trajetória;
- **bool draw** - Define se a curva deve ser desenhada;
- **int curve_segments** - Número de segmentos em que a curva desenhada deve ser dividida.
  Nota: Isto não afeta a translação, apenas o desenho da curva.

– **float last_y[3]** - Último Y calculado para matriz de rotação. (Ver figura 4)
– **int offset** - Offset temporal na animação.

4 Cena desenvolvida

4.1 Script em Python

Para gerarmos o ficheiro xml com o sistema solar especificado pelo enunciado, decidimos criar um *script* em Python para o gerar automaticamente. Isto tem o intuito de, não só facilitar a criação do ficheiro, mas também a de permitir adicionar elementos à cena que não seriam exequíveis manualmente. Entre estas estão:

– Criação das trajetórias dos planetas e do cometa;
– Adição de asteroides aos anéis de Saturno. Cada asteroide move-se a uma velocidade diferente;
– Adição de partículas à cauda do cometa. Cada partícula move-se a uma velocidade diferente e tem um ângulo de trajetória ligeiramente diferente;

Na secção dos anexos podemos observar o resultado final da cena desenvolvida.

5 Multiview

Na fase 2, nós introduzimos como funcionalidade extra o modo multiview. Nesta fase, decidimos expandir a utilidade desta funcionalidade do nosso programa, ao permitir um número variável e, teoricamente, ilimitado de vistas possíveis, bem como a seleção e o ajuste do tamanho de cada perspetiva.

Nota: A estratégia apresentada a seguir foi idealizada inteiramente pelo nosso grupo. Por causa disto, pode não ser a melhor nem a mais eficiente forma de implementar estas funcionalidades.

Para acomodarmos as funcionalidades mencionadas anteriormente, tivemos que alterar a forma como guardamos as vistas. Assim, criamos uma classe View que, essencialmente é uma *full tree* em que os nodos contêm um valor de divisão que especifica onde repartir o espaço que lhe foi alocado e as folhas são apontadores para perspetivas.

Cada perspetiva é guardada numa classe com o mesmo nome que inclui todos os valores relativos ao *look at* dessa perspetiva. Estas perspetivas estão, por sua vez guardadas numa lista à qual o nosso programa irá aceder para obter as várias perspetivas quando uma atualização da árvore de vistas for executada.

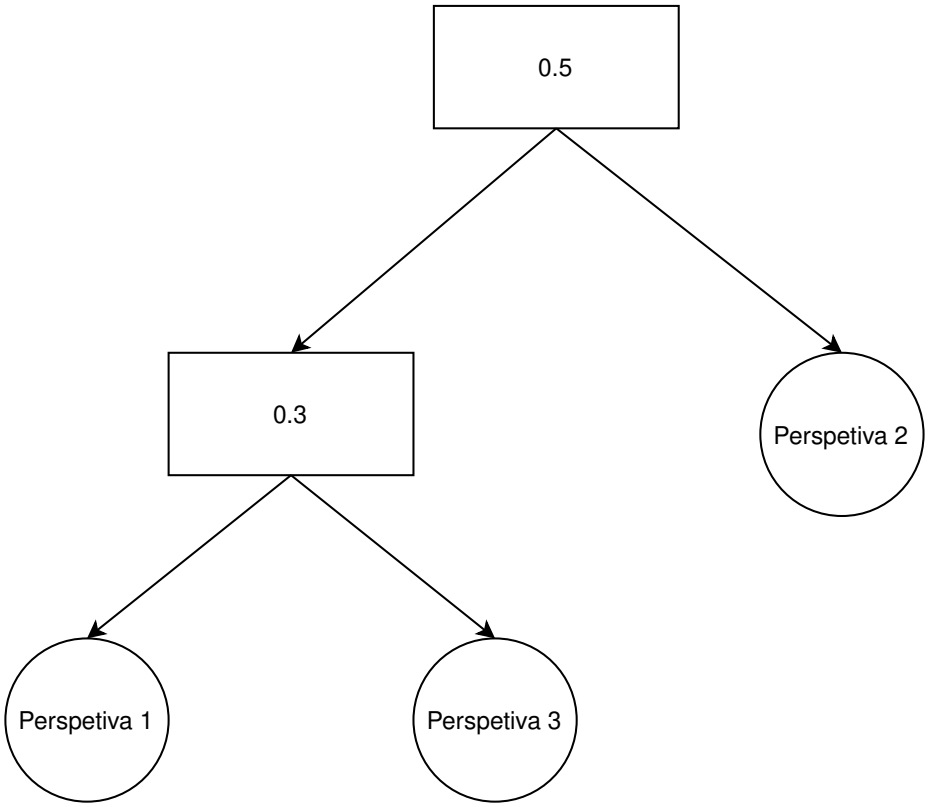

Fig. 4. Exemplo de uma árvore de vistas

Neste exemplo, numa janela 1920x1080 de resolução, o nodo raiz, com uma divisão de 0.5 repartiria a janela em dois retângulos de 990x1080, sendo a metade esquerda, por sua vez repartida verticalmente num rácio de 0.30/0.70, ficando este retângulo dividido em dois pedaços menores, uma de 990x324 e outra de 990x756.

Recapitulando, com esta árvore acabamos no lado direito com a perspetiva 2 com um tamanho de 990x1080 e no lado esquerdo há ainda uma divisão vertical adicional, ficando a perspetiva 1 com o lado inferior e um tamanho de 990x324 e, por fim, a perspetiva 3 fica em cima com um tamanho de 990x756.

É possível detetar que os nomes das perspetivas do diagrama estão ordenados de uma maneira "estranha". Isto é intencional, pois é assim que a nossa função que constrói a árvore de vistas atribui as perspetivas às diferentes vistas.

Observemos o seguinte diagrama:

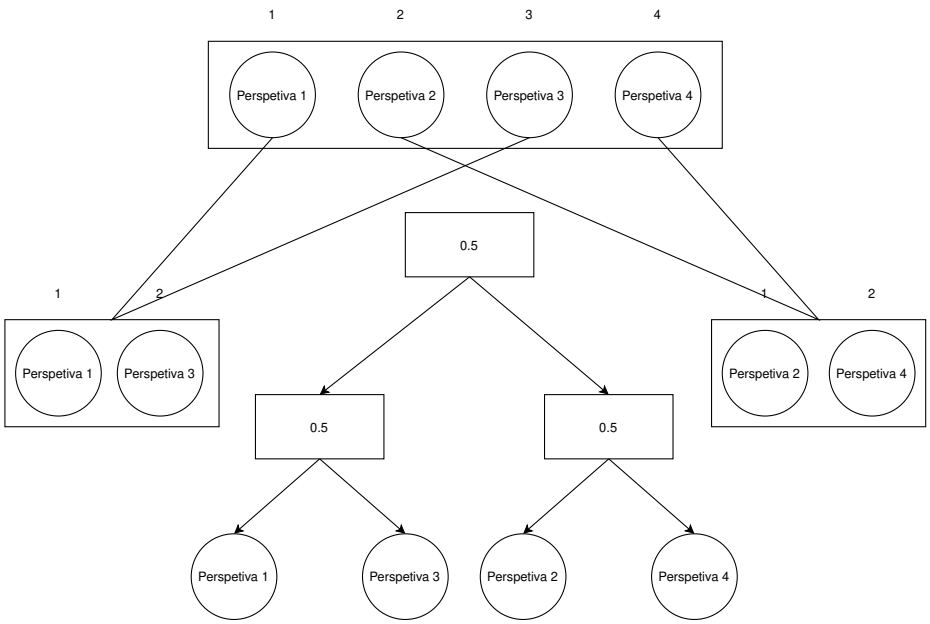

Fig. 5. Distribuição das perspetivas pelas vistas

Neste exemplo temos uma lista de quatro perspetivas a partir da qual queremos criar uma árvore de vistas. Assim, dividimos a lista em duas com metade do tamanho, com base na paridade do índice dos elementos e passamos uma metade para cada um dos lados. Isto é feito recursivamente até só termos uma lista de um elemento onde atribuímos a perspetiva à folha.

Este método é melhor que apenas dividir a lista ao meio, pois garante que, ao atualizarmos o número de vistas, estas se mantêm do mesmo lado do ecrã.

5.1 Seleção de vistas

Para permitir a seleção, temos primeiro que detetar que retângulo é que o utilizador está a selecionar.

Para isto, temos que descer a árvore de acordo com as coordenadas do rato no ecrã até chegarmos a uma folha, onde procedemos a substituir a vista. No entanto, surge um problema: o utilizador quer clicar outra vez para retornar ao modo em que se encontrava anteriormente e, por causa disto, não podemos substituir a árvore que tínhamos.

Para resolver este problema, precisamos de uma árvore de vistas adicional para guardarmos a perspetiva selecionada sem destruirmos a árvore atual. Adicionamos, também uma *flag selected* que indica se uma perspetiva está selecionada ou não. Depois, para renderizar basta verificar se a *flag* está selecionada ou não e renderizar a árvore correta.

5.2 Redimensionamento das vistas

O redimensionamento das vistas ocorre de forma semelhante à seleção das perspetivas.

Começamos por descer pela árvore até encontrarmos um **nodo** em que o rato esteja próximo o suficiente da sua divisão e retornamos o apontador para esse **nodo**.

De seguida, usamos a posição do rato para calcular a nova divisão que esse nodo deve assumir e substituímos o valor.

Caso cheguemos a uma folha, significa que o utilizador não clicou em nenhuma divisão e não acontece nada.

6 Conclusão

Concluindo, nesta fase melhoramos os nossos conhecimentos sobre a criação e manipulação de curvas de Bezier e as curvas cúbicas de *Catmull-Rom*. Por outro lado, também decidimos melhorar a funcionalidade extra de multi-view implementada anteriormente. Também conseguimos melhor o desempenho do *engine* guardando os vértices dos modelos na placa gráfica através da utilização dos *VBO*'s.

De um modo geral, o grupo encontra-se bastante satisfeito com os resultados obtidos.

7 Anexos

De seguida mostramos algumas imagens das cenas que geramos:

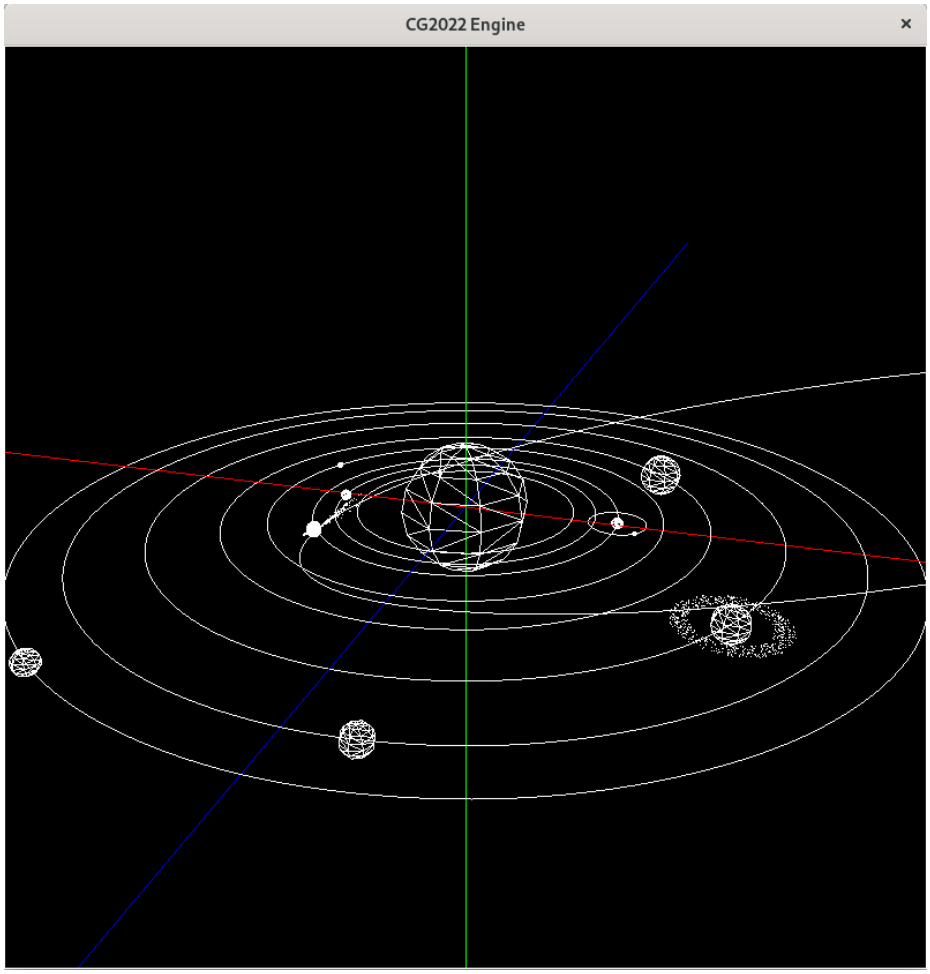

Fig. 6. Cena do sistema solar.

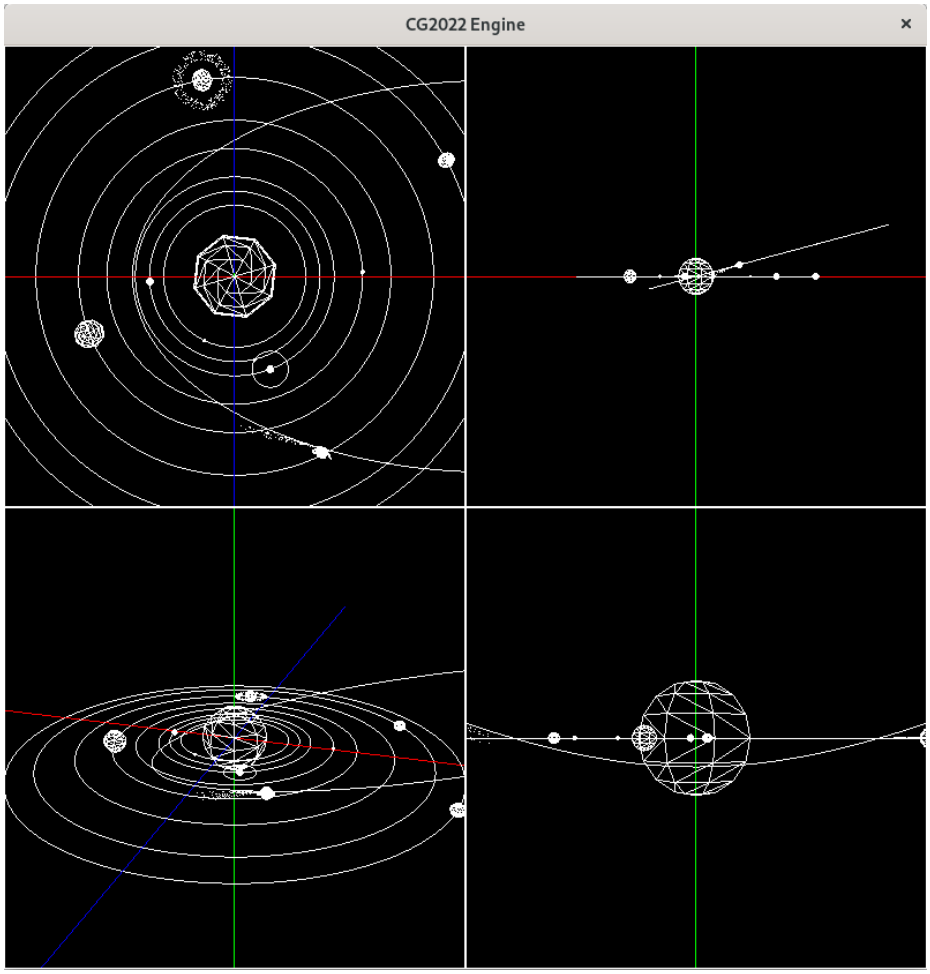

Fig. 7. Cena do sistema solar no modo multi-view. (4 perspetivas)

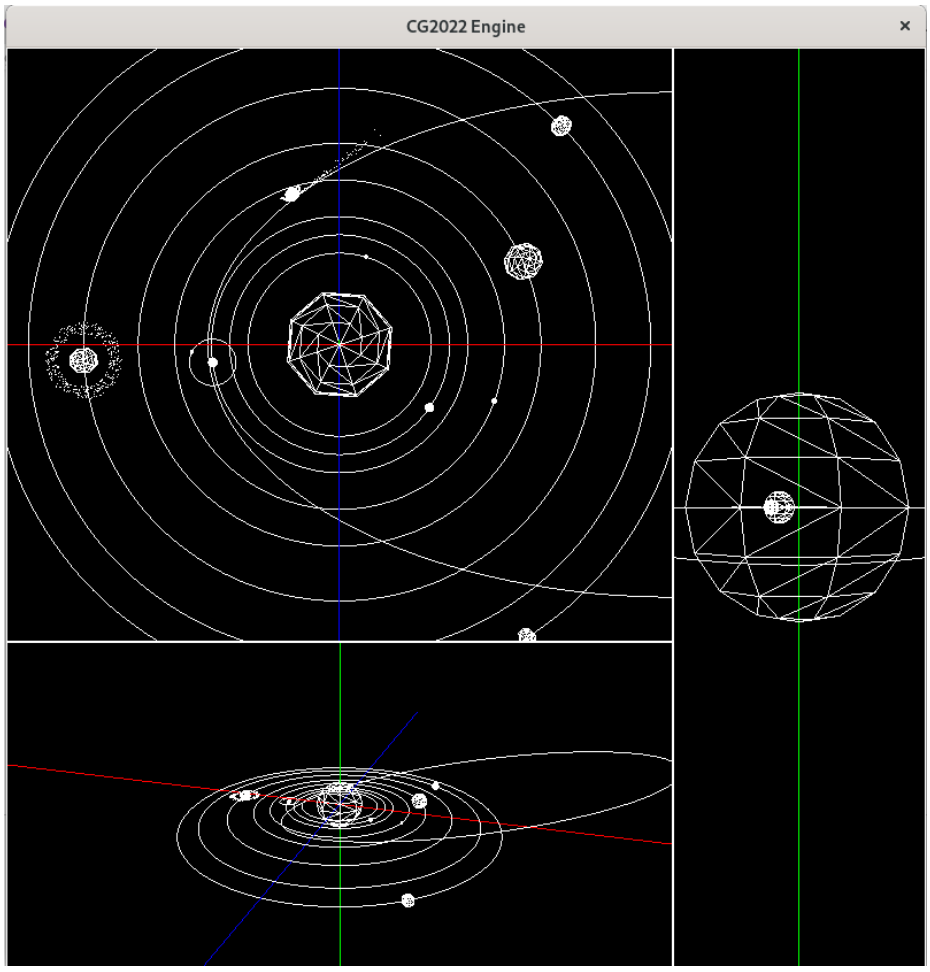

Fig. 8. Cena do sistema solar com as vistas redimensionadas