Computação Gráfica: Fase 2 – Primitivas Gráficas

Afonso Faria,
André Ferreira,
Bruno Campos e
Ricardo Gomes

Universidade do Minho, Braga, Portugal
{a83920,a93211,a93307,a93785}@alunos.uminho.pt

Resumo Nesta fase foi-nos proposta a implementação de *scenes* hierárquicas com transformações geométricas. Uma *scene* é definida como uma árvore onde cada nodo contém um conjunto de transformações geométricas (translação, rotação e escala) e, opcionalmente, um conjunto de modelos. Cada nodo também pode ter nodos filhos.

1 Introdução e Contextualização

O presente relatório diz respeito à segunda fase do projeto da unidade curricular de Computação Gráfica.

Nesta fase foi-nos proposta a implementação de *scenes* hierárquicas com transformações geométricas.

Nas seguintes secções do relatório iremos apresentar a abordagem tomada na resolução dos problemas e os obstáculos que surgiram durante a realização do projeto.

É também de notar que, embora algumas alterações foram feitas ao *generator*, todas as melhorias significantes para esta fase foram feitas no *engine*.

2 *Engine*

Na fase anterior já tínhamos desenvolvido as estruturas necessárias para a representação de transformações, assim como o código necessário para executar o seu *parsing*.
Também já tinha sido desenvolvida a funcionalidade de ler os ficheiros XML e renderizar todos os grupos recursivamente.
Assim, iremos relembrar o que já foi descrito no relatório da fase anterior. De seguida, iremos descrever aquilo que de facto foi desenvolvido para esta fase.

Comportamento que foi alterado desde o último relatório será marcado com ~~*strikethrough*~~, e o novo comportamento será descrito imediatamente a seguir.

2.1 Transformações Geométricas

Estruturas de dados Para representar um *group* em memória são usadas as seguinte estruturas, organizadas hierarquicamente de forma similar ao formato dos ficheiros XML:

```
struct group {
    std::vector<transform> transforms;
    std::vector<model> models;
    std::vector<group> children;
};

struct transform {
    enum class kind_t {
        translate,
        rotate,
        scale
    } kind;

    union {
        glm::vec3 translate;
        glm::vec4 rotate;
        glm::vec3 scale;
    };
};

struct model {
    std::vector<float> coords;
};
```

O tipo `glm::vec3` representa um *array* de três `float`s, e o tipo `glm::vec4` representa um *array* de quatro `float`s.

~~Inicialmente, era usado o tipo `float` para todos os números reais. No entanto, as funções da API **GLUT** que manipulam a câmara utilizam o tipo `double`. Por esta razão, todas as estruturas, à exceção da câmera, utilizam o tipo `float`, e a câmera utiliza o tipo `double`.~~
Nesta fase substituímos todo o uso de `double`s por `float`s, de forma a reduzir a quantidade de código repetido e obter uma precisão aritmética consistente em todo o projeto.
Esta alteração causou alguns problemas com as funções `gluLookAt` e `gluPerspective`, que recebem valores do tipo `double` em vez de `float`. **GLUT** não fornece nenhuma versão alternativa destas funções.
Assim sendo, tentamos implementar estas funções manualmente com recurso à biblioteca **glm** que já estávamos a utilizar. No entanto, estávamos a obter resultados ligeiramente diferentes daqueles esperados, por isso contentamo-nos por usar as funções do **GLUT** e efetuar *casts* para silenciar avisos do compilador. No futuro este será um problema que iremos resolver.

Um `group` é composto por três *arrays* de tamanho variável: um para *transforms*, outro para *models*, e outro para sub-grupos. Deste modo, um `group` pode ser considerado uma *rose tree*.

Um `transform` pode ser um `translate`, `rotate`, ou `scale`. Por motivos de eficiência de espaço, optamos por usar uma *union* e um discriminante (~~`transform::kind`~~ `transform::kind_t`) para o representar. Alternativamente, podíamos recorrer a herança e *dynamic dispatch*, mas essa solução implicaria uma maior quantidade de código *boilerplate* e possivelmente um custo no tempo de execução.

Um `model` é composto por todas as coordenadas cartesianas necessárias para o representar. Assim, o comprimento de `model::coords` será sempre um múltiplo de três.

Algoritmo De forma a aplicar as transformações, foi apenas necessário de modificar o seguinte código do *renderer*:

```
auto render_group(group const& root) noexcept -> void {
    glPushMatrix();

    for (auto const& transform : root.transforms) {
        switch (transform.kind) {
            using enum transform::kind_t;
            case translate:
                glTranslatef(
                    transform.translate.x,
                    transform.translate.y,
                    transform.translate.z
                );
                break;
            case rotate:
                glRotatef(
                    transform.rotate[0],
                    transform.rotate[1],
                    transform.rotate[2],
                    transform.rotate[3]
                );
                break;
            case scale:
                glScalef(
                    transform.scale.x,
                    transform.scale.y,
                    transform.scale.z
                );
                break;
        }
    }

    // código que renderiza modelos omitido por efeitos de brevidade.
    // ...

    for (auto const& child_node : root.children) {
        render_group(child_node); // chamada recursiva.
    }

    glPopMatrix();
}
```

Como se pode observar, os grupos são percorridos recursivamente numa travessia *pre-order*. As transformações de cada grupo são aplicadas aos modelos desse mesmo grupo e aos modelos dos seus sub-grupos.

3 Resultados

Utilizando os ficheiros de configuração XML fornecidos pelos docentes para fins de testes, bem como os modelos especificados nestes, obtemos os seguintes resultados:

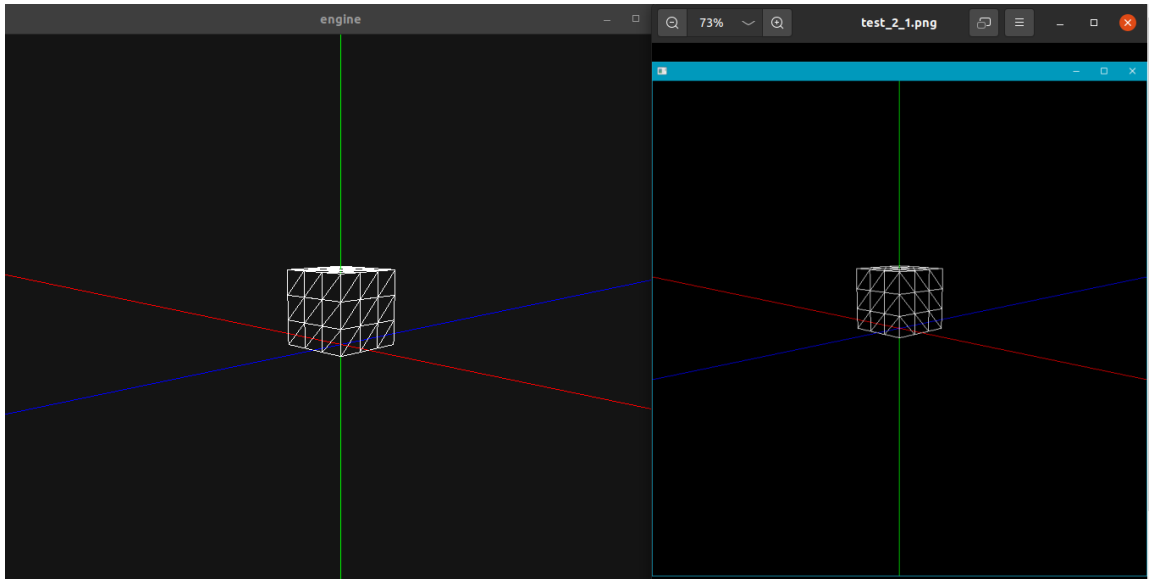

Figura 1. Resultado de **test_2_1.xml**.

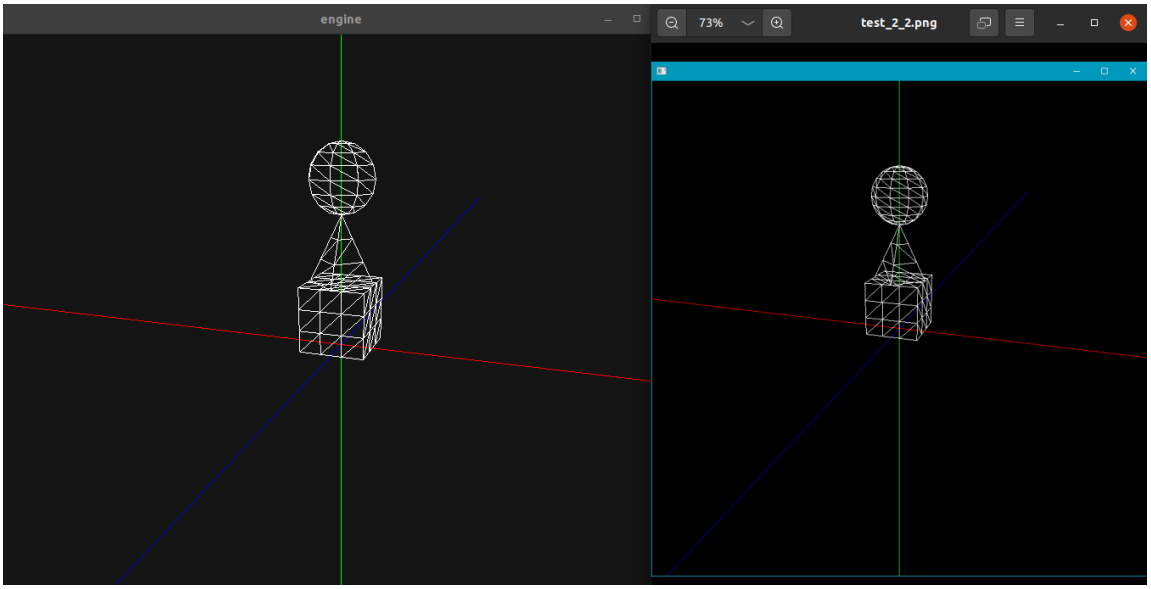

Figura 2. Resultado de **test_2_2.xml**.

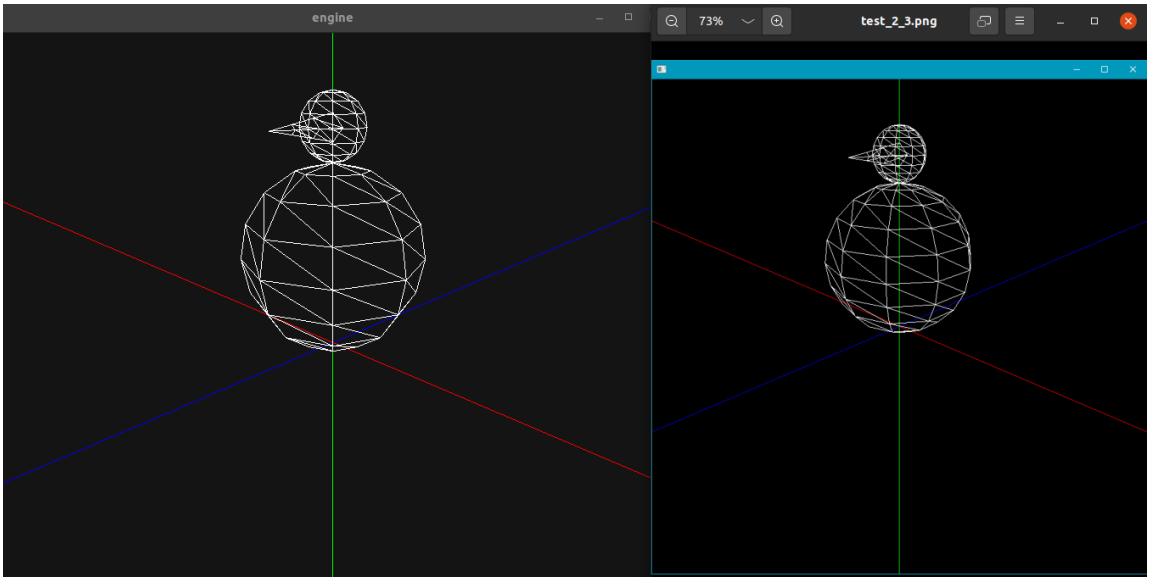

Figura 3. Resultado de **test_2_3.xml**.

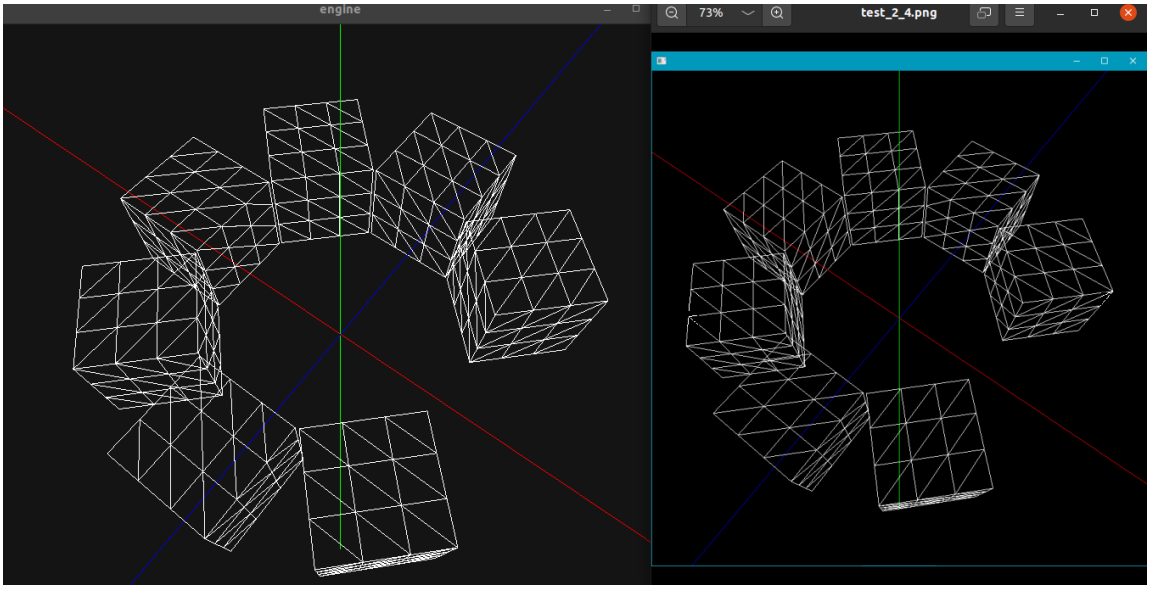

Figura 4. Resultado de **test_2_4.xml**.

Comparando as imagens produzidas pelo programa com as imagens fornecidas juntamente com os ficheiros XML, podemos concluir que foram atingidos os resultados pretendidos.

3.1 Sistema Solar

O demo do sistema solar foi criado a partir de um ficheiro XML manual com as transformações e os modelos de cada astro. A imagem seguinte demonstra a renderização deste ficheiro XML:

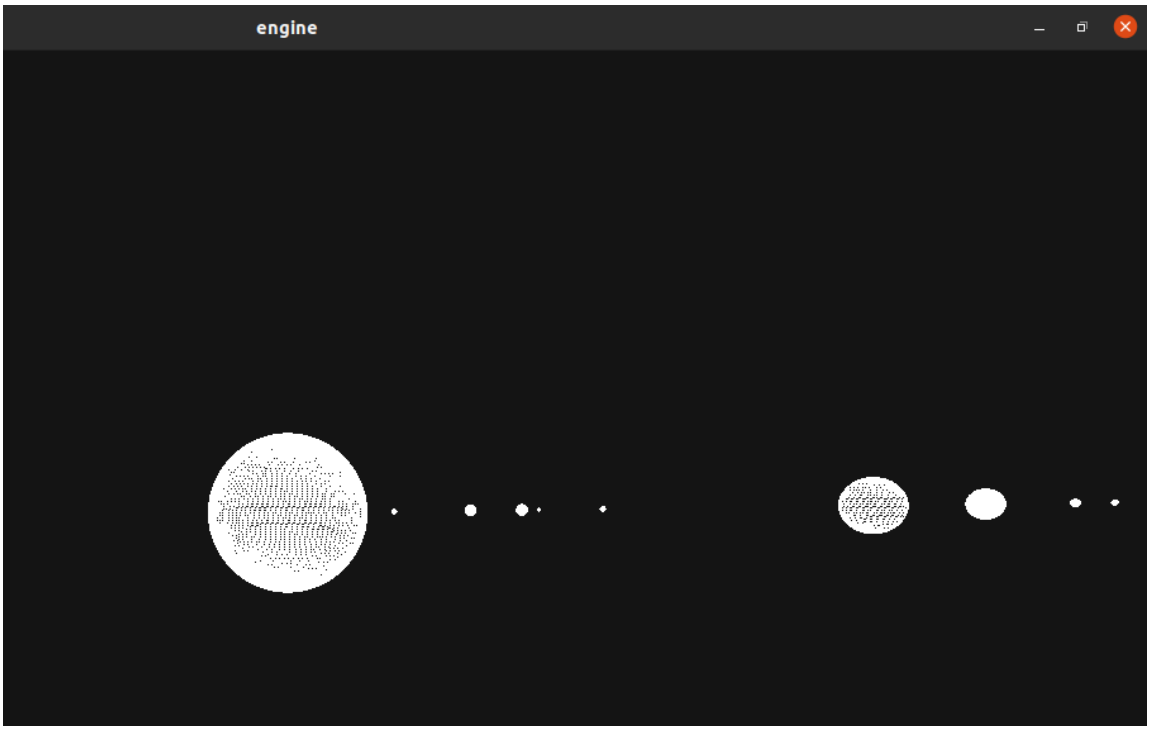

Figura 5. *Demo scene* do Sistema Solar.

Foram levadas em conta as dimensões reais de cada entidade do sistema solar. A tabela seguinte descreve as propriedades de cada astro:

Corpo celeste	Raio (km)	Distancia ao Sol (milhões de km)
Sol	696 340	0
Mercúrio	2 4397	57
Vénus	6 051,8	108
Terra	6 371	149
Marte	3 389,5	228
Júpiter	69 911	780
Saturno	58 232	1437
Úrano	25 362	2871
Neptuno	24 622	4530

Adicionalmente, a distância da Lua à Terra é de **384,400 km** e o seu raio é **1,737.4 km**.

É de notar que as ordens de grandeza do Sol e da Lua, quando comparados com o resto dos planetas, são *'gigantesca'* e *'minúscula'*, respetivamente. De forma a possibilitar a sua visualização, foi usada uma escala individual para cada um destes astros.

Cada planeta usa uma escala de **1000** para **1** no seu raio e uma escala de **400,000** para **1** na sua distância ao Sol.

O Sol tem uma escala de **10,000** para **1** no seu raio. Deste modo, o seu raio real será **69.6**, ou seja, será representado por um modelo **10 vezes menor** do que realmente é quando comparado com os outros astros.

A Lua terá a mesma escala para o raio que os planetas, mas para que seja possível visualizá-la, foi usada uma escala de **10,000** para **1** na sua distancia à Terra. Assim, a Lua é representada **40 vezes** mais distante da Terra do que realmente é.

De seguida apresentamos uma versão mais natural do sistema solar, na qual os astros não estão perfeitamente alinhados:

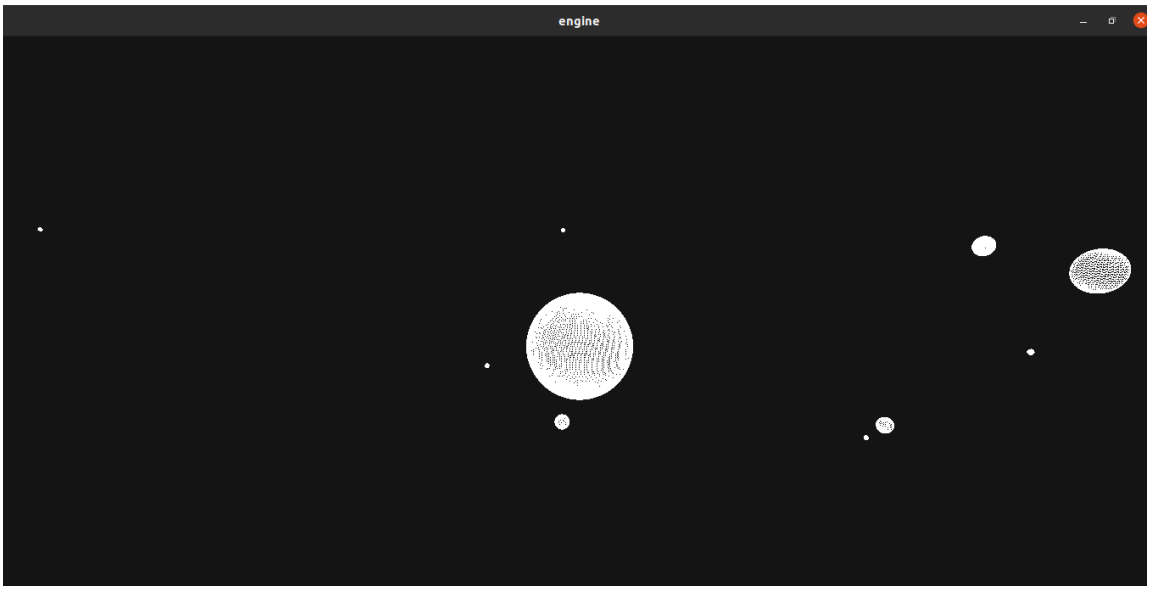

Figura 6. *Demo scene* do Sistema Solar com os astros não alinhados.

4 Extras

Visto que grande parte da funcionalidade desta fase já tinha sido implementada na fase anterior, utilizamos o tempo que nos foi dado para implementar funcionalidades extras, que iremos enumerar de seguida.

4.1 *Parsing* e *Rendering* de ficheiros `.obj`

Com o auxílio da bilbioteca **tinyobjloader**, implementamos a funcionalidade de renderizar qualquer modelo no formato **Wavefront .obj**.

Para já, apenas os vértices destes ficheiros são aproveitados. No futuro, também serão utilizados os vetores normais e as coordenadas de texturas na renderização destes modelos.

Seguem-se exemplos de modelos renderizados em tempo real pelo nosso *renderer*:

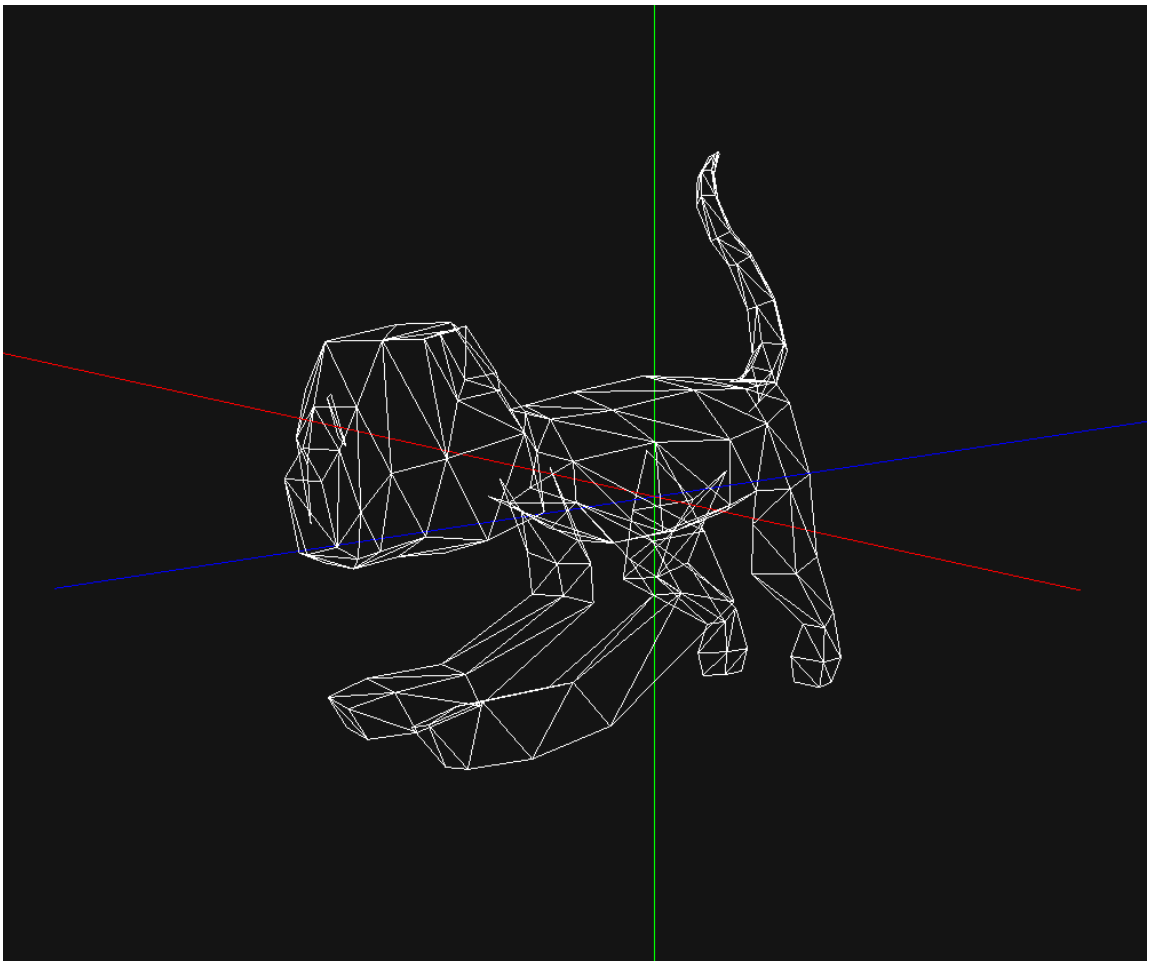

Figura 7. Renderização de um macaco.

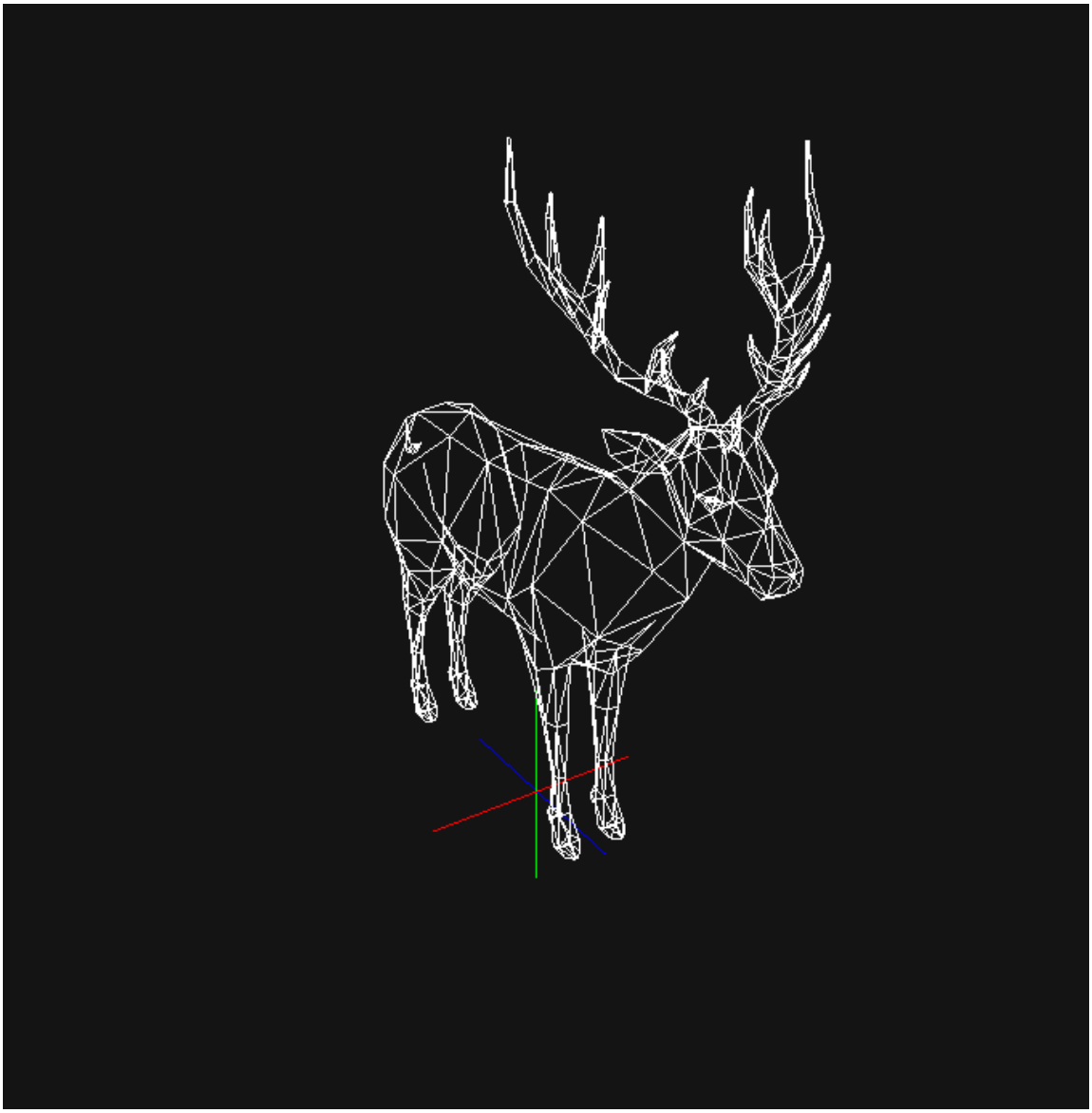

Figura 8. Renderização de um veado.

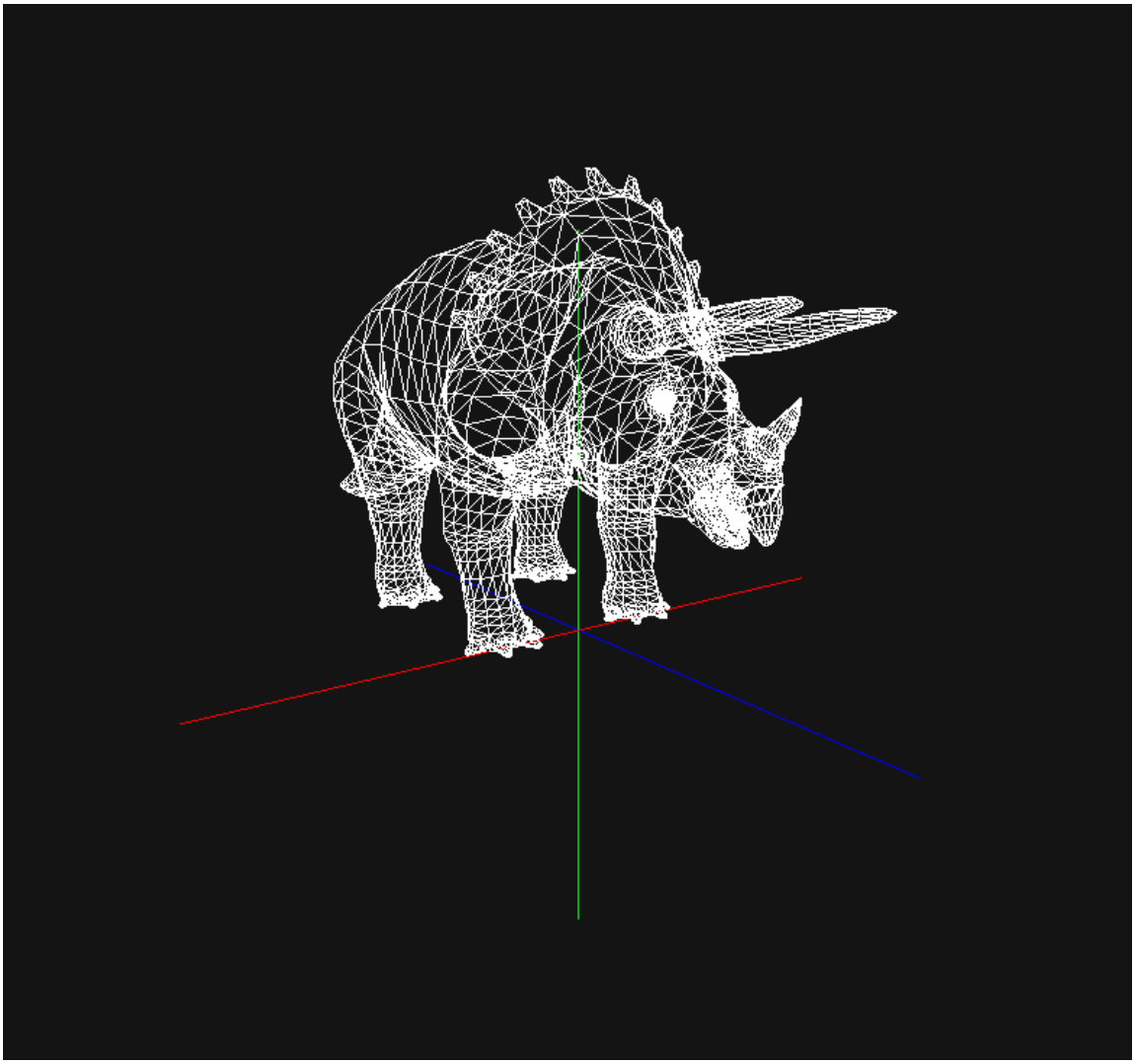

Figura 9. Renderização de um Tricerátops.

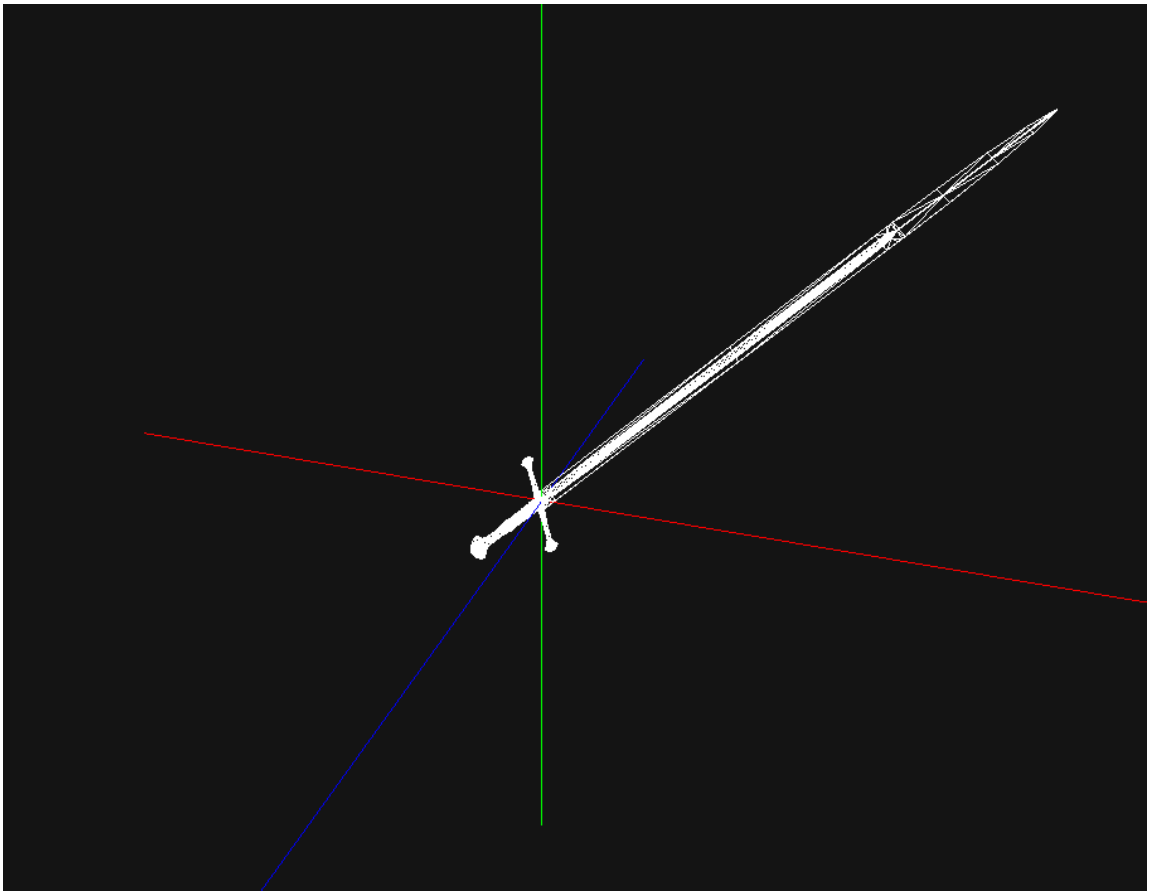

Figura 10. Renderização da espada *Andúril, Flame of the West*.

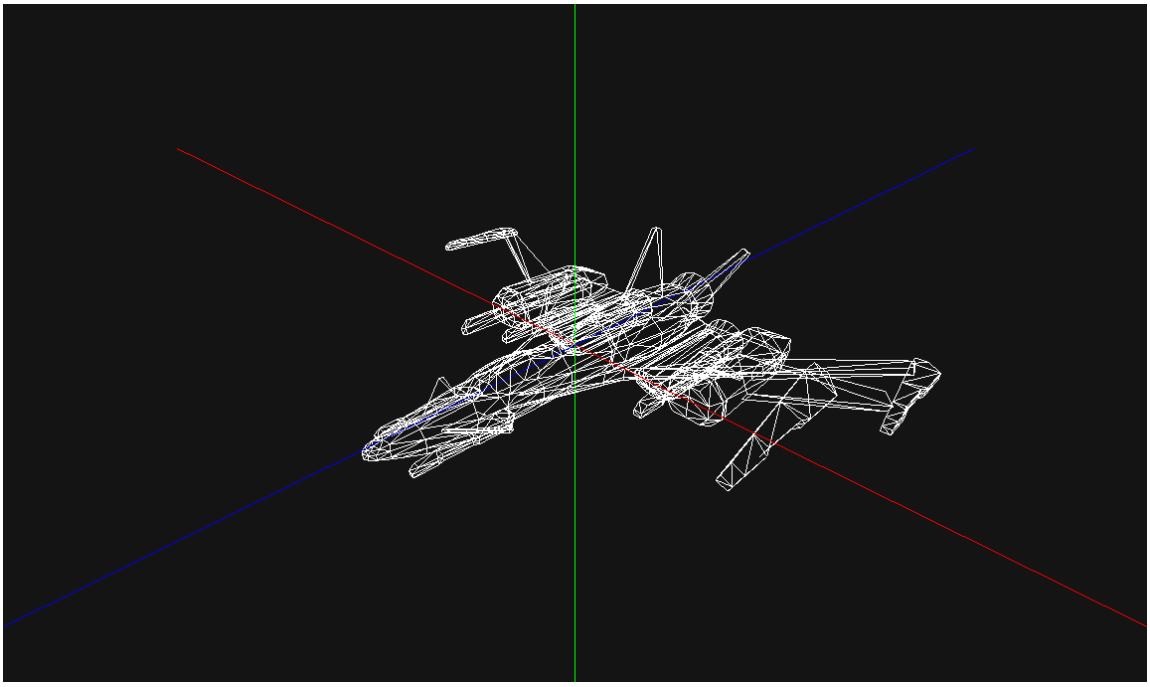

Figura 11. Renderização de uma nave espacial.

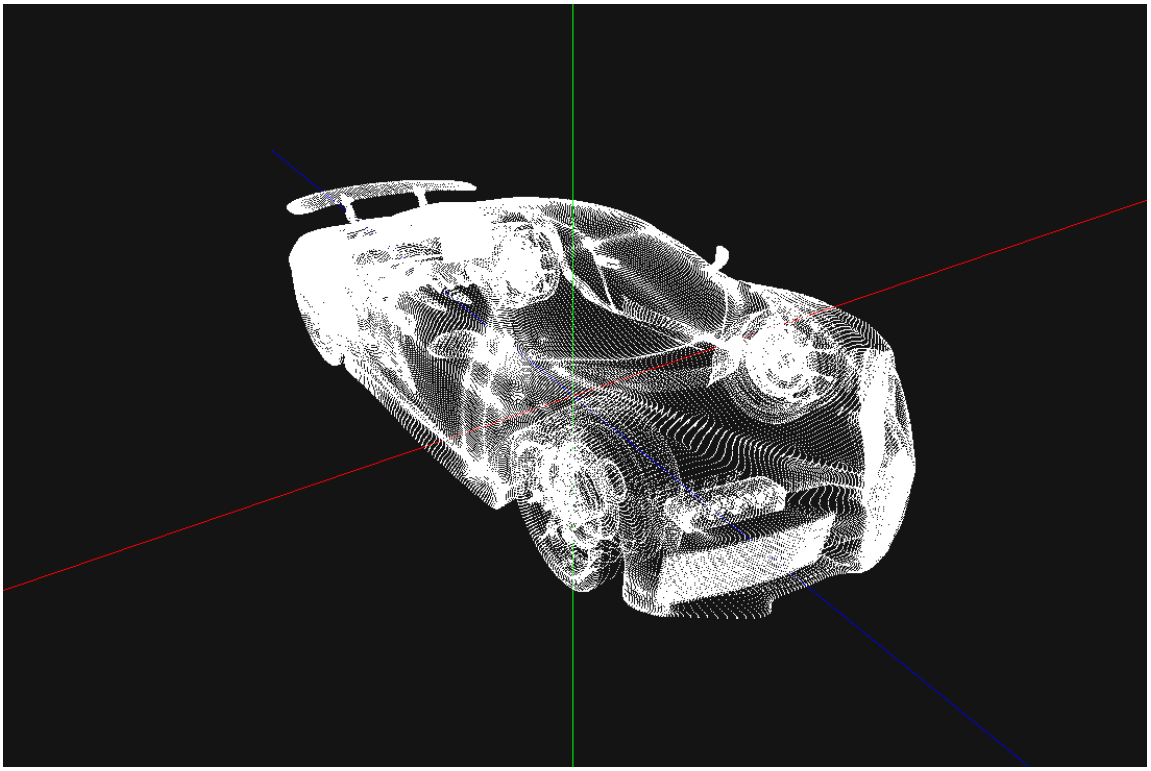

Figura 12. Renderização de um *Bugatti Veyron 2011* (algumas centenas de milhares de vértices).

Eixos x, y, z Como se pode inferir a partir das imagens apresentadas, foi implementada a funcionalidade de opcionalmente mostrar os três eixos. Esta opção é des/ativada com a tecla '**.**'.

Polygon Mode Também foi implementada a opção de alterar o *polygon mode* dos modelos.
A tecla '**m**' passa por cada um dos modos **GL_POINT**, **GL_LINE** e **GL_FILL**.
Por exemplo, o veado que foi apresentado anteriormente é representado da seguinte forma no modo **GL_FILL**:

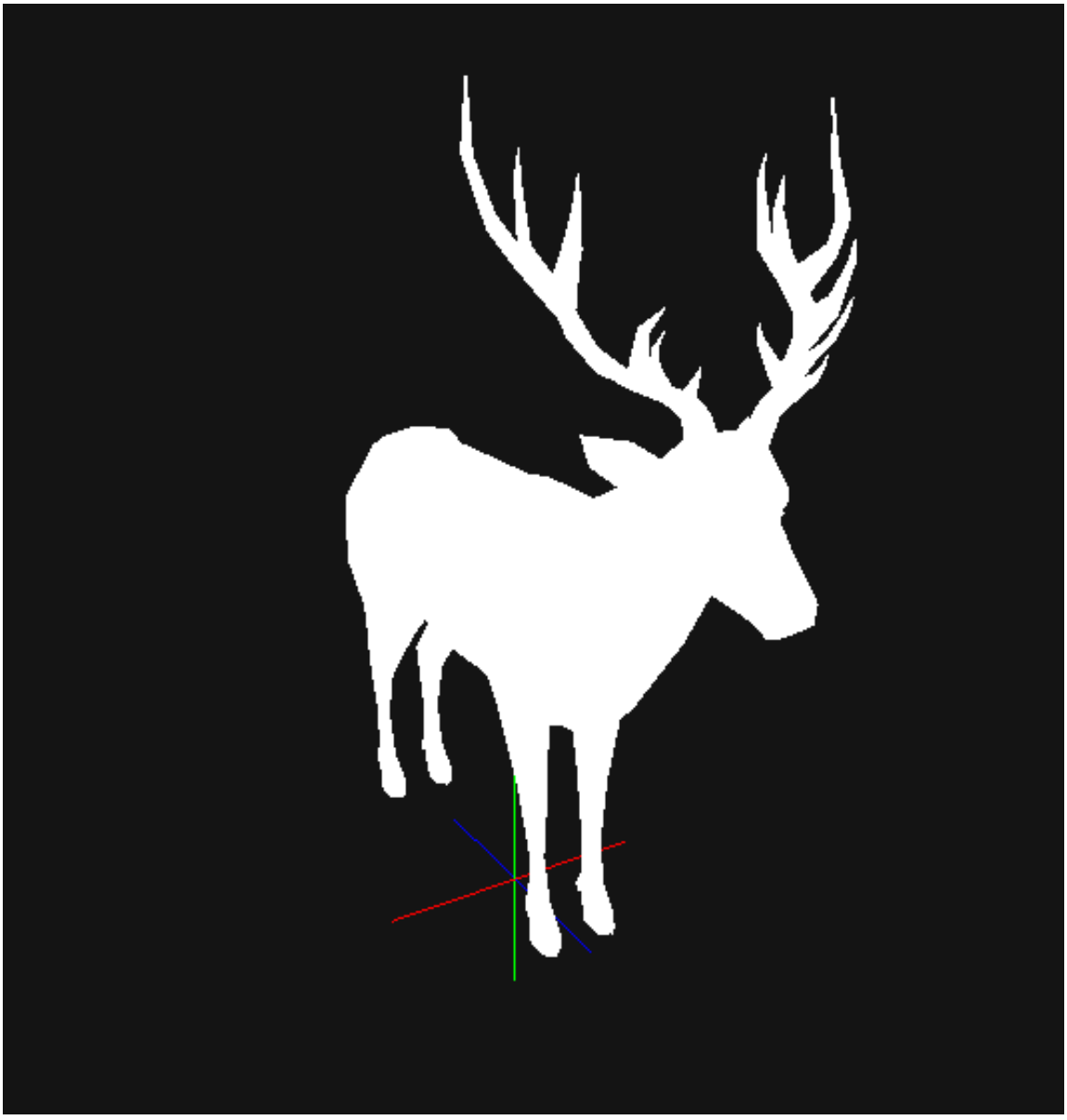

Figura 13. Renderização de um veado em modo **GL_FILL**.

O Tricerátops que foi apresentado anteriormente é representado da seguinte forma no modo **GL_POINT**:

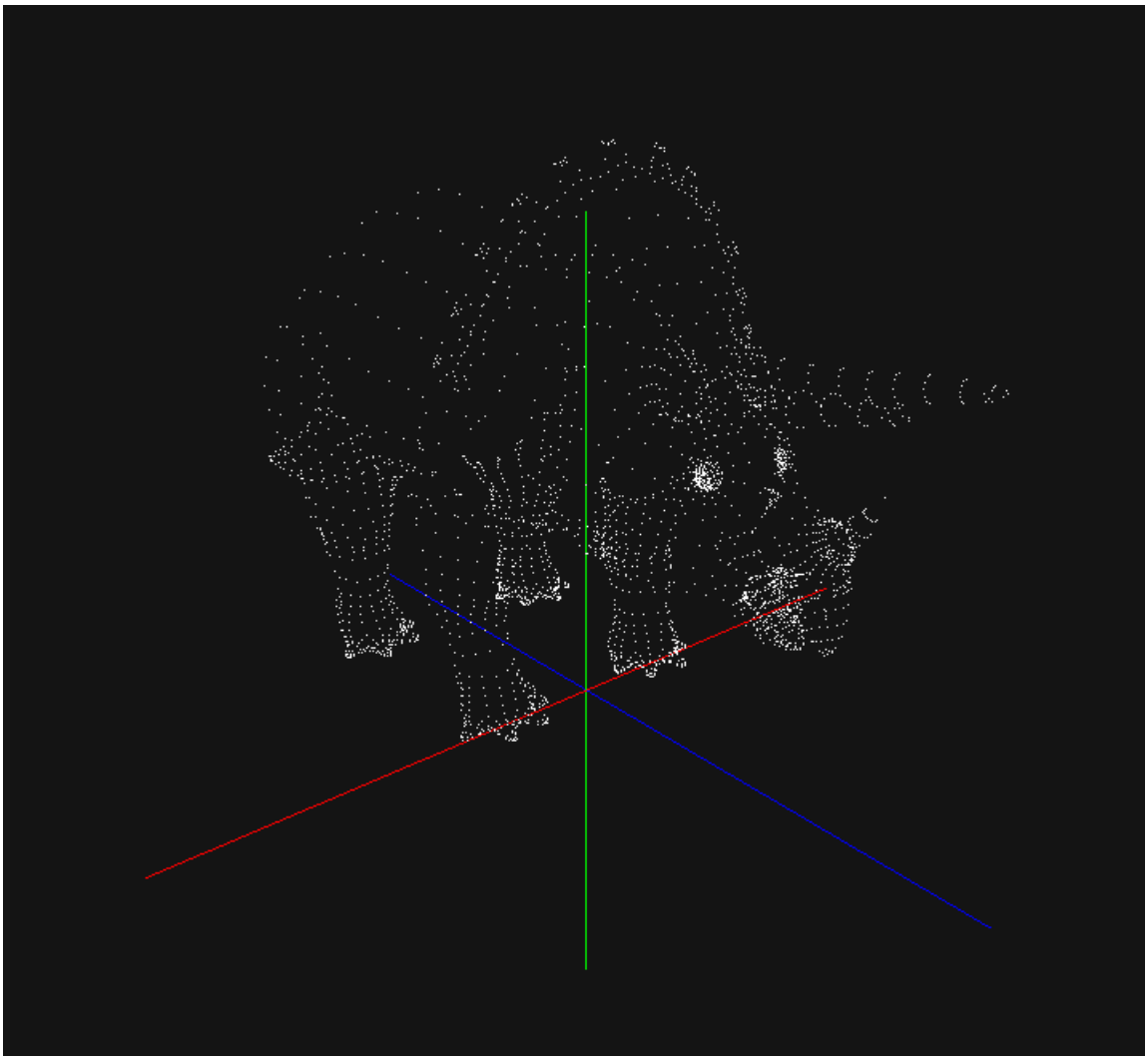

Figura 14. Renderização de um Tricerátops em modo **GL_POINT**.

Espessura das linhas Foi implementada a opção de aumentar e diminuir a espessura das linhas formadas pelos vértices.
A tecla '*****' aumenta a espessura, e a tecla '**_**' diminui. Por exemplo, o nave que foi apresentada anteriormente é representada da seguinte forma com linhas mais espessas:

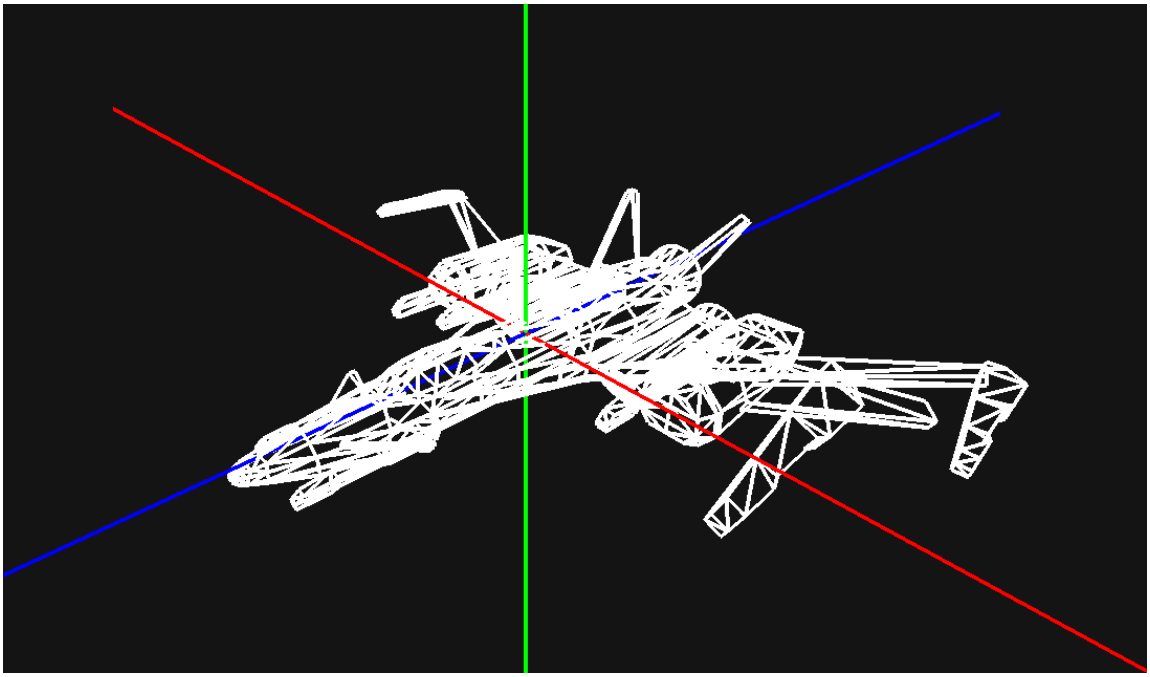

Figura 15. Renderização de uma nave espacial com linhas espessas.

Movimentação da Câmara Embora impossível de demonstrar no formato de relatório, também foi implementada a movimentação da câmara em torno de um ponto fixo. Estes movimentos incluem movimento radial, através das teclas '**WASD**', e *zoom in* e *zoom out*, através das teclas '**E**' e '**Q**', respetivamente.

Na fase anterior, o movimento radial já estava implementado, mas era assumido que o ponto fixo era sempre a origem.
Esta suposição já não é feita.

Para além disso, ocorriam problemas de *flickering* quando a câmara se encontrava no eixo dos y, devido a conversões de coordenadas cartesianas em coordenadas esféricas quando o ângulo polar era aproximadamente **0** ou π.
Este problema foi corrigido impondo um limite mínimo (maior que **0**) e um limite máximo (menor que π) no ângulo polar.

No futuro irá ser implementado o modo *Free Camera*, e será fornecido suporte para movimento com o rato.

5 Conclusão

O projeto e todos os testes foram desenvolvidos em Linux. Várias funcionalidades implementadas tiram partido de bibliotecas standard de C++20 usando o compilador g++-11. Também recorremos às bibliotecas:

- **{fmt}**, que facilita operações de *input/output* com formatação;
- **glm**, que oferece tipos de dados e funções de matemática para OpenGL;
- **rapidxml**, para efetuar a leitura de ficheiros XML;
- **result**, para efetuar *value-based error handling*;
- **speedlog**, para imprimir informação de *logging*;
- **tinyobjloader**, para efetuar a leitura de ficheiros Wavefront .obj.

Finalizando este relatório, podemos afirmar que os problemas propostos para esta fase foram resolvidos com sucesso.

Além do proposto, corrigimos alguns problemas da fase anterior e implementamos funcionalidades extra.

Como trabalho futuro, segue-se a implementação de curvas, superfícies cúbicas e *VBOs*.