Universidade do Minho

LEI 3ºAno 2ºSemestre

Computação Gráfica

Mini Motor 3D

Fase 1 - Primitivas gráficas simples

Daniel Caldas a67691
José Cortez a67716
Marcelo Gonçalves a67736
Ricardo Silva a67728

20 de Março de 2015

Conteúdo

1 Introdução

1.1 Contexto

No âmbito da UC Computação Gráfica, surge um projeto prático para consolidar os conceitos adquiridos ao longo do semestre: a criação de um **mini motor 3D**.

Usando as tecnologias *C++* e *OpenGL*, será então desenvolvido o motor 3D ao longo de quatro fases, sendo que esta primeira está resumida em tópicos na subsecção seguinte.

1.2 Resumo

- Implementação **independente** do motor de primitivas geométricas: **áreas planas**, **paralelepípedos**, **esferas** e **cones**.
- Desenhar arquitetura do motor 3D para cumprir os requisitos funcionais do mesmo.
- Implementar a aplicação a partir do ponto anterior e juntar à mesma as primitivas geométricas do primeiro ponto.
- Desenvolver a interatividade do **motor 3D** (interpretador) de modo a facilitar a chamada de comandos para gerar ficheiros com triângulos, bem como de comandos que desenham uma cena **pré-definida** num ficheiro input no formato *xml*.

2 Planificação da aplicação

A aplicação descrita será composta essencialmente por duas partes: uma primeira - comando **gerador** - que recebe instruções, através dum interpretador, para desenhar formas específicas com parâmetros de entrada à escolha do utilizador. As nossas implementações para formas geométricas seguem os parâmetros do *GLUT*. Posteriormente, é definido **manualmente** um ficheiro *xml* que contém as figuras a ser desenhadas(comando **desenhar**).

Todo este processo pode ser visualizado na figura seguinte:.

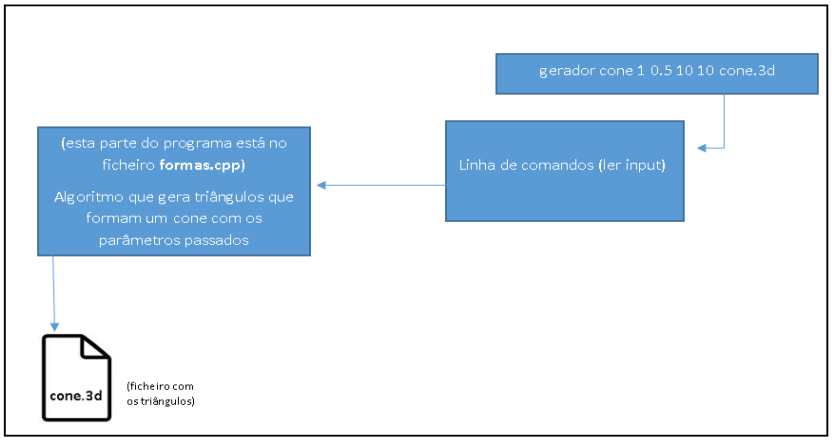

Figura 1: Esquema da primeira parte da aplicação onde são utilizados algoritmos para gerar triângulos e armazená-los em ficheiros.

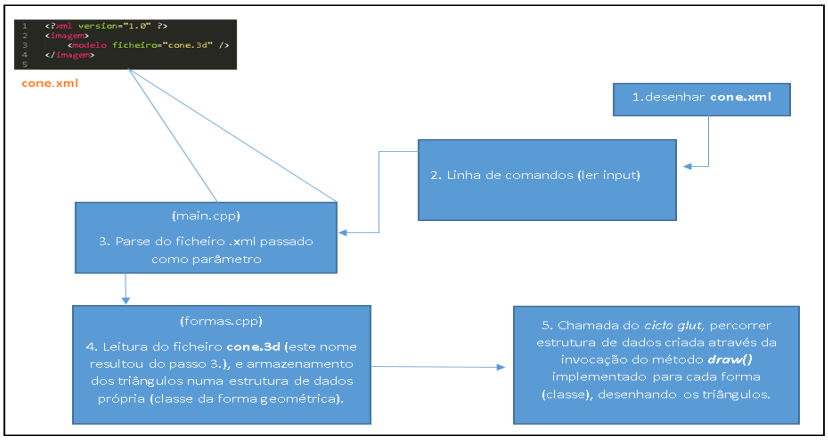

Figura 2: Esquema da segunda parte onde reunimos os dados de forma estruturada dos objetos a incluir na cena e desenhamos a mesma.

3 Arquitetura do código

Após uma análise rigorosa do problema e estudo do fluxo da aplicação, foram analisadas várias hipóteses de implementação, sendo que a solução final resulta numa peça robusta de Engenharia de *Software* que tira proveito do caráter **imperativo** da linguagem *C++* e mais, contempla conceitos da **Programação Orientada aos Objetos** num **módulo** da aplicação, que nos permitiu a criação de estruras de dados elegantes e **extensíveis**.

O nosso projeto é constituído essencialmente por dois ficheiros **.cpp**:

- **main.cpp** - ficheiro que contém uma linha de comandos, parser xml e *skeleton OpenGL* com as funcionalidades clássicas (inicialização, changeSize(), renderScene(), interação com teclado e menus).
- **formas.cpp** - onde estão definidas as estruturas de dados das formas geométricas desta fase do projeto.

3.1 formas.cpp

Esta é a parte que alimenta a aplicação com as funcionalidades principais, portanto vamos detalhar sucintamente o seu conteúdo e estrutura.

3.1.1 Classe Ponto3D

Classe que agrupa três valores que juntos representam um ponto no espaço tri-dimensional. Servirão de base para estruturas mais complexas.

3.1.2 Classe Forma

Uma classe abstrata que representa, **do ponto de vista abstrato, um corpo geométrico amorfo**. Esta classe é formada pelas variáveis:

- **vector<Triangulo> tgls** - um conjunto de pontos que forma uma certa forma geométrica.
- **string nome** - o nome da forma geométrica que toma.

Os métodos:

- **read3DfromFile(string filename)** - método que lê um ficheiro que contêm os triângulos que geram a forma, podendo o ficheiro conter informação adicional (por exemplo, nome da forma).
- **write3DtoFile(string filename)** - Este método, implementado por todas as formas geométricas, gera os vértices dos triângulos da respectiva forma e escreve-os num ficheiro 3D.
- **draw()** - método invocado pelo ciclo *GLUT* que desenha a forma.

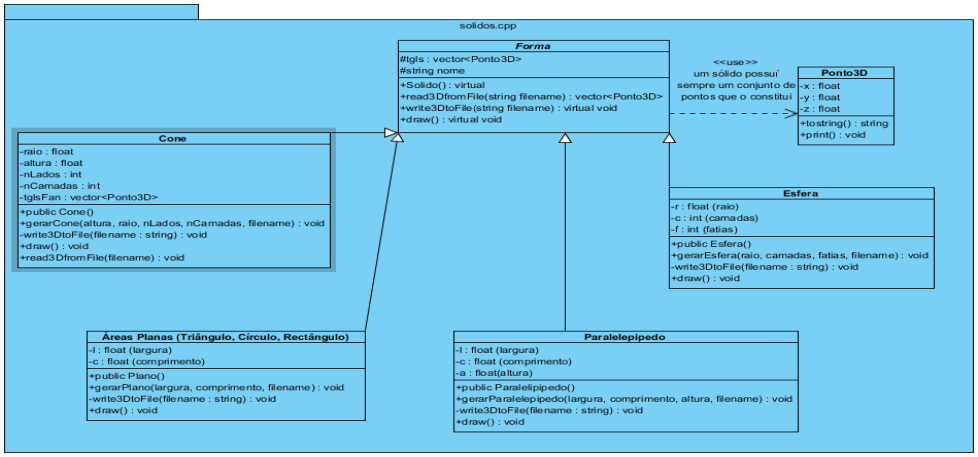

Figura 3: Diagrama de classes que ilustra a solução implementada.

3.1.3 Herdar a classe Forma

Como vimos na secção anterior, existem métodos que classes das formas geométricas terão de implementar em conformidade - i.e. cada forma terá uma implementação diferente dos métodos que escrevem e desenham triângulos - já o método **de leitura** pode ser herdado por defeito da superclasse, visto que por vezes só temos de percorrer um ficheiro e ler **Pontos3D**, armazenando-os num vector, fazendo-se notar já a vantagem da estratégia de implementação.

3.1.4 Aproveitar o polimorfismo

Como todas as estruturas de dados são extensões da classe **Forma**, tiramos proveito do polimorfismo, simplificando a implementação do ciclo *GLUT* que desenha a cena, bem como da manipulação geral das formas geométricas pois todas são tratadas de igual modo. Fornecemos assim uma maneira simples de lidar com a heterogeneidade de formas que uma cena pode incluir!
Na **main.cpp**, todos os objetos a incluír na cena são armazenados na variável global **vector<Forma*> Formas** e, por fim, esta estrutura é percorrida gerando as respectivas figuras.

```
glEnable(GL_CULL_FACE);
glPolygonMode(GL_FRONT_AND_BACK, option);

// Loop para desenhar formas
for (std::vector<Forma*>::iterator it = Formas.begin(); it != Formas.end(); ++it){
    (*it)->draw();
}

glutSwapBuffers();
```

Figura 4: *Loop* que percorre vector de apontadores e desenha cada forma nela contida.

4 Primitivas Geométricas

4.1 Áreas planas

4.1.1 Rectângulo

Um rectângulo é uma superfície plana composta por dois triângulos que partilham dois vértices entre eles, os parâmetros para a sua construção são **l (largura)** e **c (comprimento)**.

Para que se possa observar a face do rectângulo voltada para cima foi ne-

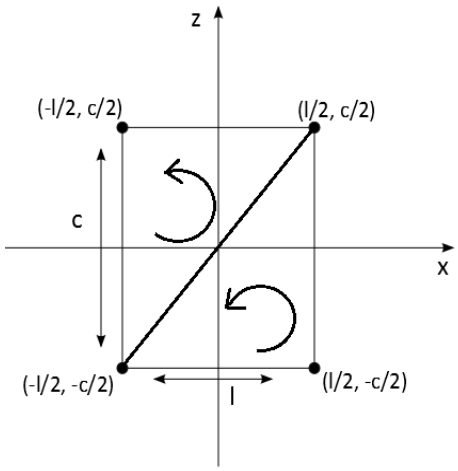

Figura 5: Podem-se ver os pontos e ordem de definição dos mesmo, num **rectângulo**.

cessário gerar os vértices na ordem especificada na figura 5. . Mas como se trata de uma área plana, neste caso um rectângulo, queremos que se possa ver tanto a face superior como inferior da mesma área. Tomemos como exemplo uma **folhas de papel** em que é visível tanto a frete como o verso da folha como duas **superfícies planas uniformes**. Para produzir tal efeito teremos que desenhar os pontos da fig. 5 pela ordem inversa à que foram apresentados. **Este método é aplicado a todas as restantes áreas planas que o nosso motor implementa**.

4.1.2 Círculo

Um círculo é uma superfície plana com centro na origem (0,0,0), cujos parâmetros de entrada são **r (raio)** e **nlados (nº de lados)**, este último parâgrafo é que define a definição do círculo uma vez que teoricamente um círculo (circunferência) é definido por um número infinito de lados, quanto maior for este

último parâmetro melhor definidas serão as arestas da figura.

Para desenhar o círculo damos como ponto comum o centro do referencial e usamos a opção *GL_TRIANGLE_FAN* para a contrução da superfície.

```
file << 0.0f << " " << 0.0f << " " << 0.0f << "\n";
for (int i = 0; i <= nlados; i++){
    file << raio*sinf(alpha) << " " << 0 << " " << raio*cosf(alpha) << "\n";
    alpha -= decAngulo;
}
```

Figura 6: Pedaço de código que gera os vértices de um **círculo**.

4.1.3 Triângulo (Equilátero)

Para a construção de um triângulo equilátero é necessário saber o tamanho do lado. A construção do triângulo equilátero, resume-se á construção de três pontos igualmente distanciados. Apenas existe um parâmetro **lado (o comprimento do lado)**.

```
float h = sinf(M_PI / 3)*(lado / 2); // altura
ofstream file(filename);

file << "TRIANGULO\n";

file << 0.0f << " " << 0.0f << " " << 0.0f << "\n";
file << h << " " << 0.0f << " " << 0.0f << "\n";
file << 0.0f << " " << 0.0f << " " << (-lado / 2) << "\n";

file << 0.0f << " " << 0.0f << " " << 0.0f << "\n";
file << 0.0f << " " << 0.0f << " " << (lado / 2) << "\n";
file << h << " " << 0.0f << " " << 0.0f << "\n";
```

Figura 7: Pedaço de código que gera os vértices de um **triângulo**.

Como podemos observar na fig. 7, os pontos para a formação do triângulos são gerados em torno da origem do referencial e de maneira que se encontrem distanciados conforme a largura pretendida. Os pontos são gerados para um ficheiro, para posterior leitura.

4.2 Paralelepípedo

Um paralelepípedo é um prisma com seis faces, e para a sua construção são necessários três parâmetros: **largura** (l), **comprimento** (c) e **altura**(a).

Para a construção de cada face são necessários dois triângulos e, por conseguinte, seis pontos. Consideramos que o centro do paralelepípedo esta sobre a origem.

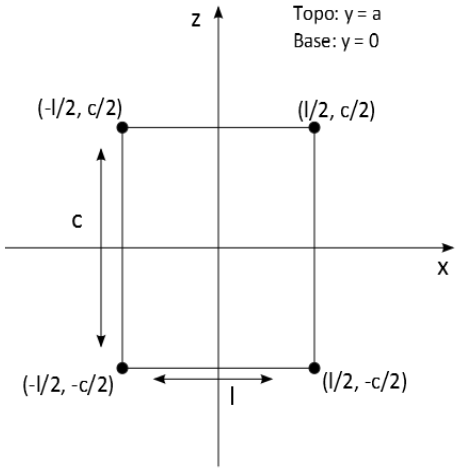

Figura 8: Podem-se ver os pontos genéricos dum paralelepípedo.

4.3 Esfera

Uma esfera é um sólido geométrico formado por uma superfície curva contínua cujos pontos estão equidistantes do centro. A classe esfera possui um conjunto de métodos que nos permitem criar um modelo de uma esfera através de um conjunto de parâmetros.

Para a criação de uma esfera é necessário definir os parâmetros:

- **raio**: raio da base.
- **fatias**: número de lados da esfera, ou seja, deve ser o mais elevado possível, para a superfície ser o mais curva possível.
- **camadas**: número de camadas da esfera.

```
double passoH = (2 * M_PI) / f;
double passoV = (M_PI) / c;
double altura = r* sin((M_PI / 2) - passoV);
double alturaCima = r;

for (i = 0; i < f; i++) {

    double actualX = r*sin(i*passoH);
    double actualZ = r*cos(i*passoH);
    double nextX = r*sin((i + 1)*passoH);
    double nextZ = r*cos((i + 1)*passoH);
    double actX, actZ, nexX, nexZ, cimaActX, cimaActZ, cimaNexX, cimaNexZ;

    for (j = 1; j < c + 2; j++){


        double aux = cos(asin(altura / r));
        actX = actualX * aux;
        actZ = actualZ * aux;
        nexX = nextX * aux;
        nexZ = nextZ * aux;

        aux = cos(asin(alturaCima / r));
        cimaActX = actualX * aux;
        cimaActZ = actualZ * aux;
        cimaNexX = nextX * aux;
        cimaNexZ = nextZ * aux;
```

Figura 9: Pedaço de código que gera vértices da **esfera**.

Na fig. 9 podemos ver a função que gera um ficheiro com os pontos, que definem os triângulos necessários para a criação de uma esfera.

4.4 Cone

Um cone é um sólido geométrico obtido quando se tem uma pirâmide cuja base é um polígono regular, e o **número de lados da base** tende ao infinito. Os parâmetros para gerar um cone são **h (altura)**, **r (raio)**, **nlados (número de lados)** e **ncamadas (número de camadas)**.

```
int i;
float incAngulo = (2 * M_PI) / (float)nlados;
float incAltura = altura / (float)ncamadas;
float incRaio = raio / (float)ncamadas;
float alpha, h;
h = 0; alpha = 2 * M_PI;

file << "CONE\n";
file << altura << "\n";
file << raio << "\n";
file << nlados << "\n";
file << ncamadas << "\n";
file << 0.0f << " " << 0.0f << " " << 0.0f << "\n";
for (i = 0; i <= nlados; i++){
    file << raio*sinf(alpha) << " " << 0 << " " << raio*cosf(alpha) << "\n";
    alpha -= incAngulo;
}

alpha = 0;

for (i = 0; i < ncamadas; i++){
    alpha = 0;
    file << 0 << " " << altura << " " << 0 << "\n";
    for (int j = 0; j <= nlados; j++) {
        file << raio*sinf(alpha) << " " << h << " " << raio*cosf(alpha) << "\n";
        alpha += incAngulo;
    }
    h += incAltura;
    raio = raio - incRaio;
}
```

Figura 10: Pedaço de código que gera vértices de um **cone**.

Para a construção do cone é necessário criar uma base, e o *"tubo"* (parte superior do cone). Na fig. 10 pode ver a função que gera um ficheiro com os pontos, que definem os triângulos necessários para a criação de um cone.

5 Desenhos

5.1 Esfera

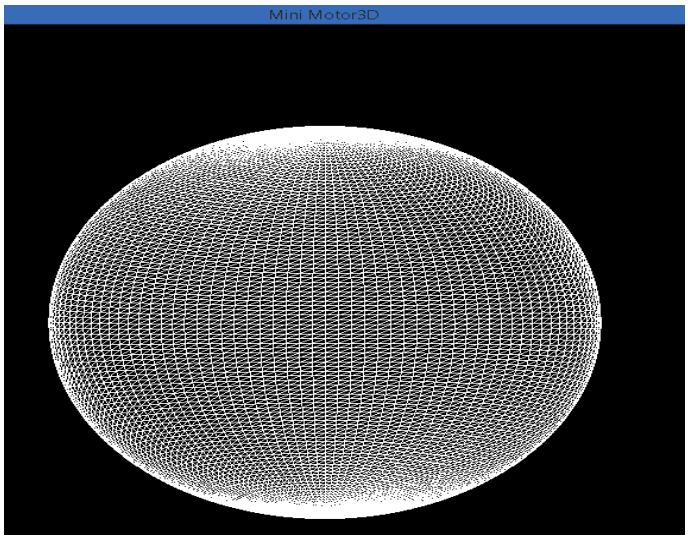

Figura 11: **esfera** com raio 1, 100 camadas e 100 lados.

5.2 Cone

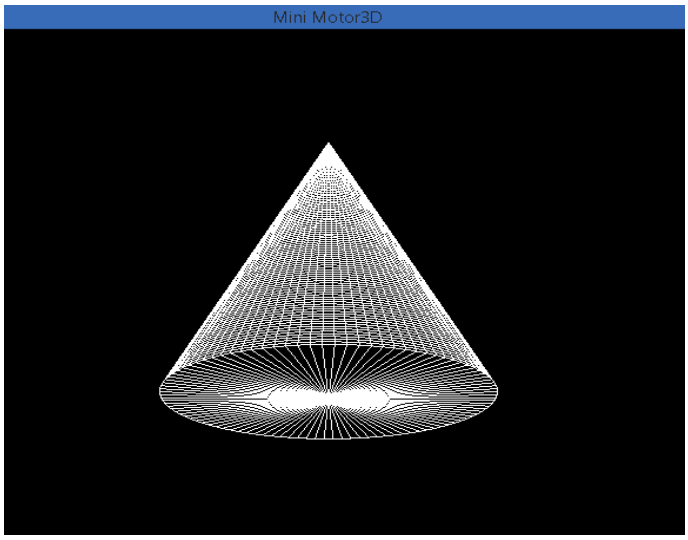

Figura 12: **cone** com altura 1, raio 0.5, 100 lados e 100 camadas.

6 O Motor 3D

6.1 Linha de comandos

Foi desenvolvida uma pequena linha de comandos para acomodar as funcionalidades desejadas, para a manipulação da mesma foi escrito um pequeno **manual** de instruções que facilita o uso de comandos expondo-os de uma forma **explícita**.

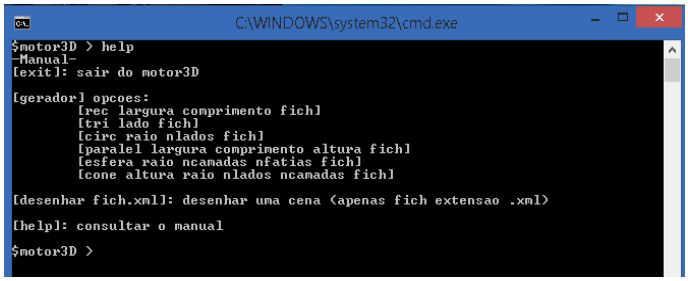

Figura 13: Manual de instruções para utilização do terminal do motor3D.

6.2 Demonstração

Vamos nesta subsecção construír pela linha de comandos, formas para desenhar à *posteriori* numa cena. Invocaremos explicitamente os comandos que geram as formas, e por último desenhamos a cena através da leitura dos ficheiros cujos nomes nos são dados num pequeno *script* em **xml** (**Nota**: o *trigger* destes últimos dois acontecimentos é a invocação do comando **desenhar** cujo parâmetro é o nome do *script* em xml).

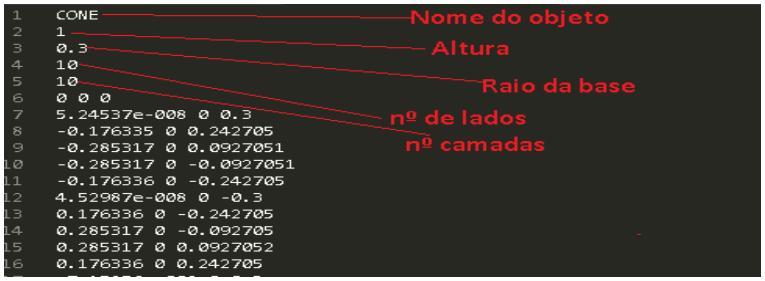

Figura 14: Excerto de um ficheiro que contém os pontos a desenhar para a construção de um **cone**.

Na figura em cima temos um exemplo de um ficheiro que armazena pontos que formam uma figura, neste caso do cone. De seguida veremos qual o comando que gera este ficheiro.

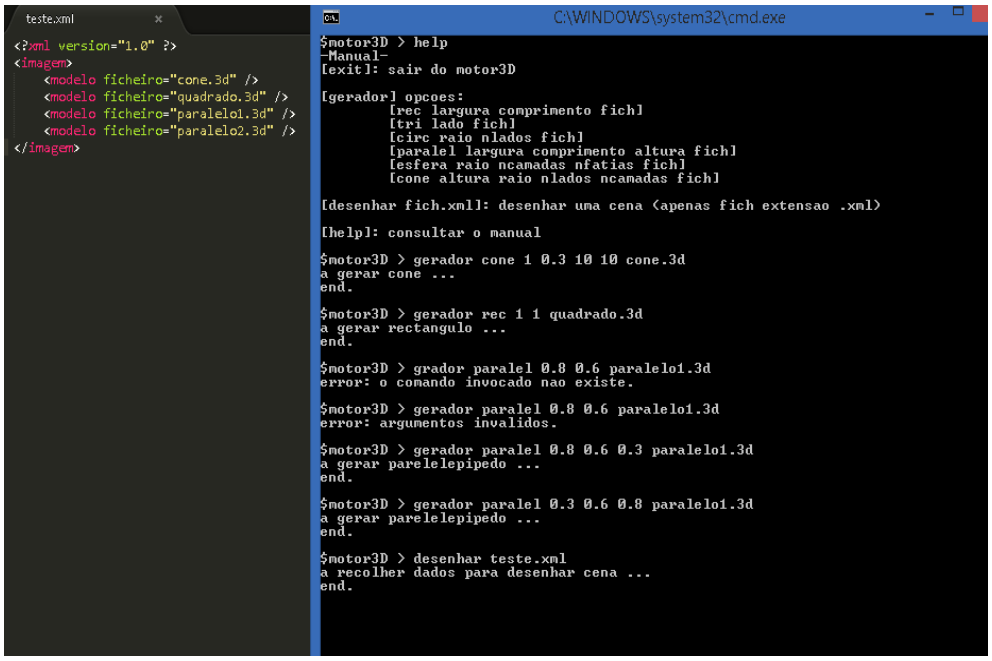

Figura 15: Comandos invocados desde a criação dos objetos até desenhar a cena, ao lado o ficheiro **.xml** (**input**).

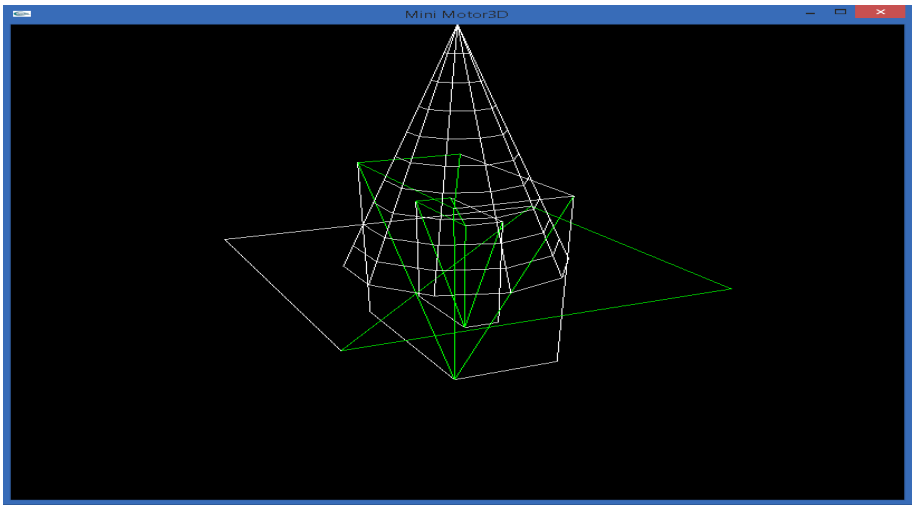

Figura 16: Resultado final, podemos observar um **cone** sobre um **quadrado** e ainda dois **paralelepípedos**.

7 Conclusão

Nesta primeira fase treinamos através da implementação de primitivas gráficas a prespetiva tridimensional, e alguns algoritmos básicos para gerar as mesmas. Construímos a base para o mini motor 3D a desenvolver nas próximas três fases. Estamos agora familiarizados com prototipagem gráfica rápida utilizando a libraria *GLUT* do *OpenGL*.

7.1 Notas para trabalho futuro

Prevemos que a arquitetura será essecial para nas seguintes fases do desenvolvimento do motor 3D, portanto contruímos o mesmo de forma a que seja passível de extensão às novas funcionalidades pretendidas num futuro próximo.