# Minicurso de OpenGL

Valério N. R. Júnior

25 de outubro de 2020

# Sumário

# Capítulo 1

# Introdução

## 1.1 Apresentação do curso

### 1.1.1 Objetivo

O objetivo deste curso é aplicar os conceitos de computação gráfica utilizando a biblioteca OpenGL. Como resultado, você terá um programa que renderiza uma cena semelhante à da figura 1.1, que pode servir de apoio para o desenvolvimento do projeto final da disciplina.

### 1.1.2 Metodologia

O curso foi pensado para ser ministrado em duas partes: *Modelagem e transformações* e *Iluminação*. Na primeira, será ensinado:

- como funciona um programa com OpenGL
- modelagem de objetos utilizando primitivas
- como interagir com o programa

Já na segunda parte, o foco será na apresentação visual, adicionando iluminação e texturas. O código será escrito na linguagem C, e o todo o código utilizado será disponibilizado[1].

### 1.1.3 Materiais recomendados

- The Official Guide to Learning OpenGL

[1] Plágio é crime

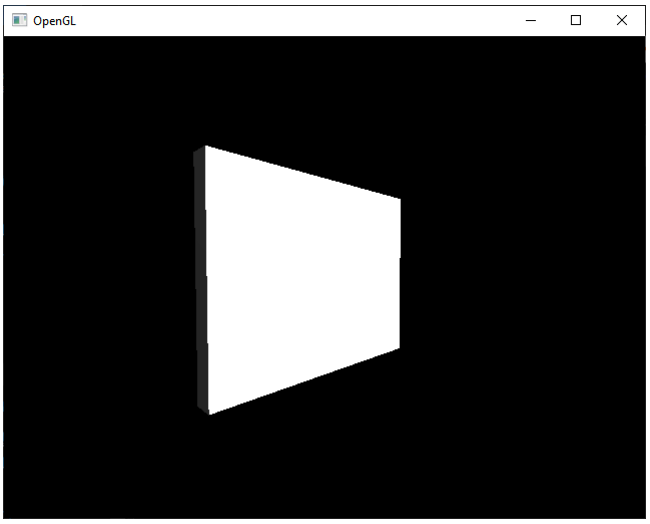


Figura 1.1:

- OpenGL Wiki
- https://learnopengl.com/
- How Rendering Graphics Works in Games!
- How Lighting (Basically) Works in Games
- Canal GuerrillaCG no YouTube

## 1.2 Instalação

### 1.2.1 *OpenGL*

O *OpenGL* já vem instalado nos três principais sistemas operacionais atuais (*Windows*, *Linux*, *MacOS*). Certifique que seus *drivers* estão atualizados.

### 1.2.2 *freeglut*

Para criar e gerenciar a janela da aplicação iremos utilizar a biblioteca *freeglut*. Se preferir, você pode utilizar qualquer outra que suporte *OpenGL* (*Allegro*, *SFML*, *SDL* etc).

***Windows***

Os arquivos já compilados podem ser baixados no endereço `http://www.transmissionzero.co.uk/software/freeglut-devel/`. Extraia os arquivos e guarde-os em um local de fácil acesso no seu computador.

**Linux**

Instale os seguintes pacotes:

```
sudo apt-get install freeglut3 freeglut3-dev libglew-dev
sudo apt-get install mesa-utils
```

## 1.3 Criando de uma janela

Antes de começar o projeto, vamos escrever um programa que abre uma janela para termos certeza que conseguimos linkar as bibliotecas corretamente. Crie um arquivo chamado *test.c* e cole o código da listagem 1.1.

```
1  #include <GL/gl.h>
2  #include <GL/glut.h>
3
4  #define WINDOW_WIDTH 640
5  #define WINDOW_HEIGHT 480
6
7  /*
8  Aqui faremos as operacoes de renderizacao
9  */
10 void display()
11 {
12 }
13
14 int main(int argc, char** argv)
15 {
16     // Inicializacao
17     glutInit(&argc, argv);
18     glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
19     glutInitWindowSize(WINDOW_WIDTH, WINDOW_HEIGHT);
20     glutCreateWindow("OpenGL");
21
22     // Registra o callback de renderizacao (por enquanto nao
       faz nada)
23     glutDisplayFunc(display);
24
25     // Inicia o loop de eventos da GLUT
26     glutMainLoop();
27
```


Figura 1.2: Janela aberta

```
28      return 0;
29  }
```

Listagem 1.1: Código básico para abrir uma janela

Para gerar o arquivo executável precisamos linkar o código compilado com o *OpenGL* e com a *freeglut*:

### 1.3.1 *Windows (MinGW)*

```
gcc test.c -o test.exe -I"<caminho/para/freeglut/include>"
-L"<caminho/para/freeglut/lib>" -lfreeglut -lopengl32
```

Copie o arquivo *freeglut.dll* da pasta *freeglut/bin* para o mesmo local de *test.exe* e execute o programa.

### 1.3.2 *Linux*

```
gcc test.c -o test.out -lfreeglut -lGL
```

Uma janela como a da figura 1.2 deve ser mostrada.

# Capítulo 2

# Modelagem e transformações

## 2.1 Desenhando o primeiro triângulo

Em computação gráfica, podemos representar objetos através de malhas de polígonos, que por sua vez, podem ser representados por triângulos. Nesta seção veremos como renderizar um triângulo na tela e alterar sua aparência utilizando cores e transformações. Utilizaremo como base o mesmo código da seção 1.3, mas dessa vez implementaremos a função *display*.

O primeiro passo é definir as coordenadas dos vértices do triângulo. Por padrão, as coordenadas devem estar no intervalo $[-1, 1]$. Adicione o código da listagem 2.1 globalmente. O *array* `vertices` contém as coordenadas dos três vértices do triângulo.

```
1 float vertices[] = {
2     -0.5f, -0.5f, 0.0f,
3     0.0f, 0.5f, 0.0f,
4     0.5f, -0.5f, 0.0f
5 };
```

Listagem 2.1: Definindo os vértices do triângulo

Todos objetos serão desenhados utilizando primitivas geométricas. O *OpenGL* disponibiliza vários tipos de primitivas. As utilizadas nesse curso serão:

- `GL_POINTS`: desenha pontos
- `GL_LINES`: desenha linhas
- `GL_TRIANGLES`: desenha triângulos
- `GL_QUADS`: desenha quadriláteros

- `GL_POLYGON`: desenha polígonos

Observe que apenas `GL_POLYGON` não está no plural. Isso porque o desenho com primitivas funciona da seguinte maneira: os vértices e as informações utilizadas na renderização (cor, vetor normal, coordenadas de textura etc) de cada um são passados um por um para a *GPU*. Por exemplo, se o objeto é formado por triângulos, podemos utilizar a primitiva `GL_TRIANGLES` e passar todos os vértices da malha e o *OpenGL* saberá que deverá desenhar triângulos utilizando os vértices passados de três em três. A listagem 2.2 mostra como isso pode ser utilizado para renderizar nosso triângulo.

```
// limpa a tela
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// sinaliza para o OpenGL que queremos desenhar triangulos
glBegin(GL_TRIANGLES);

// coordenadas x, y e z do primeiro vertice
glVertex3f(vertices[0], vertices[1], vertices[2]);
// coordenadas x, y e z do segundo vertice
glVertex3f(vertices[3], vertices[4], vertices[5]);
// coordenadas x, y e z do terceiro vertice
glVertex3f(vertices[6], vertices[7], vertices[8]);

// sinaliza para o OpenGL que terminamos o desenho do objeto
glEnd();

// Troca os buffers
glutSwapBuffers();
```

Listagem 2.2: Desenhando o triângulo com a primitiva `GL_TRIANGLES`

Compile e execute o programa. O resultado esperado é mostrado na figura 2.1.

## 2.2 Definindo o vértice

Embora seja perfeitamente aceitável utilizar um *array* de *floats* para definir os vértices e suas propriedades, quanto mais informações forem adicionadas mais ilegível ficará o código. Para simplificá-lo, definiremos um novo tipo `Vertex` que conterá todas as informações necessárias para a renderização.

```
typedef struct Vertex
{
    float x, y, z; // posicao do vertice
    float r, g, b; // cor do vertice
```

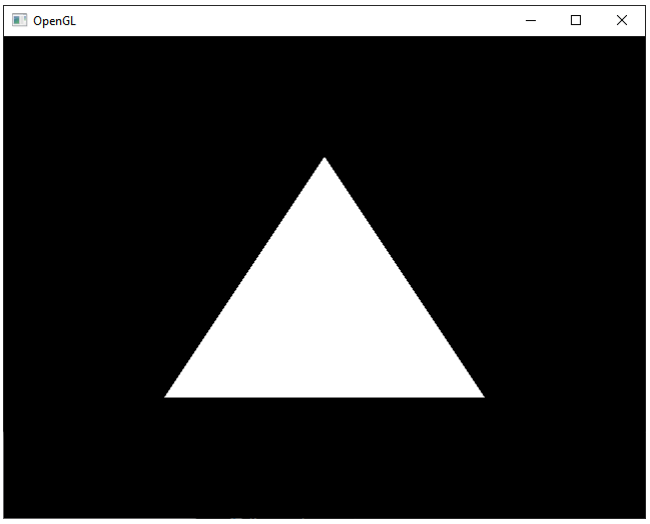


Figura 2.1: Triângulo renderizado com a primitiva `GL_TRIANGLES`

```
} Vertex;
```

Listagem 2.3: Definição da estrutura `Vertex`

Desse modo, ao invés de `float vertices[] = {...}` teremos `Vertex vertices[] = {...}`, organizando o código e facilitando a iteração.

## 2.3 Colorindo o triângulo

Como você pôde observar na seção 2.1, o triângulo desenhado era completamente branco. Isso aconteceu porque o *OpenGL* é uma máquina de estados. Ao desenhar uma primitiva, a cor utilizada nos vértices é lida do estado interno do *OpenGL*, e por padrão é branca. Desse modo, para alterar a cor de um vértice basta alterarmos o estado interno, como na listagem 2.4

```
glColor3f(1.0f, 0.0f, 0.0f); // vermelho
glVertex3f(0.0f, 0.0f, 0.0f);
```

Listagem 2.4: Mudando a cor dos vértices. Após a chamada da função `glColor3f` qualquer desenho de primitiva usará a cor vermelha

Utilizando a estrutura de vértice da seção 2.2, podemos adicionar uma cor diferente para cada vértice. Modifique a renderização do triângulo de acordo com a listagem 2.5.

```
// ...
Vertex vertices[] = {
    {-0.5f, -0.5f, 0.0f, 1.0f, 0.0f, 0.0f},
```


Figura 2.2: Triângulo colorido

```
4       {0.0f, 0.5f, 0.0f, 0.0f, 1.0f, 0.0f},
5       {0.5f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f}
6   }
7
8   // ...
9
10  void display()
11  {
12      int i;
13      Vertex v;
14
15      // ...
16
17      for(i = 0;i < sizeof(vertices);i++)
18      {
19          v = vertices[i];
20          glColor3f(v.r, v.g, v.b);
21          glVertex3f(v.x, v.y, v.z);
22      }
23
24      // ...
25  }
26
27  // ...
```

Listagem 2.5: Implementação da função `diplay` com a estrutura `Vertex`

O resultado é mostrado na figura 2.2.

## 2.4 Transformações

### 2.4.1 Sistemas de coordenadas

Um sistema de coordenadas é um sistema que associa uma *n-upla* de escalares a cada ponto num espaço. Basicamente é definido por um ponto, a origem, e uma orientação para servir de referência. Alguns exemplos são as coordenadas cartesianas, cilíndricas, esféricas etc. A seguir são apresentados os sistemas de coordenadas utilizados na computação gráfica no geral, mas mais especificamente no *pipeline* fixo do *OpenGL*.

#### Espaço local

O espaço local é o espaço de coordenadas do objeto, um ponto é definido em relação à origem individual do objeto. Geralmente é aqui que acontece a modelagem dos objetos, pois assim não precisamos nos preocupar com o posicionamento final dele na cena.

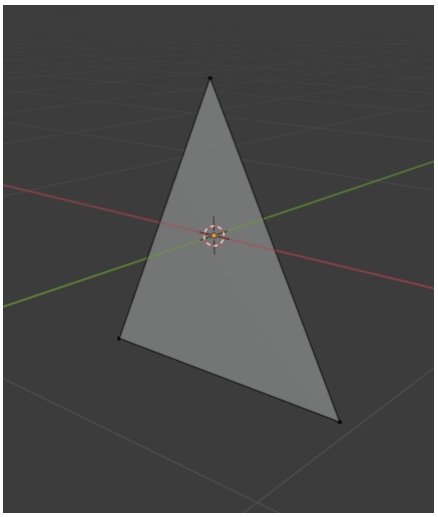

Figura 2.3: Espaço local

#### Espaço global

O espaço global é o espaço de coordenadas do mundo, isto é, qualquer vértice de qualquer objeto é definido em relação à mesma origem. Geralmente usamos as coordenadas do mundo para posicionar os objetos.

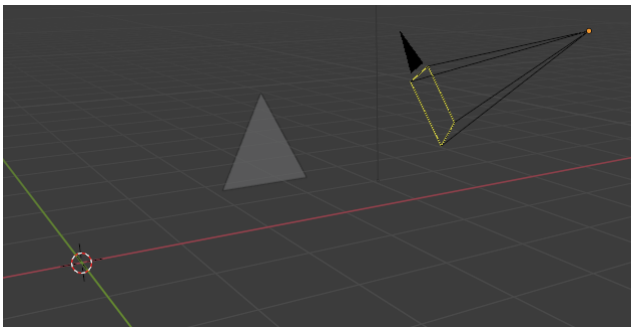

Figura 2.4: Espaço global. Podemos ver a origem da cena no canto inferior esquerdo, o triângulo no centro e a câmera no canto superior direito

### Espaço da câmera

Na computação gráfica, é comum utilizar a analogia de que o que é renderizado na tela é visto a partir de uma câmera virtual. No entanto, o que realmente acontece é que a cena inteira é transladad/rotacionada de acordo com a posição/rotação da câmera. É o mesmo efeito que acontece ao olhar para fora de um carro: temos a ilusão que estamos parados e o ambiente está se movendo.

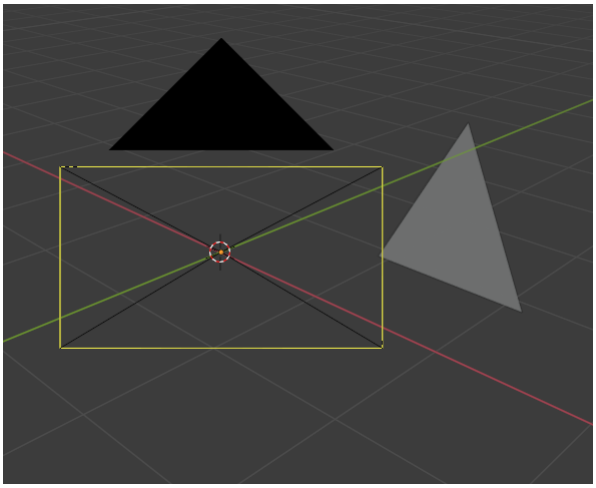

Figura 2.5: Espaço da câmera

### Espaço de recorte

O espaço de recorte é obtido a partir da projeção do espaço da câmera. Essa projeção pode ser de dois tipos: ortográfica e perspectiva. Na ortográfica

a distância de um objeto da "câmera"não altera o seu tamanho, por isso essa projeção geralmente é usada quando estamos renderizando uma cena bidimensional. Já a projeção perspectiva corrgige o tamanho dos objetos de acordo com a distância, dando a sensação de profundidade.

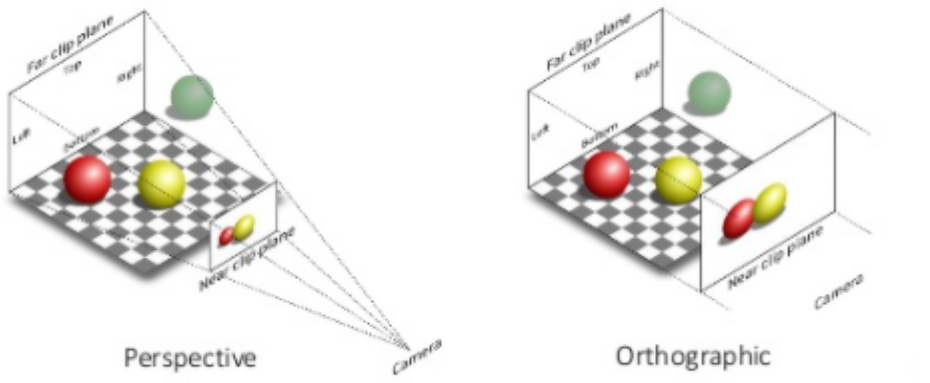


Figura 2.6: Projeção ortográfica vs. projeção perspectiva. Retirado de `https://image.slidesharecdn.com/presentation-161031003006/95/enei16-webgl-with-threejs-44-638.jpg?cb=1477873857`

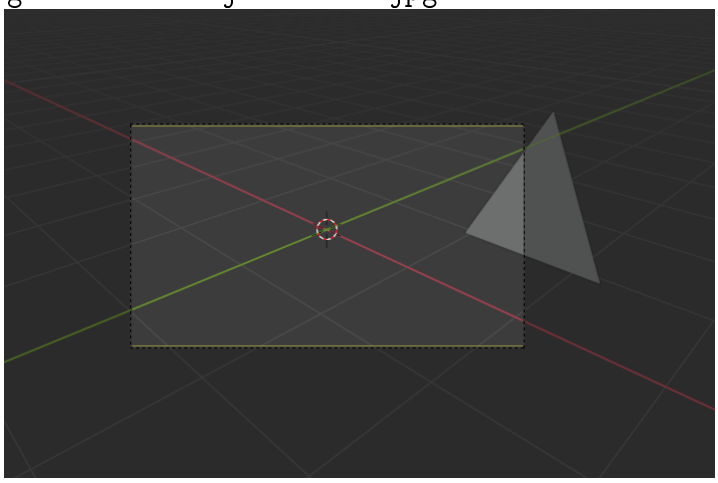

Figura 2.7: Espaço de recorte, utilizando uma projeção perspectiva. A área mais escura será cortada da imagem final

**Espaço da tela**

O espaço da tela, como o próprio nome sugere, é o espaço de coordenadas da tela do dispositivo. No *OpenGL* essa etapa é dividida em duas. Primeiramente, as coordenadas são passadas para um espaço padrão chamado *NDC*(*Normalized Device Coordinates*) e depois para as dimensões verdadeiras da tela.

Figura 2.8: Espaço da tela

Na próxima seção aprenderemos como passar de um sistema de coordenados para outro.

### 2.4.2 Matrizes e transformações lineares

Em álgebra linear, aprendemos que toda transformação linear do $\mathbb{R}^n$ possui uma matriz $nxn$ associada. Também vemos que transformações lineares podem ser compostas, basta multiplicar suas matrizes e teremos uma transformação resultante que é equivalente à aplicação delas sequencialmente. A seguir são apresentadas as principais transformações utilizadas na computação gráfica.

### Translação

A translação por um vetor $(d_x, d_y, d_z)$ pode ser obtida com a matriz da equação abaixo:

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + d_x \\ y + d_y \\ z + d_z \\ 1 \end{bmatrix} \tag{2.1}$$

Em *OpenGL*, o comando para realizar essa transformação é:

```
glTranslatef(dx, dy, dz);
```

### Rotação

No espaço 3D, as rotações geralmente são representadas por um vetor $(pitch, yaw, roll)$, onde $pitch$, $yaw$ e $roll$ são a rotação em torno do eixo $x$, a rotação em torno do eixo $y$ e a rotação em torno do eixo $z$, respectivamente. O Teorema de Euler diz que qualquer rotação 3D pode ser representado por um ângulo $\alpha$ e um eixo especificado por um vetor unitário $\vec{v}$, onde a rotação será de $\alpha$ em torno de $\vec{v}$. A seguir são apresentadas as matrizes para as rotações em torno de cada eixo cartesiano ($x$, $y$ e $z$).

Rotação em torno do eixo $x$:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y\cos\theta - z\sin\theta \\ y\sin\theta + z\cos\theta \\ 1 \end{bmatrix} \tag{2.2}$$

Rotação em torno do eixo $y$:

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x\cos\theta + z\sin\theta \\ y \\ -x\sin\theta + z\cos\theta \\ 1 \end{bmatrix} \tag{2.3}$$

Rotação em torno do eixo $z$:

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \\ z \\ 1 \end{bmatrix} \tag{2.4}$$

Em *OpenGL*, a função utilizada para realizar rotações no OpenGL recebe um ângulo em graus e as coordenadas $x, yez$ do eixo de rotação:

```
glRotatef(angle, axis_x, axis_y, axis_z);
```

### Escala

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} s_x * x \\ s_y * y \\ s_z * z \\ 1 \end{bmatrix} \tag{2.5}$$

### Transformações no *OpenGL*

Para aplicar transformações de translação, rotação ou escala no *OpenGL* utilizamos as funções `glTranslatef`, `glRotatef` e `glScalef`, respectivamente. Internamente, o *OpenGL* armazena algumas transformações na forma de matrizes. Portanto, antes de chamar as funções precisamos dizer em qual matriz estamos realizando a transformação. O comando que faz isso é:

```
1 glMatrixMode(matrix_mode);
```

onde `matrix_mode` pode ser `GL_PROJECTION` ou `GL_MODELVIEW`[1]. A primeira diz que queremos modeificar a matriz de projeção, que é usada na passagem do espaço da câmera para o espaço de recorte. A segunda, é utilizada para passar do espaço local diretamente para o espaço da câmera (a matriz `MODELVIEW` é equivalente à multiplicação da matriz *Model*, que passa do espaço local para o global, e a matriz *View*, que passa do espaço global para o da câmera). Juntas, as matrizes *Model*, *View* e *Projection* formam a matriz *MVP*. Portanto, os passos a serem seguidos são:

- Definir a matriz de projeção (*Projection*)
- Definir a matriz de visualização (*View*)
- Definir a matriz do modelo (*Model*)

Você deve ter percebido que a ordem dos passos descritos foi a inversa do que foi falado no texto anteriormente. Isso acontece porque a multiplicação de matrizes é feita pela esquerda, ou seja, a ordem dos fatores deve ser o inverso do que queremos. Vale ressaltar também que todas as transformações são realizadas em relação à origem $(0, 0, 0)$, por isso a ordem delas afeta a posição final dos objetos, como na figura 2.9.

[1] Há também os valores `GL_TEXTURE` e `GL_COLOR`, porém não utilizaremos eles neste curso (com exceção de `GL_TEXTURE`, mas com outro uso)

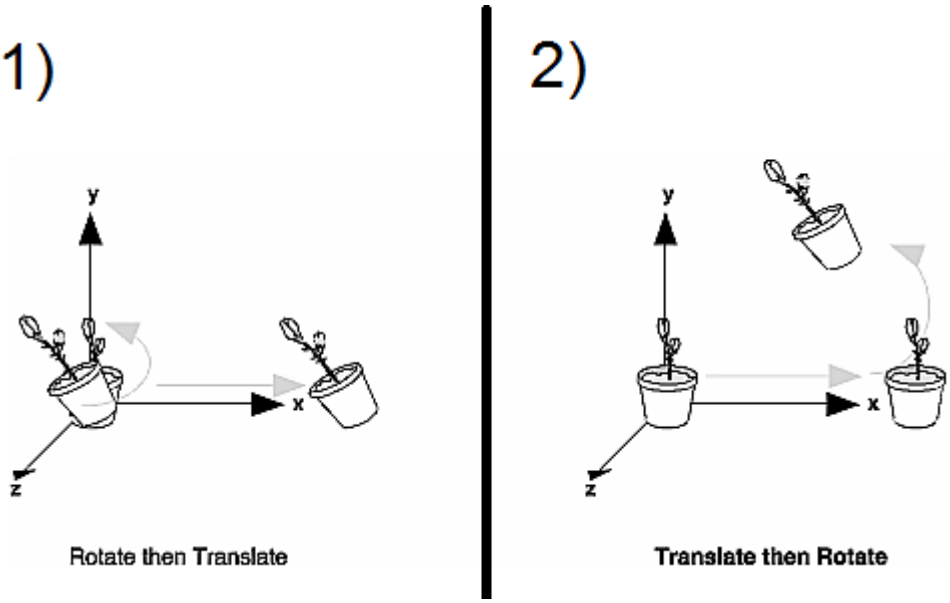


Figura 2.9: Ordem das transformações. Retirado de `https://i.stack.imgur.com/LQtiX.png`

**Projeção** Para gerar a matriz de projeção existem duas funções da biblioteca *GLU*: `gluOrtho` para a projeção ortográfica e `gluPerspective` para a projeção perspectiva. Adicione o seguinte código após a inicialização:

```
// ...
fov_y = 75.0f; // 75 graus
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(fov_y, 1.0f * WINDOW_WIDTH / WINDOW_HEIGHT,
    0.001f, 1000.0f);
// ...
```

onde `fov_y` é o campo de visão vertical, em graus.

**Visualização e Modelo** A matriz *ModelView* deve ser definida em duas partes: uma para a matriz de visualização, e depois a matriz de modelo para cada objeto renderizado. Lembre-se que o estado anterior é mantido e como as funções de transformação multiplicam a matriz atual você sempre deve carregar a matriz identidade para ter os resultados desejados.

```
// ...

// Variaveis para controlar a camera
float fov_y;
float cam_x, cam_y, cam_z;
```

```
6  float center_x, center_y, center_z;
7
8  // Variaveis para posicionar a parede
9  float parede_x, parede_y, parede_z; // posicao da parede
10 float parede_rotacao; // rotacao da parede
11 float parede_largura, parede_altura, parede_espessura; //
       dimensoes da parede
12
13 // ...
14
15 void display()
16 {
17     // ...
18
19     // Seleciona a matriz ModelView e reseta todas as
       transformacoes
20     glMatrixMode(GL_MODELVIEW);
21     glLoadIdentity();
22
23     // Define a matriz View
24     gluLookAt(cam_x, cam_y, cam_z, center_x, center_y,
       center_z, 0.0f, 1.0f, 0.0f);
25
26     // Define a matriz Model
27     /*
28     * Lembre-se, a ordem das transformacoes eh o inverso de
29     * como aparecem no codigo, por isso aplicamos a escala,
30     * depois as rotacoes e finalmente a translacao
31     */
32     glTranslatef(parede_x, parede_y, parede_z);
33     glRotatef(parede_rotacao, 0.0f, 1.0f, 0.0f); // rotacao
       em torno do eixo y
34     glScalef(parede_largura, parede_altura, parede_espessura)
       ; // como o cubo tem lado 1 podemos escalar ele para as
       dimensoes desejadas
35
36     /*
37     * Desenha o cubo de lado 1 com todas as transformacoes
       acima
38     * aplicadas.
39     * A origem do cubo eh no centro, entao lembre de colocar
40     * altura/2 na coordenada y da posicao para ficar
41     * perfeitamente no chao
42     */
43
44     glutSolidCube(1.0f);
45
46     // ...
47 }
```

Você pode definir a posição, rotação e dimensões como quiser. Neste exemplo, a parede terá 15cm de espessura, altura de 3m e largura de $4m$.

```
void main(int argc, char** argv)
{
    // Inicializando camera e parede
    cam_z = 5.0f; // 5m atras da parede
    cam_y = 1.7f; // altura de uma pessoa
    parede_altura = 3.0f; // 3m
    parede_espessura = 0.3f; // 30cm
    parede_largura = 4.0f; // 4m
    parede_rotacao = 45.0f;
    parede_x = 0.0f;
    parede_y = 1.5f; // metade da altura
    parede_z = 0.0f;
    /*
    * center representa o ponto que estamos olhando,
    * nesse caso sera o centro da parede
    */
    center_x = parede_x;
    center_y = parede_y;
    center_z = parede_z;

    // ...
```

Adicione o código acima na função `display` (removendo a renderização do triângulo) e compile o programa. O resultado deve ser semelhante ao da figura 2.10.

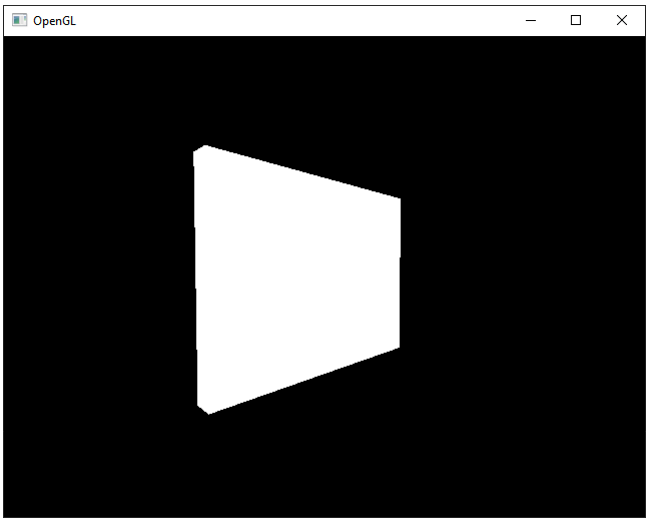


Figura 2.10: Parede renderizada com a função `glutSolidCube`

# Capítulo 3

# Iluminação

Para dar mais realismo à nossa cena, nesta seção aprenderemos como funciona o modelo de iluminação empregado pelo *OpenGL*, como simular diferentes materiais, e os principais tipos de fontes de luz. Para habilitar a iluminação no programa, adicione a função `setup_lighting`:

```
1 void setup_lighting()
2 {
3     glEnable(GL_LIGHTING); // Habilita a iluminacao
4     glEnable(GL_LIGHT0); // Habilita a luz 0
5     glEnable(GL_DEPTH_TEST); // Habilita o teste de
      profundidade
6 }
```

## 3.1 Modelo de iluminação

O modelo de iluminação utilizado pelo *OpenGL* é o modelo *Blinn-Phong*. Esse modelo é bastante conhecido por aproximar muito bem o comportamento da luz sem comprometer o desempenho da apicação. O modelo *Blinn-Phong* é um modelo de iluminação local, isto é, considera apenas a interação entre a luz, um objeto e o observador. Nele, a luz pode ser dividida em três componentes:

### 3.1.1 Luz ambiente

A luz ambiente é uma luz uniforme que ilumina toda a cena com a mesma cor e intensidade. Ela dá mais realismo pois na natureza raramente temos locais não ilumniados, porque a luz rebate nos objetos e acaba iluminando mesmo partes que estão opostas à fonte.

```
float light_ambient[] = {0.2f, 0.2f, 0.2f};
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
```

Figura 3.1: Luz ambiente em um objeto

### 3.1.2 Luz difusa

A luz difusa representa a luz que a superfície do objeto reflete. Portanto, quanto mais perpendicular for a direção da luz em relação ao vetor normal da superfície em um determinado ponto menor será a sua contribuição para o resultado final.

```
float light_diffuse[] = {0.0f, 0.0f, 0.5f}; // Luz azul
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
```

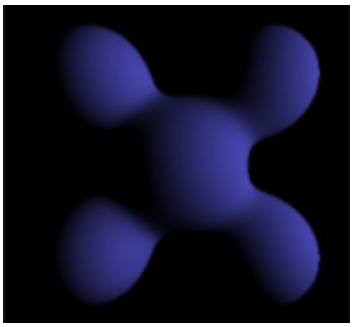

Figura 3.2: Luz difusa em um objeto

### 3.1.3 Luz especular

```
float light_specular[] = {1.0f, 0.0f, 0.0f}; // Luz branca
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
```

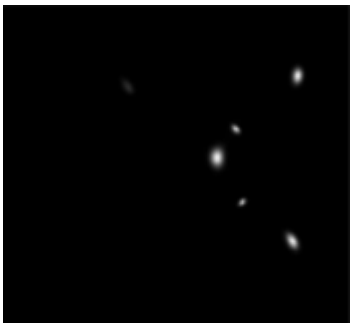

Figura 3.3: Luz especular em um objeto

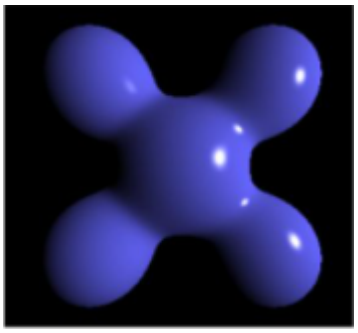

Figura 3.4: Resultado final, somando todas as componentes

## 3.2 Materiais

```
float mat_ambient[] = {0.2f, 0.2f, 0.2f, 1.0f};
float mat_diffuse[] = {1.0f, 0.0f, 0.0f, 1.0f}; // Cor azul
float mat_specular[] = {1.0f, 1.0f, 1.0f, 1.0f}; // Destaques
    brancos
float mat_shininess[] = {50.0f}; // O quao polida eh a
   superficie
// O primeiro parametro diz qual lado da face (GL_FRONT,
   GL_BACK etc)
glLightfv(GL_FRONT, GL_AMBIENT, mat_ambient);
glLightfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glLightfv(GL_FRONT, GL_SPECULAR, mat_specular);
glLightfv(GL_FRONT, GL_SHININESS, mat_shininess);
```

Além de definir o material utilizado, para que a iluminação seja calculada corretamente precisamos definir um vetor normal de cada vértice. Para isso, antes de definir a posição de cada vértice você deve chamar a função `glNormal3f` (semelhante à `glColor3f`):

```
// ...
glColor3f(1.0f, 0.0f, 0.0f); // vermelho
```

```
3 glNormal3f(0.0f, 0.0f, 1.0f); // normal do triangulo,
     definida no espaco local
4 glVertex3f(-0.5f, -0.5f, 0.0f);
```

Listagem 3.1: Definindo vetor normal do vértice. No caso das funções glutSolid* as normais já estarão definidas

## 3.3 Fontes de luz

### 3.3.1 Pontual

Uma luz pontual ilumina em todas as direções, como uma lâmpada. Sua luz decai de acordo com a distância. A direção de uma luz pontual é irrelevante. Para definir uma luz pontual é necessário definir a 4ª coordenada do vetor posição como 1:

```
1 GLfloat light_position[] = { 1.0, 1.0, 1.0, 1.0 };
2 glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

Listagem 3.2: Definindo uma fonte de luz pontual em (1 1 1)

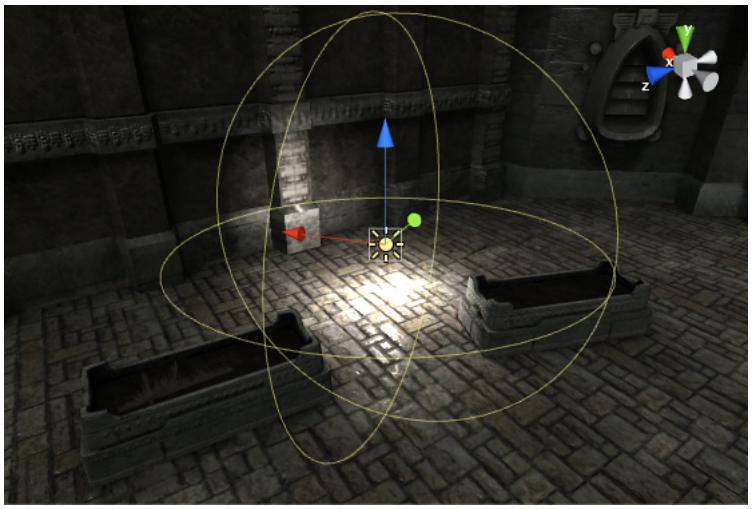


Figura 3.5: Fonte de luz pontual numa cena

### 3.3.2 Direcional

Uma luz direcional pode ser compreendida como o Sol, ilumina todos objetos na mesma direção. É equivalente a uma luz pontual no infinito. A

posição de uma luz direcional é irrelevante. Para definir uma luz direcional é necessário definir a 4ª coordenada do vetor posição como 0:

```
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

Listagem 3.3: Definindo uma fonte de luz direcional com vetor diretor (1 1 1)

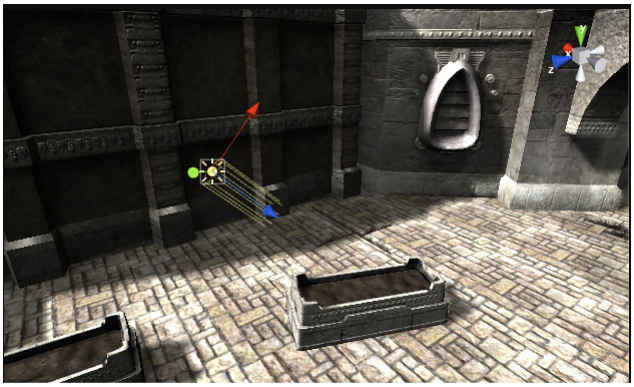

Figura 3.6: Fonte de luz direcional numa cena

### 3.3.3 Holofote

Um holofote é uma fonte de luz que ilumina uma região em forma de cone. Além da posição e direção, para definir um holofote precisamos do ângulo de abertura.

```
float spot_direction[] = {0.0f, -1.0f, 0.0f};
float spot_cutoff[] = {90.0f};
glLightfv(GL_LIGHT0, GL_SPOT_DIRECITON, spot_direction);
glLightfv(GL_LIGHT0, GL_SPOT_CUTOFF, spot_cutoff);
```

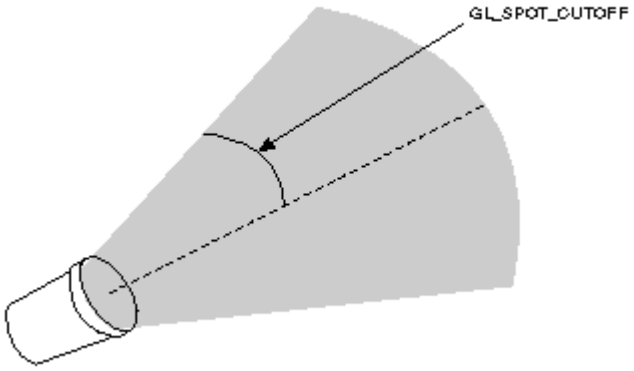


Figura 3.7: Representação de um holofote

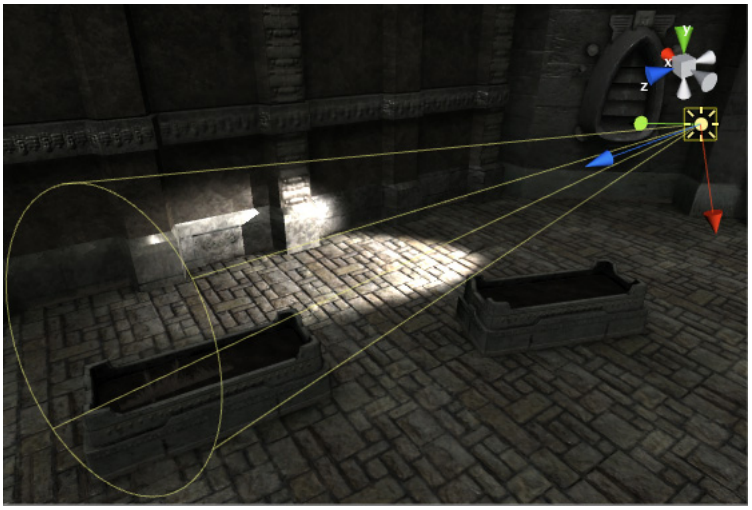


Figura 3.8: Holofote numa cena