Cenários 3D Interativos com *Software* Livre

Liliane dos Santos Machado[1]
Ronei Marcos de Moraes [2]

Resumo: Com a evolução tecnológica e o surgimento da realidade virtual, observa-se o desenvolvimento de sistemas gráficos cada vez mais realistas. Neste contexto, a criação de cenários tridimensionais, acrescida da visualização também tridimensional, vem se destacando. Este tutorial apresenta os passos para o desenvolvimento de cenários 3D interativos utilizando *software* livre. O artigo não se limita à criação das cenas tridimensionais, mas apresenta também a geração da visualização tridimensional das cenas, apresentando como integrar a esta, a interatividade e aspectos de realismo. Todo o código e os exemplos mostrados foram desenvolvidos utilizando a API OpenGL e/ou o pacote de modelagem Blender presentes no KLabteve, uma remasterização do Kurumin Linux, desenvolvida pelo Laboratório de Tecnologias para Ensino Virtual e Estatística (LabTEVE).

Palavras-chave: Computação Gráfica, Visualização Estereoscópica, *Software* Livre.

Abstract: With the technological evolution and the emergence of the virtual reality, graphic systems have becoming more realistic. In this context, there is great interest in the creation of three-dimensional scenes added by three-dimensional visualization. This tutorial presents the steps for the development of interactive 3D applications using free software. The paper also presents how to generate the stereoscopic visualization of the scenes and how to integrate it to an interactive and realistic application. Program code and examples presented were developed with the OpenGL API and/or the Blender modeling package. These tools are in the KLabteve, a Kurumin Linux remaster developed by the Laboratório de Tecnologias para Ensino Virtual e Estatística (LabTEVE).

Keywords: Computer Graphics, Stereoscopic Visualization, Free Software.

[1] Departamento de Informática, Universidade Federal da Paraíba. e-mail: liliane@di.ufpb.br
[2] Departamento de Estatística, Universidade Federal da Paraíba. e-mail: ronei@de.ufpb.br

1 Introdução

Aplicações gráficas interativas ganharam destaque com os jogos. Na verdade, os jogos são apenas uma das categorias mais conhecidas de aplicação da computação gráfica. Com a constante evolução tecnológica e o surgimento da realidade virtual, observa-se o desenvolvimento de aplicações gráficas cada vez mais realistas. Neste contexto, a criação de cenários tridimensionais, acrescida da visualização também tridimensional, vem se destacando, principalmente com o uso de *software* livre.

Este tutorial propõe-se a apresentar os passos para o desenvolvimento de cenários 3D interativos baseados em *software* livre com a utilização da OpenGL e da GLUT. A OpenGL é uma API (*Application Programming Interface*) com comandos usados para especificar objetos e operações necessárias para produzir aplicativos gráficos 2D e 3D. Diante das funcionalidades providas pela OpenGL, esta tem se tornado um padrão amplamente adotado na indústria de desenvolvimento de aplicações gráficas, sendo que grande parte das placas de vídeo produzidas já oferece aceleração por *hardware* às suas rotinas. As bibliotecas GLU e GLUT também são utilizadas para um acesso facilitado a algumas funcionalidades da OpenGL. Para a modelagem tridimensional será também é utilizado o pacote Blender.

Todo o código e os exemplos mostrados serão desenvolvidos com o uso do KLabteve, uma remasterização do Kurumin Linux desenvolvida pelo Laboratório de Tecnologias para Ensino Virtual e Estatística (LabTEVE) da UFPB. Esta remasterização contém todas as ferramentas em *software* livre necessárias para a criação de aplicações gráficas, incluindo modelagem bi e tridimensional, tratamento e edição de imagens, linguagens de programação e bibliotecas para programação gráfica.

Assim, partindo da modelagem com o programa Blender, serão gerados objetos tridimensionais. Será abordada a organização dos arquivos de objetos 3D e a sua integração a um código em linguagem C para geração de cenários tridimensionais. Em seguida serão tratados os métodos utilizados para projeção estereoscópica destas cenas e os dispositivos necessários para esta visualização. Finalmente será apresentado o processo de integração aspectos de interatividade por mouse ou teclado. Aspectos de iluminação também serão tratados com o objetivo de fornecer maior realismo às aplicações.

2 KLabteve

O LabTEVE – Laboratório de Tecnologias para Ensino Virtual e Estatística – da Universidade Federal da Paraíba (UFPB) foi criado em 2000 com o objetivo de desenvolver e integrar tecnologias voltadas ao Ensino Virtual, Ensino à Distância e Estatística. O objetivo do laboratório é prover e apoiar o aprendizado dos alunos por sistemas que disponibilizem informações acessíveis de qualquer parte e a qualquer momento, bem como treinar alunos e professores para manipularem tais tecnologias.

Nessa linha de trabalho foi desenvolvido o KLabTEVE remasterizado pelo LabTEVE (http://www.de.ufpb.br/~labteve). Essa remasterização Linux é baseada na famosa distribuição Linux nacional Kurumin (http://www.guiadohardware.net/kurumin/), que por sua vez baseia-se no Knoppix com base no Debian, todas elas desenvolvidas por voluntários. O KLabTEVE roda a partir do CD-ROM, sem a necessidade de instalação, mas podendo também ser instalado no disco rígido do computador, como qualquer distribuição Linux tradicional. O arquivo ISO do KLabTEVE pode ser obtido gratuitamente sob licença GPL (GNU *Public License*) a partir da página http://www.de.ufpb.br/~labteve/kurumin.html.

No KLabTEVE, atualmente na versão 1.2, foram incluídas várias versões livres de linguagens de programação como GNU C/C++, GNU Java, GNU Fortran e FreePascal, além de conversores como o f2c. Também possui a API OpenGL, as bibliotecas GLU e GLUT para programação gráfica e os pacotes de modelagem Qcad e Blender. Inclui também os editores gráficos GIMP e Inkscape, dentre outras facilidades, oferecendo todas as ferramentas necessárias para a criação de aplicações gráficas sem a necessidade de uso de programas proprietários. A Figura 1 apresenta o ambiente do KLabTEVE.

Figura 1: Ambiente de trabalho do KLabTEVE com os ícones das ferramentas e aplicações instaladas.

3 API OpenGL, GLU e GLUT

A OpenGL é uma API (*Application Programming Interface*) com comandos usados para especificar objetos e operações necessárias para produzir aplicativos gráficos 2D e 3D. Diante das funcionalidades providas pela OpenGL, esta tem se tornado um padrão amplamente adotado na indústria de desenvolvimento de aplicações gráficas. Este fato tem sido encorajado também pela facilidade de aprendizado, pela estabilidade das rotinas, pela documentação disponível e pela qualidade dos resultados produzidos pelo uso da OpenGL [1]. Para obter tal estabilidade, a OpenGL não inclui comandos de tratamento de janelas ou interação com o usuário.

Visto que a OpenGL não oferece comandos de alto-nível para definir objetos gráficos tridimensionais foi desenvolvida a GLU (*OpenGL Utility Library*), uma biblioteca padrão de utilitários para OpenGL. A GLU oferece diversas facilidades na definição de objetos tridimensionais, como superfícies quadráticas ou curvas e superfícies com NURBS (*Non-Uniform Rational B-Splines*) [2].

A GLUT (*OpenGL Utility Toolkit*) é um conjunto de ferramentas para desenvolvimento de programas baseados em OpenGL. Sua principal vantagem é que ela oferece meios de lidar com janelas gráficas e interações realizadas pelo usuário. Por ser independente do sistema gráfico utilizado, permite a escrita de aplicações que podem ser compiladas e executadas em qualquer sistema de janelas, facilitando a criação de aplicações gráficas. Pelo fato da GLUT possuir código aberto ela também é um excelente recurso para aprender como funciona a OpenGL e um sistema de janela [3].

Para ambiente Linux, a implementação OpenGL utilizada é a biblioteca *Mesa*. Esta biblioteca contém a OpenGL, a GLU e a GLUT e sua documentação está disponível em http://www.opengl.org/.

4 Criando Cenários 3D

Criar uma cena gráfica equivale ao processo de obtenção de uma cena por uma câmera fotográfica. Quando queremos fazer uma foto, inicialmente posicionamos a câmera para captar a cena desejada, eventualmente usando um tripé. No computador, posicionamos um volume de visualização (*viewing*). Na fotografia, após posicionar a câmera, pelo visor escolhemos a cena que será fotografada. Já no computador, posicionamos os objetos na cena virtual (*modeling*). Podemos ainda, na fotografia, escolher diferentes lentes ou ajustar o *zoom*, o que determinamos no computador com o formato do volume de visualização (*projection*). Por fim, definimos o tamanho da foto, o que equivale a *viewport* ou janela de visualização no computador. Ao final desses passos, na fotografia temos a cena capturada e estampada e no computador a cena estará pronta para ser desenhada. Para criar uma cena 3D com a OpenGL deve-se também obedecer estes quatro passos. O formato básico de criação de uma aplicação OpenGL com linguagem GNU C e usando a GLUT é:

```
#include <gl/glut.h>
int main(int argc,char **argv)
{
   glutInit(&argc,argv);                          // inicia a GLUT
   glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB); // define o modo de exibição
   glutInitWindowSize(100,100);                   // tamanho da janela gráfica
   glutInitWindowPosition (0,0);                  // posição da janela
   glutCreateWindow ("Cena 3D");                  // cria a janela gráfica

   glutReshapeFunc(Reshape);                      // redimensionamento
   glutDisplayFunc(Display);                      // apresentação da cena

   glutMouseFunc(Mouse);                          // interação com o mouse
   glutMotionFunc(Motion);
   glutKeyboardFunc(Key);                         // funções de teclado
   glutSpecialFunc(Special);                      // teclas especiais

   glutMainLoop();                                // inicia o laço gráfico
}
```

onde *Reshape* trata do redimensionamento da janela gráfica, *Display* é responsável pela apresentação da cena gráfica, *Mouse* reconhece o pressionamento de botões do *mouse*, *Motion* atua quando um movimento de *mouse* é executado junto com um botão pressionado, *Key* reconhece o acionamento das teclas alfanuméricas do teclado e *Special* reconhece o acionamento de teclas especiais do teclado, como Home, End, PgUp e PgDn. Todas estas funções são escritas pelo programador e devem ser inseridas no código através da chamada da função GLUT relacionada.

Nas próximas seções serão apresentados os passos necessários para definição das operações de visualização (*viewing*), modelagem (*modeling*), projeção (*projection*) e definição da janela de visualização (*viewport*) com a OpenGL. A OpenGL utiliza um sistema de matrizes para combinar as operações destes quatro passos, o que nos obriga a sempre informá-la a qual destas etapas refere-se um determinado comando utilizado. Para isso, antes de chamar os comandos de cada operação é importante habilitar a matriz correspondente com *glMatrixMode(matriz_operante)*, onde *matriz_operante* pode ser: GL_MODELVIEW, GL_PROJECTION, GL_VIEWPORT. Observe que o argumento GL_MODELVIEW permite o uso das operações de visualização e/ou modelagem.

4.1 Visualização (*viewing*)

A etapa de visualização permite posicionar o observador em relação à cena. Como padrão o observador estará sempre na posição xyz = (0,0,0) olhando para o centro da cena também em xyz = (0,0,0). Na maioria das vezes, opta-se por afastar o usuário da cena para que este possa vê-la por completo. A função responsável por este posicionamento é a *gluLookAt* e permite não apenas mover o observador, mas também alterar a sua direção de visualização. A Figura 2 ilustra o funcionamento do comando *gluLookAt* utilizado para definir a visualização. Sua sintaxe é:

```
gluLookAt(observador xyz, centro da cena xyz, verticalxyz)
```

onde vertical identifica a direção vertical de visualização estabelecida pelo eixo ativado com valor 1.

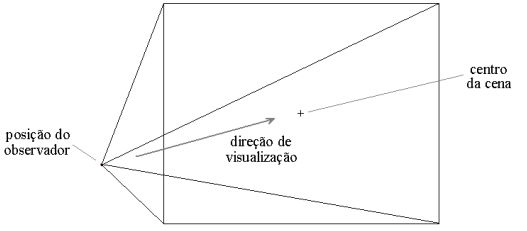
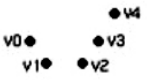
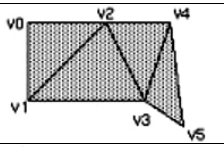
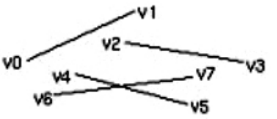
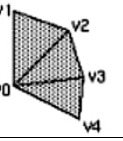
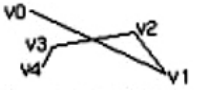
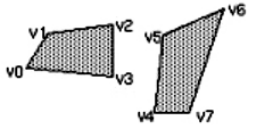
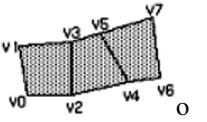


Figura 2: Cone de visualização gerado pela função *gluLookAt*.

4.2 Modelagem e Representação dos Objetos (*modeling*)

Modelar um objeto significa (re)produzir sua forma. Isto pode ser feito a partir de primitivas como pontos e retas. A GLUT oferece alguns objetos 3D pré-definidos e que podem ser usados para compor outros objetos. Como exemplo pode-se citar as funções *glutSolidCube(tamanho)* e *glutWireCube(tamanho)* para representar cubos no formato sólido ou aramado. A OpenGL por sua vez, permite desenhar qualquer objeto 3D a partir de pontos e suas conexões. Estas conexões podem ser de diferentes tipos (Figura 3) e devem ser informadas dentro da função de *display*, responsável pela exibição da cena.

GL_POINTS		GL_TRIANGLE_STRIP	
GL_LINES		GL_TRIANGLE_FAN	
GL_LINE_STRIP		GL_QUADS	
GL_LINE_LOOP		GL_QUAD_STRIP	

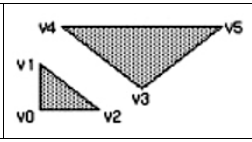
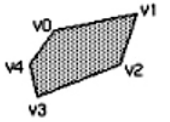
GL_TRIANGLES		GL_POLYGON	

Figura 3 – Tipos de primitivas geométricas na OpenGL.

Os pontos do objeto podem ser inseridos individualmente no código ou lidos de um arquivo de modelo (Seção 4.2.1). Neste último caso, o arquivo deve conter os pontos e conexões que formam o objeto e o código do programa deve importá-los para uma estrutura de dados a ser utilizada pela OpenGL.

Também é na etapa de modelagem onde são definidas operações de transformação geométrica sobre os objetos, como rotações, translações e escalas. Isso permite que um mesmo objeto possa ser utilizado várias vezes, aplicando-se sobre estes as transformações, para a composição de um cenário 3D completo. Algumas das operações de transformação possíveis são (Figura 4):

```
glRotate(ângulo,x,y,z)    // rotação
glTranslate(x,y,z) // translação
glScale(x,y,z)            // escalonamento
```

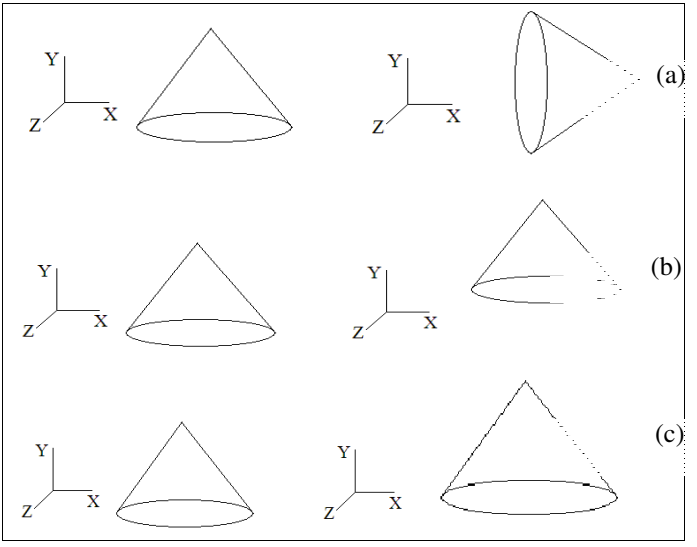


Figura 4: As operações de transformação (a) *glRotate*, (b) *glTranslate* e (c) *glScale*.

É importante lembrar que cena definida deve sempre estar contida em uma função de *Display* que será executada todas as vezes que o redesenho da cena for necessário. Isso

poderá ocorrer devido a um redimensionamento da janela ou devido a uma interação do usuário (vide Seção 6).

O código a seguir apresenta um trecho de código para desenhar um quadrado na cor verde, cujos pontos foram inseridos individualmente (Figura 5).

```
// seleciona a matriz a ser utilizada
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

// especifica que a cor de desenho verde (RGB)
glColor3f(0.0f, 1.0f, 0.0f);

// desenha um quadrado preenchido com a cor especificada
glBegin (GL_QUADS);
    glVertex3f(0.0,0.0,0.0);
    glVertex3f(1.0,0.0,0.0);
    glVertex3f(1.0,1.0,0.0);
    glVertex3f(0.0,1.0,0.0);
glEnd();

// rotaciona 25 graus o objeto ao redor do eixo x
glRotate(25,1,0,0);
```

onde *glColor3f* determina a cor que será usada para o desenho (linhas e preenchimento) e *glVertex3f* define um ponto no espaço xyz. Os parâmetros de *glColor* são dados por valores no espaço de cores RGB (vermelho, verde e azul).

Figura 5: Apresentação de um quadrado rotacionado definido na OpenGL com a diretiva GL_QUADS.

4.2.1 Modelagem com o Blender

Uma maneira rápida de modelar objetos tridimensionais é utilizando um editor 3D, como o Blender. O Blender é um *software* livre com licença GPL (GNU *Public License*) para modelagem tridimensional, animação, pós-produção e criação de jogos, disponível para plataformas Windows, Linux, Irix, Sun Solaris, FreeBSD e Mac OS X [4]. Este pacote

permite que qualquer objeto modelado seja exportado para o formato VRML ou OBJ. Arquivos nestes formatos basicamente descrevem vértices e arestas que podem ser importados para uso na OpenGL com uma simples leitura do arquivo.

Figura 6: Interface do pacote de modelagem gratuito Blender 2.37a para Linux.

A interface do Blender é composta por duas partes principais: uma área de trabalho localizada na parte superior e as ferramentas de trabalho localizadas na parte inferior (Figura 6). As ferramentas de trabalho apresentadas dependem do menu selecionado, dentre quinze opções de menu disponíveis. Para modelar um cubo, por exemplo, pode-se escolher a opção "Add" "Mesh" "Cube" no menu "User Preferences", apresentado inicialmente na parte superior da área de trabalho. Essa opção desenha um cubo com dimensões pré-definidas, como um conjunto de pontos e linhas interligadas. É possível aplicar transformações de rotação, translação e escala para modificar o cubo utilizando o menu "3D View". A adição de outros elementos permitirá a criação de uma cena. Nesta cena, os modelos, as fontes de luz (vide Seção 7) e o observador/câmera são todos considerados objetos. Para adicionar uma fonte de luz deve ser utilizada a opção "Add" "Lamp" no menu "User Preferences". Para exportar esta cena para o formato VRML ou OBJ basta selecionar a opção "File" "Export" no menu "User Preferences". A Figura 7 apresenta uma cena, composta por um cubo e três fontes de luz, modelada no Blender e a sua visualização a partir da câmera.

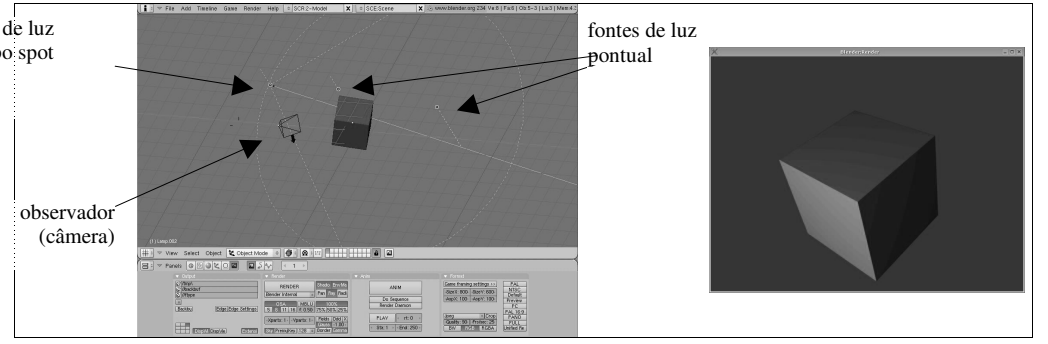


Figura 7: Cena modelada no Blender e a sua visualização pelo observador.

4.3 Projeções (*projection*)

O processo de projeção consiste em definir como objetos tridimensionais serão visualizados, uma vez que os dispositivos de saída dos computadores, como o monitor, são bidimensionais. Na projeção as coordenadas 3D dos objetos são convertidas em coordenadas 2D, de acordo com o tipo de projeção selecionada. Podemos ter projeções paralelas ou perspectivas.

Na projeção paralela, a distância entre a câmera e a cena não afeta o tamanho dos objetos. Este tipo de projeção é indicado para aplicações na arquitetura ou desenho auxiliado por computador (CAD) onde o tamanho dos objetos e os ângulos entre eles devem ser mantidos na visualização. Na projeção perspectiva, a distância entre a câmera e a cena afeta o tamanho dos objetos. Isso ocorre porque o volume de visualização é piramidal, fazendo com que um objeto posicionado no fundo apareça menor que um objeto de mesmo tamanho posicionado mais à frente. As projeções perspectivas são as mais indicadas no caso da produção de cenas realistas, uma vez que podem reproduzir o modo de visualização do mundo pelo ser humano.

As funções OpenGL utilizadas para operações de projeção em perspectiva são a *gluPerspective* e a *glFrustum*. A principal diferença entre elas é que a *gluPerspective* cria um volume simétrico de visualização a partir de um ângulo de visada enquanto a função *glFrustum* cria um volume não necessariamente simétrico, o que pode ser útil em algumas situações (Seção 5). A sintaxe destas funções é:

```
gluPerspective (alfa, aspecto, proximidade, distancia)
glFrustum (esquerda, direita, inferior, topo, proximidade, distancia)
```

onde *alfa* é ângulo em y para determinar a altura do volume; *aspecto* é a razão entre a largura e altura do volume; *proximidade* e *distancia* referem-se à distância do observador aos planos frontal e posterior do volume; esquerda e direita são os limites laterais do volume; inferior e topo são os limites verticais do volume. Para usar qualquer uma destas funções é necessário que a matriz de projeção *glMatrixMode* esteja configurada como GL_PROJECTION.

4.4 Janela de Visualização (*viewport*)

A janela de visualização é a área retangular da janela gráfica onde é desenhada a imagem final produzida. Ela é medida em coordenadas que refletem a posição dos pixels em relação à sua margem inferior esquerda. A proporção horizontal e vertical dessa janela permite a produção de distorções na apresentação da imagem. Geralmente, mantém-se na definição desta as mesmas proporções dos monitores, ou seja 4 por 3. A função responsável por essa tarefa é:

```
glViewport(x, y, largura, altura);
```

onde x e y definem o canto inferior esquerdo da janela de visualização dentro da janela gráfica. Em geral, x e y recebem o valor zero, mas esse valor pode ser modificado para mais de uma cena em diferentes regiões da janela gráfica. Para utilizar *glViewport* a função *glMatrixMode* deve ser habilitada com GL_VIEWPORT.

5 Criando Estereoscopia

A estereoscopia é a ciência e arte que trabalha com imagens para produzir um modelo visual tridimensional com características análogas às características da mesma imagem quando vista através da visão binocular real. Neste caso, cada um dos olhos humanos, ao observar o mundo real, recebe imagens diferentes devido à distância que existe entre os olhos. Estas imagens são fundidas pelo cérebro nos permitindo a percepção visual tridimensional. Desse modo, criar e visualizar imagens tridimensionais significa produzir um par de imagens, ligeiramente separadas horizontalmente, de uma mesma cena.

Nas imagens estereoscópicas geradas por computador, a quantidade de paralaxe - distância horizontal entre imagens esquerda e direita - determina a distância aparente dos objetos virtuais em relação ao observador [5]. A paralaxe é importante porque o seu valor no par estéreo determinará a distância ou intervalo horizontal entre quaisquer dois pontos nas imagens. Pode-se citar quatro tipos básicos de paralaxe: paralaxe zero, paralaxe positiva, paralaxe negativa e paralaxe divergente. Quando os pares estéreos possuem paralaxe zero, não há qualquer intervalo entre as imagens. Quando as imagens possuem paralaxe positiva é possível notar profundidade na imagem de fusão. A paralaxe negativa ocorre quando as linhas de visão estão cruzadas, ou seja, o olho esquerdo visualiza a imagem da direita e o olho direito visualiza a imagem da esquerda. Nestes casos a imagem final deixa de aparecer sobre o plano de visualização (tela). Chama-se a paralaxe de divergente quando o seu valor é maior que o espaçamento interocular. Esta situação, no entanto, nunca ocorre no mundo real e deve ser evitada nos pares estéreo devido ao grande desconforto gerado ao observador. A Figura 8 apresenta os tipos de paralaxe possíveis.

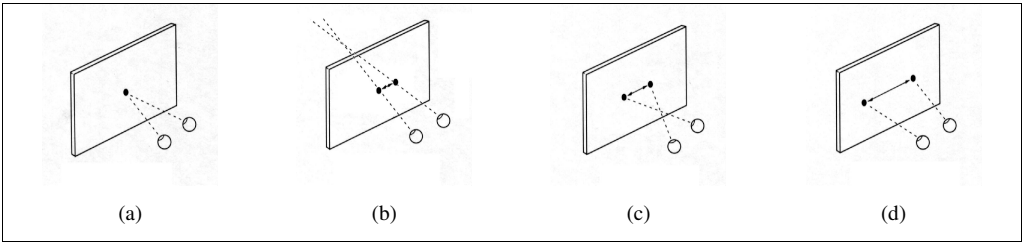


Figura 8: Tipos de paralaxe: (a) zero, (b) positiva, (c) negativa e (d) divergente.

Para a obtenção do par estereoscópico, podem ser utilizados três métodos: projeção *off-axis*, projeção *on-axis* e rotação [6]. Na projeção *off-axis* assume-se a existência de dois centros de projeção, sendo que a visão esquerda é produzida baseada no centro de projeção esquerdo e a visão direita é produzida baseada no centro de projeção direito. Assim, um mesmo objeto é observado de dois pontos diferentes, podendo ocasionar assimetria no volume de visualização. Já na projeção *on-axis*, ao invés de dois centros de projeção, é utilizado um único centro de projeção em conjunto com translações horizontais dos dados. Neste caso, a obtenção de cada imagem do par estereoscópico é feita através de três passos: translação da imagem para a direita ou esquerda (dependendo da imagem do par estereoscópico a ser gerada), projeção perspectiva, e translação da imagem para o sentido contrário da primeira translação. Na aplicação de rotação para a obtenção do par estereoscópico, observa-se que esta técnica é bastante rápida computacionalmente, uma vez que as imagens são obtidas através da simples rotação vertical do centro de projeção (um total de 4 graus, normalmente). No entanto, este método é bastante utilizado em projeções paralelas de cenas. No caso de rotação implementada juntamente com projeção perspectiva, as imagens obtidas apresentam deficiências, como paralaxe vertical e distorções que afetam a qualidade da imagem.

Dentre os três métodos, o que mais se aproxima do modo como o ser humano visualiza o mundo real é o método o *off-axis*. Este método supõe a existência de dois centros de projeção mirando para um ponto em comum. A Figura 9 apresenta o campo de visão gerado com o uso das técnicas *on-axis* e *off-axis*.

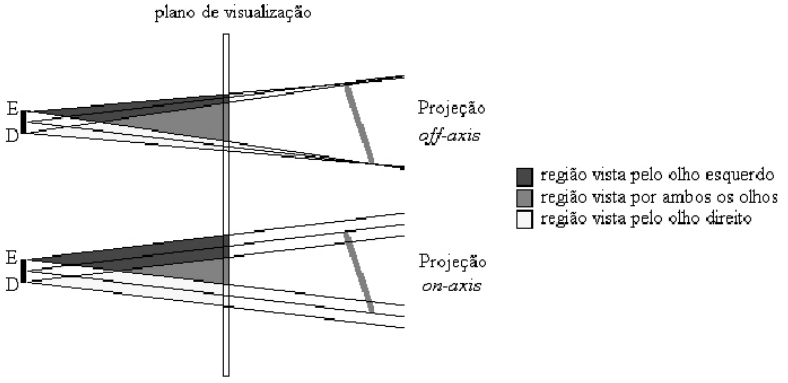


Figura 9: Comparação do plano visual comum obtido pelos métodos de projeção perspectiva estereoscópica *off-axis* e *on-axis*.

Com a OpenGL é possível gerar cenários 3D com qualquer um dos três tipos de projeção estereoscópica. A obtenção dos pares esteroscópicos no processo de rotação pode ser feita fixando o observador e girando a cena ao redor de si mesma. Conforme visto anteriormente, a imagem 3D resultante apresenta paralaxe vertical quando este método é utilizado com projeção perspectiva, tornando-o pouco indicado neste caso. Esse efeito pode ser observado na Figura 10.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60,4/3,dnear,dfar);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

// imagem esquerda
gluLookAt(x0,y0,z0, xref,yref,zref, Vx,Vy,Vz);
glutWireTeapot(30);

// imagem direita
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

gluLookAt(x0,y0,z0, xref,yref,zref, Vx,Vy,Vz);
glRotated(4,0,1,0);
glutWireTeapot(30);
```

RESULTADO

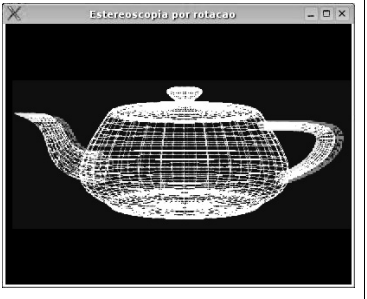


Figura 10: Geração de um par estereoscópico através de rotação.

Para o método *on-axis* pode ser utilizada a função *gluPerspective* para a projeção, visto que a cena final é obtida com o deslocamento horizontal da imagem e o volume de visualização é simétrico. Veja na Figura 11 uma possível codificação do método *on-axis*.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(45,4/3,dnear,dfar);

// imagem esquerda
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslated(-d,0,0);
glutWireTeapot(30);

// imagem direita
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslated(d,0,0);
glutWireTeapot(30);
```

RESULTADO

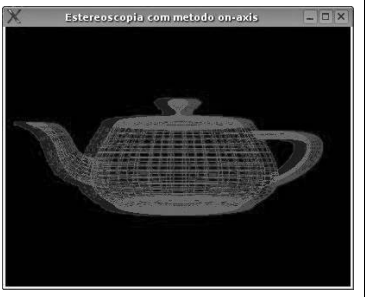


Figura 11: Implementação do método *on-axis* para geração de um par estereoscópico.

O método *off-axis*, conforme dito anteriormente, supõe a existência de dois volumes de visualização, correspondentes à imagem esquerda e à imagem direita. A assimetria destes volumes está relacionada ao efeito de imagem "saindo da tela". Para que isso ocorra, apenas a função *glFrustum* pode ser utilizada visto que permite a definição do formato assimétrica do volume de visualização e relacionado ao deslocamento horizontal das imagens. A Figura 12 apresenta o código e o resultado da aplicação deste método.

```
// esquerdo
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(xwMin-d,xwMax-d,ywMin,ywMax,dnear,dfar);
glTranslated(-d,0,z);
glutWireTeapot(30);

// direito
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(xwMin+d,xwMax+d,ywMin,ywMax,dnear,dfar);
glTranslated(+d,0,z);
glutWireTeapot(30);
```

RESULTADO

Figura 12: Implementação do método *off-axis* para geração de um par estereoscópico.

5.1 Dispositivos de Visualização

Para visualizar o par de imagens criado é necessário o uso de algum dispositivo que permita a separação das imagens esquerda e direita para cada olho do espectador/usuário. Dentre as técnicas mais conhecidas, destaca-se o uso óculos obturadores (Figura 13a) e o uso de óculos com filtros polarizados (Figura 13b) ou coloridos (Figura 13c). Para o uso de óculos obturadores é necessário que o computador seja equipado com uma placa de vídeo com um conector de saída estéreo. A este conector é ligado um emissor infravermelho que sincronizará as imagens alternadas no vídeo com a obturação total e também alternada de uma das lentes dos óculos [7]. O uso de óculos com filtros polarizadores, por sua vez, exige mecanismos que reconheçam a alternância das imagens produzidas por uma mesma placa. Neste caso, as imagens são exibidas por projetores, diante dos quais são posicionados filtros polarizadores [5]. Por sua vez, o uso de óculos com filtros coloridos é provavelmente o método mais popular de visualização tridimensional, pois exige apenas o uso de óculos com lentes vermelhas e azuis/verdes que podem ser produzidos pelo próprio usuário com cartolina e papel celofane. Este método de visualização é conhecido por anaglifo.

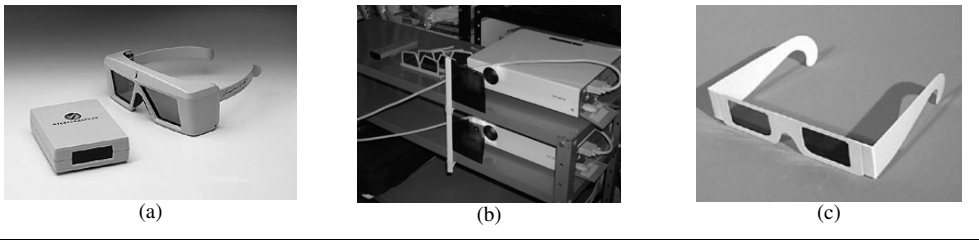


Figura 13: Dispositivos de visualização estereoscópica. (a) óculos obturadores e emissor de sincronização, (b) óculos polarizadores com projetores e filtros, e (c) óculos com filtros coloridos.

Durante a geração do par estéreo pela OpenGL o tipo de dispositivo de visualização já deve ser conhecido para que a imagem gerada possa se adequar ao dispositivo. A OpenGL reconhece as placas de vídeo com conector estéreo e gera o sinal de sincronização necessário para a visualização das imagens separadamente. Esta opção deve ser habilitada durante a definição do modo de exibição na função *glutInitDisplayMode* com a diretiva GL_STEREO, específica para este tipo de placa de vídeo.

Para o uso de óculos com filtros polarizadores a visualização não poderá ser feita em um monitor de vídeo convencional. Geralmente utiliza-se um par de projetores com filtros polarizadores posicionados na frente das lentes e algum método baseado em *hardware* para separação do sinal de vídeo para cada projetor. Um projetor ficará responsável por exibir a imagem esquerda e o outro por exibir a imagem direita. Por essa razão, não é necessário configurar opções específicas na OpenGL. A mesma situação pode ocorrer quando são utilizados óculos com filtros coloridos para visualização por anaglifo. Mas, como a separação das imagens é dada pelo uso de cores complementares, esta separação pode ocorrer durante a exibição em um monitor convencional desde que estas mesmas cores sejam utilizadas para gerar a imagem. Isto pode ser feito habilitando *buffers* de cor específicos para

escrita. No sistema RGB (vermelho, verde, azul), e supondo óculos com lentes vermelha e azul, seria habilitado para escrita apenas o *buffer* vermelho para a escrita da imagem esquerda do par estereoscópico. O *buffer* azul seria habilitado apenas para a escrita da imagem direita. Desse modo, durante a apresentação no monitor será observada uma grande imagem magenta com deslocamentos à esquerda e à direita em vermelho ou azul. A cor magenta é o resultado da combinação do que foi desenhado no *buffer* da cor azul com o que foi desenhado no *buffer* da cor vermelha. A definição de um *buffer* específico de cor para escrita é feito na OpenGL com a função *glColorMask*. Como padrão a OpenGL mantém todos os 3 *buffers* de cor (modo RGB) habilitados para escrita. Assim, para a escrita da imagem esquerda os *buffers* de cor verde e azul devem ser desabilitados e, para a escrita da imagem direita, apenas os *buffers* de cor vermelha e verde devem ser desabilitados. A sintaxe da função *glColorMask* é:

```
glColorMask (R,G,B,A)
```

sendo que o *buffer* habilitado para escrita deve ter valor um (1) e o buffer desabilitado deve ter valor zero (0).

6 Interação

Para a interação do usuário com o sistema, a GLUT disponibiliza palavras reservadas próprias para algumas teclas especiais do teclado e para os botões do mouse. Dessa forma, ações do usuário podem ser capturadas pelo sistema para atualizar a cena gráfica. Para reconhecimento de comandos de teclado é necessário incluir uma chamada para:

```
glutKeyboardFunc(Key);                  // funções de teclado
glutSpecialFunc(Special);               // teclas especiais
```

onde *Key* é a função que determinará a ação de cada tecla, reconhecendo-a como o valor alfa-numérico correspondente (ASCII). A função *Special* é utilizada para reconhecer as teclas especiais do teclado (não alfanuméricas) às quais é associado pela GLUT um valor inteiro.

O exemplo abaixo apresenta uma implementação das funções *Key* e *Special* para interação do usuário. Na função *Key* as teclas "+" e "-" permitem alterar uma variável utilizada na função de *Display* para escalonar objetos. Além disso, a tecla "Esc" e a letra "q" ou "Q" são utilizadas para encerrar a execução do programa. Na função *Special* as teclas especiais "Home" e "End"são utilizadas para aproximar ou afastar o observador da cena através do incremento ou decremento de uma variável utilizada na função *Display*. Já as teclas "Page Up" e "Page Down" são utilizadas para rotacionar o objeto ao redor de z através de uma variável também utilizada na função *Display*. A chamada de *glutPostRedisplay* no final de cada função é necessária para forçar o re-desenho da cena gráfica, chamando a função de *Display*, permitindo que as ações do usuário surtam efeito.

```
void Key(unsigned char key,int x,int y)
   {
      switch(key) {
         case '+':
            escala += 1.0;
             break;
         case '-':
            escala -= 1.0;
             break;
         case 27:
         case 'q':
         case 'Q':
             exit(0);
      }
      glutPostRedisplay();
   }
```

```
void Special(int key,int x,int y)
   {
      switch(key) {
         case GLUT_KEY_PAGE_UP:
            zoom+=1.0;
            break;
         case GLUT_KEY_PAGE_DOWN:
            zoom-=1.0;
            break;
         case GLUT_KEY_HOME:
            anglez-=0.1;;
            break;
         case GLUT_KEY_END:
            anglez-=0.1;
            break;
      }
      glutPostRedisplay();
   }
```

Para reconhecimento de comandos do *mouse* a GLUT precisa de duas funções: uma para reconhecer os movimentos e outra para reconhecer os botões pressionados. As funções são:

```
void Motion(int x, int y);
void Mouse(int button, int state, int x, int y);
```

Com a função *Motion* é possível identificar a posição do mouse no plano *xy* e incrementar, por exemplo, o ângulo de rotação da cena gráfica. A função *Mouse*, por sua vez, verifica se e qual botão do dispositivo foi pressionado e em que posição do plano *xy* isto ocorreu. O exemplo a seguir mostra o que ocorre quando o botão esquerdo do mouse é pressionado junto com a movimentação para rotacionar um objeto.

```
void Motion(int x, int y)
{
  if (moving)
  {
     angley+= ((double)(x -
               beginx)*100 /
              (double)winwidth);
     anglex+= ((double)(y -
               beginy)*100 /
              (double)winheight);

      glutPostRedisplay();
      beginx = x;
      beginy = y;
   }
}
```

```
void Mouse(int button, int state, int
x, int y)
{
   if (button == GLUT_LEFT_BUTTON
       && state == GLUT_DOWN)
   {
       moving = 1;
       beginx = x;
       beginy = y;
   }
   else
    if (button == GLUT_LEFT_BUTTON
        && state == GLUT_UP)
       moving = 0;

   glutPostRedisplay();
}
```

A função *Mouse* identifica o botão pressionado e ativa a detecção da posição do mesmo pela função *Motion*. Com isso, uma variável utilizada no *Display* para rotação é incrementada ou decrementada. Do mesmo modo que nas interações por teclado, a chamada de *glutPostRedisplay* no final de cada função é necessária para forçar o re-desenho da cena gráfica.

6.1 Explorando os *buffers* de quadro para aumentar o desempenho

Quando as cenas 3D são interativas, imagens diferentes são geradas a cada interação do usuário, o que pode exigir o processamento e apresentação de diversos quadros diferentes por segundo. O processamento das operações gráficas e a sua apresentação na tela podem ser otimizados explorando os bancos de memória de vídeo, ou *buffers* de quadro. Isto porque cada quadro processado da cena gráfica precisa ser completamente preenchido para ser apresentado.

As placas de vídeo atuais possuem pelo menos dois *buffers* de vídeo. Um desses *buffers* é responsável por aquilo que o usuário está de fato visualizando. Outro *buffer* fica disponível para escritas em segundo plano. (Obs: no caso de placas de vídeo próprias para estereoscopia existem 4 *buffers* de quadro, um par com primeiro e segundo planos para cada imagem do par estereoscópico.) A otimização de aplicações gráficas interativas pode ser obtida explorando esse sistema de duplo *buffer*. Assim, enquanto um quadro do *buffer* em primeiro plano é exibido, o outro já vai sendo processado e escrito no *buffer* de segundo plano. Concluída esta etapa, o conteúdo do *buffer* em segundo plano passa para o *buffer* de primeiro plano e o processo de escrita recomeça.

A OpenGL permite o uso do sistema de duplo *buffer*, desde que esta funcionalidade seja habilitada. A GLUT facilita essa tarefa com a diretiva GLUT_DOUBLE ao invés de GLUT_SINGLE durante a definição do modo de exibição:

```
// define o modo de exibição ew o sistema de cores
glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
```

Uma vez habilitado o sistema de duplo buffer, é necessário informar o *buffer* de escrita durante o processo de composição de cada quadro com a função *glDrawBuffer*. Como forma de otimização da aplicação gráfica, utiliza-se sempre o *buffer* de segundo plano para escrita com:

```
glDrawBuffer(GL_BACK)
```

e, após a completa composição da cena, descarrega-se o conteúdo do *buffer* de segundo plano no primeiro com a função:

```
glutSwapBuffers()
```

7 Iluminação

A iluminação permite adicionar realismo a uma cena gráfica, simulando o comportamento de fontes de luz sobre os objetos. A Figura 14 apresenta o resultado de uma cena composta por um cone sem iluminação (a) e com iluminação (b). A iluminação é obtida através da definição de propriedades de luzes e de materiais que constituem os objetos da cena e da definição de parâmetros globais de iluminação A OpenGL trata a luz e a iluminação através da sua decomposição nas cores vermelha, verde e azul (RGB). Desse modo, fontes de luz são caracterizadas pela quantidade de luz vermelha, verde e azul que emitem. O material da superfície dos objetos, por sua vez, é caracterizado pela quantidade de luz vermelha, verde e azul que refletem.

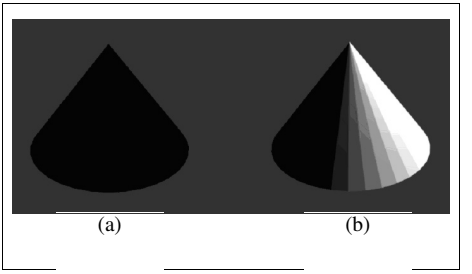


Figura 14: (a) Cone sem iluminação e (b) cone com iluminação.

Uma fonte de luz pode ser caracterizada por três componentes: ambiente, difusa e especular. A componente ambiente caracteriza a luz que se espalha pelo ambiente de modo que não é possível determinar sua direção. A componente difusa caracteriza uma luz unidirecional que atinge a superfície e é refletida em todas as direções. A componente especular também caracteriza uma luz unidirecional, mas esta tende a ser refletida em uma única direção. A OpenGL permite definir de forma independente os valores de vermelho, verde e azul para cada componente de luz.

Da mesma forma que as luzes, os materiais possuem cor ambiente, difusa e especular que determinam como será a luz refletida. Adicionalmente, materiais apresentam uma componente emissiva que simula luz proveniente do próprio material. Esta emissão, no entanto, não adiciona luz à cena.

Nas componentes de cor especificados para a luz os números correspondem a uma porcentagem da intensidade total para cada cor. Se os valores R, G e B para a cor da luz são 1, a luz é branca com o maior brilho possível. Se os valores são 0.5 a cor ainda é branca, mas possui metade da intensidade, por isso parece cinza. Se R=G=1 e B=0, a luz parece amarela. A função OpenGL utilizada para definir uma luz é:

```
glLightfv(número, tipo, vetor)
```

onde *número* identifica o número da fonte de luz, que pode variar de GL_LIGHT0 a GL_LIGHT8; *tipo* identifica o tipo de luz, entre GL_AMBIENT, GL_DIFFUSE e GL_SPECULAR, dentre outras e cujas componentes serão descritas por *vetor*. Tipo também pode indicar a posição (GL_POSITION) descrita por *vetor*.

O exemplo a seguir descreve uma fonte de luz com componente ambiente verde e difusa azul localizada na posição xyz = (1,1,1). (O último valor dos vetores indica a componente alfa ou a coordenada *w* do sistema de coordenadas homogêneas [8].)

```
GLfloat light_ambient[] = { 0.0, 1.0, 0.0, 1.0 };
GLfloat light_diffuse[] = { 0.0, 0.0, 1.0, 1.0 };
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

Para habilitar ou desabilitar a fonte de luz devem ser utilizadas as funções:

```
glEnable(GL_LIGHTx)
glDisable(GL_LIGHTx)
```

Para os materiais, os números correspondem às proporções refletidas das cores. Se R=1, G=0.5 e B=0 para um material, este material reflete toda luz vermelha incidente, metade da luz verde e nada da luz azul. Assim, de modo simplificado, a luz que chega no observador é dada por (LR.MR, LG.MG, LB.MB), onde (LR, LG, LB) são os componentes da luz e (MR, MG, MB) os componentes do material. A função OpenGL utilizada para definir as propriedades de material é:

```
glMaterialfv(face, tipo, dado)
```

onde *face* identifica o lado do objeto a ser considerada para a propriedade do material dentre GL_FRONT, GL_BACK, ou GL_FRONT_AND_BACK; *tipo* identifica o tipo de propriedade e *dado* o valor da propriedade selecionada.

Para habilitar ou desabilitar a fonte de luz devem ser utilizadas as funções:

```
glEnable(GL_LIGHTx)
glDisable(GL_LIGHTx)
```

O código a seguir define uma fonte de luz e estabelece propriedades para o material de um bule. O resultado pode ser observado na Figura 15.

```
void Ilumina()
{

  // propriedades da fonte de luz
  GLfloat luzAmbiente[4]={0.2,0.2,0.2,1.0};
  GLfloat luzDifusa[4]={0.8,0.8,0.8,1.0};
  GLfloat luzEspecular[4]={1.0, 1.0, 1.0, 1.0};
  GLfloat posicao [] = {100.0, 100.0, 80.0, 1.0 };
```

```
 // propriedades de material do objeto
 GLfloat posEspecular[4]={1.0,1.0,1.0,1.0};
 GLint matEspecular = 50;

 // refletância do material
 glMaterialfv(GL_FRONT,GL_SPECULAR, posEspecular);
 // concentração do brilho
 glMateriali(GL_FRONT,GL_SHININESS,matEspecular);

 // Ativa o uso da luz ambiente
 glLightModelfv(GL_LIGHT_MODEL_AMBIENT, luzAmbiente);

 glLightfv(GL_LIGHT0, GL_AMBIENT,  luzAmbiente);
 glLightfv(GL_LIGHT0, GL_DIFFUSE,  luzDifusa);
 glLightfv(GL_LIGHT0, GL_SPECULAR, luzEspecular );
 glLightfv(GL_LIGHT0, GL_POSITION, posicao);

 glLightModelfv(GL_LIGHT_MODEL_AMBIENT, luzAmbiente);

 // habilita a luz, material e z-buffer
 glEnable(GL_COLOR_MATERIAL);
 glEnable(GL_LIGHTING);
 glEnable(GL_LIGHT0);
 glEnable(GL_SMOOTH);
 glEnable(GL_DEPTH_TEST);
}
```

Figura 15: Exemplo de aplicação de fontes de luz e propriedades de material na geração de uma cena.

8 Considerações finais

Este artigo apresenta um breve tutorial sobre a geração de cenários tridimensionais interativos utilizando ferramentas de domínio público como a linguagem GNU C, as bibliotecas OpenGL, GLU e GLUT e o pacote de modelagem Blender. Como destaque foram

apresentados os métodos utilizados para a geração dos pares estereoscópicos para a visualização tridimensional destes cenários. Esses métodos podem ser utilizados com *hardware* de baixo-custo, bem como em plataformas avançadas independentemente de sistema operacional.

Informações adicionais podem ser encontradas nas páginas WEB oficiais da OpenGL em http:://www.opengl.org e do Blender em http://www.blender.org ou http://www.blender.com.br (Blender Brasil).

Referências

[1] Woo, M.; Neider, J.; Davis, T.; Shreiner, D. OpenGL Programming Guide, 3a edição. Addison Wesley, 1999.

[2] Shreiner, D. OpenGL Reference Manual, 3a edição. Addison Wesley, 1999.

[3] Kilgard, M. J. OpenGL Programming for the X Window System. Addison Wesley, 1996.

[4] Roosendaal, T e Selleri, S. The Official Blender 2.3 Guide: Free 3D Creation Suite for Modeling, Animation, and Rendering. 2004.

[5] Netto, A.V., Machado, L.S., Oliveira, M.C.F., Realidade Virtual. Visual Books, 2002.

[6] Hodges, L.F. Tutorial: Time-Multiplexed Stereoscopic Computer Graphics. IEEE Computer Graphics & Applications, 12(3):20-30, Mar. 1992.

[7] Stereographics Developers' Handbook. Stereographics Corporation. 1997. Disponível online em: http://www.stereographics.com.

[8] Foley, J. et al. Computer Graphics: principles and practice. 2a edição. Addison Wesley, 1990.

Referências Adicionais

Azevedo, E.; Conci, A. Computação Gráfica - Teoria e Prática; Campus, 2004.

Hearn, D; Baker. Computer Graphics with OpenGL. Prentice Hall, 2003.

Lin, N. Linux 3D Graphics Programming. Wordware Publishing, 2001.

Lin, N. Advanced Linux 3D Graphics Programming. Wordware Publishing, 2001.