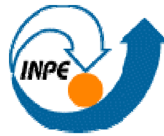



MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

# RECONSTRUÇÃO GRÁFICA TRIDIMENSIONAL DE EDIFICAÇÕES URBANAS A PARTIR DE IMAGENS AÉREAS

RELATÓRIO PARCIAL DE PROJETO DE INICIAÇÃO CIENTÍFICA
(PIBIC/CNPq/INPE)

Daniel Moisés Gonzalez Clua (UNIVAP, Bolsista PIBIC/CNPq)
E-mail: dmgc29785@yahoo.com.br

Dr. Valdemir Carrara (DMC/ETE/INPE, Orientador)
E-mail: val@dem.inpe.br

Julho de 2007

SUMÁRIO

**CAPÍTULO 1 – INTRODUÇÃO**

1.1 Organização do documento

**CAPÍTULO 2 – DESENVOLVIMENTO**

2.1 – Formulação geométrica
2.2 – Visualização da reconstrução
2.3 – Implementação de Menus
2.4 – Leitura de Parâmetros
2.5 – Diálogo Inicial

**CAPÍTULO 3 – ALGORITMOS PARA RECONSTRUÇÃO**

3.1 – Triangularização de Polígonos Planos
3.2 – Algoritmo para a Textura das Laterais dos Edifícios
3.3 – Algoritmo para Correção da Iluminação na Imagem
3.4 – Algoritmo para a Eliminação da Sombra do Edifício no Solo
3.5 – Algoritmo para Sombreamento das Faces do Edifício
3.6 – Edição de Edifícios Construídos
3.7 – Cenas Noturnas

**CAPÍTULO 4 – CONCLUSÕES**

**REFERÊNCIAS BIBLIOGRÁFICAS**

**APÊNDICE A – FUNÇÕES DO OPENGL**

A.1 – Comandos para configuração da janela de visualização
A.2 – Criação de Objetos
A.3 – Movimento e Rotação
A.4 – Aplicação de Texturas
A.5 – Iluminação

**APÊNDICE B – ENTRADA E GRAVAÇÃO DE DADOS**

B.1 – Entrada de Dados
B.2 – Gravação de Dados

## RESUMO

Este trabalho, iniciado em abril de 2005, teve como objetivo o desenvolvimento de algoritmos para a obtenção das dimensões de construções de edifícios a partir do processamento de imagens aéreas de alta resolução. Estas dimensões são posteriormente utilizadas para compor objetos gráficos tridimensionais utilizando a própria textura obtida da imagem. No presente trabalho, os vértices das construções são fornecidos por um dispositivo apontador. O trabalho compreende também a correção da iluminação em função do ângulo de elevação solar, a eliminação de sombra na textura e a alteração das imagens e da iluminação para simular um ambiente noturno da cena. A metodologia utilizada no desenvolvimento do trabalho envolve a aplicação de geometria analítica e vetorial no desenvolvimento de algoritmos para compor as dimensões das construções tridimensionais. O algoritmo foi desenvolvido em linguagem C++, com visualização realizada por meio de OpenGL. Os principais planos para o trabalho envolveram: elaborar uma revisão bibliográfica a respeito da elaboração poligonal e geometria vetorial, familiarizar-se com técnicas de elaboração poligonal do tipo OpenGL e DirectX, desenvolver algoritmos para composição de objetos geométricos simples e para a manipulação destes em imagens, e para efetuar transformações e correções em imagens. Tais algoritmos foram implementados para compor objetos gráficos tridimensionais a partir das informações obtidas das imagens, com aplicação de texturas.

# ABSTRACT

The main objective of this work was to develop algorithms to reconstruct, in a 3D graphic environment, the urban buildings based on high-resolution aerial images taken from satellites or airborne cameras. The image also serves as a graphical texture to compose the virtual environment. Building dimensions, mainly height, are calculated based on points in the image, previously selected by a pointer device (mouse). The texture image is processed in order to correct for intensity due to Sun's elevation angle, to eliminate or reduce the building shadow on ground and to simulate a nocturne ambient of the scene. The algorithm was programmed in C++, and OpenGL was employed to visualize the reconstructed buildings. The main plans for this work involved: a bibliographic revision regarding OpenGL and DirectX polygonal elaboration, development of algorithms for simple geometric objects composition and manipulation of these on images and for making transformations and corrections on images. Such algorithms were implemented to compose three-dimensional graphic objects based on information gathered from the images, with texture application.

# CAPÍTULO 1

# INTRODUÇÃO

A construção de ambientes gráficos tridimensionais (ambiente virtual) no computador tem encontrado diversas aplicações recentemente. Embora ambientes virtuais sejam freqüentes em jogos, cenários tridimensionais que representam ambientes reais são cada vez mais empregados no turismo, no urbanismo, no tratamento de distúrbios nervosos (fobias) e simuladores de vôo, veículos e máquinas. Também é usual, em tais ambientes, a reconstrução virtual de partes de cidades ou mesmo cidades inteiras. Uma vez que esta reconstrução pode ser bastante complexa e difícil, em virtude dos vários cenários existentes, buscam-se ferramentas automáticas ou semi-automáticas para tal reconstrução (VTP 2005, Carmenta 2005). Imagens aéreas e imagens obtidas por satélites com alta resolução são utilizadas usualmente, o que permite reconstruir grandes áreas com poucas imagens. O processo de reconstrução virtual pode ser dividido em duas partes: na primeira buscam-se formas automáticas ou semi-automáticas de extração de características (ou informações) das imagens para detectar as construções, e na segunda efetua-se a reconstrução propriamente dita. Pretende-se neste trabalho investir na reconstrução de ambientes urbanos (edifícios e casas) a partir de informações previamente fornecidas a um programa computacional, o que caracteriza um método semi-automático.

Portanto, o objetivo deste trabalho é elaborar um programa que permite reconstruir espacialmente e visualizar construções e edifícios a partir de imagens aéreas. A linguagem usada é C++, com recursos de OpenGL para a visualização.

O OpenGL é definido como “uma interface em *software* para *hardware* gráfico”. Em essência, é uma biblioteca de gráficos tridimensionais e modelagem, portátil e rápida, o que permite gerar imagens interativas em tempo-real. O OpenGL não é uma linguagem, como C ou C++, mas sim uma biblioteca de funções pré-programadas. Na realidade não existe um “programa em OpenGL” mas sim um programa que o desenvolvedor escreve usando o OpenGL como um de seus APIs.

As principais funções a serem implementadas para efetuar a reconstrução virtual são:

1) Exibir uma imagem e permitir ao usuário o fornecimento de pontos (vértices) que definem a construção, que irá definir um prisma vertical com base não necessariamente retangular.
2) Obter parâmetros que permitam avaliar a altura da construção a partir da sombra projetada na imagem ou da própria altura projetada (em caso de vista em perfil).
3) Extrair a textura a ser aplicada ao topo da construção da própria imagem.
4) Utilizar imagens adicionais para crias as texturas das laterais da construção.
5) Eliminar ou minimizar a imagem projetada do edifício na textura do solo.
6) Exibir o resultado na forma tridimensional.

## 1.1 Organização do documento

Este relatório parcial foi dividido em três capítulos: no Capítulo 2 – Desenvolvimento – são descritos os métodos e funções estudadas e utilizadas no trabalho; no Capítulo 3 – Algoritmos para reconstrução, são apresentados as principais contribuições deste trabalho e alguns resultados; no Capítulo 4 são apresentadas as conclusões finais. Apresenta-se no Apêndice A uma descrição sucinta das principais funções do OpenGL utilizadas no desenvolvimento do trabalho e, no Apêndice B, as funções de entrada de dados (leitura do teclado e mouse), bem como o formato utilizado para gravação dos edifícios.

# CAPÍTULO 2

# DESENVOLVIMENTO

Para cumprir os objetivos do trabalho e para implementar as funções descritas no capítulo precedente, procedeu-se a um estudo dividido em duas partes: na primeira formulam-se as relações geométricas que permitem efetuar a reconstrução a partir de informações colhidas das imagens, e, na segunda, implementam-se o equacionamento em OpenGL de forma a visualizar o resultado.

## 2.1 – Formulação geométrica

A formulação geométrica possui por objetivo determinar a altura dos edifícios a partir de informações da própria imagem. Tomando como exemplo a imagem mostrada na Figura 2.1, deseja-se obter a altura do edifício em unidades de *pixel*, representada pela magnitude do vetor $H$ perpendicular ao plano da imagem, e cuja projeção no plano da imagem é o vetor $h$.

Nota-se que, embora a altura do edifício não seja vista na sua real dimensão na imagem, por outro lado, a sombra tem sua verdadeira dimensão (ou bem próxima dela), se for suposto que a câmera tenha seu eixo perpendicular ao solo, como ilustra a Figura 2.2. Contudo, se o ângulo $\alpha$ que representa a elevação do Sol sobre o horizonte for conhecido, então a altura $H$ do edifício pode ser obtida por meio do comprimento da sombra do edifício na imagem, ou seja, do módulo do vetor $s$, conforme mostra a figura 2.1, resultando então:

$$|H| = |s| \tan \alpha$$

Caso, porém, este ângulo não seja conhecido, pode-se ainda assim estimá-lo desde que se conheça a altura real de alguma construção e a verdadeira resolução da imagem, isto é, o tamanho de cada *pixel*, ou ainda o comprimento da sombra deste edifício. Seja então $H'_m$ a altura em metros de uma determinada construção conhecida e

seja |*s*'| o comprimento de sua sombra na imagem (em *pixels*), também visto na Figura 2.1. Se *R* for a resolução (em metros/*pixel*), então a altura do edifício *H*' será dada por:

$$\left|H'\right| = \frac{H'_m}{R} \text{ (em } pixels\text{).}$$

Fig. 2.1 – Imagem aérea e os principais pontos que devem ser utilizados no processo de reconstrução. O contraste da imagem foi reduzido artificialmente para evidenciar os pontos a serem fornecidos externamente.

O ângulo de elevação do Sol pode agora ser obtido por:

$$\alpha = \arctan\left(\frac{\left|H'\right|}{\left|s'\right|}\right),$$

e poderá ser utilizado na reconstrução de outros edifícios da mesma imagem. Caso a resolução não seja conhecida, pode-se medir a distância real em linha reta horizontal entre dois pontos conhecidos de uma imagem e esta mesma distância na própria imagem. A relação entre as duas distâncias irá fornecer a resolução.

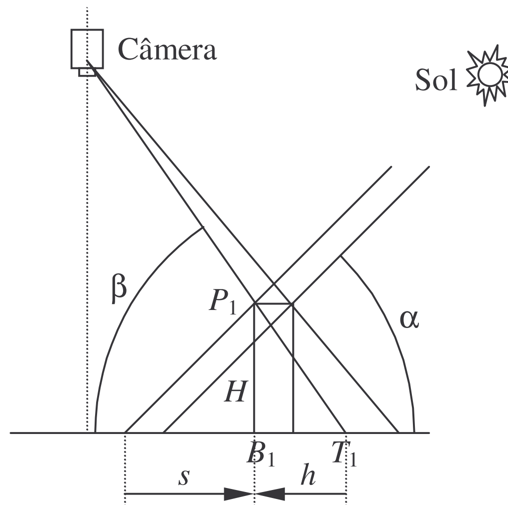



Fig. 2.2 – Vista lateral da geometria do problema.

Se o comprimento *s* da sombra não estiver disponível para um dado edifício (por exemplo, se a sombra for projetada sobre uma outra construção), pode-se estimar a altura tendo como base a altura projetada, desde que seja conhecida a altura *H'* e sua projeção *h'* de um outro edifício próximo. Neste caso emprega-se uma regra de proporcionalidade:

$$|H| = |h| \frac{|H'|}{|h'|}$$

É necessário que as duas construções estejam próximas, caso contrário os efeitos da projeção em perspectiva podem influir nos resultados, isto é, o ângulo β, mostrado na Figura 2.2 deve ser aproximadamente igual em ambos os edifícios.

Sendo então fornecidos externamente os pontos $T_1$, $T_2$, $T_3$ e $T_4$, que delimitam o topo da construção, e o vetor *h* que representa a projeção da altura do edifício na imagem, as coordenadas dos pontos da base da construção, $B_i$, são facilmente obtidas

pela relação $B_i = T_i + h$. Estes pontos serão utilizados para a posicionar a base do edifício virtual. Num sistema de eixos cartesianos, onde $x$ e $y$ formam o plano da imagem e $z$ é perpendicular a este plano, os pontos $P_i$ do topo do edifício são obtidos por $P_i = B_i – H$.

## 2.2 – Visualização da reconstrução

O início do trabalho consistiu na familiarização dos principais comandos de programação em OpenGL. Estes envolviam a criação de uma janela simples, passando-se informações tais como a cor de fundo, o seu tamanho, sua escala, modo de visualização, etc. Os principais comandos OpenGL utilizados na visualização são comentados separadamente, em função do tipo de saída provocada. As informações coletadas para a configuração adequada do OpenGL e para a preparação do programa vieram do livro OpenGL Super Bible (Wright, 2000) e de programas tutores encontrados na Web (Neon-Helium Productions, 2005).

Para o correto funcionamento do OpenGL é necessário a inclusão no programa dos arquivos `gl.h`, `glu.h`, e `glaux.h`, além da biblioteca `windows.h` que contêm valores previamente associados a constantes. São também definidas nestes arquivos as especificações de tipos de variáveis, uma vez que o OpenGL, por ser multi-plataforma, estabelece regras para os tipos usuais, como `GLfloat`, `GLint`, etc. Estes arquivos em geral são fornecidos com o compilador C++, mas podem ser obtidos facilmente na *web* caso contrário. Além disso as seguintes bibliotecas devem ser incluídas no projeto para se poder utilizar os comandos de OpenGL no ambiente Windows: `OpenGL32.lib`, `GLu32.lib`, e `GLaux.lib`.

Os principais comandos do OpenGL utilizados na visualização do processo de reconstrução de edifícios são mostrados no Apêndice A. No Apêndice B são descritas funções do sistema operacional Windows para entrada de dados por meio do teclado e *mouse*.

## 2.3 – Implementação de Menus

Para maior facilidade na utilização do programa foram implementados alguns menus, com opções antes acessíveis somente pelo teclado. Dois tipos de menus foram implementados no programa. O primeiro é do tipo barra de menu e é exibido na parte superior da janela, como mostra a Figura 2.3. O menu permite o acesso às opções de sair do programa, salvar ou carregar dados, mudar o tipo de movimentação na imagem feita pelo mouse, iniciar a leitura de parâmetros para construção do edifício e, uma vez passados os pontos, realizar a construção. Estas opções também podem ser acessadas mais facilmente pelos botões da janela mostrada na Figura 2.4. Ainda foi criado uma terceira janela de menu, utilizado para edição de edifícios construídos e que será melhor explicado na sessão 3.6.

Fig. 2.3 – Barra de menu.

Fig. 2.4 – Menu de botões.

## 2.4 – Leitura de Parâmetros

O objetivo da leitura de parâmetros é permitir que o usuário, através do mouse, indique os pontos para a reconstrução de edifícios, a partir da exibição de uma imagem aérea na tela. Para tanto a imagem é aplicada como textura – usando os métodos descritos anteriormente – a um objeto retangular de mesmas proporções em vista frontal. O programa permite também que o usuário mova, acerque ou distancie a imagem, através de entradas do teclado. O programa é dividido em duas janelas. Ambas mostram a mesma imagem, uma usando projeção em perspectiva e outra usando projeção paralela. A projeção em perspectiva serve para visualização e pode ser movimentada sob diferentes ângulos. A projeção paralela pode ser acercada ou afastada, mas o ângulo de rotação é fixo, já que é nela que é feita a leitura de pontos para construção do edifício. Uma vez construído, a representação do edifício é exibida na janela com projeção em perspectiva, para melhor visualização.

Na implementação atual, o usuário deve fornecer um número qualquer de pontos para o topo do edifício (como por exemplo, os pontos $T_1$ a $T_4$, da figura 2.1), dois pontos que representem o vetor da altura do edifício ($h$) e mais dois pontos que representam o vetor da sombra ($s$). Uma vez que a leitura das coordenadas destes pontos, efetuada pelo mouse, estão em unidades de *pixel* da janela, é então necessário convertê-las para unidades de *pixel* da imagem por meio de uma transformação linear:

$$T_{xi} = a \; x_i + d_x$$

$$T_{yi} = a \, y_i + d_y$$

onde $a$ é a ampliação (fator de escala) utilizada na visualização. Estes valores são então atribuídos a dois vetores, um para as coordenadas no eixo $x$ e outra para as do eixo $y$. Ambos possuem os valores de todos os pontos que representam o topo do edifício e também os dos vetores altura e da sombra. O vetor da altura serve para calcular o deslocamento que o programa deve aplicar ao objeto para que ele se origine da base do edifício na imagem e não do topo. Já o vetor da sombra é necessário para calcular a

altura do edifício, através de um cálculo com o ângulo de elevação do sol (ver Fig. 2.2). Passados estes pontos, o programa constrói o objeto que representa o edifício, usando os valores armazenados das coordenadas, como visto nas Figuras 2.5 e 2.6.

Para possibilitar a reconstrução de vários edifícios, foi criada uma classe para representá-los. Deste modo, podem ser instanciados diversos objetos desta classe, sendo que cada um contém todas as suas informações necessárias, como a sua posição no plano, os vértices entrados pelo usuário, a altura, etc. Além de facilitar a construção dos edifícios na cena, a classe irá permitir que, mais adiante, o usuário possa selecionar um edifício já construído e modificá-lo.

Fig. 2.5 – Pontos recebidos para a construção do edifício.

Fig. 2.6 – Edifício reconstruído.

## 2.5 – Diálogo Inicial

O programa desenvolvido utiliza informações que podem variar de acordo com o desejo do usuário: a textura que será utilizada para o solo, a posição da fonte de luz (Sol) e o local onde as imagens estão armazenadas. Além disso, é possível iniciar o programa a partir de um novo projeto ou carregando uma cena previamente salva. Para facilitar estas configurações, foi criado um diálogo utilizando o Microsoft Visual Basic que proporciona estas opções.

Na tela inicial do programa, mostrada na Figura 2.7, pode-se optar entre iniciar ou carregar um projeto. Se a opção de criar um novo projeto é escolhida, uma nova janela é aberta (Figura 2.8) pela qual se deve selecionar a imagem para a textura do solo e o diretório das imagens (sendo que o padrão é o próprio local onde se encontra o programa). Utilizando duas imagens, deve-se posicionar a fonte de luz no local desejado nas três coordenadas espaciais $x$, $y$ e $z$. Quando as posições são modificadas, o valor do ângulo solar é mostrado em graus e também graficamente, por meio de linhas. Estas posições serão depois utilizadas para a formulação geométrica (sessão 2.1) e iluminação da cena (sessão 3.5). Ainda pode-se selecionar entre corrigir ou não a iluminação na textura do solo (sessão 3.3).

Terminada a configuração, ou carregando um projeto existente, o programa de construção de edifícios é aberto, utilizando as informações coletadas para construir a cena.

Fig. 2.7 – Tela inicial do programa.

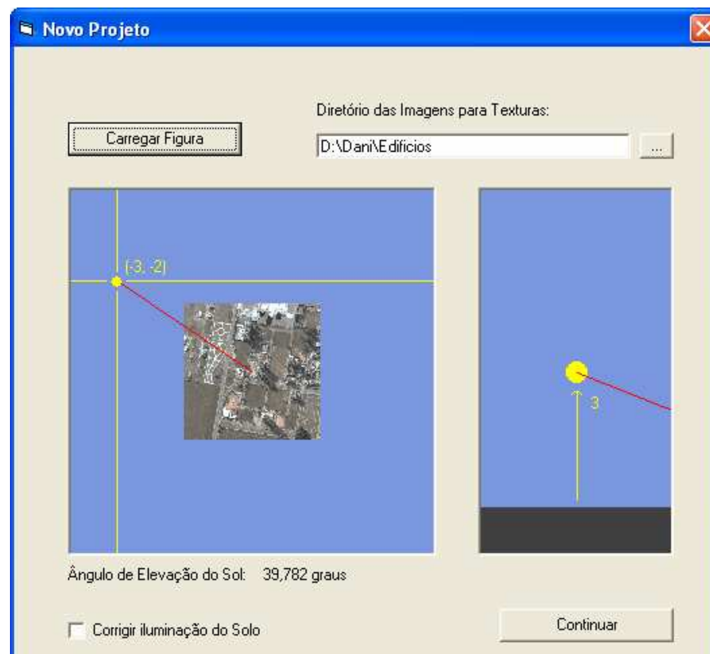



Fig. 2.8 – Janela de criação de um novo projeto.

## CAPÍTULO 3

## ALGORITMOS PARA RECONSTRUÇÃO

### 3.1 – Triangularização de Polígonos Planos

A construção de objetos em OpenGL deve ser feita exclusivamente com triângulos planos. Portanto, para possibilitar a construção de edifícios com qualquer formato – não apenas retangulares – foi necessário desenvolver um algoritmo para realizar a triangularização do polígono definido pelos pontos passados pelo usuário, que representa o contorno do topo do edifício.

O problema da triangularização de polígonos planos consiste em obter um conjunto de triângulos $T_i$, com $i = 1, 2, ..., l \leq n - 2$, tal que este conjunto corresponda a uma divisão do polígono formado por uma seqüência de coordenadas na forma $P_i = (x_i, y_i)$, com $i = 1, 2, ..., n$, $(n \geq 3)$ que definem um polígono fechado como mostra a figura 3.1.

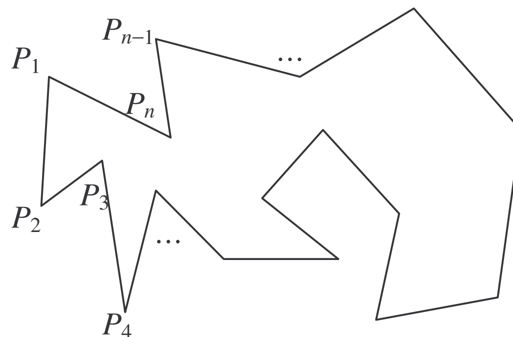



Fig. 3.1 – Pontos $P_1$ a $P_n$, que definem o polígono.

A solução adotada neste algoritmo será eliminar triângulos do polígono um a um, a partir dos triângulos externos, isto é, tais que duas de suas arestas sejam arestas consecutivas do polígono. Desta forma os triângulos escolhidos são formados por 3 vértices em seqüência, como $P_4$, $P_5$, $P_6$, por exemplo. Uma vez eliminado um triângulo, será constituído um novo polígono com um vértice a menos. O processo será então repetido para este novo polígono, até que reste apenas um triângulo.

O algoritmo será dividido em duas partes. Na primeira, deve-se modificar o polígono original para que satisfaça condições necessárias para a segunda parte. Na segunda, os triângulos serão removidos. A primeira destas condições é que o polígono seja formado por 3 ou mais pontos. Além disso, nenhuma aresta deve apresentar comprimento nulo, isto é, tal que $P_i$ seja igual a $P_{i+1}$. Em seguida, deve-se verificar se não existem vértices "supérfluos", isto é, inseridos numa aresta, fazendo com que haja 3 ou mais pontos em seqüência alinhados a uma mesma reta. Outra condição necessária é que o polígono não seja "dobrado", isto é, tal que não ocorra uma interseção entre duas arestas quaisquer. Finalmente, a última condição é que o sentido de percurso dos vértices seja único, horário ou anti-horário. Será adotado aqui que o sentido normal é o anti-horário, o que significa que se o percurso for horário a ordem dos vértices deverá ser invertida para gerar um polígono anti-horário, como mostra a Figura 3.2:

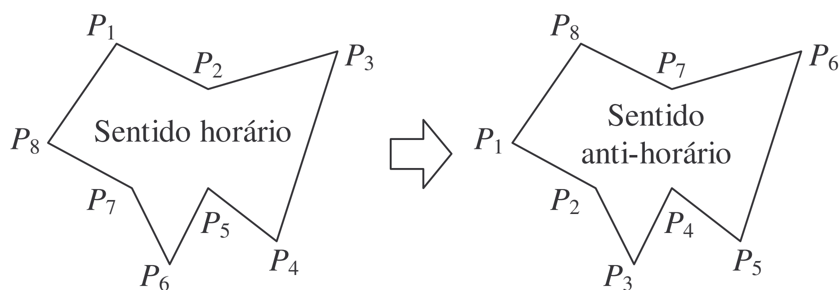



Fig. 3.2 – Inversão do sentido dos pontos que definem o polígono.

Foram desenvolvidos algoritmos para testar e corrigir todas as condições apontadas acima. A condição de que o polígono não seja "dobrado" sobre si mesmo foi inserida no código que armazena os pontos fornecidos pelo usuário. Este algoritmo impede a tentativa de se fornecer um ponto cuja aresta formada com o ponto anterior cruze qualquer outra aresta já fornecida.

A segunda parte do algoritmo consiste na subdivisão do polígono em triângulos, de tal forma que, no final, o conjunto destes triângulos forme o polígono original. Ao todo serão removidos $n$ – 2 triângulos do polígono original, sendo que $n$ é o número de vértices do polígono. A figura abaixo ilustra um polígono e a ordem de retirada dos triângulos válidos.

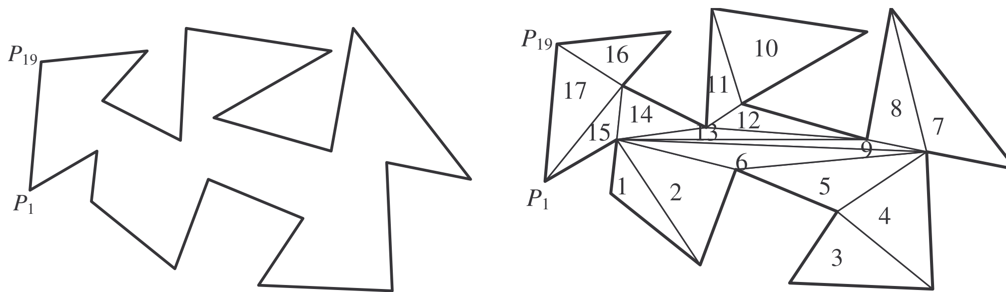



Fig. 3.3 – Subdivisão do polígono em triângulos.

Os triângulos a serem removidos são selecionados com base no produto vetorial de duas arestas consecutivas. Como o polígono é anti-horário, então se o produto vetorial for positivo o triângulo a ser removido faz parte do polígono. Caso contrário trata-se de um triângulo externo e o polígono é côncavo neste local. É necessário cuidar ainda para que o triângulo a ser eliminado não contenha um vértice qualquer do polígono em seu interior, o que poderia causar erros na triangularização. Caso isto ocorra, o algoritmo procura então os próximos triângulos a serem removidos até que encontre um cujo interior não contenha vértices.

A Figura 3.4 ilustra o processo de triangularização realizado pelo algoritmo num polígono qualquer. Os triângulos foram coloridos aleatoriamente na figura para facilitar a sua identificação.

Fig. 3.4 – Processo final do algoritmo de triangularização.

### 3.2 – Algoritmo para a Textura das Laterais dos Edifícios

Para dar uma aparência mais real aos edifícios tridimensionais construídos, usou-se um método para aplicar texturas às suas laterais. Estas, porém, ao contrário do topo, nem sempre estarão totalmente visíveis nas imagens. Por isso, usam-se outras imagens para simular estas texturas e não a própria do edifício.

Estas imagens armazenam desenhos de diferentes tipos de padrões que simulam laterais de edifícios e foram construídas com o uso de um editor de imagens comum (*MS Paint*). Para melhor eficiência e resultado, o tamanho destas imagens é pequeno – aproximadamente a extensão de um andar. O algoritmo repete então esta textura o número de vezes necessário para cobrir toda a área de cada face do edifício. Para dar um maior realismo, procurou-se manter uma escala compatível com a real. Por exemplo, se um edifício real possui 15 andares, espera-se que sua reconstrução digital tenha aproximadamente o mesmo número de andares. Foi necessário, então, fazer um algoritmo que calcula o tamanho da face e determina quantas vezes a textura deve ser aplicada, tanto na vertical como na horizontal. Para o comprimento, isto é feito calculando-se a distância euclidiana entre dois vértices consecutivos da base (ou do topo). Para a altura não é necessário nenhum cálculo, já que cada edifício armazena, em uma variável, a sua respectiva altura que será a mesma para todas as suas faces. Também se deve pré-definir qual será o tamanho da textura aplicada (*tamTextura*). Isto depende do tamanho da imagem e da escala em que se está trabalhando, mas deve-se usar um valor que gere um bom resultado, de modo que a textura não fique nem muito pequena nem muito grande em relação ao edifício.

Tendo o comprimento e altura tanto do edifício quanto da textura, é possível calcular quantas vezes está terá que ser repetida para cobrir toda a lateral. Como este número deve ser inteiro, faz-se um arredondamento do valor calculado para o número inteiro mais próximo. Para isto, usam-se as funções `floor()` e `ceil()` da biblioteca matemática do C++. O resultado será que a última cópia da textura será esticada ou

comprimida, mas como foi calculado qual destas possibilidades está mais perto do tamanho real, a distorção não será tão significativa.

Com os valores dos números de repetições, dividem-se as distâncias vertical e horizontal calculadas anteriormente pelo número de repetições, o que resultará no tamanho que cada cópia da imagem terá. A partir do ponto inicial da lateral, e a cada aplicação da imagem, a distância na horizontal é incrementada. Quando o final da face é atingido, retorna-se ao lado inicial e incrementa-se a distância na vertical. Isso é repetido sucessivamente até que a face seja totalmente preenchida pela textura, como ilustra a Figura 3.5. Este conceito de repetição de texturas permite que se construa uma lateral completa de um edifício a partir de uma ou poucas imagens elementares (contendo apenas uma janela, por exemplo). Além disso, podem ser criadas diferentes padronizações com imagens e selecioná-las de forma aleatória em cada edifício reconstruído para que não fiquem todos iguais.

*A* *b*

Fig. 3.5 – Exemplo de textura (*a*) e sua aplicação em um edifício (*b*).

**3.3 – Algoritmo para Correção da Iluminação na Imagem**

Deve-se considerar que, além da iluminação do OpenGL aplicada durante a reconstrução tridimensional, a imagem do solo possui a iluminação original do momento em que a fotografia foi tirada. Uma possibilidade para evitar esta “dupla iluminação” seria não aplicar a iluminação do OpenGL no solo, apenas nos edifícios. Contudo, como a cena recriada não necessariamente estará nas mesmas condições de

luz, deve-se buscar uma maneira de "anular" a iluminação original da imagem, de modo que esta esteja influenciada apenas pela luz artificial.

Para isto, divide-se o valor de cada componente de cor (vermelho, verde e azul) de cada *pixel* da imagem, pelo seno do ângulo de elevação do Sol:

```
valor = valor / sin(alphaRad);
```

Sendo que *valor* é o valor numérico inteiro, de 0 a 255, que representa a quantidade de um dos componentes de cor do *píxel*. A função `sin()` é encontrada na biblioteca *math.h* e calcula o seno de um ângulo dado em radianos. A variável `alphaRad` armazena o ângulo de elevação do Sol, em radianos.

A Figura 3.6 mostra um exemplo da diferença entre a imagem do solo sendo ou não influenciada pela correção.

*a* *b*

Fig. 3.6 – Imagem do solo sem (a) e com (b) correção de iluminação utilizando um ângulo de elevação do Sol de 20 graus.

## 3.4 – Algoritmo para a Eliminação da Sombra do Edifício no Solo

Uma vez que a reconstrução dos edifícios é realizada graficamente, deseja-se que a iluminação na cena reconstruída possa ser alterada sob ação externa. Em outras palavras, deseja-se que as sombras dos objetos na cena acompanhem o movimento da posição da fonte de luz artificial quando esta for modificada, de modo que a projeção

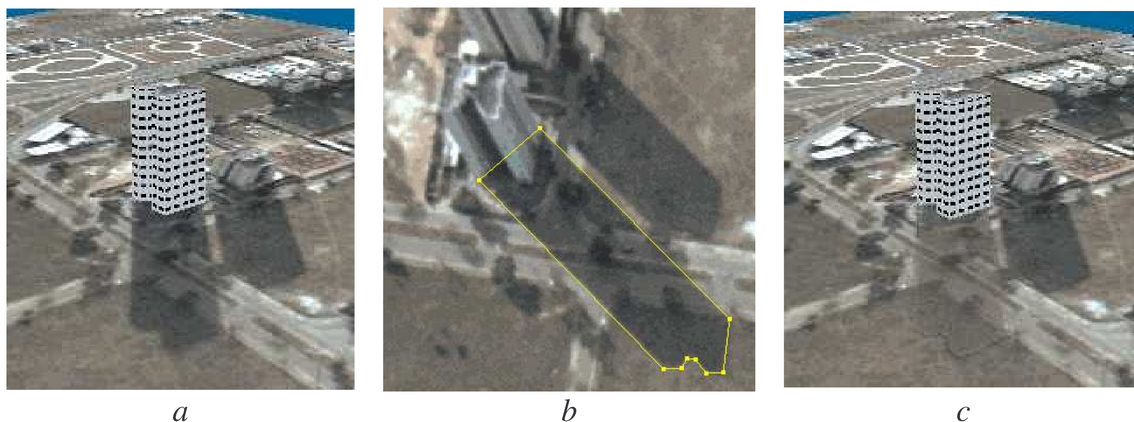
tenha um aspecto real. Isto cria um problema, pois em geral as imagens aéreas que servem como textura do solo apresentam sombras naturais dos edifícios. Para que não houvesse uma "sobreposição" de sombras (uma natural e outra artificial), o que causaria uma indefinição na direção real de iluminação, foi necessário encontrar uma maneira de se "eliminar" a sombra natural. Admitindo-se que a área que a sombra ocupa no solo (polígono de sombra) seja conhecida, pode-se reaplicar a própria textura do solo a este local com uma claridade maior que a do solo. Isto é feito elevando-se os valores de cores vermelha, verde e azul de cada *pixel*. As quantidades elevadas em cada canal de cor são definidas pelo usuário e devem ser ajustadas empiricamente. Na verdade, cria-se uma nova imagem da textura do solo com os valores dos *pixels* todos alterados, como mostra o seguinte algoritmo:

```
for (int i=0; i<3*Image->sizeX*Image->sizeY; i++){
      int j = i%3;
      int valor = Image->data[i];
      valor+=brilho[j];
      if(valor>255)
            valor=255;
      else if(valor<0)
            valor=0;
      Image->data[i]=valor;
}
```

O laço `for` da primeira linha, percorre todos os *pixels*. Como cada *pixel* armazena três valores (para as cores vermelho, verde e azul), deve-se começar no primeiro, que tem índice zero e terminar em três vezes o número total de *pixels*, que por sua vez é obtido pela multiplicação do tamanho em *X* pelo tamanho em *Y* (a variável `Image` é um ponteiro para a imagem utilizada). A operação `i%3` na segunda linha calcula o resto da divisão de `i` por três. Isto determina a qual cor do *pixel* a variável `i` atual se refere e é armazenado em `j`. Em seguida, armazena-se em *valor* a quantidade de uma cor no *pixel* atual (o vetor `data[]` armazena as informações dos *pixels* da imagem) e soma-se a ele a quantidade de brilho adicional para aquela cor. Deve-se verificar se o valor do *pixel* não está acima de 255 ou abaixo de zero, para não ultrapassar os limites de cor e resultar em efeitos indesejáveis na imagem. Finalmente, o valor do *pixel*

original é substituído pela variável `valor` que agora está incrementada pelo valor `brilho`. Esta variável é controlada pelo usuário e pode ser incrementada ou diminuída de modo a encontrar o melhor resultado. Esta textura é então aplicada ao polígono da sombra.

Aplicando-se a quantidade certa de brilho, obtêm-se o desaparecimento praticamente completo da sombra na imagem original, possibilitando a projeção de sombras dinâmicas na cena (ainda não implementado atualmente). Na presente fase do projeto, o usuário deve indicar os vértices do polígono da sombra. Essa entrada funciona de maneira semelhante à entrada de vértices do edifício, e o polígono passa pelo processo de triangularização de forma idêntica àquela realizada no topo (Seção 3.1). A Figura 3.7 mostra um exemplo no qual a sombra do solo foi minimizada. Percebe-se, neste exemplo, que a completa eliminação da sombra é algo difícil, senão impossível de ser realizada. Contudo a redução do seu efeito na textura do solo irá permitir que a direção de iluminação artificial seja evidenciada sem ambigüidade.



*a* *b* *c*

Fig. 3.7 - Exemplo de edifício com a sombra original (*a*), seleção da área da sombra (*b*) e eliminação da sombra (*c*).

## 3.5 – Algoritmo para Sombreamento das Faces do Edifício

Para dar maior realismo à cena reconstruída, é desejável que as faces do edifício tridimensional sejam diferentemente afetadas pela luz, dependendo de sua orientação. Ou seja, se uma lateral está inclinada em direção à posição do Sol, deve ficar bem iluminada, enquanto que outra que está virada contra o Sol, deve ficar mais escura.

Entretanto, faces que estão contra a luz não devem ficar totalmente escuras. Isto seria desejável apenas em cenários onde há apenas luz fraca (como à noite, por exemplo). Em um cenário diurno, se uma lateral não recebe luz direta do Sol, esta ainda é iluminada indiretamente por outros objetos que recebem a luz e a refletem.

Para permitir este efeito, o OpenGL possui dois tipos de iluminação: difusa e ambiente. A luz difusa é geralmente bem forte e é aquela que vem diretamente da fonte de luz e ilumina apenas objetos que estão em sua direção. A luz ambiente é geralmente mais fraca e representa a reflexão da luz difusa pelos objetos que a recebem e ilumina toda a cena e os objetos de maneira idêntica.

Portanto, laterais do edifício que estão direcionadas à luz recebem a iluminação difusa e ambiente e aquelas que estão contra a luz recebem apenas a luz ambiente. Se alguma está parcialmente virada em direção à luz, esta deve receber parte da iluminação difusa e parte da ambiente. É necessário saber, então, a quantidade de cada iluminação que cada lateral receberá. Fornecendo-se a normal da face (vetor com direção perpendicular à face e sentido apontando para sua frente), o OpenGL pode calcular automaticamente estas quantidades.

Pare encontrar os componentes do vetor normal de uma face, são feitos os seguintes cálculos:

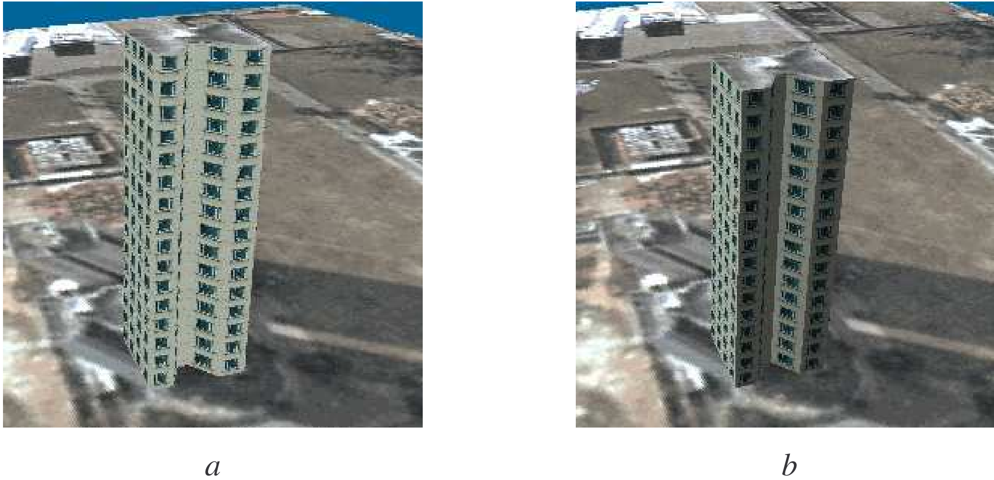
```
float dX = Xf - Xi;
float dY = Yf - Yi;
float dist = sqrt((dX*dX)+(dY*dY));
float nX =  dY / dist;
float nY = -dX / dist;
```

Onde, `dX` e `dY` são, respectivamente, a diferença dos valores finais e iniciais em *X* e *Y* da face (vista de cima), `dist` é a distância do ponto inicial ao ponto final da face, calculada pela fórmula de Pitágoras. As variáveis `nX` e `nY` representam os componentes *X* e *Y* do vetor normal.

Tendo estes componentes, basta informá-los ao OpenGL (através da função `glNormal3f()`), antes da entrada dos vértices para a construção da lateral e a iluminação será calculada automaticamente (o terceiro componente, referente ao eixo *Z*, é mantido em zero, já que nenhuma lateral estará virada para cima ou para baixo):

```
glNormal3f(nX, nY, 0.0f);
```

A Figura 3.8 mostra a diferença entre um edifício sem e com sombreamento nas laterais.



*a* *b*

Fig. 3.8 – Edifício sem (a) e com (b) sombreamento de laterais.

**3.6 – Edição de Edifícios Construídos**

Uma vez reconstruído, ainda pode ser que se queira modificar um determinado edifício, como a sua altura, posição de vértices, textura ou ainda, removê-lo por completo. Para isto, foi criado um menu de edição de edifícios, mostrado pela Figura 3.9.

Ao entrar no modo de edição, um edifício deve ser selecionado por um clique na imagem da cena em visão paralela. O edifício que estiver mais próximo à posição do cursor será selecionado para edição e o menu é ativado.

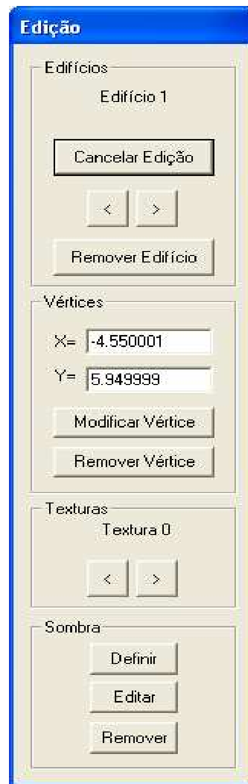


Fig. 3.9 – Menu de Edição.

Na área do menu relativo à "Edifícios", pode-se ver com qual edifício se está trabalhando, cancelar a edição, percorrer pelos demais edifícios existentes na cena ou remover completamente o edifício selecionado.

É possível modificar a posição dos vértices na janela de visualização, posicionando o cursor sobre o vértice desejado, pressionando o botão e movendo-o livremente, como mostra a Figura 3.10. Ao soltar o botão, o programa modifica a posição do vértice selecionado, alterando os valores correspondentes nas matrizes de coordenadas e, em seguida, constrói novamente a cena tridimensional, com o vértice em sua nova posição. Antes desta alteração, contudo, faz-se uma verificação para garantir que não fiquem arestas sobrepostas, da mesma maneira que era feita no momento da construção de um novo edifício.

Fig. 3.10 – Modificação livre de um vértice.

Para maior precisão, pode-se também alterar um vértice no menu de edição, indicando as coordenadas numéricas em X e Y do vértice selecionado e pressionando o botão "Modificar Vértice". Com um vértice selecionado, ainda é possível removê-lo, desde que isto não cause nenhum cruzamento de arestas.

Outra opção do menu é alterar a imagem de textura do edifício selecionado, pressionando os botões de setas, na área de "Texturas". Com eles, é possível percorrer todas as imagens existentes e escolher aquela que melhor se assente ao edifício em questão.

Por último, a edição permite também definir, alterar ou remover a área da sombra projetada do edifício, a fim de minimizá-la, utilizando o algoritmo explicado anteriormente. A alteração da área segue o mesmo funcionamento da alteração de vértices do edifício, onde o usuário pressiona o botão sobre um dos pontos e o arrasta a uma nova posição, havendo sempre a verificação de cruzamento de arestas.

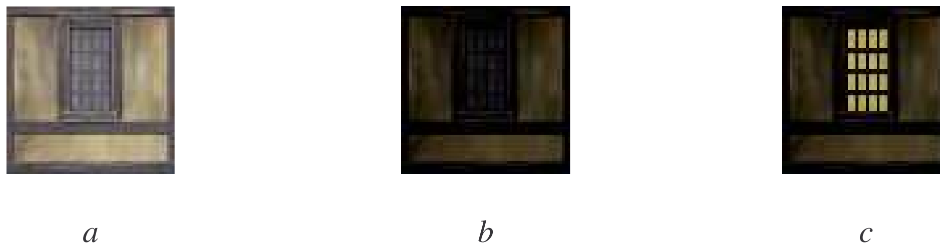
## 3.7 – Cenas Noturnas

Para proporcionar uma maior diversidade na visualização da cena criada, foi criada uma opção que permite alterar a cena entre dia e noite. Ao fazer a alteração para ambiente noturno, primeiramente deve-se reduzir a iluminação ambiente. Além disso, a imagem utilizada para a textura do solo será substituída por uma que represente o mesmo local, mas visto durante a noite, como exemplificado pela Figura 3.11. Para tanto, já deve existir esta segunda imagem no mesmo diretório da primeira.

*a* *b*

Fig. 3.11 – Imagem diurna (a) e noturna (b) de um mesmo local.

A fim de dar maior realismo à cena noturna, também foi feito com que as texturas dos edifícios sejam modificadas. Não apenas são utilizadas imagens mais escuras, como também – para aquelas que contêm janelas – imagens que representam habitações com a luz acesa, como pode ser visto na Figura 3.12.



*a* *b* *c*

Fig. 3.12 – Imagem diurna (a), noturna com a luz apagada (b) e com a luz acesa (c).

Como explicado anteriormente, as texturas das laterais do edifício são repetidas um determinado número de vezes, a fim de cobrir toda a extensão das faces. Quando a cena está em modo noturno, no momento de aplicar cada cópia da imagem, é feita uma escolha aleatória entre aquela com a luz apagada e com a luz acesa, utilizando o seguinte código:

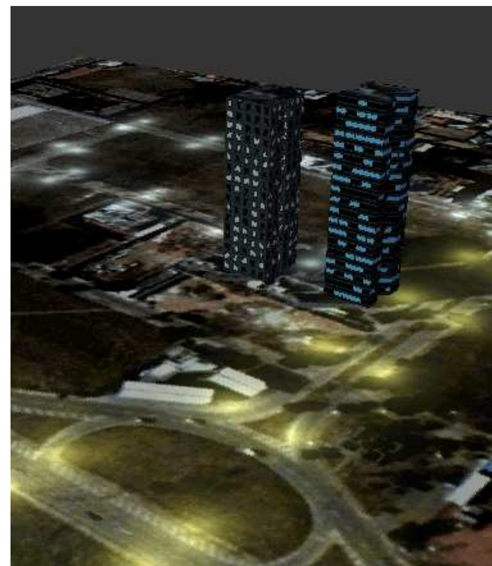
```
if(((int)rand()%2) > 0){
      glBindTexture(GL_TEXTURE_2D, t1);
```

```
}
else{
      glBindTexture(GL_TEXTURE_2D, t2);
}
```

A primeira linha do código gera um número randômico inteiro entre 0 e 1 e verifica se este é maior que zero. Ou seja, existem 50% de chance que a comparação resulte em verdadeiro. Caso isto aconteça, a textura “t1”, que representa a imagem com a luz apagada, será utilizada. Caso contrário será usada “t2”, que armazena a imagem com a luz acesa.

Desta forma, o resultado – exemplificado pela figura 3.13 - é que algumas habitações estarão com a luz apagada e outras com a luz acesa, aproximando-se do que seria esperado em uma cena real.

*a*

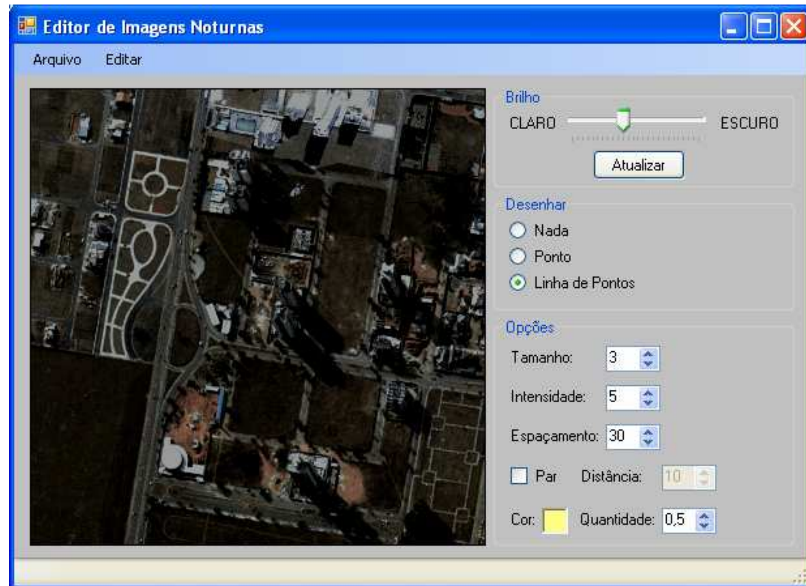


*b*

Fig. 3.13 – Cena em ambiente diurno (a) e noturno (b).

Nota-se, entretanto, que dificilmente poderão ser obtidas duas imagens (diurna e noturna) que correspondam exatamente ao mesmo ponto de vista do local. Portanto, foi elaborado e desenvolvido, em Visual Basic.NET, um pequeno programa externo que, a partir de uma imagem tirada durante o dia, aplica transformações a esta, a fim de simular um ambiente noturno da mesma cena, escurecendo a imagem original e aplicando focos de luz, que representam a iluminação de postes e lâmpadas.

A única janela do programa, mostrada na Figura 3.14, consiste em uma imagem que exibe o resultado das transformações aplicadas à cena original e um menu com por onde é possível selecionar o nível de escurecimento da imagem e escolher entre diversas opções para a aplicação de focos de luz.




3.14 – Editor de Imagens Noturnas.

O algoritmo para escurecimento da imagem é semelhante ao que foi utilizado para correção da iluminação na textura do solo (sessão 3.3):

```
For cont1 = 0 To bmpTemp.Height - 1
      For cont2 = 0 To bmpTemp.Width - 1
            cor = bmpTemp.GetPixel(cont1, cont2)
            corR = cor.R - corDif
            corG = cor.G - corDif
            corB = cor.B - corDif
            If corR < 0 Then
                  corR = 0
            End If
            If corG < 0 Then
                  corG = 0
            End If
            If corB < 0 Then
                  corB = 0
```

```
            End If
            cor = Color.FromArgb(CType(corR, Byte), CType(corG, Byte),
      CType(corB, Byte))
            bmpTemp.SetPixel(cont1, cont2, cor)
      Next
Next
```

Os dois laços do tipo “`for`” do início varrem todos os *pixels* da imagem. Em seguida, a cor do *pixel* atual é armazenada na variável `cor` que é do tipo `Color`, uma classe já existente no Visual Basic.NET que representa uma cor, armazenando os valores das três componentes: vermelho, azul e verde. Para cada uma das componentes de todos os *pixels*, é restado um valor armazenado em `corDif`, uma variável inteira que é determinada pelo controle do menu. Estes novos valores são atribuídos a `corR`, `corB` e `corG`, as quais são utilizadas em seguida para testar se o valor ficou abaixo de zero. Caso isto tenha acontecido, o valor é reajustado para zero, que é o menor possível. Por último, a variável cor é atualizada com os novos valores e aplicada novamente ao *pixel*, que ficará mais escuro.

Com relação aos focos de luz, estes podem ser indicados pelo usuário por um único ponto ou por uma linha, sendo que neste caso, os focos de luz serão distribuídos seguindo um espaçamento uniforme, também determinado pelo usuário. Para ambos os casos é possível também optar entre criar um “par” para cada foco de luz. Ou seja, para cada ponto de luz criado, um outro será colocado paralelamente, a uma determinada distância. Isto é bastante útil para fazer a iluminação de ruas que têm postes de luz nos dois lados.

Quando a opção selecionada é de criar um único ponto, o programa simplesmente faz o processo de iluminação sobre a posição do cursor, no momento em que o botão é clicado. Já se foi optado por desenhar uma linha de pontos, é necessário primeiro calcular o tamanho total da linha:

```
dist = New Point(pMouse.X - linhaIni.X, pMouse.Y - linhaIni.Y)
angLinha = Math.Atan(dist.Y / dist.X)
tamanho = Math.Sqrt(Math.Pow(dist.X, 2) + Math.Pow(dist.Y, 2))
e.Graphics.DrawLine(pic2Pen,    linhaIni.X,    linhaIni.Y,    pMouse.X,
pMouse.Y)
```

No código acima, primeiro obtêm-se as dimensões nos eixos *x* e *y* da linha criada e armazenada na variável `dist`. Na segunda linha, calcula-se o ângulo de inclinação da linha, usando a função de arco tangente, `Atan()`, que retorna o valor em radianos do ângulo que resulta numa dada tangente. Em seguida, é calculado o tamanho da linha, aplicando-se a forma de Pitágoras (as funções `Sqrt()` e `Pow()` correspondem, respectivamente, ao cálculo de raiz quadrada e de multiplicação exponencial), utilizando os valores armazenados em `dist`. As últimas linhas do código desenham a linha na tela, utilizando parâmetros que indicam a espessura e cor da linha (no exemplo, dado por `pic2pen`) e dois pares de coordenadas que representam o início e fim da reta (`linhaIni` e `pMouse`, no exemplo).

Após o desenho da linha, devem-se distribuir os pontos para iluminação, conforme a distância escolhida pelo usuário. Para isto, haverá duas variações, dependendo se foi selecionado ou não a opção de “pares”, explicada acima. Nos dois casos, utiliza-se o seguinte laço de repetição:

```
For cont = 0 To tamanho Step espUpDown.Value
      posX = linhaIni.X + Int(Math.Cos(angLinha) * cont)
      posY = linhaIni.Y + Int(Math.Sin(angLinha) * cont)
      (…)
Next
```

Este laço percorre os valores de zero a `tamanho` - variável que foi calculada no algoritmo anterior – utilizando um passo determinado por `espUpDown.Value`, que é o valor do espaçamento, selecionado pelo usuário. Em seguida, determinam-se as coordenadas do ponto (`posX` e `posY`) sobre a linha que corresponde ao espaçamento.

Se a opção de pareamento não foi selecionada, este próprio ponto corresponderá à posição do foco de luz. A Figura 3.15 mostra o resultado da criação de uma linha de pontos únicos:

Fig. 3.15 – Desenho de uma linha de pontos únicos.

Entretanto, se o ponto corresponde a um par de luzes, as posições destas deverão estar a uma mesma distância da linha, sendo que cada ponto ficará de um lado:

```
Dim novoAng As Double = angLinha + (Math.PI / 2.0)
difX = Int(Math.Cos(novoAng) * parDist)
difY = Int(Math.Sin(novoAng) * parDist)
p1 = New Point(posX - difX, posY - difY)
p2 = New Point(posX + difX, posY + difY)
e.Graphics.DrawEllipse(pic2Pen, p1.X - 2, p1.Y - 2, 4, 4)
e.Graphics.DrawEllipse(pic2Pen, p2.X - 2, p2.Y - 2, 4, 4)
e.Graphics.DrawLine(pic2Pen, p1.X, p1.Y, p2.X, p2.Y)
```

A primeira linha calcula o ângulo da reta dada pelos dois pontos de iluminação. Está será perpendicular à linha desenhada, portanto, basta-se somar 180º (ou $\pi/2$, em radianos) ao ângulo da linha, `angLinha`, calculado no algoritmo anterior. Em seguida, é determinada a distância, tanto no eixo $x$ (`difX`) como no eixo $y$ (`difY`), que os pontos estarão da linha desenhada, utilizando-se o valor do ângulo obtido e a distância entre os pontos pareados (`parDist`), que também é escolhida pelo usuário. Esta diferença é então utilizada para posicionar os dois pontos, subtraindo e somando a diferença ao ponto sobre a linha. Tendo os dois pontos, são desenhadas elipses que representam estas posições, utilizando como parâmetros as opções de desenho (dadas por `pic2Pen`, da mesma maneira que no desenho da linha), as coordenadas do canto superior esquerdo da elipse e suas dimensões, que aqui foram fixados em 4, tanto para a altura como para a largura. Por último, desenha-se uma linha conectando os dois pontos. A Figura 3.16 mostra o resultado do desenho de uma linha com a opção de pareamento de pontos:

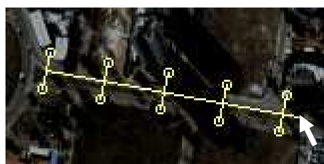


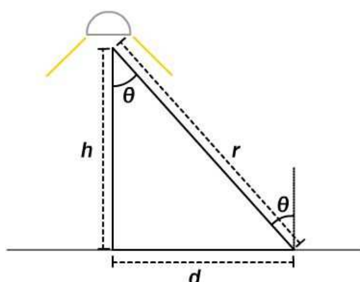
Fig. 3.16 – Desenho de uma linha de pontos pareados.

Para fazer o processo de iluminação, primeiro estipula-se uma intensidade e cor inicial para a iluminação, que são determinadas pelo que o usuário selecionou nas opções correspondentes do menu. A iluminação é aplicada aos *pixels* que estão próximos à posição da luz, sendo que a intensidade é reduzida proporcionalmente ao dobro da distância do *pixel* em questão ao ponto de onde a luz se origina:

$$I = \frac{I_0}{d^2} \times \cos\theta$$

, onde $I$ e $I_0$ são as intensidades final e inicial, $d$ é a distância e $\Theta$ é o ângulo de incidência da luz.

Entretanto, deve-se considerar também a altura dos postes de luz em relação ao solo, como mostra a figura 3.15:




3.15 – Diagrama para cálculo da distância da luz.

Neste caso, a distância entre a origem da luz e o ponto iluminado é $r$, que corresponde a:

$$r = \sqrt{d^2 + h^2}$$

O co-seno do ângulo, por sua vez, pode ser determinado por:

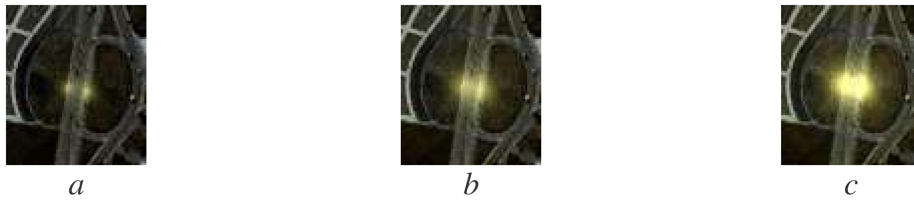
$$\cos\theta = \frac{h}{r}$$

Substituindo-se esta na equação original da intensidade, temos:

$$I = \frac{I_0}{r^2} \times \frac{h}{r} \quad \leftrightarrow \quad I = \frac{I_0 \times h}{r^3}$$

Esta fórmula será então aplicada a todos os *pixels* próximos ao foco de luz, resultando em uma iluminação com atenuação gradual da forma que seria esperado no modelo real. Ainda existem opções para modificar o tamanho e a intensidade da luz. A mudança no tamanho é feita aplicando-se uma alteração à distância de modo a criar um maior ou menor espalhamento da luz. Já a intensidade pode ser ampliada fazendo-se uma repetição do processo de iluminação um determinado número de vezes.

A Figura 3.16 mostra diferentes resultados obtidos, modificando-se os atributos de tamanho e intensidade:



*a* *b* *c*

Fig. 3.16 – Aplicação de um foco de luz com tamanho e intensidade iguais a, respectivamente, 2 e 4 (a), 4 e 4 (b) e 4 e 7 (c).

# CAPÍTULO 4

# CONCLUSÕES

Nesta fase final do desenvolvimento do trabalho conclui-se que os objetivos, que consistiam na familiarização com as funções do OpenGL, construção dos edifícios através das entradas do usuário e modificações na textura do solo foram atingidos.

O programa desenvolvido apresenta uma fácil utilização e permite que usuários sem conhecimento necessário de computação gráfica criem representações tridimensionais de cidades, utilizando apenas imagens.

Embora o trabalho tenha atingido as metas iniciais, ainda deixa lugar a possíveis aprimoramentos, como aperfeiçoamentos nos processos de iluminação, mais tratamentos na textura do solo e otimizações nos algoritmos em geral.

## REFERÊNCIAS BIBLIOGRÁFICAS


Carmenta – *SpatialAce*, 2005. (http://www.spatialace.com/)

Desenho de Objetos em OpenGL, 2006 (http://www.dcc.ufla.br/~bruno/aulas/cg/monte-mor/index.htm)

Gouraud, H. Continuous Shading of Curved Surfaces. In *IEEE Transactions on Computers.* Vol. C-20, nº 6, June 1971.

Neon-Helium Productions, *NeHe*, 2005 (http://nehe.gamedev.net/)

VTP - *The Virtual Terrain Project*, 2005 (http://www.vterrain.org/)

Wright, S. R.; Sweet, M. *OpenGL Super Bible. Second Edition*. Waite Group Press, 2000.

# APÊNDICE A

# FUNÇÕES DE OPENGL

## A.1 – Comandos para configuração da janela de visualização

Os comandos para configuração da janela de visualização permitem que se crie uma janela com dimensões previamente especificadas e o tipo de projeção a ser empregada na visualização.

- `glViewport(GLint x, GLint y, GLsizei width, GLsizei height);`

A função `glViewport()` acima indica o tamanho em *pixels* da área de visualização com o qual o OpenGL irá trabalhar. Os parâmetros *x* e *y* se referem ao canto superior da área e *width* e *height* são a largura e altura da área, respectivamente. Se for passado o tamanho da própria janela onde se está trabalhando, a área ocupará toda ela, mas pode-se dividir a tela em duas ou mais áreas diferentes, passando valores menores daqueles correspondentes à janela.

Os objetos criados em OpenGL podem ser projetados de duas maneiras: em perspectiva ou paralelamente. A projeção em perspectiva (Fig. A.1(a)) é aquela que acontece no processo de formação de imagens em nossos olhos ou numa câmera fotográfica, por isso é a que gera imagens mais realistas. Esta projeção considera a profundidade como elemento de seu cálculo e apresenta um resultado mais familiar ao observador humano.

A projeção paralela ou ortogonal é mais simples e a imagem de um ponto é definida como a projeção normal deste ponto no plano de projeção (Figura A.1(b)). A projeção paralela pode ser vista como uma projeção perspectiva onde o centro de projeção está no infinito. Conforme se observa na figura A.1(b), na projeção paralela as linhas que unem os pontos A e B às suas projeções A' e B' são paralelas, isto faz com que o segmento projetado tenha o mesmo tamanho qualquer que seja a distância entre o plano de projeção e o objeto. Este tipo de projeção é utilizado em projetos de arquitetura

onde não há preocupação com a distância aparente dos objetos; o que realmente interessa é que a escala e os ângulos dos objetos seja preservada.

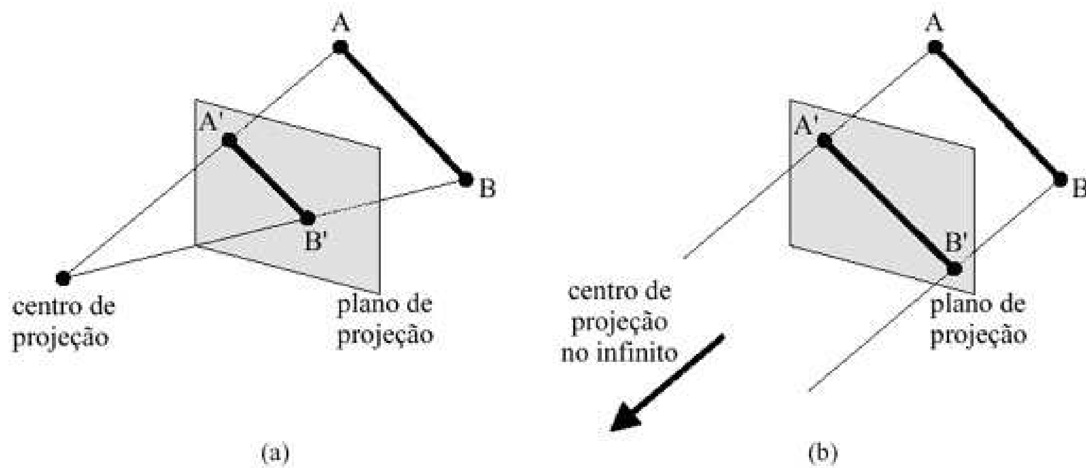



Fig. A.1 – Projeção em perspectiva (a) e paralela (b).
Fonte: (Desenho de Objetos em OpenGL, 2006)

- `gluPerspective(GLdouble angle, GLdouble aspect, GLdouble zNear, GLdouble zFar);`

Esta função especifica o tipo de projeção em perspectiva e a pirâmide truncada de recorte. O primeiro parâmetro (*angle*) indica o ângulo (em graus) de abertura na direção *y* (vertical). O segundo parâmetro (*aspect*) fornece a razão de aspecto, isto é, a relação entre a altura e a largura da janela. Os outros dois (*zNear* e *zFar*) indicam a profundidade mínima e máxima de recorte dos objetos e devem sempre ser positivos.

- `glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);`

A função acima também especifica os parâmetros de projeção, mas desta vez uma projeção do tipo paralelo, ou ortogonal. Os seis valores que a função requer delimitam a área de recorte (que neste caso é um retângulo): *left* e *right* são os limites esquerdo e direito, *bottom* e *top* são os limites superior e inferior e *near* e *far* são os limites de profundidade.

- `glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);`

Esta função especifica a cor de fundo da janela em código RGB (Red-Green-Blue). Os parâmetros *red*, *green* e *blue* indicam o quanto de cada cor compõe a cor final e o parâmetro *alpha* representa a sua transparência. Os quatro valores podem variar de 0 a 1.

- `glShadeModel(GLenum mode);`

A função glShadeModel indica o tipo de *shading*. O parâmetro *mode* pode receber dois valores:

  - `GL_SMOOTH`: suaviza a transição de cores em um polígono, através da interpolação de cores entre os vértices (método de Gouraud, 1971; Wright e Sweet, 2000);
  - `GL_FLAT`: impede a interpolação.

- `glHint(GLenum target, GLenum mode);`

Esta função especifica aspectos que dependem da implementação do OpenGL utilizado. O parâmetro *target* pode receber os seguintes valores:

  - `GL_FOG_HINT` (para cálculo de neblina);
  - `GL_LINE_SMOOTH_HINT` (suavização de linhas);
  - `GL_PERSPECTIVE_CORRECTION_HINT` (qualidade da interpolação das coordenadas de cor e textura);
  - `GL_POINT_SMOOTH_HINT` (suavização de pontos);
  - `GL_POLYGON_SMOOTH_HINT` (suavização de polígonos);

Já o parâmetro `mode` tem por função selecionar uma dada característica de `target` e pode receber os valores:

- `GL_FASTEST` (a opção mais eficiente é escolhida);
- `GL_NICEST` (a opção mais correta e com melhor qualidade é escolhida)
- `GL_DONT_CARE` (não há preferência)

## A.2 - Criação de Objetos

A criação de objetos se dá por meio do comando `glVertex3f()`, onde os argumentos transferidos correspondem à posição de cada vértice do objeto na tela. Usando-se este comando, pode-se criar pontos, linhas, triângulos e quadriláteros e, unindo-se vários destes, obtém-se uma aproximação da forma tridimensional desejada. A sintaxe para a criação de uma figura é:

- `glBegin(GLenum mode);`

Este comando informa que os próximos comandos irão fornecer os vértices de um ou mais objetos do tipo descrito pelo parâmetro `mode`, que pode receber os valores: `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_QUADS`, `GL_QUAD_STRIP`, e `GL_POLYGON`, dependendo de que figura será desenhada.

- `glVertex3f(GLfloat x, GLfloat y, GLfloat z);`

Este comando define um vértice do objeto. Se o objeto a ser criado é, por exemplo, um triângulo, este comando deve ser usado três vezes, um para cada vértice. Os três parâmetros correspondem às coordenadas *x*, *y* e *z* do vértice. Novos vértices podem seguir o primeiro triângulo, e a cada conjunto de 3 vértices um novo triângulo é formado.

- `glEnd();`

O `glEnd()` indica que todos os vértices do objeto já foram passados. Triângulos incompletos são descartados após `glEnd()`.

- `glColor3f(GLfloat red, GLfloat green, GLfloat blue);`

Com o comando `glColor3f` pode-se aplicar cores aos objetos. Os números passados como argumentos correspondem aos valores RGB da cor e podem variar de 0 a 1. Este comando deve ser acionado antes do fornecimento dos vértices e pode variar de vértice para vértice, como mostra a pirâmide da Figura A.2.

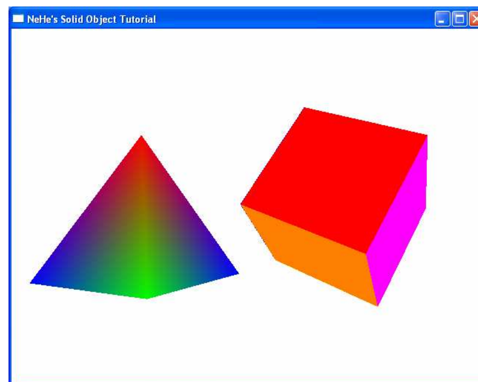



Fig. A.2 – Pirâmide e Cubo com métodos de coloração diferentes.

- `glEnable(GLenum cap);`
- `glDisable(GLenum cap);`

Estas funções habilitam e desabilitam, respectivamente, diversas opções do OpenGL, como iluminação, mapeamento de texturas, teste de profundidade, entre muitas outras. A opção que será habilitada ou desabilitada é passada pelo parâmetro *cap*.

## A.3 - Movimento e Rotação

Objetos no OpenGL podem ser movidos e orientados por meio das funções `glTranslatef()` e `glRotatef()`. As ações de movimentação devem ser efetuadas

antes da criação dos objetos que serão movidos ou orientados pela ação. Estes comandos seguem a sintaxe dada abaixo:

- `glTranslatef(GLfloat x, GLfloat y, GLfloat z);`

Esta função indica a posição no espaço em que o objeto será desenhado (nos eixos *x*, *y* e *z*, respectivamente). Ao mudarem-se estes valores de forma constante, o objeto desloca-se pela tela.

- `glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);`

A função `glRotatef` faz o objeto girar ao redor de um ou mais de seus eixos. O primeiro argumento (*`angle`*) corresponde ao ângulo de rotação, em graus. Os valores seguintes indicam o quanto o objeto irá girar em cada um dos eixos (*x*, *y* e *z*, respectivamente).

- `glPushMatrix();`
- `glPopMatrix();`

As funções `glTranslatef` e `glRotatef` atualizam o valor corrente da matriz de transformação, multiplicando-a pela matriz de translação, rotação ou variação de escala desejada. Caso se deseje preservar uma dada matriz, como por exemplo numa árvore de articulações (um personagem com braços e pernas), deve-se utilizar as funções `glPushMatrix` e `glPopMatrix` descritas a seguir, para salvar e recuperar a matriz de transformação.

A função `glPushMatrix` salva a matriz corrente numa pilha (*stack*) do tipo LIFO (*last in, first out*, ou "o último a entrar é o primeiro a sair"). Simultaneamente, ela mantém a matriz corrente com seu último valor. Novas transformações podem ser aplicadas aos objetos que serão criados usando-se a translação e rotação já descritas aplicadas à matriz corrente. O valor anterior da matriz de transformação poderá ser

então recuperado da pilha pela função `glPopMatrix`. A matriz corrente é inicializada como uma matriz identidade.

- `glLoadIdentity();`

Para substituir a matriz atual pela matriz identidade (reinicializar as transformações), usa-se a função comando `glLoadIdentity`.

## A.4 - Aplicação de Texturas

A aplicação de texturas permite melhorar o realismo do ambiente virtual pela aplicação de uma imagem à superfície de objetos. A primeira coisa a ser feita, ao se trabalhar com texturas, é criar uma ou mais variáveis para armazenar as texturas que serão utilizadas, como por exemplo `GLuint texture[1];`.

Para se aplicar uma textura a um objeto, deve-se primeiro carregar a imagem, criando a seguinte função:

- `AUX_RGBImageRec *LoadBMP(char *Filename){...}`.

Esta é uma função para fazer o armazenamento de um arquivo bitmap. A variável *`Filename`* é a que irá conter o nome do arquivo a ser carregado (observa-se que para carregar arquivos, é necessário incluir a biblioteca `stdio.h`). Dentro desta função são colocados comandos para verificar se o arquivo realmente existe e, nesse caso retornar o arquivo, ou exibir uma mensagem de erro caso o arquivo não exista.

O passo seguinte é criar a função que carregará as texturas. Dentro desta função, serão colocadas outras funções, que abrem os arquivos desejados e o armazenam como texturas, nas variáveis do vetor criado.

Dentro desta função será criado outro vetor, que armazenará as informações das imagens que serão utilizadas. No exemplo a seguir, o nome do vetor criado é `TextureImage[]` e tem apenas um elemento:

```
AUX_RGBImageRec *TextureImage[1];
```

Entre outras informações, esta estrutura irá armazenar a altura (*sizeY*), largura (*sizeX*) e os dados (*data*) da imagem no formato bitmap. Em seguida, deve-se indicar o caminho da imagem a ser utilizada como textura e verificar se este caminho existe. Isto é feito através da função `LoadBMP()` criada anteriormente:

```
if (TextureImage[0]=LoadBMP("Data/imagem.bmp")){ ... }
```

A função `LoadBMP()` retorna a referência para o arquivo indicado, e será armazenado no vetor `TextureImage[]`. Foi usada uma condição do tipo `if`, pois se houver algum erro, como o arquivo não existir, por exemplo, o programa pode exibir uma mensagem e encerrar o processamento. Se não houver erros, a imagem será carregada e armazenada como textura, por meio dos comandos seguintes:

- `glGenTextures(GLsizei n, GLuint *textures);`

Com esta função, as texturas serão geradas. A variável *n* indicará o número de texturas a serem geradas e o parâmetro *textures* deve apontar para o vetor onde elas serão armazenadas, como o vetor `texture[]` declarado no início.

- `glBindTexture(GLenum target, GLuint texture);`

Esta função associa a textura criada a uma variável. O primeiro valor (*target*) pode ser `GL_TEXTURE_1D` ou `GL_TEXTURE_2D`, dependendo da textura que está sendo utilizada. O parâmetro *texture* é a variável à qual a textura será associada. Neste caso, foi criada a variável `texture[]`, portanto seria colocado `texture[0]` como valor do parâmetro.

- `glTexImage2D(GLenum` *`target`*`, Glint` *`level`*`, GLint` *`internalformat`*`, GLsizei` *`width`*`, GLsizei` *`height`*`, Glint` *`border`*`, Glenum` *`format`*`, GLenum` *`type`*`, const GLvoid` *`*pixels`*`);`

O primeiro argumento (*target*) indica o tipo de textura bidimensional. Seus dois possíveis valores são GL_TEXTURE_2D (os dados são lidos através do argumento *pixels* e armazenados em variáveis) e GL_PROXY_TEXTURE_2D (nenhum dado é lido de *pixels*, mas todo o estado da imagem de textura é recalculado e são checados sua consistência e problemas na sua implementação). O argumento *level* se refere ao nível de detalhe. O nível 0 é o nível base da imagem. O nível *n* é enésima redução *mipmap* feita na imagem (Wright e Sweet, 2000).

O terceiro argumento (*internalformat*) indica o número de componentes de cor da textura e pode variar de 1 a 4 (vermelho, verde, azul e alfa). Também podem ser usadas certas constantes simbólicas pré-definidas, como GL_ALPHA, GL_LUMINANCE_ALPHA, GL_INTENSITY, GL_RGB, entre várias outras. Os dois argumentos seguintes (*height* e *width*) indicam os tamanhos vertical e horizontal da imagem. Como estas informações estão armazenadas em uma variável (*TextureImage*[]), pode-se usá-la para obter os valores com os comandos:

```
TextureImage[0]->sizeX,
TextureImage[0]->sizeY
```

O parâmetro *border* é referente à espessura borda e seus valores possíveis são 0 e 1. O parâmetro *format* diz ao OpenGL do que serão compostos os *pixels* da imagem. Existem nove valores simbólicos pré-definidos possíveis, mas o mais comum é o GL_RGB, com o qual cada *pixel* será composto de uma combinação das cores vermelho, verde e azul. O argumento seguinte (*type*) especifica os tipos de dados dos *pixels* da textura (GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, ou GL_FLOAT). E, por último, deve ser indicado um ponteiro para os dados da imagem, através do parâmetro *pixels*.

Novamente, isto pode ser feito através do `TextureImage[0]`, que contém estas informações:

```
TextureImage[0]->data
```

- `glTexParameteri(GLenum target, GLenum pname, GLint param);`

Esta função define os parâmetros de filtragem e repetição da textura. Para `target` há dois valores possíveis: `GL_TEXTURE_1D` ou `GL_TEXTURE_2D`, dependendo do tipo da textura que está sendo utilizada. O valor seguinte, `pname`, é o parâmetro da textura selecionada e pode receber, como valor:

- `GL_TEXTURE_MIN_FILTER`: modo de filtragem para a redução da textura;
- `GL_TEXTURE_MAX_FILTER`: modo de filtragem para a ampliação da textura;
- `GL_TEXTURE_WRAP_S`: tratamento da coordenada *s* da textura fora do intervalo 0.0 a 1.0.
- `GL_TEXTURE_WRAP_T`: tratamento da coordenada *t* da textura fora do intervalo 0.0 a 1.0.
- `GL_BORDER_COLOR`: especifica a cor da borda da imagem.

O terceiro parâmetro (`param`) especifica o tipo de filtragem para o modo escolhido em `pname` e dependerá do valor deste. Para `GL_TEXTURE_MIN_FILTER`, `param` pode ser:

- `GL_NEAREST`: método de filtragem de vizinho mais próximo. Oferece pouca filtragem, mas boa performance;
- `GL_LINEAR`: Interpolação linear. Boa filtragem, mas exige um pouco mais de processamento;
- `GL_NEAREST_MIPMAP_LINEAR`: Textura *mipmap* com interpolação linear;
- `GL_LINEAR_MIPMAP_NEAREST`: Interpolação linear com textura *mipmap*;
- `GL_LINEAR_MIPMAP_LINEAR`: Interpolação linear com textura *mipmap* interpolada;

Para `GL_TEXTURE_MAX_FILTER` os valores podem ser apenas `GL_NEAREST` ou `GL_LINEAR`. Para `GL_TEXTURE_WRAP_S` e `GL_TEXTURE_WRAP_T`, *`param`* pode receber os valores:

- `GL_REPEAT`: a textura se repetirá sobre a superfície do polígono;
- `GL_CLAMP`: usa as cores da borda quando a imagem sai do range de 0.0 a 1.0.

Finalmente, quando *`pname`* é `GL_BORDER_COLOR`, *`param`* deve ser uma matriz de cor RBGA para especificar a cor da borda.

Por fim, finalizada a função de carregar as texturas, basta aplicá-las ao objeto que está sendo criado. Deve-se observar que é necessário antes incluir o comando `glEnable(GL_TEXTURE_2D)` para permitir o mapeamento de texturas. Usa-se novamente o comando `glBindTexture(GLenum target, GLuint texture)` para especificar o tipo de textura e em qual variável ela está armazenada. Deve-se, também, relacionar cada vértice do objeto com um ponto da imagem da textura com a função:

- `glTexCoord2f(GLfloat x, GLfloat y);`

Esta função deve ser chamada antes de cada vértice do objeto (`glVertex3f()`) e as variáveis *`x`* e *`y`*, devem informar o ponto da imagem que será posicionado sobre o vértice correspondente. O sistema de coordenadas usado pelas texturas varia de 0 a 1 tanto no eixo *x* (no sentido esquerdo para o direito) como no eixo *y* (no sentido de baixo para cima). Na Figura A.3 tem-se um exemplo de uma textura aplicada às seis faces de um cubo.

Fig. A.3 – Aplicação de textura.

## A.5 - Iluminação

A iluminação em uma cena virtual realça o aspecto tridimensional dos objetos, por meio de diferenças nas intensidades das cores entre as partes mais e iluminadas e menos iluminadas.

Há dois tipos importantes de iluminação que podem ser usadas em uma cena. A primeira é a iluminação ambiente que não vem de nenhum local em particular e ilumina todos os objetos da cena igualmente. A segunda é a iluminação direta, que vem de uma fonte num ponto específico da cena. Objetos atingidos pela luz ficarão bem iluminados enquanto as partes não iluminadas ficam escuras. Para ambos os casos, deve-se primeiro criar um vetor onde serão armazenados os valores que indicam a intensidade e a cor da luz:

- `GLfloat LightAmbient[]= { GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha};`

ou

- `GLfloat LightDiffuse[]= { GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha};`

Os valores armazenados no vetor correspondem, respectivamente, à quantidade de cor vermelha, verde e azul da luz e a sua intensidade. Todos os valores podem variar de 0.0 (intensidade mínima) a 1.0 (intensidade máxima).

Como a luz direta provém de um ponto específico, deve-se indicar também a sua posição, através de outro vetor de variáveis:

- `GLfloat LightPosition[]= { GLfloat x, GLfloat y, GLfloat z, GLfloat pos};`

Se o quarto valor não for nulo, a luz é posicional e os três valores anteriores indicam sua posição nos eixos *x*, *y* e *z,* respectivamente. Se for nulo, a luz é direcional e os três valores anteriores indicam a direção em que a luz aponta.

Para gerar a iluminação, são usados os seguintes comandos (lembrando que os valores estão sendo passados pelos vetores criados anteriormente):

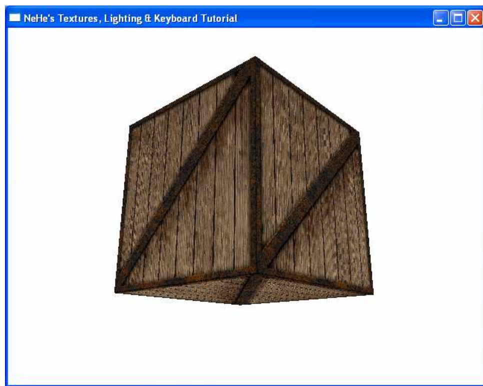
- `glLightfv(GLenum light, GLenum pname, const GLfloat *params);`

A variável *`light`* indica a luz na quais as informações que estão sendo passadas serão armazenadas como, por exemplo, `GL_LIGHT1`. O valor seguinte (*`pname`*) informa a que se referem os valores passados:

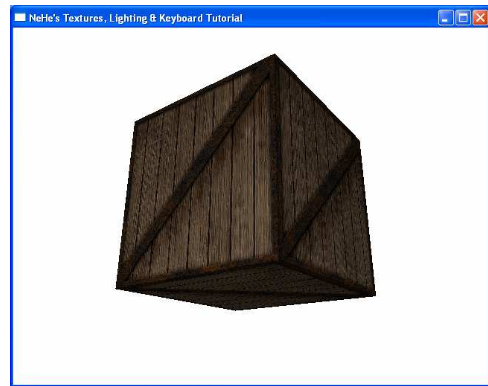
  - `GL_AMBIENT`: cor e intensidade da luz ambiente;
  - `GL_DIFFUSE`: cor e intensidade da luz difusa;
  - `GL_POSITION`: posição da luz.

O último parâmetro, *`params`* é o que recebe os valores que serão utilizados. Neste caso, os valores foram armazenados em vetores (*`LightAmbient`*`[]`, *`LightDiffuse`*`[]` e *`LightPosition`*`[]`), portanto poderiam ser utilizados para passar os valores.

Para ativar e desativar a luz direta na cena, usa-se os comandos `glEnable(GL_LIGHT1)` e `glDisable(GL_LIGHT1)`, respectivamente. Ainda, pode-se alternar entre a luz difusa e ambiente com os comandos `glEnable(GL_LIGHTING)` e `glDisable(GL_LIGHTING)`. A Figura A.4 mostra a diferença entre iluminação ambiente (*a*) e iluminação difusa (*b*).




(*a*)




(*b*)

Fig. A.4 – Iluminação ambiente (a) e difusa (b).

# APÊNDICE B

# ENTRADA E GRAVAÇÃO DE DADOS

## B.1 – Entrada de Dados

Uma vez que os pontos utilizados para a reconstrução devem ser fornecidos externamente, então é necessário contar-se com recursos que permitam ao programa receber entradas do usuário, tanto pelo teclado como pelo mouse. Para identificar as entradas do teclado, primeiro deve-se criar um vetor de 256 variáveis booleanas, que indicarão a existência de uma determinada tecla sendo pressionada:

- `bool keys[256];`

Em seguida, deve-se passar ao programa o que fazer a cada vez que uma tecla for pressionada. Isso é feito na função `LRESULT CALLBACK` que é aquela que recebe todas as mensagens da janela. Para verificar o tipo de mensagem recebida, pode-se usar uma comparação do tipo `switch`:

- `switch (uMsg){ ... }`

A classe principal da janela tem a instrução de executar a função `LRESULT CALLBACK` sempre que forem executas ações dentro dela e a variável *`uMsg`* indica que mensagem foi recebida. Alguns exemplos são: entradas de teclado (`WM_KEYDOWN`), cliques do mouse (`WM_LBUTTONDOWN` ou `WM_RBUTTONDOWN`), redimensionamento (`WM_SIZE`) e encerramento da janela (`WM_CLOSE`), etc.

Pode-se então testar o valor de *`uMsg`* de acordo com o que se quer e executar os comandos necessários:

```
case WM_KEYDOWN:
{
        keys[wParam] = TRUE;
        return 0;
}
```

No exemplo acima, quando uma tecla for pressionada, o comando "*keys*[wParam] = TRUE" será executado, sendo que *keys*[] é o vetor que havia sido criado anteriormente para armazenar o estado das teclas e *wParam* é o código numérico ASCII correspondente à tecla pressionada. Portanto, a variável do vetor *keys*[] com o código da tecla terá seu estado modificado para TRUE, indicando que está pressionada. A linha "return 0" provoca a saída do bloco switch sem executar os demais comandos.

Pode-se, então, verificar se uma determinada tecla está pressionada com uma simples comparação if:

- `if (keys['F']){...}`

Devido à letra estar entre aspas simples, é passado o valor numérico (ASCII) da letra no teclado e apontará para a variável correspondente à letra no vetor *keys*[]. Se o estado da variável for TRUE, a tecla testada está pressionada e serão executados os comandos dentro do bloco if.

Para as leituras de mouse, é necessário armazenar a posição do ponteiro, em *pixels*, quando o botão é clicado:

```
case WM_LBUTTONDOWN:
{
        x = LOWORD(lParam);
        y = HIWORD(lParam);
        return 0;
}
```

Com estes comandos, a variável *x* recebe o valor em *pixels* referente à distância do ponteiro do mouse à borda esquerda da janela (LOWORD(*lParam*)) e a variável *y* recebe o valor em *pixels* referente à distância do ponteiro do mouse à borda superior da janela (HIWORD(*lParam*)).

## B.2 – Gravação de Dados

Para possibilitar a gravação dos dados de edifícios construídos para recuperação futura, foi adicionada uma opção de gravação de dados, que armazena todos os dados necessários referentes ao edifício, como seus vértices, sua altura, posição, etc. Isto é feito através de comandos como:

- `fout.open(char *Filename);` para abrir um arquivo para escrita, ou:
- `ifstream fin(char *Filename);` também para abrir um arquivo, mas desta vez para leitura.

São ainda utilizados os comandos

- `fout << char *String`
- `fin >> char *String`

para gravar e recuperar uma linha do arquivo, respectivamente. É importante lembrar que, para que este tipo de entrada e saída de dados seja possível, é necessário incluir a biblioteca `fstream.h`.

As informações são gravadas em um arquivo de extensão "`.txt`", como mostra a Figura B.1. Cada linha representa o valor das variáveis necessárias para construir os edifícios salvos. Ao ler o arquivo, os valores serão novamente armazenados em suas respectivas variáveis:

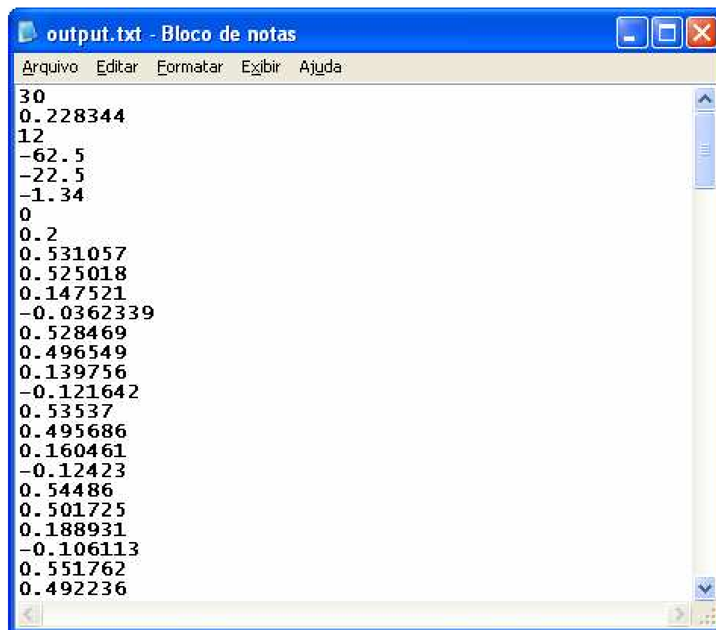



Fig. B.1 – Informações armazenadas no arquivo "`output.txt`".