Uma simulação do QuicktimeVR para mundos poligonais

Relatório Técnico VIS001

André Machado de Matos

12 de Novembro, 1996

1.0 Introdução

Neste trabalho apresentamos um programa que simula o programa QuicktimeVR da Apple. Este programa permite que o usuário tire um número de fotos de um ponto de um ambiente e, à partir dessas, cria uma representação do que é visto por um observador no ponto onde as fotos foram tiradas. Desse modo o programa mostra numa janela o que o usuário estaria vendo daquele ponto do ambiente, permitindo que ele mude sua direção de visualização. As fotos podem ser obtidas de um ambiente modelado, utilizando-se uma câmera virtual. Nesse caso, o ambiente é renderizado um número de vezes do mesmo ponto mas olhando em direções diferentes. O programa que desenvolvemos simula esse processo mas, ao invés de renderizar o ambiente um número de vezes, armazenamos uma representação das arestas do ambiente modelado e fazemos a renderização em tempo real. Na verdade o termo renderização não é muito apropriado, já que a saída do programa é em wire-frame.

Nas seçoes 2 e 3, apresentamos o Quicktime VR. Começamos por apresentar o que é o programa e como pode ser utilizado. Em seguida mostramos em linhas gerais como funciona os processos de criação da representação do ambiente e de visualização dessa representação. Na seção 4, apresentamos a fundamentação teórica para a implementação do nosso programa. Descrevemos de modo detalhado como será a nossa representação e quais são os algoritmos que utilizamos para criá-la e visualizá-la. Na seção 5 apresentamos a arquitetura da do programa, e como foi feita a sua implementação.

2.0 O que é o Quicktime VR

O Quicktime VR é um dos primeiros softwares comerciais que utiliza a tecnologia de image-based rendering para realidade virtual. Essa abordagem consiste em se obter um número de imagens de um ambiente e, à partir destas, reconstruir em tempo real o que seria visto por uma câmera virtual navegando nesse ambiente. Essa reconstrução à partir de imagens apresenta algumas vantagens em relação à tradicional reconstrução à partir de modelos geométricos. Primeiro, utilizando-se imagens, a interatividade da navegação não está limitada pela complexidade da cena, como ocorre com ambientes modelados onde há tipicamente um limite para o número de polígonos que podem aparecer numa cena. Segundo, essa abordagem permite naturalmente a inserção de imagen obtidas do mundo real nesses ambientes virtuais. Vale ressaltar que a tecnologia de image-based rendering se propõe a reconstruir visões do ambiente e não o ambiente propriamente dito.

Essa tecnologia no entanto é relativamente nova e muito ainda há que ser feito para que uma boa navigabilidade do ambiente virtual seja possível em tempo real. É provavel que a utilização dessas duas abordagens em conjunto represente um grande avanço para a área de realidade virtual.

Tendo em vista as limitações dessa tecnologia atualmente, o Quicktime VR permite apenas duas formas de interação com o mundo virtual: as *panoramas* e os *object movies*. Numa *panorama*, o usuário permanece em um ponto fixo do ambiente e pode "olhar" para qualquer direção na horizontal e até um determinado ângulo para cima e para baixo. Já nos *object movies* o usuário interage com um objeto sobre um fundo preto, podendo rotacioná-lo para visualiza-lo a diferentes ângulos.

Neste trabalho vamos tratar apenas das panoramas, sendo um dos nossos principais objetivos entender como essas panoramas funcionam no Quicktime VR.

3.0 Como funciona o Quicktime VR

O Quicktime VR é composto basicamente de dois conjuntos de programas distintos. O *programas de autoria* são utilizados para se criar uma representação do ambiente visto de um ponto fixo à partir de imagens, enquanto que o *viewer* é utilizado para reconstruir o que seria visto desse ponto à partir da representação criada pelos *programas de autoria.* Veremos a seguir, de forma simplificada, como são feitos esses dois passos.

3.1 Representação do ambiente

Para representar a visão do ambiente de um determinado ponto *P*, o Quicktime VR cria um environment map cilíndrico deste ambiente. Esse environment map poderia ser criado como explicado a seguir. Um cilindro é criado com centro no ponto *P* (chamamos de centro do cilindro o ponto no seu eixo à metade da sua altura). Cada ponto *Q* do ambiente visível de *P* é projetado em *P*, e a cor do ponto *Q* é armazenado no ponto onde o raio de projeção intercepta o cilindro. A figura 1(a) ilustra esse processo. É facil verificar que dessa maneira armazenamos toda a informação necessária para se reconstruir a visão do ambiente à partir do ponto *P*, a menos de alguns ângulo que ultrapassam a superfície do cilindro.

Para representar esse environment map, o Quicktime VR parametriza a superfície do cilindro e faz uma discretização do espaço de parâmetros, armazenando a cor de cada ponto discretizado. Uma imagem é utilizada para representar essa discretização, como vemos na figura 1 (b). Esse environment map é na realidade a própria panorama, sendo inclusive similar ao que tem sido

históricamente chamado de panorama. De agora em diante vamos nos referir ao environment map como panorama.

(a) Figura 1 (b)

(Obtida do vídeo apresentado por Leonard McMillan na Siggraph 96)

3.1.1 Captura de imagens

Na prática o Quicktime VR se utiliza de meios mais eficientes para criar essas panorama. Algumas alternativas são possíveis para a geração dessas à partir de um mundo modelado ou à partir do mundo real. A geração de panoramas à partir de um mundo modelado pode ser feita utilizando-se alguns modeladores para Macintosh, como o KPT Bryce e o Strata Pro que geram a panorama automaticamente. Outra alternativa é simular uma câmera virtual tirando fotos do ambiente modelado a diferentes ângulos e proceder como explicaremos abaixo.

Para a criação de panoramas à partir do mundo real, uma câmera panorâmica pode ser utilizada. O modo mais comum, porém, é a utilização de uma câmera fotográfica normal, sobre um tripé especial. A câmera é posicionada no ponto em torno do qual se deseja criar a panorama. O tripé especial permite que esta seja rotacionada em torno do seu ponto nodal, em intervalos de ângulos iguais, mantendo seu eixo focal numplano horizontal. Tira-se então uma foto do ambiente para cada ângulo diferente permitido pelo tripé. A figura 2 ilustra esse processo.

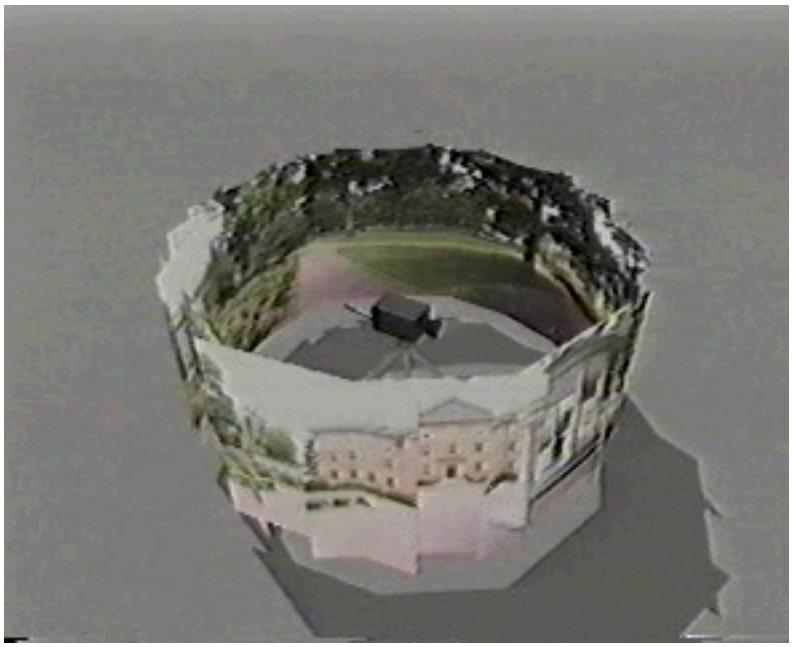


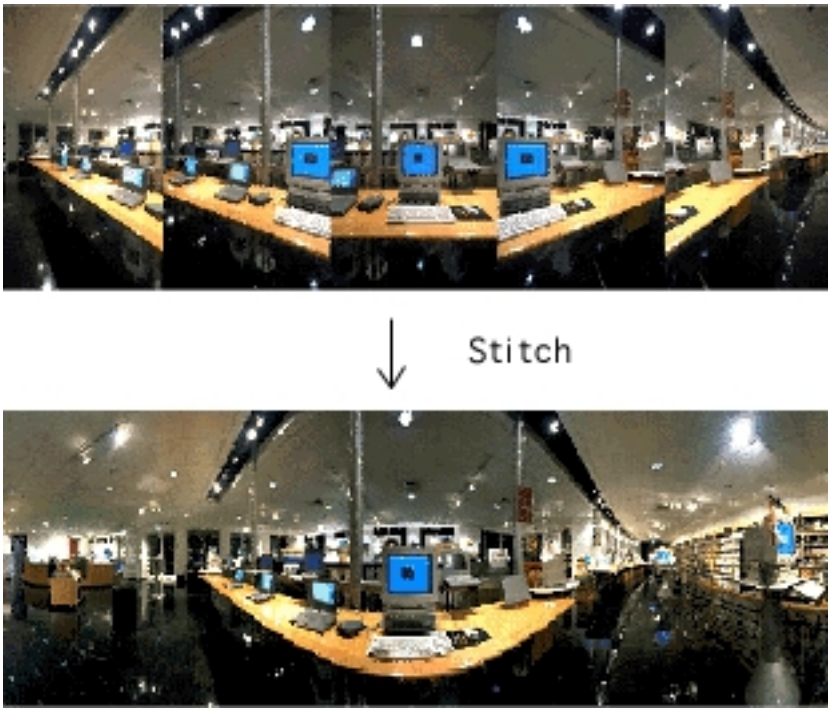
Figura 2
(Obtida do vídeo apresentado por Leonard McMillan na Siggraph 96)

As fotos obtidas são então fornecidas ao *stitcher*, um dos programas do ambiente de autoria do Quicktime VR, juntamente com vários parâmetros da câmera.

3.1.2 Formação da panorama à partir das imagens

À partir das fotos que recebe, o *stitcher* vai determinar automaticamente como juntá-las para criar a panorama. Como podemos ver pela figura 3b, a projeção cilíndrica causa distorções na imagem que representa a panorama. Assim o stitcher tem que distorcer as fotos originais de modo a juntá-las corretamente. Feitas essas distorções, ele desloca cada foto em relação às suas vizinhas na panorama, até achar a melhor correlação entre elas. O resultado final do processo pode ser visto na figura 3b.

(a)



(b)

Figura 3

(Obtida da Apple Computer)

Esse mesmo processo pode ser aplicada para fotos obtidas por uma câmera virtual em um ambiente modelado. Nesse caso os parâmetros da câmera virtual, correspondenter a uma câmera real devem ser fornecidos corretamente ao *stitcher*.

Uma vez gerada a imagem da panorama, o Quicktime VR ainda realiza algum processamento para tornar o acesso aos dados mais eficientes e para criar um arquivo no formato Quicktime da Apple. Esse arquivo pode então ser visualizado pelo viewer.

3.2 Visualização do ambiente

O viewer do Quicktime VR tem como objetivo reconstruir o que seria visto do ambiente à partir do ponto onde a panorama foi gerada. Para isso ele vai simular uma câmera virtual cujo ponto nodal está no centro do cilindro (Figura 4). Essa câmera pode ser rotacionada em torno desse ponto, permitindo ao usuário olhar em diversas direções. Sendo assim o problema que deve ser resolvido pelo viewer pode ser sintetizado como: dado uma direção de visualização qualquer,

determinar a região da panorama que vai aparecer na tela. Essa região é dada pela interseção do “cone” de visão com a superfície do cilindro.

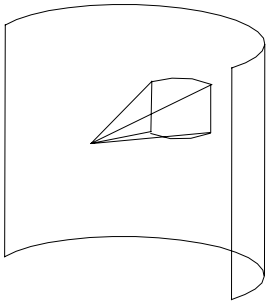


Figura 4

A projeção cilíndrica, no entanto, causa uma deformação na imagem da panorama no espaço de parâmetros do cilindro (mais adiante veremos porque essa deformação ocorre). Sendo assim, o viewer deve, além de determinar a região que deve ser mostrada, deformá-la apropriadamente, corrigindo a deformação causada pela projeção.Esse processo pode ser visto na Figura 5.
O viewer do Quicktime VR permite algumas funções adicionais, como por exemplo a ligação de duas ou mais panoramas através de hot-spots permitindo ao usuário “pular” de uma panorama para outra. Essas funcionalidades estão fora do escopo desse trabalho.

(a) (b)

Figura 5

4.0 Fundamentação teórica

O programa faz uma simulação do Quicktime VR para mundos poligonais. Ou seja, recebe um mundo modelado por polígonos e gera uma panorama de qualquer ponto deste mundo. Permite então que o usuário visualize esta panorama, simulando o que é feito pelo viewer do Quicktime VR.

Um programa deste tipo tem diversas utilidades. Primeiro, nos permite entender melhor como é feito o processo de visualização de panoramas no Quicktime VR. Segundo, é uma ferramenta de auxilio ao artista envolvido na autoria de ambientes virtuais através do Quicktime VR. Este poderia, por exemplo, fazer uma modelagem simples em 3D do ambiente em que está trabalhando e utilizar o programa para vizualizar como ocorrem as deformações, facilitando no processo de edição das imagens panorâmicas.

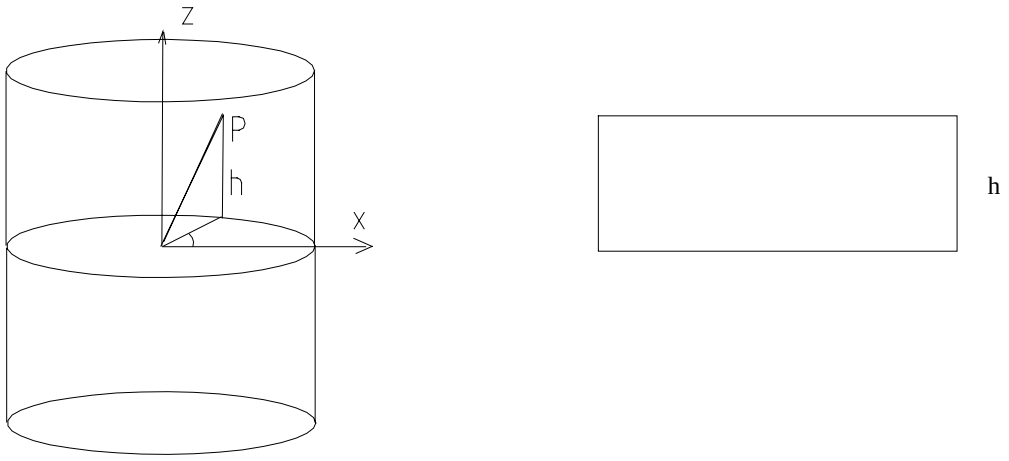
Uma pergunta natural em relação a esta proposta seria: porque se restringir a mundos poligonais ? Porque não trabalhar com imagens, como faz o Quicktime VR ? O motivo principal é as dificuldades introduzidas na implementação do warping de imagens. Se trabalhassemos com imagens, teríamos que nos preocupar com o fato de que os pixels da imagem original se contraem e se expandem por causa do warping. Sendo assim, teríamos que fazer a interpolação dos pixels onde houver expansão e point sampling onde houver compressão. Todo isso deve ser feito em tempo real pelo viewer, o que não é uma tarefa simples.

Conceitualmente podemos decompor o programa em dois componentes, o primeiro é responsável pela criação da panorama, e o segundo pela sua visualização. A seguir apresentamos cada um desses componentes separadamente. Repare que essa divisão é apenas conceitual, e, como veremos adiante, não é tão clara na implementação do programa.

4.1 Criação da panorama

Antes de começarmos a falar de como será criada a panorama é necessário introduzirmos como será parametrizado o cilindro, poiso isso é essencial para se entender o restante do processo. Cada ponto na superfície do cilindro será parametrizado pela tupla (ϕ,h), onde ϕ é o ângulo que a projeção do ponto no plano xy forma com o eixo x (assumindo que a origem do sistema esteja no centro do cilindro), e h é a coordenada z do ponto, como indica a figura abaixo. Assim o valor de ϕ varia de 0 a 2π, enquanto h varia de $-H/2$ a $H/2$, onde H é a altura do cilindro. Além disso, vamos sempre utilizar um cilindro com raio unitário de modo que $h = tg\ \beta$, onde β é a inclinação do ponto em relação ao plano *xy*, como indicado na Figura 6.

Sempre que mencionarmos o espaço de parâmetros estamos nos referindo ao cunjunto de valores (ϕ,h) de todos os pontos do cilindro (Figura 6).



(a) Figura 6 (b)

A criação da panorama será composta de três passos básicos: amostragem das arestas do polígonos, projeção das amostras no cilindro e representação dessas projeções no espaço de parâmetros deste. Note que não temos nenhum passo equivalente ao stitcher do Quicktime VR pois como já sabemos a posição dos polígonos no espaço podemos projetálos diremtamente na superfície do cilindro.

4.1.1 Amostragem das arestas dos polígonos

Enquanto o Quicktime VR faz uma amostragem do ambiente por pixels, ou seja armazenando a cor do ambiente através de cada pixel no cilindro, neste trabalho faremos uma amostragem das arestas dos polígonos. Assim, percorremos cada aresta de cada polígono do ambiente tomando amostras a intervalos regulares (o tamanho do intervalo de amostragem é algo a se determinar no trabalho). Cada amostra dessas, projetamos no cilindro armazenando a sua posição no seu espaço de parâmetros do cilindro.

Podemos dizer então que o equivalente neste trabalho à imagem armazenada pelo Quicktime VR é um conjunto de pontos em duas dimensões representando as coordenadas de cada amostra no espaço de parâmetros do cilindro, acrescentado de uma informação topológica relativa às arestas dos polígonos. Do mesmo modo que a imagem armazenada pelo Quicktime VR é distorcida pela projeção, as coordenadas das amostras sofrem a mesma distorção, e consequentemente faremos a correção dessas coordenadas no processo de visualização. A Figura 7 ilustra o processo de amostragem e projeção.

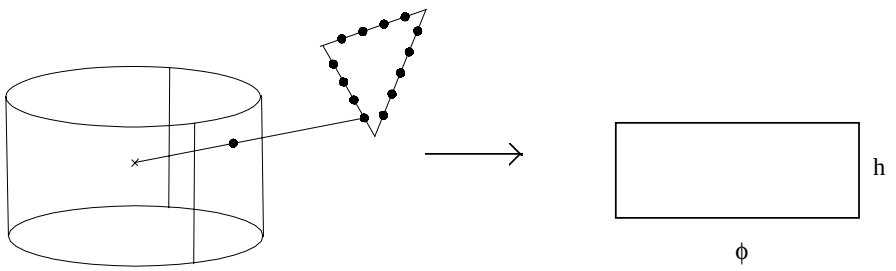


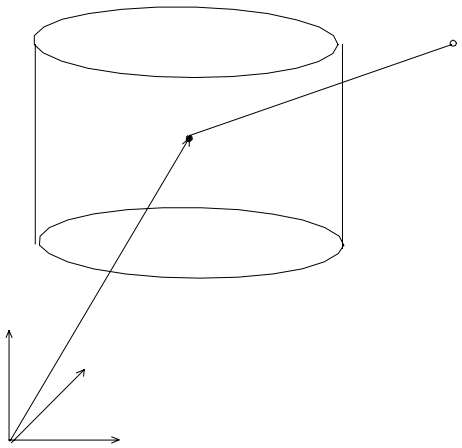
Figura 7

Uma última questão a ser respondida é: Porque tomar amostras ao longo das arestas ? Não seria suficiente tomar os vértices dos polígonos, unindo-os por uma linha reta já na tela do viewer ? De fato este procedimento resultaria numa imagem correta. Mas um dos nossos objetivos é verificar se a correção da deformação no nosso algoritmo é correta. Um meio de se fazer esse teste é verificar se as amostras ao longo das arestas formam uma linha reta na imagem final.

4.1.2 Projeção das amostras no cilindro

A projeção de cada amostra no cilindro e a sua representação no espaço de parâmetro será feito simultaneamente, utilizando-se três passos: Translação do centro do cilindro par a a origem, determinação do parâmetro ϕ do ponto projetado e determinação do parâmtro h do ponto projetado.

Transladando o centro do cilindro para a origem do sistema de coordenadas, facilitamos as contas nos próximos dois passos. Essa translação é feita simplesmente pela multiplicação do ponto por uma matrix 4x4, que desloca o espaço de um vetor (-cx,-cy,-cz), onde (cx,cy,cz) são as coordenadas do centro do cilindro (Figura 8).



$$\begin{pmatrix} x1 \\ y1 \\ z1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & -cx \\ 0 & 1 & 0 & -cy \\ 0 & 0 & 1 & -cz \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x0 \\ y0 \\ z0 \\ 1 \end{pmatrix}$$

Figura 8

Para a determinação do parâmetro ϕ_p da projeção de um ponto $P_1 = (x_1,y_1,z_1)$ no cilindro, notamos que este ângulo é o mesmo que o ângulo ϕ_1 que o ponto P_1 faz com o eixo x ao ser projetado no plano xy (Figura 9). Esse último por sua vez é dado pela equação:

$$\phi_p = \phi_1 = \text{atan}(x_1 / y_1)$$

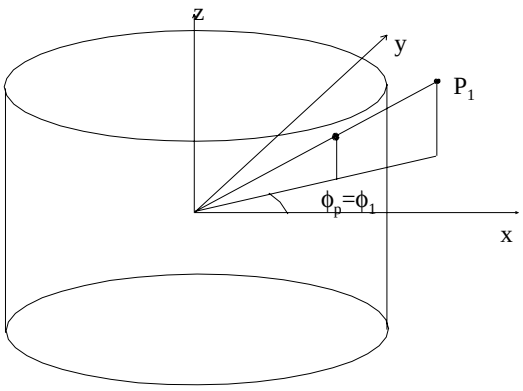
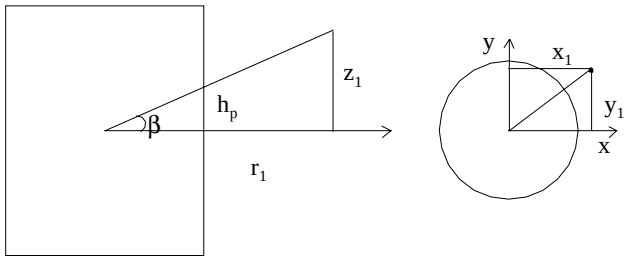


Figura 9

Para a determinação da altura h_p da projeção do ponro P_1 vamos nos referir à Figura 10. Conforme havíamos dito antes, tomamos sempre um cilindro com raio unitário (adiante veremos porque podemos fazer isso), de modo que h_p= tg β, como vemos na figura. Vemos também que tg $\beta = z_1 / r_1$, onde r_1 é a distância da projeção do ponto P_1 no plano *xy* à origem. Sendo assim, obtemos a seguinte equanção final para h_p:

$$h_p = tg(\beta) = \frac{z_1}{r_1} = \frac{z_1}{\sqrt{x_1^2 + y_1^2}}$$



vista lateral do cilindro

vista de topo do cilindro

Figura 10

Tendo sido determinados a projeção de todas as amostras no espaço de parâmetros, temos a representação final da panorama como um conjunto de pontos nesse espaço, mais uma informação topológica das arestas (Figura 11). As coordenadas dos pontos ainda serão modificadas adiante pelo viewer antes de ser visualizadas, mas as informações topológicas permanecem inalteradas.

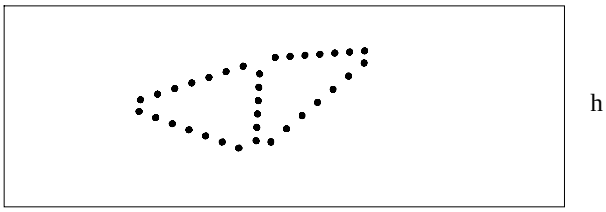


Figura 11

Um último ponto a ser resolvido é a questão do tamanho do cilindro. Já afirmamos algumas vezes que temos flexibilidade de escolher o tamanho do cilindro sem alterar os valores ϕ e h obtidos para cada ponto projetado. Isso é sem dúvida verdade para ϕ como podemos ver na Figura 12a. Mas para h, isso só é verdade se alterarmos o tamanho do cilindro de modo que o ângulo α

permaneça constante (Figura 12b). Na verdade, de qualquer modo o valor de h é alterado, porém a proporção entre o valor de h obtido para cada ponto e o tamanho do cilindro permanece constante, o que é suficiente pois ao final do processo a altura h é normalizada, como veremos mais à frente.

Notamos então que o valor de α, é importante e pode alterar os resultados. Por esse motivo, vamos exigir que o usuário do programa forneça o valor de α que deseja utiliazar para gerar a panorama. Esse valor é equivalente ao ângulo de visão vertical de uma cIamera panorâmica.

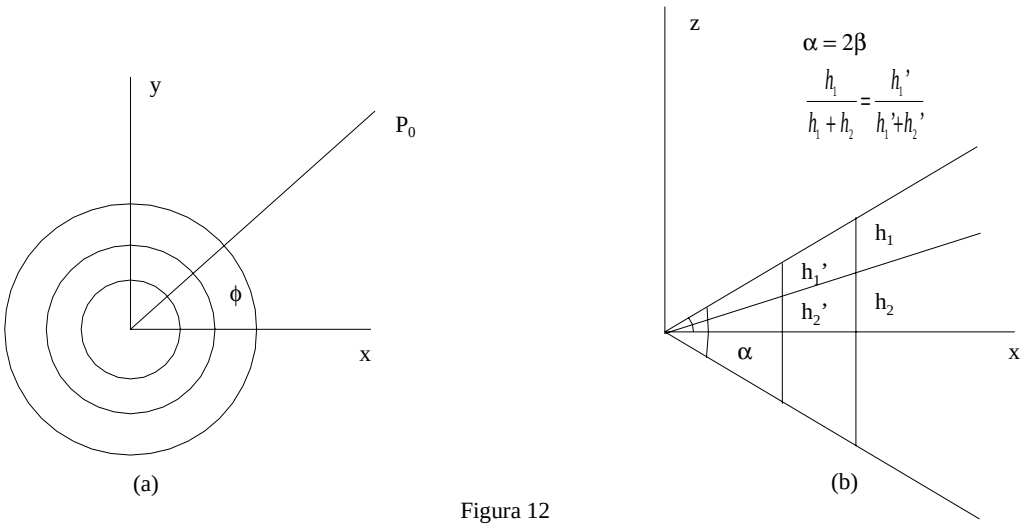


Figura 12

4.2 Implemenatação do visualizador

Como já mencionamos, o viewer pode ser visto como uma câmera virtual com ponto nodal no centro do cilindro, e que pode modificar sua direção de visualização. Essa câmera virtual vai ser completamente especificada pelo seu ângulo de visão vertical e horizontal (*hFOV* e *vFOV*). Sua direção de visualização será especificada pelos ângulos ϕ e β (Figura 13).

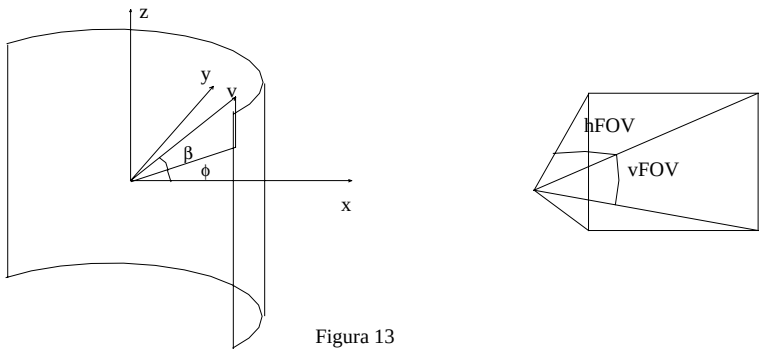


Figura 13

O problema a ser resolvido pelo visualizador é: dado hFOV, vFOV, ϕ e β , achar a região de interseção do viewing frustrum com o cilindro no espaço de parâmetros do cilindro, e mostrar essa região na tela sem deformações.

Como a projeção cilíndrica não causa deformações horizontais, se soubermos como uma linha horizontal na tela é deformada no espaço de parâmetros do cilindro, podemos descobrir como fazer a deformação no sentido inverso. Uma linha horizontal na tela define um plano no cone de visão que faz um ângulo β_0 com o plano horizontal e inclui uma linha perpendicular à direção de visualização (Figura 14). A interseção deste plano com o cilindro é a representação desta linha no espaço de parâmetros.

Depois de descoberta a linha no espaço de parâmetros, temos que determinar uma equação que faça a deformação inversa, ou seja, mapeia essa linha na linha horizontal da tela.

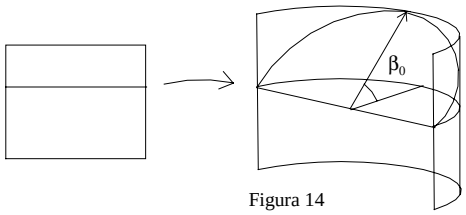


Figura 14

4.2.1 Determinação da deformação

Como indicamos acima, o primeiro passo é determinar a curva no espaço de parâmetros para onde um linha horizontal na tela seria mapeada. Já vimos que para isso temos que achar a interseção do plano inclinado corrrespondente à essa linha com o cilindro. Seja ϕv o ângulo ϕ da direção de visualização (Figura 15). Tomemos um ponto qualquer sobre o plano com um ângulo $\phi 2$ diferente de ϕv. Podemos ver pela figura que $tg\ \beta_0 = h_2 / \cos(\phi_2 - \phi_v)$, onde h_2 é a altura do novo ponto. Daí podemos tirar a seguinte equação que define a curva que queremos:

$$h = tg\ (\beta_0)\ \cos(\phi - \phi_v)$$

Note que essa depende não apena da inclinação β_0 do plano, mas também do ângulo ϕ da direção de viasualização (ϕ_v).

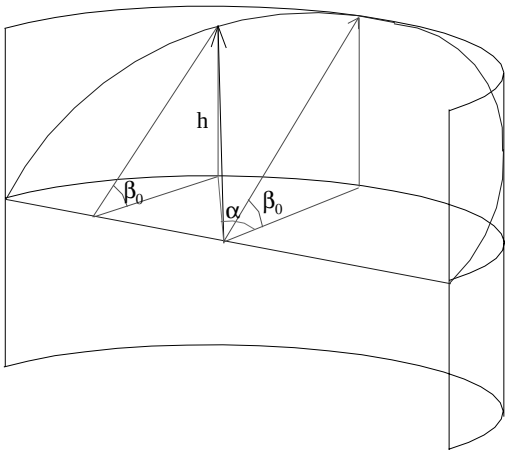


Figura 15

A Figura 16 apresenta diversas curvas que representam a equação obtida acima para h, com valores de β_0 diferentes, ou seja são curvas de interseção de planos com inclinações variadas, correspondentes a linha horizontais diferentes na tela.

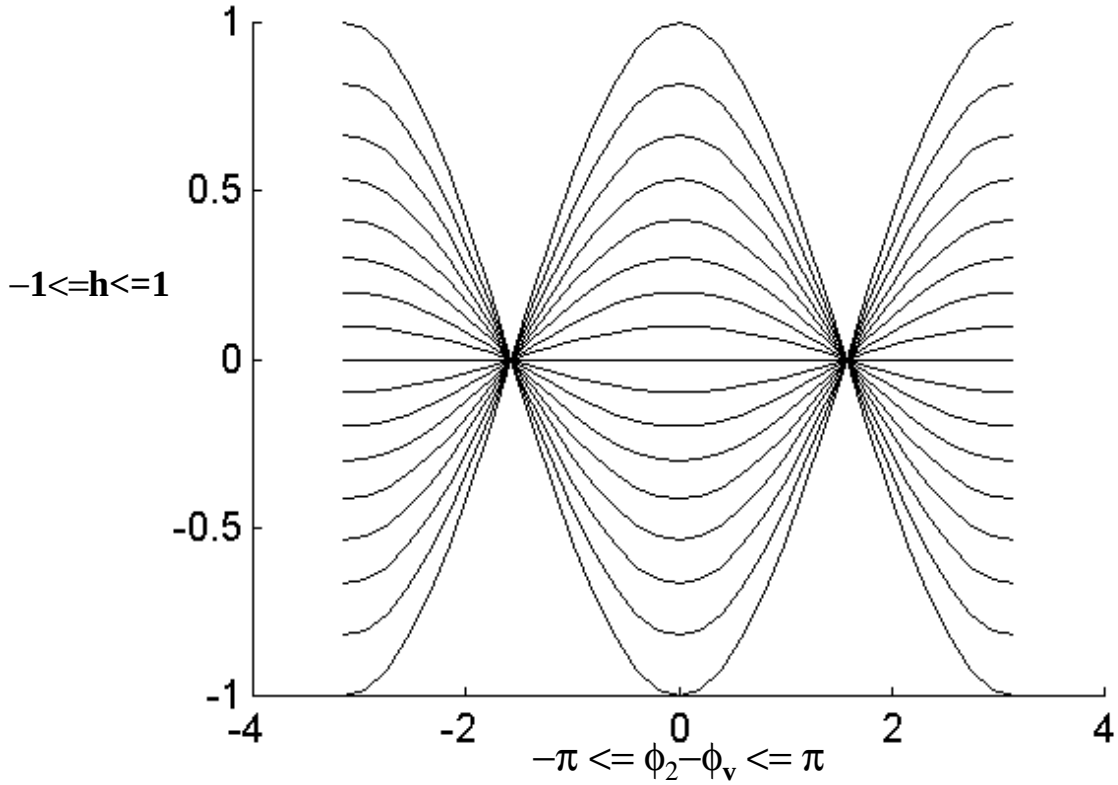


Figura 16

Sabemos que para corrigir a deformação causada pela projeção temos que levar cada curva dessas numa curva horizontal, pois sabemos que elas são obtidas à partir de uma linha horizontal na tela. Isso é fácil se queremos que a altura da linha horizontal seja tg β, onde β é a inclinação do plano correspondente à linha: Seja um ponto (f_1,h_1), queremos que sua nova altura seja h_w = tg b_0 . Sabemos que:

$$h_1 = tg(\beta_0) \cdot \cos(\phi_1 - \phi_v)$$

Logo obtemos o valor de h_w a altura corrigida:

$$h_w = \frac{h_1}{\cos(\phi_1 - \phi_v)}$$

4.2.2 Algoritmo para o visualizador

Agora que já sabemos como fazer o warping, aplicamos o seguinte algoritmo para implementar o viewer dados hFOV, vFOV, ϕ e β:

- Determinar a região de interseção do cone de visão com o cilindro.
- Achar todos as amostras interiores ao bounding box dessa região.
- Aplicar o warping a cada amostra, normalizando as coordenadas.

A seguir explicamos cada passo desses em detalhes.

A região de interseção do cone de visão com o cilindro é completamente especificada pelos 6 pontos indicados na Figura 17. Isso pode ser mostrado já que as duas curvas acima e abaixo da região são quadráticas no cilindro, e logo são definidas por 3 pontos. Esses pontos podem facilmente ser obtidos pelos parâmetros fornecidos, como indicado na figura.

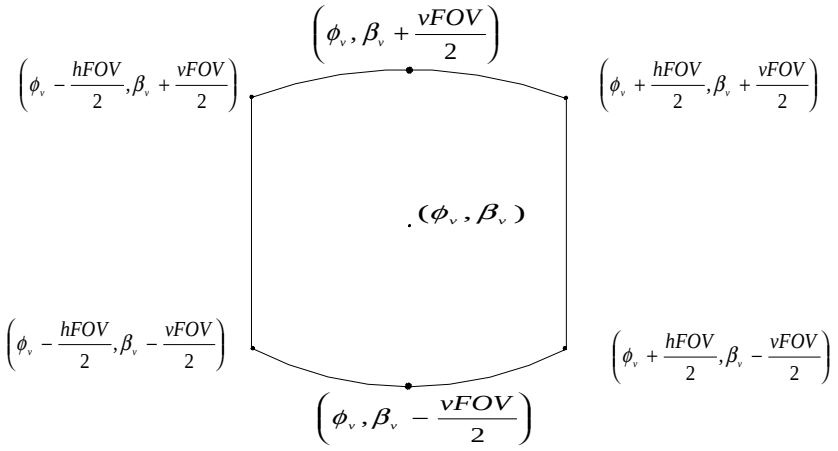


Figura 17

Tendo achado esses 6 pontos, obtemos o bounding box deles, ou seja determinamos quem são os valores mínimos e máximos de ϕ e h (ϕ_{min}, ϕ_{max}, h_{min}, h_{max}). Achamos então todas as amostras que estejam dentro desse bounding box, ou seja, todas as amostras P = (ϕ, h) que satisfazem:

$\phi_{min} <= \phi <= \phi_{max}$
$h_{min} <= h <= h_{max}$

Vamos chamar os dois pontos de maior e menor altura com ângulo ϕ igual a ϕ_v de $P_a = (\phi_v , h_a)$ e $P_b = (\phi_v , h_b)$ (Figura 18). Os pontos com esse valor de ϕ não tem sua altura modificada. Assim h_a e h_b serão a maior e menor altura entre todos os pontos da região válida. Utilizando esses valores para normalizar a altura dos pontos no intervalo [0,1], os pontos fora da região de interesse, mas dentro da bounding box serão jogados para fora desse intervalo.

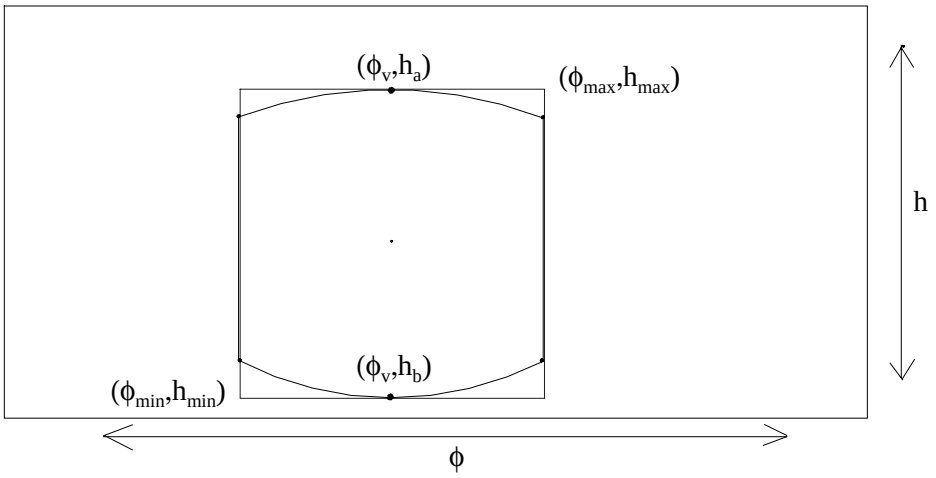


Figura 18

Finalmente, fazemos a correção dos pontos conforme calculamos obtendo a tupla (ϕ_W, h_W).

$$h_w = \frac{h_p}{\cos\left(\phi_p - \phi_v\right)}, \phi_w = \phi_p$$

Normalizamos então os valores obtidos resultando na tupla (x_s,y_s) que representam a posição do ponto num viewport com limites [0,1][0,1]. Pontos que resultam em alguma dessas coordenadas fora desse intervalo devem ser descartados.

$$y_s = \frac{h_a - h_w}{h_a - h_b}$$

$$x_s = \frac{\phi_p - \phi_{\min}}{\phi_{\max} - \phi_{\min}}$$

Ressaltamos que a coordenada y é orientada de cima para baixo da tela.

O conjunto de todos as tuplas (x_s,y_s) obtidas à partir das amostras deve então ser desenhado na tela levando-se em conta as informações topológicas armazenadas. Se tudo estiver correto, deve aparecer na tela o ambiente poligonal original visto do ponto especificado e a direções de visualização diferentes.

5.0 Implementação

Tendo visto os algoritmos que devemos utilizar, veremos agora como esses são efetivamente implementados no programa. Este foi feito em C++ para Windows, utilizando a biblioteca de classes MFC da Microsoft.

O código do programa também pode ser considerado como tendo dois grupos de componentes. O primeiro é a parte de manipulação das panoramas, ou seja, essencialmente a implementação dos algoritmos apresentados acima. O segundo grupo é a parte de interface com o usuário, ou seja, como se obtem as entradas do usuario e como é mostrado o resultado. Apresentaremos em mais detalhes os componentes do primeiro grupo, em seguida falamos um pouco de como é feita a interface com o usuário.

5.1 Implementação das panoramas

Os algoritmos anteriormente apresentados são implementados em um número de classes que representam os objetos envolvidos no processo de criação e visualização de panoramas. Ao contrário do modo como foram apresentados os algoritmos, as classes não refletem um divisão numa parte de visualização e outra de criação. Sendo assim a interface oferecida pelos objetos tem alguns métodos utilizados para a visualização, outros para a criação e outros para ambos.

Os objetos que identificamos que estão envolvidos nesses processos são: panorama, imagem de panoramas, superfície de panoramas e câmera virtual. Foi criado então uma classe para cada um desses objetos. A seguir mostramos a interface oferecida por cada uma dessas classes e comentamos a sua funcionalidade.

5.1.1 Panorama

Como já vimos, uma panorama é composta de uma superfície onde uma imagem é mapeada e a imagem propriamente dita. Desse modo a classe panorama serve basicamente para englobar dois objetos representando uma imagem e uma superficie. A listagem 1 mostra a interface oferecida por essa classe.

```
//        Panorama.h
//
//        This file defines a class for representing a virtual panorama.
//        The panorama can be is represented by its panorama surface,
//        and the panoramic image. These are defined in addtional classes.

#include "PanoramaImage.h"
#include "PanoramaSurface.h"
#include "Geometry.h"
```

```
class Panorama    {

public:

        //
        // Constructors
        //

        Panorama() {}
        Panorama( const PanoramaImage& _image,
                        const PanoramaSurface& _surface );

        // Initialization

        void                    InitPanorama( Environment3d&      _environment,
                                                const Point3d& center );
        //
        // Data access
        //

        PanoramaImage&          Image() { return image; }
        PanoramaSurface&        Surface() { return surface; }
private:
        PanoramaImage                   image;
        PanoramaSurface                 surface;

};
```

Listagem 1.

O principal método desta classe é o método InitPanorama que faz essencialmente toda a criação da panorama à partir de uma descrição de uma ambiente 3D e da posição do observador neste ambiente. Esse ambiente, um objeto da classe Environment3d, é na realidade uma lista de arestas no espaço 3D. Essa classe, bem como um número de outras classes relacionadas com geometria, estao definidas no arquivo geometry.cpp, mas não entraremos em detalhes sobre a sua implementação. O método percorre a lista de arestas do ambiente 3D, quebra as arestas em um número de segmentos, projeta cada segmento na superfície da panorama, e armazena a posição desse segmento na imagem da panorama. Essa posição, como já vimos, é a posição da aresta no epsaço de parâmetros da superfície.

5.1.2 Imagem da Panorama

A imagem da panorama é representada na classe PanoramaImage, cuja definição está na listagem 2.

```
//      PanoramaImage.h
//
//      This file defines the image being mapped in the panorama
```

```
#include "geometry.h"


class PanoramaImage        {

public:

        // constructors

        PanoramaImage() {}

        // This function draws a bitmap
        // in a device context passed as parameter
        // and with the size specified that displays the panoramic
        // image.

        void       DrawBitmap( CDC *pDC, int Width, int Height );

        Environment2d& Samples() { return uvsamples; }

private:

        Environment2d    uvsamples;
};
```

Listagem 2.

Como podemos ver, essa classe tem uma única variável membro do tipo Environment2d. Essa classe representa uma lista de arestas no plano, o que será a representação da imagem que vamos utilizar, como já vimos anteriomente. O método InitPanorama da classe Panorama será responsável por preencher essa estrutura como já vimos. O único método dessa classe é o DrawBitmap que desenha, utilizando funções do Windows, as arestas em um bitmap que pode ser desenhado na tela. Note que com isso, essa classe não é portável para outra plataformas, e logo uma posição para esse método deve ser estudada no futuro.

5.1.3 Superfície da panorama

A superfície da panorama está definida na classe PanoramaImage, cuja inteface está na listagem 3.

```
// This file defines the class that represents a panorama
// surface. For now this is a cylinder surface
// in the future ot should become a virtual base class
// for a number of different types of specifica surfaces
//
// The virtical angle is always specified in radians
```

```
#include "geometry.h"
#include "Transform.h"

class PanoramaSurface {

public:

        // Constructor

        PanoramaSurface(){}

        PanoramaSurface( const Point3d& _center, const float vangle );

        // Data Access

        void    SetCenter( const Point3d&_center );
        void    SetVAngle( const float      vangle );

        Point3d         Center() { return center; }
        float           VerticalAngle() { return vertical_angle; }
        float           Radius()        { return radius; }
        float           HalfHeight()            { return half_height; }

        Point2d         Project( const Point3d& pt );

private:

        // Calculate the height given the angle

        void CalcDimensions();

        Point3d         center;
        float           vertical_angle;
        float           radius;
        float           half_height;

        Matrix          Xform;

};
```

Listagem 3.

Embora o nome não indica, essa classe representa um supefície cilínrica de uma panorama, que a única que utilizamos nesse programa. Vemos que as variáveis armazenada nessa classe são essencialmente aquelas que mencionamos quando descrevemos o algoritmo. Uma das variáveis é uma matriz que armazena a translação que deve ser feita na criação da panorama, levando o centro do cilindro na origem. O único método desta classe que não serve para acesso a variáveis é o método Project, que recebe um ponto do espaço e retorna as coordenadas no espaço de parâmetros da projeção deste ponto na superfiície. Esse método é utilizado pela função InitPanorama para criar a imagem da panorama.

5.1.4 Camera Virtual

A câmera virtual é representada pela classe Câmera cuja definição está apresentada na listagem 4.

```
// Camera.h
//
// This file defines a class for representing a virtual camera
// The camera is specified by its horizontal and vertical
// viewing angles and by its vizualization direction

#include "Panorama.h"
#include "PanoramaImage.h"

class Camera {


public:

        // Constructors

        Camera();

        Camera( float _hFOV, float _vFOV );

        // Data access

        float           HFOV() { return hFOV; }
        float           VFOV() { return vFOV; }
        float           Pan()  { return pan;  }
        float           Tilt() { return tilt; }

        // Data modification

        void            SetHFOV( float _hFOV ) { hFOV = _hFOV; }
        void            SetVFOV( float _vFOV ) { vFOV = _vFOV; }
        void            SetPan( float _pan ){ pan = _pan;}
        void            SetTilt( float _tilt ) { tilt=_tilt; }
        void    IncPan( float _delta ) { pan+=_delta; }
        void    IncTilt( float _delta ) { tilt+=_delta; }
        void            IncHFOV( float _delta ) { hFOV += _delta; }
        void            IncVFOV( float _delta ) { vFOV += _delta; }

        // Reprejects the panorama image onto the camera, writing
        // the result to a DIB passed in as parameter

        void            Reproject( Panorama& panorama, CDC *pMemDC );


private:

        float hFOV, vFOV, pan, tilt;
```

```
};
```

Listagem 4.

Vemos que as variáveis dessa classe são os parâmetros que definem o estado de uma câmera virtual, como apresentamos na seção 4. Além dos métodos de manipulação dessas variáveis, essa classe contém o método Reproject que basicamente faz toda a funcionalidade do visualizador. O método recebe a panorama como parâmetro e retorna uma superfície de desenho com o desenho do que está sendo mostrado pela câmera na panorama. Esse método faz todos os testes necessários para verificar se os parâmetros da câmera podem ser usados pela panorama, caso contrário, modifica esses valores. Os demais métodos são chamados pelas rotinas de interface para modificar os parâmetros da câmera.

5.2 Interface com o usuário

Tendo apresentado como é implementado procedimento das panoramas, veremos agora como isso é suportado a nível de interface com o usuário.

O programa recebe como entrada um conjunto de arestas 2D representando paredes e uma altura para essas. Essa informação é utilizada para criar as paredes em 3D, como uma lista de arestas. O programa recebe também uma lista de arestas 3D, que podem ser obtidas de qualquer objeto poligonal, qua são adicionadas à lista de paredes, criando assim a representação do ambiente.

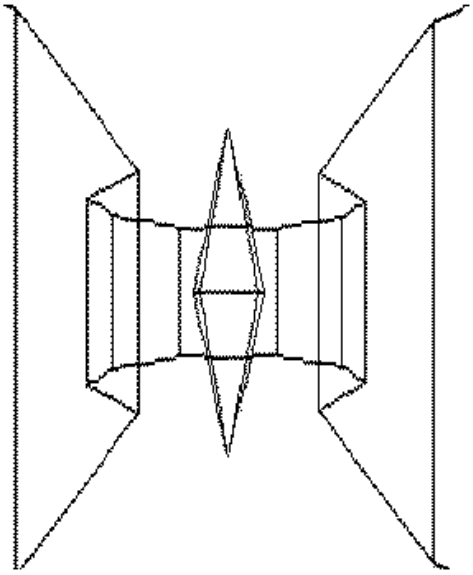
A janela do programa é dividida em três partes. A primeira parte mostra uma vista superior do ambiente, em 2D. Clicando em pontos dessa janela, o usuário pode posicionar o observador. A segunda janela mostra a imagem panorâmica gerada da posição atual do observador. Finalmente a terceira janela implementa o visualizador, semelhante ao do QuicktimeVR. Sempre que o observador muda de posição, o que é mostrado nas duas últimas janelas é modificado.

O programa foi feito utilizando-se a biblioteca de classes MFC da Microsoft. Esse ambiente tem uma arquitetura baseada em Documents e Views. Os documents são uma representação dos dados do seu programa. Enquanto que os views são objetos associados aos Documents e a uma janela da aplicaçao. A view define como os dados do document são mostrados na janela e como as entradas do usuário afetam esses dados. No caso deste programa, foi criado um objeto Document com uma panorama e uma câmera. Cada uma das três janelas representam então um View diferente que utiliza operações diferentes para mostrar os dados. No caso, uma chama as funções do viewer, outra mostra a imagem panorâmica e a outra desenha o ambiente. As entradas do usuário também são diferentes para cada janela. Clicando-se o mouse na primeira

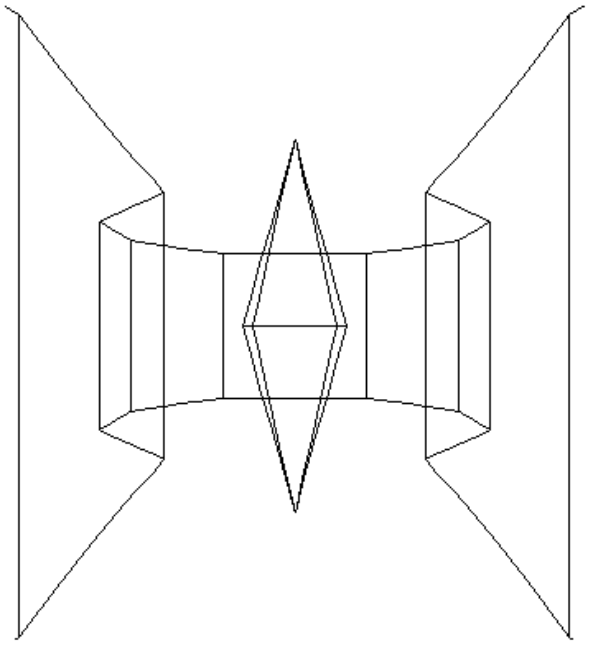
modifica-se a posição do observador. Na terceira, modifica-se a direção de visualização.

5.0 Conclusão

Tendo feito um programa para a simulação do Quicktime VR, é importante compararmos a saída do programa com aquela do Quicktime VR. Para isso, geramos um panorama de um determinado ponto do ambiente e exportamos a imagem panorâmica como um arquivo de bitmap. Esse arquivo fornecemos como entrada para o programa de autoria do Qucktime VR como sendo uma imagem panorâmica. A figura 20 mostra a saída mostrada pelos viewers do nosso programa e do Quicktime VR.



(a)



(b)

Figura 20.
(a) Saída do QTVR. (b) Saída do simulador.

Como podemos ver o resultado obtido por ambos os programas são muito parecedidos. Uma dificuldade que surge para que a saída seja exatamente igual, é a falta de informação quanto ao ângulo de zoom que o Quicktime VR usa.

Além de permitir um maior entendimento do processo realizado pelo Quicktime VR, o programa pode auxiliar um artista no desenvolvimento de um ambiente interno com um número de pontos onde Quicktime VR's serão gerados. Uma planta do ambiente poderia ser entrado no programa e o usuário poderia ver como seriam geradas panoramas de diferentes pontos do ambiente. Com isso pode decidir os melhores pontos de onde tirar as fotos. Tendo a imagem panorâmica com as paredes, o programa também vai ajudar no processo de desenhar figuras adicionais diretamente numa imagem panorâmica, servindo de guia para as deformações causadas.