UNIVATES

# ESTUDO SOBRE FERRAMENTAS DE REALIDADE AUMENTADA EM AMBIENTES DE BAIXO CUSTO

NASAIR JÚNIOR DA SILVA

Lajeado, dezembro de 2007

CENTRO UNIVERSITÁRIO UNIVATES

CURSO DE ENGENHARIA DA COMPUTAÇÃO



# ESTUDO SOBRE FERRAMENTAS DE REALIDADE AUMENTADA EM AMBIENTES DE BAIXO CUSTO

Trabalho de conclusão de curso apresentado ao Centro Universitário UNIVATES para a obtenção de título de Bacharel em Engenharia da Computação.

Orientador: Marcelo de Gomensoro Malheiros

Lajeado, 2007

*Dedico este trabalho a todos que me acompanharam durante essa caminhada, mas principalmente a meus pais, a quem devo não só meu passado e minha educação, mas também meu futuro, e à minha namorada por todo o amor, apoio e paciência.*

# AGRADECIMENTOS

*Agradeço a Deus pela minha existência e toda a energia para chegar até aqui; agradeço a meus pais, à minha namorada, aos meus irmãos, meus cunhados, meus sobrinhos e meus afilhados por todo o apoio e amor necessário para chegar até aqui; agradeço a meus amigos e colegas de trabalho pela ajuda e compreensão necessária; agradeço aos mestres por todos seus ensinamentos e companheirismo; agradeço ao meu orientador por todo o apoio e incentivo para conseguir elaborar este trabalho.*

*Quando você quer alguma coisa, todo o universo conspira para que você realize o seu desejo.*

***Paulo Coelho***

# RESUMO

A Realidade Aumentada vem cada vez mais tomando espaço como uma forma alternativa de interação com o usuário, principalmente por causa do grande poder de processamento das máquinas atuais. Este trabalho visa estudar as principais ferramentas disponíveis em Software Livre para criação de aplicações usando Realidade Aumentada, analisando suas características, aplicabilidade, funcionalidades, comparativos e dificuldades na utilização.

Palavras-chave: Realidade Aumentada, Computação Gráfica, detecção de marcadores fiduciais.

# ABSTRACT

Augmented reality is becoming an interesting alternative to traditional user interface systems, being now of practical use because of the increased processing power of current machines. This work aims to study several tools, available as free software, to develop augmented reality applications, analyzing their characteristics, applicability, functionalities and difficulty of use.

Keywords: Augmented Reality, Computer Graphics, fiducial marker detection.

# SUMÁRIO

# LISTA DE FIGURAS

# LISTA DE TABELAS

# 1 INTRODUÇÃO

A cada dia que passa, há novos equipamentos que nos impressionam com seu poder de processamento e tecnologia. Essas novas tecnologias exigem maneiras de interação para melhor aproveitá-las. Uma destas é a Realidade Aumentada (RA, por brevidade), ramo da Realidade Virtual (RV, por brevidade) que permite misturar um ambiente real com um ambiente virtual. A RV permite ao usuário visualizar e interagir com situações imaginárias, como os cenários de ficção, envolvendo objetos e cenários criados a partir de aplicações executadas pelo computador. Estas aplicações também evoluem, permitindo a criação de ambientes virtuais tridimensionais ricos em detalhes, próximos da perfeição, estimulando cada vez mais os sentidos humanos (tais como visão e audição). Os ambientes virtuais podem ser modificados à medida que o usuário interage com a aplicação. A RV induz o participante a um sentimento de imersão em um mundo irreal através de técnicas computacionais nas quais ele pode agir e interagir. É uma tecnologia que abrange uma gama enorme de idéias e ações, buscando levar o usuário a se sentir completamente imerso em um ambiente.

Já a RA procura levar a agilidade digital a realçar aspectos do mundo real, permitindo que o usuário possa interagir, em um ambiente real, com objetos ou situações virtuais. Para tanto, RA faz uso da combinação de RV e Mundo Real, propiciando uma melhor percepção do usuário e sua interação. São características básicas de sistemas de RA:

- Processamento em tempo real;
- Combinação de elementos virtuais com o ambiente real;
- Uso de elementos virtuais concebidos em 3D.

A RA vem se fortalecendo como uma nova forma de interação entre usuário e computador, sendo utilizada em várias áreas de conhecimento, que vão desde a robótica até a medicina. Vem se mostrando cada vez mais importante, e motivou uma análise mais detalhada de ferramentas de baixo custo de visão computacional, principalmente voltada à extração de marcadores, os quais serão definidos no Capítulo 2.

Inicialmente, ainda no Capítulo 2, iremos descrever alguns conceitos, como imagem, forma de digitalização, processamento e alguns outros conceitos necessários para o entendimento do trabalho. No Capítulo 3, iremos apresentar ferramentas para desenvolvimento de aplicativos em RA e maiores detalhes sobre o ARToolKit, ferramenta escolhida para análise neste trabalho. No Capítulo 4, iremos apresentar detalhes da ferramenta, como algoritmo utilizado para a identificação de marcadores, obtenção de características da câmera e outros, assim como serão apresentadas outras bibliotecas utilizadas: OSG e osgART: a primeira é utilizada para renderização de objetos 3D, enquanto a segunda provê a integração entre o OSG e o ARToolKit.

No Capítulo 5, será descrito o protótipo de aplicativo desenvolvido utilizando interação com RA: um jogo de “pong”. Neste mesmo capítulo, serão demonstrados os principais códigos do protótipo, assim como testes efetuados.

Finalizando, serão apresentadas as conclusões deste trabalho, contribuições do mesmo e trabalhos futuros a serem realizados.

# 2 EMBASAMENTO TEÓRICO

Nesta seção são definidos conceitos importantes para a compreensão deste trabalho, juntamente com a descrição de trabalhos prévios relacionados.

## 2.1 Imagens

### 2.1.1 Definição

Segundo (Azevedo, 2003), a luz, tradicionalmente referida como a porção visível do espectro eletromagnético, ocupa uma faixa muito pequena deste, podendo o olho humano detectar somente os comprimentos de onda que se situam entre 380 a 750 nm.

Segundo (Aumont, 1993), as informações que nos chegam através da luz são processadas e em sua base nos remetem a três características: intensidade (responsável pela percepção da luminosidade, que provém das reações do sistema visual à luminância dos objetos), comprimento de onda (percepção da cor) e distribuição no espaço (noção de borda visual, designando o limite entre duas superfícies de diferentes luminâncias).

Quando se observa um objeto opaco, na verdade vê-se a luz que ele reflete, e a cor percebida é uma conseqüência do material do qual ele é composto e da luz que ele reflete. Em dispositivos digitais, a luz branca é representada por uma série de cores distintas e, de fato, pode-se produzir cores a partir da divisão da luz branca utilizando um prisma. Os objetos refletem parte das cores que incidem sobre eles e

absorvem o restante. Os raios refletidos pelo objeto vão formar o que chamamos de cor (Fogagnoli, 2000).

Quando o olho humano é atingido por um raio de luz (difusa ou refletida), instantaneamente o absorve através de células pigmentadas. Parte desses raios são transformados em calor e parte em impulso elétrico. É a interpretação dessa onda pelo cérebro humano que dá a sensação de cor (Sobel, 1987).

A maioria das sensações de cor pode ser produzida através da composição de proporções adequadas das cores vermelho, verde e azul. Quando estes comprimentos de ondas se sobrepõem, eles se adicionam e combinam seus efeitos. Aos pares, essas três cores quando combinadas produzem, no computador: o ciano, resultado do verde e azul, o magenta, resultado do azul e vermelho e o amarelo, resultado do vermelho e verde. Quando as três cores primárias são combinadas, formam novamente a luz branca. São portanto chamadas de cores aditivas (Fogagnoli, 2000).

Uma imagem (natural) é uma variação contínua de cores e tons. No caso de uma fotografia, por exemplo, os tons variam de claros a escuros e as cores variam segundo todo o espectro de cores visíveis.

As imagens digitais são exibidas em níveis discretos de brilho. Segundo (Gonzales, 2000), o brilho subjetivo (percebido pelo olho humano) é uma função logarítmica da intensidade de luz incidente no olho.

O termo imagem refere-se a uma função de intensidade luminosa bidimensional, denotada por *f(x,y)*, em que o valor ou amplitude de *f* nas coordenadas planares *(x,y)* dá a intensidade (brilho) da imagem naquele ponto. Como a luz é uma forma de energia, *f(x,y)* deve ser positiva e finita (Azevedo, 2003).

As imagens que as pessoas percebem em atividades visuais corriqueiras consistem de luz refletida dos objetos. A natureza básica de *f(x,y)* pode ser caracterizada por dois componentes: (1) a quantidade de luz incidindo na cena sendo observada e (2) a quantidade de luz refletida pelos objetos na cena. Apropriadamente, esses dois componentes são chamados iluminação e

reflectância, respectivamente, e são representadas por *i(x,y)* e *r(x,y)*. O produto destas duas funções resulta em *f(x,y)* (Azevedo, 2003): $f(x,y)=i(x,y)r(x,y)$ , onde $0<i(x,y)<\infty$ e $0<r(x,y)<1$ . A reflectância indica que a mesma é limitada entre 0 (absorção total) e 1 (reflectância total). A natureza de *i(x,y)* é determinada pela fonte de luz, e *r(x,y)* é determinada pelas características dos objetos na cena. Os valores dados são cotas teóricas.

### 2.1.2 Armazenamento em memória

Para ser adequada para processamento computacional, uma função *f(x,y)* precisa ser digitalizada tanto espacialmente quanto em amplitude. A digitalização das coordenadas *(x,y)* é denominada *amostragem da imagem* e a digitalização da amplitude é chamada *quantização em níveis de cinza* (Azevedo, 2003).

Suponha que uma imagem contínua *f(x,y)* seja aproximada por amostras igualmente espaçadas, arranjadas na forma de uma matriz *N x M* como mostrado na Equação 1 abaixo, em que cada elemento é uma quantidade discreta:

$$f(x,y)=\begin{bmatrix} f(0,0) & f(0,1) & . & f(0,M-1) \\ f(1,0) & f(1,1) & . & f(1,M-1) \\ . & . & . & . \\ . & . & . & . \\ f(N-1,0) & f(N-1,1) & . & f(N-1,M-1) \end{bmatrix}$$

Equação 1: Matriz de elementos da imagem

O lado direito da igualdade representa o que é normalmente denominado uma *imagem digital*. Cada elemento da matriz denomina-se um *elemento de imagem*, *pixel* (abreviação de *picture element*) ou *pel*.

O processo de digitalização envolve decisões a respeito dos valores para *N*, *M* e o número de níveis de cores permitidos (*c*) para cada *pixel*. Então, para armazenar-se uma imagem digitalizada, o número *de* bits (*b*) necessário pode ser calculado pela Equação 2.

$$b = M.N.C$$

Equação 2: Cálculo do número de bits de uma imagem digitalizada

Por exemplo, uma imagem de 640 x 480 *pixels* com 64 níveis de cores (leva-se em consideração que podem ser utilizados 6 bits para fazer a representação destes 64 níveis – $2^6 = 64$ ) requer 1.843.200 bits (ou 230.400 bytes ou 225 Kb) para armazenamento.

Quanto aos níveis de cores, vejamos nos próximos tópicos algumas maneiras de representar as cores através dos mapas de bits (*bitmap)*.

#### 2.1.2.1 1 Bit por *Pixel* (*Black or White*)

Nesta forma de armazenamento, cada *pixel* é armazenado como um bit apenas (0 ou 1, indicando a ausência ou presença de cor – respectivamente preto e branco). Desta maneira, uma imagem de 640x480 requer apenas 37.5 Kb de espaço de armazenamento.

#### 2.1.2.2 Grayscale

Nesta forma de armazenamento, a imagem é armazenada em tons de cinza, e cada *pixel* é armazenado como um byte (valor entre 0 e 255). Uma imagem em 640x480 requer mais de 300Kb de espaço de armazenamento.

#### 2.1.2.3 Indexada (com uso de palhetas)

O número de cores atualmente entendido como o mínimo para representar razoavelmente uma imagem complexa é 256. Nas placas de vídeo de 256 cores, cada *pixel* é representado por um byte e o significado de cada byte depende de uma tabela de cores. Ou seja, ao invés de armazenar a cor, este sistema armazena um índice de uma tabela. A cor é encontrada na linha correspondente da tabela de

cores, como ilustra a Figura 1. Note que a memória que armazena os índices de imagem de 640×480 *pixels* possui 64 mil bytes e a tabela de cores apenas 768 bytes. A tabela de cores é também denominada palheta de cores (em inglês "*color table*", "*look up table*" ou "*LUT*").

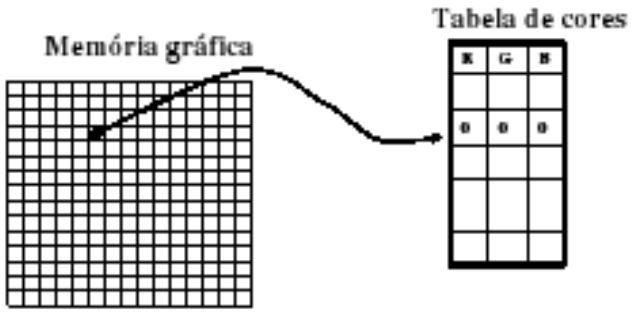



Figura 1: Entrada de índice na palheta de cores.

Fonte: Azevedo, 2003.

### 2.1.2.4 True color (24/32 bpp)

Este é o tipo de imagem que melhor representa as cores, utilizando o sistema RGB: este sistema utiliza três bytes de dados para armazenar as informações de cores: Vermelho (*Red*), Verde (*Green*) e Azul (*Blue*). Com essas cores conseguimos formar todas as outras, como citado na seção 2.1.1. Então, neste caso, são necessários três bytes de informações (ou 24 bits) para armazenar a informação de cor para cada *pixel*, resultando em mais de 16 milhões de cores distintas.

Algumas imagens armazenam um byte extra (*alpha*) com informações sobre transparência, neste caso ocupando 32 bits de dados por *pixel.*

### 2.1.3 Relacionamento entre *pixels*

#### 2.1.3.1 Vizinhança

A vizinhança de um *pixel* é definida como o conjunto de *pixels* que ficam ao redor de um ponto P (Facon, 2002). A proximidade entre o *pixel* e o ponto será definida conforme a classificação, que pode ser de dois tipos:

- 4-vizinhança: não leva em conta os pontos localizados nas diagonais passando pelo *pixel* P (Figura 2). A 4-vizinhança $V_4(P)$ é baseada na noção de uma distância $D_4$ dos quatro vizinhos mais próximos:

$$V_4(P)=\{Q \mid D_4(P,Q)\leq 1\}$$

Onde $D_4[P(x_p,y_p),Q(x_q,y_q)]=\|x_q-x_p|-|y_q-y_p\|$ e $|a|$ é módulo de a.

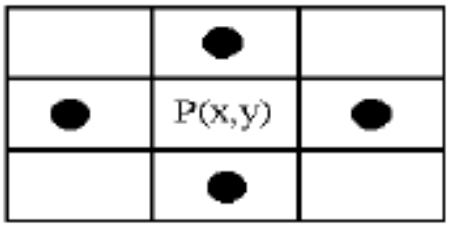



Figura 2: Exemplo de 4-vizinhança.

Fonte: Facon, 2002.

- 8-vizinhança: leva em consideração todos os *pixels* que ficam ao redor de um *pixel* P (Figura 3). A 8-vizinhança $V_8(P)$ é baseada na noção de distância $D_8$ dos 8 vizinhos mais próximos:

$$V_8(P)=\{Q \vee D_8(P,Q)<1\}$$

Onde $D_8[P(x_p,y_p),Q(x_q,y_q)]=max(|x_q-x_p|-|y_q-y_p|)$ e $|a|$ é módulo de a.

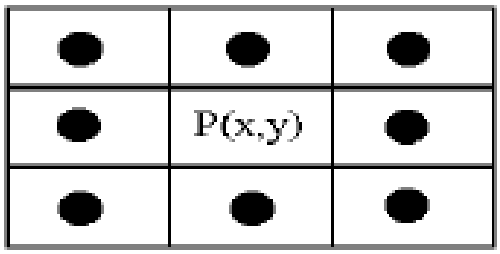



Figura 3: Exemplo de 8-vizinhança.

Fonte: Facon, 2002.

### 2.1.3.2 Conectividade

A conectividade entre *pixels* é um conceito importante usado no estabelecimento das bordas de objetos e componentes de regiões em uma imagem. Para estabelecer se dois *pixels* estão conectados, é preciso determinar se eles são de alguma forma adjacentes (digamos, se são 4-vizinhos) e se seus níveis de cinza satisfazem um certo critério de similaridade (digamos, se eles são iguais). Por exemplo, em uma imagem binária com valores de 0 e 1, dois *pixels* podem ser 4-vizinhos, mas eles não são ditos conectados a menos que tenham o mesmo valor (Gonzales, 2000).

## 2.1.4 Processamento de imagens

O processamento de imagens digitais é uma prática comum atualmente. Pode ser utilizada em diversas áreas, como por exemplo: detecção e reconhecimento de placas de automóveis e informações sobre um jogo de futebol (por exemplo, a distância de um chute realizado em direção ao gol). Na aplicação de reconhecimento das placas de automóveis, por exemplo, podemos dizer que o sistema vai “analisar” a imagem obtida e realizar as operações necessárias, conforme definido pelo software. Esse processo de análise é um processo da área de visão computacional. Em um sistema de visão computacional, o procedimento, conforme (Fogagnoli, 2000), pode ser resumido na Figura 4:

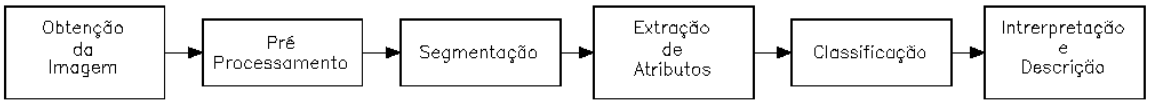



Figura 4: Procedimento resumido do processamento de imagens.

Fonte: Fogagnoli, 2000.

Em seguida, serão detalhadas cada uma das etapas do processamento de imagens.

### 2.1.4.1 Obtenção e características da imagem

Segundo (Gonzalez, 2000), dois elementos são necessários para a aquisição de imagens digitais: o primeiro é um dispositivo físico que seja sensível a uma banda do espectro de energia eletromagnética (como raios X, ultravioleta, etc) e que produza um sinal elétrico de saída proporcional a um nível de energia recebido; o segundo, chamado *digitalizador*, é um dispositivo para a conversão da saída elétrica deste dispositivo para a forma digital. Basicamente, esse dispositivo precisa "detectar" as cores e discretizá-las conforme já comentado anteriormente.

Podemos citar alguns dispositivos de entrada:

- câmera digital: dispositivo que captura as informações de luminosidade e armazena em sua memória interna, que pode ser transferida para o computador através de um cabo ou dispositivo de armazenamento;
- mesa digitalizadora: dispositivo vetorial que consiste em uma mesa e em um apontador. Cada vez que o usuário toca a mesa com o apontador é informado ao computador a coordenada deste ponto da mesa. Existem diversos trabalhos em andamento para a substituição deste periférico por sistemas mais baratos como câmeras digitais e reconhecimento de padrões;
- *scanner*: dispositivo que discretiza as informações de uma foto para a memória do computador.

Como dispositivo de entrada utilizado neste trabalho, iremos detalhar um pouco as imagens adquiridas pela câmera digital. Existem alguns parâmetros que são importantes no momento de processar as imagens, sendo os principais:

- coordenadas da câmera: é a posição (X, Y, Z) em relação a um referencial cartesiano.
- distorções da lente: segundo (Espinhosa, 2006), um raio luminoso ao atravessar uma lente, ou sistema de lentes, sofre uma série de desvios, provocando assim um deslocamento indesejável na imagem. Os principais efeitos são a distorção radial (ou de barril, que pode ser definida como a componente radial indesejável da refração sofrida por um raio de luz ao atravessar uma lente) e a distorção descentrada (que possui as componentes tangenciais e radiais assimétricas).

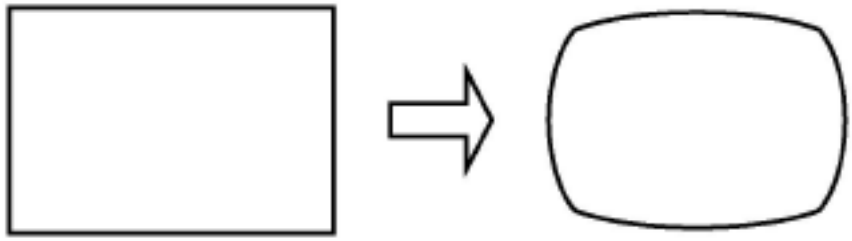


Figura 5: Imagem real e imagem distorcida.

Fonte: Kato, 1999.

### 2.1.4.2 Pré-processamento

O pré-processamento, também conhecido como realce, inclui atividades desde o próprio processo de formação da imagem até compensações, como redução de ruído ou do borramento da imagem. Essas funções podem ser comparadas aos processos de sensação e adaptação que ocorrem com uma pessoa tentando achar uma poltrona imediatamente depois de entrar em um teatro escuro vindo de um lugar iluminado com a luz do sol. O processo (inteligente) de busca por uma poltrona desocupada não pode começar enquanto uma imagem apropriada não estiver disponível (Gonzalez, 2000).

Um exemplo de pré-processamento está citado na Figura 6.

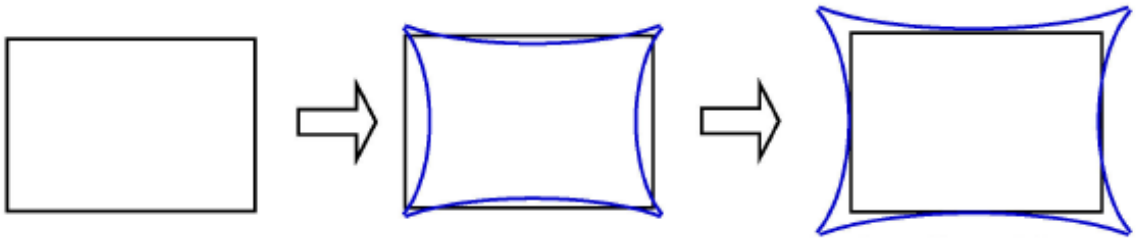


Figura 6: Exemplo de pré-processamento de imagem.

No primeiro quadro, exemplificamos a imagem observada, no segundo a imagem compensada (que não cabe na tela) e no terceiro a imagem ajustada em escala.

Fonte: Kato, 1999.

### 2.1.4.3 Segmentação

A segmentação é considerada o primeiro passo na análise da imagem. Basicamente, subdivide uma imagem em suas partes ou objetos constituintes. O nível até o qual essa subdivisão deve ser realizada depende do problema sendo resolvido. Ou seja, a segmentação deve parar quando os objetos de interesse na aplicação tiverem sido isolados. Em geral, a segmentação é uma das tarefas mais difíceis em processamento de imagens. Esse passo determina o eventual sucesso ou fracasso da análise (Gonzales, 2000).

Os algoritmos de segmentação para imagens monocromáticas são geralmente baseados em uma das seguintes propriedades básicas de valores de níveis de cinza: descontinuidade e similaridade. Na primeira categoria, a abordagem é de particionar a imagem baseado em mudanças bruscas nos níveis de cinza. As principais áreas de interesse nessa categoria são a detecção de pontos isolados e a detecção de linhas e bordas na imagem. As principais abordagens da segunda categoria baseiam-se em limiarização, crescimento de regiões e divisão e fusão de regiões.

O conceito de segmentação em uma imagem baseado em descontinuidade ou em similaridade dos valores de nível de cinza de seus *pixels* pode ser aplicado

tanto em imagens estáticas como em imagens dinâmicas (que variam com o tempo). Nesse último caso, porém, o movimento pode freqüentemente ser usado como uma pista poderosa para melhorar a performance dos algoritmos de segmentação.

## Detecção de descontinuidades

Há basicamente três tipos de descontinuidades em imagens digitais: pontos, linhas e bordas. Na prática, a maneira mais comum de procura por descontinuidade é através da varredura de imagem por uma pequena matriz (digamos 3x3, na qual os valores dos coeficientes determinam a natureza do processo) denominada máscara ou molde. No caso da máscara 3x3 mostrada na Figura 7, esse procedimento envolve o cálculo da soma dos produtos dos coeficientes pelos níveis de cinza contidos na região englobada pela mesma. Ou seja, a resposta da máscara em qualquer ponto da imagem é descrita na Equação 3 abaixo:

$$R = w_1 z_1 + w_2 z_2 + \ldots + w_9 z_9 = \sum_{i=1}^{9} (W_i Z_i)$$

Equação 3: Resposta da máscara para um ponto da imagem.

| | | |
|---|---|---|
| $w_1$ | $w_2$ | $w_3$ |
| $w_4$ | $w_5$ | $w_6$ |
| $w_7$ | $w_8$ | $w_9$ |

Figura 7: Máscara 3x3 genérica.

Fonte: Gonzales, 2000.

em que $z_i$ é o nível de cinza do *pixel* associado com o coeficiente $w_i$ da máscara. Como de costume, a resposta da máscara é definida em relação à sua posição central. Quando a máscara é posicionada em um *pixel* da borda, a resposta é computada, utilizando-se a vizinhança parcial apropriada.

## Detecção de pontos

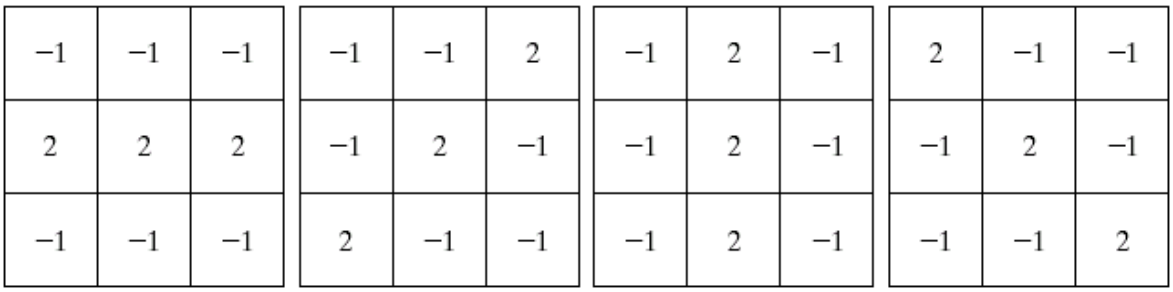
A detecção de pontos isolados em uma imagem pode ser obtida de maneira direta. Usando a máscara mostrada na Figura 7, nós dizemos que um ponto foi detectado na posição da máscara se:

$$|R| > T$$

em que $T$ é um limiar não-negativo e $R$ é dado pela Equação 3. Basicamente, tudo o que essa formulação faz é medir as diferenças ponderadas entre o ponto central e seus vizinhos. A idéia é que o nível de cinza de um ponto isolado será completamente diferente do nível de cinza de seus vizinhos.

## Detecção de linhas

Para a detecção de linhas, considere a máscara mostrada na Figura 8. Se a primeira máscara fosse movida por toda parte em uma imagem, ela deveria responder fortemente a linhas (largura de um *pixel*) orientadas horizontalmente. Com o fundo constante, uma resposta máxima deveria resultar quando a linha passasse pela linha do meio da máscara. Isso pode ser facilmente verificado rascunhando-se uma matriz simples de elementos -1 com uma linha composta por elementos com um nível de cinza diferente (por exemplo, 2) posicionada horizontalmente na mesma. Um experimento similar revelaria que a segunda máscara da Figura 8 responde melhor a linhas orientadas a 45º; a terceira máscara a linhas verticais; e a quarta máscara a linhas na direção – 45º. Essas direções podem também ser estabelecidas notando-se que a direção preferencial de cada máscara é ponderada com um coeficiente maior (ou seja, 2) que outras direções possíveis.



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| −1 | −1 | −1 | −1 | −1 | 2 | −1 | 2 | −1 | 2 | −1 | −1 |
| 2 | 2 | 2 | −1 | 2 | −1 | −1 | 2 | −1 | −1 | 2 | −1 |
| −1 | −1 | −1 | 2 | −1 | −1 | −1 | 2 | −1 | −1 | −1 | 2 |

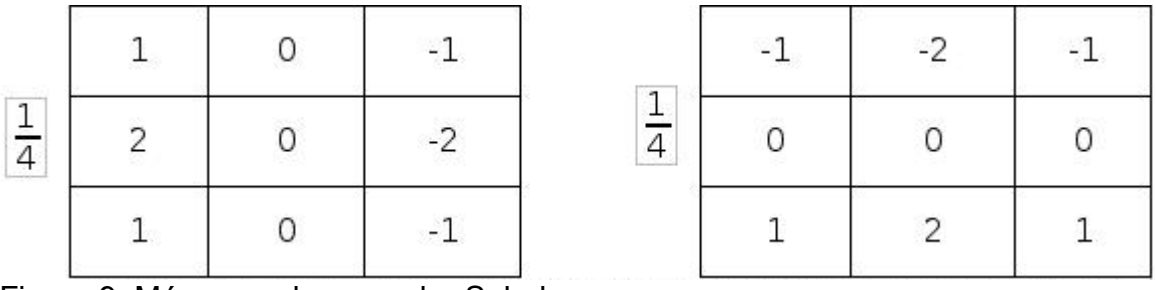
Figura 8: Máscaras para detecção de linhas.

Fonte: Gonzales, 2000.

## Detecção de bordas

A detecção de bordas é a abordagem mais comum para a detecção de descontinuidades significantes nos níveis de cinza. A razão é que pontos e linhas ficam isoladas e não são ocorrências freqüentes na maioria das aplicações práticas.

Segundo (Marquês Filho, 1999), borda é a fronteira entre duas regiões cujos níveis de cinza predominante são razoavelmente diferentes. Entre várias técnicas e máscaras existentes, há vários operadores para detecção de bordas: os operadores de Roberts, Sobel, Prewitt e Freichen, sendo que suas máscaras são divididas em sentido horizontal e vertical (Borges, 2005). Abaixo, exemplificamos as máscaras correspondentes do operador Sobel para os sentidos vertical e horizontal, respectivamente:



$\frac{1}{4}$

| | | |
|---|---|---|
| 1 | 0 | -1 |
| 2 | 0 | -2 |
| 1 | 0 | -1 |

$\frac{1}{4}$

| | | |
|---|---|---|
| -1 | -2 | -1 |
| 0 | 0 | 0 |
| 1 | 2 | 1 |

Figura 9: Máscaras do operador Sobel.

Fonte: Marquês Filho, 1999.

As Figuras abaixo ilustram um exemplo de utilização destes operadores:

Figura 10: Exemplo de utilização do operador Sobel

a) Imagem original; b) Resultado do operador Sobel vertical; c) Resultado do operador Sobel horizontal.

Fonte: Marquês Filho, 1999.

## Limiarização

Esta é uma das mais importantes abordagens para a segmentação de imagens, porém antes de entrarmos em detalhes sobre a limiarização, veremos um pouco sobre os histogramas.

Segundo (Gonzales, 2000), o histograma de uma imagem digital com níveis de cinza no intervalo *[0, L-1]* é uma função discreta $p(r_k) = \frac{n_k}{n}$, em que $r_k$ é o k-ésimo nível de cinza, $n_k$ é o número de *pixels* na imagem com esse nível de cinza, $n$ é o número total de *pixels* na imagem, $k = 0,1,2,...,L-1$ e $L$ é o número de níveis de cinza existentes.

De forma aproximada, $p(r_k)$ dá uma estimativa da probabilidade de um *pixel* qualquer da imagem ter esse nível de cinza $r_k$ na imagem analisada. Um gráfico dessa função para todos os valores de $k$ fornece uma descrição global da aparência de uma imagem.

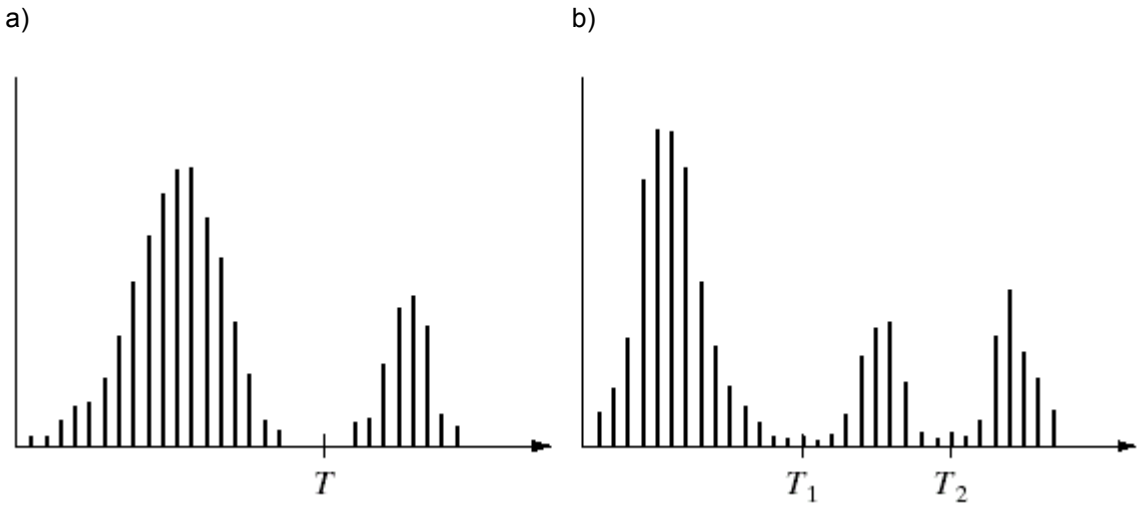



Figura 11: Exemplos de histogramas.

a) à esquerda com um limiar único. b) à direta com múltiplos limiares.

Fonte: Gonzales, 2000.

Suponha que o histograma de níveis de cinza mostrado na Figura 11a corresponde a uma imagem, *f(x,y)*, composta por objetos iluminados sobre um fundo escuro, de maneira que os *pixels* do objeto e os do fundo tenham seus níveis de cinza agrupados em duas classes dominantes. Uma maneira óbvia de extrair os objetos do fundo é através da seleção de um limiar *T* que separe os dois grupos. Então, cada ponto *(x,y)* tal que *f(x,y) > T* é denominado um ponto do objeto; caso contrário, o ponto é denominado um ponto do fundo. A Figura 11b mostra um caso ligeiramente mais geral dessa abordagem. Nesse caso, três grupos dominantes caracterizam o histograma da imagem (por exemplo, dois tipos de objetos iluminados sobre um fundo escuro). A mesma abordagem clássica classifica um ponto *(x,y)* como pertencendo à classe de um dos objetos se $T_1 < f(x,y) \leqslant T_2$ , à classe do outro objeto se $f(x,y) > T_2$ e ao fundo se $f(x,y) \leqslant T_1$ . Esse tipo de *limiarização multiníveis* é geralmente menos confiável que a de limiar único. A razão é a dificuldade do estabelecimento de múltiplos limiares que isolem efetivamente as regiões de interesse, especialmente quando o número de grupos correspondentes do histograma for grande. Tipicamente, problemas dessa natureza, se tratados por limiarização são melhor resolvidos com um limiar único variável.

Baseado na discussão precedente, a limiarização pode ser vista como uma operação que envolve testes de uma função *T* da forma

$$T = T[x, y, p(x, y), f(x, y)]$$

em que *f(x,y)* é o nível de cinza do ponto *(x,y)* e *p(x,y)* denota alguma propriedade local desse ponto, por exemplo, o nível de cinza médio em uma vizinhança centrada em *(x,y)*. Uma imagem limiarizada *g(x,y)* é definida como:

$$g(x, y) = \begin{cases} (1 \text{ se } f(x, y) > T) \\ (0 \text{ se } f(x, y) \leq T) \end{cases}$$

Portanto, *pixels* rotulados como 1 (ou qualquer outro nível de cinza conveniente) correspondem ao objeto, enquanto que aqueles rotulados como 0 correspondem ao fundo.

Quando *T* depender apenas de *f(x,y)*, o limiar será chamado *global* (a Figura 11a mostra um exemplo de tal limiar). Se *T* depender tanto de *f(x,y)* quanto de *p(x,y)*, então o limiar será chamado *dinâmico* ou *adaptativo*, que consiste em perceber alterações de iluminação no ambiente (ou parte da imagem).

### 2.1.4.4 Extração e seleção de atributos

O objetivo da extração de atributos é caracterizar medidas associadas ao objeto que se deseja extrair, de forma que estas sejam semelhantes para objetos similares e diferentes para objetos distintos (Duda, 2001).

Basicamente, a extração de atributos permite separar a imagem em mais de um objeto presente. A seleção de atributos pode ser vista como um processo de busca onde o algoritmo usado deve encontrar o menor subconjunto destes com a melhor acurácia de classificação (Pappa, 2002). Uma grande dimensionalidade do espaço de atributos pode causar degradação na classificação e um alto custo de processamento (Santos, 2007).

### 2.1.4.5 Classificação

A classificação de imagens consiste em associar cada *pixel* ou região da imagem a uma classe ("rótulo") que descreve um objeto real. A tarefa do classificador é usar o vetor fornecido pelo processo de extração e seleção de atributos para atribuir o objeto a esta classe (Duda, 2001). A obtenção de uma classificação ideal, que corresponda totalmente a realidade, é quase impossível, sendo uma tarefa mais geral estimar a probabilidade de que um padrão (objeto) pertença a uma determinada classe, baseando-se nos valores de alguns atributos ou em um conjunto de atributos.

### 2.1.4.6 Reconhecimento e interpretação

Interpretação de imagens refere-se a atividade de transformar figuras em descrições ou símbolos comumente entendidos. Técnicas de reconhecimento de padrões são amplamente utilizadas nesta atividade, realizando a classificação da imagem para um dos tipos de figura já conhecidos, o que requer também um aprendizado anterior (Antunes, 1999).

## 2.1.5 Identificação de marcadores fiduciais

Segundo (Owen, 2002), imagens fiduciais são imagens localizadas em um ambiente físico para prover informações sobre o ambiente: alinhamento, identificação, configurações. Por exemplo, placas de circuito possuem marcadores que as permitem ser alinhadas precisamente a outros circuitos ou camadas, para que máquinas ou robôs possam inserir componentes corretamente. Em sistemas de Realidade Aumentada, as imagens fiduciais são geralmente utilizadas para atribuir elementos em seu ambiente. Estes marcadores podem ser posicionados no ambiente para permitir um ajuste e configuração da câmera ou podem ser posicionados em objetos que se movimentam (ou podem ser movimentados por alguma pessoa que esteja interagindo com o ambiente). Os marcadores mais simples podem ser definidos através de pontos, retângulos, círculos ou outras formas simples.

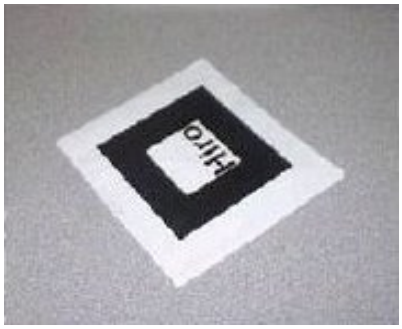


Figura 12: Exemplo de marcador utilizado no ARToolKit.

Fonte: Lamb,2007.

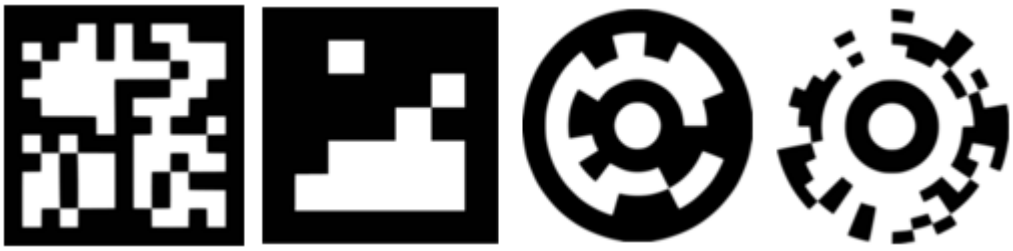


Figura 13: Exemplo de marcador utilizado pelo Cantag.

Fonte: Cantag, 2007.

O maior problema neste tipo de marcador é o processamento e localização do mesmo em uma imagem capturada: há grandes chances de o sistema localizar algum outro ponto na imagem e "interpretar" este ponto como um marcador, apesar de ser apenas uma característica da imagem sendo processada. Então podemos definir alguns critérios para definir um marcador, segundo (Owen, 2002):

- um marcador ideal deve suportar a determinação de posição e orientação única em uma câmera calibrada;
- a imagem não deve sobrepor outras;
- a imagem deve ser parte de um grupo de imagens que não podem ser confundidas entre si ou com algum outro objeto;
- a imagem precisa ser fácil de localizar e identificar, utilizando algoritmos simples e rápidos;
- as imagens precisam funcionar sobre uma faixa de captura da câmera.

Analisando a Figura 12, podemos perceber que o marcador contém uma palavra escrita. Caso alguma pessoa no ambiente tiver usando uma camiseta com uma palavra semelhante, há grande probabilidade de o sistema “detectar” a camiseta como o marcador, exibindo o objeto virtual (cuja posição será definida pelo marcador) em uma posição incorreta. Já existem várias ferramentas para localização de marcadores, que veremos em mais detalhes na seção 2.4.

### 2.1.6 Síntese

Na síntese de imagens, os objetos, modelados no sistema cartesiano, são, inicialmente, projetados no sistema de coordenadas projetivas e, finalmente, descritos em coordenadas de dispositivo, para poderem ser materializadas segundo primitivas geométricas na imagem (Gonçalvez, 2004).

Os algoritmos de síntese de imagens dependem estritamente do modelo utilizado para descrever o ambiente. Não é possível sintetizar uma imagem se não se conhece como descrever um objeto tridimensional (Kim, 1992). Abaixo iremos detalhar um pouco a modelagem geométrica, utilizada para descrever-se um objeto tridimensional.

#### 2.1.6.1 Modelos geométricos

Para que possamos visualizar um objeto devemos antes criar uma descrição deste. Este processo de criação de descrições de objetos é denominado na Computação Gráfica de *modelagem*, a qual gera um modelo através do qual fica mais fácil e prático analisar e testar (como por exemplo a aerodinâmica de automóveis e aviões) (Casacurta, 1998).

O intuito de construir-se um modelo é o de criar uma representação da forma mais conveniente possível para facilitar o seu uso e análise. Dependendo da aplicação, o modelo geométrico pode requerer uma descrição detalhada das propriedades das faces como reflexão, transparência, textura, cor, ou por outro lado apenas informações sobre as propriedades geométricas e de massa do modelo. Se

o modelo é bastante rico em detalhes, pode-se executar operações sobre ele tais que produzam os mesmos resultados que aquelas sobre o objeto real.

Existem sistemas de desenho que trabalham com modelos bidimensionais mas os sistemas tridimensionais apresentam uma complexidade bem maior conforme o grau de realismo do sistema.

### 2.1.6.2 Modelos 2D

Os modelos bidimensionais são descritos através do plano cartesiano X e Y, onde cada ponto do modelo é representado por um par ordenado de coordenadas. Nos modelos de duas dimensões as entidades básicas são os pontos, os segmentos de reta e os polígonos.

#### Ponto

É representado como uma seqüência de dois números reais, $\dot{P}=(P_x, P_y)$ .

#### Segmento de Reta

Um segmento de reta (também chamada linha ou aresta) é representado como dois pontos interligados.

#### Polígono

É representado como uma seqüência de segmentos de reta interligados. Pode ser armazenado num vetor ou numa lista ligada e circular. Vamos escrever um polígono *G* como *(G0 , G1 , ..., Gn−1 , Gn = G0 )* ou como *(G0 , G1 , ..., Gn−1 )*.

Um polígono no plano pode estar orientado em sentido horário ou anti-horário. Por exemplo, *G = ((0, 0), (1, 0), (0, 1), (0, 0))* está em sentido anti-horário enquanto que *H = ((0, 0), (0, 1), (1, 0), (0, 0))* está em sentido horário.

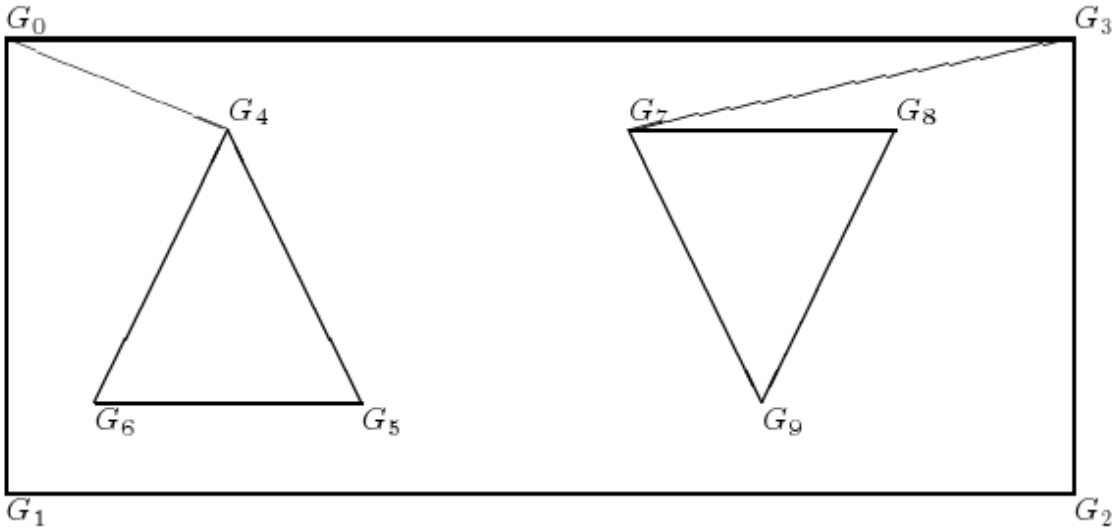



Figura 14: Exemplos de polígonos.

Fonte: Kim, 1992.

### 2.1.6.3 Modelos 3D

Os modelos tridimensionais são descritos através de suas coordenadas espaciais, ou seja, coordenadas X, Y e Z. Nos modelos de três dimensões além das entidades ponto e aresta surge a entidade face, que é uma seqüência ordenada de pontos e arestas. O sentido correto da seqüência dos pontos será importante para os algoritmos de remoção de elementos ocultos.

### 2.1.6.4 Iluminação

Se considerarmos o funcionamento da visão humana, o requisito essencial para se enxergar objetos é a existência de luz. A luz, presente de várias formas, é necessária para que os objetos possam refletir e absorver seus raios, e então, serem percebidos pelo olho humano. Dessa forma, quando se fala de realismo em imagens geradas por Computação Gráfica, é necessário inicialmente compreender a natureza das fontes de luz e suas interações com os objetos.

Denomina-se fonte de luz puntual (Figura 15a) aquela cujos raios de luz emanam uniformemente em todas as direções a partir de um único ponto no

espaço. Esse tipo de fonte de luz é comparável a uma lâmpada, e a iluminação na superfície de um objeto evidentemente irá variar de acordo com a distância e direção da fonte de luz.

Já uma fonte de luz direcional (Figura 15b) é aquela cujos raios de luz vêm sempre da mesma direção. Dessa forma, a orientação de cada superfície e a direção da fonte de luz determinam o seu efeito. Por exemplo, uma face é plenamente iluminada se estiver perpendicular aos raios de luz incidentes. Quanto mais oblíqua uma face estiver em relação aos raios de luz, menor será a sua iluminação. A luz solar é o exemplo clássico de fonte de luz direcional: apesar do Sol estar em um ponto no espaço (o que a princípio poderia sugerir que é uma fonte de luz puntual), como a sua distância até a Terra é muito grande, considera-se que os raios de luz chegam praticamente paralelos à nossa superfície.

O terceiro tipo mais comum de fonte de luz é a do tipo *spot* (Figura 15c). Esse tipo é na verdade uma combinação de uma luz puntual com um componente direcional: os raios de luz são emitidos na forma de um cone, apontado para uma determinada direção. O exemplo mais comum desse tipo de fonte de luz é um abajur ou uma lanterna. Na luz do tipo *spot*, a intensidade diminui conforme o raio de luz é desviado da direção para a qual a fonte de luz está apontada. Freqüentemente, modela-se fontes de luz desse tipo através de parâmetros adicionais, que definem o ângulo de abrangência e a função de atenuação da intensidade de luz (Manssour, 2006).

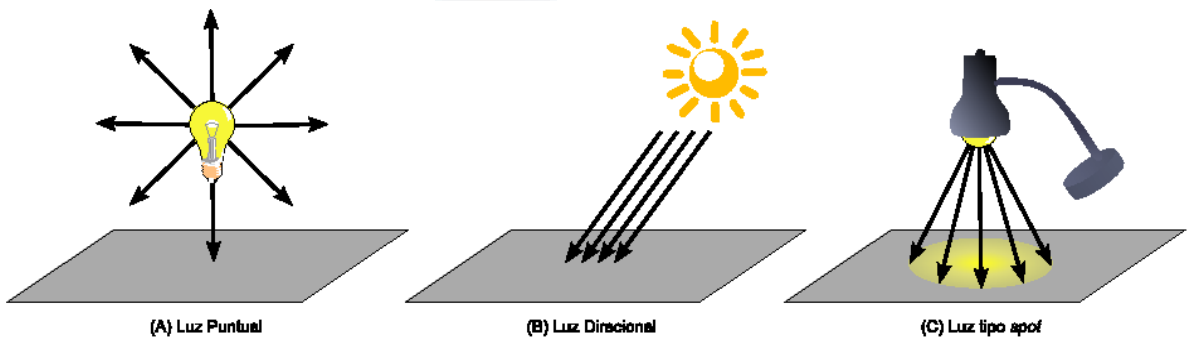
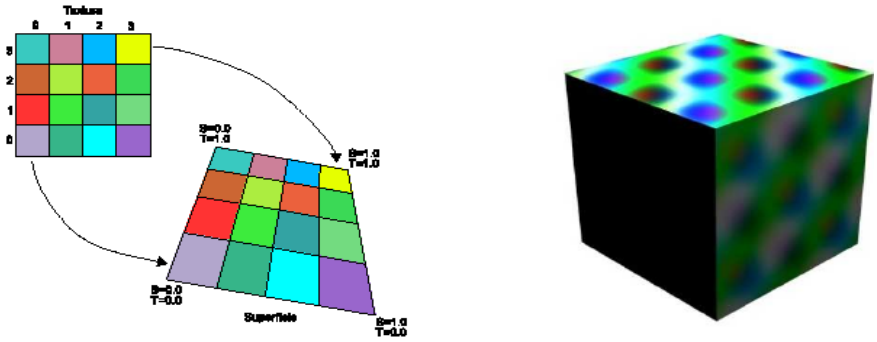



Figura 15: Tipos de iluminação.

Fonte: Manssour, 2006.

### 2.1.6.5 Textura

Mesmo através do recurso de iluminação, freqüentemente os objetos não terão uma aparência realística. Isso acontece porque os materiais na realidade não tem simplesmente cores diferentes, mas sim uma série de características físicas, como rugosidade e textura, que determinam como exatamente refletem os raios de luz. Em Computação Gráfica, costuma-se modelar essas características através de um conjunto de diversas técnicas, que podem ser combinadas. A técnica mais comum é denominada genericamente mapeamento de textura, ou textura mapeada. A idéia é utilizar uma imagem que contenha a aparência da superfície desejada (por exemplo, madeira ou metal). Essa imagem é então, durante o processo de rasterização, mapeada sobre a superfície através de coordenadas de textura. Estas coordenadas (usualmente chamadas de *set*) determinam como o mapeamento é realizado, conforme a Figura 16a apresenta. A textura funciona então, como um decalque ou um papel de parede, sendo "colada" à superfície. É importante observar que o mapeamento de textura pode (e geralmente é) combinado com o processo de iluminação, gerando um resultado ainda melhor (Figura 16b) (Manssour, 2006).



(a) Coordenadas de textura (b) Efeito com iluminação

Figura 16: Mapeamento de textura.

Fonte: Manssour, 2006.

### 2.1.6.6 Visualização

Existe um conjunto de técnicas em Computação Gráfica que permite transformar as informações a respeito de um modelo contidas em uma estrutura de dados, em uma imagem que pode ser exibida em um monitor. Portanto, considera-se que uma imagem consiste em uma matriz de pontos e um modelo é uma representação computacional de um objeto.

### 2.1.6.7 Transformações geométricas

Transformações geométricas são aquelas que modificam a posição, a dimensão e a forma de objetos tridimensionais (Ros, 2001). Em seguida, veremos algumas das principais transformações geométricas.

### Translação

A translação pode ser especificada por uma matriz de translação tridimensional, que determina como um objeto deve ser deslocado em cada uma das três direções.

Em uma representação por coordenadas homogêneas[1] tridimensionais, um ponto é transladado da posição *P=(x,y,z)* para a posição *P'=(x',y',z')* com a operação apresentada na Equação 4:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Equação 4

1 Coordenadas homogêneas são coordenadas normalizadas, que normalmente adicionam uma nova dimensão para implementar esse efeito de normalização. Por isso que transformações em três dimensões utilizam matrizes 4x4.

### Escalamento

Escalamento (ou transformação em escala) é uma operação bastante usual em Computação Gráfica. Ela permite compor um objeto, redimensionando horizontalmente ou verticalmente cada um de seus elementos. Na Equação 5, podemos ver a matriz de transformação para escala. Devemos observar que se os parâmetros $s_x$, $s_y$, $s_x$ não são iguais, as dimensões relativas a cada eixo do objeto são alteradas separadamente.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Equação 5

### Rotação

Para gerar uma transformação de rotação em um objeto, deve ser designado um eixo de rotação, sobre o qual o objeto será rotacionado, além de um ângulo de rotação. Para cada eixo, é necessária uma matriz de transformação, conforme demonstra as Equações 6, 7 e 8 para os eixos x, y e z, respectivamente.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(a) & sen(a) & 0 \\ 0 & -sen(a) & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Equação 6

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(a) & 0 & -sen(a) & 0 \\ 0 & 1 & 0 & 0 \\ sen(a) & 0 & \cos(a) & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Equação 7

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(a) & sen(a) & 0 & 0 \\ -sen(a) & \cos(a) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Equação 8

## 2.2 Vídeo

Basicamente, um vídeo corresponde a uma seqüência de quadros ou imagens. Segundo (Nuñez, 2006), um quadro de vídeo capturado de uma transmissão de TV e uma foto tirada com uma máquina fotográfica digital de última geração trazem uma diferença muito grande em termos de resolução e qualidade.

Esta imagem de péssima qualidade passa despercebida, pois normalmente é vista por apenas aproximadamente 0,03 segundos (na taxa de amostragem de televisão, que é de aproximadamente 30 quadros por segundo). Ao congelar uma imagem e analisá-la detalhadamente, é possível perceber que sua qualidade não é nem próxima à de uma fotografia.

### 2.2.1 Processamento de vídeo

Sendo o vídeo uma seqüencia de imagens, pode-se afirmar que o processamento de vídeo é o processamento de várias imagens seqüencialmente.

#### 2.2.1.1 Desafios

No que se trata de processamento de imagens para a extração de marcadores fiduciais, segundo (Nuñez, 2006), os trinta quadros por segundo iludem a vista humana, mas quando são processados individualmente se mostram um sério problema. Apesar das máquinas atualmente possuirem características de processamento muito avançadas, ainda é complicado processar 30 imagens em um segundo. Além disso, é necessário que os algoritmos de processamento sejam robustos o suficiente para localizar os marcadores, assim como resolver alguns dos problemas comuns:

- distorção de cores (com maior ou menor luminosidade, tem-se cores diferentes);
- oclusão dos marcadores (quando o marcador fica oculto por outro objeto ou ator da cena);
- limitação espacial: o marcador está muito distante da câmera, e devido à baixa resolução da imagem fica difícil localizá-lo;
- inclinação dos marcadores: conforme são inclinados, fica mais difícil “reconhecer” os padrões do centro;
- movimentação dos marcadores, visto que eles podem mover-se.

#### 2.2.1.2 Coerência espacial

A coerência espacial é basicamente a relação de um determinado objeto entre os diversos quadros do vídeo. Na localização de marcadores, por exemplo, podemos localizar o marcador no primeiro quadro e nos próximos quadros podemos procurar apenas nos *pixels* mais próximos, levando em consideração que o marcador pode ter sido movimentado.

### *2.2.2 Tracking*

O registro ou o alinhamento dos objetos virtuais com a cena real é feito com o auxílio de rastreadores ou *trackers*. Para compreender bem este relacionamento, é interessante seguir todos os passos deste processo. O problema aqui é saber em que posição e com qual orientação os objetos virtuais deverão ser colocados em relação a cena real. Para identificar este posicionamento, é necessário relacionar as coordenadas dos objetos virtuais com as da cena real.

Torna-se necessário também saber qual é a posição e orientação do usuário. Para obter esta informação, usa-se os equipamentos rastreadores. Rastreadores são dispositivos capazes de medir a posição e orientação de um sensor em relação a uma fonte. Após adquiridos, os dados do rastreador são passados para o sistema de renderização com o objetivo de produzir resultados que sejam realistas e precisos em relação ao mundo real.

Há vários tipos de rastreadores, porém vamos focar este estudo nos rastreadores ópticos, basicamente devido ao baixo custo do mesmo.

Segundo (Souza Silva, 2005), os mais bem sucedidos métodos de rastreamento óptico lidam com o uso de marcadores colocados no ambiente para serem detectados através de algoritmos de reconhecimento de imagem.

### 2.2.3 Animação

Segundo (Kuhn, 2005), o conceito de animação é antigo e analisando as suas origens, nota-se que os primeiros trabalhos nesta área são datados muito antes de invenção dos computadores. Segundo (Parent, 2002), a definição mais simples para animação é a geração de uma seqüência de imagens que retrata o movimento relativo de objetos de uma cena sintética, e possivelmente, do movimento de uma câmera virtual.

## 2.3 Realidade Virtual

Segundo (Tori, 2006), nos ambientes virtuais, a interação mais simples consiste na navegação, que ocorre quando o usuário se movimenta no espaço tridimensional, usando algum dispositivo, como o mouse 3D, ou gestos detectados por algum dispositivo de captura, tendo como resposta a visualização de novos pontos de vista do cenário. Nesse caso, não há mudanças no ambiente virtual, somente um passeio exploratório. Interações, propriamente ditas, com alterações no ambiente virtual ocorrem quando o usuário entra no espaço virtual das aplicações e visualiza, explora, manipula e aciona ou altera os objetos virtuais, usando seus sentidos, particularmente os movimentos tridimensionais de translação e rotação naturais do corpo humano. O mais importante nesta interação é que o usuário tenha a impressão de estar presente neste ambiente virtual (que ele tenha a impressão que o ambiente é real), apontando, pegando, manipulando e executando outras ações sobre os objetos virtuais, em tempo-real, ou seja, dentro de limites de tempo bem definidos, ou com atrasos que não lhe causem desconforto.

Segundo (Braga, 2007), a interação do usuário com o ambiente virtual é um dos aspectos importantes da interface e está relacionada com a capacidade do computador detectar as ações do usuário e reagir instantaneamente, modificando aspectos da aplicação. A possibilidade de o usuário interagir com um ambiente virtual tridimensional realista em tempo-real, vendo as cenas serem alteradas como resposta aos seus comandos, característica dominante nos *vídeo-games* atuais, torna a interação mais rica e natural propiciando maior engajamento e eficiência. A grande vantagem desse tipo de interface está no fato de as habilidades e

conhecimento intuitivos do usuário poderem ser utilizados para a manipulação dos objetos virtuais.

### 2.3.1 Interação no ambiente virtual

Um ambiente virtual exige uma interação em tempo real, normalmente feita com dispositivos não convencionais. O dispositivo mais tradicional quando se fala em Realidade Virtual é o capacete de RV (ou HMD - *Head Monted Display*), porém existem vários dispositivos que permitem essa interação, conforme pode ser visto nas figuras a seguir:


Figura 17: Tipo de capacete RV.

Fonte: Keller, 2007.

Figura 18: Luva de RV.

Fonte: Keller, 2007.

Figura 19: Tipo de mouse RV.

Fonte: Keller, 2007.

Figura 20: Roupa de RV.

Fonte: Keller, 2007.

O maior problema destes equipamentos para interação é o preço, que ainda está pouco acessível (os equipamentos mais baratos custam algumas centenas de dólares). Neste estudo, iremos utilizar um equipamento mais comum e acessível: uma *webcam*.

### 2.3.2 Tipos de Realidade Virtual

#### 2.3.2.1 Realidade Virtual de simulação

Segundo (Netto, 2002), a simulação representa o tipo mais antigo de sistema de RV porque se originou com os simuladores de vôo desenvolvidos pelos militares americanos depois da Segunda Guerra Mundial.

Um sistema de RV de simulação basicamente imita algum ambiente, como o interior de um carro, avião ou jato, colocando o participante dentro de uma cabine com controles. Dentro dessa cabine, telas de vídeo e monitores apresentam um mundo virtual que reage aos comandos do usuário. As imagens aparecem de forma bastante rápida. Em alguns sistemas as cabines são montadas sobre plataformas móveis, além de dispor de controles com *feedback* tátil e auditivo. Nesse caso, o usuário, através dos dispositivos de RV, participa de um mundo virtual gerado pelo aparato tecnológico, interagindo com ele em tempo real. O ambiente virtual, além de simular um ambiente real, como nas aplicações militares ou médicas, pode ainda criar um mundo imaginário com seus elementos e comportamentos.

As restrições físicas e comportamentais do mundo real podem ser quebradas no mundo virtual. O usuário pode optar por voar, passar por paredes e objetos, mergulhar no solo, navegar em alta velocidade, ser teletransportado de um ambiente para outro, e o que mais vier à sua imaginação (desde que o software permita).

A semelhança nos sistemas de simulação e telepresença está no uso de interfaces bem elaboradas, diferindo apenas na atuação sobre o ambiente. O sistema de simulação faz com que a interface atue direto sobre o computador que

vai atuar, por sua vez, com um mundo virtual real ou imaginário, enquanto a telepresença faz com que a interface atue sobre o telerobô que vai atuar sobre o mundo real.

#### 2.3.2.2 Telepresença

Este tipo de RV utiliza câmeras de vídeo e microfones remotos para envolver e projetar o usuário profundamente no mundo virtual. Em alguns casos onde possa haver dificuldade de transferência ou tratamento em tempo real de imagens reais complexas, a substituição do mundo real por um mundo virtual equivalente pode resolver o problema, na medida em que as imagens podem ser geradas localmente. As transferências de informações são reduzidas a dados de posicionamento.

O controle de robôs e a exploração planetária são exemplos de pesquisas em desenvolvimento. No entanto, existe um grande campo de pesquisa no uso de telepresença em aplicações médicas. Em intervenções cirúrgicas, já se utilizam câmeras de vídeo e cabos de fibra óptica para visualizar os campos operatórios de seus pacientes. Através da RV eles podem, literalmente intervir, indo direto ao ponto de interesse e/ou vistoriar outros procedimentos.

#### 2.3.2.3 Realidade Aumentada

A Realidade Aumentada ou Realidade Mista, é um misto de Realidade Virtual com a vida real. Segundo (Braga, 2007), RA é uma área em desenvolvimento na pesquisa da Realidade Virtual que se torna cada vez mais importante. Enquanto a RV busca levar o usuário a se sentir completamente imerso em um ambiente, a RA procura levar a agilidade digital a realçar aspectos do mundo real. A RV induz o participante a um sentimento de imersão em um mundo irreal através de técnicas computacionais nas quais ele pode agir e interagir e é uma tecnologia que abrange uma gama enorme de idéias e ações.

Um sistema de RA amplia o mundo real necessitando que o usuário mantenha o sentimento de sua presença naquele mundo. É preciso haver um mecanismo que combine os dois mundos e que não está presente no trabalho de

RV. A RA pode adicionar gráficos, sons e odores provenientes de um mundo natural. Ela trata da modificação do mundo real pela sobreposição – fusão – de objetos virtuais. Ela pode aumentar a nossa percepção para aspectos de um objeto invisíveis à primeira vista, como, por exemplo, tornando visível a textura do interior da orelha da estátua do David, de Michelangelo.

Para que as imagens do mundo real e do virtual sejam percebidas, registradas, fundidas em posição correta, é necessário que a posição e orientação da câmera estejam constantemente em estado de varredura. A maior parte das aplicações de RA usa a técnica de visão computacional para executar a varredura (Cunha, 2007). Ao contrário da RV, na qual há total imersão, a RA usa o mundo digital e sua flexibilidade para sublinhar detalhes de nosso meio ambiente. A RA está sendo cada vez mais pesquisada e desenvolvida em universidades e em empresas de alta tecnologia. Ao comparar RV e RA, percebemos que a última está em uma escala mais próxima do mundo real, já que ela usa os dois parâmetros - o real e o virtual. A RA está situada dentro dessas variações: (Milgram,1994) (Figura 21) descreve uma taxonomia que mostra como RV e RA estão relacionadas.


Figura 21: Variações de ambientes.

Fonte: Milgram,1994.

## 2.4 Trabalhos anteriores

Neste tópico, iremos descrever algumas das ferramentas e bibliotecas disponíveis para desenvolvimento de aplicações em RA.

### 2.4.1 ARToolKit

O ARToolKit foi originalmente desenvolvido para servir de apoio na concepção de interfaces colaborativas (telepresença) pelo Dr. Hirokazu Kato, na Universidade de Osaka, Japão. E, desde então, tem sido mantido pelo Human Interface Technology Laboratory (HIT Lab) da Universidade de Washington, EUA, e pelo HIT Lab NZ da Universidade de Canterbury, em Christchurch, Nova Zelândia. Segundo (Sementille, 2003), o ARToolKit é uma biblioteca escrita em C e permite o desenvolvimento facilitado de aplicações de RA.

O ARToolKit usa técnicas de visão computacional para calcular a relação entre o ponto de vista real da câmera e um marcador no mundo real. Há vários passos, conforme mostra as Figuras abaixo. Primeiro a imagem de vídeo (Figura 22a) é transformada em uma imagem binária (em preto e branco) baseada no valor do limiar de intensidade (Figura 22b). Depois, busca-se nesta imagem por bordas retas. O ARToolKit encontra todos os quadriláteros na imagem binária com base nas bordas, muitos dos quais não correspondem a marcadores de referência. Para cada quadrilátero, o desenho padrão dentro dele é capturado e comparado com alguns gabaritos pré-treinados. Se houver alguma similaridade, então o ARToolKit considera que encontrou um dos marcadores de referência. O ARToolKit usa então o tamanho conhecido do quadrilátero e a orientação do padrão encontrado para estimar a posição real da câmera em relação a posição real do marcador. Uma matriz 3x4 conterá a transformação da câmera em relação ao marcador. Esta matriz é usada para definir coordenadas da câmera virtual. Se as coordenadas virtuais e reais da câmera forem as mesmas, o modelo de Computação Gráfica pode ser desenhado precisamente sobre o marcador real (Figura 22c). A API OpenGL é usada para calcular as coordenadas virtuais da câmera e desenhar as imagens virtuais (Cardoso, 2004).

Figura 22: Etapas da detecção de marcadores

a) a captura da imagem; b) segmentação da imagem; c) posicionamento do objeto virtual sobre a imagem.

Fonte: Cardoso, 2004.

O diagrama da Figura 23 ilustra os passos do processamento de imagens usado para detectar a geometria do marcador e depois o posicionamento de um objeto virtual sobre este marcador detectado (Cardoso, 2004).

O ARToolkit será exposto em maiores detalhes no capítulo 3.

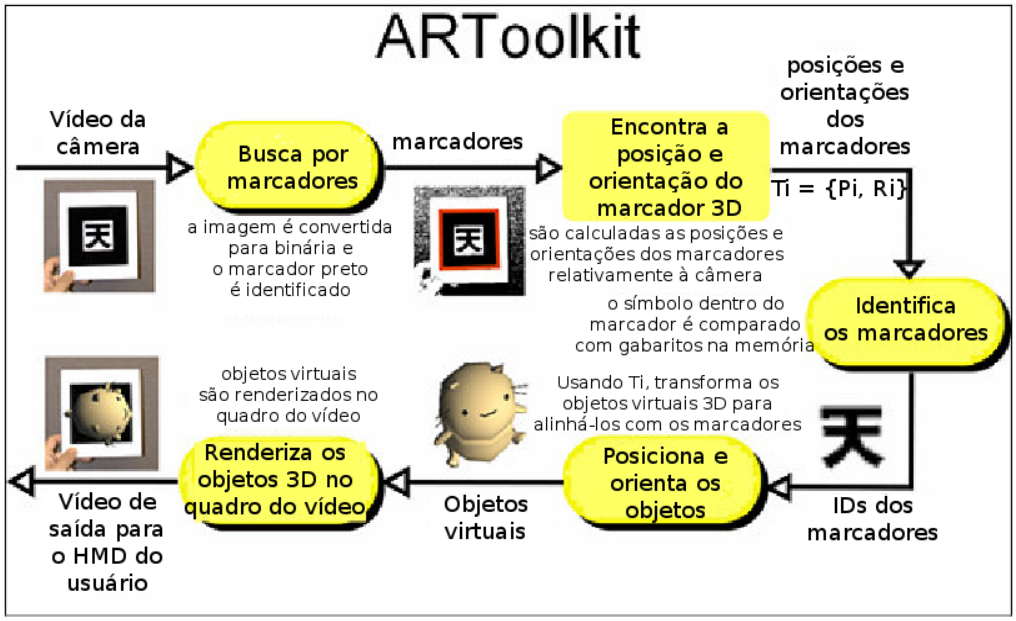



Figura 23: Diagrama descrevendo os passos da detecção dos marcadores e o posicionamento de objetos virtuais sobre os mesmos.

Fonte: Cardoso, 2004.

### 2.4.2 ARTag

O ARTag (ARTag, 2007) foi desenvolvido pelo National Research Council of Canada e inspirado no ARToolKit. Igualmente *open source*, porém com um grande diferencial: a definição do marcador não é feita apenas por uma imagem dentro de uma borda, mas sim por uma matriz 6x6 de quadrados pretos e brancos que representam códigos digitais – formado por zeros e ums. O principal objetivo do ARTag é eliminar alguns dos problemas mais comuns na localização de marcadores fiduciais, conforme (Bastos, 2005):

- falso positivo: acontece quando a presença de um marcador é reportada, mas não existe nenhum no ambiente;
- confusão entre marcadores: acontece quando a presença de um marcador é reportada, mas com o ID errado;
- falso negativo: acontece quando existe um marcador no ambiente, mas sua presença não é reportada.

Figura 24: Exemplo de marcadores utilizados pelo ARTag.

Fonte: ARTag, 2007.

Outra vantagem do ARTag é que, devido a conter um código com redundância em sua imagem, parte do marcador pode ficar oculto e mesmo assim o sistema reconhece o marcador. Isso deve-se ao código ser construído efetivamente de 10 bits, porém como já citado anteriormente, a imagem contém 36 bits, sendo que os 26 bits extra são bits de redundância para reduzir as chances de detecção e identificação falsas e também para prover a rotação correta.

O processo para reconhecimento de um marcador ARTag é semelhante ao do ARToolKit: primeiramente deve ser localizado o contorno da borda do quadrilátero; após isso, a região interna que contém a malha é retirada e são determinados os códigos de zeros e ums contidos. Todo o processamento subseqüente de identificar e verificar o marcador são feitos digitalmente. Quatro

seqüências binárias de 36 bits são obtidas da matriz de códigos binários originários da malha, uma para cada uma das quatro possíveis posições rotações, sendo que dessas apenas uma será validada pelo processo de decodificação (Fiala, 2004).

### 2.4.3 ARStudio

O ARStudio (ARStudio, 2007) também foi desenvolvido pelo National Research Council of Canada e possui basicamente os mesmos princípios do ARTag. O ARStudio, tal como os anteriores, também tem como padrão um quadrado de bordas pretas, mas no seu interior se encontram retângulos brancos igualmente distribuídos, conforme Figura 25 abaixo.

Para o reconhecimento dos marcadores, os quatro cantos de cada retângulo são armazenados em um arquivo de dados que é associado a um padrão. Embora não seja definido o arranjo dos retângulos nos marcadores, é importante que ao gerar os padrões eles sejam unicamente identificáveis com respeito aos padrões já existentes, e também com relação às rotações (Malik, 2002).

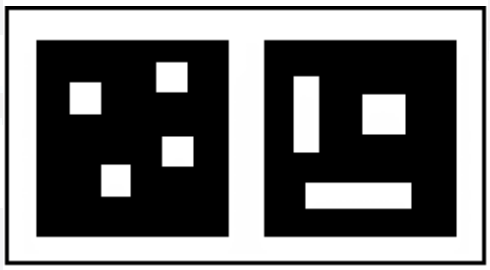


Figura 25: Exemplo de marcadores utilizados pelo ARStudio.

Fonte: Malik, 2002.

### 2.4.4 JARToolKit

O JARToolKit foi desenvolvido pelo C-Lab alemão, e nada mais é do que um invólucro escrito na linguagem Java para o ARToolKit, ou seja, ele consiste em várias camadas que irão prover abstrações para as bibliotecas integradas do

ARToolKit. O JARToolKit acessa as funções do ARToolKit usando a Java Native Interface (JNI) (Geiger, 2004).

### 2.4.5 ARToolKit Plus

O ARToolKitPlus foi desenvolvido na Graz University of Technology, Áustria, dentro do projeto Studierstube, e é uma biblioteca de RA desenvolvida em Software Livre e baseada no ARToolKit, embora o primeiro se preocupe exclusivamente com a questão da detecção de marcadores, não oferecendo funções para captura de vídeo ou renderização de objetos 3D (ARToolKitPlus, 2007).

Seu código fonte conta com várias otimizações, como por exemplo a utilização de aritmética de ponto fixo com o intuito de gerar aplicações eficientes para os dispositivos móveis como, por exemplo, PDAs e smartphones.

O ARToolKitPlus suporta três tipos de detecção de marcadores, uma baseada em *template* (comparando a área do interior com a “base de dados” de marcadores, de forma semelhante ao ARToolKit) e outras duas baseadas em ID's: enquanto a “identificação simples” suporta até 512 marcadores, a identificação mais avançada permite em torno de 4096 marcadores (Wagner, 2007). A identificação baseada em ID's foi criada utilizando-se conceitos do ARTag, já apresentados anteriormente.

O ARToolKitPlus utiliza a técnica de limiar adaptativo, que consiste em perceber alterações de iluminação no ambiente capturado pela câmera, de forma que a detecção dos marcadores não seja prejudicada. Esta biblioteca também oferece suporte à utilização de marcadores com borda de largura variável. Isso permite diminuir a largura da borda para melhor aproveitar o espaço para codificação, fazendo com que marcadores menores possam ser detectados. Outra característica consiste na capacidade de realizar uma compensação sobre imagens de câmeras que geram uma queda radial de luminância. Este fato ocorre em algumas câmeras onde a imagem capturada por elas se apresenta mais escura nos cantos, fazendo com que os marcadores situados nessas regiões não seja detectados (Bastos, 2005).

Figura 26: Visualização do marcador sobreposto com o objeto virtual através de um smartphone.

Fonte: ARToolKitPlus, 2007.

Figura 27: Marcador utilizado.

Fonte: ARToolKitPlus, 2007.

# 3 TECNOLOGIAS UTILIZADAS

## 3.1 Justificativa

Baseado em algumas pesquisas feitas inicialmente, identificamos basicamente duas funcionalidades necessárias para o desenvolvimento de um aplicativo em RA: 1) a detecção e identificação de marcadores para possibilitar a interação e 2) a visualização e sobreposição dos objetos virtuais sobre os marcadores.

No início do projeto, procuramos testar várias bibliotecas para *tracking* de vídeo, entre estes a ARToolKit, o ARToolkitPlus e o ARTag. O ARTag foi estudado superficialmente e obteve-se uma boa impressão do mesmo, porém o mesmo é proprietário, o que nos levou à descartar um estudo mais avançado.

O ARToolKitPlus possui as mesmas funcionalidades do ARToolKit com algumas melhorias (das quais a identificação de marcadores por ID's redundantes é a mais interessante), porém é uma biblioteca sem funções de captura de vídeo, apenas *tracking*. Esta característica, aliada com dificuldades encontradas, que vão desde a documentação (principalmente relacionado com a integração com ferramentas de captura de vídeo, não disponíveis nesta biblioteca) até a utilização de aplicativos desenvolvidos com essa biblioteca, fez com que descartássemos os estudos detalhados da mesma, sendo que foram feitas inúmeras tentativas para obtenção do vídeo, inclusive a combinação com outras ferramentas de captura. Ainda essa característica fez com que os exemplos fornecidos com o código fonte da biblioteca não pudesse ser utilizado com vídeos, como era a intenção do protótipo. Ainda assim, foram localizados alguns projetos que utilizam o

ARToolKitPlus, todos integrantes do pacote Studierstube (Studierstube, 2007), para o qual o ARToolKitPlus foi desenvolvido, porém a compilação dos códigos não foi possível, devido ao excesso de problemas reportados pelo compilador (principalmente devido ao grande número de dependências e bibliotecas necessárias). Ainda com o Studierstube, foi obtida uma versão compilada para o sistema operacional *Windows*, a qual também não funcionou.

Tento em vista os problemas citados e depois de muitas tentativas frustradas com a utilização do ARToolKitPlus, optou-se pela utilização do ARToolKit, biblioteca com código fonte livre e que foi facilmente compilada, a qual possui maior número de publicações e projetos desenvolvidos, os quais pode-se obter maiores informações na seção 3.2.2. Um dos projetos que utiliza o ARToolKit é o osgART (osgART, 2007): uma biblioteca de integração entre o ARToolKit e o OpenSceneGraph (OSG, 2007). Esse projeto supre as necessidades relacionadas a visualização e sobreposição de objetos 3D, visto que o OpenSceneGraph é uma biblioteca que implementa várias funções de um grafo de cena (Silva, 2003).

## 3.2 ARToolKit

O ARToolKit é uma biblioteca de software desenvolvida em linguagem de programação C, e que utiliza técnicas de visão computacional para calcular a posição e orientação da câmera relativas aos marcadores, permitindo que sejam sobrepostos objetos virtuais sobre os marcadores, em tempo real.

### 3.2.1 Histórico

O ARToolKit foi desenvolvido visando permitir o trabalho colaborativo (telepresença) através da Realidade Aumentada. Uma das maiores conquistas da biblioteca é a associação precisa de imagens virtuais com imagens reais, através de técnicas de visão computacional. Foram desenvolvidas algumas técnicas e algoritmos específicos para registro rápido e correto de objetos em tempo real (Kato, 1999).

Segundo (Lima, 2007), entre outras vantagens do ARToolKit, pode-se citar a utilização de câmeras baratas e marcadores simples. Além do rastreamento de marcadores, o ARToolKit também provê funções para acesso à câmera e renderização gráfica. Outra característica importante da biblioteca é o suporte a uma gama de sistemas operacionais, como Windows, Linux, Mac OS e SGI IRIX. Como desvantagem, pode-se salientar que a utilização de marcadores baseados em padrão acarreta muitas vezes em confusão entre marcadores com padrões semelhantes. Outro problema está relacionado à dificuldade em se detectar marcadores que aparecem com tamanho reduzido na imagem, pois a biblioteca não consegue distinguir o padrão presente no interior do marcador. O ARToolKit também é muito sensível à iluminação do ambiente onde a imagem é capturada. Marcadores inseridos em ambientes com pouca ou muita luz não são detectados pela biblioteca, embora seja possível ajustar os parâmetros de detecção para se obter melhores resultados.

### 3.2.2 Aplicações e publicações anteriores

Como já citado anteriormente, o ARToolKit possui uma série de publicações e projetos desenvolvidos com ele, dos quais podemos citar alguns exemplos:

- Criação de aplicações em Realidade Aumentada em dispositivos móveis baseados em Symbian OS (Gomes Neto, 2006): o autor compartilha as experiências no desenvolvimento de uma aplicação em RA para dispositivos móveis;

- Redes Neurais Artificiais Associadas ao ARToolKit para Reconhecimento de Padrões (Camilo, 2006): o autor oferece uma alternativa utilizando redes neurais para o reconhecimento de marcadores;

- Um Framework de Realidade Aumentada para o Desenvolvimento de Aplicações Portáveis para a Plataforma Pocket PC (Lima, 2007): apresenta um framework para o desenvolvimento de aplicações em plataforma Pocket PC, integrando com outras bibliotecas para a renderização de objetos em 3D;

- MagicMouse: um mouse de baixo custo com seis graus de liberdade (Woods, 2003): o autor apresenta o projeto de um mouse com seis graus de liberdade, pretendendo apresentar uma plataforma mais intuitiva para a interação com o usuário;

- Colaboração Móvel com Realidade Aumentada (Filippo, 2005): apresenta conceitos e projetos de RA;

- SignPost (SignPost, 2007): projeto que apresenta um sistema de navegação cujo objetivo é guiar uma pessoa num ambiente interno que não lhe é familiar. Na Figura 28 podemos ver o usuário com o equipamento de navegação, na Figura 29 podemos observar a sinalização gerada pelo sistema e vista pelo usuário e na Figura 30 podemos observar o SignPost para PDA com câmera acoplada (Filippo, 2005);

Figura 29: Usuário com equipamento do SignPost.

Fonte: Fillipo, 2005.

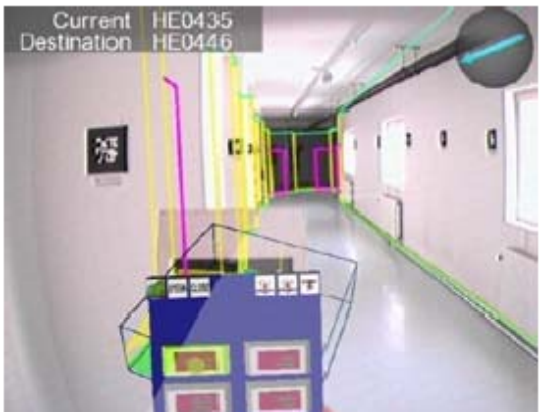



Figura 28: Sinalização vista pelo usuário no sistema SignPost.

Fonte: Fillipo, 2005.

Figura 30: SignPost para PDA.

Fonte: Filippo, 2005.

- Etala – sistema de manutenção remota (Harmo, 2007): sistema para auxiliar a instalação, diagnóstico de problemas e manutenção de equipamentos e sistemas remotos que necessitam do conhecimento de profissionais especializados (Filippo, 2005). Na Figura 31 podemos observar um equipamento a ser reparado, enquanto na Figura 32 podemos observar a imagem aumentada, integrando a imagem real e virtual;

Figura 32: Imagem do equipamento a ser reparado.

Fonte: Harmo, 2007.

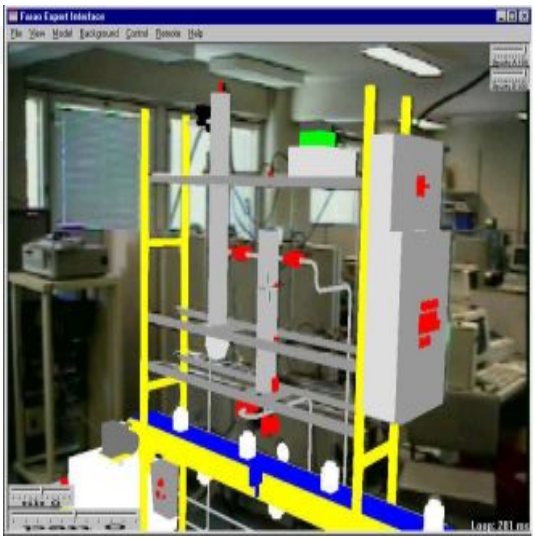


Figura 31: Imagem do equipamento a ser reparado, com o modelo virtual sobreposto.

Fonte: Harmo, 2007.

– Determinando a Posição e Orientação da Mão Através de Imagens de Vídeo (Lopes, 2006): apresenta uma alternativa ao rastreador de posição e orientação utilizado para rastrear a mão. Utiliza o mesmo algoritmo do ARToolKit, porém permitindo que a interação seja com a própria mão, ao invés de marcadores;

– Um sistema de rastreamento óptico baseado em marcadores (Sementille, 2003): apresenta um sistema de rastreamento óptico de movimentos (*motion capture*) utilizando o ARToolKit, com uma abordagem para aumentar a confiabilidade do sistema de captura de movimento através da utilização de um modelo atômico humano simplificado. Na Figura 33, pode-se observar a utilização dos marcadores do ARToolKit para identificar as juntas do modelo atômico;

Figura 33: Marcadores utilizados para captura de movimentos.

Fonte: Sementille, 2003.

– Sistema de Realidade Aumentada para Desenvolvimento Cognitivo da Criança Surda (Dainese, 2003): o autor apresenta uma proposta de utilização de RA para auxiliar no trabalho de interação com portadores de deficiência auditiva;

- Usando Realidade Aumentada no Desenvolvimento de Quebra-cabeças Educacionais (Zorzal, 2006): apresenta o uso da RA no desenvolvimento de quebra-cabeças tridimensionais, enriquecidos, motivadores e de fácil usabilidade.

### 3.2.3 Funcionamento

As principais etapas de funcionamento do ARToolKit são:

- captura do vídeo pela câmera;
- busca do marcador na imagem;
- cálculo de orientação e posição do marcador;
- renderização do objeto virtual na posição localizada.

A Figura 34 exemplifica um quadro capturado, enquanto a Figura 35 exemplifica o mesmo quadro com o resultado do algoritmo de binarização e o objeto virtual sobreposto.


Figura 35: Quadro capturado sem modificações.

Fonte: do autor.

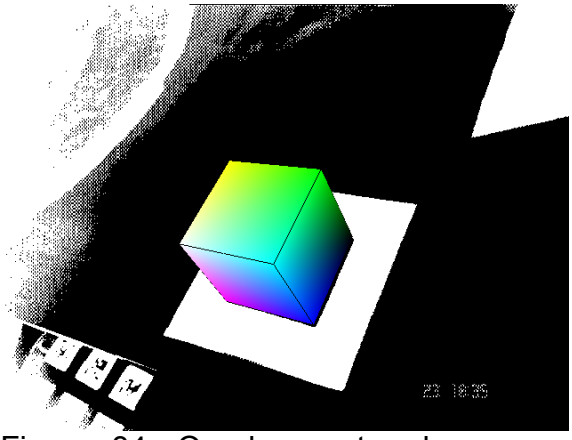


Figura 34: Quadro capturado com o algortimo de binarização e objeto virtual sobreposto.

Fonte: do autor.

Para realizar os passos citados, é aconselhável primeiro efetuar a calibração da câmera a ser utilizada, conforme descrito em seguida.

### 3.2.4 Calibração da câmera

A calibração da câmera é uma etapa importante: é ela que vai definir a relação entre as coordenadas do vídeo capturado (da câmera) e as coordenadas da tela sendo observada. É importante destacar que cada câmera cria uma distorção na imagem capturada, sendo necessário encontrar uma matriz de transformação (rotação e translação) e um fator de distorção, responsável por mapear a imagem adquirida pela câmera na imagem a ser exibida na tela. O ARToolKit utiliza um arquivo de configuração que contêm as informações da câmera, o qual é fornecido originalmente com o ARToolKit e atende à grande maioria das câmeras, porém é sempre recomendável realizar a calibração da câmera.

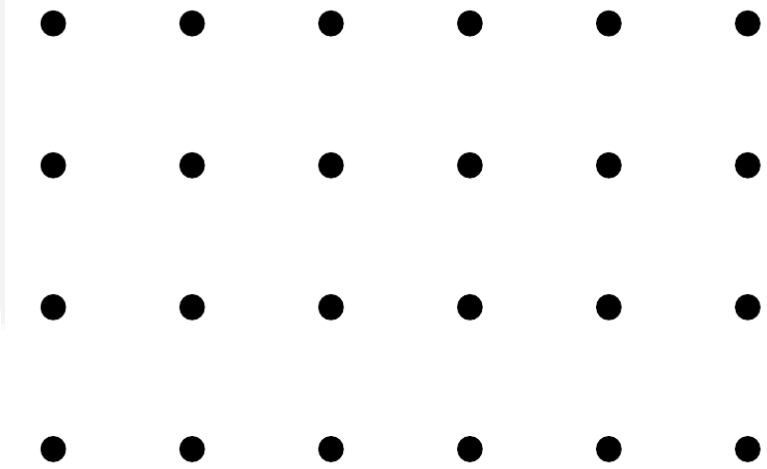


Figura 36: Marcador utilizado no primeiro passo da calibração com o ARToolKit.

Fonte: ARToolKit, 2007.

A principal calibração realizada pelo ARToolKit utiliza o marcador da Figura 36 para obter o centro da imagem e a distorção da lente. O programa *calib_dist* mede o espaçamento entre os pontos e utiliza essa informação para calcular a distorção da lente e posteriormente o centro da câmera. Após obter esses parâmetros, deve-se utilizar o programa *calib_param* para encontrar a distância focal da câmera.

Após o processo de calibração da câmera, já teremos os parâmetros de configuração da câmera, os quais serão salvos em um arquivo (o programa irá solicitar o nome do arquivo ao término do processo). Esses parâmetros, citados abaixo, serão utilizados para cálculo da estimativa e posição e perspectiva dos marcadores:

- *f*: distância focal;
- $S_x$: fator de escala (pixel/mm) do eixo x;
- $S_y$: fator de escala (pixel/mm) do eixo y;
- $x_0$, $y_0$: posição por onde passa o eixo z das coordenadas visualizadas na tela.

Além do processo de calibração citado, pode-se utilizar o programa *calib_camera2*, que é mais rápido porém menos confiável. Esse processo executa a calibração em apenas uma etapa.

### 3.2.5 Detalhamento do algoritmo de calibração

No processo de calibração, o marcador identificado na Figura 36 é utilizado, onde são conhecidos todos os pontos cruzados nas coordenadas 3D. Depois de capturada a imagem, essas coordenadas podem ser detectadas por processamento da imagem. Vários pares de coordenadas 3D do marcador ($X_t$, $Y_t$, $Z_t$) e as coordenadas da câmera na tela ($x_c$, $y_c$) são utilizados para encontrar a matriz de transformação perspectiva *P*.

Podemos então definir a relação entre as coordenadas da tela $(x_c, y_c)$, as coordenadas da câmera $(X_c, Y_c, Z_c)$ e as coordenadas do marcador ($X_t$, $Y_t$, $Z_t$), conforme demonstrado na Equação 9 e ilustrado na Figura 37.

$$\begin{bmatrix} \mathrm{hx}_c \\ \mathrm{hy}_c \\ h \\ 1 \end{bmatrix} = P \cdot \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = P \cdot T_{\mathrm{ct}} \cdot \begin{bmatrix} X_t \\ Y_t \\ Z_t \\ 1 \end{bmatrix} = C \cdot \begin{bmatrix} X_t \\ Y_t \\ Z_t \\ 1 \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} \\ C_{21} & C_{22} & C_{23} & C_{24} \\ C_{31} & C_{32} & C_{33} & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X_t \\ Y_t \\ Z_t \\ 1 \end{bmatrix}$$

$$P = \begin{bmatrix} s_x f & 0 & x_0 & 0 \\ 0 & s_y f & y_0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad T_{ct} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & T_x \\ R_{21} & R_{22} & R_{23} & T_y \\ R_{31} & R_{32} & R_{33} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equação 9

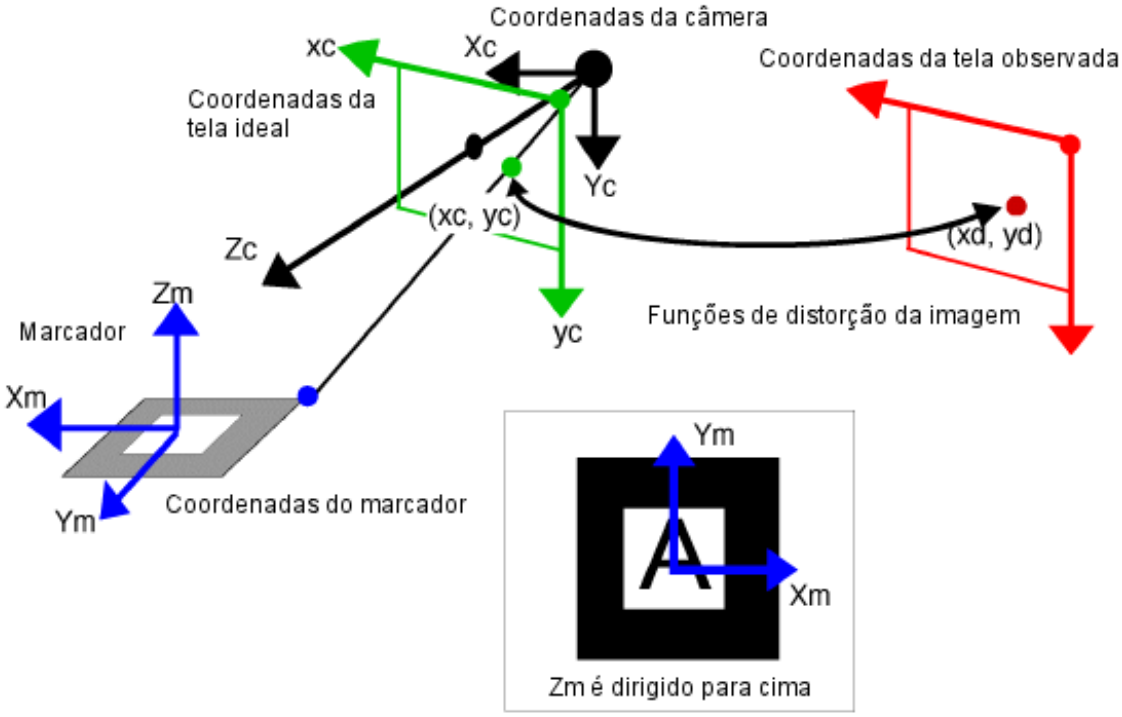



Figura 37: Relação entre sistemas de coordenadas.

Fonte: Kato, 1999.

onde *P* é a matriz de transformação perspectiva que deve ser encontrada aqui, *f* é a distância focal, $s_x$ é o fator de escala no eixo x (pixel/mm), $s_y$ é o fator de escala no eixo y (pixel/mm), $(x_0, y_0)$ é a posição que cruza o eixo z nas coordenadas

da tela, $T_{ct}$ representa as transformações de rotação e translação das coordenadas do marcador para as coordenadas da câmera e *C* é a matriz de transformação obtida na combinação de *P* e $T_{ct}$.

$$\begin{bmatrix} \mathrm{hx}_c \\ \mathrm{hy}_c \\ h \\ 1 \end{bmatrix} = \begin{bmatrix} s_x f & k & x_0 & 0 \\ 0 & s_y f & y_0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} R_{11} & R_{12} & R_{13} & T_x \\ R_{21} & R_{22} & R_{23} & T_y \\ R_{31} & R_{32} & R_{33} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x_t \\ y_t \\ z_t \\ 1 \end{bmatrix}$$

Equação 10

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} V_{11} & V_{12} & V_{13} & W_x \\ V_{21} & V_{22} & V_{23} & W_y \\ V_{31} & V_{32} & V_{33} & W_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_m \\ Y_m \\ Z_m \\ 1 \end{bmatrix} = \begin{bmatrix} V_{3x3} & W_{3x1} \\ 0 \quad 0 & 0 \quad 1 \end{bmatrix} \begin{bmatrix} X_m \\ Y_m \\ Z_m \\ 1 \end{bmatrix} = T_{\mathrm{cm}} \begin{bmatrix} X_m \\ Y_m \\ Z_m \\ 1 \end{bmatrix}$$

Equação 11

Depois de efetuar a limiarização da imagem, as regiões que contêm contorno podem ser cercadas por quatro segmentos de reta extraídas. Os parâmetros para estes quatro segmentos e coordenadas dos quatro vértices das regiões encontradas pelas intersecções dos segmentos de reta são armazenados para processamento posterior.

A Equação 12 representa a transformação de perspectiva utilizada. Todas as variáveis na matriz de transformação são determinadas pela substituição das coordenadas da tela e coordenadas dos quatro vértices do marcador detectado ($X_c$, $Y_c$) e ($X_m$ e $Y_m$) respectivamente. Depois disso, o processo de normalização pode ser feito utilizando esta matriz.

$$\begin{bmatrix} hx_c \\ hy_c \\ h \end{bmatrix} = \begin{bmatrix} N_{11} & N_{12} & N_{13} \\ N_{21} & N_{22} & N_{23} \\ N_{31} & N_{32} & 1 \end{bmatrix} \begin{bmatrix} X_m \\ Y_m \\ 1 \end{bmatrix}$$

Equação 12

Após o processo de normalização, a imagem interna às retas é comparada com padrões já conhecidos, buscando identificar marcadores fornecidos pelo usuário, que podem ser palavras curtas ou até fotos. Essa comparação depende tanto da resolução da imagem interna capturada (a identificação do marcador propriamente dita) quanto da resolução dos padrões já conhecidos.

Quando dois lados paralelos de um marcador quadrado são projetados na imagem, a equação destes segmentos de reta nas coordenadas da câmera é a seguinte:

$$a_1 + b_1 y + c_1 = 0 \, , a_2 + b_2 y + c_2 = 0$$

Equação 13

Para cada marcador, o valor destes parâmetros é obtido no processo de localização das retas. Dado a matriz de projeção de perspectiva *P* que é obtida no processo de calibração na Equação 14, as equações dos planos que incluem esses dois lados respectivamente podem ser representadas nas coordenadas da câmera com a Equação 15, pela substituição de $x_c$ e $y_c$ na Equação 14 por *x* e *y* na Equação 13.

$$\begin{bmatrix} \mathrm{hx}_c \\ \mathrm{hy}_c \\ h \\ 1 \end{bmatrix} = P \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}, \qquad P = \begin{bmatrix} P_{11} & P_{12} & P_{13} & 0 \\ 0 & P_{22} & P_{23} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equação 14

$$a_1 P_{11} X_c + (a_1 P_{12} + b_1 P_{22}) Y_c + (a_1 P_{13} + b_1 P_{23} + c_1) Z_c = 0$$

$$a_2 P_{11} X_c + (a_2 P_{12} + b_2 P_{22}) Y_c + (a_2 P_{13} + b_2 P_{23} + c_2) Z_c = 0$$

Equação 15

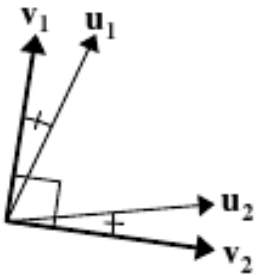



Figura 38: Dois vetores de direção perpendiculares: $v_1$, $v_2$ são calculadas de $u_1$ e $u_2$.

Fonte: Kato, 1999.

Conforme ilustra a Figura 38, dado que o vetor normal destes planos é $n_1$ e $n_2$ respectivamente, o vetor de direção de dois lados paralelos do quadro do marcador é dado pelo produto $n_1 \times n_2$. Dado que dois vetores de direção que são obtidos por dois conjuntos de dois lados paralelos do quadrilátero são $u_1$ e $u_2$, estes vetores devem ser perpendiculares. No entanto, erros de processamento podem fazer que os vetores não sejam exatamente perpendiculares. Para compensar, dois vetores perpendiculares são definidos em $v_1$ e $v_2$, no plano que inclui $u_1$ e $u_2$, como mostra a Figura 38. Dado que o vetor direcional $v_3$ é perpendicular a $v_1$ e $v_2$, o componente de rotação $V_{3x3}$ na matriz de transformação tem das coordenadas dos marcadores para coordenadas da câmera especificado na Equação 11 é $[V_{11}, V_{21}, V_{31}]$.

Sendo que o componente de rotação $V_{3x3}$ na matriz de transformação foi encontrado, pela utilização da Equação 11, Equação 14, as coordenadas dos quatro vértices do marcador nas coordenadas do marcador e também nas coordenadas da câmera, oito equações incluindo o componente de translação $W_x$, $W_y$, $W_z$ são

geradas e o valor deste componente de translação $W_x$, $W_y$, $W_z$ pode ser obtido destas equações.

A matriz de transformação encontrada no método mencionado acima pode incluir erros. Entretanto, o erro pode ser reduzido com o seguinte processo: as coordenadas dos vértices dos marcadores nas coordenadas do marcador podem ser transformadas em coordenadas da câmera usando a matriz de transformação obtida. Então a matriz de transformação é otimizada como a soma das diferenças entre estas coordenadas transformadas e as coordenadas medidas da imagem. Através dessas seis variáveis independentes na matriz de transformação, apenas os componentes de rotação são otimizados e então os componentes de translação são reestimados pela utilização do método citado. Pela interação deste processo um número de vezes, a matriz de transformações fica mais exata. Entretanto, o custo computacional deve ser considerado.

## 3.3 osgART

O osgART é uma biblioteca que simplifica o desenvolvimento de aplicações em RA, combinando a biblioteca ARToolKit, descrita anteriormente, e a biblioteca OpenSceneGraph, que descreveremos em seguida. Basicamente, o osgART provê três funcionalidades principais: integração de alto nível com entrada de vídeo, registro espacial (baseado em marcadores, inclusive com múltiplos marcadores) e registro fotométrico (oclusão, sombra, textura, etc). O principal benefício em utilizar o osgART é obtido com a série de funcionalidades providas pelo OpenSceneGraph diretamente com os aplicativos em RA (OsgART, 2007).

## 3.4 OpenSceneGraph

Grafos de cena são uma das ferramentas utilizadas em tempo real e aplicativos gráficos interativos. O interessante em utilizar grafos de cena é a eficiência para renderização que eles provêm, principalmente devido à otimizações da visualização e ao alto nível de funcionalidades, comportamentos e processamento de dados que implementam (OSG, 2007).

Um grafo de cena representa um mundo 3D como um grafo, normalmente dividido em partes, com a raiz ao topo e *folhas* ao final. As folhas de um grafo de cena geralmente armazenam a geometria e propriedades do material (cores, textura, etc), enquanto que os nós internos armazenam os padrões de renderização e operações de transformação dos objetos.

O OpenSceneGraph é uma ferramenta desenvolvida em linguagem de programação C++ que provê uma série de classes, funções, tipos de dados e características para tornar o desenvolvimento de aplicativos 3D mais fácil e produtivo.

### 3.4.1 Implementação de cena exemplo com o OSG

A estrutura básica de uma implementação no OpenSceneGraph segue:

- criação de um visualizador, com as opções configuradas;
- carregar ou criar um grafo de cena;
- enquanto o programa não finalizar, manipular os eventos e atualizar o grafo de cena e realizar as transformações definidas.

Abaixo, segue um código de exemplo para executar o procedimento citado:

```
#include <osgProducer/Viewer>
#include <osgDB/ReadFile>

int main( int, char **)
{
    // cria o visualizador, com as configurações padrão
    osgProducer::Viewer viewer;
    viewer.setUpViewer( osgProducer::Viewer::STANDARD_SETTINGS );

    // lê o grafo de cena de um arquivo
    osg::Node *scene = osgDB::readNodeFile( "/usr/local/src/osg/cow.osg" );
    if ( !scene ) return 1;

    // indica o grafo de cena lido para o visualizador
    viewer.setSceneData( scene );

    // cria a janela e executa as threads necessárias
    viewer.realize();

    // laço principal
    while( !viewer.done() )
    {
        // sincroniza a thread principal com as restantes
        viewer.sync();

        // manipula os eventos com mouse e teclado
        viewer.update();

        // efetua as transformações necessárias na figura sendo
        // visualizada e finaliza com um swap buffers
        viewer.frame();
    }
}
```

A configuração padrão de um grafo de cena, citado anteriormente, possui o mouse e teclado como manipulador, tendo as seguintes funcionalidades:

- botão esquerdo do mouse: rotação;
- botão direito do mouse: escalamento;
- tecla “ “ (espaço): reinicia as configurações da câmera;
- tecla s: alterna entre os níveis de estatística de performance;
- tecla w: alterna entre visão wireframe, pontos e polígonos preenchidos;
- tecla t: habilida ou desabilita a textura;
- tecla h: imprime texto de ajuda;
- tecla f: alterna entre tela cheia e modo em janela.

O exemplo citado acima é o mais simples possível: basicamente cria a visualização, com um objeto a ser visualizado, e permite a interação através de eventos de teclado ou mouse. Aperfeiçoando um pouco o exemplo citado, podemos definir algumas características do objeto, como textura, tamanho e posicionamento.

```
// lê o arquivo com textura
osg::Image* image = osgDB::readImageFile("/usr/local/src/osg/textura.png");
osg::Texture2D *texture = new osg::Texture2D;
texture->setImage(image);

// seta o tipo do mapa de reflexo como esférico
osg::TexGen* texgen = new osg::TexGen;
texgen->setMode(osg::TexGen::SPHERE_MAP);

osg::StateSet* stateset = new osg::StateSet;
stateset->setTextureAttributeAndModes(0,texture,osg::StateAttribute::ON);
stateset->setTextureAttributeAndModes(0,texgen ,osg::StateAttribute::ON);

// adiciona um mapa de reflexo ao objeto
objeto2->setStateSet(stateset);
```

E em seguida, podemos observar o código necessário para realizar algumas transformações no objeto:

```
// cria objetos para configurar transformações
osg::Matrix modelScale;
osg::Matrix modelTranslate;
osg::Matrix modelRot;

// cria matriz de transformações
osg::MatrixTransform*   unitTransform = new osg::MatrixTransform;

// seta objeto para escala
modelScale.makeScale(0.7,0.7,0.7);
// seta objeto para rotação
modelRot.makeRotate(90.,osg::Vec3f(0,0,1));
//seta objeto para translação
modelTranslate.makeTranslate(osg::Vec3f(10,10,10));

// seta matriz de transformações
unitTransform->postMult(modelTranslate);
unitTransform->postMult(modelRot);
unitTransform->postMult(modelScale);

// adiciona o objeto a lista de nodos a sofrerem transformação
unitTransform->addChild(objeto2);
```

O código-fonte completo deste exemplo pode ser verificado no Anexo 1.

### 3.4.2 Implementação com o osgART

Na implementação com o osgART é muito semelhante à implementação com o OpenSceneGraph: basicamente o osgART terá uma matriz de transformação que será obtida junto ao marcador, conforme configuração setada inicialmente. Em seguida, podemos verificar as principais funções de um programa com o osgART.

```
osgARTInit(&argc, argv);
```

Inicializa o osgART.

```
osg::ref_ptr<osgART::GenericVideo> video =
osgART::VideoManager::createVideoFromPlugin("osgart_artoolkit");
```

Carrega o plugin do osgART para integração com o ARToolKit.

```
osg::ref_ptr<osgART::GenericTracker> tracker =
osgART::TrackerManager::createTrackerFromPlugin("osgart_artoolkit_tracker")
;
```

Carrega o plugin do osgART pata integração com o tracker do ARToolKit.

```
osg::ref_ptr<osgART::Marker> marker = tracker->getMarker(0);
```

Obtém o marcador de identificação '0'.

```
osg::ref_ptr<osg::MatrixTransform> markerTrans = new
osgART::ARTTransform(marker.get());
```

Cria uma matriz de transformação obtida do marcador.

```
tracker->setImage(video.get());
```

Seta imagem para o *tracker*, conforme imagem obtida do vídeo.

Estas são algumas das chamadas de funções utilizadas. O código-fonte completo do exemplo pode ser encontrado no Anexo 2.

# 4 IMPLEMENTAÇÃO E AVALIAÇÃO DO PROTÓTIPO

O protótipo desenvolvido visa implementar um jogo de *pong* em RA. O jogo visa avaliar a possibilidade e dificuldade do desenvolvimento de um aplicativo utilizando RA.

## 4.1 Detalhamento do projeto

Para o desenvolvimento do protótipo, foi utilizada a biblioteca osgART, já descrita anteriormente. Basicamente, a raquete é um retângulo com uma textura aplicada. A bola é uma esfera com cores aplicadas. O movimento da raquete é controlado pelo marcador, enquanto o movimento da bola é em linha reta: inicialmente move-se em direção ao topo e à direita; ao chegar ao topo da tela, move-se em direção à base da mesma e ao chegar à base, move-se novamente em direção ao topo. Ao chegar no espaço máximo à direita, deve mover-se à esquerda, e vice-versa.

A Figura 39 exemplifica o protótipo desenvolvido.

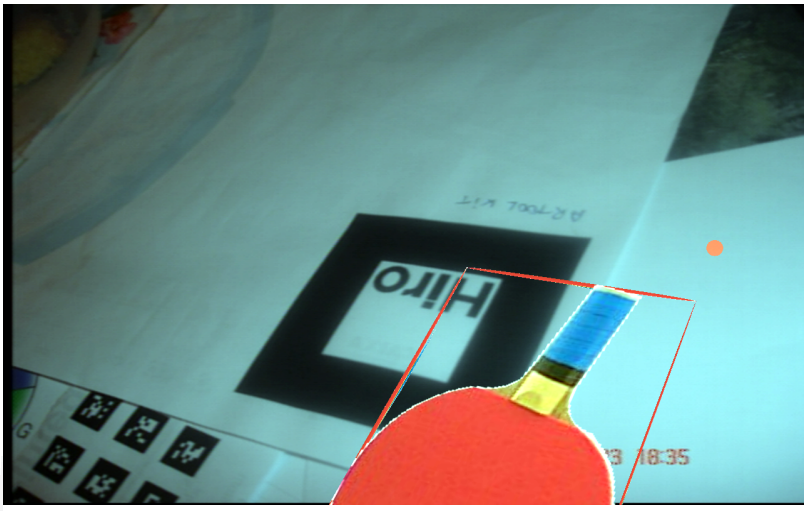



Figura 39: Protótipo desenvolvido, com a raquete abaixo e a bolinha ao centro, no lado direito.

Fonte: do autor.

Em seguida, iremos comentar os principais trechos do código-fonte do protótipo, iniciando pela criação da raquete: é criada uma caixa, com tamanho inicial de 10 x 18 x 0.5:

```
    //cria a caixa que vai conter a raquete, na posição 0,0,-10, com
tamanho 10, 18, 0.5.
    osg::Box *raquete = new osg::Box   (osg::Vec3( 0, 0, -10 ), 10, 18,
0.5f);
    osg::ShapeDrawable* sd  = new osg::ShapeDrawable(raquete);
```

O código abaixo aplica textura à caixa:

```
    //carrega a imagem de textura da raquete
    osg::Image* img =
osgDB::readImageFile("/home/njunior/osg/raquete.png");

    //configura a textura
    osg::TextureRectangle* texture = new osg::TextureRectangle(img);
    osg::TexMat* texmat = new osg::TexMat;
    texmat->setScaleByTextureRectangleSize(true);

    //configura o estado da textura
    osg::StateSet* state = sd->getOrCreateStateSet();
    state->setTextureAttributeAndModes(0, texture,
osg::StateAttribute::ON);
    state->setTextureAttributeAndModes(0, texmat, osg::StateAttribute::ON);
    state->setMode(GL_BLEND,osg::StateAttribute::ON);
```

O código abaixo, cria um grupo para conter a raquete, aplica uma transformação de escala e logo em seguida adiciona a raquete à matriz de transformação do marcador:

```
    //cria o grupo para conter a raquete.
    osg::Geode* geode = new osg::Geode();

    //adiciona a raquete ao grupo.
    geode->addDrawable(sd);

    //cria matriz de transformação para setar escala da raquete.
    osg::MatrixTransform* unitTransform = new osg::MatrixTransform;
    osg::Matrix modelScale;
    modelScale.makeScale(10,10,10);
    unitTransform->postMult(modelScale);

    //adiciona o grupo da raquete à matriz de transformação.
    unitTransform->addChild(geode);
    markerTrans->addChild(unitTransform);

    //cria objeto para conter a cena.
    osg::Group* sceneGroup = new osg::Group();
    sceneGroup->getOrCreateStateSet()->setRenderBinDetails(5, "RenderBin");

    //adiciona a matriz de transformação do marcador à cena.
    sceneGroup->addChild(markerTrans.get());
```

No código abaixo, é criado um objeto para conter a bolinha, aplicada a cor à este e adicionado ao grupo:

```
    //cria objeto da bolinha.
    osg::ShapeDrawable* bola = new osg::ShapeDrawable(new
osg::Sphere(osg::Vec3( 0, 0, 0), 2));
    //seta cor do objeto
    bola->setColor(osg::Vec4(1, 0, 0, 0));

    //cria grupo que vai conter a bolinha
    osg::Geode* geode2 = new osg::Geode();
    geode2->addDrawable(bola);

    //cria matriz de transformação para a bola e adiciona o grupo da mesma
à matriz.
    osg::MatrixTransform* ballXform = new osg::MatrixTransform();
    ballXform->addChild(geode2);

    //cria grupo para conter a cena da bolinha e do vídeo e adiciona-os.
    osg::Group* sceneGroup2 = new osg::Group();
    sceneGroup2->addChild(foregroundGroup);
    sceneGroup2->addChild(ballXform);
```

Em seguida, as variáveis que controlam a bolinha. A variável direcaoX indica a direção em X que a bolinha deve mover-se: se for 1, deve mover-se à direita. Se for 0, deve mover-se à esquerda. Analogamente, a variável direcaoY: se for 1, deve mover-se para cima. Se for 0, deve mover-se para baixo. No mesmo bloco de código, é criada e inicializada a variável localizou, que vai identificar se o marcador foi ou não localizado (na primeira vez que é localizado é setada a posição inicial da bolinha). Em seguida, são inicializadas as variáveis quadros e invalidos, que identifica, respectivamente, o número de quadros obtidos e o número de quadros onde não foi localizado o marcador.

```
    //inicializa variáveis para controle da bolinha
    int direcaoX = 0, direcaoY = 0;

    //variável para controlar se localizou o marcador.
    int localizou=0;

    //inicializa variáveis para controlar o número de quadros e quadros
inválidos.
    long invalidos=0, quadros=0;
```

O código abaixo testa se é a primeira vez que o marcador foi localizado e, caso afirmativo, seta o posicionamento da bolinha:

```
        if ( localizou==0 && marker->isValid( ) )
        //caso o marcador for válido e for a primeira vez que localiza.
        {
            //obtém posicionamento do marcador.
            osg::Vec3f pos2=matrix.getTrans();

            //seta a matriz de transformação obtida para a bolinha, para
setar posicionamento inicial.
            ballXform->setMatrix(matrix);

            //cria matriz de translação e aplica uma transformação.
            osg::Matrix mTransl;
            mTransl.makeTranslate(osg::Vec3f(0,0,10.f));
            ballXform->postMult(mTransl);

            localizou = 1;
        }
```

No código a seguir, caso o marcador foi localizado, são aplicadas algumas transformações à bolinha, que irá definir a movimentação da mesma:

```
            //seta translação (movimentação da bolinha).
            //Caso a variável direcaoY for 1, indica que é pra mover-se
para baixo. Caso for 0, deve mover-se para cima.
            //Caso a variável direcaoX for 1, indica que é pra mover-se
para a esquerda. Caso for 0, deve mover-se para a direita.
            matrixBall.makeTranslate( translation+osg::Vec3f(direcaoX ?
-1.f : 1.f, direcaoY ? -1.f : 1.f, 0.f) );

            //obtém a posição da bola (pos1) e do marcador (pos2)
            osg::Vec3f pos1=matrixBall.getTrans( );
            osg::Vec3f pos2=matrix.getTrans();
            osg::Vec3f scal=unitTransform->getMatrix( ).getScale( );
            osg::Vec3f tams=raquete->getHalfLengths( );

            //obtém matriz com escala do marcador.
            osg::Vec3f prod=matrix.getScale( );

            //distância entre o marcador e a bolinha.
            float distance_markers=(pos1-pos2).length();

            /*
            verifica a distância entre marcador e bolinha (caso a distância
em y for menor que a altura da raquete, então considera que colidiu em y.
Caso a distância em x for menor que a largura da raquete, considera que
colidiu em x. Caso colidiu em ambos os planos, troca a movimentação da
bolinha.
            */
            if ( direcaoY && abs((int)(pos1.y( )-pos2.y( )))<=18 &&
abs((int)(pos1.x( )-pos2.x( )))<=10 )
            {
                //Caso colidiu em ambos os planos
                direcaoY = direcaoY ? 0 : 1;
                direcaoX = direcaoX ? 0 : 1;
            }
            else
            {
                //caso a posição bolinha em y for maior que 75, considera
que chegou ao topo e inverte a movimentação neste sentido.
                if ( matrixBall.getTrans( ).y( ) > 75 )
                    direcaoY = 1;

                //caso a posição bolinha em x for maior que 125, considera
que chegou ao limite direito e inverte a movimentação neste sentido.
                if ( matrixBall.getTrans( ).x( ) > 125 )
                    direcaoX = 1;

                //caso a posição bolinha em y for menor que -75, considera
que chegou à borda da tela e inverte a movimentação neste sentido.
                if ( matrixBall.getTrans( ).y( ) < -75 )
                    direcaoY = 0;

                //caso a posição bolinha em x for menor que -125, considera
que chegou ao limite esquerdo e inverte a movimentação neste sentido.
                if ( matrixBall.getTrans( ).x( ) < -125 )
                    direcaoX = 0;
            }


            //seta matriz de transformação da bolinha.
            ballXform->setMatrix(matrixBall);
        }
```

Basicamente, foram comentados os principais trechos do código-fonte do protótipo. O código completo pode ser obtido no Anexo 3.

## 4.2 Avaliação do protótipo

Apesar do protótipo não procurar utilizar grande realismo visual, apenas empregando efeitos de movimentação simples, pode-se afirmar que a aplicação em RA é viável e funcional. No tópico a seguir, detalhamos os testes realizados com o protótipo, de forma a identificar limitações da tecnologia empregada.

## 4.3 Testes efetuados

Os testes foram feitos em um ambiente com apenas uma fonte de luz, conforme Figura 40:

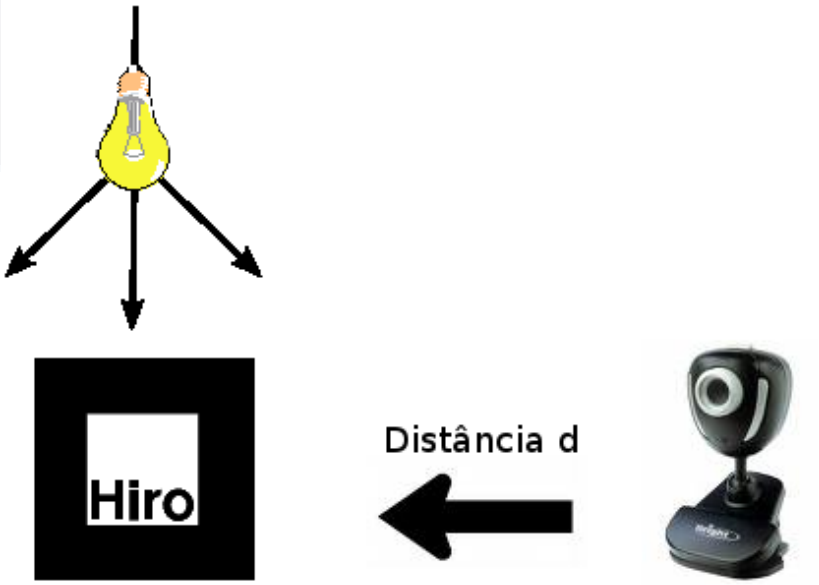



Figura 40: Disposição da fonte de luz, marcador e câmera nos testes.

Fonte: do autor.

Para os testes foi utilizado um ambiente onde a única fonte de luz é uma lâmpada, que foi alternada entre 100W e 150W. Além disso, foram utilizados dois

tamanhos de marcadores: um pequeno, de 6cm, e outro grande, de 12cm. Outro fator testado foi a distância do marcador em relação à câmera: foram feitos testes em três distâncias: 40cm, 80cm e 120cm. Ainda como configuração para os testes, utilizou-se o arquivo padrão de calibração da câmera e um arquivo calibrado para a câmera utilizada (uma câmera digital Sony Cybershot P200 – 7.2Mp e resolução de 640x480), conforme procedimento já citado.

### 4.3.1 Calibração da câmera

As Figuras 41 e 42 mostram duas imagens obtidas no processo de calibração. Quanto mais imagens forem coletadas, maior será a precisão dos parâmetros obtidos. Ao obter uma imagem, deve-se identificar cada um dos pontos na tela. É importante que as imagens obtidas estejam com a câmera em posições variadas, em relação ao padrão utilizado.

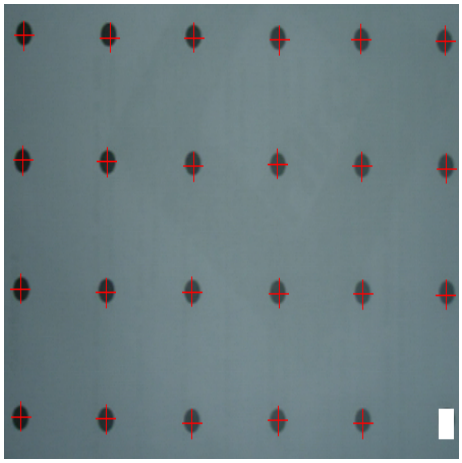


Figura 41: Primeira imagem utilizada no primeiro processo de calibração.

Fonte: do autor.

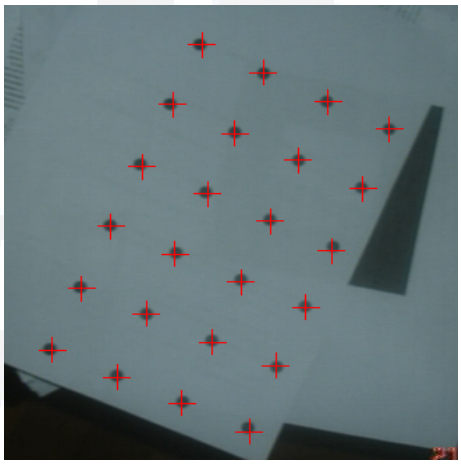


Figura 42: Quinta e última imagem utilizada no primeiro processo de calibração.

Fonte: do autor.

Como resultado deste primeiro processo, o programa de calibração exibe as imagens utilizadas com retas cruzando os pontos. Caso essas retas cruzarem os pontos corretamente, a calibração foi bem sucedida. Caso contrário, é necessário refazer este primeiro passo. As Figuras 43 e 44 abaixo contêm as retas:

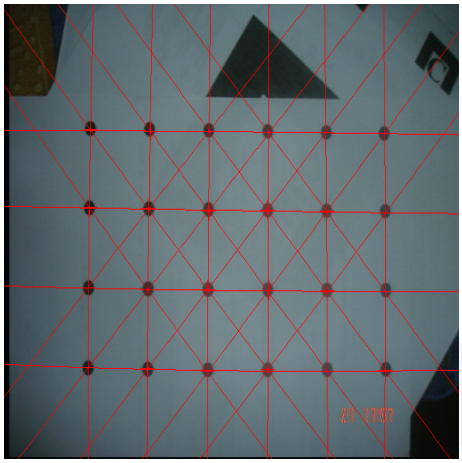


Figura 43: Retas cruzando os pontos da primeira imagem.

Fonte: do autor.

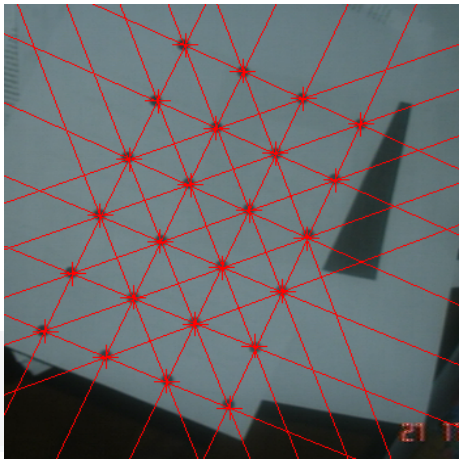


Figura 44: Retas cruzando os pontos da quinta e última imagem.

Fonte: do autor.

Como resultado deste processo de calibração, obtivemos o centro da distorção nas coordenadas x e y (375 e 185 respectivamente), o fator de distorção (16,8) e o fator de ajuste de tamanho (1,005851).

Após obter esses parâmetros, é necessário utilizar o programa *calib_cparam* para encontrar a distância focal da câmera. Este programa utiliza a Figura 45 para obtenção dos parâmetros. Este programa vai requisitar algumas informações ao início de sua execução: o centro da distorção, o fator de distorção e o fator de ajuste de tamanho (todos obtidos no primeiro passo da calibração), o número de linhas horizontal (número de linhas horizontal contido no marcador de calibração, demonstrado na Figura 45), o número de linhas vertical (número de linhas vertical contido no marcador de calibração, demonstrado na Figura 45), o número de iterações (número de imagens a ser coletado), a distância entre as linhas (entre as linhas do marcador utilizado para calibração, demonstrado na Figura 45), a distância a ser movido o marcador (é essa distância que deverá ser aumentada entre a câmera e o marcador entre cada imagem coletada – na calibração demonstrada foi utilizada a distância de 40 mm) e o tamanho das imagens coletadas.

Depois de digitados os parâmetros, deve-se iniciar o processo de calibração. Inicialmente, o marcador e a câmera deverão estar a uma distância *d*,

perpendicularmente dispostos. Deve-se então obter uma imagem e identificar as linhas horizontais (de cima para baixo) e depois as linhas verticais (da esquerda para a direita). A ordem de identificação das linhas é importante nesse processo. Depois de localizadas as linhas em uma imagem, deve-se aumentar a distância *d* conforme especificado no parâmetro na inicialização do segundo passo da calibração. As Figuras 46 e 47 demonstram esse procedimento.

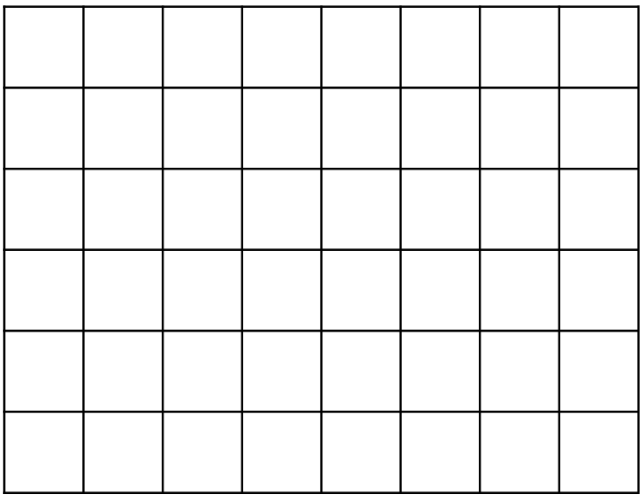


Figura 45: Imagem utilizada no segundo passo da calibração.

Fonte: ARToolKit, 2007.

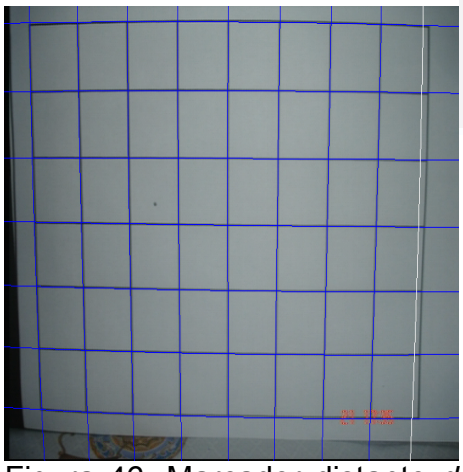


Figura 46: Marcador distante *d* mm da câmera para calibração.

Fonte: do autor.

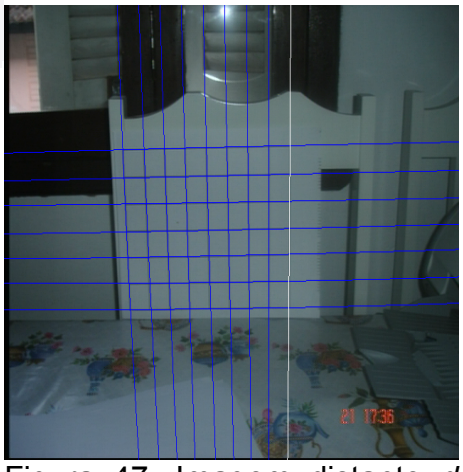


Figura 47: Imagem distante *d* mm + 160 mm da câmera para calibração.

Fonte: do autor.

### 4.3.2 Comparativos

Para comparativo entre os testes, foi utilizado o mesmo protótipo do jogo, porém com uma modificação: limitando a execução do programa em 1000 quadros (quando chegar nos 1000 quadros sai do programa) e contando o número de quadros com marcadores não-reconhecidos. Durante o tempo em que o programa executava, o marcador era movimentado, porém sempre ficando perpendicular à câmera. Em todos os testes, foram feitos os mesmos padrões de movimentos. Para cada configuração citada acima, foram feitas três leituras (execuções do programa) e nas tabelas abaixo, segue a média dessas leituras:

Tabela 1: Dados obtidos nos testes.

| Calibração | Lâmpada (W) | Distância (cm) | Erros por tamanho do marcador | |
|---|---|---|---|---|
| | | | Pequeno | Grande |
| Para a câmera | 100 | 40 | 149,67 | 257,67 |
| | | 80 | 27,33 | 54,67 |
| | | 120 | 164 | 81,67 |
| | 150 | 40 | 9,33 | 74,33 |
| | | 80 | 14 | 67,67 |
| | | 120 | 20,33 | 121 |
| Padrão | 100 | 40 | 264,67 | 168,33 |
| | | 80 | 22,67 | 140,33 |
| | | 120 | 39,33 | 48 |
| | 150 | 40 | 11,33 | 133 |
| | | 80 | 6,33 | 50,67 |
| | | 120 | 20 | 25,67 |
| Média de erros por tamanho | | | 62,42 | 101,92 |

Fonte: do autor.

Tabela 2: Média de erros por iluminação.

| Lâmpada(W) | Média de erros |
|---|---|
| 100 | 74,82 |
| 150 | 42,06 |

Fonte: do autor.

Nos dados obtidos, há algumas situações muito problemáticas, porém já conhecidas, como o caso do marcador grande muito próximo: devido aos problemas com limiarização única, o marcador grande próximo da câmera acaba tendo o reconhecimento prejudicado. Com base nessas situações, vamos analisar apenas as melhores situações obtidas (considerando apenas as situações em que ocorreram menos de 150 erros).

Tabela 3: Média de erros por calibração.

| Calibração | Média de erros |
|---|---|
| Padrão | 59,79 |
| Para a câmera | 56,17 |

Fonte: do autor.

Tabela 4: Média de erros por distância.

| Distância (cm) | Média de erros |
|---|---|
| 40 | 73,19 |
| 80 | 157,79 |
| 120 | 55,94 |

Fonte: do autor.

Com base nos testes realizados, pode-se perceber que o marcador de tamanho pequeno teve menor quantidade de erros de localização, resultando em um melhor desempenho. Outro ponto que pode-se verificar nestes dados é relacionado à iluminação: na configuração com maior luminosidade a quantidade de

erros foi menor. Quanto à calibração, pode-se concluir, pela proximidade dos valores, que a calibração padrão atende perfeitamente o modelo de câmera utilizado, e que a calibração não se faria necessária.

Quanto à distância, não podemos tirar nenhuma conclusão analisando apenas as médias de erros por distância. Para facilitar a conclusão utilizando a distância, vamos analisar o Gráfico 1, que demonstra que a influência da distância vai depender também do tamanho do marcador: se o marcador grande for utilizado, a distância muito próxima causa muitos erros. Caso o marcador grande for utilizado a uma distância maior, os erros diminuem.

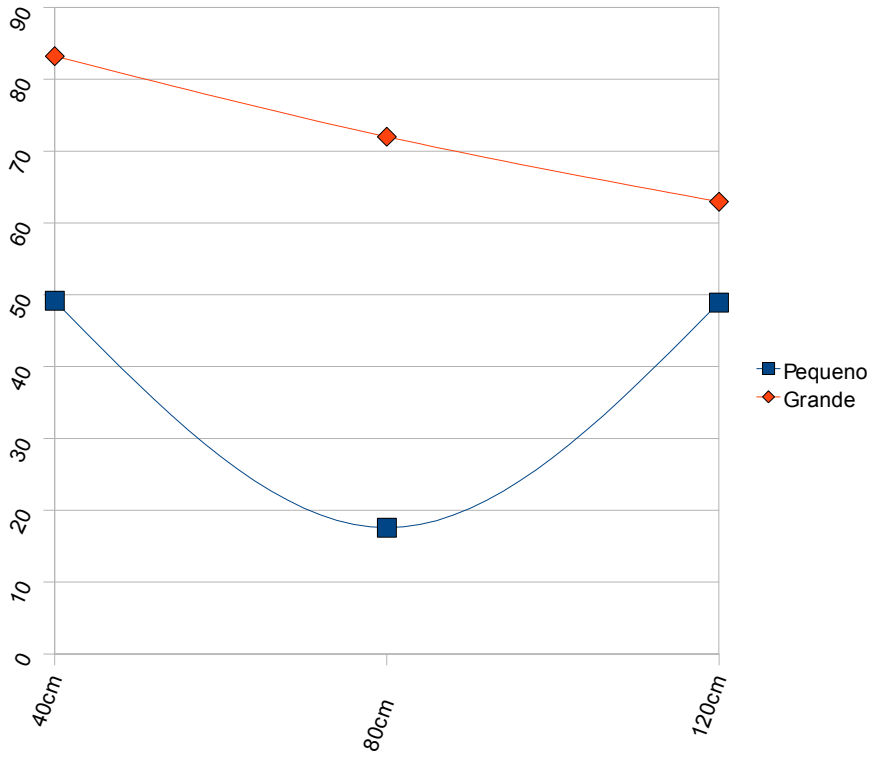



Gráfico 1: Ocorrência de erros por distância e tamanho dos marcadores.

Fonte: do autor.

## 4.4 Problemas detectados

O grande problema detectado é com relação à mudança de luminosidade, que faz com que o *tracker* não localize o marcador, perdendo o movimento realizado pelo jogador. O problema da luminosidade pode acontecer devido à mudanças na reflexão do material utilizado, à movimentação do mesmo ou à fonte de luz do ambiente. O principal fator deste problema é o tipo de limiarização utilizado: o ARToolKit utiliza limiarização única. Caso utilizasse limiarização adaptativa, como o ARToolKitPlus (Lima, 2007), possivelmente não seria tão sensível às mudanças de luminosidade.

Outro problema que afeta bastante a precisão na localização e identificação dos marcadores é quanto à oclusão: devido à biblioteca exigir que todo o marcador fique visível, ao ocultar uma parte do mesmo a biblioteca já não o localiza. No mesmo sentido, ocorre problemas com relação à inclinação do marcador: dependendo do ângulo de visão, o mesmo não é detectado.

## 4.5 Dificuldades encontradas

A principal dificuldade encontrada no desenvolvimento do aplicativo é relacionado à documentação: apesar de todas as bibliotecas utilizadas serem Software Livre, encontramos pouca documentação disponível, muitas das quais desatualizadas. Muitas vezes, o desenvolvimento foi feito utilizando como documentação o próprio código fonte das bibliotecas, na maioria utilizando os cabeçalhos com definição das funções para localização de funções adequadas à cada momento.

Esse problema extende-se à basicamente todas as bibliotecas utilizadas, com exceção do OpenSceneGraph, que possui uma base maior de documentação (principalmente com códigos de exemplo) e a mesma mais atualizada, apesar de bastante complexo, como também conclui (Silva, 2003). A biblioteca exige conhecimentos avançados em orientação a objetos, além de conhecimento de sua estrutura como um todo, o que torna o processo de aprendizagem da mesma um pouco lento, prejudicando o andamento do trabalho.

Outra dificuldade encontrada foi em relação às câmeras de vídeo e compatibilidade com o sistema operacional utilizado – Linux. Foram feitas tentativas de utilização de vários modelos e marcas de câmeras de vídeo, das quais apenas dois modelos funcionaram com êxito: a câmera utilizada (Sony CyberShot P200) e um modelo da Genius (esta funcionou no Linux, porém o código de cores utilizado pela câmera não é suportado pelo ARToolKit). Foram feitas inúmeras tentativas com câmeras de outras marcas e modelos, porém sem sucesso.

## 4.6 Análise dos resultados

Com base nos testes realizados, pode-se chegar a uma situação favorável na utilização do aplicativo: em um ambiente com iluminação controlada, sem grandes variações na luminosidade, podemos identificar que a melhor situação é não estar muito próximo, mas também não muito distante da câmera durante a execução do aplicativo, com risco de perder a interatividade e controle do aplicativo.

# 5 CONSIDERAÇÕES FINAIS

## 5.1 Principais contribuições

A Realidade Aumentada é uma área ainda muito recente, com várias questões que ainda devem ser trabalhadas. No que diz respeito à precisão de registro, os testes efetuados em ambientes internos produziram resultados satisfatórios, identificando a viabilidade de aplicativos desenvolvidos com essa tecnologia, inclusive com Software Livre.

Durante o trabalho, foram estudadas algumas bibliotecas e selecionada a biblioteca ARToolKit para maiores detalhamentos e desenvolvimento de um protótipo, disponível nos Apêndices deste trabalho. O maior resultado deste estudo foi comprovar que é viável o desenvolvimento de aplicativos em Realidade Aumentada, documentando os principais passos para obter sucesso com o desenvolvimento, além de encontrar os principais problemas da ferramenta utilizada. Ainda relacionado à documentação, identifica as principais classes, métodos e estruturas do OpenSceneGraph para a implementação e visualização de interfaces em 3D.

## 5.2 Trabalhos futuros

Como continuação deste projeto, poderíamos citar algumas funcionalidades a serem implementadas ou melhoradas nas bibliotecas utilizadas:

- no ARToolKit, poderíamos implementar a limiarização adaptativa, melhorando o desempenho da biblioteca sobre variações de iluminação;

- ainda no ARToolKit, poderia-se desenvolver um algoritmo para localização e identificação dos marcadores por mais de uma câmera, para evitar problemas já citados neste trabalho;

- outra sugestão para o ARToolKit é implementar um marcador semelhante ao do ARTag ou ARToolKitPlus: possuindo uma identificação com redundância, pra auxiliar no combate ao problema da oclusão e até mesmo da inclinação do marcador;

- no osgART, poderíamos desenvolver, em código aberto, uma interface para integração com o ARToolkitPlus, biblioteca citada neste trabalho;

- relacionado ao protótipo aqui desenvolvido, poderia-se melhorar o realismo do protótipo criado, criando alguns efeitos visuais e maior detalhamento e dinâmica na movimentação tanto na bolinha quanto da raquete.

# BIBLIOGRAFIA

ANTUNES, Eurico J. **Processamento de imagens**: uma abordagem interdisciplinar aplicada a correção de prognósticos meteorológicos. Pelotas: 1999.

ARTag. Disponível em: http://www.cv.iit.nrc.ca/research/ar/arstudio.html. Acesso em Nov/2007.

ARToolKit. Disponível em: http://www.hitl.washington.edu/artoolkit/. Acesso em Dez/2007.

ARToolKitPlus. Disponível: Studierstube Augmented Reality Project site. URL: http://studierstube.icg.tu-graz.ac.at/handheld_ar/artoolkitplus.php. Acesso em Jun/2007.

AUMONT, Jacques. **A imagem**. Campinas, Papirus, 1993.

AZEVEDO, Eduardo. Aura Conci. **Computação Gráfica** – Teoria e Prática. Rio de Janeiro, Elsevier, 2003.

BASTOS, Nacha Costa. **Arquitetura para Dispositivos Não-convencionais de Interação Utilizando Realidade Aumentada**: Um Estudo de Caso. Pernambuco, 2005.

BORGES, Thiago Silva. **Um protótipo de editor de imagens via WEB**. 2005. Relatório de Estágio. Centro Universitário Luterano de Palmas.

BRAGA, Isis Fernandes. **Realidade Aumentada em Museus**: As batalhas do Museu Nacional de Belas Artes, RJ. Rio de Janeiro, 2007.

CAMILO, Celso. Gomes, Wneiton L. Cardoso, Alexandre *et al*. **Redes Neurais Artificiais Associadas ao ARToolKit para Reconhecimento de Padrões**. III Workshop de Realidade Aumentada, 2006 (p. 12-15).

CANTAG. University of Cambridge. Cantag – DTG Research. Disponível em: http://www.cl.cam.ac.uk/research/dtg/research/wiki/Cantag. Acesso em Jun/2007.

CASACURTA, Alexandre. **Introdução à Computação Gráfica**. São Leopoldo, 1998. Disponível em: http://www.inf.unisinos.br/~osorio/CG-Doc/CG-Web/cg.html . Acesso em Jun/2007.

CARDOSO, A.; Lamounier Jr, E. editores. **Realidade Virtual**: Uma Abordagem Prática. Livro dos Minicursos do SVR2004, SBC, São Paulo, 2004.

CUNHA, Gerson. **Augmented Reality:** Realidade Aumentada e Visão computacional. In - http://www.lamce.ufrj.br/grva/data/realidade_aumentada/index.php. Acesso em Jun/2007

DAINESE, Carlos Alberto. Garbin, Tânia Rossi. Kirner, Cláudio. **Sistema de Realidade Aumentada para Desenvolvimento Cognitivo da Criança Surda**. In: SVR 2003 - VI Symposium on Virtual Reality, 2003, Ribeirão Preto. Proceedings of SVR 2003 - VI Symposium on Virtual Reality, 2003. v. 01. p. 273-282.

DUDA, R. O; Hart, P. E. and Stork, D. G. (2001), **Pattern Classification**, Wiley, 2th edition.

ESPINHOSA, Dalila R. S. **InfluÍncia da Injunção da Base na Fototriangulação de Imagens Obtidas com uma Unidade de Mapeamento Móvel**. Dissertação (Pós-graduação). 2006, 112 p. Faculdade de Ciências e Tecnologia, Universidade Estadual Paulista, Presidente Prudente.

FACON, J. **Processamento e Análise de Imagens**. Pontifícia Universidade Católica do Paraná. Curso e Mestrado em Informática Aplicada. 2002.

FIALA, M. **ARTag, an Improvement Marker System Based on ARToolKit**. NRC Technical Report, n. 47166, jul. 2004.

FILIPPO, Denise. Endler, Markus. Fuks, Hugo. **Colaboração Móvel com Realidade Aumentada**. 2005. 27 f. Monografia (Ciência da Computação). Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro.

FOGAGNOLI, Marcos. **Desenvolvimento de metodologia para análise dos atributos da cor através do processamento digital de imagens**. Campinas, SP, 2000.

GEIGER, C.; SCHMIDT, T.; STOCKLEIN, J. **Rapid Development of Expressive AR Applications**. In: INTERNATIONAL SYMPOSIUM ON MIXED AND AUGMENTED REALITY (ISMAR), 3., 2004, USA, Proceedings... Washington: IEEE Computer Society, 2004. p. 292-293.

GOMES NETO, Severino. Farias, Thiago. Teichrieb, Veronica. Kelner, Judith. **Criação de aplicações de Realidade Aumentada em dispositivos móveis baseados em Symbian OS**. III Workshop de Realidade Aumentada, 2006 (p. 16-19).

GONÇALVEZ, Glauber A. Centeno, Jorge. Abreu, Fernando C. S. Pedro, Patrícia de Castro. **Um Experimento de Realidade Aumentada Utilizando Técnicas de Síntese, Processamento Digital de Imagens e Fotogrametria Digital**. Florianópolis: 2004.

GONZALES, Rafael C. Woods, Richard E. **Processamento de imagens digitais**. São Paulo, Editora Edgard Blücher LTDA, 2000.

HARMO, Panu. **Etala: Virtual Reality Assisted Tele-Existence System for Remote Maintenance**. Disponível em http://automation.tkk.fi/files/etala/. Acesso em Nov/2007.

KATO, Billinghurst, **Marker Tracking and HMD Calibration for a video-based Augmented Reality Conferencing System**. In Proceedings of the 2nd International Workshop on Augmented Reality (IWAR 99). Outubro, 1999, San Francisco.

KELLER, Rodrigo dos Santos. **Realidade Virtual: equipamentos**. Disponível em http://www.pgie.ufrgs.br/siterv/equipamentos.htm. Acesso em Nov/2007.

KIM, Hae Young. **Síntese de imagem e rastreamento de raio**. São Paulo: 1992.

KUHN, Giovane R. Gomes, Paulo C. **Animação de um Personagem Virtual Utilizando Captura Óptica de Movimento com Marcações Especiais**. Blumenau, 2005.

LAMB, Philip. ARToolKit. Disponível em http://www.hitl.washington.edu/artoolkit/. Acesso em Nov/2007.

LIMA, João Paulo Silva do Monte. **Um Framework de Realidade Aumentada para o Desenvolvimento de Aplicações Portáveis para a Plataforma Pocket PC**. 2007. 49. Trabalho de Graduação (Ciência da Computação). Centro de Informática, Universidade Federal de Pernambuco.

LOPES, Eduardo Costa. **Determinando a Posição e Orientação da Mão Através de Imagens de Vídeo**. 2006. 86 f. Dissertação de Mestrado (Ciência da Computação). Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre.

MALIK, S.; ROTH, G.; MCDONALD, C. **Robust 2D Tracking for Real-Time Augmented Reality**. In: VISION INTERFACE (VI), 2002, Canada, Proceedings... p. 399-406.

MANSSOUR, Isabel H. Cohen, Marcelo. **Introdução à Computação Gráfica**. PUCRS: 2006.

MARQUÊS FILHO, Ogê; VIEIRA NETO, Hugo. **Processamento Digital de Imagens**. Rio de Janeiro: Brasport, 1999.

MILGRAM, P., TAKEMURA, H., UTSUMI, A., KISHINO, F.1994. **Augmented Reality**: A Class of Displays on the Reality-Virutality Continuum. In Proceedings SPIE vol. 2351:Telemanipulator and Telepresence Technologies.

NETTO, Antonio Valerio. Machado, Liliane dos Santos. Oliveira, Maria Cristina Ferreira. **Realidade Virtual - Definições, Dispositivos e Aplicações**. 2002. Notas Didáticas, número 34. ICMC-USP, São Carlos-SP, 33p. ISSN 0103-2585.

NUÑES, Juan Rafael. **Segmentação automática e classificação em vídeos esportivos**, 2006.

OWEN, Charles B. **What is the best fiducial?** In: The First IEEE. International Augmented Reality Toolkit Workshop. (2002) 98–105.

PAPPA, G. L. **Seleção de atributos usando algoritmos genéticos multiobjetivos**. 85p.(Mestrado em Informática Aplicada) – Pontifícia Universidade Católica do Paraná, Curitiba, 2002. Disponível em: http://www.ppgia.pucpr.br/ensino/defesas/Gisele_Lobo_2002.PDF Acesso em Jun/2007.

PARENT, Rick. **Computer animation**: algorithms and techniques. San Francisco: Morgan Kaufmann, 2002.

ROS, Giovana Angélica. **Visualização 3D de uma imagem digital**. 84p. Dissertação de Mestrado. Universidade Estadual Paulista, Faculdade de Ciências e Tecnologia. 2001.

SANTOS, J. C. **Extração de atributos de forma e seleção de atributos usando algoritmos genéticos para classificação de regiões**. São José dos Campos: INPE, 2007. 99p. ; (INPE-14671-TDI/1224).

SEMENTILLE, A. C. BREGA, José Remo Ferreira. ESCARAMUZI JÚNIOR, L. RODELLO, Ildeberto Aparecido. **Um Sistema de Rastreamento Óptico baseado em Marcadores**. In: SVR 2003 - VI Symposium on Virtual Reality, 2003, Ribeirão Preto. v. 01. p. 37-47.

SIGNPOST. **Mobile AR Navigation System**. Disponível em http://studierstube.icg.tu-graz.ac.at/projects/mobile/SignPost/. Acesso em Nov/2007.

SILVA, Romano J. M. Wagner, Gustavo N. Raposo, Alberto B. Gattas, Marcelo. **Experiência de Portais em Ambientes Arquitetônicos Virtuais**. Procedings of VI Symposium on Virtual Reality. 2003 (p. 117-128).

SOBEL, M. I. **Light.** The University of Chicago Press, 1987.

SOUZA SILVA, Márcio Alexandre, SILVA, Victor Vitiello, MIRANDA, Fábio R. **Visualização de órgão em Realidade Aumentada**. II Workshop de Realidade Aumentada. 2005 (60-63).

STUDIERSTUBE. Disponível em: http://studierstube.icg.tu-graz.ac.at. Acesso em Nov/2007.

TORI, Romero. Kirner, Claudio. Siscouto, Robson. **Fundamentos e Tecnologia de Realidade Virtual e Aumentada**. Belém, 2006.

WAGNER, Daniel. Schmalstieg, Dieter. **ARToolKitPlus for Pose Tracking on Mobile Devices**. Computer Vision Winter Workshop 2007, 2007.

WOODS, Eric. Mason, Paul. Billinghurst, Mark. **MagicMouse: an Inexpensive 6-Degree-of-Freedom Mouse**. Proceedings of Graphite 2003, Feb 11th-13th, 2003, Melbourne.

ZORZAL, Ezequiel R. Buccioli, Arthur A. B. Kirner, Claudio. **Usando Realidade Aumentada no Desenvolvimento de Quebra-cabeças Educacionais**. In: VIII Symposium on Virtual Reality, 2006. v. 01. p. 221-232.

# ANEXOS

# ANEXO 1

Código-fonte de exemplo de aplicativo desenvolvido com o Open Scene Graph. Neste exemplo, são criados dois objetos, sendo que um deles será exibido sem modificações, enquanto o segundo terá uma textura e transformações aplicadas.

```
#include <osgProducer/Viewer>
#include <osgDB/ReadFile>
#include <osg/Group>
#include <osg/Texture>
#include <osg/Texture2D>
#include <osg/TexGen>
#include <osg/MatrixTransform>

int main( int, char **)
{
    // cria o visualizador, com as configurações padrão
    osgProducer::Viewer viewer;
    viewer.setUpViewer( osgProducer::Viewer::STANDARD_SETTINGS );

    // cria um grupo para conter os grafos de cena
    osg::Group *grupo  = new osg::Group;

    // lê o grafo de cena de um arquivo
    osg::Node  *objeto1=
osgDB::readNodeFile( "/usr/local/src/osg/cow.osg" );

    // lê o grafo de cena de um arquivo. Este grafo de cena
    // sofrerá as transformações e conterá uma textura
    osg::Node  *objeto2= osgDB::readNodeFile( "/usr/local/src/osg/cow2.osg"
);

    // valida se os dados foram lidos com sucesso
    if ( !objeto1 ) return 1;
    if ( !objeto2 ) return 1;

    //adiciona o objeto sem transformações ao grupo
    grupo->addChild(objeto1);

    // lê o arquivo com textura
```

```
    osg::Image* image =
osgDB::readImageFile("/usr/local/src/osg/textura.png");
    if (image)
    {
        osg::Texture2D *texture = new osg::Texture2D;
        texture->setImage(image);

        // seta o tipo do mapa de reflexo como esférico
        osg::TexGen* texgen = new osg::TexGen;
        texgen->setMode(osg::TexGen::SPHERE_MAP);

        osg::StateSet* stateset = new osg::StateSet;
        stateset-
>setTextureAttributeAndModes(0,texture,osg::StateAttribute::ON);
        stateset-
>setTextureAttributeAndModes(0,texgen ,osg::StateAttribute::ON);

        // adiciona um mapa de reflexo ao objeto
        objeto2->setStateSet(stateset);
    }
    else
    {
        fprintf( stderr, "Erro ao ler imagem do reflexo!\n");
        exit(0);
    }

    // cria objetos para configurar transformações
    osg::Matrix modelScale;
    osg::Matrix modelTranslate;
    osg::Matrix modelRot;

    // cria matriz de transformações
    osg::MatrixTransform*   unitTransform = new osg::MatrixTransform;

    // seta objeto para escala
    modelScale.makeScale(0.7,0.7,0.7);

    // seta objeto para rotação
    modelRot.makeRotate(90.,osg::Vec3f(0,0,1));

    //seta objeto para translação
    modelTranslate.makeTranslate(osg::Vec3f(10,10,10));

    // seta matriz de transformações
    unitTransform->postMult(modelTranslate);
    unitTransform->postMult(modelRot);
    unitTransform->postMult(modelScale);

    // adiciona o objeto a lista de nodos a sofrerem transformação
    unitTransform->addChild(objeto2);

    // adiciona matriz de transformação ao grupo de objetos
    // a serem exibidos
    grupo->addChild(unitTransform);

    // indica o grupo de objetos para visualizador
    viewer.setSceneData( grupo );
```

```
    // cria a janela e executa as threads necessárias
    viewer.realize();

    // laço principal
    while( !viewer.done() )
    {
        // sincroniza a thread principal com as restantes
        viewer.sync();

        // manipula os eventos com mouse e teclado
        viewer.update();

        // efetua as transformações necessárias na figura sendo
        // visualizada e finaliza com um swap buffers
        viewer.frame();
    }
}
```

# ANEXO 2

Código-fonte de exemplo de aplicativo desenvolvido com o osgART. Neste exemplo, é associada uma imagem em 3D ao marcador.

```
#include <Producer/RenderSurface>
#include <osgProducer/Viewer>
#include <osg/Notify>
#include <osg/Node>
#include <osg/Group>
#include <osg/Geode>
#include <osg/Projection>
#include <osg/AutoTransform>
#include <osg/ShapeDrawable>
#include <osg/Geometry>
#include <osg/Image>
#include <osgDB/ReadFile>
#include <osgART/Foundation>
#include <osgART/VideoManager>
#include <osgART/ARTTransform>
#include <osgART/TrackerManager>
#include <osgART/VideoBackground>
#include <osgART/VideoPlane>
#include <osgART/VideoForeground>

int main(int argc, char* argv[])
{
    // inicializa o osgART
    osgARTInit(&argc, argv);

    // cria visualizador do openSceneGraph
    osgProducer::Viewer viewer;

    // seta configurações padrão
    viewer.setUpViewer(osgProducer::Viewer::ESCAPE_SETS_DONE);

    // carrega o plugin do osgART para integração com o ARToolKit
    osg::ref_ptr<osgART::GenericVideo> video =
osgART::VideoManager::createVideoFromPlugin("osgart_artoolkit");

    // verifica se o plugin foi carregado corretamente e caso contrário
    // exibe erro e sai do programa
    if (!video.valid())
    {
```

```
        osg::notify(osg::FATAL) << "Não foi possível inicializar o plugin
de vídeo!" << std::endl;
        exit(1);
    }

    // carrega o plugin do osgART pata integração com o tracker
    // do ARToolKit
    osg::ref_ptr<osgART::GenericTracker> tracker =
osgART::TrackerManager::createTrackerFromPlugin("osgart_artoolkit_tracker")
;

    // verifica se o tracker carregado é válido e caso contrário
    // exibe erro e sai do programa
    if (tracker.valid())
    {
        // obtém configuração do tracker
        osg::ref_ptr< osgART::TypedField<int> > _threshold =
reinterpret_cast< osgART::TypedField<int>* >(tracker->get("threshold"));

        // verifica se o valor é válido
        if (_threshold.valid())
        {
            // seta o valor do parâmetro
            _threshold->set(100);
            //imprime o valor setado
            osg::notify() << "Parâmetro 'threshold' = " << _threshold-
>get() << std::endl;
        }
        else
        {
            osg::notify() << "Plugin de tracker não tem suporte ao
parâmetro 'threshold'" << std::endl;
        }
    }
    else
    {
        std::cerr << "Não pode inicializar plugin de tracker!" <<
std::endl;
        exit(-1);
    }

    // Abre o vídeo
    video->open();

    // Inicializa o tracker, com parâmetro do vídeo inicializado
    // na linha anterior
    tracker->init(video->getWidth(), video->getHeight());

    // Cria grupo para objetos no primeiro plano
    osg::Group* foregroundGroup = new osg::Group();

    // Cria objeto que conterá o vídeo capturado
    osgART::VideoBackground* videoBackground=new
osgART::VideoBackground(video.get());

    // inicializa objeto de vídeo
    videoBackground->init();
```

```
    // adiciona o objeto do vídeo ao grupo de primeiro plano
    foregroundGroup->addChild(videoBackground);

    // cria matriz de projeção, conforme matriz obtida do tracker
    osg::Projection* projectionMatrix = new
osg::Projection(osg::Matrix(tracker->getProjectionMatrix()));

    // cria marcador de identificação '0'
    osg::ref_ptr<osgART::Marker> marker = tracker->getMarker(0);

    // verifica se o marcador existe e é válido e caso contrário
    // exibe erro e sai do programa
    if (!marker.valid())
    {
        osg::notify(osg::FATAL) << "Nenhum marcador encontrado!" <<
std::endl;
        exit(-1);
    }

    // ativa o marcador
    marker->setActive(true);

    // cria uma matriz de transformação obtida do marcador
    osg::ref_ptr<osg::MatrixTransform> markerTrans = new
osgART::ARTTransform(marker.get());

    // lê o modelo do arquivo
    osg::Node*  modelNode =
osgDB::readNodeFile("/usr/local/src/osg/cow.osg");

    // adiciona modelo à matriz de transformação
    markerTrans->addChild(modelNode);

    // cria objeto para conter matriz
    osg::Group* sceneGroup = new osg::Group();
    sceneGroup->getOrCreateStateSet()->setRenderBinDetails(5, "RenderBin");

    // adiciona matriz de transformação ao objeto
    sceneGroup->addChild(markerTrans.get());

    // adiciona objeto ao grupo no primeiro plano
    foregroundGroup->addChild(sceneGroup);

    // cria matriz de transformação e adiciona à matriz de projeção
    // como a matriz de transformação não tem valores, utiliza posição
    // como origem, rotação como 0 e escala 1
    osg::MatrixTransform* modelViewMatrix = new osg::MatrixTransform();
    modelViewMatrix->addChild(foregroundGroup);
    projectionMatrix->addChild(modelViewMatrix);

    // cria grupo para conter todos os objetos
    osg::ref_ptr<osg::Group> root = new osg::Group;
    root->addChild(projectionMatrix);

    // seta grupo de objetos à cena
    viewer.setSceneData(root.get());
```

```
    // cria a janela e executa as threads necessárias
    viewer.realize();

    // inicializa captura de vídeo
    video->start();

    // laço principal
    while (!viewer.done())
    {
        // sincroniza a thread principal com as restantes
        viewer.sync();

        // atualiza vídeo
        video->update();

        // seta imagem para o tracker
        tracker->setImage(video.get());
        tracker->update();

        // manipula os eventos com mouse e teclado
        viewer.update();

        // efetua as transformações necessárias na figura sendo
        // visualizada e finaliza com um swap buffers
        viewer.frame();
    }

    // atualiza visualizador e limpa objetos do frame
    viewer.sync();
    viewer.cleanup_frame();
    viewer.sync();

    // finaliza captura de vídeo
    video->stop();

    // finaliza leitura do vídeo
    video->close();
}
```

# ANEXO 3

Código fonte do protótipo de aplicativo desenvolvido no trabalho: o jogo “pong” em Realidade Aumentada.

```
#include <osg/PositionAttitudeTransform>
#include <Producer/RenderSurface>
#include <osgProducer/Viewer>

#include <osg/Notify>
#include <osg/Depth>
#include <osg/Node>
#include <osg/Group>
#include <osg/Geode>
#include <osg/Projection>
#include <osg/AutoTransform>
#include <osg/ShapeDrawable>
#include <osg/Geometry>
#include <osg/Image>
#include <osg/Texture>
#include <osg/TexMat>

#include <osgDB/ReadFile>

#include <osgART/Foundation>
#include <osgART/VideoManager>
#include <osgART/ARTTransform>
#include <osgART/TrackerManager>
#include <osgART/VideoBackground>
#include <osgART/VideoPlane>
#include <osgART/VideoForeground>

#include <osg/MatrixTransform>
#include <osgText/Text>

int main(int argc, char* argv[])
{
    //seta nível de notificação como apenas erros fatais.
    osg::setNotifyLevel(osg::FATAL);

    //inicializa o osgART.
    osgARTInit(&argc, argv);

    //cria objeto de visualização.
```

```
    osgProducer::Viewer viewer;

    //configura ações padrão do visualizador.
    viewer.setUpViewer(osgProducer::Viewer::ESCAPE_SETS_DONE);

    //seta modo de oclusão de objetos não visíveis.
    viewer.getCullSettings().setComputeNearFarMode(osg::CullSettings::DO_NO
T_COMPUTE_NEAR_FAR);

    // lê o plugin de vídeo.
    osg::ref_ptr<osgART::GenericVideo> video =
osgART::VideoManager::createVideoFromPlugin("osgart_artoolkit");

    //confere se o plugin foi carregado com sucesso.
    if (!video.valid())
    {
        //caso não carregou, finaliza o programa.
        osg::notify(osg::FATAL) << "Não pode inicializar plugin de vídeo!"
<< std::endl;

        //sai do programa
        exit(1);
    }

    //obtém as configurações de vídeo.
    osgART::VideoConfiguration* _config = video->getVideoConfiguration();

    //lê o plugin do tracker
    osg::ref_ptr<osgART::GenericTracker> tracker =
osgART::TrackerManager::createTrackerFromPlugin("osgart_artoolkit_tracker")
;

    if (tracker.valid())
    {

        //acessa o campo de configuração do nível de treshold.
        osg::ref_ptr< osgART::TypedField<int> > _threshold =
reinterpret_cast< osgART::TypedField<int>* >(tracker->get("threshold"));

        //caso conseguiu obter o campo, seta o valor do treshold.
        if (_threshold.valid())
        {
            _threshold->set(100);

            //exibe o treshold.
            osg::notify() << "Campo 'threshold' = " << _threshold->get() <<
std::endl;
        }
        else
        //caso não conseguiu obter o campo de treshold, exibe informação.
        {
            osg::notify() << "Campo 'threshold' não suportado para essa
biblioteca." << std::endl;
        }
    }
    else
    {
```

```
        //caso não conseguiu ler o tracker, exibe erro e finaliza.
        std::cerr << "Não pode ler o plugin do tracker!" << std::endl;

        //sai do programa
        exit(-1);
    }

    //abre o vídeo. Ainda não começa a captura, mas obtém informações sobre
o formato e outras configurações.
    video->open();

    //inicializa o tracker com as configurações de vídeo obtida.
    tracker->init(video->getWidth(), video->getHeight());

    //inicia construção da cena.

    //cria grupo de objetos do primeiro plano.
    osg::Group* foregroundGroup = new osg::Group();

    //cria o objeto para segundo plano, contendo o vídeo sendo capturado.
    osgART::VideoBackground* videoBackground=new
osgART::VideoBackground(video.get());

    //especifica a textura de vídeo (a USE_TEXTURE_RECTANGLE é mais rápida)
    videoBackground-
>setTextureMode(osgART::GenericVideoObject::USE_TEXTURE_RECTANGLE);

    //inicializa o vídeo em segundo plano.
    videoBackground->init();
    //seta transparência do vídeo.
    videoBackground->setTransparency(0.5);

    //adiciona o vídeo ao grupo no primeiro plano.
    foregroundGroup->addChild(videoBackground);

    //obtém a matriz de projeção do tracker
    osg::Projection* projectionMatrix = new
osg::Projection(osg::Matrix(tracker->getProjectionMatrix()));

    //obtém o marcador com identificação 0.
    osg::ref_ptr<osgART::Marker> marker = tracker->getMarker(0);

    //verifica se conseguiu ler o marcador com sucesso.
    if (!marker.valid()) //caso não conseguiu ler o marcador, exibe erro e
sai do programa.
    {
        osg::notify(osg::FATAL) << "Marcador não definido!" << std::endl;
        exit(-1);
    }

    //ativa o marcador
    marker->setActive(true);

    //cria a matriz de transformação relacionada ao marcador.
    osg::ref_ptr<osg::MatrixTransform> markerTrans = new
osgART::ARTTransform(marker.get());
```

```
    //cria a caixa que vai conter a raquete, na posição 0,0,-10, com
tamanho 10, 18, 0.5.
    osg::Box *raquete = new osg::Box   (osg::Vec3( 0, 0, -10 ), 10, 18,
0.5f);
    osg::ShapeDrawable* sd  = new osg::ShapeDrawable(raquete);

    //carrega a imagem de textura da raquete
    osg::Image* img =
osgDB::readImageFile("/home/njunior/osg/raquete.png");

    //configura a textura
    osg::TextureRectangle* texture = new osg::TextureRectangle(img);
    osg::TexMat* texmat = new osg::TexMat;
    texmat->setScaleByTextureRectangleSize(true);

    //configura o estado da textura
    osg::StateSet* state = sd->getOrCreateStateSet();
    state->setTextureAttributeAndModes(0, texture,
osg::StateAttribute::ON);
    state->setTextureAttributeAndModes(0, texmat, osg::StateAttribute::ON);
    state->setMode(GL_BLEND,osg::StateAttribute::ON);

    //desabilita a iluminação
    state->setMode(GL_LIGHTING, osg::StateAttribute::OFF);

    //cria o grupo para conter a raquete.
    osg::Geode* geode = new osg::Geode();

    //adiciona a raquete ao grupo.
    geode->addDrawable(sd);

    //cria matriz de transformação para setar escala da raquete.
    osg::MatrixTransform* unitTransform = new osg::MatrixTransform;
    osg::Matrix modelScale;
    modelScale.makeScale(10,10,10);
    unitTransform->postMult(modelScale);

    //adiciona o grupo da raquete à matriz de transformação.
    unitTransform->addChild(geode);
    markerTrans->addChild(unitTransform);

    //cria objeto para conter a cena.
    osg::Group* sceneGroup = new osg::Group();
    sceneGroup->getOrCreateStateSet()->setRenderBinDetails(5, "RenderBin");

    //adiciona a matriz de transformação do marcador à cena.
    sceneGroup->addChild(markerTrans.get());

    //adiciona cena à matriz de visualização e depois à matriz de projeção.
    osg::MatrixTransform* modelViewMatrix = new osg::MatrixTransform();
    modelViewMatrix->addChild(sceneGroup);
    projectionMatrix->addChild(modelViewMatrix);

    //cria grupo "raíz" e adiciona a matriz de projeção.
    osg::ref_ptr<osg::Group> root = new osg::Group;
    root->addChild(projectionMatrix);
```

```
    //cria objeto da bolinha.
    osg::ShapeDrawable* bola = new osg::ShapeDrawable(new
osg::Sphere(osg::Vec3( 0, 0, 0), 2));
    //seta cor do objeto
    bola->setColor(osg::Vec4(1, 0, 0, 0));

    //cria grupo que vai conter a bolinha
    osg::Geode* geode2 = new osg::Geode();
    geode2->addDrawable(bola);

    //cria matriz de transformação para a bola e adiciona o grupo da mesma
à matriz.
    osg::MatrixTransform* ballXform = new osg::MatrixTransform();
    ballXform->addChild(geode2);

    //cria grupo para conter a cena da bolinha e do vídeo e adiciona-os.
    osg::Group* sceneGroup2 = new osg::Group();
    sceneGroup2->addChild(foregroundGroup);
    sceneGroup2->addChild(ballXform);

    //adiciona grupo no grupo "raíz".
    root->addChild(sceneGroup2);

    //seta dados do visualizador
    viewer.setSceneData(root.get( ));
    viewer.realize();

    //inicia captura do vídeo.
    video->start();

    //inicializa variáveis para controle da bolinha
    int direcaoX = 0, direcaoY = 0;

    //variável para controlar se localizou o marcador.
    int localizou=0;

    //inicializa variáveis para controlar o número de quadros e quadros
inválidos.
    long invalidos=0, quadros=0;

    //configura para sempre exibir notificações do OSG.
    osg::setNotifyLevel(osg::ALWAYS);


    //obtém objeto da matriz de transformação do marcador.
    osg::ref_ptr<osg::MatrixTransform> transf = markerTrans.get();

    //laço principal do programa.
    while (!viewer.done())
    {
        osg::Vec3 newPosition;
        osg::Quat newRotation;

        //sincroniza threads.
        viewer.sync();

        //atualiza o vídeo.
```

```
        video->update();

        //atualiza o tracker com a nova imagem do vídeo.
        tracker->setImage(video.get());

        //atualiza tracker
        tracker->update();

        //obtem matriz de transformação do marcador
        osg::Matrix matrix = transf->getMatrix();

        if ( localizou==0 && marker->isValid( ) )
        //caso o marcador for válido e for a primeira vez que localiza.
        {
            //obtém posicionamento do marcador.
            osg::Vec3f pos2=matrix.getTrans();

            //seta a matriz de transformação obtida para a bolinha, para
setar posicionamento inicial.
            ballXform->setMatrix(matrix);

            //cria matriz de translação e aplica uma transformação.
            osg::Matrix mTransl;
            mTransl.makeTranslate(osg::Vec3f(0,0,10.f));
            ballXform->postMult(mTransl);

            localizou = 1;
        }
        if ( marker->isValid( ) )
        {
            //obtém matriz de transformação da bolinha.
            osg::Matrixd matrixBall = ballXform->getMatrix();
            osg::Vec3f translation, scale;
            osg::Quat rotation, so;

            //decompõe a matriz de transformação
            matrixBall.decompose( translation, rotation, scale, so );

            //seta translação (movimentação da bolinha).
            //Caso a variável direcaoY for 1, indica que é pra mover-se
para baixo. Caso for 0, deve mover-se para cima.
            //Caso a variável direcaoX for 1, indica que é pra mover-se
para a esquerda. Caso for 0, deve mover-se para a direita.
            matrixBall.makeTranslate( translation+osg::Vec3f(direcaoX ?
-1.f : 1.f, direcaoY ? -1.f : 1.f, 0.f) );

            //obtém a posição da bola (pos1) e do marcador (pos2)
            osg::Vec3f pos1=matrixBall.getTrans( );
            osg::Vec3f pos2=matrix.getTrans();
            osg::Vec3f scal=unitTransform->getMatrix( ).getScale( );
            osg::Vec3f tams=raquete->getHalfLengths( );

            //obtém matriz com escala do marcador.
            osg::Vec3f prod=matrix.getScale( );

            //distância entre o marcador e a bolinha.
            float distance_markers=(pos1-pos2).length();
```

```
            /*verifica a distância entre marcador e bolinha (caso a
distância em y for menor que a altura da raquete, então considera que
colidiu em y. Caso a distância em x for menor que a largura da raquete,
considera que colidiu em x. Caso colidiu em ambos os planos, troca a
movimentação da bolinha.
            */
            if ( direcaoY && abs((int)(pos1.y( )-pos2.y( )))<=18 &&
abs((int)(pos1.x( )-pos2.x( )))<=10 )
            {
                //Caso colidiu em ambos os planos
                direcaoY = direcaoY ? 0 : 1;
                direcaoX = direcaoX ? 0 : 1;
            }
            else
            {
                //caso a posição bolinha em y for maior que 75, considera
que chegou ao topo e inverte a movimentação neste sentido.
                if ( matrixBall.getTrans( ).y( ) > 75 )
                    direcaoY = 1;

                //caso a posição bolinha em x for maior que 125, considera
que chegou ao limite direito e inverte a movimentação neste sentido.
                if ( matrixBall.getTrans( ).x( ) > 125 )
                    direcaoX = 1;

                //caso a posição bolinha em y for menor que -75, considera
que chegou à borda da tela e inverte a movimentação neste sentido.
                if ( matrixBall.getTrans( ).y( ) < -75 )
                    direcaoY = 0;

                //caso a posição bolinha em x for menor que -125, considera
que chegou ao limite esquerdo e inverte a movimentação neste sentido.
                if ( matrixBall.getTrans( ).x( ) < -125 )
                    direcaoX = 0;
            }

            //imprime posicionamento da bolinha.
            fprintf( stderr, "x:%f y:%f z:%f\n",
matrixBall.getTrans( ).x( ), matrixBall.getTrans( ).y( ),
matrixBall.getTrans( ).z( ));

            //seta matriz de transformação da bolinha.
            ballXform->setMatrix(matrixBall);
        }

        //caso o marcador não for válido, incrementa a variável que conta o
número de quadros que não localizou o marcador.
        if  ( ! marker->isValid( ) )
        {
            invalidos++;
        }

        //atualiza o visualizador
        viewer.update();
        viewer.frame();
```

```
            //incrementa o número de quadros
            quadros++;
        }
        //imprime o número de quadros obtidos e o número de quadros que não
    obteve o marcador.
        fprintf( stderr, "Total de invalido: %d em %d quadros\n", invalidos,
    quadros);

        //atualiza o visualizador, limpa a memória e sincroniza as threads.
        viewer.sync();
        viewer.cleanup_frame();
        viewer.sync();

        //finaliza e fecha o vídeo.
        video->stop();
        video->close();

    }
```