UNIVERSIDADE ESTADUAL PAULISTA
Faculdade de Ciências - Bauru
Bacharelado em Ciência da Computação

Daniel Zuniga Vielmas

# O Aprimoramento de um Software de Estúdio Virtual por Meio de Técnicas de Matting Digital e Registro de Objetos Virtuais

UNESP

2014

Daniel Zuniga Vielmas

# O Aprimoramento de um Software de Estúdio Virtual por Meio de Técnicas de Matting Digital e Registro de Objetos Virtuais

Orientador: Prof. Adj. Antonio Carlos Sementille
Co-orientador: Prof. Adj. João Fernando Marar

Monografia apresentada junto à disciplina Projeto e Implementação de Sistemas II, do curso de Bacharelado em Ciência da Computação, Faculdade de Ciências, Unesp, campus de Bauru, como parte do Trabalho de Conclusão de Curso.

UNESP

2014

Vielmas, Daniel Zuniga.
O aprimoramento de um software de estúdio virtual por meio de técnicas de matting digital e registro de objetos virtuais / Daniel Zuniga Vielmas, 2014
87 f. : il.

Orientador: Antonio Carlos Sementille

Monografia (Graduação)–Universidade Estadual Paulista. Faculdade de Ciências, Bauru, 2014

1. Estúdio virtual. 2. Realidade aumentada. 3. Matting digital. 4. Chroma-key. I. Universidade Estadual Paulista. Faculdade de Ciências. II. Título.

Daniel Zuniga Vielmas

# O Aprimoramento de um Software de Estúdio Virtual por Meio de Técnicas de Matting Digital e Registro de Objetos Virtuais

Monografia apresentada junto à disciplina Projeto e Implementação de Sistemas II, do curso de Bacharelado em Ciência da Computação, Faculdade de Ciências, Unesp, campus de Bauru, como parte do Trabalho de Conclusão de Curso.

BANCA EXAMINADORA

Antonio Carlos Sementille
Professor Adjunto
UNESP - Bauru

Simone das Graças Domingues Prado
Professora Assistente Doutora
UNESP - Bauru

Renê Pegoraro
Professor Assistente Doutor
UNESP - Bauru

Bauru, 24 de Junho de 2014.

# RESUMO

Um sistema de estúdio virtual pode utilizar-se de tecnologias como realidade aumentada e *matting* digital para diminuir os custos de produção ao mesmo tempo em que fornece os mesmos recursos de um estúdio convencional. Com isso, é possível aos estúdios atuais, a um baixo custo e utilizando dispositivos convencionais, criarem produções com maior qualidade de imagem e efeitos.

Algumas dificuldades são recorrentes em aplicações de estúdios virtuais que utilizam realidade aumentada e *matting* digital. O registro de objetos virtuais na técnica de realidade aumentada sofre de problemas causados por distorções ópticas da câmera, erros no sistema de rastreamento de marcadores, falta de calibração dos equipamentos ou de controle do ambiente (por exemplo, iluminação), ou mesmo por atrasos na exibição dos objetos virtuais. Já o principal problema do *matting* digital é sua execução em tempo real para pré-visualização da cena, devendo ter velocidade de processamento otimizada ao mesmo tempo em que mantem a melhor qualidade possível de imagem.

Levando em consideração o contexto exposto, este trabalho deu continuidade a um sistema de estúdio virtual denominado ARStudio, por meio do aperfeiçoamento do *matting* digital, registro de objetos virtuais e a introdução da segmentação baseada em mapa de profundidade, além de adicionar maior controle sobre as funcionalidades já implementadas no sistema.



**Palavras-chave:** Estúdio Virtual; Realidade Aumentada; *Matting* Digital; *Chroma-key*.

# ABSTRACT

A virtual studio system can use technologies as augmented reality and digital matting to decrease production costs at the same time it provides the same resources of a conventional studio. With this, it's possible for the current studios, with low cost and using conventional devices, to create productions with greater image quality and effects.

Some difficulties are recurrent in virtual studio applications that use augmented reality and digital matting. The virtual objects registration in augmented reality techniques suffer from problems caused by optical distortions in the camera, errors in the marker tracking system, lack of calibration on the equipments or on the environment (lighting, for example), or even by delays in the virtual objects display. On the other hand, the digital matting's main problem is the real-time execution to preview the scene, which must have optimized processing speed at the same time while maintain the best image quality possible.

Taking the given context into consideration, this work aims to give continuity to a virtual studio system called ARStudio, by enhancing digital matting, virtual objects registration and introducing a segmentation based on depth map, yet adding better control over functionalities previously implemented.



**Keywords:** Virtual Studio; Augmented Reality; Digital Matting; Chroma-key.

# LISTA DE ILUSTRAÇÕES

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS

| | |
|---|---|
| API | *Application Programming Interface* |
| BBC | *British Broadcasting Corporation* |
| CMOS | *Complementary Metal-Oxide Semicondutor* |
| CPU | *Central Processing Unit* |
| ELSET | *Electric Set for Broadcast Studios* |
| FIFO | *First In First Out* |
| FPS | *Frames Per Second* ou *Frames* Por Segundo |
| GLSL | *OpenGL Shading Language* |
| GMD | *German National Research Center for Information Technology* |
| GPU | *Graphics Processing Unit* |
| GUI | *Graphical User Interface* |
| HMD | *Head-Mounted Display* |
| IDE | *Integrated Development Environment* |
| IR | *Infrared* |
| NHK | *Nippon Hōsō Kyōkai* |
| OpenCV | *Open Source Computer Vision Library* |
| OpenGL | *Open Graphics Library* |
| OSGART | *OpenSceneGraph + ARToolKit* |
| RA | Realidade Aumentada |
| RGB | *Red*, *Green*, *Blue* |
| RGBA | *Red*, *Green*, *Blue*, *Alpha* |
| SACI | Sistemas Adaptativos e Computação Inteligente |
| SDK | *Software Development Kit* |
| VAP | *Video Art Production* |

# ÍNDICE

# 1 INTRODUÇÃO

Nas últimas décadas, avanços na tecnologia de hardware dos computadores causaram mudanças drásticas na produção de conteúdo para cinema e televisão, principalmente no que diz respeito a gravação e pós-produção. Com equipamentos menores, mais leves e flexíveis, os estúdios começaram a se voltar à manipulação das imagens virtualmente, utilizando todo o potencial dos computadores para realizar processamento de vídeo em tempo real.

Assim surgiram os estúdios virtuais, um conceito de estúdio que proporciona a composição de vídeo real com imagens sintéticas. Também chamada de realidade virtual em terceira pessoa, essa técnica de composição permite que as pessoas que assistirem a esse “sinal mixado” vejam pessoas e outros objetos físicos combinados com um ambiente virtual (GIBBS et al., 1998).

Na maneira tradicional de produção de filmes, a construção física dos cenários exige que, depois de seu planejamento e desenho, os objetos idealizados desse cenário sejam construídos, transportados até o estúdio em que a cena será gravada, montados de acordo com o cenário planejado e, depois de serem utilizados, armazenados para uso posterior ou descartados. Esse ciclo de construção física do cenário é o aspecto mais caro de uma produção televisiva (BLONDÉ et al., 1996), despendendo muitos esforços e criando alguns inconvenientes: nem sempre os objetos construídos saem de acordo com o projeto desenhado; para cenas mais complexas é necessário um número muito grande de objetos no cenário que devem ser construídos; o transporte pode se tornar um problema se os objetos do cenário forem muito grandes ou se forem muito numerosos; o tempo de montagem e desmontagem do cenário pode ser muito grande; os objetos construídos para o cenário podem nunca mais ser utilizados, o que traz um desperdício de material, ou se forem armazenados para uso posterior precisarão de um espaço para serem armazenados. Na abordagem dos estúdios virtuais, o cenário é construído virtualmente, eliminando custos e restrições de tempo com sua flexibilidade. O cenário pode ser modificado e os resultados das modificações são obtidos em tempo real.

Para isso, os estúdios virtuais podem utilizar técnicas de realidade aumentada, uma variação da realidade virtual. A tecnologia de realidade virtual imerge o usuário completamente em um ambiente tridimensional gerado por computador, com o qual o mesmo pode interagir em tempo real. Por outro lado, a realidade aumentada permite

ao usuário ver o mundo real acrescido de objetos virtuais com os quais também pode interagir, compondo uma cena aumentada. Dessa forma, a realidade aumentada complementa a realidade em vez de substituí-la (AZUMA, 1997). Para identificar a posição e orientação em que os objetos virtuais devem aparecer na cena, marcadores são posicionados na cena real e, através de rastreamento, é possível identificá-los e associá-los aos objetos virtuais.

Outra técnica que estúdios virtuais utilizam é o *matting* digital. Trata-se da aplicação de métodos para a extração do *foreground* (área de interesse, como por exemplo os atores) de uma imagem ou um vídeo (WANG; COHEN, 2007). Tendo a estimativa do *foreground* de uma imagem, é possível segmentá-la, separando o plano de fundo da imagem do resto da mesma. Dessa forma, o estúdio virtual deve retirar o *background* (plano de fundo), substituindo-o por outro que contenha o cenário desejado. Porém, essa segmentação pode ocasionar o efeito de serrilhamento ou *aliasing*, problema causado quando a transparência das bordas da área de interesse segmentada é desprezada (SMITH, 1995). A solução para o serrilhamento é a atribuição de um valor de transparência para cada ponto da imagem, suavizando assim os pontos serrilhados. Além da criação de transparência na segmentação, filtros podem ser aplicados na imagem para melhorar o resultado do *matting*.

Este trabalho propõe o aperfeiçoamento de um sistema de estúdio virtual iniciado em 2010 chamado ARStudio (CAMPOS et al., 2010; SEMENTILLE et al., 2012) adicionando um novo algoritmo de *matting* com aplicação de transparência e filtros, além de novas formas de controle sobre os elementos de realidade aumentada.

## 1.1 Problema

O *matting* digital analisa uma imagem para determinar seu *foreground* e extraí-lo, combinando-o com outra imagem. Uma forma de facilitar a análise da imagem é utilizando a técnica de *chroma-key*. O *chroma-key* consiste em estabelecer uma cor uniforme (cor chave) e utilizá-la como *background* na cena de filmagem, assim essa cor se tornará uma máscara para a extração do *foreground* (VAN DEN BERGH; LALIOTI, 1999). Para que essa técnica funcione, é preciso que o *foreground* não contenha a cor chave, isolando o *background* através da cor e permitindo sua extração.

Porém, o problema está na relação entre o tempo de execução do *chroma-key* e a qualidade do resultado gerado por ele. Um algoritmo de *matting* digital simples produz resultados rápidos mas pouco refinados, enquanto um algoritmo mais complexo pode gerar bons resultados mas que não são executados em tempo real. Para estúdios virtuais, é interessante, tanto para atores quanto para diretores, visualizar a cena composta virtualmente sem que seja necessário que a filmagem passe pela pós-produção, fornecendo um *feedback* dos resultados das filmagens em tempo real. Esse dinamismo pode ser atingido selecionando um algoritmo de *matting* digital que se aproveite da informação extra de cor assim como o *chroma-key* e, ao mesmo tempo, que aplique transparência nos pontos da imagem, gerando os melhores resultados possíveis em tempo real.

Outro problema que é recorrente em aplicações de realidade aumentada é a dificuldade na realização do registro de um objeto virtual, que é o alinhamento preciso entre objetos reais e virtuais. O registro pode ser feito usando técnicas baseadas em padrões e técnicas baseadas em características naturais. A primeira adota o uso de marcadores (símbolos que representam um padrão) que são rastreados para efetuar o registro, enquanto a segunda analisa características naturais do cenário para que, uma vez posicionado o objeto virtual, seja possível realizar seu registro observando a mudança de posicionamento dessas características.

Os erros de registro utilizando marcadores podem ser causados por distorções ópticas da câmera que captura a cena, erros no sistema de rastreamento dos padrões dos marcadores, falta de calibração dos equipamentos ou de controle do ambiente (por exemplo, iluminação), ou mesmo por atrasos na exibição dos objetos virtuais no cenário (AZUMA, 1997). Esses erros de registro se tornam mais evidentes quando o marcador está fixo em uma posição, com a câmera parada, pois nessa situação, enquanto o objeto virtual deveria permanecer fixo assim como o marcador, ele geralmente é renderizado com pequenas variações de posição ao longo dos quadros do vídeo, dando a impressão de movimento. Isso pode ser resolvido aplicando uma abordagem especial para marcadores fixos, impedindo que sejam rastreados quadro a quadro, pois suas posições já são conhecidas no primeiro quadro em que foram rastreados.

Em outra situação que acarreta o erro do registro do objeto virtual, o marcador associado ao objeto pode ser obstruído por um objeto real em cena, ou ainda pode sair do campo de visão da câmera que captura a cena. Nesse caso, o objeto

desaparecerá da cena mesmo que não devesse, já que seu marcador não consegue ser rastreado. Uma possível solução para isso é o uso de multimarcadores, ou seja, o objeto virtual não estará associado apenas a um marcador, mas a vários deles. Dessa forma, mesmo que um marcador deixe de ser rastreado, os outros ainda permitirão o registro do objeto virtual, fazendo com que ele apareça na posição correta.

Ainda que não haja erro no rastreamento do marcador, permanece o problema de identificação da posição do objeto em relação aos atores presentes na cena. Mesmo que o marcador seja posicionado no fundo do cenário e que o objeto associado a ele deva aparecer atrás dos atores, o objeto será exibido na frente dos atores, pois não há informação que indique a distância do objeto em relação aos mesmos. Para obter essa informação, é possível utilizar o Kinect, um dispositivo que possui um sensor de luz estruturada e que calcula a distância dos objetos (MICROSOFT, 2009). Assim, posicionando o Kinect próximo à câmera utilizada para gravar a cena, ele fornecerá a distância dos atores em relação à câmera, e através do rastreamento do marcador obtém-se a distância do objeto virtual em relação à câmera. Com essas duas informações de distância é possível estimar a distância do objeto em relação aos atores e definir se o objeto deve aparecer à frente ou atrás dos atores.

Além desses problemas citados, um problema específico do sistema de estúdio virtual ARStudio desenvolvido por Campos et al. (2010) é o da disponibilização do fluxo de vídeo de entrada. Enquanto o estúdio virtual desenvolvido permite o armazenamento do vídeo processado pelo software, com o *matting* digital aplicado e os objetos virtuais inseridos na cena, ele não permite ao usuário armazenar o vídeo de entrada do software, ou seja, não permite o armazenamento do vídeo pré-processamento. Essa é uma funcionalidade útil para que esse vídeo não processado seja utilizado posteriormente para inclusão de efeitos de pós-produção.

Tendo em vista resolver os problemas mencionados, novas funcionalidades foram estudadas e aplicadas ao software de estúdio virtual.

## 1.2 Justificativa

Um estúdio virtual diminui os custos de produção de uma filmagem ao mesmo tempo em que traz uma flexibilidade maior à produção, permitindo uma maior liberdade à criatividade dos produtores uma vez que o cenário e outros objetos dele serão criados virtualmente e podem ser alterados rapidamente.

Tendo isso em vista, a criação de um software de estúdio virtual que agregue uma gama de efeitos e tecnologias ao mesmo tempo em que permite um alto nível de controle sobre suas funcionalidades é de grande utilidade para estúdios cinematográficos e de televisão.

Aumentar as funcionalidades do sistema e permitir total controle sobre essas funcionalidades terá impacto na qualidade final das produções. Por isso, é importante que o software esteja em constante desenvolvimento, implementando novos algoritmos de processamento de imagens e realidade aumentada de forma a se adequar aos diversos tipos de produções.

Com isso, reduz-se o custo da criação de conteúdo para cinema e televisão, abrindo caminho para que estúdios de baixo orçamento consigam produzir filmagem com mais opções de efeitos e, principalmente, com melhor qualidade.

## 1.3 Objetivos do Trabalho

Por se tratar de um projeto em andamento, o foco foi o desenvolvimento de novas funcionalidades ao software ao mesmo tempo em que as funcionalidades já existentes foram sendo aprimoradas. Para isso, foi necessário o estudo do software ARStudio, de novos algoritmos a serem implementados e novas opções de controle sobre os aspectos já presentes no programa.

Os objetivos deste trabalho foram:

- Investigar algoritmos de *matting* digital com aplicação de transparência capazes de serem executados em tempo real e selecionar um desses algoritmos para aplicação no software de estúdio virtual;
- Estender o uso de marcadores para casos com objetos virtuais fixos (marcadores fixos) e com obstrução de marcadores (multimarcadores), com o intuito de melhorar o registro dos objetos virtuais nesses casos;
- Aprimorar o registro de objetos virtuais com relação aos atores em cena utilizando o Kinect, a fim de fazer com que os objetos sejam inseridos na profundidade correta em relação ao ator;
- Permitir o armazenamento do fluxo de vídeo não processado concorrentemente ao armazenamento do fluxo de vídeo processado através do software.

## 1.4 Organização do documento

O capítulo 2 apresenta os conceitos envolvidos no trabalho e utilizados durante seu desenvolvimento. Os temas de estúdios virtuais, *matting* digital e realidade aumentada são abordados nesse capítulo.

O capítulo 3 detalha as etapas do desenvolvimento realizadas para alcançar os objetivos do trabalho.

No capítulo 4, os testes realizados no desenvolvimento do trabalho são apresentados e seus resultados são analisados.

O capítulo 5 apresenta as conclusões e os trabalhos futuros.

# 2 FUNDAMENTAÇÃO TEÓRICA

Em um estúdio virtual, técnicas de realidade aumentada e *matting* digital são aplicadas às filmagens para compor o resultado final da produção. Portanto, é essencial descrever os conceitos dessas técnicas para que se possa entender melhor o processo realizado em um sistema de estúdio virtual.

Aprofundando-se mais sobre o *matting* digital, também é importante ressaltar as técnicas que compõem o método de *matting* digital utilizado no software de estúdio virtual, como o *chroma*-key e a aplicação de transparência nos *pixels* da imagem para suavizar o serrilhamento. O algoritmo de *color difference key*, técnica de *matting* digital desenvolvida por Petro Vlahos (1971) que foi estudada e implementada no ARStudio, também é apresentada, juntamente com uma breve apresentação técnica do dispositivo Kinect e da programação de *shaders* no OpenGL, ambos utilizados no desenvolvimento do trabalho para o registro de objetos virtuais em relação aos atores.

Neste capítulo, uma breve introdução ao conceito de estúdio virtual e seu histórico também são expostos.

## 2.1 Estúdios virtuais

Estúdios virtuais são alternativas mais flexíveis para a produção de conteúdo televisivo do que os estúdios convencionais, e têm sido recentemente utilizados na produção de vídeos para televisão e cinema. Sistemas de estúdio virtual atuam compondo filmagens de cenas reais com objetos 3D que são renderizados em tempo real e sincronizados com o movimento da câmera (GÜNSEL; TEKALP; VAN BEEK, 1997). Eles possibilitam uma maior criatividade ao mesmo tempo em que reduzem custos. A flexibilidade oferecida pela facilidade de mudança automática de cenário elimina custos e restrições de tempo para construir, armazenar e transportar os objetos do mesmo. Várias produções podem compartilhar o mesmo espaço e equipamento, e a rapidez de troca de cenários acarreta um aumento na produtividade do estúdio (BLONDÉ et al., 1996). Além disso, um estúdio virtual torna possível a habilidade de ver efeitos de vídeo em tempo real ao invés de vê-los durante a pós-produção.

Segundo Rahbar e Pourreza (2008), sistemas de estúdios virtuais contêm três componentes principais:

- Um sistema de rastreamento de câmera, que cria um fluxo de dados descrevendo a perspectiva da câmera, que geralmente é chamado de processo de estimativa da pose da câmera;
- Um software de renderização em tempo real, que usa os dados de rastreamento da câmera e gera uma imagem sintética de um estúdio de gravação (ou set de filmagem);
- Um sistema de mixagem de vídeo, que combina a saída da câmera do estúdio com o vídeo do software de renderização em tempo real, a fim de produzir o vídeo final combinado.

O processo que envolve o software de estúdio virtual começa com uma câmera filmando os atores em um ambiente controlado. As imagens dos atores são segmentadas em tempo real com precisão usando técnicas de *matting* digital, como o *chroma-key*. A silhueta isolada dos atores então é integrada em um novo ambiente sintético. Além disso, para introduzir interação dos atores com o ambiente virtual, objetos virtuais são colocados em cena utilizando técnicas de realidade aumentada (GRAU; PRICE; THOMAS, 2002). Os conceitos de *matting* digital, *chroma-key* e realidade aumentada são definidos nas próximas seções deste capítulo.

Todo esse processo exige uma grande quantidade de processamento computacional, e só é possível graças aos avanços tecnológicos na área de eletrônica e hardware de computador. Esses avanços acarretaram o surgimento de processadores mais potentes e rápidos, além de equipamentos de câmera e iluminação menores, mais eficientes e que produzem resultados de melhor qualidade. Pesquisas na área de produção de conteúdo televisivo também são responsáveis pelo desenvolvimento dos estúdios virtuais existentes atualmente.

Na final da década de 80, pesquisadores da companhia de transmissão japonesa NHK desenvolveu uma forma de segmentação da imagem chamada *Synthevision*, que acabou virando um produto. Na técnica utilizada no *Synthevision*, sensores eram acoplados à câmera para gerar informações sobre as operações realizadas pela mesma, como mudanças no zoom ou no foco da câmera. Essas operações eram então simuladas na imagem de fundo que seria aplicada à filmagem, alterando a perspectiva do fundo para corresponder à mesma perspectiva da câmera. Compondo as imagens obtidas na filmagem com a imagem de fundo modificada através de um hardware específico para tal operação, o resultado era obtido.

O *Synthevision* foi utilizado pela primeira vez nas Olimpíadas de Seul em 1988 pela NHK, e apesar de usar apenas imagens estáticas como imagem de fundo, seus desenvolvedores prenunciaram o surgimento de estúdios virtuais utilizando o mesmo conceito do *Synthevision*, usando imagens de computação gráfica como imagem de fundo, possibilitando a criação de cenários que não poderiam ser criados em um estúdio real.

Cenários totalmente virtuais gerados em tempo real começaram a surgir no Japão em 1991. A NHK usou um protótipo de sistema de estúdio virtual para produzir um documentário científico chamado "*Nanospace*". Esse protótipo já apresentava os principais aspectos de um sistema de estúdio virtual: renderização de plano de fundo em tempo real com um sistema de rastreamento da câmera em tempo real. Porém, o desempenho dos hardwares gráficos da época foram um empecilho para os planos de desenvolvimento da NHK.

Em 1993, grandes avanços na arquitetura de hardwares gráficos fizeram surgir os primeiros estúdios virtuais comerciais. Até então, os sistemas de estúdio virtual existentes eram para uso interno de companhias de transmissão (GIBBS et al., 1998).

A Tabela 1 expõe o desenvolvimento dos primeiros estúdios virtuais durante os anos.

Tabela 1 - Cronologia dos primeiros estúdios virtuais

| **Data** | **Empresa (Produto)** |
|---|---|
| 1988 | NHK (Synthevision) |
| 1991 | NHK (sistema interno) |
| 1992 | Ultimatte (cenário virtual pré-renderizado) |
| 1993 | BBC (sistema interno – cenário virtual pré-renderizado) |
| 1994 | IMP (Platform)<br>VAP (ELSET)<br>GMD (3DK) |
| 1995 | Accom (ELSET)<br>RT-Set Ltd. (Larus, Otus)<br>ElectroGIG (Reality Tracking)<br>Softimage/INA (Virtual Theater/Hybrid Vision)<br>Orad (Virtual Set) |
| 1996 | Discreet Logic (Vapour)<br>Orad (Cyberset)<br>Evans and Sutherland (MindSet)<br>Vinten (VideoScape)<br>Radamec (Virtual Scenario) |

Fonte: GIBBS et al., 1998

Na Tabela 2 estão alguns estúdios virtuais comercializados atualmente, assim como estúdios virtuais que são fruto de iniciativas de pesquisa.

Tabela 2 - Estúdios virtuais atuais

| **Empresa/Instituição** | **Estúdio virtual** |
|---|---|
| Orad | ProSet |
| For-A | digiStorm |
| Vizrt | Viz Visual Studio |
| Darim Vision | VS (1000, 2000, 4000) |
| BBC Research & Innovation | ORIGAMI<br>iview<br>RE@CT<br>VSAR |

Fonte: elaborada pelo autor

## 2.2 *Matting* digital

O termo "*matte*" em cinematografia e fotografia era originalmente utilizado para designar uma tira de filme monocromático que era transparente em algumas partes e opaca em outras. A ideia era colocar essa tira sobre a tira do rolo de filme original colorido, e quando a luz era projetada sobre as duas tiras sobrepostas, a luz iluminaria apenas a parte do filme colorido que estivesse sob a parte transparente da tira monocromática. Dessa forma, era possível segmentar a imagem do filme, fazendo com que apenas uma área de interesse demarcada pela tira monocromática fosse exibida na tela de projeção (SMITH; BLINN, 1996).

O *matting* digital funciona de uma forma análoga ao *matte* utilizado na cinematografia e fotografia originalmente: trata-se de uma estimativa da área de interesse que se pretende preservar, chamado de *foreground* ou primeiro plano, e essa estimativa é utilizada para separar o *foreground* do resto da imagem, chamado de *background* ou plano de fundo. O processo de segmentação da imagem em *foreground* e *background* é sucedido por um processo de composição que cria uma nova imagem combinando o *foreground* extraído com um novo *background* (ZHENG; KAMBHAMETTU, 2009).

Um dos problemas do *matting* digital é a identificação do *foreground*. Algumas abordagens de *matting* digital dependem da interação do usuário para identificar a área de interesse da imagem, enquanto outras recorrem a elementos ou características da imagem para identificar essa área. Esses elementos e

características podem ser extraídos por uma análise de cor, no caso do *background* ter uma cor homogênea conhecida, ou por análise estatística, no caso de um *background* arbitrário (CHUANG et al., 2001).

## 2.3 Transparência de *pixels*

No campo da ótica, transparência é a propriedade física que permite a passagem da luz através de um material. Para reproduzir essa propriedade em uma imagem digital, essa propriedade deve ser aplicada a cada unidade da imagem, ou seja, cada *pixel* de uma imagem deve receber um valor de transparência. Assim, sobrepondo uma imagem sobre outra, os valores de transparência dos *pixels* da imagem sobreposta serão considerados para que se possa calcular a cor do *pixel* resultante dessa sobreposição. Em processamento de imagens, a noção de transparência geralmente é dada pelo seu oposto, a opacidade: um *pixel* sem nenhum nível de transparência é um *pixel* com opacidade máxima.

O espaço de cores comumente utilizado na representação digital é o RGB (*Red*, *Green*, *Blue*), e cada um dos três componentes desse espaço representa a intensidade da respectiva cor: vermelho, verde ou azul. Juntando-se essas três cores, levando em consideração suas respectivas intensidades, obtêm-se a cor do *pixel*. No *matting* digital, um componente separado foi criado para determinar o quanto o *pixel* que faz parte do *foreground* está cobrindo o *pixel* pertencente ao background, representando a opacidade do *pixel*. Esse novo componente foi chamado de canal *alpha* (*α*), e recebe o valor 0 para indicar que o *pixel* não está sendo coberto, o valor 1 para indicar que o *pixel* está totalmente coberto, e um valor fracionário para indicar cobertura parcial. Assim, com esse novo componente o sistema passou a se chamar RGBA (*Red*, *Green*, *Blue*, *Alpha*), com um canal para cada cor do espaço RGB e um canal para o valor *alpha* (PORTER; DUFF, 1984).

Para determinar a cor do *pixel* utilizando o sistema RGBA, realiza-se a aritmética da composição, que é dada pela Equação 1 (PORTER; DUFF, 1984).

$$C = \alpha F + (1 - \alpha)B \quad (1)$$

Na Equação 1, *C* representa a cor resultante da composição, *F* e *B* representam as cores do *foreground* e do *background* respectivamente e *α* representa o componente de opacidade *alpha*.

## 2.4 *Chroma-key*

Uma das abordagens de *matting* digital mais utilizadas em produções televisivas é o *chroma-key*. Com o *chroma-key*, uma filmagem é feita em um estúdio com um plano de fundo de cor uniforme. O ator, que não deve estar vestindo roupas com a mesma cor do plano de fundo, pode então ser diferenciado do plano de fundo pela cor (VAN DEN BERGH; LALIOTI, 1999).

As primeiras formas de realizar o processamento da imagem utilizando a técnica de *chroma-key* faziam uso de um hardware específico, na qual os sinais de vídeo passavam por um circuito eletrônico de discriminação de cor para comparar os componentes de cor de cada *pixel* e, depois de separado o sinal do *foreground*, ele passava por um circuito de mixagem para compor a imagem com um novo *background* (VLAHOS, 1971). Atualmente, esse processo é comumente realizado através de software, eliminando a necessidade de hardware especial com alto custo (VAN DEN BERGH; LALIOTI, 1999).

A cor uniforme escolhida para constituir o *background* é chamada cor chave. Usualmente escolhe-se como cor chave as cores azul ou verde pelo fato de não serem predominantes na pigmentação da pele humana, causando assim menos interferência no processo de *chroma-key* com atores em cena. Porém a cor verde apresenta mais um fator de preferência sobre as demais cores: o mosaico de filtros de cor que fica sobre os fotosensores dos dispositivos de captura de imagens (como câmeras e filmadoras) possui uma predominância de filtros da cor verde para simular a fisiologia do olho humano, que é mais sensível à luz verde (HAILEY, 2002). Assim, a imagem capturada pelo dispositivo é mais próxima à imagem vista pelo olho humano, com uma fidelidade maior na captura da cor verde, fazendo com que a cor verde seja a ideal para uso no *chroma-key* já que ela é capturada com maior precisão que as demais.

Um algoritmo de *chroma-key* desenvolvido por Van Den Bergh e Lalioti (1999) realiza o processo de forma simplificada utilizando um máximo de cinco operações por *pixel*. Os autores do algoritmo utilizam a cor azul como cor chave do *chroma-key*.

O primeiro passo do algoritmo é identificar quando um *pixel* é considerado azul dada uma imagem no espaço de cor RGB. Para que um *pixel* seja identificado como azul, o componente azul do *pixel* deve ser o componente dominante, e para verificar essa condição um cálculo da distância da cor do *pixel* no espaço de cor RGB é realizado de acordo com a Equação 2 (VAN DEN BERGH; LALIOTI, 1999).

$$d = \sqrt{(B-R)^2 + (B-G)^2} > d_{max} \quad (2)$$

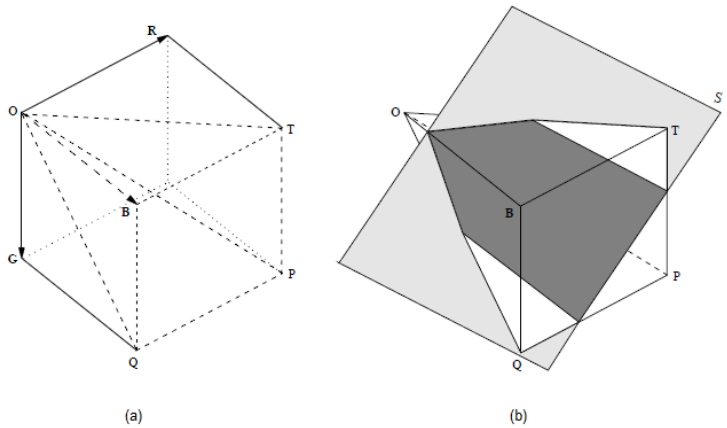
Na Equação 2, a distância *d* é calculada com os componentes *R*, *G* e *B* do *pixel*. O valor $d_{max}$ é pré-estabelecido para ser o limiar da distância. Se o valor *d* do *pixel* for maior que $d_{max}$, então o *pixel* pertence ao *background*. Caso contrário, pertence ao *foreground*.

As operações de multiplicação (potência de dois) e raiz quadrada consomem maior tempo de processamento, por isso o cálculo da distância foi simplificado nesse algoritmo como visto na Equação 3 (VAN DEN BERGH; LALIOTI, 1999).

$$d = 2 \times B - R - G \quad (3)$$

A Figura 1 ilustra a distância calculada no espaço de cor RGB. A Figura 1(a) mostra uma representação do espaço de cor RGB através de um cubo, em que cada eixo do cubo corresponde a um dos componentes (*R*, *G*, *B*). Nesse cubo, a pirâmide *OBTPQ* foi destacada com linhas tracejadas para expor o volume do cubo em que *B* ≥ *R* e *B* ≥ *G*. A Figura 1(b) identifica o valor de $d_{max}$ pré-definido pelo plano *S*. Se o valor de *d* calculado para o *pixel* ultrapassar o plano *S*, esse *pixel* faz parte do *foreground*. Caso contrário, o *pixel* faz parte do *background*.

Figura 1 - Cubo RGB




(a) Cubo RGB com pirâmide *OBTPQ*; (b) Pirâmide *OBTPQ* com plano *S*.

Fonte: VAN DEN BERGH; LALIOTI, 1999

Essa segmentação feita pelo algoritmo de Van Den Bergh e Lalioti (1999) gera um *matting* binário chamado *bitmasking*. O *bitmasking* é um caso especial de *matting* em que existem apenas duas possibilidades: o *pixel* será incluído ou não na composição final, sem possibilidade intermediária (SMITH, 1995). Essa segmentação binária faz com que o algoritmo seja executado com rapidez, porém a qualidade do resultado é comprometida. Com o *bitmasking*, as bordas do objeto no *foreground* acabam apresentando imperfeições pois nessas áreas existem *pixels* que misturam as cores do objeto em *foreground* e as cores do *background*, que deveriam ser tratados com transparência parcial.

O algoritmo de Van Den Bergh e Lalioti (1999) era o único algoritmo de *matting* digital implementado no software de estúdio virtual ARStudio antes do desenvolvimento deste trabalho.

## 2.5 *Color difference key*

Esse método de *matting* digital foi desenvolvido por Petro Vlahos (1971) e foi implementado como novo algoritmo de *matting* digital do software de estúdio virtual ARStudio neste trabalho.

Petro Vlahos foi um dos inventores que se destacou na área de *matting* da indústria cinematográfica. Ele foi o responsável por definir o problema e inventou soluções tanto em fotografia quanto em vídeo. Para a aplicação de sua técnica de *color difference key* em vídeo, ele desenvolveu o equipamento de estúdio virtual chamado *Ultimatte*, dispositivo analógico que aplicava o algoritmo do método através de hardware (SMITH; BLINN, 1996). Ele foi premiado pela Academia de Artes e Ciências Cinematográficas com o Oscar diversas vezes pelos avanços tecnológicos que ele obteve na área cinematográfica.

Esse algoritmo, assim como o *chroma-key*, utiliza uma cor chave para realizar a extração do *foreground*. A ideia central do algoritmo é determinar a transparência de cada *pixel* da imagem baseado na diferença entre os canais das cores R, G e B (SCHULTZ, 2006). Tendo selecionado a cor chave do algoritmo, uma comparação é feita entre as duas cores restantes para determinar qual delas tem maior intensidade no *pixel*. Com as duas cores selecionadas, a cor chave e a cor com maior intensidade, é calculada a diferença entre elas. A Equação 4 (SCHULTZ, 2006) apresenta o cálculo efetuado para estipular a transparência do *pixel* em uma filmagem com fundo verde,

com β representando a transparência e R, G e B representando os canais das respectivas cores. O resultado do cálculo é limitado pelo valor mínimo de transparência, ou seja, se o resultado for inferior ao valor mínimo de transparência, o valor de transparência atribuído ao *pixel* será o valor mínimo.

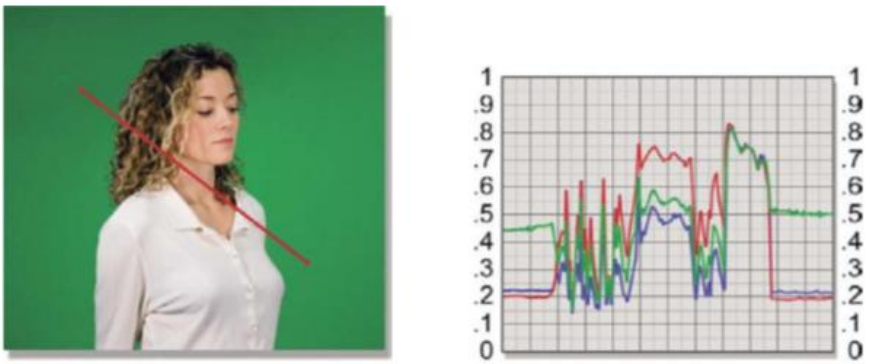
$$\beta = G - \max(R, B) \tag{4}$$

Tendo o valor da transparência deve-se calcular o valor de opacidade do canal *alpha*, que é o complemento da transparência. Para isso, considerando que o intervalo do valor de transparência e opacidade seja de 0 a 1, o valor de opacidade α é calculado de acordo com a Equação 5.

$$\alpha = 1 - \beta \tag{5}$$

Efetuando o cálculo da transparência no caso da filmagem com fundo verde, quanto maior a intensidade da cor verde do *pixel*, maior é sua transparência. Para um *pixel* verde puro, ou seja, um *pixel* com valor máximo no canal G e valores mínimos nos canais R e B, o valor de transparência será máximo, indicando que o *pixel* é completamente transparente. A Figura 2 apresenta uma imagem com fundo verde, e um gráfico com os valores dos canais dos *pixels* no decorrer de uma linha vermelha marcada na imagem. Nela é possível notar que a diferença entre o canal verde e o maior entre os outros dois canais é maior na parte verde da imagem, e próximo a zero ou negativo no restante da imagem.

Figura 2 - Imagem com fundo verde e gráfico representando a intensidade das cores através da marcação vermelha na imagem




Fonte: WRIGHT, 2010

O valor de opacidade do canal *alpha* resultante da aplicação do algoritmo ainda precisa sofrer um processo de escala, para que os valores mínimo e máximo de opacidade resultantes sejam transformados nos valores mínimo e máximo absolutos da escala de opacidade (WRIGHT, 2010).

O algoritmo de *color difference key* foi alcançado através de anos de experiência na área de *matting* e de resultados de experimentos, ele não se baseia em uma abordagem abstrata, não sendo matematicamente válido (SMITH; BLINN, 1996; CHUANG, 2004). Apesar disso, é um algoritmo eficiente que aplica transparência de *pixels* e que consegue ser executado em tempo real utilizando computadores atuais.

## 2.6 Realidade aumentada

Realidade aumentada (RA) é uma tecnologia emergente que surgiu nos anos 90. Ela usa o mundo virtual para ampliar a percepção das pessoas do mundo real combinando informações virtuais geradas por computador com o ambiente do mundo real capturado por um dispositivo como, por exemplo, uma câmera filmadora. A tecnologia de RA tem três principais características: combinação entre real e virtual, registro de objetos virtuais e interação em tempo real (LI; QI; WU, 2012).

Alguns aspectos da definição de realidade aumentada são importantes de serem mencionados. O primeiro deles é que a tecnologia não é restrita a aparelhos de *display* como o HMD (*Head-Mounted Display*, dispositivo de vídeo usado na cabeça como um capacete que permite ao usuário visualizar objetos virtuais). Além disso, ela não é limitada ao sentido da visão, mas pode também ser aplicada a todos os outros sentidos, como audição, tato e olfato. E finalmente, remover objetos reais de um ambiente sobrepondo objetos virtuais sobre eles (abordagem conhecida como realidade diminuída) também é considerada realidade aumentada.

O registro de objetos virtuais é um dos principais componentes de RA. Registro é definido como o alinhamento preciso de objetos reais e virtuais. Sem o registro preciso, a ilusão de que o objeto virtual existe no ambiente real é comprometida (AZUMA et al., 2001). Para isso, foram criadas algumas técnicas para realizar o registro: técnicas baseadas em padrões e técnicas baseadas em características naturais. A primeira adota o uso de marcadores (símbolos que apresentam um padrão) que são rastreados para efetuar o registro, enquanto a segunda analisa

características naturais do cenário no mundo real para que, uma vez posicionado o objeto virtual, seja possível realizar seu registro observando a mudança de posicionamento dessas características. A Figura 3 mostra um marcador usado na técnica baseada em padrões e sua utilização para registrar um objeto em realidade aumentada. O ambiente do mundo real é rastreado em busca do padrão do marcador e, quando encontrado, sua posição e orientação é calculada, permitindo assim alinhar um objeto virtual sobre o marcador no mundo real.

Figura 3 - Marcador e sua utilização no registro de um objeto virtual

Fonte: KATO et al., 2000

A tecnologia baseada em padrões é mais antiga e se encontra em um estágio maduro de desenvolvimento (GEIGER; SCHMIDT; STOCKLEIN, 2004), porém existem problemas que ainda não foram completamente solucionados, como o da obstrução do padrão. Quando o padrão é obstruído por outro objeto, seu reconhecimento é dificultado. Alternativas foram criadas para amenizar o problema, como o uso de multimarcadores. Através da combinação de múltiplos marcadores, um multimarcador consegue continuar sendo identificado mesmo se uma porção de seu conjunto de marcadores esteja obstruído ou fora do campo de visão da câmera. A Figura 4 mostra uma aplicação de um multimarcador: o posicionamento do objeto não é feito apenas com um dos marcadores, mas todos estão sendo usados em conjunto para determinar a posição do objeto.

Figura 4 - Multimarcador sendo utilizado

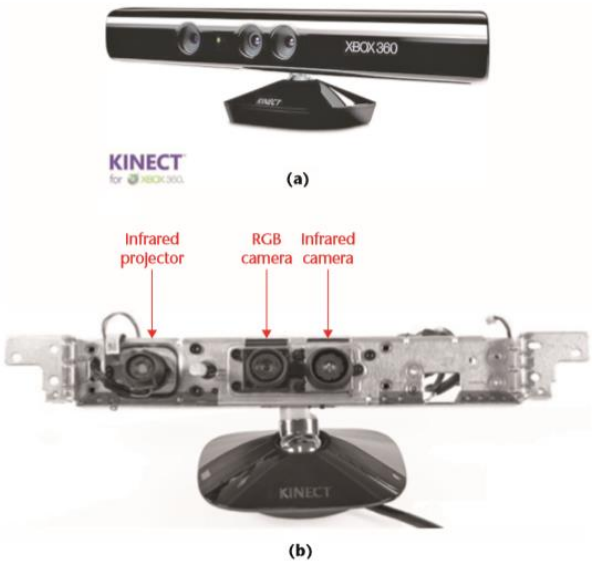
Fonte: ANAGNOSTOU; VLAMOS, 2011

A área de realidade aumentada apresentou crescimento e progresso notáveis nos últimos anos. Hoje, a área conta com diversas conferências especializadas, e alguns consórcios de pesquisa interdisciplinares são financiados para desenvolver a tecnologia, como o *Mixed Reality Systems Laboratory* no Japão e o *Project ARVIKA* na Alemanha.

## 2.7 Microsoft Kinect

O Kinect é um dispositivo criado pela empresa israelita PrimeSense que dispõe de uma câmera RGB (câmera colorida), um sensor de profundidade e um conjunto de quatro microfones integrado. Com esses componentes, o Kinect fornece reconhecimento facial, captura de movimento corporal em 3D e reconhecimento de voz com remoção do som ambiente (MICROSOFT, 2010). A Figura 5 mostra o exterior e o interior do Kinect, indicando os principais componentes do dispositivo.

Figura 5 - Dispositivo Kinect

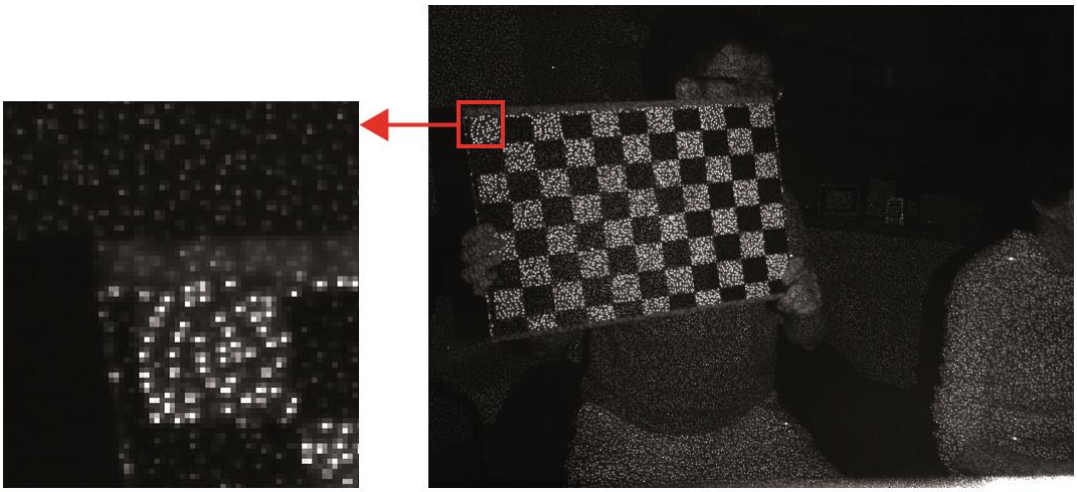



(a) Parte externa do Kinect; (b) Interior do Kinect e seus principais componentes.

Fonte: ZHANG, 2012

O sensor de profundidade consiste de um projetor IR (*infrared* ou infravermelho) combinado com uma câmera IR, que é um sensor semicondutor metal-óxido complementar (CMOS), com tecnologia baseada no princípio de luz estruturada. O projetor IR é um laser IR que passa por uma grade de difração e se transforma em um conjunto de pontos IR (ZHANG, 2012). A Figura 6 mostra os pontos IR capturados pela câmera IR.

Figura 6 - Imagem capturada pela câmera IR do Kinect, com pontos IR



Fonte: ZHANG, 2012

Com a imagem capturada pela câmera IR com os pontos IR, é possível obter os pontos da imagem em 3D usando triangulação, já que a geometria relativa entre o projetor IR, a câmera IR e o padrão de pontos IR projetados é conhecida. Assim, é possível obter o valor de profundidade de cada ponto da imagem, criando um mapa de profundidade. A Figura 7 mostra um mapa de profundidade, em que os valores de profundidade codificados como tons de cinza; quanto mais escuro o *pixel*, mais próximo da câmera está o ponto. *Pixels* pretos indicam que não há um valor de profundidade disponível para esses pixels. Isso pode acontecer se os pontos estiverem muito longe (profundidade não pode ser calculada com precisão), se os pontos estiverem muito perto (criam-se pontos cegos devido ao campo de visão limitado do projetor e da câmera), se os pontos estiverem na sombra do projetor (não há pontos IR), ou se a luz IR não for refletida corretamente (em superfícies espelhadas, por exemplo) (ZHANG, 2012).

Figura 7 - Representação do mapa de profundidade em tons de cinza

Fonte: ZHANG, 2012

O Kinect apresenta diversas aplicações na área de computação, e foi utilizado por esse trabalho para realizar o registro de objetos virtuais em relação aos atores, permitindo ao software de estúdio virtual determinar se o objeto virtual está à frente ou atrás dos atores no mundo real.

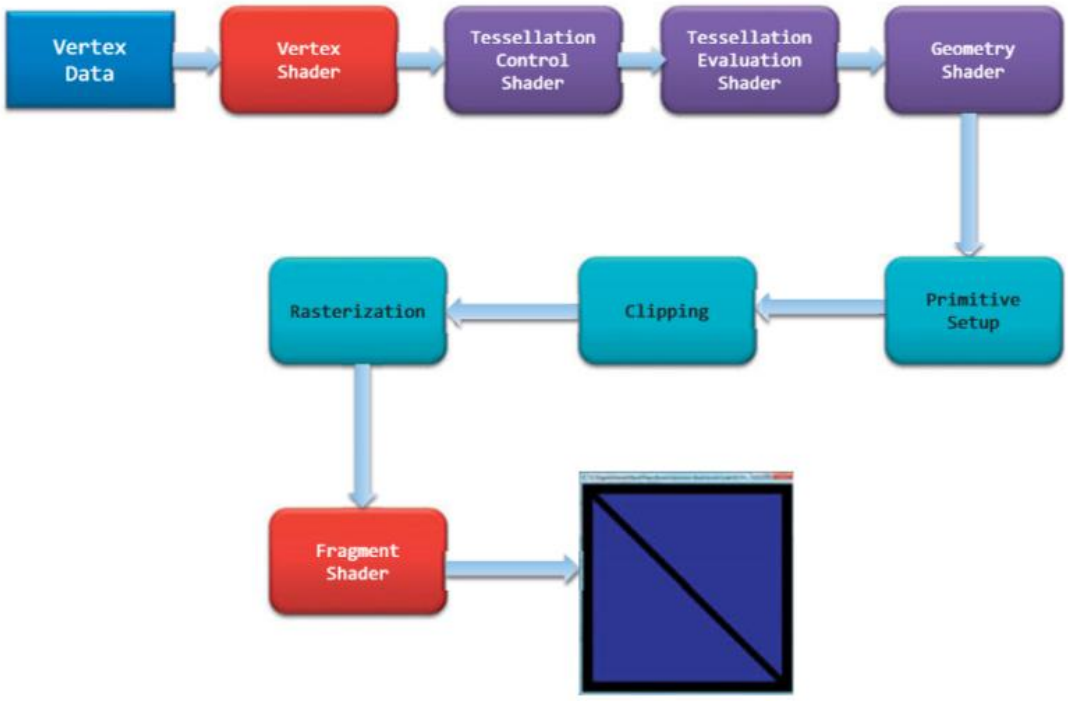
## 2.8 Programação de *shaders* no OpenGL

OpenGL é uma API (*Application Programming Interface* ou Interface de Programação de Aplicativos) gráfica, ou seja, ela é uma biblioteca que permite acesso

a recursos do hardware gráfico. Ela contém diversos comandos usados para especificar objetos, imagens e operações necessárias para produzir aplicações interativas com gráficos tridimensionais. Desenvolvida pela Silicon Graphics Computer Systems em 1994, ela é uma interface independente de hardware e de sistema operacional, ou seja, as aplicações criadas com ela podem ser usadas em computadores com qualquer placa de vídeo de qualquer fabricante e com qualquer sistema operacional, sem que mudanças precisem ser realizadas na aplicação.

O OpenGL constrói modelos 3D através das formas geométricas mais simples que existem, chamadas primitivas geométricas (pontos, linhas e triângulos). Especificando os vértices dessas primitivas geométricas, é possível criar os modelos e transformá-los em imagens através de um processo chamado renderização. Para realizar a renderização, uma linha de montagem (*pipeline*) é criada, ou seja, uma sequência de estágios é percorrida para converter os dados da aplicação em uma imagem final renderizada (SHREINER et al., 2013). A Figura 8 ilustra esse *pipeline* do OpenGL.

Figura 8 - *Pipeline* do OpenGL




Fonte: SHREINER et al., 2013

Dentro desse *pipeline*, existem funções especiais executadas pelo hardware gráfico, chamadas *shaders*. Essas funções são como programas especificamente compilados para GPU (*Graphics Processing Unit* ou Unidade de Processamento Gráfico) utilizando uma linguagem própria do OpenGL chamada GLSL (*OpenGL Shading Language*). O OpenGL pega os código-fontes dos *shaders* criados pelo programador na linguagem GLSL e os compila, criado o código que a GPU precisa para executar. Existem quatro estágios de *shader* que podem ser executados: *vertex shader*, *tesselation shader*, *geometry shader* e *fragment shader*. Os *shaders* comumente mais utilizados são o *vertex shader* e o *fragment shader*.

Para cada vértice que é emitido pelo comando de desenho, um *vertex shader* será chamado para processar os dados associados ao vértice. Esse *shader* efetuará cálculos para computar a posição do vértice na tela, determinando a cor do vértice utilizando cálculos de iluminação, entre outras técnicas. Já o *fragment shader*, estágio final do *pipeline*, processa os fragmentos gerados pela etapa de rasterização (transformação das formas geométricas dos modelos 3D em pixels). Com ele, é possível manipular a cor de um fragmento, além de modificar o valor de profundidade do fragmento (SHREINER et al., 2013).

No software de estúdio virtual desenvolvido neste trabalho, um *fragment shader* foi programado para auxiliar na realização do registro de objetos virtuais em relação aos atores.

## 2.9 Sistema de estúdio virtual ARStudio

O sistema de estúdio virtual estudado nesse trabalho é o ARStudio, desenvolvido por Campos et al. (2010). Ele foi desenvolvido em uma arquitetura composta por módulos, para facilitar a manutenção e reutilização do código. Os cinco módulos são: captura, *chroma-key*, interface com o usuário, detecção de marcadores e gerador de cena combinada (CAMPOS et al., 2010).

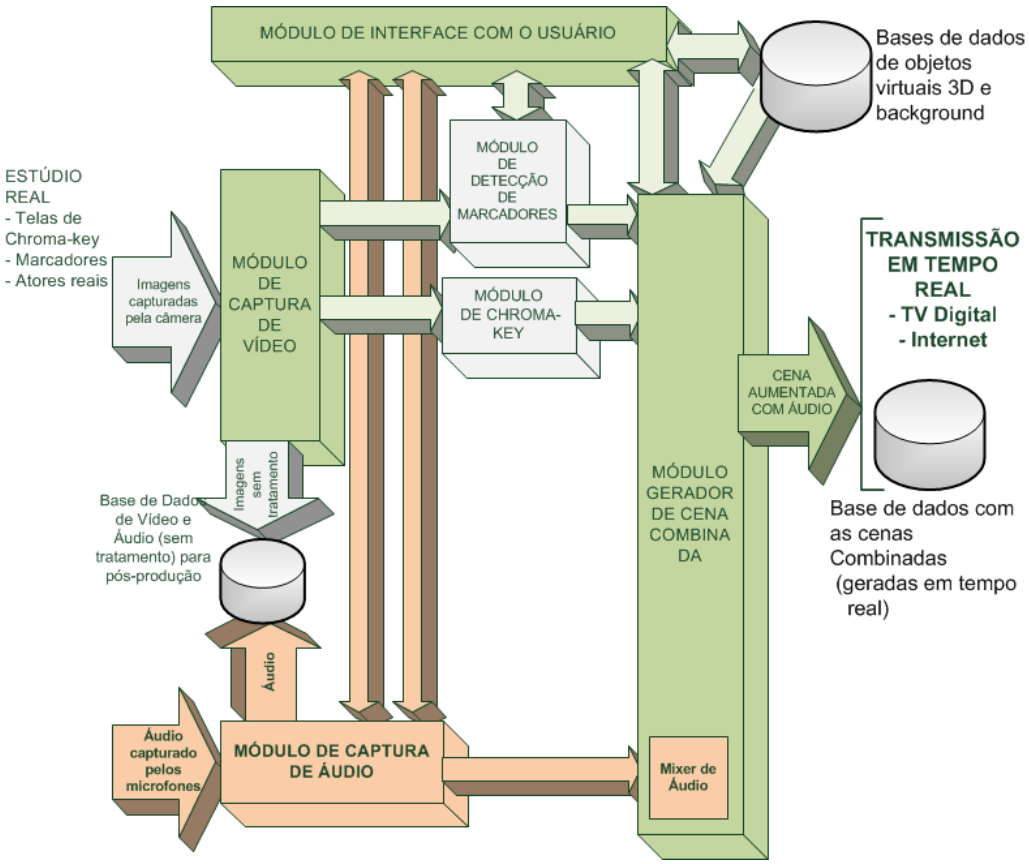
Segundo Campos et al. (2010), as funções dos módulos são:

- Módulo de captura: obtenção de imagens a partir de uma câmera e seu armazenamento, num formato apropriado, em um *buffer* para posterior processamento;

- Módulo de *chroma-key*: análise das imagens, *pixel* a *pixel*, e obtenção de uma imagem apenas com os elementos de interesse (sem o fundo de cor uniforme);
- Módulo de interface com o usuário: permitir que o usuário configure o ambiente do estúdio virtual;
- Módulo de detecção de marcadores: reconhecimento de padrões (marcadores) na cena e geração de um mapa de profundidade;
- Módulo de geração de cena combinada: uso do mapa de profundidade para a construção da cena final.

A Figura 9 apresenta um diagrama com os módulos do sistema.

Figura 9 - Módulos do sistema ARStudio




Fonte: FERNANDES, 2011

As funcionalidades relativas a interface com o usuário, captura de marcadores e geração de cena aumentada foram aperfeiçoadas, além da adição de serviços de captura de áudio e sincronização com vídeo por Fernandes (2011).

# 3 DESENVOLVIMENTO

O desenvolvimento do trabalho foi feito em etapas, cada uma visando alcançar um dos objetivos definidos para o mesmo. Além dessas etapas, foi necessária uma etapa inicial para recriar o ambiente de desenvolvimento do software de estúdio virtual. Incluindo a etapa inicial, elas totalizam 6 etapas. Essas etapas foram:

- Reconstrução do ambiente de desenvolvimento;
- Implementação de armazenamento de vídeo não processado;
- Implementação de marcadores fixos;
- Implementação de multimarcadores;
- Implementação de algoritmo de *matting* digital aprimorado;
- Implementação de registro de objetos virtuais com uso do Kinect.

Estas etapas são descritas nas seções seguintes.

## 3.1 Reconstrução do ambiente de desenvolvimento

Por se tratar de um software que já estava em desenvolvimento antes do início deste trabalho, foi necessário reconstruir o ambiente de desenvolvimento original. Para isso, todas as bibliotecas utilizadas no projeto tiveram que ser instaladas, a fim de permitir a modificação, compilação e ligação do código fonte do sistema.

A etapa de reconstrução do ambiente de desenvolvimento também foi usada na aprendizagem do funcionamento do software ARStudio e das bibliotecas utilizadas por ele, pois foi necessário estudar a estrutura do software e a função de cada uma das bibliotecas dentro dessa estrutura.

Esta seção apresenta as bibliotecas instaladas, os papéis desempenhados por cada uma delas dentro do software ARStudio e o processo realizado para a instalação dessas bibliotecas.

As bibliotecas utilizadas pelo projeto original do ARStudio foram:

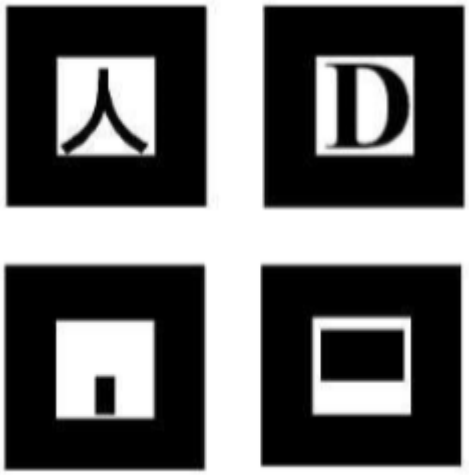
- ARToolKit;
- OpenSceneGraph;
- OSGART;
- OpenCV;
- Qt Framework;
- FMOD Ex API.

### 3.1.1 ARToolKit

ARToolKit é uma biblioteca que facilita aos programadores o desenvolvimento de aplicações de realidade aumentada. Ela usa técnicas de visão computacional para calcular a posição da câmera real e a orientação relativa dos marcadores, permitindo a sobreposição de objetos virtuais sobre esses marcadores (KATO et al., 2000).

Desenvolvida por Hirokazu Kato da Universidade de Osaka e apoiada pela Universidade de Washington, a biblioteca utiliza marcadores que consistem de uma borda quadrada preta com diferentes padrões em seu interior. Alguns exemplos de marcadores podem ser vistos na Figura 10. Multimarcadores também são suportados pela biblioteca.

Figura 10 - Exemplos de quatro marcadores do ARToolKit



Fonte: FIALA, 2004

Seu funcionamento é baseado no grande contraste entre a parte branca e a parte preta do marcador. A imagem do mundo real capturada pela biblioteca é binarizada, ou seja, transformada em uma imagem branco e preto (sem tons de cinza), para evidenciar o contorno das bordas quadradas dos marcadores e permitir sua identificação. Para identificar o marcador, a biblioteca precisa de um arquivo contendo o padrão do marcador utilizado (FIALA, 2004).

Após o ARToolKit rastrear o padrão do marcador e identifica-lo, sua posição em relação à câmera é determinada. Para renderizar o objeto virtual nas coordenadas fornecidas pela biblioteca é necessário a utilização de uma API de gráficos 3D como

OpenGL ou DirectX, e toda a manipulação dos objetos virtuais deve ser feita através dessa API.

No software ARStudio, a biblioteca ARToolKit é responsável pelo rastreamento dos marcadores e fornecimento da posição dos mesmos. Ela também faz o gerenciamento do fluxo de vídeo capturado pela câmera.

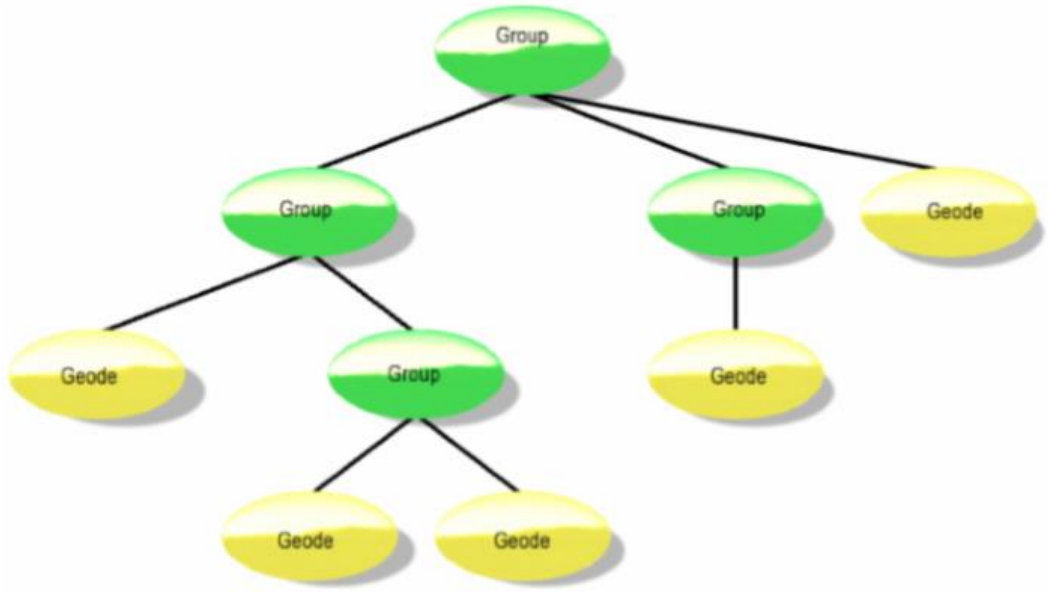
### 3.1.2 OpenSceneGraph

OpenSceneGraph é uma biblioteca *middleware* de renderização, ou seja, ela eleva o nível de abstração e ameniza a complexidade do uso da API de renderização OpenGL, que é uma API que utiliza comandos mais próximos aos comandos de baixo nível (WANG; QIAN, 2010).

Usando OpenGL é necessário interagir com a GPU através da manipulação de matrizes e da definição de extensões, operações muito próximas aos comandos utilizados pelo sistema operacional para controlar o hardware. A biblioteca OpenSceneGraph se encarrega de realizar essa interação com a GPU através do OpenGL, facilitando assim a programação da aplicação gráfica.

Outra característica importante do OpenSceneGraph é seu sistema baseado em grafos de cena. Um grafo de cena é uma estrutura de dados que define a relação espacial e lógica de uma cena gráfica a fim de obter maior eficiência no gerenciamento e na renderização dos dados gráficos. O grafo de cena é representado como um grafo hierárquico, que contém uma coleção de nós gráficos com um nó raiz no topo do grafo, nós grupo que podem ter nós filhos, e nós folha que não possuem filhos e que se posicionam na base do grafo. Esses nós estão relacionados de forma que os nós filhos compartilham as informações de seus nós pais, fazendo com que operações realizadas nos nós pais tenham seus efeitos propagados para todos seus nós filhos (WANG; QIAN, 2010). A Figura 11 apresenta um exemplo de um grafo de cena, com nós grupo e nós folha (nesse caso em particular, *Geodes* ou *Geometry Nodes*).

Figura 11 - Exemplo de um grafo de cena




Fonte: MARTZ, 2007

O OpenSceneGraph é responsável pela criação do grafo de cena e da manipulação dos objetos virtuais dentro do software ARStudio.

### 3.1.3 OSGART

A biblioteca OpenSceneGraph oferece maior facilidade de programação de aplicações com renderização de objetos virtuais aliada a uma estrutura de grafo de cena que facilita a manipulação desses objetos na cena, porém não oferece suporte às funcionalidades básicas de realidade aumentada (entrada de fluxo de vídeo através de câmera, rastreamento de marcador, posicionamento de objeto virtual no mundo real). Enquanto isso, a biblioteca ARToolKit fornece os recursos necessários para a criação de uma aplicação de realidade aumentada, porém depende de APIs de renderização (como OpenGL ou DirectX) que apresentam uma maior complexidade de programação. Para se beneficiar dos recursos de ambas as bibliotecas ao mesmo tempo, a biblioteca OSGART foi criada para integrar ARToolKit e OpenSceneGraph.

A biblioteca OSGART foi desenvolvida como uma extensão da biblioteca OpenSceneGraph. Ela implementa uma abordagem hierárquica baseada no grafo de cena utilizando os marcadores de realidade aumentada através do ARToolKit, criando uma transição entre ambientes virtuais imersivos e realidade aumentada (LOOSER et al., 2006).

As classes implementadas na biblioteca OSGART adaptam as principais classes da biblioteca ARToolKit (como controle de fluxo de entrada de vídeo pela câmera, rastreamento de marcador e registro de objetos virtuais) para que sejam inseridas como nós de um grafo de cena através das classes da biblioteca OpenSceneGraph. Dessa forma, todos os recursos do ARToolKit são integrados ao grafo de cena, e manipulados através do OpenSceneGraph.

No software de estúdio virtual ARStudio, a biblioteca OSGART cumpre esse papel de integrar ARToolKit e OpenSceneGraph.

### 3.1.4 OpenCV

OpenCV é uma biblioteca de código aberto desenvolvida pela iniciativa de pesquisa da Intel, e seu objetivo original era de criar avanços na área de aplicações orientadas a uso intensivo de CPU. Atualmente ela é uma biblioteca de visão computacional com foco em aplicações em tempo real, e para isso foi desenvolvido com código altamente otimizado. Seu objetivo é fornecer uma infraestrutura de visão computacional simples de usar, contendo centenas de algoritmos implementados (BRADSKI; KAEHLER, 2008). O OpenCV contém módulos de processamento de imagens, análise de vídeo (estimativa de movimento, rastreamento de objetos), calibração de câmera estéreo (reconstrução 3D), detecção de objetos (reconhecimento facial), captura de vídeo, entre outros.

A principal função da biblioteca OpenCV dentro do ARStudio é o processamento de imagens.

### 3.1.5 Qt Framework

Qt é um *framework* de GUI (*Graphical User Interface* ou Interface Gráfica do Usuário) que inclui um vasto conjunto de *widgets*, APIs de renderização gráfica, mecanismos de *layout* e folha de estilo, e ferramentas que podem ser usadas para criar interfaces de usuário mais elaboradas. A variedade de *widgets* vai desde objetos simples, como botões e rótulos, a *widgets* avançados como editores completos de texto, calendários e objetos com navegadores *web* completos (MIKKONEN; TAIVALSAARI; TERHO, 2009).

O Qt Framework foi utilizado no ARStudio para auxiliar na criação da interface do usuário.

### 3.1.6 FMOD Ex API

A FMOD Ex API faz parte de um conjunto de ferramentas para criação de conteúdo de áudio desenvolvido pela Firefly Technologies. Ela é uma API de baixo nível, contendo comandos primitivos e recursos que auxiliam a manipulação de áudio, como *mixers*, módulos de saída da interface de hardware, recursos de som 3D, entre outros.

No estúdio virtual ARStudio, essa biblioteca é utilizada no módulo de áudio, que permite a captação e edição do áudio.

### 3.1.7 Processo de reconstrução

Antes de instalar as bibliotecas, foi necessário instalar as ferramentas e os SDKs (*Software Development Kit* ou Kit de Desenvolvimento de Software) necessários.

O projeto do software de estúdio virtual ARStudio foi desenvolvido utilizando a IDE (*Integrated Development Environment* ou Ambiente Integrado de Desenvolvimento) Microsoft Visual Studio 2010, por isso foi necessária a sua instalação. Além da IDE, foi necessária a instalação dos Microsoft SDK 6.1 e 7.1, que são pré-requisitos de algumas das bibliotecas usadas pelo ARStudio.

Com exceção da FMOD Ex API, que é uma API proprietária (livre para uso não comercial), as bibliotecas necessárias para a criação do ambiente de desenvolvimento possuem código aberto. Assim, elas são distribuídas pelo código fonte, devendo ser compilada para a utilização.

O projeto do software ARStudio anterior a este trabalho estava funcionando com versões antigas das bibliotecas, e neste trabalho as versões dessas bibliotecas foram atualizadas para corrigir possíveis falhas e se beneficiar de possíveis aperfeiçoamentos implementados nelas nas versões mais atuais. A versões das bibliotecas utilizadas neste trabalho foram:

- ARToolKit: permaneceu na versão 2.72.1;
- OpenSceneGraph: atualizada da versão 3.0.1 para a versão 3.2.1 RC1;
- OSGART: permaneceu na versão 2.0 RC3;
- OpenCV: atualizada da versão 2.2.0 para a versão 2.4.8;
- Qt Framework: atualizada da versão 4.7.3 para a versão 4.8.5;
- FMOD Ex API: atualizada da versão 4.36.02 para a versão 4.44.26.

O Qt Framework possui um aplicativo próprio para gerar o *workspace* da biblioteca, e após utilizá-lo, o código fonte foi compilado utilizando o prompt de comando do Visual Studio 2010. Um suplemento do Qt para o Visual Studio 2010 também foi instalado; esse suplemento permite a integração da biblioteca com a IDE, assim quando o projeto do software em desenvolvimento é compilado na IDE os recursos do projeto criados no Qt também são compilados automaticamente.

O CMake, ferramenta para a geração automatizada do *workspace* escolhido através de arquivos de configuração, foi utilizado para gerar os projetos das bibliotecas restantes a serem compilados no Visual Studio 2010.

Após todas as bibliotecas terem sido compiladas e instaladas, o projeto do ARStudio foi configurado no Visual Studio 2010. Devido à atualização de algumas das bibliotecas, algumas adaptações no código tiveram que ser feitas para adequar o software às novas versões das bibliotecas, como mudanças nos nomes das classes ou métodos, variações na ordem ou na quantidade de parâmetros de um método, troca do nome de arquivos de cabeçalho, entre outras modificações.

Na configuração do projeto do ARStudio no Visual Studio 2010, variáveis de ambiente foram adicionadas para facilitar a configuração em futuras ocasiões, em outros computadores. Assim, não é necessário alterar a configuração do projeto, basta adicionar as variáveis de ambiente no novo computador em que o ambiente será instalado.

Com a configuração do projeto concluída, a etapa de reconstrução do ambiente de desenvolvimento foi finalizada; o código fonte estava pronto para ser modificado e compilado.

## 3.2 Implementação de armazenamento de vídeo não processado

O objetivo dessa etapa era implementar o armazenamento de vídeo de entrada antes de seu processamento, funcionalidade que não estava presente até então no software ARStudio. Para implementar o armazenamento do vídeo não processado foi necessário entender como o vídeo processado é armazenado pelo ARStudio.

Quando o software é iniciado, um grafo de cena que representará a cena virtual junto com os elementos de realidade aumentada é construído. Nesse grafo de cena, o OSGART, utilizando funcionalidades do ARToolKit e do OpenSceneGraph, se

encarrega de inserir a imagem obtida da câmera do mundo real no grafo de cena assim que o fluxo de vídeo é iniciado.

Para poder visualizar esse grafo de cena, uma câmera virtual é criada e o grafo de cena é definido como sendo a cena virtual observada por ela. Essa câmera virtual representa a visão que usuário tem da cena virtual, e mostra ao usuário o resultado de todas as manipulações feitas pelo estúdio virtual, como a inserção de objetos virtuais e aplicação de técnicas de *matting* digital. Durante a criação dessa câmera virtual, uma classe de *callback* (retorno de chamada) de desenho personalizada é vinculada à ela. Essa classe possui um método virtual que pode ser sobrescrito pelo programador, e toda vez que os elementos do grafo de cena são desenhados esse método é chamado (após a operação de desenho ter sido finalizada, no retorno da chamada do método de desenho).

Na classe de *callback* de desenho personalizada para o ARStudio, esse método virtual foi sobrescrito para que a cada vez que esse método seja chamado a imagem desenhada pela câmera virtual seja obtida e armazenada em um arquivo de vídeo utilizando as funcionalidades do OpenCV. Nesse método virtual, uma variável de controle é utilizada para saber se o usuário ativou a opção de gravação de vídeo, e a imagem só é armazenada no arquivo de vídeo se essa opção tiver sido ativada.

Pela interface gráfica, o usuário pode iniciar ou parar a gravação de vídeo através de botões. Quando a gravação de vídeo é iniciada, o arquivo de vídeo é criado para que as imagens sejam armazenadas nele, e quando a gravação é interrompida o arquivo de vídeo é fechado.

### 3.2.1 Implementação sequêncial

Uma implementação inicial para o armazenamento do vídeo não processado seguia o mesmo processo de gravação do vídeo processado que já estava presente no ARStudio, mas além de obter a imagem desenhada pela câmera virtual (imagem processada) a imagem capturada pela câmera real (imagem não processada) também era obtida. Assim, a cada vez que o método virtual sobrescrito era executado, ambas as imagens eram obtidas e armazenadas em arquivos de vídeo separados. Também, variáveis de controle eram utilizadas para saber qual dos fluxos deveriam ser armazenados de acordo com a opção do usuário.

Figura 12 - Código de início e parada de gravação

```
void VideoRecord::Start(bool gravarVideoNaoProcessado, std::string nomeArquivoNaoProcessado,
                        bool gravarVideoProcessado, std::string nomeArquivoProcessado, double taxaFPS)
{
    if (gravarVideoNaoProcessado)
    {
        this->gravarVideoNaoProcessado = true;
        writer1 = cvCreateVideoWriter(nomeArquivoNaoProcessado.c_str(), -1, taxaFPS,
                                      cvSize(video->getVideoPlugin().get()->s(),
                                      video->getVideoPlugin().get()->t()), 1);
    }
    if (gravarVideoProcessado)
    {
        this->gravarVideoProcessado = true;
        writer2 = cvCreateVideoWriter(nomeArquivoProcessado.c_str(), -1, taxaFPS,
                                      cvSize(Definicoes::getLarguraVideo(),
                                      Definicoes::getAlturaVideo()), 1);
    }
}

void VideoRecord::Stop(void)
{
    if (gravarVideoNaoProcessado)
    {
        gravarVideoNaoProcessado = false;
        cvReleaseVideoWriter(&writer1);
    }
    if (gravarVideoProcessado)
    {
        gravarVideoProcessado = false;
        cvReleaseVideoWriter(&writer2);
    }
}
```

Fonte: elaborada pelo autor

A Figura 12 apresenta o código dos métodos de início de gravação (VideoRecord::Start) e parada de gravação (VideoRecord::Stop) com essa implementação de armazenamento do vídeo não processado. Antes dessa implementação, somente um arquivo de vídeo era criado no início da gravação, e com a implementação mais um arquivo de vídeo é gerado. Na parada de gravação, o fechamento desse novo arquivo de vídeo gerado também foi adicionado ao código.

Figura 13 - Código do método virtual sobrescrito (armazenamento de vídeo)

```
virtual void operator () (osg::RenderInfo& renderInfo) const
{
    if (gravarVideoNaoProcessado)
    {
        osg::ref_ptr<osg::Image> img1 = video->getVideoPlugin().get();
        IplImage * imgOpenCV1 = VideoUtils::osgImage2IplImage(img1, 4);
        cvWriteFrame(writer1, imgOpenCV1);
        cvReleaseImage(&imgOpenCV1);
    }

    if (gravarVideoProcessado)
    {
        osg::ref_ptr<osg::Image> img2 = new osg::Image();
        img2->readPixels(0, 0, Definicoes::getLarguraVideo(),
                         Definicoes::getAlturaVideo(), GL_BGR, GL_UNSIGNED_BYTE);
        img2->flipVertical();
        IplImage * imgOpenCV2 = VideoUtils::osgImage2IplImage(img2);
        cvWriteFrame(writer2, imgOpenCV2);
        cvReleaseImage(&imgOpenCV2);
    }
}
```

Fonte: elaborada pelo autor

O método virtual sobrescrito da classe de *callback* de desenho está representado na Figura 13. As duas imagens, processada e não processada, são obtidas e armazenadas, cada uma em seu próprio arquivo de vídeo criado no início da gravação.

Todo o processo de gravação é feito de forma sequencial, ou seja, uma etapa precisa ser finalizada antes que a outra seja iniciada. Assim, as duas imagens precisam ser obtidas e armazenadas em disco para que o método virtual sobrescrito retorne, e só então uma nova chamada ao método de desenho da câmera virtual será feita e um novo *frame* da cena virtual será desenhado.

O armazenamento das imagens em disco é um processo demorado, e por isso a nova chamada ao método de desenho acaba sendo atrasada, causando uma perda de *frames*. Mesmo antes dessa implementação, armazenando apenas um fluxo de vídeo, uma perda era sentida. Realizando o armazenamento dos dois fluxos ao mesmo tempo, essa perda foi acentuada. Para solucionar esse problema, uma outra implementação foi realizada com o uso de *threads*.

### 3.2.2 Implementação com uso de *threads*

Para a implementação com uso de *threads*, duas *threads* foram criadas para cada um dos fluxos de vídeo: uma *thread* armazena as imagens na memória utilizando

uma estrutura de fila e outra grava as imagens da fila em disco. A estrutura de fila foi utilizada para manter a sequência das imagens, uma vez que a fila opera no contexto FIFO (*First In First Out*), ou seja, os elementos são inseridos no final da fila e retirados do começo dela.

As *threads* são criadas no início da gravação e a cada chamada do método virtual sobrescrito um evento é disparado para sinalizar à *thread* que armazena as imagens na fila de que um novo *frame* foi desenhado e deve ser armazenado na fila. A fila tem seu tamanho limitado, para evitar que sobrecarregue a memória. Quando a gravação é interrompida, outro evento é disparado para sinalizar às *threads* que a gravação foi finalizada e elas devem ser terminadas. Após receber o sinal do evento, a *thread* que grava as imagens em disco terminar de gravar as imagens que ainda restam na fila.

Essa solução com o uso de *threads* caracteriza um sistema produtor-consumidor, pois enquanto uma *thread* produz dados e os insere em um *buffer* (*thread* produtora), concorrentemente a outra *thread* consome os dados desse *buffer*, gravando-os em disco (*thread* consumidora). Para evitar que produtor e consumidor tentem modificar os ponteiros da fila ou uma imagem ao mesmo tempo, a exclusão mútua é necessária. Por isso, uma variável *mutex* foi utilizada para cada fluxo.

A Figura 14 mostra o código da *thread* que grava as imagens do vídeo não processado na fila (VideoRecord::criaFilaVideoNaoProcessado) e da *thread* que armazena as imagens não processadas no arquivo de vídeo em disco (VideoRecord::salvaFilaVideoNaoProcessado).

Figura 14 - Código das duas *threads* do fluxo de vídeo não processado

```
DWORD WINAPI VideoRecord::criaFilaVideoNaoProcessado()
{
    const std::queue<IplImage *>::size_type maxQueue = MAX_QUEUE;
    do {
        if ((WaitForSingleObject(eventoCapturarVideoNaoProcessado, 0) == WAIT_OBJECT_0)
        && (filaVideoNaoProcessado.size() < maxQueue)) {
            osg::ref_ptr<osg::Image> img1 = video->getVideoPlugin().get();
            IplImage * imgOpenCV1 = VideoUtils::osgImage2IplImage(img1, 4);
            IplImage * imgOpenCV2 = cvCloneImage(imgOpenCV1);
            if (WaitForSingleObject(handleMutexVideoNaoProcessado, INFINITE) == WAIT_OBJECT_0) {
                filaVideoNaoProcessado.push(imgOpenCV2);
                ReleaseMutex(handleMutexVideoNaoProcessado);
            }
            cvReleaseImage(&imgOpenCV1);
        }
    } while (WaitForSingleObject(eventoPararThreadVideoNaoProcessado1, 0) != WAIT_OBJECT_0);
    return 0;
}

DWORD WINAPI VideoRecord::salvaFilaVideoNaoProcessado()
{
    bool parar = false;

    do {
        if ((filaVideoNaoProcessado.size()) > 0
        && (WaitForSingleObject(handleMutexVideoNaoProcessado, INFINITE) == WAIT_OBJECT_0)) {
            cvWriteFrame(writer1,filaVideoNaoProcessado.front());
            cvReleaseImage(&filaVideoNaoProcessado.front());
            filaVideoNaoProcessado.pop();
            ReleaseMutex(handleMutexVideoNaoProcessado);
        }
        if (WaitForSingleObject(eventoPararThreadVideoNaoProcessado2, 0) == WAIT_OBJECT_0)
            parar = true;
    } while (!parar || filaVideoNaoProcessado.size() > 0);

    cvReleaseVideoWriter(&writer1);
    return 0;
}
```

Fonte: elaborada pelo autor

## 3.3 Implementação de marcadores fixos

O intuito desta etapa era melhorar o registro dos objetos virtuais quando os marcadores utilizados tivessem que permanecer parados na mesma posição durante toda a gravação (sem movimentação de câmera). A ideia era fixar o objeto na posição em que marcador foi encontrado pela primeira vez, ignorando as posições obtidas pelo rastreamento nas vezes seguintes. Dessa forma, possíveis erros de rastreamento que causariam o movimento do objeto mesmo que estivesse parado seriam evitados. Em um passo inicial, foi necessário estudar como a inserção dos objetos virtuais e o rastreamento do marcador são feitos.

Para a inserção de objetos virtuais na cena, a biblioteca de realidade aumentada ARToolKit realiza um rastreamento em cada *frame* do fluxo de vídeo de entrada em busca de um marcador. Quando o padrão de um marcador é encontrado na imagem pelo algoritmo de rastreamento da biblioteca, sua posição no mundo real é identificada e, utilizando essa informação, a posição e a orientação do marcador no mundo virtual é calculada.

Por sua vez, a biblioteca OSGART se encarrega de transformar a posição e a orientação do marcador em uma matriz de transformação que será inserida como um nó de transformação no grafo de cena do OpenSceneGraph. Inserindo um modelo 3D (objeto virtual) como nó filho desse nó de transformação fará com que a matriz de transformação seja aplicada ao modelo 3D, assim o modelo será transladado e rotacionado de acordo com essa matriz, que representa o marcador no mundo virtual. A cena virtual é então desenhada pela câmera virtual com a imagem de entrada da câmera real colocada ao fundo da cena; a impressão visual dada pelo resultado é a de que o modelo 3D está na mesma posição que o marcador no mundo real, criando o efeito de realidade aumentada.

Para atualizar essa matriz de transformação a cada novo *frame* do fluxo de vídeo de entrada, é necessário vincular uma classe de *callback* ao nó de transformação, e essa classe deve implementar um método virtual que atualize a matriz de transformação do nó. A biblioteca OSGART oferece uma classe padrão de *callback* que, além de atualizar a matriz de transformação do nó, também determina a visibilidade dos nós filhos, ou seja, se o rastreamento da imagem não encontrar o marcador, os objetos virtuais vinculados a ele desaparecerão da cena.

Antes deste trabalho, o ARStudio utilizava essa classe padrão de *callback* do OSGART, porém para a implementação de marcadores fixos, ao invés de vincular essa classe padrão ao nó, foi necessário criar uma classe personalizada e vinculá-la.

Figura 15 - Código do método virtual sobrescrito (marcador fixo)

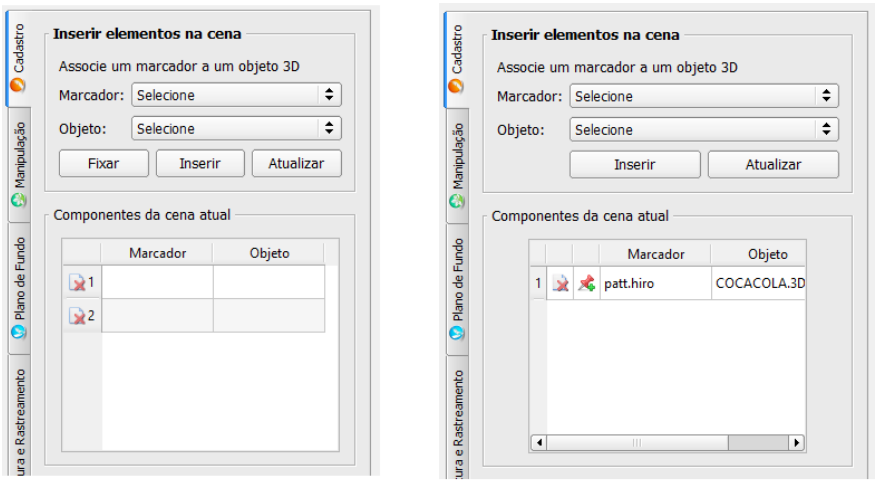
```
virtual void operator()(osg::Node * node, osg::NodeVisitor * nv) {
    if (objeto->getFixo()) {
        if (osg::MatrixTransform* mt = dynamic_cast<osg::MatrixTransform*>(node))
            if (marcador->valid() && !fixou) {
                mt->setMatrix(marcador->getTransform());
                fixou = true;
            }
        nv->setNodeMaskOverride(0xFFFFFFFF);
        node->setNodeMask(fixou ? 0xFFFFFFFF : 0x0);
    }
    else {
        if (fixou)
            fixou = false;
        if (osg::MatrixTransform* mt = dynamic_cast<osg::MatrixTransform*>(node))
            mt->setMatrix(marcador->getTransform());
        nv->setNodeMaskOverride(0xFFFFFFFF);
        node->setNodeMask(marcador->valid() ? 0xFFFFFFFF : 0x0);
    }
    traverse(node,nv);
}
```

Fonte: elaborada pelo autor

A Figura 15 mostra o método virtual sobrescrito pela classe personalizada criada. Uma variável de controle indica se o usuário selecionou a opção de fixar o marcador. Se o marcador foi selecionado como fixo pelo usuário, a matriz de transformação será definida apenas uma vez, quando o rastreamento conseguir localizar o marcador e identificar sua posição. Sua visibilidade também será fixada, ou seja, o objeto virtual vinculado ao marcador sempre estará visível e parado, mesmo que o marcador não esteja mais sendo capturado pela câmera do mundo real.

Para dar controle ao usuário sobre essa nova funcionalidade, a interface do usuário teve que ser remodelada para incluir uma forma de selecionar o estado do marcador.

Figura 16 - Interfaces para fixar o marcador




(a) (b)

(a) Primeira versão da interface; (b) Segunda versão da interface.

Fonte: elaborada pelo autor

A primeira versão da interface criada fornecer esse controle ao usuário está demonstrada na Figura 16(a). O usuário deveria selecionar o marcador e o objeto vinculado a ele, e então escolher se o marcador seria fixo apertando o botão “Fixar” ou se o marcador seria móvel apertando o botão “Inserir”. Porém essa interface não se mostrou flexível, pois seria possível fixar o marcador apenas no momento de inserção na cena, sendo impossível desafixá-lo depois. Uma outra versão da interface foi desenvolvida para solucionar esse problema.

Na Figura 16(b), a segunda versão da interface é exibida. Nessa segunda versão, o usuário deve associar um objeto ao marcador e inseri-lo na cena. Após inserido, uma nova linha na tabela de componentes da cena atual será inserida com o nome do marcador e o nome do objeto inseridos. Para fixar o marcador, basta clicar no botão com um ícone de alfinete. O marcador será fixado e o ícone mudará, indicando que o procedimento foi concluído. Para desafixar o marcado, basta clicar novamente no ícone, e ele voltará ao seu estado normal. Com essa nova interface, é possível fixar o marcador a qualquer momento, além de permitir que o marcador seja desafixado também.

## 3.4 Implementação de multimarcadores

A intenção dessa etapa era de utilizar multimarcadores para amenizar o efeito de obstrução de marcadores. A biblioteca ARToolKit já possui suporte a rastreamento de multimarcadores, porém o software ARStudio não havia sido programado para carregar os arquivos de padrão dos multimarcadores, que são diferentes dos arquivos padrão dos marcadores simples.

Com os multimarcadores, mais de um padrão de marcador deve ser carregado, por isso além dos arquivos de padrão de cada um dos marcadores que compõem o multimarcador, um arquivo principal de configuração contém informações sobre os marcadores e suas respectivas posições dentro do multimarcador. A Figura 17 mostra o trecho de um arquivo de configuração de um multimarcador.

Figura 17 - Trecho de arquivo de configuração de um multimarcador

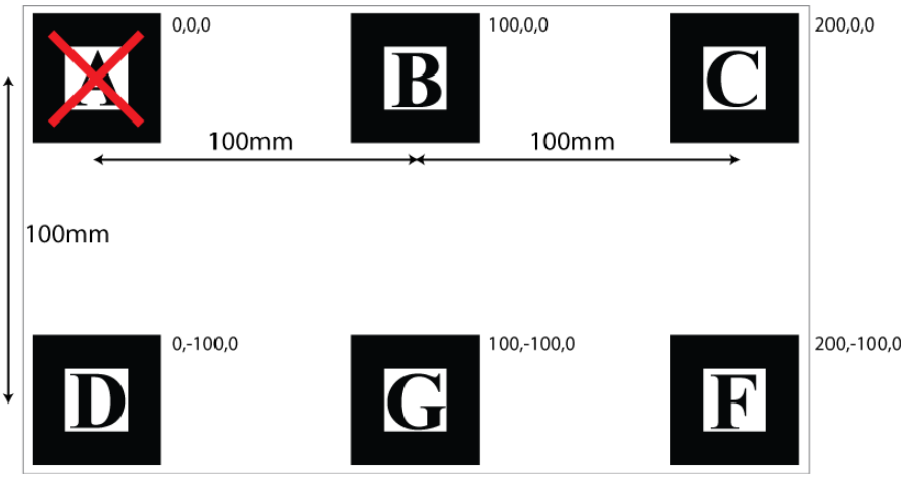
```
#the number of patterns to be recognized
6

#marker 1
Data/multi/patt.a
40.0
0.0 0.0
1.0000 0.0000 0.0000 0.0000
0.0000 1.0000 0.0000 0.0000
0.0000 0.0000 1.0000 0.0000

#marker 2
Data/multi/patt.b
40.0
0.0 0.0
1.0000 0.0000 0.0000  100.0000
0.0000 1.0000 0.0000    0.0000
0.0000 0.0000 1.0000    0.0000

#marker 3
Data/multi/patt.c
40.0
0.0 0.0
1.0000 0.0000 0.0000  200.0000
0.0000 1.0000 0.0000    0.0000
0.0000 0.0000 1.0000    0.0000

#marker 4
Data/multi/patt.d
40.0
0.0 0.0
1.0000 0.0000 0.0000    0.0000
0.0000 1.0000 0.0000 -100.0000
0.0000 0.0000 1.0000    0.0000

#marker 5
Data/multi/patt.g
40.0
0.0 0.0
1.0000 0.0000 0.0000  100.0000
0.0000 1.0000 0.0000 -100.0000
0.0000 0.0000 1.0000    0.0000

#marker 6
Data/multi/patt.f
40.0
0.0 0.0
1.0000 0.0000 0.0000  200.0000
0.0000 1.0000 0.0000 -100.0000
0.0000 0.0000 1.0000    0.0000
```

Fonte: elaborada pelo autor

Nesse arquivo de configuração, o número de marcadores simples que formarão o multimarcador deve ser especificado. Além disso, para cada marcador simples, o caminho do arquivo de padrão deve ser informado, junto com a largura do padrão em milímetros (não é necessário fornecer altura pois os padrões são quadrados, possuem largura igual à altura), posição do centro do padrão e uma matriz de transformação (3x4) que especifica a posição e orientação do marcador dentro do multimarcador (a

matriz 3x3 à esquerda indica a rotação e a coluna da esquerda indica a translação). Uma representação do marcador gerado por esse arquivo de configuração pode ser vista na Figura 18.

Figura 18 - Representação do multimarcador com indicações




Fonte: OSGART, 2014

O software ARStudio possui uma pasta específica para armazenar os arquivos de padrão dos marcadores, e quando ele é inicializado os nomes dos marcadores presentes nessa pasta aparecem em uma lista de marcadores para que sejam inseridos na cena virtual. Para permitir que o usuário utilize multimarcadores, eles precisam antes serem inseridos nessa lista de marcadores criada automaticamente no início da execução do ARStudio. Para isso, um método para verificação dos marcadores disponíveis e inserção deles na lista foi criado, como demonstrado na Figura 19. Os arquivos de padrão dos marcadores simples ficam no caminho “\data” e possuem prefixo “patt.”, enquanto que os arquivos de configuração dos multimarcadores ficam no caminho “\data\multi” e possuem extensão “.dat”.

Figura 19 - Método que verifica e insere os marcadores na lista de seleção

```
void Janela::InsereMarcadoresNaInterface()
{
    QDir currentDir;
    QStringList files;

    currentDir = QDir::currentPath()+"\\data";
    QStringList filters;
    filters << "patt.*";

    files = currentDir.entryList(filters,QDir::Files|QDir::NoSymLinks);

    for (int i = files.size()-1; i >=0 ; i--)
    {
        ui.marcadorBox->insertItem(1,files[i]);
        ui.marcadorPlanoRealBox->insertItem(1,files[i]);
    }

    currentDir = QDir::currentPath()+"\\data\\multi";
    QStringList filters2;
    filters2 << "*.dat";

    files = currentDir.entryList(filters2,QDir::Files|QDir::NoSymLinks);

    for (int i = files.size()-1; i >=0 ; i--)
    {
        ui.marcadorBox->insertItem(1,files[i]);
        ui.marcadorPlanoRealBox->insertItem(1,files[i]);
    }
}
```

Fonte: elaborada pelo autor

Quando selecionado na lista, o marcador deve ser identificado como sendo um marcador simples ou um multimarcador e o caminho do arquivo de padrão (marcador simples) ou de configuração (multimarcador) é fornecido a um método que inicia o reconhecimento marcador pelo rastreador da biblioteca de realidade aumentada. O código desse processo está exibido na Figura 20.

Figura 20 - Código que determina o tipo de marcador e o caminho dos arquivos do marcador

```
std::string caminho;
QString stdAux(marcadorSelecionado.c_str());

if (stdAux.contains("patt.",Qt::CaseInsensitive))
{
    marcadorSelecionado = stdAux.toStdString();
    caminho = "single;data/" + marcadorSelecionado + ";80;0;0";
}
else
{
    marcadorSelecionado = stdAux.toStdString();
    caminho = "multi;data/multi/" + marcadorSelecionado;
}

listaMarcadores = rastreador->reconhecerNovoMarcador(caminho,marcadorSelecionado);
```

Fonte: elaborada pelo autor

No método que inicia o reconhecimento do novo marcador, o marcador é adicionado ao rastreador de marcadores, e o seu padrão passará a ser reconhecido pela biblioteca de realidade aumentada. O método de reconhecimento do marcador pode ser visto na Figura 21.

Figura 21 - Método que inicia o reconhecimento do marcador

```
osg::ref_ptr<osgART::Marker> Rastreador::reconhecerNovoMarcador(std::string str, std::string nome )
{
    if (getTrackerPlugin()->getMarkerCount()!=0)
    {
        for (int i=0;i<getTrackerPlugin()->getMarkerCount();i++)
        {
            if ( getTrackerPlugin()->getMarker(i)->getName() == nome )
            {
                getTrackerPlugin()->getMarker(i)->setActive(true);
                return getTrackerPlugin()->getMarker(i);
            }
        }
    }

    osgART::Marker * markerAux = getTrackerPlugin()->addMarker(str);
    markerAux->setName(nome);
    markerAux->setActive(true);
    return markerAux;
};
```

Fonte: elaborada pelo autor

## 3.5 Implementação de algoritmo de *matting* digital aprimorado

Esta etapa de desenvolvimento exigiu o estudo de algoritmos de *matting* digital que alcançassem resultados melhores do que os alcançados pelo método até então

utilizado pelo software ARStudio, o *chroma-key* de Van Den Bergh e Lalioti (1999), e que ao mesmo tempo permitissem executação em tempo real.

Como o algoritmo de *chroma-key* utilizado pelo ARStudio gerava um *matting* binário, ou seja, com apenas dois níveis de transparência (transparência total ou nenhuma transparência), uma forma simples de melhorar a qualidade do resultado do *matting* digital seria usar um algoritmo que aplicasse vários níveis de transparência nos *pixels* (transparência parcial).

Durante a pesquisa realizada sobre algoritmos de *matting* digital com uso de transparência parcial, o algoritmo chamado *color difference key* desenvolvido por Vlahos (1971) foi encontrado. Esse algoritmo calcula a transparência dos *pixels* baseado na diferença dos canais de cores desses *pixels*. A descrição detalhada do algoritmo é dada no capítulo de fundamentação teórica deste trabalho (seção 2.5).

Para testar a eficácia do algoritmo, um aplicativo separado do software ARStudio foi criado, com duas versões distintas: uma para processar apenas uma imagem e outra para processar vídeos.

Na versão para imagem única, uma imagem de entrada com fundo verde (cor chave) e uma imagem com o novo *background* eram carregadas e ambos os algoritmos, *chroma-key* e *color difference key*, aplicavam o *matting* digital à imagem de entrada, gerando dois resultados (um para cada algoritmo). Essa versão teve o intuito de analisar a diferença entre os dois algoritmos em termos de qualidade do resultado.

Na versão para vídeos, um arquivo de vídeo pré-gravado era carregado junto com uma imagem de *background* e o algoritmo de *color difference key* realizava o *matting* digital em cada *frame* do vídeo. Como o vídeo estava armazenado em disco, todos os *frames* do vídeo estavam disponíveis para o aplicativo assim que ele os requisitasse, sem uma taxa limite de *frames* por segundo. Se um vídeo capturado por uma câmera em tempo real fosse utilizado, o aplicativo estaria restrito à taxa de *frames* por segundo da captura. A ideia desta versão do aplicativo era analisar a quantidade de *frames* por segundo que o algoritmo de *color difference key* consegue processar, para definir se o algoritmo consegue manter uma taxa de *frames* por segundo aceitável e ser executado em tempo real.

Nesta etapa de desenvolvimento, também foram estudadas formas de melhorar os resultados dos algoritmos de *matting* digital através da aplicação de filtros nas imagens processadas pelos algoritmos, ou seja, aplicação de operações matemáticas

nas imagens para realçar ou reduzir certas características, tais como borramento, contraste, etc. O resultado das pesquisas feitas sobre o assunto mostrou que um filtro comumente utilizado no *matting* digital para aprimorar seus resultados é o filtro de borramento (*blur*), utilizado em duas etapas: uma etapa inicial para reduzir o ruído da imagem de entrada, ou seja, reduzir variações incorretas de brilho e cor produzidas pela câmera durante a captura da imagem, e uma etapa final para suavizar as bordas do elemento extraído pelo algoritmo de *matting* (WRIGHT, 2010). Essas etapas inicial e final são chamadas de pré-borramento (*pre-blur*) e pós-borramento (*post-blur*) respectivamente.

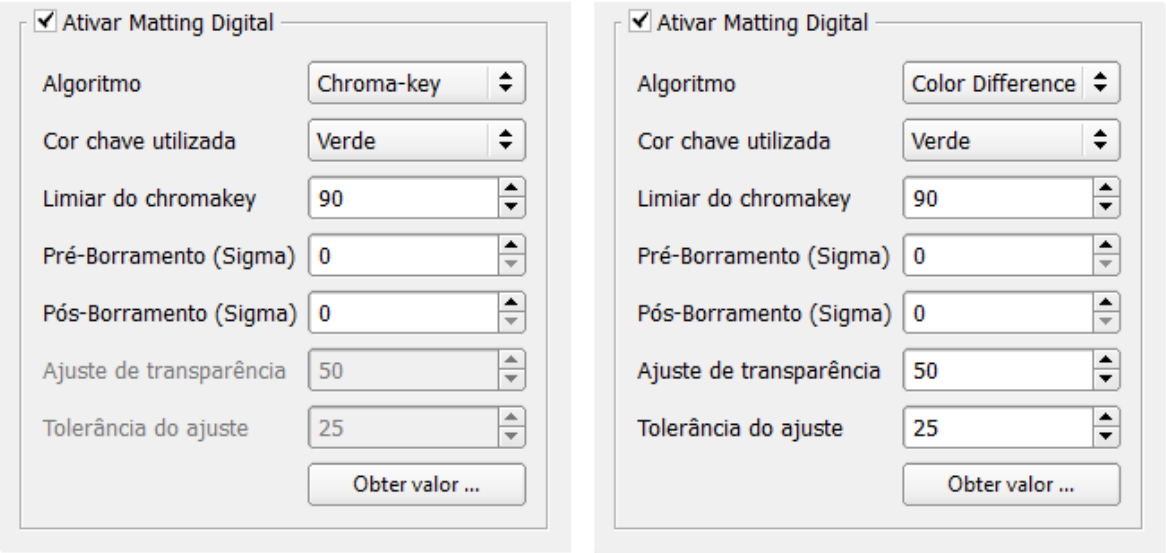
A aplicação do filtro de borramento em imagens traz um efeito negativo: a perda de detalhes da imagem (WRIGHT, 2010). Porém os efeitos da aplicação do filtro no *matting* digital são minimizados pela forma como ele é aplicado: na etapa de pré-borramento, uma cópia da imagem de entrada é borrada e usada para calcular os valores de transparência dos *pixels*, mas a imagem utilizada para compor o resultado final é a imagem de entrada original (não borrada); na etapa de pós-borramento, o filtro é aplicado apenas no canal *alpha* da imagem (canal que contém os valores de opacidade), e não nos canais de cor da imagem, preservando os detalhes de cor dos *pixels*.

As duas versões do aplicativo utilizado para testar a eficácia do algoritmo de *color difference key* foram atualizados para incluírem as etapas de aplicação do filtro de borramento. Testes foram realizados para analisar a qualidade dos resultados com a aplicação das etapas de filtragem. Os resultados obtidos podem ser observados no capítulo de testes e resultados.

Após a comprovação da eficácia do algoritmo de *color difference key* e da aplicação dos filtros para melhorar os resultados do algoritmo, essas novas funcionalidades foram incluídas no software de estúdio virtual. O algoritmo de *chroma-key* utilizado pelo ARStudio foi mantido e atualizado com a aplicação das etapas de filtragem, já que os processos de borramento podem ser aplicados a qualquer algoritmo de *matting* para melhorar os resultados. Dessa forma, o usuário pode escolher qual algoritmo quer que seja utilizado para o *matting* digital. Além disso, os parâmetros do filtro de borramento e do algoritmo escolhido podem ser manipulados pelo usuário. A Figura 22 ilustra a interface do software, com opção de escolha do algoritmo e dos parâmetros do algoritmo escolhido, na Figura 22(a) o algoritmo

selecionado foi o de *chroma-key* e na Figura 22(b) o algoritmo selecionado foi o de *color difference key*.

Figura 22 - Interface de seleção de algoritmo de *matting* digital




(a) Interface com algoritmo de *chroma-key* selecionado; (b) Interface com algoritmo de *color difference key* selecionado.

Fonte: elaborada pelo autor

Os parâmetros do filtro de borramento são os valores de *sigma* (desvio padrão), que são definidos pelo usuário para cada etapa de borramento (pré-borramento e pós-borramento). Através desses parâmetros, o usuário pode definir a intensidade do borramento efetuado em cada etapa.

Para o algoritmo de *color difference key*, dois novos parâmetros foram adicionados para fazer a escala dos valores de transparência dos *pixels*: ajuste de transparência e tolerância do ajuste de transparência. O ajuste de transparência aumenta a transparência dos *pixels* de acordo com o valor especificado pelo campo, com valores de 0 a 255. Se o ajuste exceder o valor máximo de transparência, a transparência máxima será aplicada. Já a tolerância do ajuste de transparência determina o valor mínimo de transparência que o *pixel* deve ter para que o ajuste de transparência seja aplicado. Os *pixels* que tiverem valor de transparência menor que o definido no campo de tolerância não receberão nenhum ajuste de transparência, prevenindo que *pixels* que deveriam ser opacos se tornem transparentes.

## 3.6 Implementação de registro de objetos virtuais com o uso do Kinect

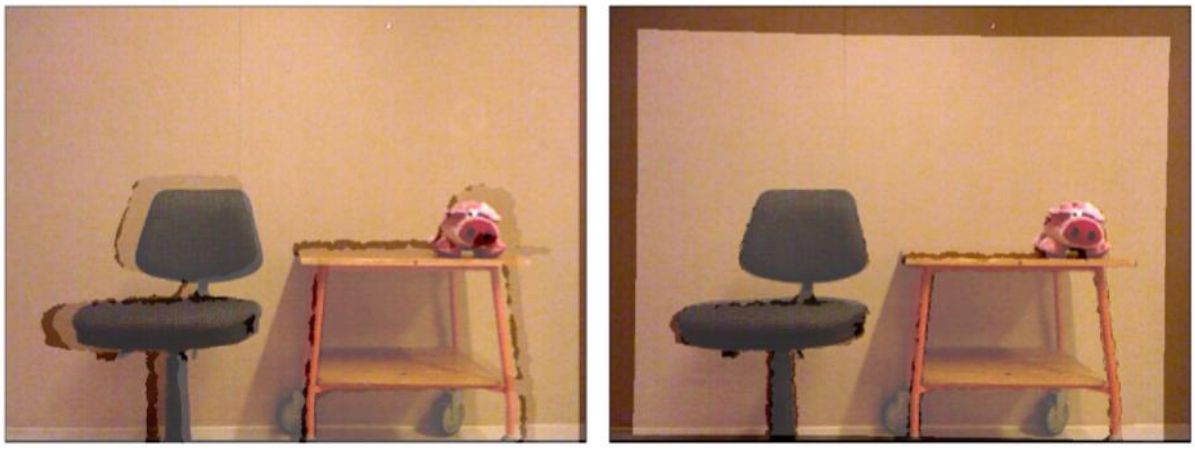
O objetivo dessa etapa era realizar o registro de objetos virtuais em relação aos atores, permitindo ao software de estúdio virtual determinar se o objeto virtual está à frente ou atrás dos atores no mundo real. Os recursos do Kinect foram usados para facilitar esse registro e para isso foi necessário o uso de uma biblioteca para manipular o dispositivo.

A biblioteca usada para acessar os recursos do Kinect foi a libfreenect, uma biblioteca de código aberto desenvolvida e mantida pela comunidade OpenKinect que permite a obtenção das imagens capturadas pela câmera RGB, controle dos LEDs e dos motores de inclinação do dispositivo, acesso às informações do acelerômetro do Kinect e às informações de profundidade geradas pelo sensor de profundidade.

As informações de profundidade obtidas pelo Kinect são representadas através de um mapa de disparidade. Esse mapa é calculado através de uma comparação entre o padrão de pontos IR conhecido do projetor IR (padrão memorizado pelo Kinect para um plano com profundidade conhecida) e o padrão de pontos IR capturado pela câmera IR no ambiente. Os pontos do padrão memorizado são comparados com os pontos do padrão capturado em busca de uma correlação, ou seja, o mesmo ponto que está em um padrão deve ser encontrado no outro padrão. Após obter a correlação, a diferença entre a posição de um ponto em relação ao outro é obtida: essa é a chamada disparidade. Tendo a profundidade conhecida do padrão memorizado e o valor de disparidade, é possível estimar o valor de profundidade através de triangulação. Aplicando a triangulação para todos os valores do mapa de disparidade, obtém-se o mapa de profundidade. No mapa de profundidade, cada pixel representa a distância entre o Kinect e o ambiente naquele ponto.

Para saber a profundidade de um ponto da imagem capturada pela câmera RGB, é necessário encontrar esse ponto no mapa de profundidade. Porém, a relação entre a imagem da câmera RGB e o mapa de profundidade não é direta pois, devido à distância entre a câmera RGB e a câmera IR, as imagens não são alinhadas (ANDERSEN et al., 2012). A Figura 23 mostra o mapa de profundidade sobreposto à imagem RGB. Na Figura 23(a), nenhum alinhamento foi feito, enquanto na Figura 23(b) as duas imagens foram alinhadas.

Figura 23 - Sobreposição da imagem RGB e do mapa de profundidade




(a) Sobreposição sem alinhamento; (b) Sobreposição com alinhamento.

Fonte: ANDERSEN et al., 2012

Nas principais bibliotecas criadas para acessar os recursos do Kinect, esse alinhamento é feito automaticamente a partir de parâmetros conhecidos das câmeras do dispositivo. Porém, esse alinhamento só funciona para a câmera RGB do Kinect, pois os parâmetros conhecidos se aplicam à câmera dele. Para utilizar uma câmera RGB externa, é necessário descobrir os parâmetros das câmeras e realizar o alinhamento entre elas: esse é processo de calibração do Kinect com uma câmera RGB externa. Como nesse trabalho o uso de uma câmera externa ao Kinect era pretendido, foi necessário realizar o processo de calibração.

### 3.6.1 Calibração do Kinect com câmera RGB externa

Para descobrir os parâmetros intrínsecos e extrínsecos das câmeras, o algoritmo desenvolvido por Herrera C., Kannala e Heikkilä (2012) em MATLAB (ambiente de programação para computação numérica) foi utilizado. Através da detecção dos cantos de um tabuleiro de xadrez, o algoritmo consegue calcular os parâmetros intrínsecos de distância focal e coeficiente de distorção da lente de ambas as câmeras (câmera RGB e câmera IR), além das matrizes de rotação e translação que devem ser aplicadas sobre o mapa de disparidade para o alinhamento com a imagem RGB. Imagens do tabuleiro em diferentes posições e distâncias devem ser capturadas pela câmera RGB junto com o mapa de disparidade correspondente a essa imagem RGB, e fornecidas ao algoritmo para o cálculo dos parâmetros. A

detecção dos cantos do tabuleiro nas imagens RGB é feita automaticamente utilizando métodos de visão computacional enquanto que no mapa de disparidade os cantos devem ser selecionados manualmente, pois os cantos do tabuleiro não são visíveis no mapa de disparidade. A Figura 24 apresenta um exemplo de imagem RGB e o mapa de disparidade usadas pelo algoritmo que calcula os parâmetros.

Figura 24 - Imagens do tabuleiro para a calibração

(a) Imagem RGB da câmera externa com tabuleiro; (b) Mapa de disparidade correspondente à imagem RGB.

Fonte: elaborada pelo autor

Os parâmetros obtidos pelo algoritmo foram armazenados em um arquivo e são carregados desse arquivo sempre que o software ARStudio é inicializado. Tendo os parâmetros intrínsecos de distância focal, coeficiente de distorção da lente e as matrizes de rotação e translação, um método de calibração teve que ser aplicado com esses parâmetros. No ARStudio, o método de calibração utilizado foi o método de Nicolas Burrus (2014).

O primeiro passo do método de calibração é tirar a distorção do mapa de disparidade e calcular o valor de profundidade em metros. A Equação 6 (HERRERA C.; KANNALA; HEIKKILÄ, 2012) mostra o cálculo da profundidade de um *pixel* do mapa de disparidade, com *z* representando o valor de profundidade, *d* representando o valor de disparidade e $c_0$ e $c_1$ representando os coeficientes de distorção da câmera IR.

$$z = \frac{1}{c_1 d + c_0} \tag{6}$$

Com o valor de profundidade do *pixel* calculado, é possível projetar cada *pixel* do mapa de disparidade em um espaço 3D, transformando cada *pixel* em um ponto 3D (com coordenadas x, y e z). A Equação 7 e a Equação 8 (BURRUS, 2014) mostram a projeção das coordenadas *x* e *y*, com a dupla *(u, v)* representando a posição do *pixel* no mapa de disparidade, $(f_{xd}, f_{yd})$ representando os parâmetros de distância focal da câmera IR e $(c_{xd}, c_{yd})$ representando o centro ótico focal da câmera IR. A coordenada *z* é o próprio valor de profundidade em metros.

$$x = \frac{(u - c_{xd}) \times z}{f_{xd}} \tag{7}$$

$$y = \frac{(v - c_{yd}) \times z}{f_{yd}} \tag{8}$$

Após obter o ponto 3D de cada pixel, a rotação e translação para o alinhamento das coordenadas do mapa de disparidade com a imagem RGB são aplicadas. A Equação 9 (BURRUS, 2014) demonstra a aplicação da rotação e translação, com $P_{3D}$ representando o ponto 3D (matriz 3x1), *R* representando a matriz de rotação (matriz 3x3), *T* representando a matriz de translação (matriz 3x1) e o ponto rotacionado e transladado e $P'_{3D}$.

$$P'_{3D} = R \times P_{3D} + T \tag{9}$$

A projeção dos *pixels* para o espaço 3D ($P_{3D}$) foi feita utilizando os parâmetros intrínsecos da câmera IR. Agora, com os pontos 3D rotacionados e transladados ($P'_{3D}$), o processo inverso é feito: a reprojeção dos pontos 3D para uma imagem 2D. Porém, dessa vez, os parâmetros intrínsecos utilizados serão os da câmera RGB externa, reprojetando os pontos 3D originados do mapa de disparidade na imagem RGB. A Equação 10 e a Equação 11 (BURRUS, 2014) mostram a reprojeção, com *(u, v)* representando a posição do *pixel* na imagem RGB, *(x, y, z)* representando as coordenadas do ponto 3D rotacionado e transladado, $(f_{xr}, f_{yr})$ representando os parâmetros de distância focal da câmera RGB externa e $(c_{xr}, c_{yr})$ representando o centro ótico focal da câmera RGB externa.

$$u = \frac{x \times f_{xr}}{z} + c_{xr} \quad (10)$$

$$v = \frac{y \times f_{yr}}{z} + c_{yr} \quad (11)$$

A reprojeção fornece a posição *(u,v)* da imagem RGB, e com o valor *z* de profundidade é possível montar um mapa de profundidade relativo à imagem RGB, ou seja, para cada ponto da imagem RGB é possível saber sua respectiva profundidade. Porém, a resolução do mapa de disparidade fornecido pelo Kinect é 640x480 *pixels*, o que é menor do que a resolução geralmente usada em câmeras RGB. Isso faz com que a quantidade de pontos 3D reprojetados na imagem RGB seja menor que sua resolução, criando um mapa de profundidade com *pixels* esparsos. Para preencher os espaços vazios do mapa de profundidade gerado, o software ARStudio faz uma interpolação do mapa.

Com o mapa de profundidade gerado e calibrado a cada *frame*, os valores de profundidade dos *pixels* da imagem RGB podem ser obtidos para saber se esse *pixel* está à frente ou atrás de um *pixel* de um objeto virtual renderizado. Porém, o valor de profundidade obtido do mapa de profundidade é medido em metros, enquanto a profundidade de um *pixel* de um objeto virtual renderizado tem profundidade em coordenadas do mundo virtual. Foi necessário criar um método para obter a relação entre as unidades de distância do mundo real e do mundo virtual.

### 3.6.2 Obtenção da relação entre distância do mundo real e do mundo virtual

O método criado para obter a relação entre a profundidade do mundo real e do mundo virtual inseria, em uma coordenada conhecida, um objeto virtual (retângulo) transparente cobrindo todo o campo de visão da câmera virtual. Esse objeto, apesar de ser transparente, tinha o modo de teste de profundidade habilitado, ou seja, na etapa de renderização a profundidade dos objetos em cena era verificada e os objetos que estivessem atrás desse objeto transparente seriam ocluídos. Esse objeto foi chamado de plano de corte, pois todos os objetos atrás dele seriam cortados da cena.

Um marcador com um objeto associado a ele foi sendo afastado da câmera IR até que o objeto começasse a ser cortado pelo plano de corte. Quando isso ocorresse,

o marcador estaria na profundidade do mundo real relativa à profundidade do plano de corte no mundo virtual. Dessa forma, medindo a distância entre a câmera IR e o marcador e relacionando esse valor com a coordenada de profundidade conhecida do plano de corte, foi possível encontrar a relação entre a profundidade do mundo real e do mundo virtual.

Com essa relação obtida, os valores do mapa de profundidade do Kinect puderam ser convertidos em coordenadas de profundidade do mundo virtual, fazendo com que a profundidade do mundo real e do mundo virtual tivessem a mesma unidade de medida. O próximo passo dessa etapa de desenvolvimento foi criar um meio de realizar a oclusão dos objetos virtuais que estivessem atrás dos objetos do mundo real utilizando essas informações de profundidade.

### 3.6.3 Oclusão dos objetos virtuais de acordo com profundidade do mundo real

Para realizar a oclusão dos objetos virtuais, a mesma ideia de plano de corte do passo anterior foi utilizada: um objeto virtual retangular transparente foi inserido cobrindo o campo de visão da câmera virtual e cortando todos os objetos que estivessem atrás dele. Porém, dessa vez, esse objeto não teria profundidade constante em toda sua superfície; cada *pixel* dessa superfície receberia o valor de profundidade do mapa de profundidade obtido através do Kinect. Esse objeto foi chamado de superfície de corte.

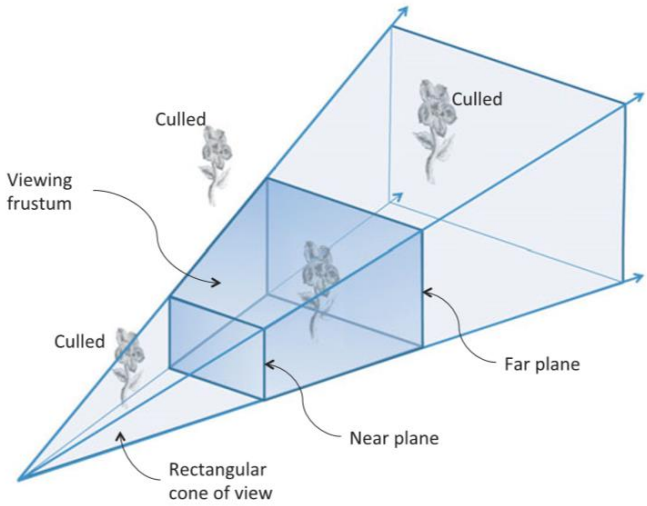
Para atribuir o valor de mapa de profundidade a cada *pixel*, um *fragment shader* foi criado. O *fragment shader* é um código compilado pelo OpenGL e executado pela placa de vídeo em cada fragmento do objeto, ou seja, as operações programadas no *fragment shader* são realizadas em cada *pixel* do objeto. As propriedades que podem ser alteradas pelo *fragment shader* são a cor e a profundidade do fragmento. Como a finalidade de seu uso no ARStudio é a de atribuir o valor do mapa de profundidade a cada *pixel*, a propriedade alterada pelo *fragment shader* criado foi a de profundidade do fragmento.

O mapa de profundidade obtido pelo Kinect foi então atribuído à superfície de corte como uma textura. Como a superfície de corte é transparente, a textura não é renderizada porém ela serve como dado de entrada para o *fragment shader*. Analisando os *pixels* da textura (mapa de profundidade), o *fragment shader* converte os valores de profundidade em metro do mapa em unidades de profundidade do

mundo virtual, e calcula o valor de profundidade que será enviado ao *buffer* de profundidade da placa de vídeo (*z-buffer*) pelo OpenGL. Os valores de profundidade enviados pelo OpenGL precisam ser calculados pelo *fragment shader* pois o OpenGL utiliza valores normalizados, ou seja, esses valores não representam as coordenadas do mundo virtual, e sim um valor entre 0 e 1. Essa normalização é feita com relação ao volume de visão da câmera virtual.

O volume de visão (*viewing frustum*) da câmera virtual determina quais objetos devem ser renderizados, descartando do processo de renderização os objetos que estiverem fora dele. Isso é feito para poupar processamento e aumentar desempenho pois, se o objeto não aparecerá na tela, não há necessidade de realizar todos as operações matemáticas necessárias para sua renderização. O volume de visão da câmera virtual é determinado pelo seu ângulo de visão, além de dois planos que determinam os limites do volume: o plano próximo (*near plane*) e o plano distante (*far plane*). Esses planos determinam a profundidade mínima e máximo que serão enxergadas pela câmera virtual. A Figura 25 ilustra o volume de visão, ressaltado entre o plano próximo e o plano distante.

Figura 25 - Volume de visão




Fonte: SHREINER et al., 2013

A normalização feita pelo OpenGL considera os planos próximo e distante para determinar os valores extremos de profundidade. Com as coordenadas de profundidade desses planos, uma equação não-linear é aplicada para normalizar o valor de profundidade de uma determinada coordenada. Essa normalização é feita de acordo com a Equação 12 (HOFF III, 2014), com *z'* representando o valor de

profundidade normalizado, *z* representando a coordenada de profundidade a ser normalizada, $z_{near}$ representando a coordenada de profundidade do plano próximo e $z_{far}$ representando a coordenada de profundidade do plano distante.

$$z' = \frac{\frac{1}{z_{near}} - \frac{1}{z}}{\frac{1}{z_{near}} - \frac{1}{z_{far}}} \quad (12)$$

No software ARStudio, o *fragment shader* é vinculado à superfície de corte e recebe a textura (mapa de profundidade) e os valores da profundidade dos planos próximo e distante como variáveis uniformes. A textura é atualizada a cada *frame* do sensor de profundidade fornecido pelo Kinect.

Figura 26 - Código do *fragment shader*

```
uniform sampler2D depthTexture;
uniform float zNear;
uniform float zFar;

void main(void)
{
    vec4 depth = texture2D(depthTexture, gl_TexCoord[0].xy);
    float z;
    if (depth.r == 0)
        z = zFar;
    else
        z = depth.r * 1000 / 0.5;
    if (z < zNear)
        z = zNear;
    else
        if (z > zFar)
            z = zFar;
    gl_FragDepth = (1/zNear - 1/z) / (1/zNear - 1/zFar);
}
```

Fonte: elaborada pelo autor

A Figura 26 mostra o código do *fragment shader*. O valor de profundidade em metros é obtido do mapa de profundidade e convertido para coordenadas de profundidade do mundo virtual (no código, 1000 unidades de coordenada equivalem a 0,5 metro). Caso o valor de profundidade do mapa seja desconhecido pelo Kinect (valor 0), a coordenada de profundidade atribuída é a máxima, ou seja, recebe a coordenada de profundidade do plano distante (*zFar*). Depois de converter o valor do mapa de profundidade, ele é limitado pelas coordenadas dos planos próximo e

distante, caso a conversão dê um resultado menor que *zNear* ou maior que *zFar*, e em seguida o valor de profundidade do fragmento é normalizado.

Com a normalização feita no *fragment shader*, o *buffer* de profundidade da placa de vídeo recebe os valores de profundidade de cada *pixel* e os aplicam no processo de renderização, criando uma superfície que recebe os valores do mapa de profundidade do Kinect. Assim, essa superfície passa ser uma representação virtual da superfície do mundo real capturada pela câmera, ocluindo todos os objetos virtuais que estiverem atrás dela, permitindo o registro dos objetos virtuais em relação aos objetos do mundo real, incluindo atores em cena.

# 4 TESTES E RESULTADOS

Este capítulo apresenta os testes realizados para cada uma das etapas do desenvolvimento, discutindo os resultados obtidos por eles. Os materiais utilizados nos testes também são discriminados.

## 4.1 Materiais

Para os testes, um computador, uma câmera webcam de alta resolução e um dispositivo Kinect foram utilizados.

As especificações do computador utilizado foram:

- Intel Core i7-3520M CPU @ 2.90GHz;
- Memória RAM de 8GB;
- Sistema operacional Microsoft Windows 8.1 (64 bits);
- Placa de vídeo NVIDIA GeForce GT635M;
- Disco rígido de 1TB (5400 RPM).

Os outros dispositivos utilizados foram:

- Logitech HD Pro Webcam C910;
- Microsoft Kinect Xbox 360 (modelo 1414).

Os testes foram realizados no laboratório SACI (Sistemas Adaptativos e Computação Inteligente). A Figura 27 mostra o ambiente do laboratório.

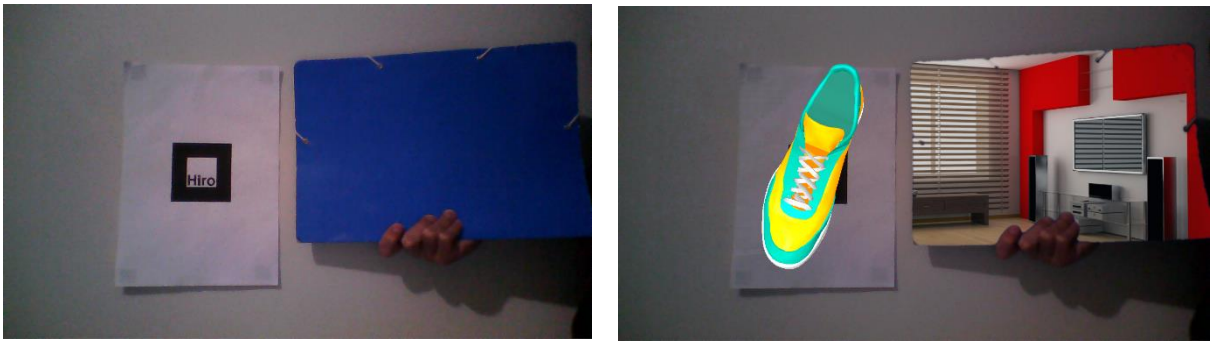
Figura 27 - Laboratório SACI

Fonte: elaborada pelo autor

## 4.2 Armazenamento de vídeo não processado

Para testar a implementação do armazenamento de vídeo não processado com uso de *threads*, alguns vídeos foram gravados com ambos os fluxos: vídeo não processado e vídeo processado. A Figura 28 mostra um dos testes realizados: um objeto foi associado a um marcador posicionado em frente à câmera, ao mesmo tempo que um objeto azul foi colocado em cena para possibilitar o uso do *color difference key* (com cor chave azul). Na Figura 28(a) está um *frame* extraído do arquivo de vídeo não processado que foi armazenado pelo ARStudio, ou seja, sem o objeto virtual e sem a aplicação do *color difference key*. Na Figura 28(b) está o *frame* extraído do arquivo de vídeo processado equivalente ao *frame* do arquivo de vídeo não processado.

Figura 28 - *Frames* dos vídeos armazenados




(a) (b)

(a) Frame do vídeo não processado; (b) Frame do vídeo processado.

Fonte: elaborada pelo autor

Nesse teste, ambos os vídeos conseguiram ser armazenados com a mesma taxa de *frames* por segundo do fluxo de entrada da câmera (30 FPS).

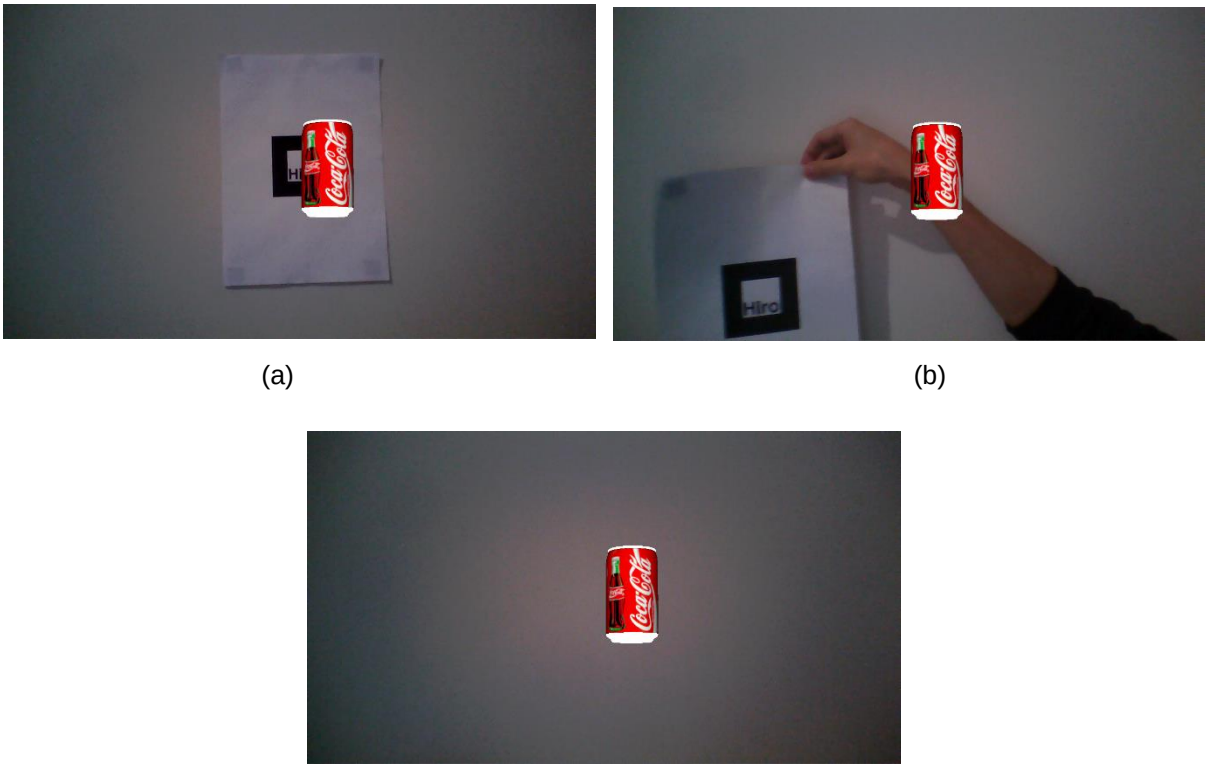
## 4.3 Aplicação de marcadores fixos

A eficiência da implementação de marcadores fixos foi testada através de uma comparação entre marcadores fixos e marcadores móveis. Dois marcadores foram colados lado a lado em uma parede e cada um deles teve um objeto vinculado a si. Um dos marcadores foi selecionado como marcador fixo pela interface do ARStudio enquanto o outro continuava atualizando a posição do objeto constantemente. Nesse teste, foi possível perceber que o objeto vinculado ao marcador fixo se encaixou

melhor no contexto da cena real, enquanto o objeto que tinha sua posição atualizada constantemente continuava se movendo devido às falhas de rastreamento do marcador, diminuindo a imersão transmitida pela realidade aumentada.

Além desse teste, outro teste foi conduzido para testar a visibilidade de um objeto vinculado a um marcador fixo quando este não está mais sendo capturado pela câmera. A Figura 29 mostra as etapas do teste: na Figura 29(a), o marcador com o objeto associado foi fixado; na Figura 29(b) o marcador estava sendo removido da visão da câmera; na Figura 29(c) já havia sido removido e não estava mais sendo capturado pela câmera.

Figura 29 - Teste com marcador fixo




(a) (b)

(c)

(a) Marcador sendo fixado; (b) Marcador sendo removido da visão da câmera; (c) Marcador fora de visão da câmera.

Fonte: elaborada pelo autor

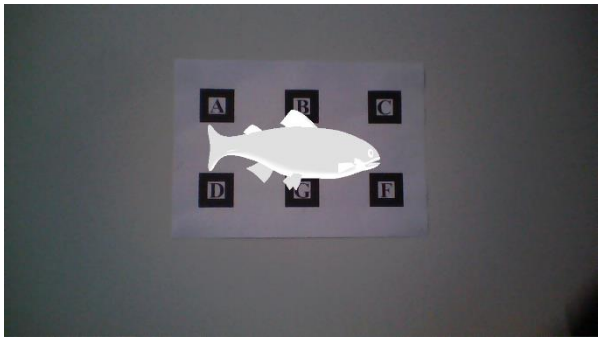
Esse resultado mostrou outra utilidade do marcador fixo: a inserção de objetos na cena virtual sem a necessidade de um marcador na cena real. Com sua utilização, é possível utilizar o marcador para posicionar um objeto virtual em um local desejado do mundo real, fixar o marcador e depois descarta-lo da cena, pois o objeto associado

a ele continuará sendo renderizado na mesma posição mesmo sem que ele esteja sendo capturado e reconhecido pela câmera.

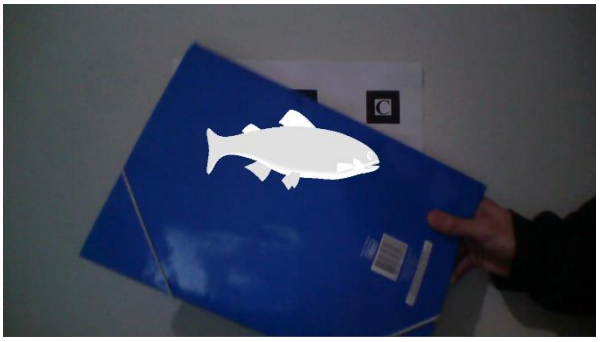
## 4.4 Aplicação de multimarcadores

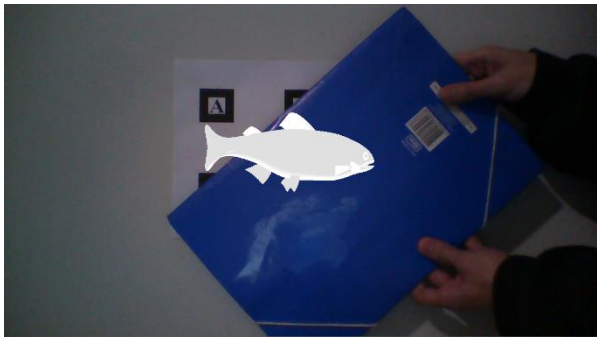
Os testes da implementação de multimarcadores tiveram como objetivo demonstrar o registro contínuo dos objetos virtuais mesmo com a obstrução parcial dos multimarcadores. A Figura 30 apresenta o teste realizado com o objeto vinculado ao multimarcador na Figura 30(a) e a oclusão parcial do multimarcador na Figura 30(b) e na Figura 30(c).

Figura 30 - Teste com multimarcador




(a)




(b) (c)

(a) Multimarcador com objeto; (b) Obstrução da porção esquerda do multimarcador; (c) Obstrução da porção direita do multimarcador.

Fonte: elaborada pelo autor

Pelo resultado do teste é possível notar o objeto continuou sendo registrado na mesma posição devido ao fato de que pelo menos um dos marcadores que compõe o multimarcador ainda estar sendo reconhecido mesmo com os outros marcadores obstruídos. Assim, a utilização de multimarcadores minimiza os efeitos da obstrução de marcadores, melhorando o registro dos objetos virtuais.

Outro resultado obtido com a implementação dos multimarcadores é a redução das falhas de rastreamento do marcador, pois a falha de rastreamento em um marcador é corrigida pelo rastreamento de outro marcador que compõe o multimarcador. Assim, o posicionamento dos objetos se torna mais preciso e estável, mesmo com o multimarcador parado, tornando-se uma alternativa à implementação de marcadores fixos.

## 4.5 Algoritmo de *matting* digital aprimorado

Um aplicativo separado do ARStudio foi desenvolvido para testar o algoritmo de *color difference key*. Nesse aplicativo, testes de comparação entre o *chroma-key* e o *color difference key* foram realizados, além de testes de aplicação de filtro (borramento) e de desempenho em vídeo.

### 4.5.1 Comparação entre *chroma-key* e *color difference key*

O aplicativo criado executava os algoritmos de *chroma-key* e *color difference key* sobre a mesma imagem de entrada, gerando duas imagens de saída, cada uma resultante de um algoritmo. Assim, foi possível comparar os resultados dos dois algoritmos. Os testes realizados utilizavam imagens de entrada com fundo verde (cor chave) e trocavam o fundo por uma imagem vermelha uniforme. O propósito de usar uma imagem de fundo vermelha uniforme era destacar as falhas de extração do *foreground* cometidas pelos algoritmos.

A Figura 31 mostra um dos testes realizados para comparar os resultados de ambos os algoritmos. Na Figura 31(b) está o resultado do algoritmo de *chroma-key*, e na Figura 31(c) está o resultado do algoritmo de *color difference key*. Nesse teste é possível notar que o *chroma-key* apresentou dificuldades em remover o verde próximo às bordas do *foreground*, pois a tonalidade da cor verde nessa região está mesclada com a cor do objeto à frente. Já o *color difference key* conseguiu superar essa dificuldade aplicando transparências aos *pixels* dessa região.

Figura 31 - Teste de comparação entre *chroma-key* e *color difference key* (Teste 1)

(a)

(b)

(c)

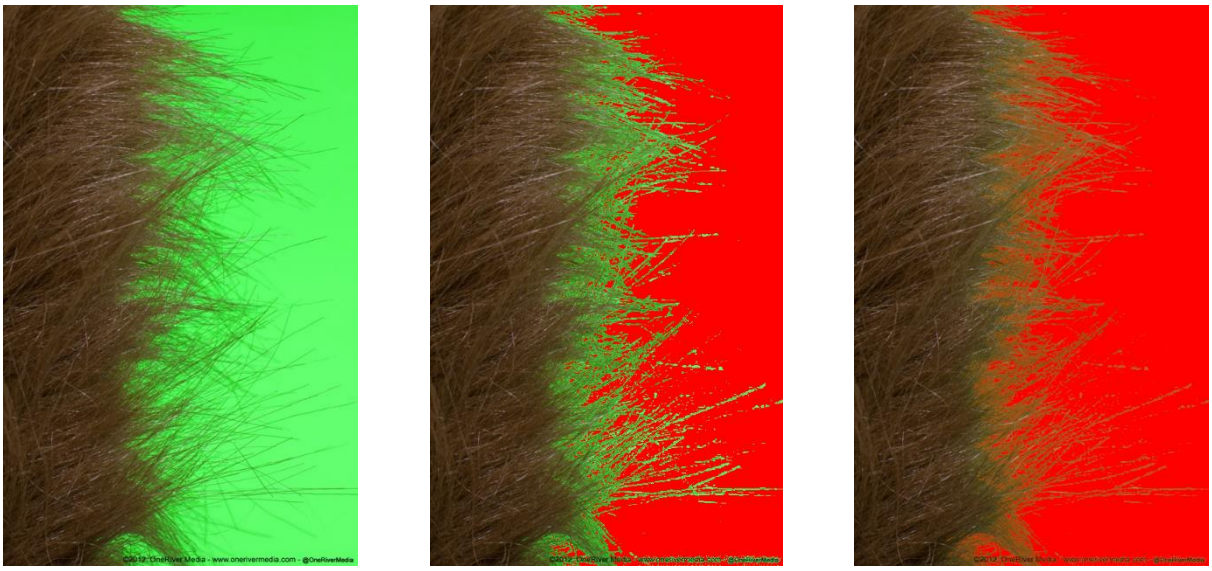
(a) Imagem original (THORNTON, 2014); (b) Resultado do *chroma-key*; (c) Resultado do *color difference key*.

Fonte: elaborada pelo autor

Outro teste realizado é demonstrado na Figura 32. Nesse teste, os pelos são a principal dificuldade de extração. Da mesma forma que no teste anterior, o *chroma-key* tem dificuldade em remover o verde próximo aos pelos, e o *color difference key* suaviza essa região através da transparência. A Figura 32(a) mostra a imagem

original. A Figura 32(b) apresenta o resultado do *chroma-key* e a Figura 32(c) o resultado do *color difference key*.

Figura 32 - Teste de comparação entre *chroma-key* e *color difference key* (Teste 2)



(a) (b) (c)

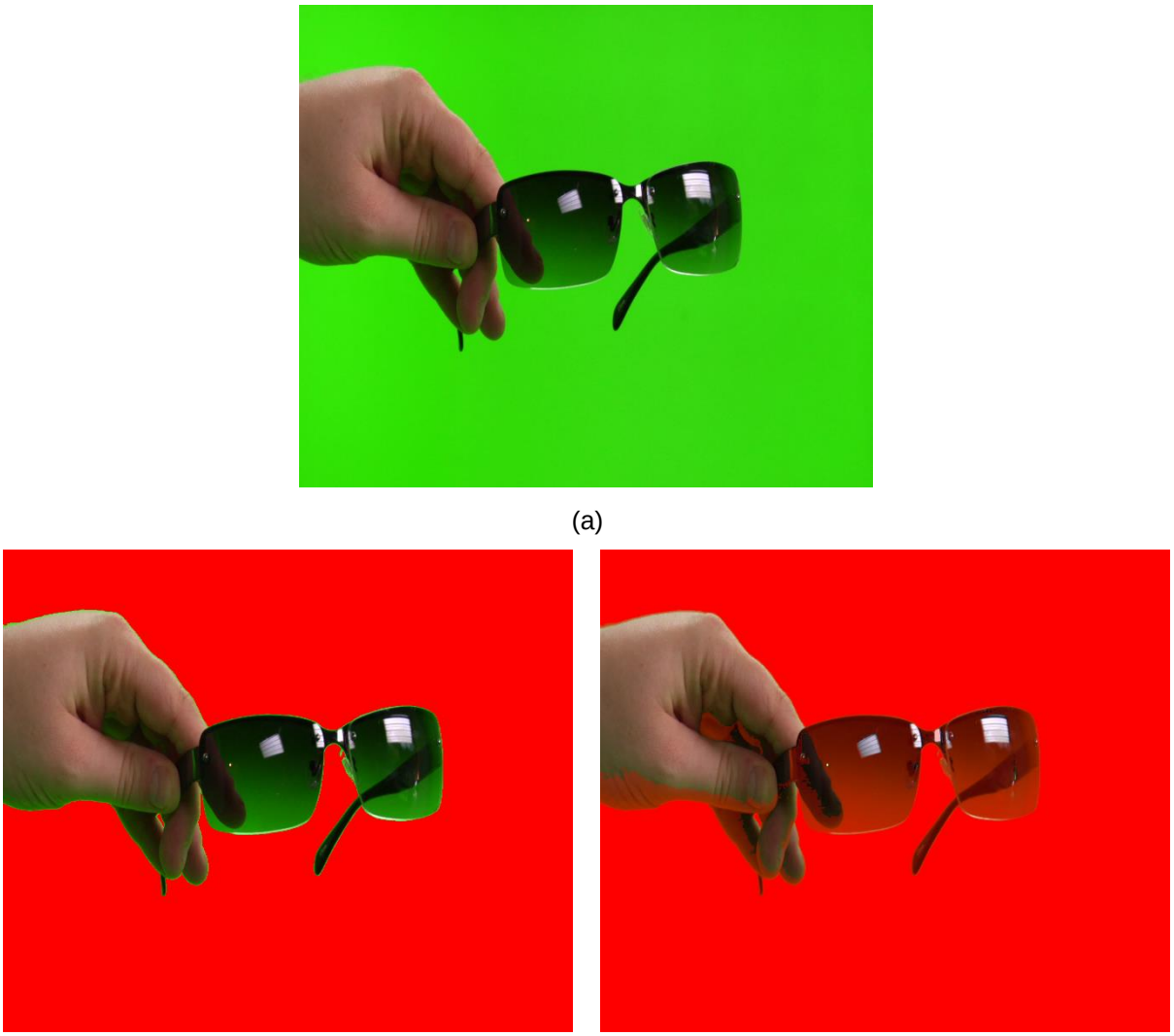
(a) Imagem original (ONERIVER MEDIA, 2014); (b) Resultado do *chroma-key*; (c) Resultado do *color difference key*.

Fonte: elaborada pelo autor

Um último teste foi realizado para comparar como cada algoritmo trata a extração de objetos translúcidos. A Figura 33(a) mostra a imagem original: um óculos escuro. Na Figura 33(b) está o resultado do *chroma-key* e é possível notar que o algoritmo não consegue identificar o verde nas lentes como *background*. Já no resultado do *color difference key* apresentado na Figura 33(c) a transparência de *pixels* foi aplicada às lentes dos óculos, dando a noção de transparência da lente.

Também na Figura 33(c), uma falha do algoritmo de *color difference key* ficou evidente: o reflexo do fundo verde na mão da pessoa que segura os óculos foi confundido com o fundo pelo algoritmo, e uma transparência foi aplicada. Esse tipo de falha pode ser corrigido com um melhor controle de iluminação no set de filmagem.

Figura 33 - Teste de comparação entre *chroma-key* e *color difference key* (Teste 3)



(a) (b) (c)

(a) Imagem original (HOLLYWOOD CAMERA WORK, 2014); (b) Resultado do *chroma-key*; (c) Resultado do *color difference key*.

Fonte: elaborada pelo autor

### 4.5.2 Aplicação de filtro nas imagens

Durante o desenvolvimento dessa etapa, a aplicação de filtro de borramento nas imagens também foi estudado. Para testar os resultados dessa aplicação, as etapas de pré-borramento e pós-borramento foram executadas junto ao algoritmo de *color difference key*. A Figura 34(a) mostra uma parte da imagem ampliada resultante do algoritmo de *color difference key* e a Figura 34(b) mostra a mesma parte da imagem, porém com a aplicação das etapas de borramento junto ao *color difference key*.

Figura 34 - Teste com aplicação de etapas de borramento

(a) (b)

(a) Imagem ampliada sem borramento; (b) Imagem ampliada com borramento.

Fonte: elaborada pelo autor

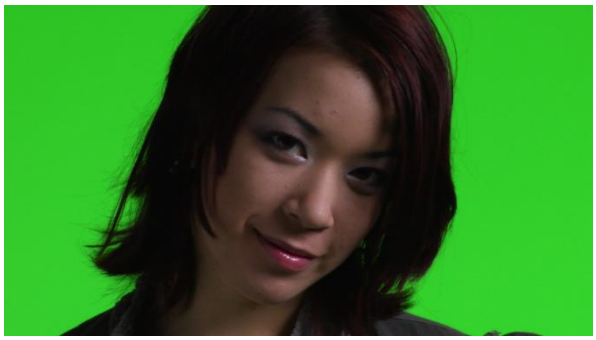
O resultado mostra uma melhora significativa na qualidade da imagem. O resultado do *color difference key*, apesar de aplicar transparência, ainda deixou bordas bem serrilhadas. Com a aplicação das etapas de borramento, a bordas foram suavizadas, tirando o serrilhamento.

### 4.5.3 Desempenho em vídeo

Como o algoritmo de *matting* digital deve ser executado em tempo real pelo ARStudio para possibilitar uma prévia da produção, testes de desempenho tiveram que ser realizados. Arquivos de vídeo armazenados em disco foram utilizados como entrada do algoritmo de *color difference key* para que ele pudesse processar os vídeos sem um limite de *frames* por segundo, processando a maior quantidade de *frames* por segundo que ele pudesse. Três testes foram realizados, com vídeos diferentes. Esses vídeos são representados pela Figura 35.

Figura 35 - Vídeos utilizados no teste

(a)



(b)

(c)

(a) Vídeo do teste 1; (b) Vídeo do teste 2; (c) Vídeo do teste3.

Fonte: HOLLYWOOD CAMERA WORK, 2014

Cada um dos testes foi realizado duas versões do mesmo vídeo, uma versão com resolução 1280x720 e outra com resolução 1920x1080. Além disso, cada teste foi realizado duas vezes, uma vez com a aplicação das etapas de borramento e outra vez sem essa aplicação.

O Quadro 1 mostra os resultados obtidos com a resolução 1280x720. Nele estão as taxas médias de *frames* por segundo que o algoritmo conseguiu obter no processamento de cada vídeo. Como as câmeras utilizadas hoje em dia costumam ter uma taxa de captura de 30 *frames* por segundo, o algoritmo conseguiria ser executado em tempo real mesmo com as etapas de borramento.

Quadro 1 - Resultados com resolução 1280x720

| | **FPS médio (sem etapas de borramento)** | **FPS médio (com etapas de borramento)** |
|---|---|---|
| **Vídeo 1** | 79,28 | 50,98 |
| **Vídeo 2** | 70,00 | 43,29 |
| **Vídeo 3** | 67,68 | 50,00 |

Fonte: elaborado pelo autor

Os resultados obtidos com a resolução 1920x1080 estão no Quadro 2. Para o processo sem as etapas de borramento, o algoritmo consegue se manter acima da taxa de 30 *frames* por segundo, podendo ser executado em tempo real com câmeras que capturem a essa taxa. Já no processo com as etapas de borramento, o algoritmo não conseguiu obter resultados satisfatórios. Portanto, para ser executado em tempo real com resolução 1920x1080, o algoritmo deve ser utilizado sem as etapas de borramento, ou então ocasionará diminuição da taxa de *frames* por segundo do vídeo de saída.

Quadro 2 - Resultados com resolução 1920x1080

| | **FPS médio (sem etapas de borramento)** | **FPS médio (com etapas de borramento)** |
|---|---|---|
| **Vídeo 1** | 59,70 | 23,25 |
| **Vídeo 2** | 60,00 | 32,28 |
| **Vídeo 3** | 55,37 | 22,89 |

Fonte: elaborado pelo autor

## 4.6 Registro de objetos virtuais com uso do Kinect

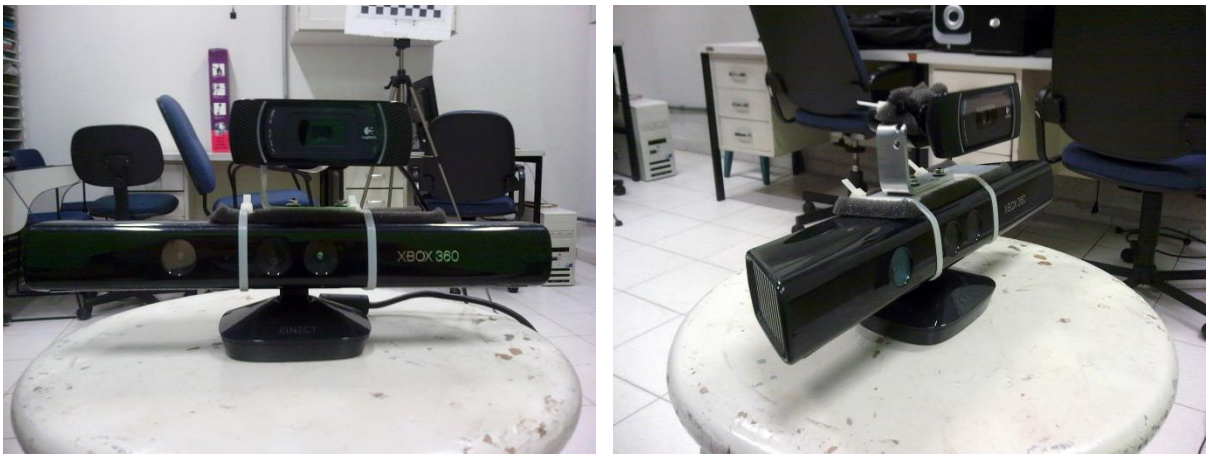
Para utilizar o Kinect com uma câmera RGB externa foi necessário realizar a calibração entre eles. Testes foram conduzidos para determinar a eficácia da calibração.

Além disso, o registro dos objetos com o Kinect foi testado para avaliar seus resultados.

### 4.6.1 Calibração do Kinect com câmera RGB externa

Nessa etapa de desenvolvimento, a calibração do Kinect com a câmera RGB externa foi efetuada. A câmera RGB foi acoplada ao Kinect para evitar a movimentação entre as duas câmeras, pois a posição entre elas precisa ser mantida para que a calibração tenha efeito. Essa acoplagem pode ser vista na Figura 36. Caso uma seja movida em relação a outra, a calibração precisa ser refeita.

Figura 36 - Câmera RGB acoplada ao Kinect




Fonte: elaborada pelo autor

Para observar os efeitos da calibração, o mapa de profundidade (representando as distâncias por tons de cinza) foi sobreposto à imagem capturada da câmera RGB externa. O resultado pode ser observado na Figura 37.

Figura 37 - Resultado da calibração do Kinect

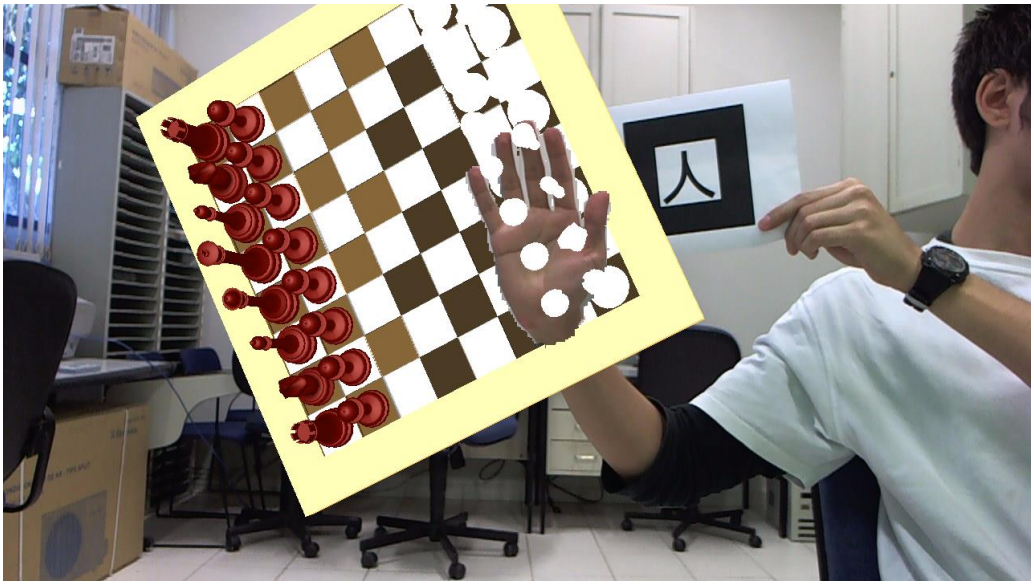
Fonte: elaborada pelo autor

### 4.6.2 Oclusão dos objetos virtuais de acordo com profundidade do mundo real

Com a conclusão dessa etapa de desenvolvimento, foi possível testar o registro dos objetos virtuais com o Kinect. A Figura 38 mostra o resultado de um teste realizado, com um objeto virtual associado a um marcador sendo atravessado

parcialmente por uma pessoa no mundo real. Nesse teste, a mão da pessoa estava entre o tabuleiro e as peças dele, por isso a mão atravessou o tabuleiro mas não atravessou completamente as peças.

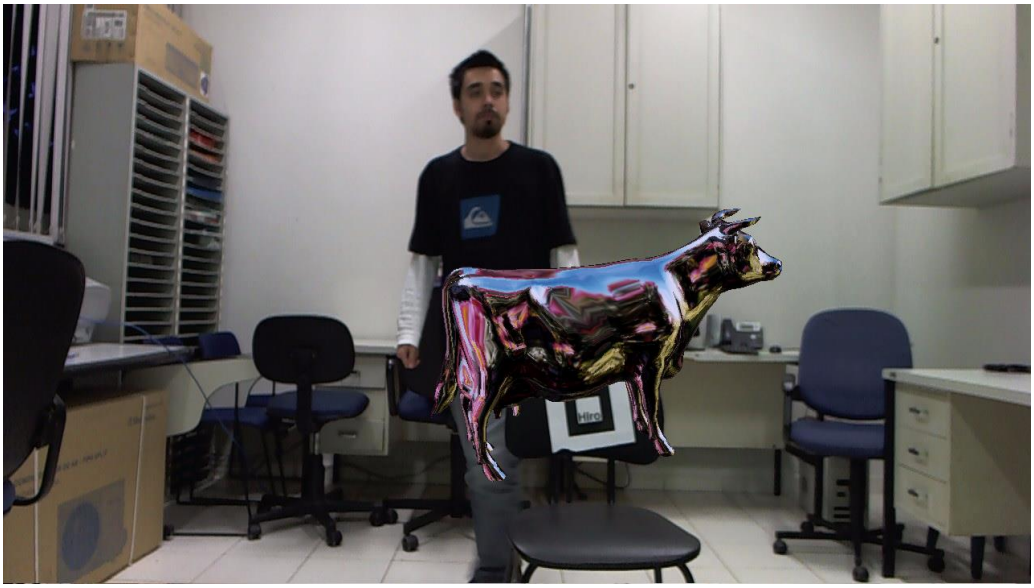
Figura 38 - Teste com tabuleiro virtual



Fonte: elaborada pelo autor

Em outro teste, um objeto virtual foi associado a um marcador colocado no meio da sala. O marcador foi então fixado utilizando a implementação de marcador fixo para evitar que sua oclusão causasse o desaparecimento do objeto. Uma pessoa passou por trás do objeto do objeto e depois pela frente do objeto para demonstrar a diferença do registro nos dois casos. O resultado desse teste pode ser visto na Figura 39.

Figura 39 - Teste de oclusão do objeto virtual com o Kinect



(a)

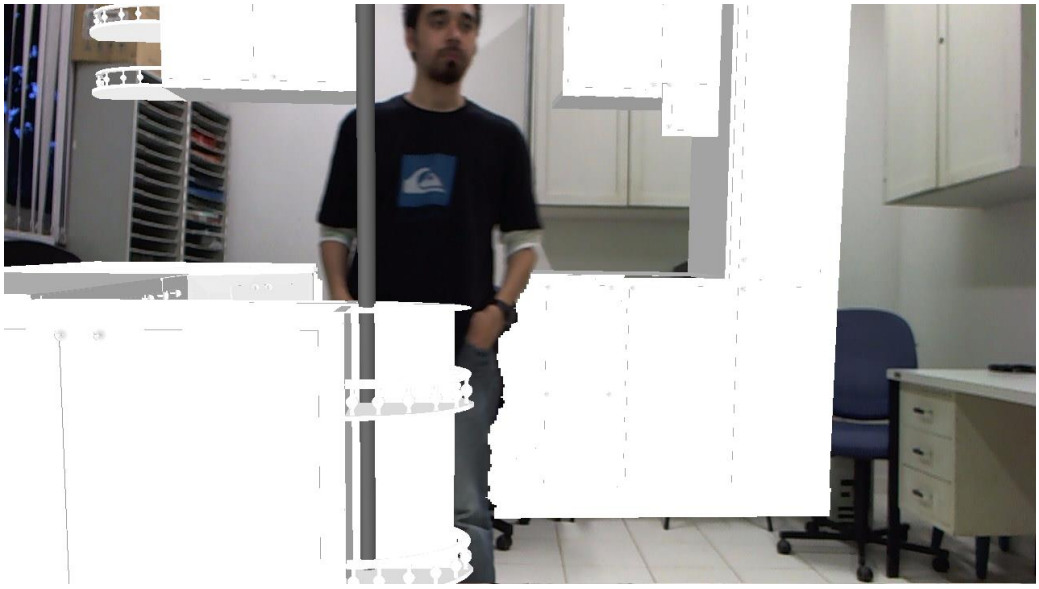
(b)

(a) Pessoa passando por trás do objeto; (b) Pessoa passando pela frente do objeto.

Fonte: elaborada pelo autor

Mais um teste foi realizado com um objeto virtual maior, para que a pessoa participando do teste pudesse entrar dentro desse objeto. A Figura 40 mostra o resultado desse teste, com um balcão virtual e uma pessoa posicionada no meio do balcão.

Figura 40 - Pessoa posicionada no meio de um balcão virtual



Fonte: elaborada pelo autor

Nesse teste, a qualidade do recorte feito pelo mapa de profundidade foi evidenciada. Por ter uma resolução baixa (640x480), o mapa de profundidade oferece pouca qualidade para o recorte.

# 5 CONCLUSÃO

Este trabalho adicionou novas funcionalidades ao sistema de estúdios virtuais ARStudio, permitindo a criação de um conteúdo mais rico em detalhes, com maior qualidade, e adicionando mais flexibilidade ao sistema, permitindo que o usuário tenha maior controle sobre o ambiente virtual criado e gerando uma maior variedade de possibilidades de uso. Isso é importante pois a criação de conteúdo é facilitada ao mesmo tempo que sua qualidade é aumentada, permitindo a produção em menor tempo e com menor custo, que é a principal vantagem dos estúdios virtuais sobre os estúdios convencionais e que possibilita aos estúdios de baixo orçamento o acesso a efeitos especiais mais complexos.

Com o armazenamento do fluxo de vídeo não processado, o vídeo capturado pela câmera pode ser guardado para ser utilizado posteriormente em uma etapa de pós-produção, com a aplicação de outros efeitos à produção que não podem ser feitos em tempo real. Essa funcionalidade reforça o uso do software ARStudio como uma ferramenta de visualização prévia da produção, pois o processo de filmagem definitivo pode ser feito ao mesmo tempo que sua visualização prévia é gerada.

Os demais recursos adicionados ao software por este trabalho firmam sua utilização como uma ferramenta de geração de conteúdo.

A implementação de marcadores fixos melhorou não só o registro dos objetos virtuais em posição estacionária, como também se mostrou uma funcionalidade útil para adicionar objetos fixos na cena virtual sem a necessidade da presença constante do marcador na cena real. Com isso, é possível utilizar os marcadores para facilitar o posicionamento de objetos virtuais em relação ao mundo real e diminuir a dependência que o registro desses objetos tem sobre o rastreamento e o reconhecimento do marcador.

O uso de multimarcadores no ARStudio também diminuiu a dependência do registro de objetos virtuais sobre o reconhecimento do multimarcador mas, diferentemente dos marcadores fixos, ele permite a movimentação dos objetos de acordo com sua movimentação no mundo real. Assim, uma outra alternativa foi oferecida para realizar o registro dos objetos virtuais com menor incidência do problema de obstrução de marcadores.

A nova técnica de *matting* digital adicionada ao ARStudio melhorou a qualidade do resultado com a aplicação de transparência dos *pixels* e pôde ser executado em

tempo real, o que possibilita a visualização prévia do resultado. Esse novo algoritmo, junto com o algoritmo de *matting* digital já existente anteriormente, permite ao usuário escolher o algoritmo que se adequa melhor à sua necessidade de uso. Em trabalhos futuros com o ARStudio, novos algoritmos de *matting* digital podem ser adicionados para aumentar seu acervo.

O uso do Kinect para efetuar o registro dos objetos virtuais em relação ao mundo real permite que atores em cena tenham maior interação com os objetos virtuais, criando uma coerência entre o posicionamento dos elementos da cena virtual em relação aos objetos no mundo real. Apesar de estar limitado à qualidade e precisão do mapa de profundidade fornecido pelo Kinect, a implementação conseguiu cumprir seu objetivo. Em trabalhos futuros, processos de visão computacional podem ser aplicados ao mapa de profundidade com a finalidade de refiná-lo, gerando melhores resultados.

# REFERÊNCIAS

ANAGNOSTOU, K.; VLAMOS, P. **Square AR: Using Augmented Reality for urban planning**. Third International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES), p. 128-131, 2011.

ANDERSEN, M. R.; JENSEN, T.; LISOUSKI, P.; MORTENSEN, A. K.; HANSEN, M. K.; GREGERSEN, T.; AHRENDT, P. **Kinect Depth Sensor Evaluation for Computer Vision Applications**. Technical report ECE-TR-6, Electrical and Computer Engineering, Aahrus University, 2012.

AZUMA, R. T. **A Survey of Augmented Reality**. Presense: Teleoperators and Virtual Environments, v. 6, n. 4, p. 355-385, 1997.

AZUMA, R. T.; BAILLOT, Y.; BEHRINGER, R.; FEINER, S.; JULIER, S.; MACINTYRE, B. **Recent Advances in Augmented Reality**. IEEE Computer Graphics and Applications, v. 21, n. 6, p. 34-47, 2001.

BLONDÉ, L.; BUCK, M.; GALLI, R.; NIEM, W.; PAKER, Y.; SCHMIDT, W.; THOMAS, G. **A Virtual Studio for Live Broadcasting: The Mona Lisa Project**. IEEE Multimedia, v. 3, n. 2, p. 18-29, 1996.

BRADSKI, G.; KAEHLER, A. **Learning OpenCV**. O'Reilly Media Inc., 2008.

BURRUS, N. **Kinect Calibration**. Disponível em: <http://nicolas.burrus.name/index.php/Research/KinectCalibration>. Acesso em: maio 2014.

CAMPOS, G. M.; MUKUDAI, L. M.; IWAMURA, V. S.; SEMENTILLE, A. C. **Sistema de Composição de Estúdios Virtuais Utilizando Técnicas de Realidade Aumentada**. Proceedings - XII Symposium on Virtual and Augmented Reality - SVR 2010, v. 1, p. 22-30, 2010.

CHUANG, Y.-Y. **New Models and Methods for Matting and Compositing**. 2004.

CHUANG, Y.-Y.; CURLESS, B.; SALESIN, D. H.; SZELISKI, R. **A Bayesian Approach to Digital Matting**. Proceedings of IEEE Computer Vision and Pattern Recognition, v. 2, p. 264-271, 2001.

FERNANDES, D. M. **Sistema Imersivo de Retorno ao Ator Orientado a Estúdio Virtual Aumentado - Módulo de Captura de Áudio e Módulo da Cena Combinada**. Bauru, SP, 2011.

FIALA, M. **ARTag, An Improved Marker System Based on ARToolkit**. NRC/ERB-1111 NRC 47166, National Research Council Canada. 2004.

GEIGER, C.; SCHMIDT, T.; STOCKLEIN, J. **Rapid development of expressive AR applications**. 3rd IEEE and ACM International Symposium on Mixed and Augmented Reality 2004 (ISMAR 2004), p. 292-293, 2004.

GIBBS, S.; ARAPIS, C.; BREITENEDER, C.; LALIOTI, V.; MOSTAFAWY, S.; SPEIER, J. **Virtual Studios: An Overview**. IEEE Multimedia, v. 5, n. 1, p. 18-35, 1998.

GRAU, O.; PRICE, M.; THOMAS, G. A. **Use of 3-D Techniques for Virtual Production**. Proceedings of SPIE 4309, Videometrics and Optical Methods for 3D Shape Measurement, 2002.

GÜNSEL, B.; TEKALP, A. M.; VAN BEEK, P. J. L. **Object-based video indexing for virtual-studio productions**. IEEE Computer Society Conference on Computer Vision and Pattern Recognition, p. 769-774, 1997.

HAILEY, K. R. **Photographic System Using Chroma-Key Processing**. U.S. Patent 6.441.865 B1, 2002.

HERRERA C., D.; KANNALA, J.; HEIKKILÄ, J. **Joint depth and color camera calibration with distortion correction**. TPAMI, 2012.

HOFF III, K. E. **Conversion Between OpenGL Depth-Buffer Z and Actual Screen-Space Depth**. Disponível em: <http://gamma.cs.unc.edu/users/hoff/techrep/openglz.html>. Acesso em: maio 2014.

HOLLYWOOD CAMERA WORKS. **Green Screen Plates**. Disponível em: <http://www.hollywoodcamerawork.us/greenscreenplates.html>. Acesso em: maio 2014.

KATO, H.; BILLINGHURST, M.; POUPYREV, I. **ARToolKit version 2.33**. 2000.

LI, H.; QI, M.; WU, Y. **A Real-Time Registration Method of Augmented Reality Based on Surf and Optical Flow**. Journal of Theoretical and Applied Information Technology, v. 42, n. 2, p. 281-286, 2012.

LOOSER, J.; GRASSET, R.; SEICHTER, H.; BILLINGHURST, M. **OSGART – A pragmatic approach to MR**. International Symposium of Mixed and Augmented Reality (ISMAR), 2006.

MARTZ, P. **OpenSceneGraph Quick Start Guide: A Quick Introduction to the Cross-Platform Open Source Scene Graph API**. Skew Matrix Software LLC, 2007.

MICROSOFT. **Kinect Fact Sheet**. 2010.

MIKKONEN, T.; TAIVALSAARI, A.; TERHO, M. **Lively for Qt: A Platform for Mobile Web Applications**. Proceedings of the 6th International Conference on Mobile Technology, Application & Systems, 2009.

ONERIVER MEDIA. **The Blackmagic Cinema Camera: Can It Run with the Big Boys**. Disponível em: < http://library.creativecow.net/solorio_marco/BMD-Cinema-Camera-holds-its-own/1>. Acesso em: maio 2014.

OSGART. **Creating Multi-Markers**. Disponível em: <http://www.osgart.org/index.php/Creating_Multi-Markers>. Acesso em: maio 2014.

PORTER, T.; DUFF, T. **Compositing Digital Images**. ACM SIGGRAPH Computer Graphics, v. 18, n. 3, p. 253-259, 1984.

RAHBAR, K.; POURREZA, H. R. **Inside looking out camera pose estimation for virtual studio**. Graphical Models, v. 70, n. 4, p. 57-75, 2008.

SCHULTZ, C. **Digital Keying Methods**. 2006.

SEMENTILLE, A. C.; MARAR, J. F.; SANCHES, S. R. R.; YONEZAWA, W. M.; CAVENAGHI, M. A.; ALBINO, J. P.; ANDRADE, A. B. **Inovação em Televisão Digital: A Aplicação de Realidade Aumentada e Virtual na Criação de Estúdios Virtuais**. In: GOBBI, M. C.; MORAIS, O. J. (Org.). Televisão Digital na América Latina: Avanços e Perspectivas, 1 ed., São Paulo: Sociedade Brasileira de Estudos Interdisciplinares da Comunicação - INTERCOM, 2012, v. 2, p. 655-680.

SHREINER, D.; SELLERS, G.; KESSENICH, J.; LICEA-KANE, B. **OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3**, 8 ed. Addison-Wesley, 2013.

SMITH, A. R. **Alpha and the History of Digital Compositing**. Microsoft Technical Memo #7, 1995.

SMITH, A. R.; BLINN, J. F. **Blue screen matting**. Proceedings of SIGGRAPH 96, p. 259–268, 1996.

THORNTON, D. **Green Screen Gallery**. Disponível em: <http://www.davethorntonphotography.co.uk/greenscreen_gallery.html>. Acesso em: maio 2014.

VAN DEN BERGH, F.; LALIOTI, V. **Software chroma keying in an immersive virtual environment**. South African Computer Journal, n. 24, p. 155-162, 1999.

VLAHOS, P. **Electronic Composite Photography**. U.S. Patent 3.595.978, 1971.

WANG, J.; COHEN, M. F. **Image and Video Matting: A Survey**. Foundations and Trends in Computer Graphics And Vision, v. 3, n. 2, p. 97-175, 2007.

WANG, R.; QIAN, X. **OpenSceneGraph 3.0: Beginner's Guide**. Packt Publishing Ltd, 2010.

WRIGHT, S. **Digital Compositing for Film and Video**. Focal Press Visual Effects and Animation, 2010.

ZHANG, Z. **Microsoft Kinect Sensor and Its Effect**. IEEE Multimedia, v. 19, n. 2, p. 4-10, 2012.

ZHENG, Y.; KAMBHAMETTU, C. **Learning Based Digital Matting**. IEEE 12th International Conference on Computer Vision, p. 889-896, 2009.