Universidade Federal de Pernambuco
Centro de Informática
Graduação em Ciência da Computação

OpenStereo: Uma Biblioteca para Suporte ao Desenvolvimento de Aplicações Estereoscópicas

Pedro José Silva Leite
pjsl@cin.ufpe.br

Trabalho de Graduação submetido ao Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para a obtenção do Grau de Bacharel em Ciência da Computação.

Orientadora: Judith Kelner (jk@cin.ufpe.br)
Co-orientadora: Veronica Teichrieb (vt@cin.ufpe.br)

RESUMO

Nos últimos anos, aplicações 3D têm aumentado o seu grau de realismo por prover meios de imersão. Luvas, óculos 3D, capacetes estão sendo, comumente, utilizados em tais aplicações. Por outro lado, com o aumento de realismo através da imersão, cresceu também a demanda de *hardwares* especializados, especialmente os voltados para estereoscopia.

Nas aplicações estereoscópicas, um dos fatores mais importantes é a renderização. Nas placas gráficas mais modernas é dado um suporte nativo para conversão de aplicações 3D em estereoscópicas, a empresa NVIDIA, por exemplo, suporta aplicações estereoscópicas DirectX nativamente em suas placas gráficas. Já a empresa ATI provê suporte ao estéreo alternado, porém requer *hardware* adicional, tais como, *Head Mounted Displays* ou LCD *shutter glasses*. Somente nos casos de aplicações implementadas em OpenGL, tal suporte é conseguido com o uso de placas gráficas profissionais, mais o custo dessas placas ainda é muito elevado.

Outras empresas não provêem suporte a aplicações estereoscópicas, ficando com o desenvolvedor a responsabilidade de tal implementação. Portanto, desenvolver aplicações estereoscópicas implica no uso do sistema operacional Microsoft Windows e da biblioteca gráfica DirectX, ou então investir em placas gráficas profissionais caras e equipamentos para visualização 3D.

Neste contexto, o objetivo deste trabalho, denominado OpenStereo, é fornecer o suporte à conversão de aplicações OpenGL existentes em aplicações estereoscópicas, independentemente da placa gráfica ou do sistema operacional. Este trabalho desenvolveu uma ferramenta que através de um subsistema de *plugins* é capaz de transformar uma aplicação de OpenGL numa aplicação estereoscópica. O usuário escolhe a cena a ser renderizada, e a partir deste ponto o OpenStereo gera os pares esquerda/direita, cria a imagem estéreo e finalmente renderiza a cena em um *viewport*. A arquitetura do OpenStereo foi construída para ser facilmente integrada em aplicações 3D existentes. Junto com o OpenStereo existe um *Software Development Kit* para desenvolvimento de *plugins*.

Palavras-chave: Estereoscopia, Realidade Virtual, OpenStereo, Percepção de Profundidade, Anáglifo.

AGRADECIMENTOS

Não existe um trabalho sequer em que somente uma pessoa faça tudo. E não será este a exceção, portanto agradecerei aqui todas as pessoas que indireta ou diretamente participaram desse trabalho.

Agradeço de cara à minha mãe (“mainha” ou “véia” para os filhos), que mesmo com os seus “vá comer”, “vá dormir” sempre esteve preocupada com o meu bem-estar. Sem todo o esforço dela, a educação que eu tive e tenho não existiria. Agradeço também aos meus “irmamãos” (todo irmão tem um certo carinho pelos outros =) Agenor e Tetê, que também se preocupam comigo.

Agradeço aos meus amigos mais próximos, que sempre me ajudaram em tudo, seja psicologicamente, emocionalmente, financeiramente, profissionalmente, etc. “Patríciammm”, que conheço desde o colégio, sempre esteve ao meu lado, seja em projetos do CIn, farras, decisões pessoais, profissionais, etc. e que é uma pessoa maravilhosa, com inúmeras qualidades. Simplesmente a pessoa mais inteligente que conheço. “Galega”, também conhecida como “pinta” ou Diana, que sempre tem uma palavra de tranqüilidade e amizade. Thiago, que eu gosto muito de tirar onda, mas que é como um irmão e também está sempre me ajudando. “Cara de concha”, conhecida como Renata, que me ajuda a acordar de manhã sempre que preciso, que sempre me dá forças para continuar trabalhando, que é a pessoa mais paciente que conheço. Papa, conhecido como Papa mesmo, apesar de ser raramente chamado de Roberto, que é um cara muito inteligente, apesar de preguiçoso assim como eu, que é com quem eu discuto milhões de assuntos relacionados à informática, onde todos se misturam com assuntos do “mundo real” e com cachaça. Nancy, que se mostrou uma amiga (im)paciente, sempre me dando conselhos, expondo pontos de vista, entrando em conflitos comigo, ajudando-me emocionalmente, profissionalmente, dando carona (porém bem menos que Patrícia), enfim uma pessoa que, como ela mesma disse, “ganhou passagem direto pro céu” por me aturar. Shi, também conhecida como Shirley, a primeira amizade que fiz no curso, pessoa sempre alegre, disposta a ajudar e com um coração enorme. Syl, também conhecida como Sylvinha ou Sylvia, que é de uma energia e alegria imensuráveis. Raul, por sempre me manter focado com os seus “vai trabalhar Pedro...” nas horas certas e por me ajudar a “desopilar” desse mundo de informática.

Ao GPRT, que é um ambiente fantástico e sensacional, com pessoas dispostas a trabalhar sempre ajudando uns aos outros. Existem poucos ambientes de trabalho como esse e eu me orgulho bastante de ter a oportunidade de trabalhar nele.

Não poderia deixar de agradecer a Márcio que me emprestou um par de óculos 3D anáglifo e a Guilherme, que além de ter emprestado-me um par desses óculos, me ajudou a criar as fotos estereoscópicas desse trabalho.

Agradeço também à melhor gerente que já tive, Veronica, que sempre tem uma palavra para me acalmar quando as coisas parecem ir por água abaixo, que me ajudou bastante dentro do grupo e que acreditou em mim, permitindo desenvolver esse trabalho como um projeto do GRVM.

À Judith, ou Professora, que foi uma orientadora de fazer inveja, ajudando até o último momento (não é todo aluno do CIn que tem isso ;), sem mencionar as dicas profissionais, acadêmicas, pessoais...

Por fim, agradeço a você leitor que conseguiu ao menos chegar ao fim dessa seção de agradecimentos (isso é um plágio do TG de meu amigo “Pinto”, ou Marden).

Conteúdo

Lista de Figuras

Lista de Tabelas

1 INTRODUÇÃO

A utilização da computação gráfica para visualização científica e modelagem geométrica tridimensional (3D) tem trazido mudanças fundamentais aos métodos de trabalho nos últimos anos, tanto na indústria como em pesquisa e desenvolvimento. A possibilidade de visualização e manipulação interativa de modelos virtuais com auxílio do computador tem revolucionado muitas atividades, pois permite a compreensão e análise de grandes quantidades de informação, de natureza espacial, com eficiência sem precedentes, explorando a capacidade humana de raciocinar e se comunicar visualmente.

Do ponto de vista do usuário final, a necessidade crescente de imersão nas aplicações decorreu do objetivo de prover uma experiência mais realística, melhor interação e aumentar a sua proficiência na utilização delas. Objetos 3D, efeitos visuais de iluminação, sombreamento, entre outros, estão cada vez mais presentes nas aplicações comuns. Novas *shells*[1], jogos e aplicações de simulação, por exemplo, fazem uso de bibliotecas gráficas 3D para oferecer um maior grau de realismo e aumentar a experiência do usuário nessas aplicações. O Xgl [1] e o Celestia [2] exemplificam perfeitamente um *shell* e uma aplicação de simulação 3D e são mostrados nas figuras 1 e 2, respectivamente.

Figura 1: Quake 3 sendo jogado no desktop Xgl

1 Em sistemas operacionais, *shell* é a interface gráfica que provê acesso às suas funções.

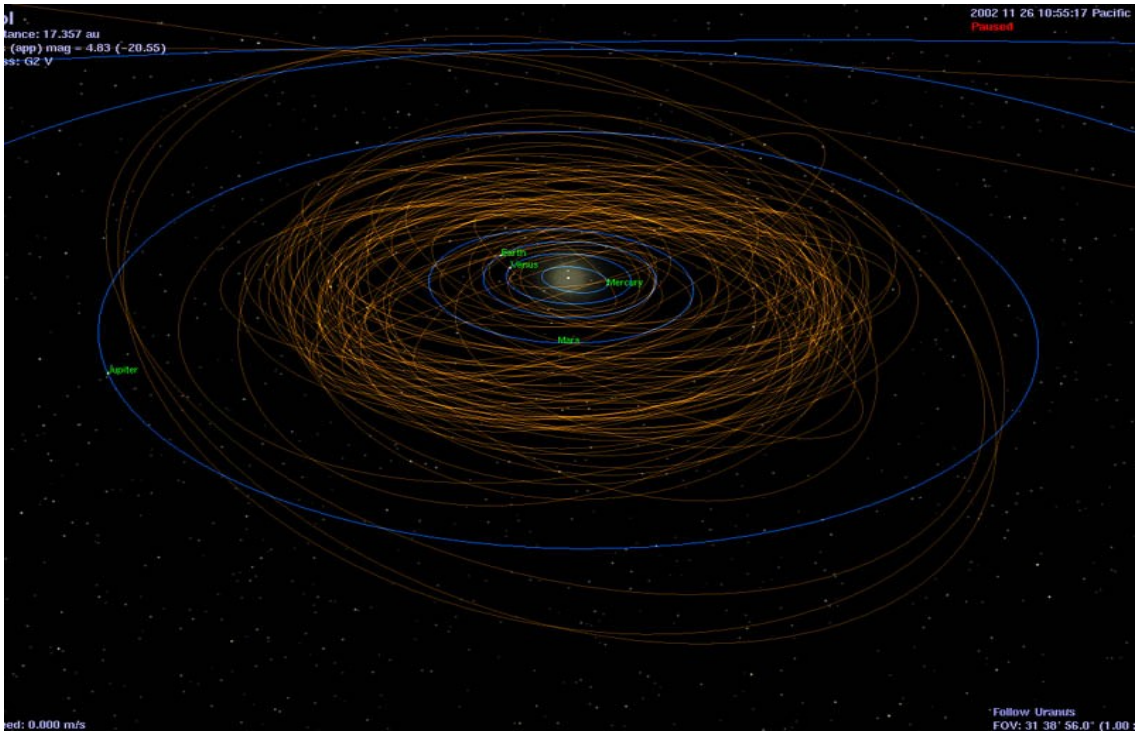

Figura 2: Órbitas de alguns asteróides e planetas visualizados no Celestia

Aplicações 3D, geralmente, são implementadas com o suporte de bibliotecas gráficas, como OpenGL [3] e DirectX [4]. Essas bibliotecas possibilitam o desenvolvimento de aplicações de alto desempenho, abstraindo o acesso ao *hardware* gráfico através de uma API (*Application Programming Interface*) de alto nível. Apesar da tela do computador ser bidimensional (2D), a percepção 3D decorre de algumas propriedades, tais como perspectiva, iluminação, oclusão, gradiente de textura, etc. Porém, não é possível perceber o quão profundo está um objeto sem o auxílio de equipamentos específicos, como *Head Mounted Displays* (HMDs), óculos vermelho-ciano, óculos obturadores, entre outros.

Com o objetivo de aumentar a imersão do usuário, aplicações 3D podem ser convertidas para aplicações estereoscópicas, permitindo desta forma uma correta percepção de profundidade do ambiente.

1.1 Definição do problema

Como dito anteriormente, a imersão do usuário em um ambiente 3D pode ser aumentada com a utilização de estereoscopia. O processo de se converter uma aplicação 3D em estereoscópica está relacionado com o *hardware* no qual a aplicação será executada. Geralmente se tem a relação custo-benefício diretamente proporcional à facilidade de conversão: quanto mais se investir em *hardwares* especializados, mais fácil se tornará converter alguma aplicação 3D em estereoscópica. Em

contrapartida, quanto menor for o investimento, mais difícil se tornará tal conversão.

Algumas placas de vídeo fornecem suporte à conversão de aplicações 3D em estereoscópicas. A empresa NVIDIA [5], por exemplo, fornece suporte a estéreo 3D nativamente, em aplicações desenvolvidas sobre o OpenGL, somente em suas placas Quadro, que possuem um alto custo. Para as placas mais acessíveis, o suporte completo é dado somente a aplicações desenvolvidas usando DirectX. Uma implicação direta disso é que aplicações estereoscópicas podem se tornar custosas, caso o ambiente de execução necessite a biblioteca gráfica OpenGL. Outro fator importante é que as aplicações ficam limitadas ao Sistema Operacional (SO) Microsoft Windows ®, uma vez que para outros SOs este suporte não é fornecido.

A empresa ATI [6] fornece suporte para o DirectX utilizando a técnica de *page flipping*[2], que requer um *hardware* adicional para a visualização estereoscópica. Esse *hardware* adicional consiste de um HMD, cuja aquisição pode se tornar inviável para usuários domésticos.

Outras fabricantes de placas gráficas não fornecem suporte a aplicações estereoscópicas, ficando o desenvolvedor responsável por tal implementação.

1.2 Objetivos do trabalho

Dado que o suporte nativo à estereoscopia pelas placas gráficas ou é de custo elevado ou restringe o ambiente de execução da aplicação, este trabalho de graduação propõe o desenvolvimento de uma biblioteca chamada OpenStereo. O OpenStereo tem como objetivo principal fornecer suporte à conversão de aplicações 3D de tempo real, baseadas em OpenGL, em aplicações estereoscópicas. O suporte é provido independente de placa gráfica ou de SO, implicando diretamente na viabilização dos custos do desenvolvimento dessas aplicações e aumentando a quantidade de plataformas suportadas para a sua execução.

1.3 Estrutura do trabalho

Este documento apresenta o OpenStereo, biblioteca cujo objetivo principal é auxiliar a criação de aplicações estereoscópicas. O primeiro capítulo forneceu uma introdução à importância da percepção de profundidade como fator de imersão do usuário em aplicações 3D, bem como abordou os principais problemas envolvidos em se trazer essa imersão ao usuário.

O segundo capítulo fornece conceitos básicos sobre estereoscopia, alguns métodos

2 Técnica que utiliza dois *buffers* de memória para evitar a cópia excessiva de dados.

estereoscópicos existentes, conceitos matemáticos envolvidos, além de ressaltar a importância da percepção visual e sonora para o ser humano.

O terceiro capítulo apresenta a proposta da biblioteca OpenStereo, descrevendo sua arquitetura, como foi implementada além das vantagens em sua utilização na geração de conteúdo estereoscópico.

O quarto capítulo demonstra o processo de utilização do OpenStereo em uma aplicação existente, ressaltando o desempenho envolvido e mostrando o ganho na imersão do usuário.

No quinto capítulo são expostas as conclusões obtidas durante a pesquisa e o desenvolvimento deste trabalho final de graduação, apontando as principais contribuições e enumerando alguns possíveis trabalhos futuros.

Por fim, o Apêndice A contém parte do código utilizado no desenvolvimento do OpenStereo e seus *plugins*.

2 Percepção e Estereoscopia

Antes de introduzir o OpenStereo, é necessário que alguns tópicos como percepção de profundidade, estereoscopia, técnicas estereoscópicas, entre outros, sejam explanados. Neste capítulo será descrita a importância da percepção na vida humana. Então será explorada a percepção de profundidade, como base para o entendimento da estereoscopia.

2.1 Percepção sonora

O ser humano consegue perceber sons através da vibração do tímpano [7] e com isso podem inferir de onde eles vêm. Segundo [8], "*o processo de percepção não é somente produzido da sensação de algum estímulo (um carro na rua, por exemplo), mas também através de uma interpretação no contexto da experiência anterior*". Sendo assim, a percepção do som só se realiza com uma interpretação posterior à sua sensação.

Caso um carro em alta velocidade siga em direção a um pedestre, o motorista pode buzinar na tentativa de evitar algum acidente. O pedestre sente o som, interpreta-o e após perceber a fonte sonora aproximando-se de si, toma uma atitude, que provavelmente será sair da frente do carro, evitando que algo de ruim aconteça a ele. Nesse sentido, a percepção sonora é vital ao ser humano, pois acidentes podem ser evitados. O mesmo se aplica à vida animal selvagem, onde o predador necessita ter uma boa noção visual e sonora para caçar sua presa. Esses requisitos também são necessários para que a presa consiga fugir. A Figura 3, tira do *cartoon* Garfield, mostra uma cena de reação provocada pela percepção sonora, onde Jon Arbuckle vira-se para trás ao perceber que Garfield riu muito alto.

Figura 3: Reação provocada pela percepção sonora

2.2 Percepção de profundidade

Assim como a percepção de sons, o ser humano, através da visão, consegue perceber a profundidade

dos objetos. O mesmo caso do carro (na seção anterior) exemplifica a importância da percepção de profundidade. Se o carro vier pela frente do pedestre ao seu encontro, o pedestre simplesmente pode desconsiderar a buzina e sair do caminho, evitando o acidente, isso porque, antes de qualquer outra percepção, o pedestre notou o carro vindo em sua direção e teve noção do quão distante ele se encontrava. Ainda na vida animal, os predadores precisam de uma correta percepção de distância até a presa, o que representa o sucesso ou fracasso na captura da caça.

Muitas são as dicas visuais que ajudam a indicar a profundidade de objetos em uma determinada cena. [9] mostra algumas delas, porém os resultados somente levam à percepção de profundidade em cenas 2D. Para que haja uma correta percepção de profundidade, é necessário que a cena seja mostrada de pelo menos dois pontos de vistas diferentes, com uma pequena distância entre eles. Esse é o funcionamento básico da visão humana, onde cada olho recebe uma imagem diferente e as mesmas são fundidas, deixando ao cérebro a responsabilidade de interpretá-la tridimensionalmente.

2.3 Estereoscopia

De acordo com [10], todos os animais que têm dois olhos possuem uma visão binocular. Ainda, segundo [10], visão estereoscópica "*significa literalmente 'vista sólida' e se refere à percepção visual de estruturas tridimensionais do mundo, quando vistas por um ou ambos os olhos.*" Em computação, estereoscopia é uma técnica de representação de objetos 3D que cria uma ilusão de profundidade a partir de uma ou mais imagens.

No decorrer da evolução, alguns animais (incluindo o homem) perderam o campo visual de 360 graus a favor de uma visão direcionada, com os olhos postados à frente da cabeça. A percepção de profundidade, nessa visão direcionada, ocorre quando se apresenta uma imagem para cada olho. Essas imagens devem possuir o mesmo foco, porém com um pequeno deslocamento do ponto de visão, responsável pela geração de pequenas diferenças entre as imagens, quase imperceptíveis quando observadas separadamente. Essas pequenas diferenças são o que causam a percepção 3D do ambiente no cérebro.

Na natureza, há muitos animais que possuem um par de olhos localizados na parte frontal da cabeça. Esta característica é típica dos tipos caçadores, como leões, gatos, gaviões, corujas, enrte outros. Isto é importante para manter a sobrevivência da espécie, pois tais animais precisam de uma correta percepção de distância até a presa. Por outro lado, animais que tipicamente se tornam presas dos caçadores, como coelhos, gazelas ou roedores, têm olhos posicionados lateralmente, o que não lhes dá uma boa percepção de distância, mas permite uma observação contínua do que há em sua volta

para, ao menor sinal de perigo, poder escapar.

Os olhos humanos estão em média a 65 milímetros um do outro e podem convergir de modo a cruzarem seus eixos em qualquer ponto, a poucos centímetros à frente do nariz, tornando-se estrábicos. Podem também divergir ou ficarem paralelos, quando focam algo no infinito. Os eixos visuais dos animais que têm olhos laterais e opostos, obviamente, nunca se cruzam. Essa capacidade visual permite que o ser humano foque em objetos mais próximos ou distantes de si, uma vez que ele tem uma noção de profundidade no ambiente onde está situado.

A estereoscopia foi fundamental para o homem, pois seu ancestral que utilizava o arborismo como meio de sobrevivência, necessitava de uma noção precisa da distância entre um galho e outro.

Ambientes virtuais estereoscópicos, por exemplo, geram imagens ao apresentar uma perspectiva da imagem para cada olho. Como resultado o usuário percebe uma única e verdadeira imagem 3D que parece existir na frente e atrás da superfície física da tela. A estereoscopia acaba sendo importante para os humanos, não somente no seu cotidiano, pois em ambientes virtuais ter uma noção de profundidade otimiza a operação de algumas tarefas. Por exemplo, profissionais da área de petróleo podem se beneficiar de ferramentas existentes para planejamento de poços de petróleo [11], porém sem a utilização de ambientes virtuais estereoscópicos, como CAVEs (*Cave Automatic Control Environments*), a realização desse planejamento da forma mais acurada possível se torna difícil. Com a popularização de aplicações 3D e com a necessidade de salas de visualização estereoscópicas, alternativas mais baratas como a Upponurkka [12] estão surgindo, provando que a visualização estereoscópica, além de necessária, é viável.

2.3.1 Efeitos psicofisiológicos da estereoscopia

Para explicar a estereoscopia e alguns dos seus efeitos psicofisiológicos pode-se explorar o seguinte exemplo: quando duas caixas acústicas são ligadas e transmitem sons diferentes, pode-se instintivamente distinguir onde se encontra a posição virtual da fonte sonora. Esse mesmo paradigma pode ser aplicado para a visão.

Para ter a capacidade de se perceber distâncias, o cérebro deve realizar análises complexas de grande quantidade de informações. Os sistemas computacionais atuais, ainda, não possuem capacidade para fazer análises tão complexas, em tempo real. Para se ter uma idéia da quantidade de informação a ser analisada, é preciso partir do sistema visual humano. O olho humano pode produzir informação instantânea capaz de sobrecarregar uma rede de computadores, de escala mundial, numa fração de segundos. Por isto, o nervo ótico é considerado o canal de comunicação mais denso

conhecido, até então, pelo Homem.

Além do grande volume de informação enviado ao cérebro, também é necessária uma análise da imagem a fim de se estabelecer a correlação da informação para encontrar os pontos comuns nas duas imagens percebidas. Isso parece ser uma tarefa trivial quando se olha para imagens correlatas em uma cena. No entanto, o cérebro deve realizar muitas operações para fornecer uma interpretação que pareça evidente à primeira vista. Lembrando que as imagens chegam invertidas na retina, conforme ilustra a Figura 4, é difícil conceber um algoritmo que tenha um desempenho razoável, em tempo real.

Dado que ambos os hemisférios do cérebro estão quase isolados um do outro, não seria possível realizar uma análise estereoscópica se cada um recebesse apenas uma informação monoscópica. Além disso, no caso de um dos hemisférios não funcionar, o homem seria *stereoblind*[3]. De acordo com [10], aproximadamente 2% da população (excluindo-se os deficientes visuais totais) são *stereoblind*. A principal causa pode ser de natureza neurológica (mau funcionamento do cérebro com relação à análise estereoscópica) ou de natureza óptica (falta de acuidade em um dos olhos).

Na computação, existem diversas técnicas para reproduzir efeitos estereoscópicos. Algumas delas serão explicadas adiante neste trabalho, levando-se em consideração suas vantagens e desvantagens.

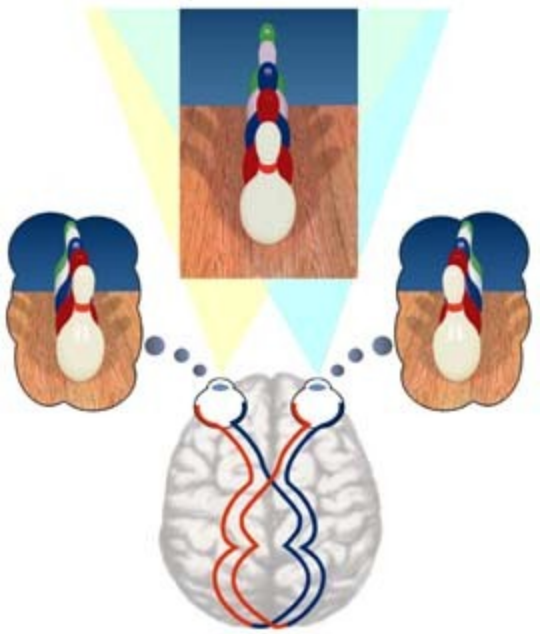

Figura 4: Imagens formadas pela retina e processadas pelos hemisférios do cérebro

3 Pessoa com dificuldade de perceber profundidade numa cena.

2.3.2 Técnicas de estereoscopia

Nesta seção serão apresentadas algumas das principais técnicas utilizadas para a geração de conteúdo estereoscópico, incluindo suas vantagens e desvantagens.

2.3.2.1 Estéreo anáglifo

É a técnica mais antiga conhecida para se obter o efeito estéreo, sendo muito utilizada na indústria cinematográfica, tendo sido introduzida por Ducos du Hauron, em 1981 [13]. Nesta técnica, utiliza-se a filtragem de cores. Inicialmente apenas duas cores são utilizadas: azul e vermelho. Um filtro vermelho é aplicado sobre a imagem do olho esquerdo, enquanto que um filtro azul é aplicado sobre a imagem do olho direito. A Figura 5 mostra como é realizado o processo de junção das imagens. Para visualizar a imagem são necessários óculos como o da Figura 6. As lentes dos óculos são projetadas de tal forma que o olho esquerdo (lente vermelha) filtra a imagem vermelha, bloqueando a imagem azul, o mesmo ocorrendo com o olho direito (lente azul) onde a imagem azul é filtrada e a vermelha é bloqueada.

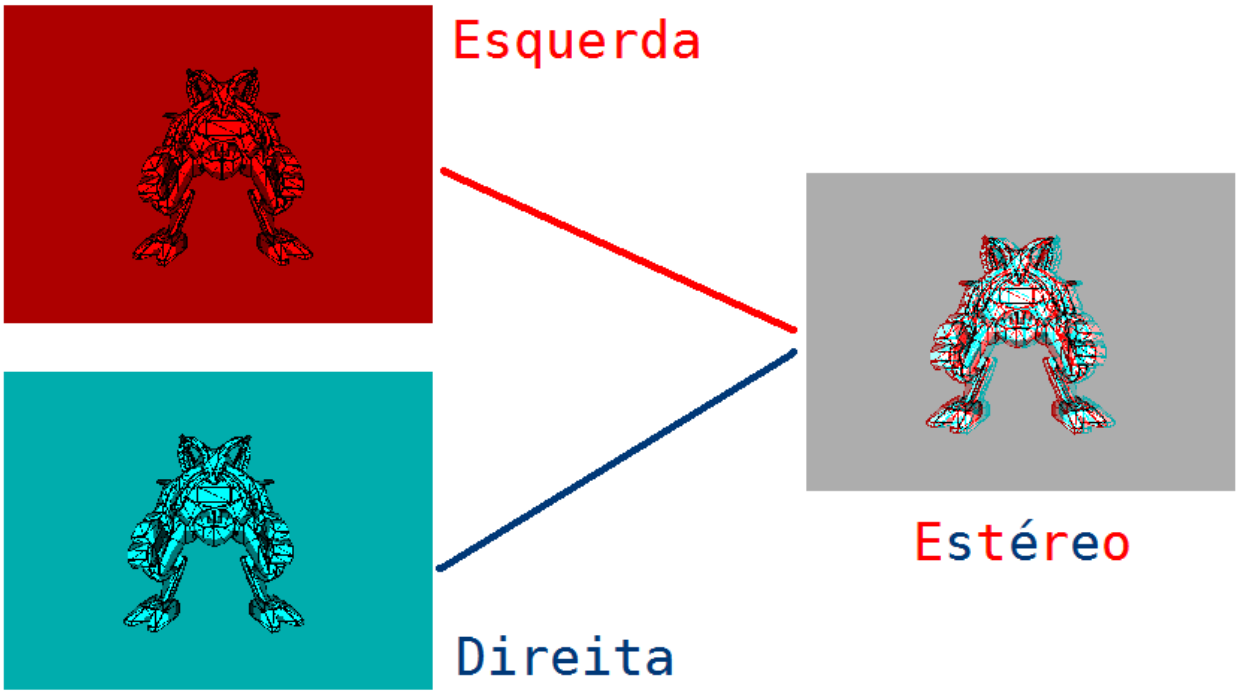

Figura 5: Criação de imagem estereoscópica usando o método anáglifo

Esta técnica é implementada através da aplicação de filtros às imagens. De acordo com o filtro, o resultado visual pode ser bastante diferente expondo ou escondendo informações da imagem. Na Figura 5, as imagens direita e esquerda já possuem um filtro aplicado em cada uma. Para a criação da

imagem estéreo, foi realizada uma sobreposição das duas imagens, adicionando as cores de *pixels* que possuíam a mesma coordenada.

Figura 6: Óculos 3D para visualizar anáglifos

As imagens adquiridas, geralmente, são como as da Figura 7, onde cada uma possui uma grande quantidade de cores. É praticamente impossível representar tais cores em uma imagem estereoscópica, pois sempre ocorrerá perda de cores em uma ou em ambas as imagens após a fusão. Todavia, um filtro pode ser cuidadosamente escolhido para minimizar essa perda de informação, bem como para melhorar o resultado visual.

Figura 7: Par de imagens para geração de conteúdo estereoscópico

Alguns filtros [14] mais comuns são: o anáglifo verdadeiro, o anáglifo cinza, o anáglifo colorido, o anáglifo semi-colorido e o anáglifo otimizado. Apenas os filtros mais importantes como algumas das suas propriedades serão explicadas a seguir.

Utilizando-se o par de imagens da Figura 7, pode-se aplicar o filtro anáglifo verdadeiro e obter o resultado da Figura 8. Para obter tal resultado, a imagem deve possuir informações da cor vermelha obtidas da imagem da esquerda e informações da cor azul obtidas da imagem da direita, por causa da natureza dos óculos (a lente vermelha permite a cor vermelha passar e bloqueia as cores azul e verde, e

a lente azul bloqueia o vermelho, permitindo o azul e o verde passarem). Isso é possível ao se aplicar a Equação 1 a cada *pixel* das imagens esquerda e direita e armazenando o resultado na imagem final[4].

Figura 8: Anáglifo verdadeiro

Na Equação 1, os valores r_a , g_a e b_a representam, respectivamente, as cores RGB (vermelho, verde e azul) da imagem anáglifo. Os valores r_e , g_e e b_e representam os valores RGB da imagem esquerda e, por fim, os valores r_d , g_d e b_d representam os valores RGB da imagem direita.

$$\begin{pmatrix} r_a \\ g_a \\ b_a \end{pmatrix} = \begin{pmatrix} 0{,}299 & 0{,}587 & 0{,}114 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} r_e \\ g_e \\ b_e \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0{,}299 & 0{,}587 & 0{,}114 \end{pmatrix} \cdot \begin{pmatrix} r_d \\ g_d \\ b_d \end{pmatrix}$$

Equação 1: Filtro anáglifo verdadeiro

Analisando-se a Equação 1, tem-se que a imagem anáglifo não possui informação da cor verde em qualquer de seus *pixels*. Uma das vantagens desse método é que ele oferece a melhor percepção de profundidade. Em contrapartida, perde informações sobre cores, além da intensidade da imagem.

A Figura 9 mostra o resultado obtido ao se aplicar o filtro anáglifo cinza, mostrado na Equação 2. A imagem anáglifo, ao ser vista com óculos semelhantes ao da Figura 6 apresenta informações em escala de cinza, uma vez que as cores azul e vermelha são bloqueadas pelo filtro. Tal resultado é conseguido aplicando-se uma distribuição de cores uniforme das imagens esquerda e direta (com mesma intensidade).

Pela Equação 2, a imagem anáglifo cinza possui informações de cores com pesos iguais, tanto para a sua componente vermelha, obtida da imagem esquerda, quanto para suas componentes verde e azul, ambas obtidas da imagem direita. O anáglifo cinza possui a vantagem de facilitar a percepção da

4 A figura é chamada de anáglifo verdadeiro porque possui apenas informação relativa às cores azul e vermelha.

profundidade, porém tem a mesma desvantagem em relação à perda de cor do anáglifo verdadeiro.

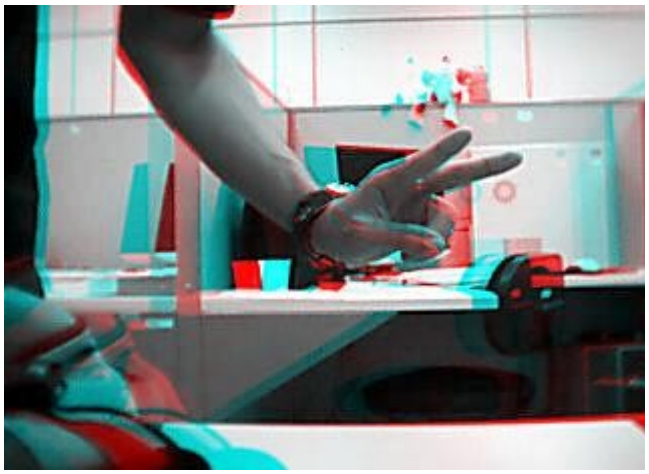

Figura 9: Anáglifo cinza

$$\begin{pmatrix} r_a \\ g_a \\ b_a \end{pmatrix} = \begin{pmatrix} 0{,}299 & 0{,}587 & 0{,}114 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} r_e \\ g_e \\ b_e \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0{,}299 & 0{,}587 & 0{,}114 \\ 0{,}299 & 0{,}587 & 0{,}114 \end{pmatrix} \cdot \begin{pmatrix} r_d \\ g_d \\ b_d \end{pmatrix}$$

Equação 2: Filtro anáglifo cinza

A Figura 10 mostra o resultado obtido ao se aplicar o filtro anáglifo colorido, definido pela Equação 3. Tal resultado é conseguido extraindo-se apenas a cor vermelha da imagem da esquerda e as cores azul e verde da imagem da direita.

Uma vantagem imediata desse método é a reprodução parcial da cor, uma vez que sempre haverá perda de cores, como já dito anteriormente.

Figura 10: Anáglifo colorido

$$\begin{pmatrix} r_a \\ g_a \\ b_a \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} r_e \\ g_e \\ b_e \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} r_d \\ g_d \\ b_d \end{pmatrix}$$

Equação 3: Filtro anáglifo colorido

As principais desvantagens da técnica anáglifo são a sua dificuldade em reproduzir as cores das imagens originais e a presença de *crosstalk* ou *ghosting*. Segundo [15], "*crosstalk ou ghosting é a apresentação de uma imagem a um olho, quando ela deveria ser apresentada exclusivamente ao outro olho*".

Dentre as vantagens da técnica anáglifo, uma delas é o baixo custo envolvido na sua utilização, pois a renderização de tais imagens, quando existe o par esquerda-direita, pode ser realizada sem a necessidade de *hardwares* especiais. O único dispositivo necessário é um óculos como o da Figura 6, que pode ser facilmente confeccionado e tem um baixo custo. Outra grande vantagem é que o material estereoscópico pode ser impresso, não restringindo a sua visualização apenas a telas e outros dispositivos.

2.3.2.2 Disparidade cromática

Esta técnica utiliza a refração da luz para simular o efeito de profundidade em imagens 2D. Raios de luz com vários comprimentos de onda, isto é, com diversas cores, são projetados em diferentes posições na retina e o cérebro humano interpreta essas posições na retina como distâncias diferentes entre os objetos e o observador [16]. Cores quentes, como o vermelho, parecem estar mais perto do que cores frias, como o azul. A Figura 11 mostra as cores utilizadas nessa técnica.

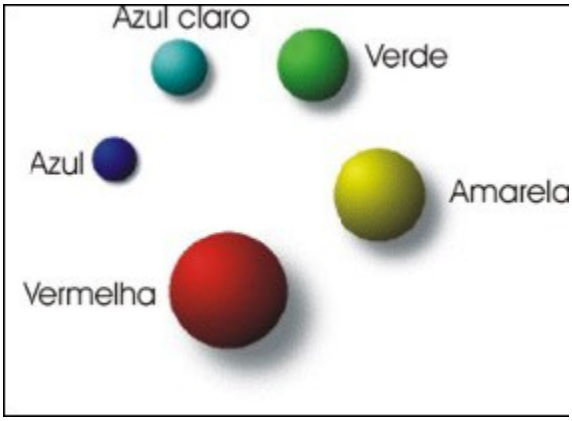

Figura 11: Cores utilizadas na técnica de disparidade cromática

Para visualizar as imagens 2D que possuem informação de profundidade codificadas nessa técnica, como na Figura 12, é necessária a utilização de óculos especiais, chamados óculos

CromaDepth™, da Chromatek [17].

As principais vantagens dessa técnica é que ela é de custo baixo e pode ser impressa. Uma grande desvantagem é que as cores que podem ser utilizadas nas imagens fica restrita àquelas mostradas na Figura 11, além de que esta técnica só funciona para imagens estáticas.

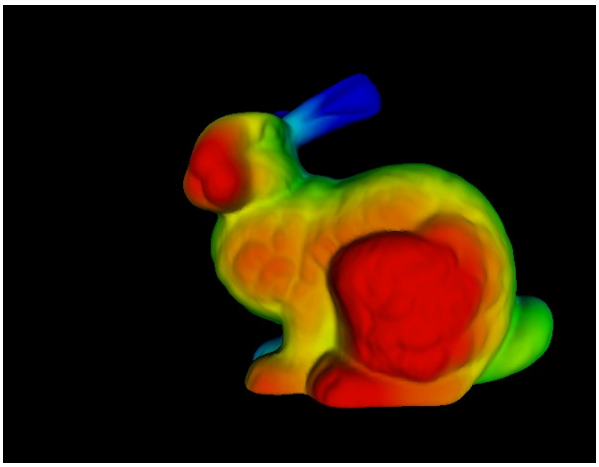

Figura 12: Modelo 3D apresentado na técnica de disparidade cromática

2.3.2.3 Polarização da luz

Esta técnica é similar à do estéreo anáglifo. A apresentação das imagens é feita por dois projetores, com luzes polarizadas, conforme a Figura 13, em eixos ortogonais. Um dos projetores apresenta as imagens para o olho esquerdo, geralmente com a luz polarizada no eixo horizontal, enquanto que o outro apresenta as imagens para o olho direito, com a luz polarizada no eixo vertical.

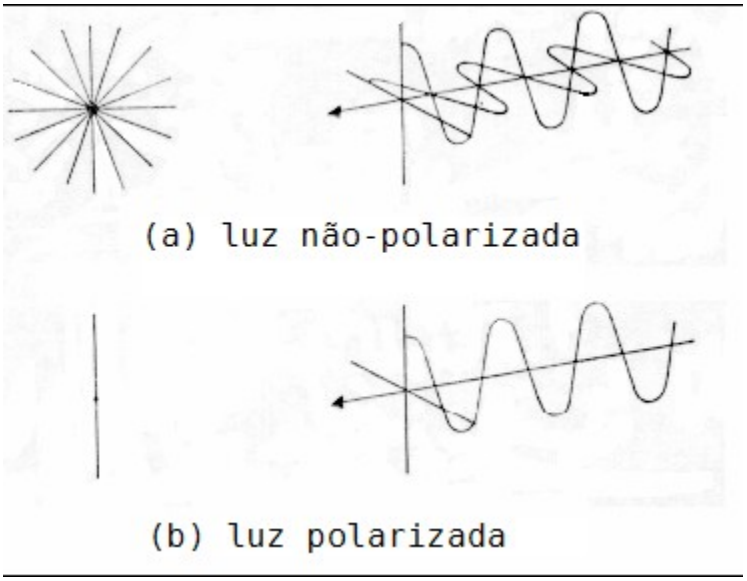

Figura 13: Luz não-polarizada e luz polarizada

Esta técnica necessita do uso de óculos especiais com filtros específicos para aquelas luzes. Em apresentações onde mais de uma pessoa visualizará as imagens 3D, recorre-se a esta técnica, uma vez que as imagens serão projetadas em uma tela de metal e cada observador usará os óculos (que possuem um baixo custo) como filtro. A desvantagem desta técnica é que o observador fica restrito em relação à inclinação do seu ponto de visualização, pois se ocorrer uma simples inclinação de sua cabeça, por exemplo, acontecerá perda da noção de profundidade e de parte do brilho da imagem.

A Figura 14 mostra o princípio desta técnica. No lado esquerdo, uma fonte de luz é polarizada, utilizando-se um polarizador. No lado direito dois projetores são utilizados para projetar pares estereoscópicos, utilizando polarização da luz. Com os óculos mostrados na mesma figura, cada imagem é devidamente filtrada.

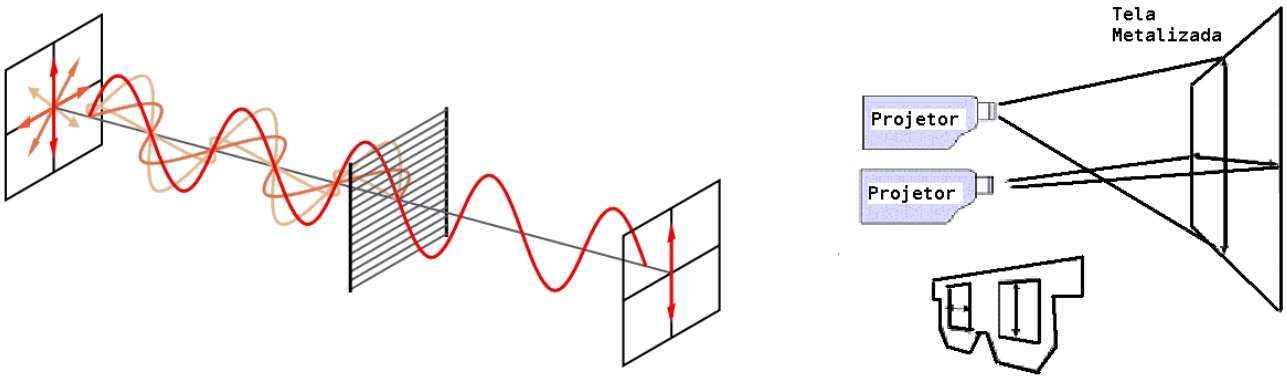

Figura 14: Polarização da luz em eixos ortogonais

2.3.2.4 Óculos obturadores

Óculos obturadores ou *shutter glasses* são óculos especiais, ilustrados na Figura 15, pois suas lentes são de cristal líquido. Tais lentes podem ficar instantaneamente transparentes ou opacas, através da monitoração por controle eletrônico. Esse controle pode ser sincronizado com o sinal de vídeo, de forma a deixar opaca uma das lentes e transparente a outra. Com isso, quando a imagem esquerda é apresentada para o observador, a lente esquerda permanece transparente e a direita opaca, quando a imagem direita é apresentada, ocorre o contrário.

A oscilação entre imagem direita/imagem esquerda juntamente com a oscilação transparente/opaca é suficientemente rápida para os olhos humanos (60Hz para cada olho), o resultado final é que cada olho acaba enxergando uma imagem diferente, induzindo o cérebro a fundir as imagens, provocando a percepção de profundidade. Uma desvantagem destes óculos é que nem sempre eles conseguem vedar completamente a luz, o que acarreta em um olho perceber pontos que deveriam ser vistos pelo outro olho. E também, nem sempre existe tempo suficiente para que o fósforo

do *display* retorne ao seu estado de mais baixa energia.

Figura 15: Óculos obturadores

2.3.3 Análise matemática da estereoscopia

Até então, a estereoscopia foi analisada através de conceitos teóricos e algumas das técnicas estereoscópicas, existentes, foram explicadas. Porém a estereoscopia pode ser analisada matematicamente, explorando alguns conceitos encontrados na visão binocular. Esta seção descreve alguns desses conceitos, bem como sua influência no desenvolvimento de sistemas e aplicações estereoscópicos.

2.3.3.1 Paralaxe

Observando a Figura 16, percebe-se que, caso o ponto de vista do observador esteja em A, o objeto a ser visualizado parece estar exatamente à frente do cubo vermelho. Porém, mudando-se o ponto de vista para B, o objeto parece ter-se deslocado para frente do cubo azul. A diferença entre os pontos onde o objeto aparece dependendo do ponto de vista é chamada de paralaxe.

Paralaxe, do grego *παραλλαγή* (*parallagé*), significando alteração, resulta da mudança da posição angular de dois pontos estacionários, um relativo ao outro, percebida por um observador e causada pela movimentação do mesmo [18].

A paralaxe é uma métrica muito importante na percepção da profundidade [9]. No olho humano, por exemplo, temos a paralaxe monocular e a paralaxe binocular. Para a primeira, quando a cabeça é movida de um lado para o outro, objetos que estão próximo se movem com maior velocidade do que objetos que estão distantes. Então, pode-se notar a percepção de profundidade de acordo com a velocidade de cada objeto. Já a paralaxe binocular diz respeito ao seguinte fenômeno: como os olhos estão separados por uma pequena distância, estando em pontos de vista diferentes, cada olho percebe uma imagem distinta do ambiente. As imagens vistas separadamente não apresentam diferenças perceptíveis, porém, a fusão de ambas no cérebro, cria a percepção de profundidade.

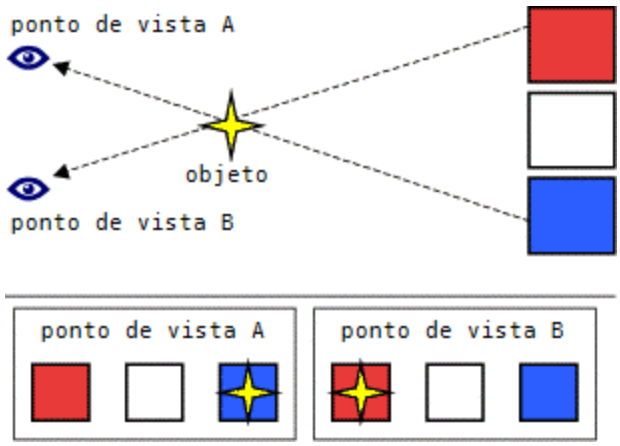

Figura 16: Paralaxe entre pontos de vista diferentes

Tomando-se um plano de projeção (a tela do computador, por exemplo) como foco do observador, a paralaxe pode possuir valores positivos, indicando que o objeto está atrás do plano de projeção, e negativos, indicando que o objeto está à frente do plano de projeção; o valor zero indica que o objeto está situado no plano de projeção, ou seja, no foco do observador. A Figura 17 mostra os três tipos diferentes de paralaxes. O valor da paralaxe é determinado pela distância entre os pontos D e E .

Analisando-se os valores da paralaxe, quando ela é positiva e possui um valor próximo da distância interaxial (distância entre os olhos do observador), a fusão das imagens é ruim, pois o foco está situado em objetos localizados muito longe. Projeções de pontos situados após o foco não possuem uma distância grande, o que causa a perda da noção de profundidade. Quando a paralaxe é maior que a distância interaxial, ocorre a sensação de perda da visão estéreo. A Figura 18 mostra as três relações entre o valor da paralaxe P e o valor da distância interaxial t_c .

As projeções dos pontos percebidos devem cair no retângulo que define o campo de visão do observador, como ilustrado pelos retângulos da Figura 18 que contêm os pontos D e E . Quando isto não ocorre, é porque apenas um dos olhos está vendo somente alguns pontos e a noção do estéreo foi perdida. Isto só é aceitável para pontos que se movem rapidamente.

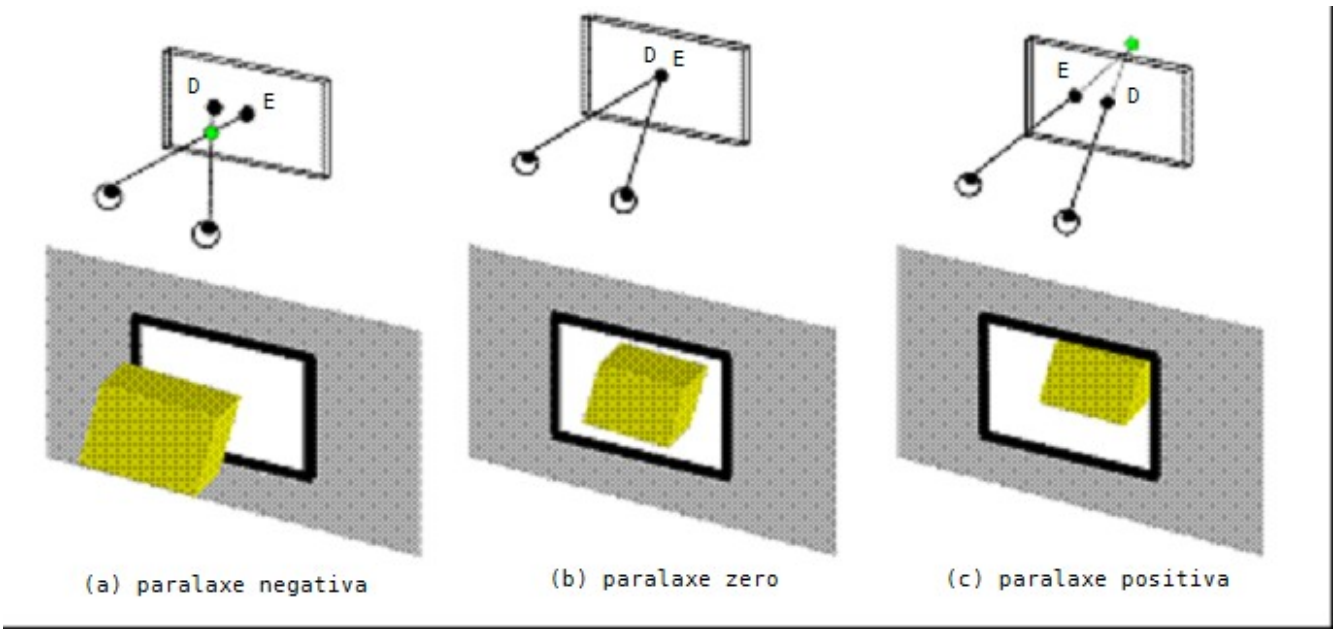

Figura 17: Tipos de paralaxe

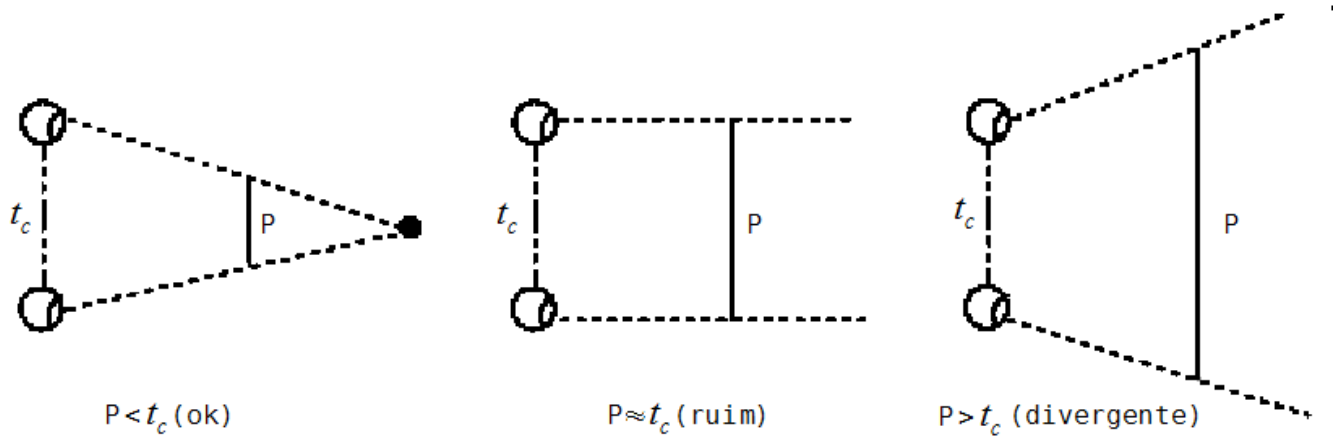

Figura 18: Relação entre a distância interaxial t_c e a paralaxe P

2.3.3.2 Disparidade da retina

Como as imagens percebidas por cada olho são diferentes, uma conseqüência imediata dessa diferença é a disparidade na retina. Essa disparidade é calculada de acordo com o ponto que está sendo focado pelo observador. A Figura 19 mostra como é calculada a disparidade na retina. Caso o observador mantenha o foco no ponto F_1, tem-se que o ponto F_2 está projetado a uma distância diferente para cada olho. A soma dos ângulos formados pelas projeções de F_1 e F_2 em cada olho é a disparidade na retina. A disparidade na retina das imagens projetadas no olho esquerdo e direito induzem o cérebro a perceber uma imagem 3D da cena.

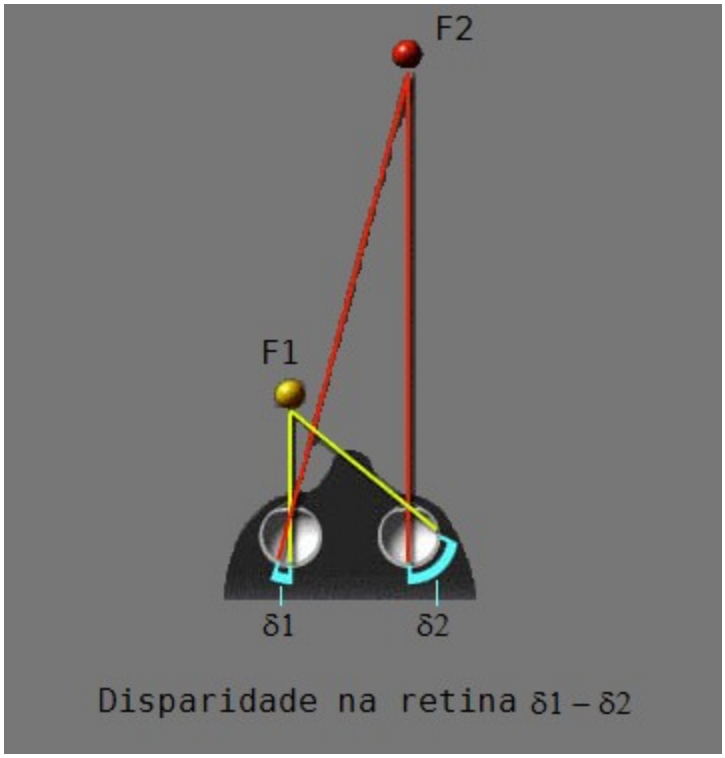

Figura 19: Disparidade na retina

Um fato importante a ser destacado é que existe uma superfície imaginária onde a disparidade na retina é igual a zero. Tal superfície é mostrada na Figura 20 e denomina-se horopter. O ponto A possui disparidade negativa na retina e o ponto C possui disparidade positiva. O ponto B, além de ser o foco do observador, está situado no horopter e, portanto, possui disparidade zero.

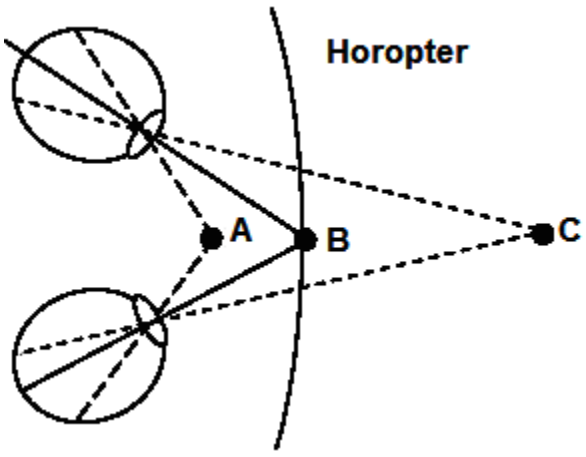

Figura 20: Superfície imaginária onde a disparidade na retina é zero (horopter)

2.3.3.3 Ângulo de vergência

Quando o observador muda seu foco de visualização do ponto A para o ponto B, na Figura 21, uma

rotação nos olhos ocorreu. A vergência pode então ser calculada através da diferença dos ângulos $\alpha - \beta$.

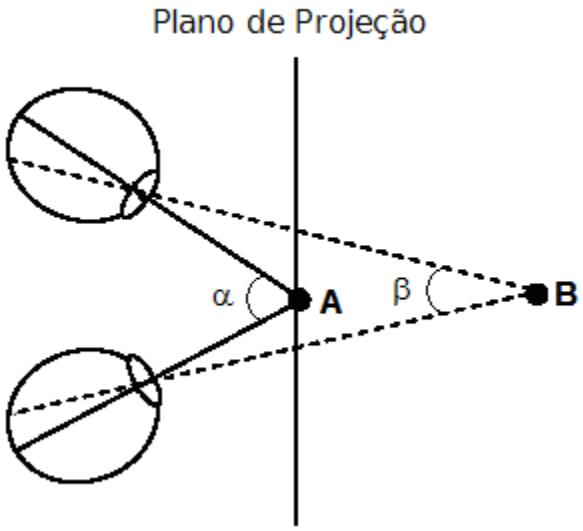

Figura 21: Cálculo do ângulo de vergência

Pode-se, então, concluir pela Figura 21 que, se o ângulo de vergência é positivo o observador afastou o foco do seu plano de visão anterior, se o ângulo de vergência for nulo o observador manteve o foco, e se o ângulo de vergência for negativo o observador aproximou o foco do seu plano de visão anterior.

3 OpenStereo

Neste capítulo serão descritos a ferramenta OpenStereo, sua arquitetura, implementação e os *plugins* que ela fornece, bem como a sua utilização. O objetivo principal é ilustrar como a biblioteca foi desenvolvida, ressaltar as vantagens de sua utilização na conversão de aplicações 3D em estereoscópicas e na criação de novos *plugins*. Outro destaque deste capítulo é a possibilidade de extensão do OpenStereo como a sua integração com diferentes bibliotecas gráficas e diversas formas de geração de conteúdo estereoscópico.

3.1 Descrição do OpenStereo

O OpenStereo tem como objetivo principal fornecer o suporte à conversão de aplicações 3D de tempo real, baseadas em OpenGL, em aplicações estereoscópicas, independente da placa gráfica ou SO usados. Uma conseqüência de tal conversão é o aumento da imersão na aplicação, pois o usuário poderá perceber profundidade entre os objetos 3D.

Para a conversão, o método escolhido foi o estéreo anáglifo, uma vez que os custos envolvidos em equipamentos para a visualização da aplicação, utilizando-se essa técnica, são baixos. Isso viabiliza tanto o seu desenvolvimento quanto a sua execução. Como o OpenStereo é independente de placa gráfica ou SO, as aplicações poderão ser executadas em mais plataformas, permitindo aos desenvolvedores e usuários escolher quais plataformas são mais adequadas às suas necessidades.

Uma cena 3D em OpenGL pode ser facilmente transformada em estereoscópica utilizando o OpenStereo, provendo, então, uma maior imersão do usuário e reduzindo os custos da aplicação, uma vez que *hardwares* especializados não são necessários. O único requisito necessário é que a aplicação e o OpenStereo mantenham um mesmo contexto gráfico. Detalhes serão apresentados na próxima seção, onde a arquitetura do OpenStereo é descrita.

3.2 Arquitetura

Com o intuito de prover uma maneira fácil de integração de uma aplicação 3D existente com o OpenStereo, sua arquitetura foi dividida em duas APIs: uma que auxilia a conversão de aplicações 3D em aplicações estereoscópicas e outra que fornece suporte ao desenvolvimento de *plugins*. A primeira API funciona expondo um conjunto de funções que podem ser chamadas por uma aplicação 3D, permitindo a criação de um *viewport*[5], onde a cena escolhida será renderizada, a manipulação de uma

5 Um *viewport* é a área onde a cena será renderizada, ou seja, é uma região 2D.

câmera, e a escolha de um método anáglifo. A segunda API provê um conjunto de funções que devem ser implementadas pelo *plugin*, validando-o no contexto do OpenStereo. Ao se implementar um *plugin*, o desenvolvedor tem a vantagem de abstrair a biblioteca gráfica subjacente, bem como o formato dos dados de saída estéreo. Cada *plugin* é reponsável por lidar com a alocação e renderização dos dados estereoscópicos. A arquitetura do OpenStereo é mostrada na Figura 22.

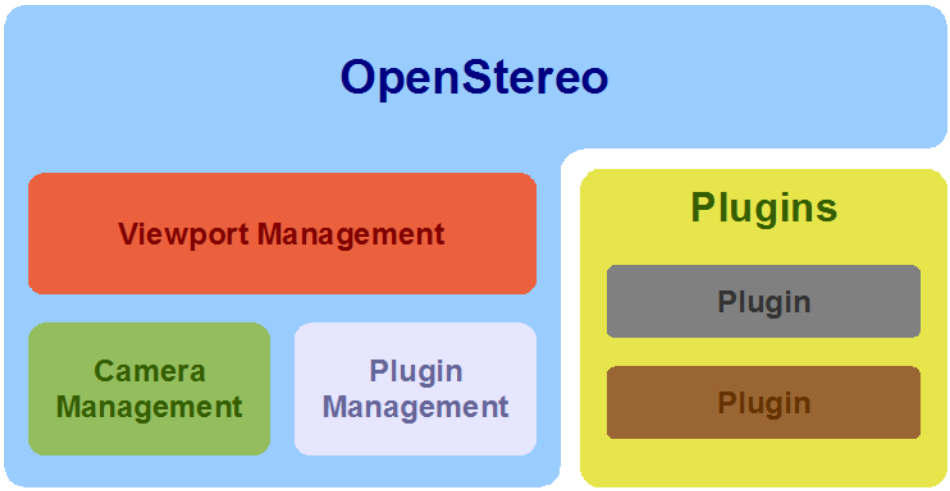

Figura 22: Arquitetura do OpenStereo

A biblioteca é composta de três módulos que gerenciam, respectivamente, câmeras, *viewports* e *plugins*. O módulo de gerenciamento de *viewports* (`Viewport Management Module`), por exemplo, lê os valores da câmera do módulo de gerenciamento de câmeras (`Camera Management Module`) e aloca os dados estereoscópicos de acordo com o seu tamanho, interagindo com o módulo de gerenciamento de *plugins* (`Plugin Management Module`), além de possibilitar a modificação do tamanho e do método anáglifo de um *viewport*.

O OpenStereo carrega os *plugins* na memória e acessa os pontos de entrada de suas funções através do `Plugin Management Module`. Porém, antes disso, o desenvolvedor precisa criar um contexto e em seguida inicializar o OpenStereo. Isto pode ser feito através das chamadas das funções `slCreateContext` e `slInitContext`, nesta seqüência. A primeira função irá criar um contexto e configurar a memória para alocação de *viewports*, *plugins* e câmeras. A segunda função será chamada com um caminho para um arquivo passado como argumento. Esse arquivo contém informações sobre quais são os *plugins* que deverão ser carregados em memória, bem como o *plugin* padrão, aquele que irá gerar a saída estereoscópica.

Com um contexto criado e a biblioteca inicializada, os *plugins* devem ser carregados em memória e terem seus pontos de entrada de funções (símbolos) acessados. Se algum símbolo estiver faltando, o *plugin* é considerado inválido pelo OpenStereo e um erro é retornado. Para que um *plugin*

seja considerado válido, sete símbolos devem ser expostos por ele: 1) `loadPlugin`, 2) `getPluginName`, 3) `getPluginDescription`, 4) `allocateStereoData`, 5) `destroyStereoData`, 6) `renderStereo` e 7) `unloadPlugin`.

O símbolo `loadPlugin` é responsável por carregar o *plugin* propriamente dito, configurando suas variáveis, se alguma estiver presente. Os símbolos `getPluginName` e `getPluginDescription` retornam o nome e a descrição do *plugin*, respectivamente. Eles não são utilizados na geração de dados estereoscópicos, mas são requeridos pela biblioteca. O símbolo `allocateStereoData` tem a função de reservar memória para os dados estereoscópicos. Enquanto que o símbolo `destroyStereoData` desaloca esses dados. Ele tem o propósito de ser usado quando o *viewport* tem seu tamanho alterado, então os dados estereoscópicos antigos devem ser destruídos, antes que novos dados sejam alocados. O símbolo `renderStereo` é o mais importante de todos, porque ele renderiza uma dada cena no *viewport*, usando os dados estereoscópicos que contêm as imagens esquerda e direita. Quando o *plugin* não é mais necessário, ele deve ser removido da memória. O símbolo `unloadPlugin` é responsável por remover as variáveis alocadas pelo símbolo `loadPlugin`.

Depois de acessar os símbolos, o *plugin* padrão tem seu símbolo `loadPlugin` chamado. O OpenStereo está, então, pronto para ser usado na geração de conteúdo estereoscópico de uma aplicação que seja compatível com o *plugin* referido.

Como dito anteriormente, por exemplo, uma aplicação DirectX não deve utilizar o OpenStereo com um *plugin* OpenGL, pois os resultados poderão ser imprevisíveis neste caso. Sendo assim, *plugins* OpenGL devem ser utilizados somente com aplicações OpenGL para manter um mesmo contexto gráfico entre o OpenStereo e a aplicação que o utiliza.

Quando o OpenStereo está pronto para ser utilizado, o desenvolvedor deve criar uma câmera e um *viewport*, e em seguida associar essa câmera ao *viewport*. É de responsabilidade da aplicação utilizada modificar os valores da câmera, bem como a dimensão do *viewport* criado, uma vez que o OpenStereo não tem conhecimento sobre qual biblioteca gráfica ou *toolkit* gráfico está em uso. Por último, a cena que será renderizada estereoscopicamente deve ser associada ao *viewport*, para então o *plugin* tratar essa cena, gerando e apresentando o conteúdo estereoscópico.

3.3 Implementação

Em aplicações 3D de tempo real um problema comum é o desempenho envolvido na renderização de uma cena. Este problema se agrava ainda mais com aplicações estereoscópicas, porque a mesma cena deve ser renderizada duas vezes: uma para o olho esquerdo, outra para o olho direito. Por isto, o uso

de linguagens de programação modernas tais como C# ou Java foi evitado para melhorar o desempenho. Das demais linguagens, restaram C e C++; C++ tem a vantagem do polimorfismo e da herança, mas não da compatibilidade, uma vez que uma aplicação em C não pode utilizar uma biblioteca C++ sem haver a conversão desta aplicação para C++.

Outros problemas a serem considerados foram compatibilidade e portabilidade com aplicações e plataformas existentes. O OpenStereo permite compatibilidade com aplicações existentes, uma vez que ele foi desenvolvido em C. Então, uma aplicação escrita em C ou C++ pode utilizá-lo facilmente.

Ele também provê portabilidade entre as plataformas Linux e Windows. Isto pôde ser conseguido utilizando-se bibliotecas portáveis, como a biblioteca GLib [19] para criar coleções que abriguem variáveis de contexto, bem como métodos portáveis, por exemplo, macros de pré-processador.

A biblioteca GLib foi implementada em C e fornece tipos de dados e funções portáveis para abstração de *threads*, carregamento de módulos, alocação de memória, facilidades de tempo e datas, diversos tipos de dados e respectivos algoritmos, dentre outras opções. Por ser compatível e portável, ela foi utilizada no desenvolvimento do OpenStereo para armazenamento de dados e carregamento de módulos.

Alguns tipos de dados do OpenStereo não estão disponíveis para o desenvolvedor. Dessa forma, modificações no *core* do OpenStereo não afetarão o código da aplicação consumidora. Um contexto, por exemplo, deve ser criado pelo desenvolvedor através da função `slCreateContext`, encontrada no *header* `sl.h`. Esta função foi implementada como mostrada no apêndice A.1, utilizando a estrutura `_SLContext` descrita no apêndice A.2. O desenvolvedor não tem acesso a essa estrutura e, dessa forma a sua implementação pode ser modificada sem afetar a aplicação do desenvolvedor.

A estrutura `_SLContext` armazena os *plugins* carregados e os *viewports* criados, gerenciando a memória do que foi adicionado ou removido. O atributo `initialized` indica se o contexto já foi inicializado, evitando alocações repetidas de memória para as estruturas de dados `GHashTable`s utilizadas.

O mesmo acontece com os *plugins*: a estrutura `_SLPlugin`, descrita no apêndice A.3, representa um *plugin* e o desenvolvedor não possui acesso ao mesmo. Mais uma vez a biblioteca GLib foi utilizada com o propósito de carregar *plugins*. A estrutura `GModule` indica o *plugin* que foi carregado. O nome e a descrição do *plugin* são mantidos e os demais atributos são os símbolos expostos pelo *plugin* para serem acessados na geração de conteúdo estereoscópico.

O apêndice A.4 mostra uma função interna utilizada para carregar um *plugin* através de um arquivo DLL, no Windows, ou um arquivo SO, no Linux através da biblioteca GLib. Primeiramente, o módulo é aberto através da função `g_module_open`, em seguida, se o módulo foi aberto com sucesso seus símbolos serão carregados; caso contrário, o erro `SL_CANNOT_OPEN_PLUGIN` é guardado na variável global `error`. Se algum símbolo não estiver presente no módulo, o *plugin* é considerado inválido e o erro `SL_INVALID_PLUGIN` é guardado na mesma variável global `error`. Essa variável global pode ser acessada através da função `slGetError`.

Plugins, por exemplo, no Windows, possuirão no nome do arquivo a extensão dll, enquanto que no Linux, a extensão so. Para solucionar a utilização de diferentes extensões no nome do arquivo são utilizadas macros, mantendo o código portável. O apêndice A.5 descreve a solução para esse problema.

O OpenStereo, então, provê uma implementação portável e evita que a aplicação que o utiliza seja dependente da implementação do OpenStereo. O mesmo acontece com a API fornecida para a implementação de *plugins*.

3.4 Plugins

O OpenStereo provê uma API para criação de *plugins* através do *header* `sl_plugin.h`, listando as funções que devem ser implementadas. A implementação do *plugin* pode ser feita de diversas maneiras, dependendo do contexto gráfico, da natureza dos dados estereoscópicos, entre outros. O OpenStereo fornece dois *plugins* que provêm suporte à conversão de aplicações 3D de tempo real (na atual versão, baseadas em OpenGL) em aplicações estereoscópicas.

Durante a execução de uma aplicação que utiliza o OpenStereo com algum *plugin*, os passos necessários para que o conteúdo estereoscópico seja devidamente gerado são: criar um contexto, carregar o *plugin*, criar um *viewport*, criar uma câmera, associar a câmera ao *viewport*, mudar o tamanho do *viewport* quando necessário, associar o *viewport* ao contexto, posicionar a câmera e renderizar a cena. Dos passos citados, o *plugin* está envolvido na criação do *viewport*, uma vez que é necessário alocar memória para conter os dados estereoscópicos, e na renderização da cena, pois o *plugin* é responsável por gerar e fundir as imagens do olho esquerdo e direito, de acordo com a cena fornecida, bem como apresentar o resultado final dessa fusão. A Figura 23 mostra detalhadamente o fluxo necessário para que a cena seja renderizada estereoscopicamente.

Como toda aplicação estereoscópica, a cena precisa ser renderizada duas vezes, uma para o olho esquerdo e outra para o olho direito. Os dados estereoscópicos são guardados na estrutura

`_SLStereoData`, que contém três ponteiros para dados da imagem direita, esquerda e estereoscópica, respectivamente. É de responsabilidade do *plugin* saber como armazenar e ler tais dados. A estrutura simplesmente foi definida por conveniência, com o intuito de facilitar o desenvolvimento do *plugin* e da função de renderização. Maiores detalhes estão descritos no apêndice A.6.

O posicionamento da câmera nos *plugins* seguem o método *toe-in*, mostrado na Figura 24. Apesar desse método apresentar um resultado satisfatório do ponto de vista visual, ele é incorreto, pois introduz uma paralaxe vertical na imagem estéro. O método correto é o *off-axis*, mostrado na Figura 25. O método *toe-in* foi utilizado por ser mais fácil a sua implementação.

Abaixo serão demonstrados como os *plugins* foram implementados, ressaltando-se a compatibilidade com placas gráficas comuns e ilustrando o desempenho de cada *plugin*.

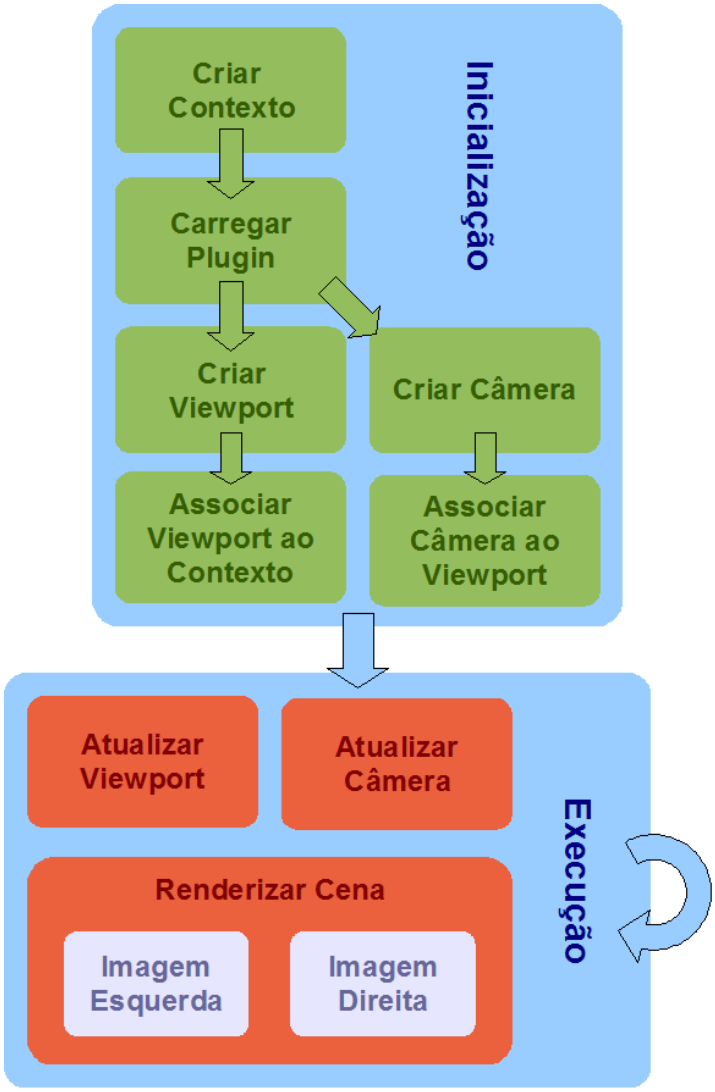

Figura 23: Fluxo de execução do OpenStereo

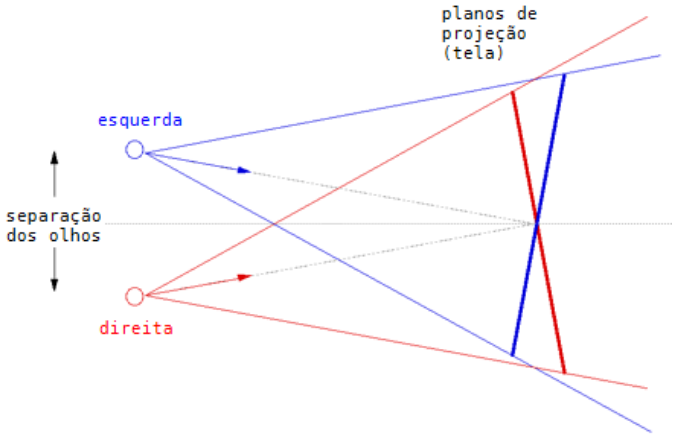

Figura 24: Método toe-in

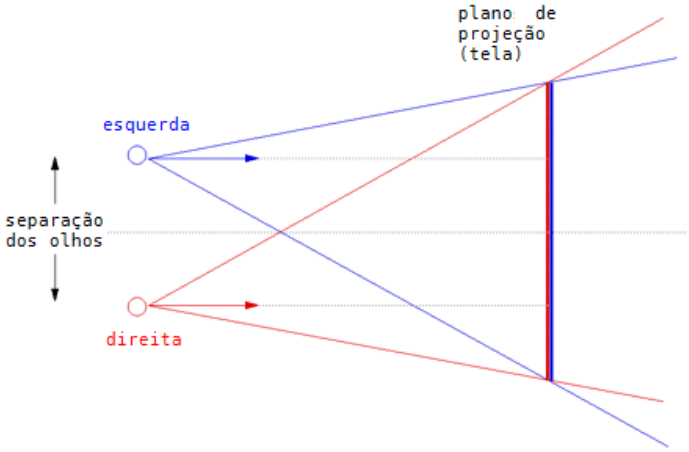

Figura 25: Método off-axis

3.4.1 Plugin libSLGL_Raw

Seguindo o *header* `sl_plugin.h`, o *plugin* libSLGL_Raw foi implementado para ser executado em qualquer placa gráfica que suporte os comandos `glReadPixels` e `glDrawPixels` de OpenGL. Todo o processamento é feito pela CPU, portanto o desempenho da aplicação depende diretamente do processador, não sendo necessária uma placa gráfica especializada.

A natureza dos dados estereoscópicos da estrutura `_SLStereoData` são blocos de memória que representam uma imagem do tamanho do *viewport*, onde somente seus *pixels* no formato de cor RGBA[6] (*Red, Green, Blue, Alpha*) são armazenados.

O apêndice A.7 mostra o trecho de código responsável pela renderização estereoscópica da cena, com a geração e fusão dos pares de imagens. O primeiro passo é acessar alguns dados do *viewport*, principalmente o método anáglifo, que influi diretamente na aparência do resultado final da

6 O formato RGBA é uma extensão do RGB, através da adição da componente *Alpha*.

renderização. Após isso são geradas as imagens esquerda e direita através da funções `readFrame`, fundidas pela função `createStereo` e por fim, o resultado da fusão é renderizado através do comando `glDrawPixels` de OpenGL. Tal comando possui baixo desempenho, pois há uma cópia dos dados da memória principal para a memória de vídeo.

A função `readFrame`, mostrada no apêndice A.8, posiciona a câmera devidamente para o olho esquerdo, caso a variável `left` seja diferente de zero, ou para o olho direito, caso contrário. Em seguida, renderiza a cena (ao chamar a função `scene`), copiando o conteúdo renderizado para a variável `frame`, que pode ser a imagem da esquerda ou da direita. Esse processo de cópia é lento, uma vez que a memória de vídeo está na placa gráfica e para que o processador possa acessá-la, ela deve ser copiada antes para a memória principal.

A função `createStereo`, como o nome sugere, cria a imagem estereoscópica, através da fusão das imagens esquerda e direita. Ela foi implementada conforme descrito no apêndice A.9, obtendo a matriz relacionada com método anáglifo utilizado e multiplicando essa matriz pelo vetor de cores de cada *pixel* das imagens previamente renderizadas.

Apesar de suportar uma grande variedade de placas gráficas, o desempenho do *plugin* libSLGL_Raw é muito baixo, pois como todo o processamento é feito pela CPU, nenhuma instrução específica da placa gráfica é utilizada e o processador acaba sendo compartilhado com outras atividades, por exemplo, o SO, subutilizando a placa gráfica. Além do mais, há uma excessiva cópia de dados entre a placa gráfica e a memória principal do computador, penalizando ainda mais o desempenho.

A vantagem da portabilidade entre diversas placas gráficas é penalizada com a perda de desempenho, porém alternativas mais eficientes poderão ser desenvolvidas, considerando as placas gráficas mais recentes. O *plugin* libSLGL, descrito na próxima seção, utiliza recursos avançados das placas gráficas comuns, melhorando o desempenho da aplicação.

3.4.2 Plugin libSLGL

Para evitar a má utilização dos recursos de processamento e aumentar o desempenho da renderização de cenas estereoscópicas, o *plugin* libSLGL foi desenvolvido utilizando recursos avançados das placas gráficas. Um impacto relevante ocorrido é a liberação da CPU para processamento de atividades não relacionadas à computação gráfica.

O *plugin* libSLGL realiza todos os cálculos de fusão das imagens na GPU (*Graphics Processing*

Unit), que é o processador da placa gráfica. Com isso a CPU é liberada para realizar outras tarefas, resultando num ganho de desempenho. O acesso à GPU é dado através da utilização da linguagem Cg (*C for Graphics*) [20]. Para evitar a cópia de memória ocorrida com o *plugin* libSLGL_Raw, é utilizada a técnica de *Rende-To-Texture* (RTT), onde todo o conteúdo da cena é renderizado para uma textura.

O primeiro passo é carregar o plugin. O apêndice A.10 mostra a função `loadPlugin`, responsável por inicializar as variáveis globais desse *plugin*. Algumas funções de extensão de OpenGL serão utilizadas. Para facilitar o acesso a tais funções, a biblioteca Glew [21] foi utilizada. A função `glewInit` inicializa a biblioteca, que pesquisa quais são as extensões suportadas pelo *driver* de OpenGL. Em seguida, a função de extensão `glGenFramebuffersEXT` é utilizada para criar um *Framebuffer* auxiliar, que conterá o conteúdo de cor e profundidade da cena. Um *Framebuffer*, em OpenGL, é um conjunto de *buffers* que contém informações sobre cor, profundidade, sombras, entre outras, de uma cena. Para utilizar a biblioteca Cg é necessário que um contexto seja criado, para em seguida o *pixel shader*[7], mostrado no apêndice A.11, possa ser compilado e criado um programa através da função `cgCreateProgram`, que será carregado na GPU, através da função `cgLoadProgram`. Por fim, os parâmetros necessários ao programa são acessados, para que futuramente possam ser modificados e lidos.

O apêndice A.11 descreve um *pixel shader*, uma vez que a sua função principal retorna conteúdo com o tipo de dados `COLOR`. Ela recebe como parâmetros: a coordenada da textura, para que seja possível acessar o *pixel* que está sendo processado quando o programa for chamado; `matrixL`, a matriz de transformação para o *pixel* da imagem esquerda; `matrixR`, a matriz de transformação para o pixel da imagem direita; `textureL` e `textureR`, que são as imagens esquerda e direita, respectivamente.

Com esses dados, a função simplesmente multiplica os *pixels* das imagens da esquerda e da direita pelas suas respectivas matrizes de transformação, soma as matrizes resultantes e retorna o valor. O valor será colocado no *Framebuffer* principal da placa gráfica. O resultado é a apresentação da imagem estéreo.

O apêndice A.12 mostra o processo de renderização do *plugin* libSLGL. O *plugin* libSLGL inicialmente renderiza a cena em duas texturas diferentes (uma para a imagem da direita e outra para a imagem da esquerda), que serão posteriormente submetidas ao *pixel shader* conforme descrito no apêndice A.11. Para utilizar a técnica de RTT, extensões de OpenGL precisam ser utilizadas tanto para armazenar as texturas como *Renderbuffers,* quanto para renderizar o conteúdo delas. Um *Renderbuffer* é uma área de memória que pode ser utilizada para renderização de cenas *off-screen*, ou

7 Programa escrito na linguagem Cg para acesso ao *buffer* de cor do *Framebuffer* da placa de vídeo.

seja, cenas que podem ser renderizadas, sem necessariamente serem apresentadas. Cada imagem possui dois *Renderbuffers*, um para o *buffer* de cor e outro para o *buffer* de profundidade.

Após criar as texturas e renderizar o conteúdo da cena através do RTT, é necessário que elas sejam fundidas, criando a imagem estéreo. Esse passo é realizado na função `writeTexture`, mostrada no apêndice A.13, que simplesmente renderiza uma das texturas na tela. Para cada *pixel* da textura, o *pixel shader* mostrado no apêndice A.11 será chamado e a imagem estéreo será gerada.

A principal vantagem em se utilizar o *plugin* libSLGL é o ganho no desempenho. Porém, há uma desvantagem em se utilizar extensões de OpenGL, pois elas são restritas para algumas placas gráficas, limitando assim o ambiente de execução das aplicações.

4 Implantação e Desempenho

Neste capítulo serão apresentados alguns resultados preliminares da utilização do OpenStereo em uma aplicação OpenGL já existente, bem como o desempenho do uso dos *plugins* nesta aplicação.

4.1 Aplicação

O OpenStereo possui a vantagem de poder ser integrado com aplicações existentes através de pequenas modificações na aplicação. O primeiro passo é verificar a aplicação que será implantada para poder adaptar seu código ao fluxo de execução do OpenStereo, descrito na Figura 23, e em seguida executá-la com os *plugins*.

Para demonstrar o funcionamento do OpenStereo, foi convertida uma aplicação dos tutoriais de OpenGL do *site* NeHe [22]. A Lição 37 (aplicação demo), que ensina a técnica de *Cel-Shading* [23], utilizada para simular o efeito encontrado em *cartoons* [24], foi escolhida e convertida para utilizar o OpenStereo. Após essa utilização, foi constatado algumas falhas na implementação atual do OpenStereo, que deverão ser melhoradas, como, por exemplo, o correto posicionamento da câmera.

4.2 Implantação do OpenStereo

Seguindo o fluxo de execução do OpenStereo, ilustrado na Figura 23, o primeiro passo é criar um contexto e em seguida carregar algum *plugin*, através da inicialização do contexto. A aplicação fornece uma função de inicialização, onde a biblioteca OpenGL é devidamente incializada. Como os *plugin*s a serem utilizados dependem da biblioteca OpenGL, o código de inicialização do contexto do OpenStereo, descrito no apêndice A.14, é colocado após a inicialização do OpenGL. Uma câmera é criada e utilizada para posicionamento do observador na cena. Então, um *viewport* é criado com a mesma dimensão que a janela da aplicação e a câmera é associada a esse *viewport*. O método anáglifo e a cena que deve ser renderizada também são associados ao *viewport*.

Outra consideração importante é relativa à atualização das dimensões do *viewport*. Cada vez que a janela da aplicação for modificada a dimensão do *viewport* deve ser atualizada para que seja mantida uma consistência visual.

Por fim, a cada *frame*, a função `slRenderScene` é chamada, renderizando a cena em estéreo. A cena que será renderizada deve simplesmente desenhar vértices, texturas, entre outras primitivas, sem apagar nenhum dos *buffers* de OpenGL, restrição essa imposta pelos p*lugins*, uma vez que eles

são responsáveis por essas ações.

A Figura 26 representa o resultado da conversão da aplicação demo. No lado lado esquerdo pode-se observar o resultado antes da conversão, enquanto que no lado direito, o resultado após a mesma.

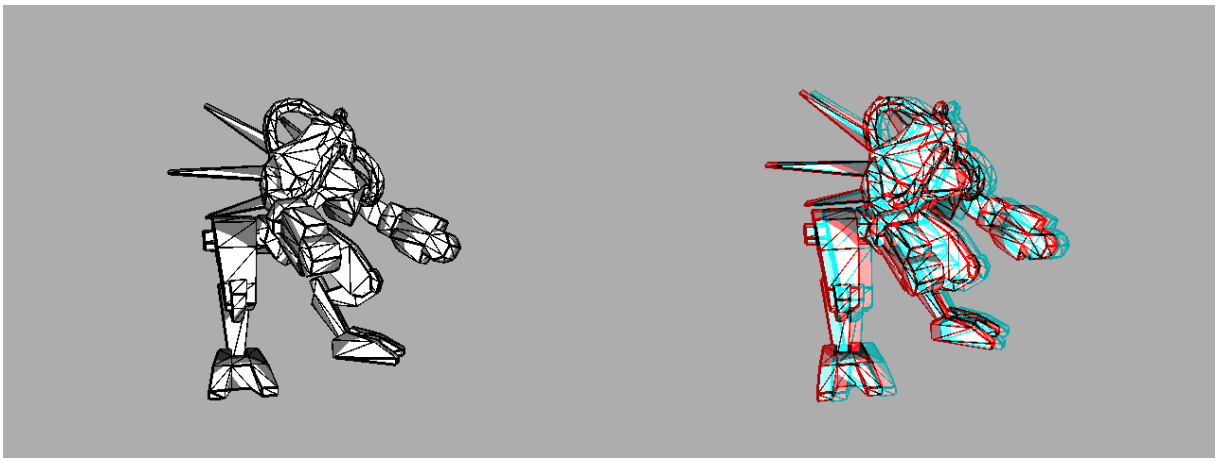

Figura 26: Aplicação demo antes e depois da conversão

4.3 Testes preliminares

Aplicações estereoscópicas necessitam do máximo de desempenho possível, pois uma mesma cena será renderizada duas vezes.

Um teste inicial de desempenho foi realizado na aplicação demo, apontando a taxa de FPS (*Frames Per Second*) antes e depois da conversão. O mesmo teste foi realizado em duas máquinas diferentes, sob o SO Microsoft Windows. A primeira (Máquina 1) possui processador Pentium 4 3.06GHz, 1GB RAM e placa de gráfica *off-board* NVIDIA FX 5200 256MB. A segunda (Máquina 2) possui um processador Athlon 64 3000+ 1.8GHz, com 512MB RAM e placa gráfica *on-board* ATI RADEON Xpress 200.

O teste consiste na obtenção da taxa de FPS. A aplicação foi executada em modo *fullscreen* numa resolução de 640 *pixels* de largura por 480 *pixels* de altura.

A Tabela 1 mostra os resultados obtidos da execução do teste.

Máquina	Taxa de FPS antes da conversão	Taxa de FPS após a conversão
1	589	66
2	570	63

Tabela 1: Resultados do teste realizado na aplicação demo

Pelos resultados acima, pode-se concluir que o OpenStereo pode ser utilizado em aplicações de pequeno porte sem comprometer muito o desempenho da aplicação. Cabe ressaltar que a perda no desempenho é relacionada com o tempo de fusão das imagens na placa gráfica. Esse tempo de fusão depende somente do tamanho das texturas. Porém, uma análise de desempenho mais criteriosa deverá ser realizada utilizando um método estatístico com intervalo de confiança para validar formalmente o OpenStereo.

5 CONCLUSÃO

Este trabalho de graduação apresentou a biblioteca OpenStereo, cujo objetivo principal é fornecer suporte à conversão de aplicações 3D de tempo real (baseadas em OpenGL, na versão atual) em aplicações estereoscópicas. Informações sobre estereoscopia e percepção foram fornecidas, além da explicação de alguns conceitos matemáticos envolvidos na área. Foi realizada uma pesquisa bibliográfica, criando-se assim um breve levantamento sobre o estado-da-arte na área de estereoscopia.

A arquitetura e implementação do OpenStereo foram detalhadamente explicadas, facilitando a reprodução desse trabalho como também a sua extensibilidade. Foi demonstrada sua implantação numa aplicação já existente, bem como a geração de conteúdo estereoscópico, possibilitando assim uma maior imersão. Após a implantação, um teste inicial de desempenho foi realizado, constatando-se o não comprometimento do desempenho da aplicação.

Através de uma arquitetura e implementação simples, o OpenStereo traz várias vantagens, entre elas a portabilidade, a compatibilidade, a extensibilidade e a viabilidade.

Portabilidade compreende a sua utilização em diversas plataformas de SOs, como o Microsoft Windows e o Linux, com diversas placas gráficas. Compatibilidade com diversos ambientes de execução, uma vez que ele foi desenvolvido utilizando-se a linguagem de programação C. Extensibilidade ao prover um sistema de *plugins*, onde o desenvolvedor pode implementar seus próprios *plugins*, de acordo com suas necessidades, sem precisar alterar o código da aplicação existente. E, por fim, viabilidade, uma vez que não é necessário um grande investimento em dispositivos para visualização, tais como placas gráficas profissionais, óculos específicos, entre outros.

A arquitetura do OpenStereo está bem consolidada, porém a sua definição não foi trivial, uma vez que o OpenStereo permita a aplicação abstrair detalhes da implementação. Uma outra dificuldade nos requisitos do desenvolvimento é a portabilidade para diferentes plataformas.

O OpenStereo em uma nova versão deverá suportar mais funcionalidades, como também mais bibliotecas gráficas. Na versão atual, a troca de *plugins* em tempo de execução não é possível. Além disso, somente foram implementados *plugins* para aplicações OpenGL. A implementação de *plugins* para a biblioteca gráfica DirectX é necessária, uma vez que existe uma vasta gama de aplicações que utiliza essa biblioteca.

Por fim, nos *plugins* fornecidos, o método atual para o cálculo do par de imagens (método *toe-in*), apesar de apresentar um resultado satisfatório, não é o mais adequado, por introduzir uma

paralaxe vertical na imagem estéreo. A correção desse método deverá ser realizada em uma nova versão do OpenStereo, através da utilização do método *off-axis*.

6 Referências Bibliográficas

[1] NOVELL. *XGL, X over OpenGL*. URL: http://en.opensuse.org/Xgl Consultado em 5 out. 2006.

[2] CHRIS LAUREL. *Celestia, a real-time 3D space simulation*. URL: http://www.shatters.net/celestia Consultado em 5 out. 2006.

[3] SGI. *OpenGL, Open Graphics Library*. URL: http://www.opengl.org Consultado em 5 out. 2006.

[4] MICROSOFT. *DirectX, an advanced suite of multimedia application programming interface*. URL: http://www.microsoft.com/directx Consultado em 5 out. 2006.

[5] NVIDIA CORPORATION. *NVIDIA*. URL: http://www.nvidia.com Consultado em 5 out. 2006.

[6] ATI TECHNOLOGIES INC. *ATI*. URL: http://www.ati.com Consultado em 5 out. 2006.

[7] AIRES, M. de M. *Fisiologia*. 2.ed. Rio de Janeiro: Guanabara Koogan, 1999. 934p.

[8] PLOMP, R. *The intelligent ear*. New Jersey: Lawrence Erlbaum Associates, 2001. 174p.

[9] MURRAY, J. Some perspectives on visual depth perception. *Special Issue on Interactive Entertainment Design, Implementation and Adrenaline*, New York, v.98 n.2, p.155-157, mai. 1994.

[10] HOWARD, I. P.; HOWARD, A. P. *Binocular Vision and Stereopsis*. Oxford: Oxford University Express, 1995. 756p.

[11] BARROS, P. G. de; PESSOA, D. A.; LEITE, P. J. S.; FARIAS, R. C.; TEICHRIEB, V.; KELNER, J. Three-dimensional Oil Well Planning in Ultra-deep Water. *Symposium on Virtual Reality*, Porto Alegre, p.285-196, abr. 2006.

[12] LOKKI, T.; ILMONEN, T.; MAKELA, W.; TAKALA, T. Upponurkka: An Inexpensive Immersive Display for Public VR Installations. *Proceedings of the IEEE Virtual Reality Conference (VR 2006)*, Washington, v.00, p.315-315, mar. 2006.

[13] GERNSHEIN, H.; GERNSHEIN, A. *The History of Photography from the Camera Obscura to the Beginning of the Modern Era*. New York: McGraw-Hill, 1969. 599p.

[14] GONZALEZ, R. C.; WOODS, R. E. *Processamento de Imagens Digitais*. São Paulo: Editora Edgard Blucher Ltda, 2000. 528p.

[15] WOODS, A. J.; ROURKE, T. Ghosting in Anaglyphic Stereoscopic Images. *Stereoscopic Displays*

and Virtual Reality Systems XI. Proceedings of the SPIE, v.5291, p.354-365, mai 2004.

[16] WALLISCH, B.; MEYER, W.; KANITSAR, A.; GRÖLLER, E. *Information Highlighting by Color Dependent Depth Perception with Chromo-Stereoscopy*. Vienna: Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2002. não paginado. Projeto Concluído.

[17] CHROMATEK INC. URL: http://www.chromatek.com/ Consultado em 5 out. 2006.

[18] ANSWERS CORPORATION. *Answers.com – Online Dictionary, Encyclopedia and much more.* URL: http://www.answers.com/ Consultado em 5 out. 2006.

[19] GNU PROJECT. *GLib 2.10.3*. URL: http://www.gtk.org/ Consultado em 5 out. 2006.

[20] FERNANDO, R. C.; KILGARD, M. J. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Boston: Addison-Wesley, 2003. 384p.

[21] IKITS, M.; MAGALLON, M. *Glew, The OpenGL Extension Wrangler Library*. URL: http://glew.sourceforge.net/ Consultado em 5 out. 2006.

[22] NEHE PRODUCTIONS. *Nehe tutorials*. URL: http://nehe.gamedev.net/ Consultado em 5 out. 2006.

[23] DALMAU, D. S. C. *Core Techniques and Algorithms in Game Programming*. Berkeley: New Riders, 2003. 888p.

[24] NOBLE, P.; TANG, W. Modelling and Animating Cartoon Hair with NURBS Surfaces. *Proceedings of the Computer Graphics International (CGI'04)*, Washington, v.00, p.60-67, jun. 2004.

7 Apêndice A

A.1 função slCreateContext

```
void SL_API_ENTRY slCreateContext(void) {
	if (context) {
		error = SL_CONTEXT_ALREADY_CREATED;
		return;
	}
	context = (struct _SLContext*) malloc(
		sizeof(struct _SLContext));
	if (context) {
		context->plugins = g_hash_table_new(0, 0);
		context->viewports = g_hash_table_new(0, 0);
		context->initialized = SL_FALSE;
	} else {
		error = SL_CANNOT_CREATE_CONTEXT;
	}
}
```

A.2 Estrutura _SLContext

```
struct _SLContext {
	GHashTable* plugins;
	GHashTable* viewports;
	SLbool initialized;
};
```

A.3 Estrutura _SLPlugin

```
struct _SLPlugin {
	GModule* module;
	const char* name;
	const char* description;
	PFNSLLOADPLUGIN loadPlugin;
	PFNSLGETPLUGINNAME getPluginName;
	PFNSLGETPLUGINDESCRIPTION getPluginDescription;
	PFNSLALLOCATESTEREODATA allocateStereoData;
	PFNSLDESTROYSTEREODATA destroyStereoData;
	PFNSLRENDERSTEREO renderStereo;
	PFNSLUNLOADPLUGIN unloadPlugin;
};
```

A.4 Função slCreatePlugin

```
SLPlugin slCreatePlugin(const char* filePath) {
	GModule* pluginModule = 0;
	SLPlugin plugin = 0;
```

```
	pluginModule = g_module_open((const gchar*) filePath,
		(GModuleFlags) 0);

	if (pluginModule) {
		plugin = (SLPlugin) calloc(1, sizeof(struct _SLPlugin));
		plugin->module = pluginModule;
		#define LOAD_FUNCTION(function) \
			if (g_module_symbol(pluginModule, #function,\
			(gpointer*) &(plugin->function)) == FALSE) {\
				error = SL_INVALID_PLUGIN;\
				slDestroyPlugin(plugin);\
				return 0;\
			}
			LOAD_FUNCTION(loadPlugin)
			LOAD_FUNCTION(getPluginDescription)
			LOAD_FUNCTION(getPluginName)
			LOAD_FUNCTION(allocateStereoData)
			LOAD_FUNCTION(destroyStereoData)
			LOAD_FUNCTION(renderStereo)
			LOAD_FUNCTION(unloadPlugin)
			plugin->name = plugin->getPluginName();
			plugin->description = plugin->getPluginDescription();
		#undef LOAD_FUNCTION
	} else {
		error = SL_CANNOT_OPEN_PLUGIN;
	}
	return plugin;
}
```

A.5 Código portável

```
#ifdef WIN32
	strcpy(&(name[len]), ".dll");
#else
	strcpy(&(name[len]), ".so");
#endif
```

A.6 Estrutura _SLStereoData

```
struct _SLStereoData {
	SLData left;
	SLData right;
	SLData stereo;
};
```

A.7 libSLGL_Raw: Função renderStereo

```
void SL_PLUGIN_API_ENTRY renderStereo(SLViewport viewport,
SLScene scene, SLStereoData stereoData) {
```

```
        SLCamera camera = slGetCamera(viewport);
        SLuint width = slGetWidth(viewport);
        SLuint height = slGetHeight(viewport);
        SLAnaglyphMethod method = slGetAnaglyphMethod(viewport);
        SLData leftD = stereoData->left;
        SLData rightD = stereoData->right;

        readFrame((GLubyte*) leftD, width, height, scene, camera,
              camera->eyeSeparation, 1);
        readFrame((GLubyte*) rightD, width, height, scene, camera,
              camera->eyeSeparation, 0);

        createStereo(method, leftD, rightD, stereoData->stereo, width,
              height);

        glDrawBuffer(GL_BACK);
        glDrawPixels(width, height, GL_RGBA,
              GL_UNSIGNED_BYTE, stereoData->stereo);
}
```

A.8 libSLGL_Raw: Função readFrame

```
void readFrame(GLubyte* frame, SLuint width, SLuint height,
SLScene scene, SLCamera camera, float eyeDist, int left) {

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();

        gluLookAt(camera->viewPosition.x +
              (left ? - eyeDist : eyeDist) / 2,
              camera->viewPosition.y, camera->viewPosition.z,
              camera->viewDirection.x, camera->viewDirection.y,
              camera->viewDirection.z, camera->viewUp.x,
              camera->viewUp.y, camera->viewUp.z);

        glDrawBuffer(GL_BACK); {
              scene();
        } glReadBuffer(GL_BACK);

        glReadPixels(0, 0, width, height, GL_RGBA,
              GL_UNSIGNED_BYTE, frame);
}
```

A.9 libSLGL_Raw: createSetereo

```
void createStereo(SLAnaglyphMethod method, GLubyte* left,
GLubyte* right, GLubyte* stereo, SLuint width, SLuint height) {
        int x, y, index;

        float* matrixL = getMatrixFromMethod(method, 1);
        float* matrixR = getMatrixFromMethod(method, 0);
```

```
      for (y = 0; y < height; y++) {
            for (x = 0; x < width; x++) {
                  index = 4 * (x + y * width);
                  stereo[index + 2] = (GLubyte)(matrixL[6] * left[index]
                  + matrixL[7] * left[index + 1] +
                  matrixL[8] * left[index + 2] +
                  matrixR[6] * right[index] +
                  matrixR[7] * right[index + 1] +
                  matrixR[8] * right[index + 2]);

                  stereo[index + 1] = (GLubyte)(matrixL[3] * left[index]
                  + matrixL[4] * left[index + 1] +
                  matrixL[5] * left[index + 2] +
                  matrixR[3] * right[index] +
                  matrixR[4] * right[index + 1] +
                  matrixR[5] * right[index + 2]);

                  stereo[index] = (GLubyte)(matrixL[0] * left[index] +
                  matrixL[1] * left[index + 1] +
                  matrixL[2] * left[index + 2] +
                  matrixR[0] * right[index] +
                  matrixR[1] * right[index + 1] +
                  matrixR[2] * right[index + 2]);

                  stereo[index + 3] = 255;
            }
      }
}
```

A.10 libSLGL: Função loadPlugin

```
void SL_PLUGIN_API_ENTRY loadPlugin(void) {
      glewInit();
      glGenFramebuffersEXT(1, &frameBufferID);

      context = cgCreateContext();

      fragmentProgram = cgCreateProgram(context, CG_SOURCE,
            programSource, CG_PROFILE_ARBFP1, 0, 0);
      cgGLLoadProgram(fragmentProgram);

      matrixL = cgGetNamedParameter(fragmentProgram, "matrixL");
      matrixR = cgGetNamedParameter(fragmentProgram, "matrixR");
      textureLP = cgGetNamedParameter(fragmentProgram, "textureL");
      textureRP = cgGetNamedParameter(fragmentProgram, "textureR");
}
```

A.11 libSLGL: *Pixel shader* utilizado para criar a imagem estéreo

```
half3 main (half2 texCoord : TEXCOORD0, uniform half3x3 matrixL,
uniform half3x3 matrixR, uniform sampler2D textureL,
uniform sampler2D textureR) : COLOR {
```

```
      return mul(matrixL, tex2D(textureL, texCoord))
            + mul(matrixR, tex2D(textureR, texCoord));
}
```

A.12 libSLGL: Função renderStereo

```
void SL_PLUGIN_API_ENTRY renderStereo(SLViewport viewport,
SLScene scene, SLStereoData stereoData) {

      SLCamera camera = slGetCamera(viewport);
      SLuint width = slGetWidth(viewport);
      SLuint height = slGetHeight(viewport);
      SLAnaglyphMethod method = slGetAnaglyphMethod(viewport);
      Data* leftD = (Data*) stereoData->left;
      Data* rightD = (Data*) stereoData->right;

      glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, frameBufferID);

      glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
            GL_DEPTH_ATTACHMENT_EXT, GL_RENDERBUFFER_EXT,
            *((GLuint*) stereoData->stereo));

      glDrawBuffer(leftD->colorAttachment);

      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glLoadIdentity();

      gluLookAt(camera->viewPosition.x - (camera->eyeSeparation / 2),
            camera->viewPosition.y, camera->viewPosition.z,
            camera->viewDirection.x, camera->viewDirection.y,
            camera->viewDirection.z, camera->viewUp.x,
            camera->viewUp.y, camera->viewUp.z);

      scene();

      glDrawBuffer(rightD->colorAttachment);

      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glLoadIdentity();

      gluLookAt(camera->viewPosition.x + (camera->eyeSeparation / 2),
            camera->viewPosition.y, camera->viewPosition.z,
            camera->viewDirection.x, camera->viewDirection.y,
            camera->viewDirection.z, camera->viewUp.x,
            camera->viewUp.y, camera->viewUp.z);

      scene();

      glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

      cgGLSetMatrixParameterfr(matrixL,
            getMatrixFromMethod(method, 1));
```

```
	cgGLSetMatrixParameterfr(matrixR,
		getMatrixFromMethod(method, 0));
	cgGLSetTextureParameter(textureLP, leftD->texID);
	cgGLEnableTextureParameter(textureLP);
	cgGLSetTextureParameter(textureRP, rightD->texID);
	cgGLEnableTextureParameter(textureRP);

	cgGLEnableProfile(CG_PROFILE_ARBFP1);
	cgGLBindProgram(fragmentProgram);

	writeTexture(width, height);

	cgGLDisableProfile(CG_PROFILE_ARBFP1);
}
```

A.13 libSLGL: Função writeTexture

```
void writeTexture(GLuint width, GLuint height) {
	SLuint max = width >= height ? width : height;
	SLuint nextPowerOfTwo = 1;
	float maxH = 0;
	float maxW = 0;

	while ((nextPowerOfTwo <<= 1) < max);

	maxH = height / (float) nextPowerOfTwo;
	maxW = width / (float) nextPowerOfTwo;

	glDrawBuffer(GL_BACK);

	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	glLoadIdentity();

	glEnable(GL_TEXTURE_2D);

	glMatrixMode(GL_PROJECTION);
	glPushMatrix();
	glLoadIdentity();
	glOrtho(0, width, height, 0, -1, 1);
	glMatrixMode(GL_MODELVIEW);
	glPushMatrix();
	glLoadIdentity();

	glBegin(GL_QUADS); {
		glTexCoord2f(0, maxH);
		glVertex2f(0, 0);

		glTexCoord2f(0, 0);
		glVertex2i(0, height);

		glTexCoord2f(maxW, 0);
		glVertex2i(width, height);
```

```
            glTexCoord2f(maxW, maxH);
            glVertex2i(width, 0);
      } glEnd();

      glDisable(GL_TEXTURE_2D);

      glMatrixMode(GL_PROJECTION);
      glPopMatrix();
      glMatrixMode(GL_MODELVIEW);
      glPopMatrix();
}
```

A.14 Inicialização do OpenStereo

```
slCreateContext();
slInitContext("pluginsFile.txt");

slCam = slCreateCamera();
slCam->aperture = 10;
slCam->focalLength = 0.5f;
slCam->eyeSeparation = 0.1f;

slCam->viewPosition.x = 0.0f;
slCam->viewPosition.y = 2.0f;
slCam->viewPosition.z = 5.0f;

slCam->viewDirection.x = 0.0f;
slCam->viewDirection.y = 0.0f;
slCam->viewDirection.z = -7.5f;

slCam->viewUp.x = 0.0f;
slCam->viewUp.y = 1.0f;
slCam->viewUp.z = 0.0f;

if (viewport == 0) {
      viewport = slCreateViewport(640, 480);
      slSetScene(viewport, DrawGLScene);
      slBindViewport(viewport);
      slSetCamera(viewport, slCam);
      slSetAnaglyphMethod(viewport, SL_OPTIMIZED_ANAGLYPH);
}

slSetDimension(viewport, 640, 480);
```

_____

Judith Kelner

_____

Veronica Teichrieb

_____

Pedro José Silva Leite