UNIVERSIDADE DO ESTADO DO RIO GRANDE DO NORTE
FACULDADE DE CIÊNCIAS EXATAS E NATURAIS
CAMPUS AVANÇADO DE NATAL
NÚCLEO AVANÇADO DE EDUCAÇÃO SUPERIOR DE SANTA CRUZ

JOSÉ RODRIGUES GUIMARÃES FILHO

UMA PLATAFORMA PARA MANIPULAÇÃO 3D NA REALIDADE AUMENTADA COM CONTEÚDO OPENGL DINÂMICO

Santa Cruz - RN
2015

JOSÉ RODRIGUES GUIMARÃES FILHO

UMA PLATAFORMA PARA MANIPULAÇÃO 3D NA REALIDADE AUMENTADA COM CONTEÚDO OPENGL DINÂMICO

Trabalho de Conclusão de Curso apresentado à Universidade do Estado do Rio Grande do Norte - UERN - como requisito obrigatório para obtenção do grau de bacharel em Ciência da Computação.

ORIENTADORA: Adriana Takahashi

SANTA CRUZ

2015

Catalogação da Publicação na Fonte.
Universidade do Estado do Rio Grande do Norte.

Guimarães Filho, José Rodrigues

Uma Plataforma Para Manipulação 3D na Realidade Aumentada Com Conteúdo OpenGL Dinâmico. / José Rodrigues Guimarães Filho– Santa Cruz, RN, 2015.

56 f.

Orientador(a): Prof. Adriana Takahashi

Monografia (Bacharelado) Universidade do Estado do Rio Grande do Norte. Curso de Ciência da Computação.

1. Realidade Aumentada. 2. Computação Gráfica. 3. Estrutura de Dados. I. I. Takahashi, Adriana. II. Universidade do Estado do Rio Grande do Norte. III.Título.

UERN/ BC CDD 004

Bibliotecário: Sebastião Lopes Galvão Neto – CRB - 15/486

JOSÉ RODRIGUES GUIMARÃES FILHO

UMA PLATAFORMA PARA MANIPULAÇÃO 3D NA REALIDADE AUMENTADA COM CONTEÚDO OPENGL DINÂMICO

Trabalho de Conclusão de Curso apresentado à Universidade do Estado do Rio Grande do Norte - UERN - como requisito obrigatório para obtenção do grau de bacharel em Ciência da Computação.

Aprovado em____/____/_____.

Banca Examinadora:

_____
Profª. Drª. Adriana Takahashi
Universidade do Estado do Rio Grande do Norte

_____
Prof. Dr. Francisco Dantas de Medeiros Neto
Universidade do Estado do Rio Grande do Norte

_____
Prof. Me. Bruno Cruz de Oliveira
Universidade do Estado do Rio Grande do Norte

Dedico este trabalho:

Em primeiro lugar à minha Mãe, que sacrificou-se por meus estudos.

Em segundo, ao meu Pai, que mostrou-me o caminho das pedras.

Em terceiro, a deus.

RESUMO

Este trabalho tem por objetivo, conferir ao usuário comum, a capacidade de criar conteúdos de Realidade Aumentada, sem que para isto seja necessário que o mesmo tenha conhecimentos prévios de programação. Foi criado um protótipo de um sistema de modelagem de sólidos que funciona diretamente na Realidade Aumentada e que possibilita a implementação dinâmica do código OpenGL em tempo de execução, através de técnicas computacionais de alocação de memória.

Palavras-Chave: Realidade Aumentada. Computação Gráfica. Estrutura de Dados. Alocação Dinâmica. Implementação Dinâmica.

ABSTRACT

This work aims, give the average user, the ability to create Augmented Reality content, unless necessarily prior knowledge of programming. A prototype of a solid modeling system, that works directly on Augmented Reality and allows the dynamic implementation of OpenGL code at runtime, using computational memory allocation techniques was created.

Keywords: Augmented Reality. Computer Graphics. Data structure. Dynamic allocation. Dynamic implementation.

LISTA DE SIGLAS

CAD: Computer-Aided Desingn

API: Application Programming Interface

A/D: Analógico/Digital

HITLAB: Human Interface Technology Laboratory

RA: Realidade Aumentada

AR: Augmented Reality

CG: Computação Gráfica

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Conforme Moraes (2008), em sua cronologia acerca da computação gráfica, o termo “computação gráfica” foi criado por Willian Fetter em 1960, para descrever o que ele fazia na Boeing, que era projetar modelos tridimensionais em computadores. Naquela época, trabalhar com computação gráfica era uma tarefa muito complicada, desenvolvida apenas por programadores. A interface gráfica de usuário e periféricos de controle, que permitem usuários comuns trabalharem com computação gráfica, só vieram estar disponíveis no mercado em 1983, 23 anos após a criação do termo. E foi também na década de 80 que surgiram os primeiros programas de CAD (Desenho Assistido por Computador) e, especificamente em maio de 1985, os programas CAD começam a trabalhar com manipulação tridimensional.

Foi à partir daí que os usuários comuns (não-programadores) começaram a manipular formas tridimensionais sem conhecimentos prévio de programação, dentro da Realidade Virtual. Esta, representada pelo ambiente de modelagem.

A Realidade Aumentada, é definida como um ambiente composto por visualizações do mundo real acrescido de informações virtuais. Ela teve seu primeiro registro de projeto, segundo KIRNER (2008), em 1981, quando o simulador Super Cockpit, da Força Aérea Americana passou a operar com um capacete dotado de um visor acrílico, que permitia a visão direta, acrescida de informações virtuais projetadas no mesmo.

Em 1999, como forma de facilitar o desenvolvimento de aplicativos com Realidade aumentada, a API (Interface de Programação de Aplicativos) ARToolKit, foi criada. Desde a criação do ARToolkit até o presente momento, somente programadores podem executar manipulações em formas geométricas na realidade aumentada, através de sucessivos testes e compilações de código. Pode-se dizer que a Realidade Aumentada encontra-se atualmente, no que diz respeito à sua disponibilidade, no mesmo estado da Realidade Virtual até 1985, quando os programas CAD não trabalhavam ainda com manipulação tridimensional e somente programadores criavam formas tridimensionais em computadores.

1.1 OBJETIVOS GERAIS

Implementar uma plataforma que confira recursos necessários à criação e edição de objetos tridimensionais, diretamente na Realidade Aumentada sem que para isso, sejam necessário conhecimentos de programação por parte do usuário.

1.2 OBJETIVOS ESPECÍFICOS

Desenvolver uma plataforma na qual seja possível criar figuras tridimensionais na Realidade Aumentada de forma direta, sem a necessidade de um segundo software modelador ou de sucessivas compilações para teste de códigos e modelagens. Para isso será necessário:

- Pesquisar tópicos relevantes sobre sobre Realidade Aumentada, ARToolKit, OpenGL e códigos dinâmicos;
- Encontrar um meio de dinamizar a manipulação tridimensional, afim de eximir o usuário final de conhecimentos avançados de informática;
- Definir um escopo de funcionamento do sistema e
- Implementar um protótipo com as ferramentas básicas necessárias.

1.3 ESTRUTURAÇÃO DO DOCUMENTO

No segundo capítulo, é abordado o referencial teórico, no qual consta os conceitos e funcionamento do ARToolKit, ferramenta utilizada para implementação de Realidade Aumentada. Ainda nesse capítulo avança-se sobre OpenGL. Posteriormente, tem-se uma abordagem sobre códigos dinâmicos, seu conceito e sua aplicação na plataforma.

O terceiro capítulo, apresenta a modelagem do sistema, onde é abordada a dinamização do código OpenGL e a formação das estruturas de dados que compõem a plataforma, assim como o funcionamento geral do sistema.

O quarto capítulo, é dedicado à implementação do sistema, no qual eu mostro os pré-requisitos para implementação e execução do mesmo, assim como os pseudocódigos das principais funções que o compõe.

Como desfecho do trabalho, o capítulo 5 traz as considerações, explicitando se os objetivos foram alcançados ou não, finalizando dessa forma, este trabalho acadêmico.

2 REFERENCIAL TEÓRICO

2.1 REALIDADE AUMENTADA

A Realidade Aumentada, é definida como um ambiente no qual as imagens geradas através de Computação Gráfica, são adicionadas às visualizações do mundo real, captadas digitalmente. (KIRNER; TORI, 2004).

Segundo Azuma (1997), a Realidade Aumentada, difere da Realidade Virtual no que diz respeito ao ambiente na qual ambas estão inseridas. A Realidade Virtual, utiliza-se de um ambiente completamente gerado através de computação gráfica, seja este ambiente um cenário fictício (como em filmes e jogos feitos em Computação Gráfica), uma cópia de um cenário real (como em sistemas simuladores) ou um ambiente de edição, no qual os elementos virtuais serão criados e editados (como em softwares de modelagem tridimensional). O ambiente utilizado pela Realidade Aumentada é uma digitalização do nosso ambiente real, capturada por câmeras digitais, e acrescido de informações visuais geradas por processamento gráfico, daí a denominação “Realidade Aumentada”.

2.1.1 Funcionamento da Realidade Aumentada.

Conforme Hautsch (2009), para que a Realidade Aumentada, com captura em tempo real das imagens do ambiente físico, funcione efetivamente, se faz necessário 3 premissas básicas. São elas:

- Um dispositivo de vídeo digital;
- Um sistema implementado para a prática da Realidade Aumentada;
- Uma ligação entre os itens supracitados.

O dispositivo de vídeo tem que ser necessariamente digital, pois as imagens precisam ser tratadas em tempo real pelo software de Realidade Aumentada. Em um dispositivo analógico (utilizado hoje em dia apenas em filmagens profissionais), por melhor que seja a qualidade da captura, se faz necessário uma conversão A/D

(Analógico/Digital) para que a imagem seja tratada pelo software de Realidade Aumentada e tal conversão denota tempo de processamento, o que vem a ser um empecilho à execução em tempo real. (FERREIRA, 2015)

O sistema utilizado necessita ser implementado especificamente para o uso da Realidade Aumentada. Pois nele, as imagens captadas pela câmera digital, serão tratadas com técnicas de visão computacional, a fim de identificar algum marcador ou estrutura específica que ative os elementos digitais na imagem capturada. No caso de marcadores, os elementos digitais serão inseridos na imagem tomando por base sua localização e posicionamento.

O elo de ligação entre a câmera digital e o *software* de Realidade Aumentada precisa ser um dispositivo que, além de ser capaz de executar o software e ser compatível com a câmera digital, tenha um periférico de saída de vídeo pois, neste periférico é onde será exibido o ambiente da Realidade Aumentada. Um micro computador é a melhor opção, tendo em vista que a maioria vêm com câmera de vídeo e monitor, embora a Realidade Aumentada também possa ser executada em aparelhos portáteis como *tablet's* e celulares.

É fundamental também, que o ambiente físico esteja bem iluminado, para que a imagem seja capturada com clareza e o *software* processe bem as informações captadas, caso contrário, não ocorrerá interação entre o vídeo e o *software* de Realidade Aumentada (Truyenque, 2005).

2.2 ARTOOLKIT

O ARToolKit é um conjunto de ferramentas de apoio à implementação de sistemas de Realidade Aumentada, desenvolvido por Hirokazo Kato em 1999, após seu ingresso no Laboratório de Tecnologia de Interface Humana (HITLab) (HITLAB, 2014a).

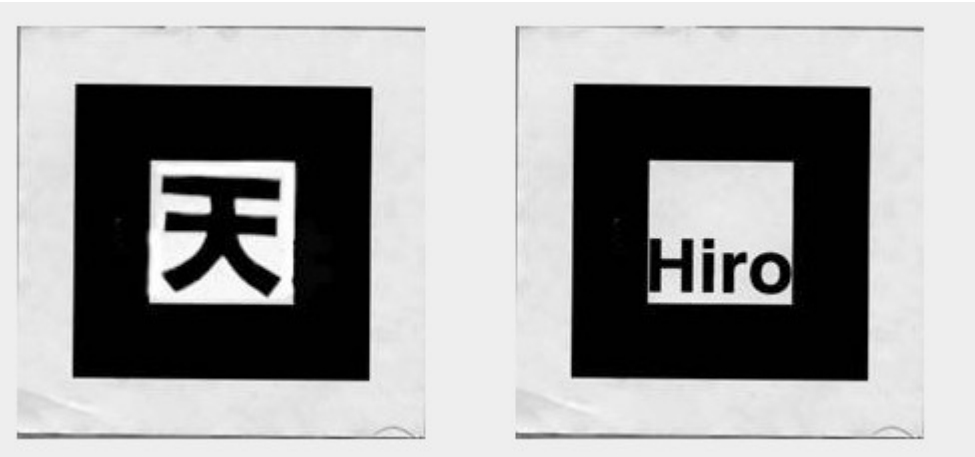
> "Uma das principais dificuldades que existiam na implementação de sistemas de Realidade Aumentada, era calcular com precisão o ponto de vista do usuário, em tempo real, para que as imagens virtuais ficassem alinhadas com os objetos do mundo real." (HITLAB, 2014b, tradução nossa)

Através de técnicas de visão computacional, o ARToolKit calcula a posição da câmera de vídeo em relação à marcas específicas, impressas em uma superfície

plana. Com a biblioteca ARToolKit realizando tais cálculos, cabe ao programador apenas inserir sólidos geométricos por sobre estas marcas e tal abstração permite um desenvolvimento mais rápido de aplicações em Realidade Aumentada.

Os marcadores são quadrados pretos identificados pelo rastreamento do ARToolKit conforme podem ser visto no exemplo abaixo, ilustrado pela Figura 1:

Figura 1: Exemplo de marcadores padrão



Fonte: Site do Laboratório de Tecnologia de Interface Humana[1]

Basicamente, a câmera captura a imagem do mundo real, manda para o computador e o *software* faz uma busca por estes quadriláteros. Caso algum quadrilátero seja identificado o *software* realiza cálculos matemáticos para determinar a posição da câmera em relação a ele. Após determinada a posição da câmera, uma figura virtual é desenhada sobre o vídeo do mundo real no mesmo local do quadrilátero marcador e no mesmo ângulo de inclinação, tendo como referência a posição da câmera. O resultado final é mostrado no periférico de saída de vídeo (monitor, projetor, óculos de realidade virtual, etc.) utilizado pelo usuário, que por sua vez enxerga gráficos sobrepostos ao mundo real. (HITLAB, 2014c) Vide infógrafo abaixo (Figura 2):

[1] Disponível em:<http://www.hitl.washington.edu/artoolkit/documentation/devmulti.htm> Acesso em Jul. 2014.

Figura 2: Infográfico explicativo

Fonte: Site do Laboratório de Tecnologia de Interface Humana[2]

O princípio de desenvolvimento pode ser resumido em seis passos:

1. Inicializar a captura de vídeo, ler os arquivos de marcadores padrão e os parâmetros de câmera;
2. Capturar um quadro (*frame*) de vídeo;
3. Detectar e reconhecer os marcadores no *frame* de vídeo;
4. Calcular a transformação de câmera relativa ao marcador detectado;
5. Desenhar os objetos virtuais nos marcadores detectados, e
6. Finalizar a captura de vídeo.

Os passos 2, 3, 4 e 5, são repetidos continuamente nesta sequência até que a aplicação termine. (HITLAB, 2014d)

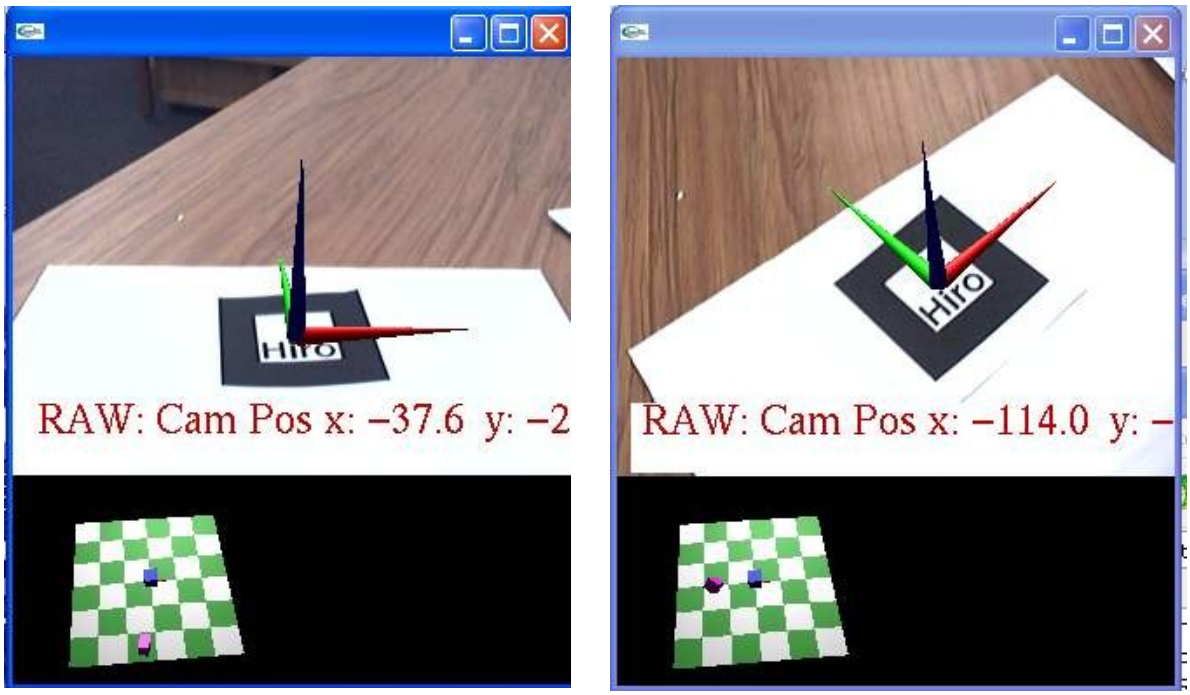
Abaixo seguem algumas imagens exemplos do uso de ARToolKit:

[2] Disponível em:<http://www.hitl.washington.edu/artoolkit/documentation/userarwork.htm> Acesso em Jul. 2014.

Figura 3: Marcador sem sobreposição 3D

Fonte: Site do Laboratório de Tecnologia de Interface Humana[3]

Figura 4: Marcadores com sobreposição 3D em diferentes ângulos de câmera



Fonte: Site do Laboratório de Tecnologia de Interface Humana[4]

[3] Disponível em:<http://www.hitl.washington.edu/artoolkit/documentation/usersetup.htm> Acesso em Jul. 2014.
[4] Disponível em:<http://www.hitl.washington.edu/artoolkit/documentation/tutorialcamera.htm> Acesso em Jul. 2014.

2.3 OPENGL

O OpenGL, é uma das API's utilizadas pelo ARToolKit para desenhar formas geométricas. O OpenGL começou como uma iniciativa da Silicon Graphics International, de criar uma API de fornecedor independente para o desenvolvimento de aplicativos gráficos em 2D e 3D (SGI, 2015).

Segundo Sheiner (2009) e Mendonça (2015), o OpenGL foi desenvolvido em 1991, como uma interface independente do hardware, otimizado para funcionar em diferentes plataformas, pois cada fabricante de tinha seu próprio conjunto de instruções para desenhos de gráficos e era muito dificultoso construir aplicações que suportassem os distintos hardwares existentes. Para alcançar tais objetivos de portabilidade, não existem no OpenGL nativo, comandos para executar gerenciamento de janelas ou obtenção de entrada de comandos por parte do usuário, tais gerenciamentos devem ser implementados pelo programador de acordo com as especificidades do hardware utilizado. Assim, o OpenGL não funciona como uma linguagem de programação e sim, como uma biblioteca de linguagem.

> “Normalmente se diz que um programa é baseado em OpenGL ou é uma aplicação OpenGL, o que significa que ele é escrito em alguma linguagem de programação que faz chamadas a uma ou mais bibliotecas OpenGL.” (MANSOUR, 2015)

Existem hoje disponíveis no mercado, várias API’s de modelagem gráfica, como Glide, Direct3D, Java3D, etc., porém o ARToolKit só utiliza OpenGL e Flash (na recente versão FLARToolKit desenvolvida pelo HITLAB). Optamos em utilizar OpenGL em detrimento ao Flash, pois a tecnologia flash, segundo Jobs (2010), ainda não está otimizada para equipamentos portáteis, fazendo com que seja usado muito processamento e consequentemente haja uma redução acelerada da carga das baterias. Isto seria um problema, no caso de uma futura migração da plataforma para equipamentos portáteis.

2.4 CÓDIGOS DINÂMICOS

A ideia de códigos dinâmicos, consiste em códigos-fonte mutáveis, que não são alterados diretamente por um programador e sim por outro programa em

execução. Os códigos dinâmicos vêm sendo utilizados com abrangência na internet e foi essa funcionalidade, em que os usuários interagem com o código fonte, segundo Bento(2015), que nomeou o conceito de web 2.0.

Geralmente, quando executamos um programa de RA, as figuras mostradas por sobre o marcador, são imagens, animadas ou não, definidas pelo código OpenGL, que foi inserido pelo programador no código fonte do *software* no momento da implementação do programa. Caso um usuário comum queira fazer alguma alteração na figura ou informação exibida, ele não vai conseguir pois, o código OpenGL é um conteúdo estático e somente um programador poderia fazê-lo, alterando o código fonte e recompilando o software.

A utilização do conceito de códigos dinâmicos, aplicado ao OpenGL tem por objetivo permitir que o usuário de Realidade Aumentada tenha a capacidade de fazer alterações em uma figura exibida em um marcador RA, sem que para isso seja necessário saber programar em C ou ter conhecimentos de OpenGL e atualmente não existe nenhum programa de edição 3D que utilize-se da edição direta na Realidade Aumentada.

3 MODELAGEM

3.1 DINAMICIDADE DO OPENGL E ESTRUTURAS DE DADOS

Conforme visto no capítulo 2.3, o ARToolKit utiliza a API OpenGL para desenhar as figuras tridimensionais que ficam por sobre seus marcadores. Porém, para que seja possível a implementação de conteúdo dinâmico com OpenGL, (alteração das figuras sem ser necessário recompilar código) precisamos encontrar um meio de armazenar as figuras (ou suas diretivas) na memória principal, em forma de variável, de modo que o usuário possa fazer as alterações que forem devidas.

Um problema encontrado a princípio, é o fato do OpenGL não armazenar primitivas (pontos, linhas, triângulos, quadriláteros, etc.) em endereços de memória. O OpenGL delineia as figuras na tela através de suas funções, conforme pode ser visto no exemplo abaixo:

```
1:   glBegin(GL_QUADS);
2:        glVertex3f(0.0f, 0.0f, 0.0f);
3:        glVertex3f(0.0f, 2.0f, 0.0f);
4:        glVertex3f(2.0f, 2.0f, 0.0f);
5:        glVertex3f(2.0f, 0.0f, 0.0f);
6:   glEnd();
```

Este trecho de código desenha um quadrilátero em um espaço tridimensional. A linha 1, é uma chamada da função `GL_QUADS`, que desenha um quadrilátero a cada quatro vértices declarados no formato "`glVertex3f(X, Y, Z);`" onde os valores "`X`", "`Y`" e "`Z`" representam seus homônimos dentro do plano cartesiano, como pode ser visto nas linhas de 2 a 5 do exemplo acima. A linha 6 conclui a função `GL_QUADS` com "`glEnd();`"

Uma forma de resolver a questão do armazenamento de primitivas na memória principal, é armazenar apenas as coordenadas de seus vértices através de variáveis, de forma tal que o usuário possa modificá-las posteriormente na mesma execução do processo que criou a primitiva gráfica. A função `glVertex3f(2.0f,`

`2.0f, 0.0f)` (linha 4) por exemplo, pode ser reescrita com variáveis no espaço correspondente às coordenadas dos vértices, da seguinte forma: `glVertex3f(var1, var2, var3);`.

Assim, o exemplo acima pode ser reescrito da seguinte maneira:

```
1:   glBegin(GL_QUADS);
2:        glVertex3f(var1, var2, var3);
3:        glVertex3f(var4, var5, var6);
4:        glVertex3f(var7, var8, var9);
5:        glVertex3f(var10, var11, var12);
6:   glEnd();
```

Com todas as variáveis (de `var1` a `var12`) sendo declaras na forma de ponto flutuante, caso sejam necessárias mudanças minuciosas como uma pequena translação em uma primitiva.

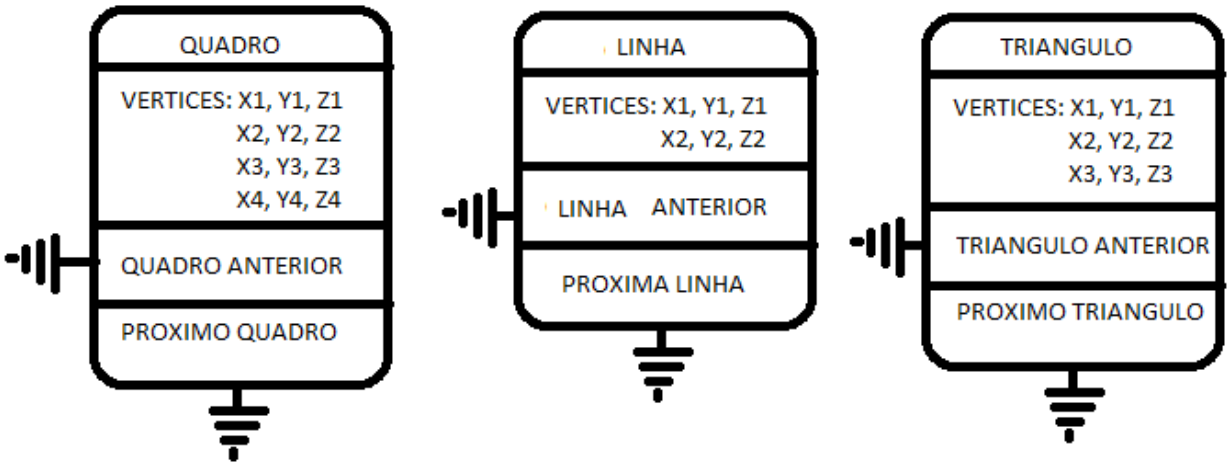
Não obstante, vale salientar, que cada usuário do sistema criará números diferentes de formas tridimensionais (dependendo da complexidade de seu projeto) e assim sendo, é inviável limitar o número de primitivas gráficas, declarando todas as variáveis no início do código fonte do programa. O ideal seria, que as reservas de endereços de memória (variáveis) fossem criadas em tempo de execução à partir das necessidades do usuário ou do projeto. Para tanto, faz-se necessário o uso de Alocação Dinâmica de Memória.

> “A alocação dinâmica é o processo que aloca memória em tempo de execução. Ela é utilizada quando não se sabe ao certo quanto de memória será necessário para o armazenamento das informações, podendo ser determinadas em tempo de execução conforme a necessidade do programa. Dessa forma evita-se o desperdício de memória.” (BATTISTI, 2015).

A estrutura de dados que forma a base de todas as implementações com alocação dinâmica de memória é a Lista Encadeada. Ela consiste em uma sequência de estruturas, onde cada uma contém uma ou mais variáveis e o endereço da estrutura seguinte seguinte. (IME-USP, 2015). Em nossa modelagem utilizaremos uma lista encadeada para a definição e armazenamento dos vértices, criando assim um tipo abstrato de dados para cada forma geométrica utilizada, além de um endereçamento extra para a célula anterior.

Exemplo:

Figura 5: Exemplos de estruturas



Neste formato, as variáveis estão alocadas dentro de suas respectivas estruturas de forma organizada, porém, temos um problema no que diz respeito à alocação de formas diferentes da anterior, pois a locação dinâmica não permite que um tipo de dado pré-definido aponte para um endereço de memória que não corresponde ao seu tamanho. Se, por exemplo, o usuário criar um ponto e logo após um triângulo, a estrutura correspondente ao ponto teria um espaço de endereçamento de 1 (um) vértice apontado para um próximo espaço de endereçamento contendo 3 (três) vértices. Isso causaria uma falha de segmentação na memória, ou até mesmo um problema maior, caso o endereço de memória esteja sendo utilizado pelo sistema operacional.

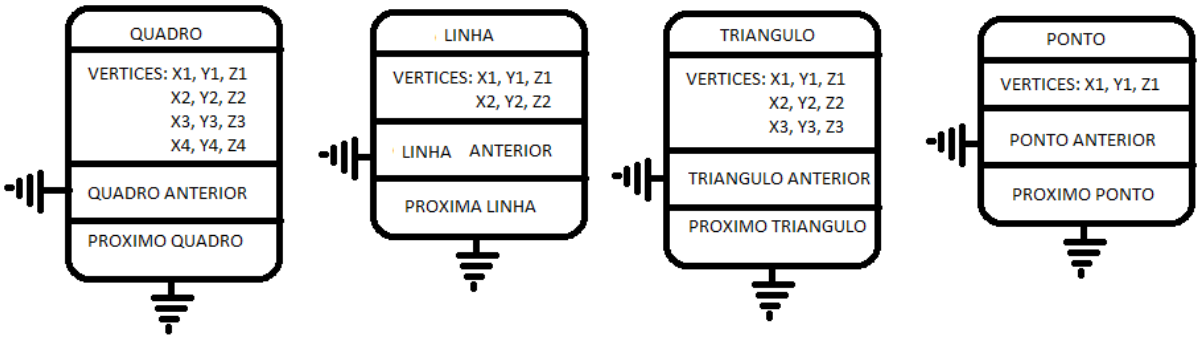
Existem duas maneiras plausíveis de contornar tal situação. A primeira, seria utilizar ponteiros genéricos nas variáveis que definem a próxima forma e a forma anterior, e tais ponteiros seriam convertidos no momento da alocação. Parece uma solução simples à primeira vista, entretanto, dificultaria todo o processo de programação. Cada tipo de forma só poderia ser alterada na lista, por um(a) função específica, pois cada função recebe um tipo de dados definido em seu escopo. Se, por exemplo, nosso programa fosse composto por apenas quatro tipos de formas (ponto, linha, triângulo e quadrado) teríamos de implementar funções de transformação para cada forma em específico. Por exemplo:

rotacionarPonto(tipoPonto), rotacionarLinha(tipoLinha),
rotacionarTriangulo(tipoTriangulo), rotacionarQuadro(tipoQuadro),
transladarPonto(tipoPonto), transladarLinha(tipoLinha),
transladarTriangulo(tipoTriangulo), transladarQuadro(tipoQuadro),
ampliarPonto(tipoPonto), ampliarLinha(tipoLinha),
ampliarTriangulo(tipoTriangulo), ampliarQuadro(tipoQuadro), etc.

Além das conversões de dados que seriam feitas em cada ponteiro genérico no momento da alocação.

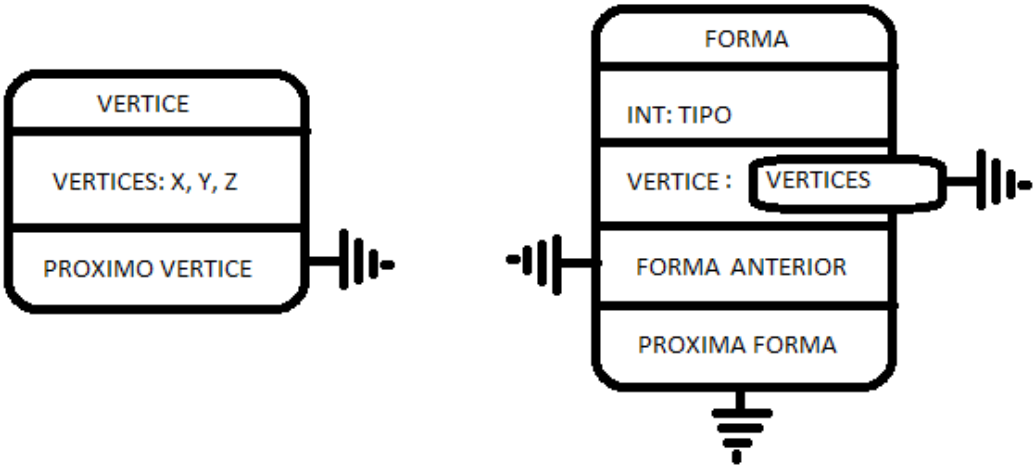
A segunda forma encontrada, consiste em utilizar apenas um tipo de dado pré-definido para todas as formas primitivas do sistema. Em uma análise rápida, dá para notar que a diferença existente entre as formas, é o número de vértices, como pode ser visto na figura abaixo:

Figura 6: Exemplos de estruturas 2



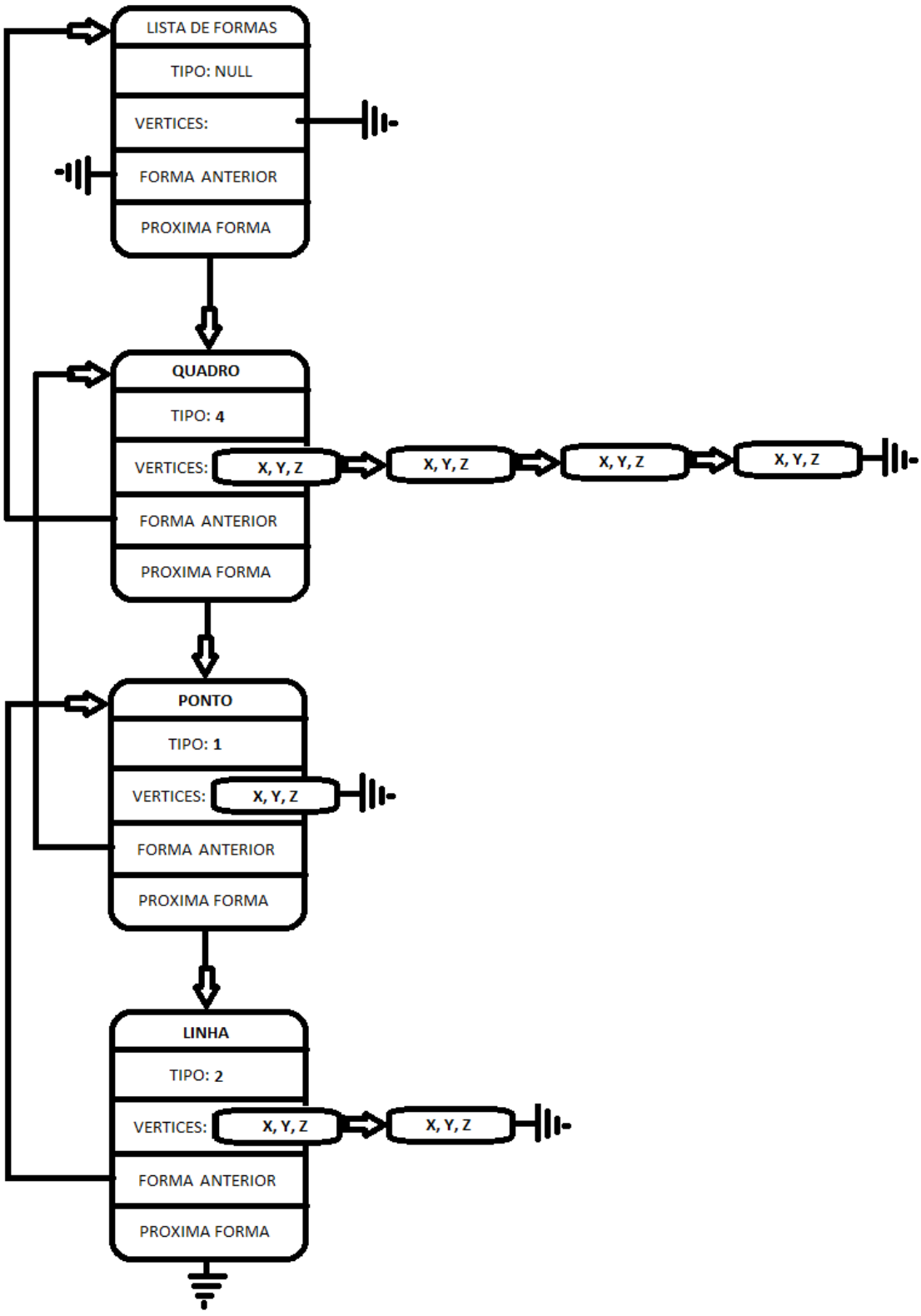
Assim sendo, para aglutinarmos todas as formas em uma só, temos que usar alocação dinâmica para os vértices, ficando assim uma lista encadeada de vértices dentro de cada elemento que compõe a lista de formas e os vértices serão alocados também em tempo de execução.

Figura 7: Estruturas definitivas



Uma nova variável (TIPO), dentro da estrutura FORMA, foi criada para que o programa saiba que primitiva manipular. Se um usuário criar respectivamente um quadrilátero, um ponto e uma linha, nossa estrutura de dados ficaria com o seguinte formato:

Figura 8: Lista com elementos inclusos



O primeiro elemento da lista, denominado listaDeFormas, do tipo NULL, é a variável global que será utilizada por todas as funções do programa. O tipo inicial dela é “nulo” pois não sabemos qual será a primeira forma desenhada pelo usuário.

Neste escopo, só precisamos delinear dois tipos de estruturas de dados (“forma” e “vertices”) ao invés de uma para cada primitiva(“ponto”, “linha”, “triangulo”, “quadro”, “cone”, “esfera”, “cilindro”, “cubo”, etc.) e assim, cabe às funções de inclusão do programa, montar a estrutura de dados de acordo com a forma escolhida pelo usuário.

Utilizaremos à partir daqui as terminologias **listaDeFormas** e **listaDeVertices**, para nos referir às supracitadas estruturas de dados.

3.2 FUNCIONAMENTO GERAL DO SISTEMA

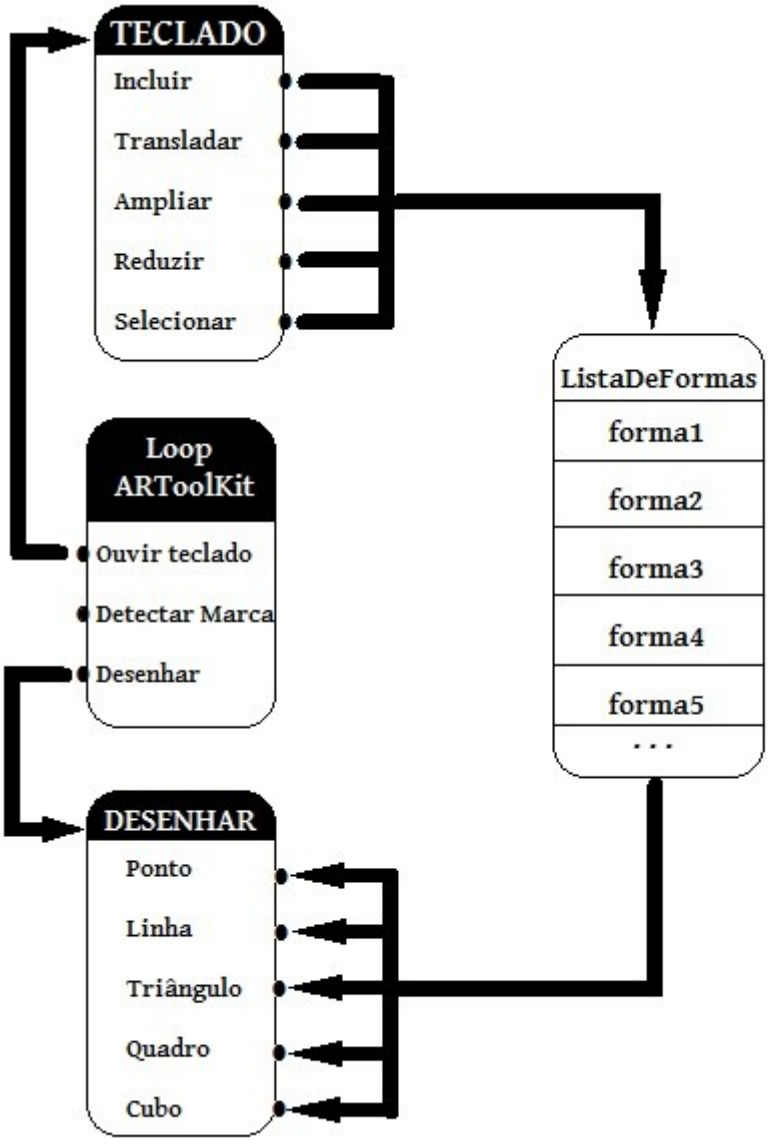
O funcionamento de um programa de edição, assim como de qualquer programa que interage com o usuário de forma dinâmica, se faz através de um loop constante, onde o programa aguarda um comando e executa uma ação, ininterruptamente até ser finalizado. O nosso sistema utiliza dois grupos de ações logo após o comando do usuário: os comando condicionais e os comandos absolutos.

As ações condicionais só serão executadas mediante um comando específico do usuário através do teclado. São as ações responsáveis pelas alterações na listaDeFormas. Em suma, toda e qualquer alteração na listaDeFormas só será feita pelo usuário. A função que ministra os comando do teclado é responsável por “chamar” as demais funções que fazem alterações na listaDeFormas. São elas, as funções de inclusão, translação, ampliação, redução e seleção de elementos na listaDeFormas.

As ações absolutas são aquelas que são executadas sem qualquer relação com os comandos do usuário. São as ações responsáveis pela detecção do marcador AR e por desenhar por sobre o marcador. As função que desenha sobre o marcador é a função responsável por “chamar” as funções que tornarão o OpenGL dinâmico. São elas que leem as variáveis da listaDeFormas e aplicam no código OpenGL.

Segue o diagrama com o funcionamento geral do sistema, explanando a relação entre o loop ARToolKit e a listaDeFormas:

Figura 9: Diagrama de Funcionamento Geral



4 IMPLEMENTAÇÃO

Neste capítulo serão expostos os requisitos de implementação e seus pseudocódigos.

4.1 REQUISITOS PARA IMPLEMENTAÇÃO E EXECUÇÃO

A implementação e execução de Realidade Aumentada com ARToolKit, não requer uma máquina com configuração muito avançada. Um computador com configuração básica (tomando como base, os vendidos no mercado atual) dotado de um dispositivo de captura de vídeo é mais do que suficiente. Porém, o Sistema Operacional precisa ter alguns componentes instalados antes da implementação e execução.

> "O ARToolKit, trata-se de uma coleção de bibliotecas de software, criada para ser usada em aplicações. Por esta razão, o ARToolKit é uma distribuição com o código aberto e você precisa compilá-lo em seu Sistema Operacional e plataforma específica." (HITLAB, 2015, tradução nossa)

Baseado nisso, o ARToolKit tem diferentes premissas de instalação para cada Sistema Operacional, apesar de oferecer funções similares em cada um deles. No Microsoft Windows você precisa utilizar:

- um ambiente de desenvolvimento;
- a biblioteca de vídeo DSVideoLib na versão 0.0.8b ou superior, que serve para desenvolvimento de aplicações que requiram entrada de vídeo;
- a biblioteca de funcionalidades GLUT, para o OpenGL;
- o DirectX, que é uma coleção de API que padroniza a comunicação entre *software* e *hardware*;
- dispositivo de entrada de vídeo.

Para o Sistema Operacional Linux:

- OpenGL e GLUT;
- biblioteca de vídeo instalada;
- dispositivo de entrada de vídeo.

E no MAC OS X:

- a ferramenta de desenvolvimento da Apple, XCode;
- o *driver* de câmera (já incluso à partir do Mac OS X 10.3)
- dispositivo de entrada de vídeo.

(HITLAB, 2015)

Neste projeto, foi utilizado o Sistema Operacional Microsoft Windows 7 e um *Notebook* Sony Vaio VPC-EE25FB. Como ferramenta de desenvolvimento, optamos pelo DEV C++, da Bloodshed Software.

Além de todas essas premissas, o ambiente físico no qual o *software* será executado deve estar bem iluminado, para que as informações sejam processadas de forma correta pelo ARToolKit, conforme visto na seção 3.2.1.

4.2 PSEUDOCÓDIGOS

O algoritmo simplificado para funcionamento do sistema consiste em cinco passos:

1. capturar *frame* do vídeo;
2. detectar um marcador AR no frame capturado;
3. fazer transformações geométricas entre o marcador e a câmera real;
4. desenhar sobre o marcador no frame capturado;
5. capturar os comandos do teclado.

esses cinco passos são repetidos continuamente até que o programa seja encerrado.

Com a abstração concedida pelo ARToolKit, não se faz necessário implementar os três primeiros passos. Em *softwares* mais simples o quarto passo também não precisaria ser implementado, pois geralmente as imagens a serem desenhadas sobre o marcador, são formas OpenGL trazidas prontas de fontes externas. Em nosso software parte desse processo deve ser implementado. A configuração do desenho em si deve ser feita em tempo de execução pelo usuário por tanto o *software* deve suprir tal necessidade com os recursos necessários. Já a parte de colocar a imagem sobre o marcador, fica por conta do ARToolKit.

O quinto passo deve ser implementado, pois apesar de existir uma função prévia de captura de comandos do teclado no próprio ARToolKit, ela vem com um padrão muito simples, onde só há o comando de fechar o programa ao pressionar a tecla “Esc”. O quinto passo vem a ser um dos mais importantes em nosso software, pois é ele quem faz as chamadas de todas as funções de modelagem e a manipulação das estruturas de dados do programa.

Começaremos a detalhar logo o passo 5, que contém premissas para execução do passo 4.

4.2.1 Função Teclado

A função de captura de teclado consiste em um módulo de leitura simples, com vários gatilhos, que fazem chamadas para as diversas funções do sistema. A grosso modo pode ser escrita da seguinte forma:

```
1:   INÍCIO funçãoTeclado
2:   .    capturarEntrada
3:   .         casoEntrada=Esc
4:   .              finalize o programa
5:   .         casoEntrada=1
6:   .              adicione um ponto na listaDeFormas
7:   .         casoEntrada=2
8:   .              adicione uma linha na listaDeFormas
9:   .         casoEntrada=3
10:  .              adicione um triângulo na listaDeFormas
```

```
11:  .          casoEntrada=4
12:  .               adicione um quadrado na listaDeFormas
13:  .          casoEntrada=s
14:  .               ampliar forma selecionada
15:  .          casoEntrada=S
16:  .               reduzir forma selecionada
17:  .          casoEntrada=t
18:  .               transladar forma selecionada
19:  .          casoEntrada=T
20:  .               transladar forma selecionada no sentido
                     contrário
21:  .          casoEntrada=+
22:  .               selecionar proxima forma da lista
23:  .          casoEntrada=-
24:  .               selecionar forma anterior da lista
25:  .          casoEntrada=X
26:  .               ativar ou desativar eixo X no pivô de
                     orientação
27:  .          casoEntrada=Y
28:  .               ativar ou desativar eixo Y no pivô de
                     orientação
29:  .          casoEntrada=Z
30:  .               ativar ou desativar eixo Z no pivô de
                     orientação
31:  FIM funçãoTeclado
```

As entradas de comandos representadas por números nas linhas 5, 7, 9 e 11, são relativas ao número de vértices contidos em cada forma geométrica. Um ponto só contém um vértice, por tanto definimos como “1”, o comando para adicionar um ponto na lista de formas, ficando assim intuitivo ao usuário a relação de comandos. Uma reta, que contém dois vértices tem como comando de inclusão “2”, Um triângulo “3”, um quadrilátero “4” e assim por diante. A adição de um cubo, por exemplo, poderia ter como comando “8”, em virtude do mesmo possuir oito vértices. Demais formas podem ser incluídas com comandos que lembrem a referida forma.

Nas linhas 13 e 15, os comando “s” e “S” chamam a função que altera a escala de uma forma geométrica, respectivamente de forma decrescente e crescente. As linhas 17 e 19 mostram os comandos para mover (transladar) um objeto em relação ao pivô do marcador através da chamada de função “transladar”. Todas essas funções serão explanadas com mais detalhes.

Os comando “+” e “-”, presentes nas linhas 21 e 23, servem para selecionar as formas presentes na estrutura de dados principal do programa (listaDeFormas), percorrendo toda lista encadeada e deixando a forma selecionada com uma cor mais evidente, para que o usuário saiba qual forma está sendo manipulada.

Os três últimos comandos desse módulo de leitura (“X”, “Y” e “Z”) servem para ativar e desativar os eixos do pivô de orientação. É importante que os eixos tenham ativação independente para que o usuário possa melhor orientar-se no momento em que pretenda fazer alguma alteração no posicionamento de uma forma através da função “transladar”.

4.2.2 Adicionando Formas

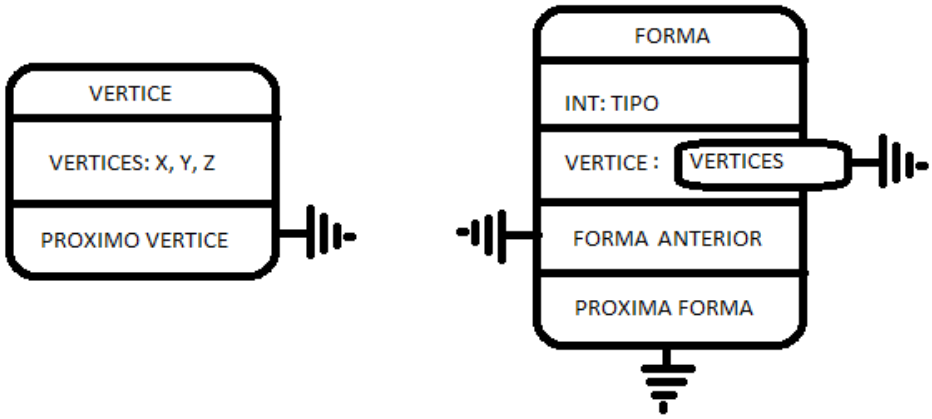
As funções que adicionam elementos da listaDeFormas na memória principal funcionam recursivamente fazendo uma auto chamada até o final da lista. Desta forma assegura-se que as formas sejam adicionadas apenas no final, conferindo assim um controle organizado para o usuário. Vide um exemplo genérico para adição de novas formas na listaDeFormas:

```
1:   INÍCIO adicionarForma(listaDeFormas)
2:   .    SE próximo elemento da listaDeFormas for vazio
3:   .    .    listaDeFormas.proximo<-nova forma
4:   .    .    listaDeFormas.proximo.vertice<-novo vertice
5:   .    SENÃO adicionarForma(listaDeFormas.proximo)
6:   FIM adicionarForma
```

Na segunda linha, ele verifica se a lista realmente encontra-se no final para poder executar a inclusão do objeto, caso contrário ele faz uma nova chamada de função referenciando o próximo nódulo da lista (linha 5) até que encontre o final da

mesma. Nas linha 3 e 4 ele adiciona uma nova forma ao final da lista e inclui uma nova lista de vértices associada a forma recém criada. (Não existem formas geométricas sem vértices). Para relembrar o formato de cada elemento da lista de forma e sua relação com a lista de vértices:

Figura 7: Estruturas definitivas



É um algoritmo simples que serve como base para todo e qualquer tipo de forma, mas levando-se em conta formas distintas têm número distintos de vértices então devem ser implementadas várias funções de inclusão, sendo uma para cada tipo (adicionarPonto, adicionarLinha, adicionarCubo...). Vejamos por exemplo como ficaria o algoritmo de uma função para adicionar uma linha (dois vértices), com as coordenadas XYZ (0;0;0) e (20;20;20) ao final da lista:

```
1:  INÍCIO adicionarLinha(listaDeFormas)
2:  .  SE próximo elemento da listaDeFormas for vazio
3:  .  .    listaDeFormas.proximo<-nova forma
4:  .  .    listaDeFormas.proximo.vertice<-novo vertice
5:  .  .    listaDeFormas.proximo.vertice.X<-0
6:  .  .    listaDeFormas.proximo.vertice.Y<-0
7:  .  .    listaDeFormas.proximo.vertice.Z<-0
8:  .  . listaDeFormas.proximo.vertice.proximo<-novo vertice
9:  .  .    listaDeFormas.proximo.vertice.proximo.X<-20
10: .  .    listaDeFormas.proximo.vertice.proximo.Y<-20
```

```
11: .   .    listaDeFormas.proximo.vertice.proximo.Z<-20
12: . SENÃO adicionarLinha(listaDeFormas.proximo)
13: FIM adicionarLinha
```

O padrão inicial é mantido até a quarta linha. À partir daí o número de vértices da forma geométrica vai definir o escopo de cada algoritmo.

Figura 10: Adicionando formas diversas

4.2.3 Percorrendo a Lista de Formas

A função de percorrer a lista de formas, tem por finalidade selecionar o elemento a ser manipulado pelo usuário. Conforme o capítulo 5.2.1, ela é ativada através das teclas “+” e “-”, representando respectivamente o percurso ascendente e descendente de seleção na lista. Podemos acrescentar uma variável booleana aos elementos da listaDeFormas para indicar se o elemento está ativo ou inativo. Segue o algoritmo modelo:

```
1:    INÍCIO percorrer(listaDeFormas, percurso)
2:    .   SE percurso=+
3:    .   .   SE listaDeFormas estiver vazia não faça nada
```

```
4:    .   .   SENÃO
5:    .   .       PARA listaDeFormas FAÇA
6:    .   .       .   SE elemento atual estiver ativo SAIA
7:    .   .       .   SENÃO analise o próximo elemento
8:    .   .       FIM PARA
9:    .   .       SE o elemento atual for o último da lista
10:   .   .       .   elemento atual é desativado
11:   .   .       .   primeiro elemento é ativado
12:   .   .        SENÃO
13:   .   .            elemento atual é desativado
14:   .   .            próximo elemento é ativado
15:   .   SENÃO
16:   .       SE listaDeFormas estiver vazia não faça nada
17:   .       SENÃO
18:   .           PARA listaDeFormas FAÇA
19:   .           .  SE elemento atual estiver ativo SAIA
20:   .           .  SENÃO analise o próximo elemento
21:   .           FIM PARA
22:   .           SE o elemento atual for o primeiro da lista
23:   .           .   elemento atual é desativado
24:   .           .   último elemento é ativado
25:   .           SENÃO
26:   .               elemento atual é desativado
27:   .               elemento anterior é ativado
28:   FIM percorrer
```

Existe um padrão comum tanto para o percurso ascendente quanto para o descendente, no que diz respeito à verificação de erros oriundos do usuário. O primeiro comando do algoritmo, é verificar se a lista encontra-se vazia (linhas 3 e 16), para o caso de o usuário erroneamente ter acionado a função percorrer, sem nenhum elemento previamente adicionado à lista. Caso o referido erro seja confirmado, a função é imediatamente encerrada. Caso contrário (existindo elementos na lista), uma estrutura de repetição é acionada (linhas 5 e 18) para que o programa encontre qual elemento encontra-se atualmente ativo na lista. Tal

verificação deve ser executada pois sempre que a função é acionada, ela recebe como um dos parâmetros de entrada, a listaDeFormas completa. Assim sendo, ela deve percorrer a lista em busca do elemento ativo, para que possa ter acesso aos elementos vizinhos e ativar o anterior ou o próximo, dependendo do percurso escolhido pelo usuário.

Por fim, a função desativa o elemento atual e ativa o próximo, (linhas de 9 a 14 e de 22 a 27) fazendo a verificação de início e final da lista para que não ocorram falhas de segmentação de memória. Se, por exemplo, a lista contenha cinco elementos e o elemento ativo seja o 5º, caso o usuário escolha o percurso ascendente, o próximo elemento a ser ativado será o 1º, pois não existe um 6º elemento na lista. O mesmo ocorre, no caso do elemento ativo ser o primeiro e o usuário ativar o percurso descendente. Imediatamente o primeiro elemento é desativado e o quinto fica ativo, pois não existem elementos incluídos antes do primeiro.

É de suma importância que estes tratamentos de falhas de segmentação sejam feitos, pois linguagens como C e C++ não apresentam nenhuma mensagem de erro ao modificar uma posição de memória não reservada (HORSTMANN, 2004) e tal posição de memória pode estar sendo utilizada por outro programa ou até mesmo pelo sistema operacional.

4.2.4 Ampliando e Reduzindo Formas Geométricas

A função que amplia e reduz formas geométricas utiliza-se do conceito matemático da matriz de ampliação aplicada ao vetor:

$$\begin{pmatrix} K & 0 & 0 \\ 0 & K & 0 \\ 0 & 0 & K \end{pmatrix} x \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} KX+0Y+0Z \\ 0X+KY+0Z \\ 0X+0Y+KZ \end{pmatrix} = \begin{pmatrix} KX \\ KY \\ KZ \end{pmatrix}$$

onde, X, Y e Z, são as coordenadas do vetor; KX, KY e KZ, são as coordenadas do vetor com a devida transformação e K é a constante multiplicativa utilizada para definir o quanto o vetor será modificado. A constante K deve sempre possuir um valor positivo diferente de 1 ($K \in R \mid 0<K<1 \, ou \, K>1$), pois a constante K=1 não confere nenhuma transformação no vetor. A diferenciação entre a ampliação e redução de formas geométricas, também é definida por K, onde a transformação de redução T^1 tem a constante maior que zero e menor que 1

$(0<K<1)$ e a transformação de ampliação T² tem a constante maior que 1$(K>1)$. Esta transformação é aplicada a todos os vértices da forma geométrica selecionada pelo usuário. Tal modelo de transformação foi escolhido, em detrimento da simples incrementação/decrementação dos vetores, pois a constante decrementação das coordenadas de uma forma geométrica pode deixá-la deformada de forma desproporcional ou até mesmo zerar todos os seus valores deixando-a com a forma de um ponto na origem(0,0). Com a tranformação utilizando a constante multiplicativa, as coordenadas diferentes de 0 nunca ficarão zeradas. Ou seja, a transformação T¹ terá:

$$\lim_{T^1 \to 0} \text{ e } T^2, \lim_{T^2 \to \infty} .$$

Pseudcódigo da função de escalonamento:

```
1:   INÍCIO escalonar(listaDeFormas, sentido)
2:   .    PARA listaDeFormas FAÇA
3:   .    .   SE elemento atual estiver ativo SAIA
4:   .    .   SENÃO analise o próximo elemento
5:   .    FIM PARA
6:   .    SE sentido = reduzir
7:   .    .    PARA listaDeVertices FAÇA
8:   .    .    .    vertice.X<-vertice.X*(0,99)
9:   .    .    .    vertice.Y<-vertice.Y*(0,99)
10:  .    .    .    vertice.Z<-vertice.Z*(0,99)
11:  .    .    .    SE vertice.proximo=vazio SAIA
12:  .    .    .    SENÃO reduza o próximo vértice
13:  .    .    FIM PARA
14:  .    SENÃO
15:  .         PARA listaDeVertices FAÇA
16:  .         .    vertice.X<-vertice.X*(1,01)
17:  .         .    vertice.Y<-vertice.Y*(1,01)
18:  .         .    vertice.Z<-vertice.Z*(1,01)
19:  .         .    SE vertice.proximo=vazio SAIA
```

```
20:   .          .     SENÃO amplie o próximo vértice
21:   .          FIM PARA
22:   FIM escalonar
```

A primeira estrutura de repetição na linha 2, faz a verificação da listaDeFormas para selecionar a forma ativa da lista. As estruturas de repetições seguintes (linhas 7 e 15) percorre a lista de vértices da forma selecionada aplicando a devida transformação (redução na linha 7 e ampliação na linha 15).

Figura 11: Ampliando e reduzindo formas

4.2.5 Pivô de Orientação

O pivô de orientação é um ponto referencial gráfico que representa a exata localização dos objetos no espaço. Todas as transformações em um objeto são relativas ao pivô de orientação (AUTODESK, 2015). Geralmente é representado por um conjunto de três setas, com suas bases na origem cartesiana, onde cada seta aponta na direção de um dos 3 eixos ortogonais (X, Y e Z) e tem como fim, orientar o usuário de forma que o mesmo não perca a referência de perspectiva enquanto manipula alguma forma geométrica. Exemplo na figura 4 da seção 3.3.

Em nosso sistema, o pivô de orientação possui eixos que podem ser ativados e desativados de forma independente, com a finalidade de definir as direções escolhidas pelo usuário no momento em que ele realize uma translação em alguma forma selecionada. Por se tratar de um procedimento simples, não se faz necessário a criação de uma função específica para o acionamento dos eixos do pivô. Tal acionamento pode ser feito dentro da função que ministra os comandos do teclado, utilizando três variáveis booleanas como interruptores.

4.2.6 Transladando Formas

Transladar objetos é um dos recursos mais utilizados em softwares de manipulação 3D. É através desse recurso, que o usuário posiciona os objetos em um determinado espaço, aperfeiçoando ao seu gosto a composição de cena. Como referência universal, os objetos devem ser incluídos próximo ao pivô de orientação e à partir daí remanejados pelo usuário, tomando como referência os eixos do pivô que estão ativos ou não.

A função de transladação, funciona incrementando as coordenadas dos vértices da forma selecionada, referente aos eixos do pivô que estão ativos:

```
1:   INÍCIO transladar(listaDeFormas, sentido)
2:   .    PARA listaDeFormas FAÇA
3:   .    .   SE elemento atual estiver ativo SAIA
4:   .    .   SENÃO analise o próximo elemento
5:   .    FIM PARA
6:   .    SE sentido for ascendente
7:   .    .    PARA listaDeVertices FAÇA
8:   .    .    .    vertice.X<-vertice.X+(2*eixoX)
9:   .    .    .    vertice.Y<-vertice.Y+(2*eixoY)
10:  .    .    .    vertice.Z<-vertice.Z+(2*eixoZ)
11:  .    .    .    SE vertice.proximo=vazio SAIA
12:  .    .    .    SENÃO vertice<-vertice.proximo
13:  .    .    FIM PARA
14:  .    SENÃO
15:  .         PARA listaDeVertices FAÇA
16:  .         .    vertice.X<-vertice.X-(2*eixoX)
17:  .         .    vertice.Y<-vertice.Y-(2*eixoY)
18:  .         .    vertice.Z<-vertice.Z-(2*eixoZ)
19:  .         .    SE vertice.proximo=vazio SAIA
20:  .         .    SENÃO vertice<-vertice.proximo
21:  .         FIM PARA
22:  FIM transladar
```

Após a seleção da forma ativa da listaDeFormas (linha 2), o código incrementa todos os vértices da forma selecionada de forma tal, que apenas os vértices relativos aos eixos ativos do pivô de orientação, recebam um novo valor. Se por exemplo, no momento de transladação ascendente de um ponto P(5;5;5), apenas os eixos X e Z encontram-se ativos, a função $f(x)=x+2y$(onde yé o valor da variável booleana do eixo referente) será aplicada à todas as coordenadas, mas apenas as coordenadas X e Z sofrerão alteração:

$$P(5;5;5), eixoX=1, eixoY=0\, e\, eixoZ=1$$
$$f(X)=X+2eixoX \Rightarrow f(X)=5+2.1 \Rightarrow f(X)=5+2 \Rightarrow f(X)=7$$
$$f(Y)=Y+2eixoY \Rightarrow f(Y)=5+2.0 \Rightarrow f(Y)=5+0 \Rightarrow f(Y)=5$$
$$f(Z)=Z+2eixoZ \Rightarrow f(Z)=5+2.1 \Rightarrow f(Z)=5+2 \Rightarrow f(Z)=7$$

assim, após a transladação o ponto P(5;5;5) ficou com as coordenadas P(7;5;7). Em formas com mais vértices a fórmula é aplicada em todos eles. Caso a transladação seja no sentido contrário, os vértices são decrementados com a fórmula

$$f(x)=x-2y.$$

Após cada alteração de coordenada, o programa verifica se ainda existem vértices naquela lista (linhas 11 e 19) para que não ocorra nenhum problema decorrente de falha de segmentação de memória.

Figura 12: Transladando formas

4.2.7 Desenhando

Assim como a função que ministra os comandos do teclado, a função responsável por desenhar sobre o marcador AR, também funciona como uma ponte entre várias outras funções. Ela recebe a listaDeFormas como entrada e realiza a chamada das funções responsáveis por aplicar os códigos OpenGL em cada tipo de forma selecionada:

```
1:   INÍCIO desenha(listaDeFormas)
2:   .    SE elemento!= vazio
3:   .         SE elemento=ponto
4:   .              desenhaPonto(elemento atual)
5:   .         SENÃO SE elemento=linha
6:   .              desenhaLinha(elemento atual)
7:   .         SENÃO SE elemento=triangulo
8:   .              desenhaTriangulo(elemento atual)
9:   .         SENÃO SE elemento=quadro
10:  .              desenhaQuadro(elemento atual)
11:  .         SENÃO SE elemento=cubo
12:  .              desenhaCubo(elemento atual)
13:  .         desenha(listaDeFormas.próximoElemento)
14:  FIM desenha
```

Existe aqui, uma série de verificações composta por `SE` e `SENÃO SE` alinhados. Diferente das funções anteriores, que faziam as verificações para evitar escrita em endereços não-reservados da memória, as verificações aqui feitas servem para otimizar o código, de forma tal que o programa não perca processamento com verificações inúteis, pois é nesta fase que a performance do software é definida.

Logo na segunda linha, o algoritmo verifica se existem elementos na lista, pois caso não tenha, a função é imediatamente finalizada. Existindo elementos, a função só realizará verificações até encontrar a chamada de função necessária. Se, por exemplo, o atual elemento for uma linha, ele verificará se o elemento é um ponto

(linha 3) e em seguida verificará se o elemento é uma linha (linha 5). Como o elemento atual é uma linha, ele faz a chamada de função correspondente e pula todas as verificações seguintes, agilizando assim o processo. No final da execução ele faz uma chamada recursiva da função desenhar, com o próximo elemento da listaDeFormas (linha 13).

4.2.7.1 Desenhando Pontos

É à partir daqui, que as funções utilizarão as variáveis presentes nas listaDeVertices dentro do código OpenGL, aplicando assim o conceito de dinamicidade do código. Todas as funções presentes nesta seção 5.2.7 foram chamadas pela função `desenha`. Recebem como entrada, os elementos da listaDeFormas e aplicam as variáveis no código OPenGL.

```
1:   INÍCIO desenhaPonto(listaDeFormas)
2:   .    glBegin(GL_POINTS);
3:   .    glColor3f(0+(0.5*listaDeFormas.ativo),
               1+(0.5*listaDeFormas.ativo),
               0+(0.5*listaDeFormas.ativo));
4:   .    glVertex3f(listaDeFormas.vertices.x,
               listaDeFormas.vertices.y,
               listaDeFormas.vertices.z);
5:   .    glEnd();
6:   FIM desenhaPonto
```

Esta função desenha um ponto verde na tela. Pode-se observar, que o comando OpenGL que define a cor verde do objeto `glColor3f(0,1,0);` encontra-se aqui com uma configuração diferente: `glColor3f(0+(0.5*listaDeFormas.ativo),1+(0.5*listaDeFormas.ativo),0+(0.5*listaDeFormas.ativo));`

Existe uma fórmula aplicada a cada valor RGB do comando, definida por $f(x) = x + (0{,}5 * y)$, onde y é a variável booleana que indica se o elemento da lista está ativo ou não. Vejamos a aplicação da fórmula em elemento ativo:

$$f(R) = 0 + (0{,}5 * 1) \Rightarrow f(R) = 0 + (0{,}5) \Rightarrow f(R) = 0{,}5$$
$$f(G) = 1 + (0{,}5 * 1) \Rightarrow f(G) = 1 + (0{,}5) \Rightarrow f(G) = 1{,}5$$
$$f(B) = 0 + (0{,}5 * 1) \Rightarrow f(B) = 0 + (0{,}5) \Rightarrow f(B) = 0{,}5$$

Resultado aplicado ao código: `glColor3f(0.5,1.5,0.5);`

Aplicação da fórmula em um elemento desativado:

$$f(R) = 0 + (0{,}5 * 0) \Rightarrow f(R) = 0 + 0 \Rightarrow f(R) = 0$$
$$f(G) = 1 + (0{,}5 * 0) \Rightarrow f(G) = 1 + 0 \Rightarrow f(G) = 1$$
$$f(B) = 0 + (0{,}5 * 0) \Rightarrow f(B) = 0 + 0 \Rightarrow f(B) = 0$$

Resultado aplicado ao código: `glColor3f(0,1,0);`

Nota-se, que as cores que representam um elemento ativo, possuem uma tonalidade mais clara. É esta diferença de tonalidade, que vai permitir ao usuário, identificar visualmente qual forma geométrica encontra-se ativa e assim facilitar a manipulação da referida forma.

Figura 13: Selecionando diferentes formas

4.2.7.2 Desenhando Outras Formas

O escopo utilizado para desenhar um ponto na tela, pode também ser utilizado para outras formas. À partir daí o número de vértices da forma geométrica vai definir as variáveis presentes no código OpenGL, em cada algoritmo. Tomemos por exemplo o algoritmo que desenha um quadrado:

```
1:   INÍCIO desenhaQuadro(listaDeFormas)
2:   .    glBegin(GL_QUADS);
3:   .    glColor3f(0+(0.5*listaDeFormas.ativo),
                0+(0.5*listaDeFormas.ativo),
                1+(0.5*listaDeFormas.ativo));
4:   .    glVertex3f(listaDeFormas.vertices.x,
                listaDeFormas.vertices.y,
                listaDeFormas.vertices.z);
5:   .    glVertex3f(listaDeFormas.vertices.proximo.x,
                listaDeFormas.vertices.proximo.y,
                listaDeFormas.vertices.proximo.z);
6:   .    glVertex3f(listaDeFormas.vertices.proximo.proximo.x,
                listaDeFormas.vertices.proximo.proximo.y,
                listaDeFormas.vertices.proximo.proximo.z);
7:   .glVertex3f(listaDeFormas.vertices.proximo.proximo.pro
                ximo.x,listaDeFormas.vertices.proximo.proximo.p
          roximo.y,listaDeFormas.vertices.proximo.proximo.prox
          imo.z);
8:   .    glEnd();
9:   FIM desenhaQuadro
```

A principal diferença entre o primeiro e o segundo algoritmo, encontra-se no número de vértices utilizados. No código acima foi acrescido mais três vértices (linhas 5, 6 e 7), e eles saem percorrendo a listaDeVertices de cada elemento, para incluir seu valores no código.

Todos os códigos descritos neste capítulo 5, foram implementados na linguagem C e encontram-se disponíveis no Apêndice I deste trabalho. O código completo assim como o programa compilado, encontra-se na internet, no link descrito no Apêndice II.

5 CONSIDERAÇÕES FINAIS

5.1 CONCLUSÕES

Após implementar o sistema, nota-se um código muito complexo, pois a linguagem C é provida de instruções detalhadas de ponteiros e manipulação de endereços de memória. No que diz respeito ao desempenho, o sistema apresentou alta performance, pois as formas geométricas são criadas e modificadas instantaneamente, sem nenhum atraso entre o comando e a informação visual, e a movimentação do marcador AR, em um ambiente bem iluminado, não confere nenhum distúrbio na identificação das marcas pelo ARToolKit, fazendo com que o sistema tenha respostas em tempo real.

Avalia-se através da plataforma implementada, que o objetivo do trabalho foi atingido. Onde antes um usuário precisaria ter conhecimentos avançados de programação, com a dinamização do OpenGL nesta plataforma tais conhecimentos não se fazem mais necessários. O sistema ainda encontra-se aquém de todas as necessidades dos profissionais que trabalham com modelagem 3D, pois o mesmo não apresenta ainda, manipulação de malhas, tratamento de partículas, independência de vértices, textura e renderização, mas fundamenta a base de futuros aprimoramentos baseado nas necessidades dos usuários.

Vale salientar ainda, como consequência positiva do trabalho, que o método de manipulação tridimensional com as próprias mãos, assemelha-se à manipulação de formas na Realidade Virtual com luvas 3D e óculos de visão estereoscópica, conforme pode ser visto na figura 14:

Figura 14: Luvas e óculos de imersão

Fonte: Site explainthatstuff [5]

Porém, luvas 3D e óculos de visão estereoscópica consistem em equipamentos de alto custo financeiro, sendo assim pouco acessíveis. O uso da edição 3D dentro da Realidade Aumentada confere o mesmo manuseio imersivo com baixo custo, tendo em vista que, além de um computador com configurações básicas, o usuário só precisa de um marcador AR impresso, que tem um custo de produção ínfimo. Isto vem a ser um benefício para universidades e estudantes com poucos recursos financeiros.

5.2 TRABALHOS FUTUROS

Conforme citado no subcapítulo anterior, o sistema ainda não está dotado de manipulação de malhas, tratamento de partículas, independência de vértices, textura ou renderização. Tais recursos podem ser adicionados em futuras implementações, o que podem ser consideradas melhorias em um trabalho futuro.

A manipulação de arquivos também pode ser acrescida ao sistema e assim, seria possível a exportação do código OpenGL para ser usado em estudos, ou implementação de outros softwares. Com a manipulação de arquivos, também seria possível a exportação das figuras criadas para um formato de arquivo nativo de outro software CAD.

[5] Disponível em <http://www.explainthatstuff.com/virtualreality.html>. Acesso em mai. 2015.

6 REFERÊNCIAS

AUTODESK. Getting Started with Maya: Pivot points. Disponível em <http://download.autodesk.com/us/maya/Maya_2014_GettingStarted/index.html?url=files/Viewing_the_Maya_3D_scene_Pivot_points.htm,topicNumber=d30e5131>. Acesso em: 10 de Nov. 2015.

AZUMA, R. T. - **A survey of augmented reality**. *Teleoperators and Virtual Environments 6,* Malibu, v.4, p.355-385, 1997.

BATTISTI, J. **Linguagem C - Alocação Dinâmica**. Disponível em: <http://juliobattisti.com.br/tutoriais/katiaduarte/cbasico009.asp>. Acesso em 05 de fev. 2015.

BENTO, E. J. - **Desenvolvimento Web com PHP e MySQL**. Disponível em: <http://www.casadocodigo.com.br/pages/sumario-php-mysql>. em Acesso em 25 de mai. de 2015.

FERREIRA, E. C. - **Sistemas de conversão AD e DA**. Disponível em: <http://www.demic.fee.unicamp.br/~elnatan/ee610/18a%20Aula.pdf>. Acesso em 25 de mai. 2015.

HAUTSCH, O. - **Como funciona a Realidade Aumentada**. 2009. Disponível em: <http://www.tecmundo.com.br/realidade-aumentada/2124-como-funciona-a-realidade-aumentada.htm>. Acesso em 25 de mai. 2015.

HITLAB - Human Interface Technology Laboratory. **ARToolKit Documentation: development principles**. Disponível em: <http://www.hitl.washington.edu/artoolkit/documentation/devprinciple.htm>. Acesso em: 14 de jul. 2014c.

HITLAB - Human Interface Technology Laboratory. **ARToolKit Documentation: history**. Disponível em: <http://www.hitl.washington.edu/artoolkit/documentation/history.htm>. Acesso em: 11 de jul. 2014a.

HITLAB - Human Interface Technology Laboratory. **ARToolKit Documentation: how does ARToolKit work?**. Disponível em: <http://www.hitl.washington.edu/artoolkit/ documentation/userarwork.htm>. Acesso em: 15 de jul. 2014d.

HITLAB - Human Interface Technology Laboratory. **ARToolKit Documentation: introduction**. Disponível em: <http://www.hitl.washington.edu/artoolkit/documentation/ userintro.htm>. Acesso em: 11 de jul. 2014b.

HITLAB - Human Interface Technology Laboratory. **ARToolKit Documentation: Setting up ARToolkit**. Disponível em: <http://www.hitl.washington.edu/artoolkit/documentation/usersetup.htm>. Acesso em: 20 de Abr. 2015.

HOSTMANN, Cay. Arrays e Lista de Arrays. **Big Java**. Porto Alegre: Bookman, 2004. P. 499.

IME-USP - Instituto de Matemática e Estatística da Universidade de São Paulo. **Projeto de Algorítmos**: Listas encadeadas. Disponível em: <http://www.ime.usp.br/~pf/algoritmos/aulas/lista.html>. Acesso em: 10 de fev. 2015.

JOBS, S. - **Thoughts On Flash**. Disponível em: <http://www.apple.com/hotnews/thoughts-on-flash/>. Acesso em 09 de Nov. de 2015.

KIRNER, C. . **Evolução da Realidade Virtual no Brasil**. In: X Symposium on Virtual and Augmented Reality, 2008, João Pessoa. Proceedings of the X Symposium on Virtual and Augmented Reality. Porto Alegre : SBC, 2008. v. 1. p. 1-11

KIRNER, C. ; TORI, R. (2004) - **Introdução à Realidade Virtual, Realidade Misturada e Hiper-realidade**. In: Kirner, C.; Tori, R. (Org.). Realidade Virtual: Conceitos, Tecnologia e Tendências. 1 ed. São Paulo: Editora SENAC, 2004, v. 1, p. 3-20.

MANSOUR, I. B. - **Introdução à OpenGL**. Disponível em: <https://www.inf.pucrs.br/~manssour/OpenGL/Introducao.html>. Acesso em 25 de mai. de 2015.

MENDONÇA, V. G. - **Conhecendo a OpenGL: história do padrão opengl**. Disponível em:<http://pontov.com.br/site/opengl/150-conhecendo-a-opengl>. Acesso em 25 de mai. de 2015.

MORAES, C. - **Computação Gráfica Para Todos**. em 31 de out. 2008. Disponível em: <http://www.hardware.com.br/artigos/computacao-grafica-iniciantes/>. Acesso em: 25 de mai. 2015.

SGI - Silicon Graphics International. **OpenGL**. Disponível em:<http://www.sgi.com/tech/opengl/>. Acesso em: 25 de mai. 2015.

SHREINER, D. - ***OpenGL Programming Guide***.7º ed. Michigan: Pearson, 2009. 885 p.

TRUYENQUE, M. A. Q. *Uma Aplicação de Visão Computacional que Utiliza Gestos da Mão para interagir com o Computador*. Rio de Janeiro, 2005. 100p. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro.

APÊNDICE I - PSEUDOCÓDIGOS CITADOS NO TRABALHO, IMPLEMENTADOS EM LINGUAGEM C

FUNÇÃO TECLADO

```
void keyboard(unsigned char key, int x, int y) {
    switch (key) {
        case 27:
            exit(0);
            break;
        case '8':
            adicionarCubo(&listaDeFormas);
            break;
        case '1':
            adicionarPonto(&listaDeFormas);
            break;
        case '2':
            adicionarLinha(&listaDeFormas);
            break;
        case '3':
            adicionarTriangulo(&listaDeFormas);
            break;
        case '4':
            adicionarQuadrado(&listaDeFormas);
            break;
        case 's':
            escalonar(&listaDeFormas, 1);
            break;
        case 'S':
            escalonar(&listaDeFormas, 0);
            break;
        case 't':
            transladar(&listaDeFormas, 1);
            break;
        case 'T':
            transladar(&listaDeFormas, 0);
            break;
        case '+':
            percorrer(&listaDeFormas, 1);
            break;
        case '-':
            percorrer(&listaDeFormas, 0);
            break;
        case 'x':
        case 'X':
            if (xativo == 0) xativo = 1;
            else xativo = 0;
            break;
        case 'y':
        case 'Y':
            if (yativo == 0) yativo = 1;
            else yativo = 0;
            break;
        case 'z':
        case 'Z':
            if (zativo == 0) zativo = 1;
            else zativo = 0;
            break;
```

```
    }
}
```

FUNÇÃO ADICIONAR LINHA

```
void adicionarLinha(forma *listaDeFormas) {
      if(listaDeFormas->proximo==NULL){
            listaDeFormas->proximo=(forma*)malloc(sizeof(forma));
            listaDeFormas->proximo->tipo=2;
            listaDeFormas->proximo->ativo=0;
            listaDeFormas->proximo->anterior=listaDeFormas;
            listaDeFormas->proximo->proximo=NULL;
            listaDeFormas->proximo->vertices=(vertice*)malloc(sizeof(ver
                  tice));
            listaDeFormas->proximo->vertices->x=0;
            listaDeFormas->proximo->vertices->y=0;
            listaDeFormas->proximo->vertices->z=0;
            listaDeFormas->proximo->vertices->proximo=(vertice*)malloc(si
                  zeof(vertice));;
            listaDeFormas->proximo->vertices->proximo->x=20;
            listaDeFormas->proximo->vertices->proximo->y=20;
            listaDeFormas->proximo->vertices->proximo->z=20;
            listaDeFormas->proximo->vertices->proximo->proximo=NULL;
      }
      else adicionarLinha(listaDeFormas->proximo);
}
```

FUNÇÃO PERCORRER

```
void percorrer(forma *listaDeFormas, int percurso) {
    forma *listaSubstituta, *listaSubstituta2;
    if(percurso==1){
        if (listaDeFormas->proximo==NULL) { } //Se a lista estiver vazia(só
                                  com o primeiro elemento), não faça nada.
        else{
            for(listaSubstituta=listaDeFormas;;){//procurar o elemento
                                                      ativo da lista
                if(listaSubstituta->ativo==1)break;
                else listaSubstituta=listaSubstituta->proximo;
            }
            if(listaSubstituta->proximo==NULL){
                listaSubstituta->ativo=0;
                listaDeFormas->ativo=1;
            }
            else{
                listaSubstituta->ativo=0;
                listaSubstituta->proximo->ativo=1;
            }
       }
    }
    if(percurso==0){
        if (listaDeFormas->proximo==NULL) { }
        else{
            for(listaSubstituta=listaDeFormas;;){

                if(listaSubstituta->ativo==1) break;
                else listaSubstituta=listaSubstituta->proximo;
            }
            if(listaSubstituta->anterior==NULL){
                for(listaSubstituta2=listaDeFormas;;){ //encontrar final
                    if(listaSubstituta2->proximo==NULL) break;
                    else listaSubstituta2=listaSubstituta2->proximo;
```

```
                }
                listaSubstituta2->ativo=1;
                listaDeFormas->ativo=0;
            }
            else{
                listaSubstituta->ativo=0;
                listaSubstituta->anterior->ativo=1;
            }
        }
    }
}
```

FUNÇÃO ESCALONAR

```
void escalonar(forma *listaDeFormas, int sentido){
    forma *listaSubstituta;
    vertice *verticeSubstituto;
    for(listaSubstituta=listaDeFormas;;){
        if(listaSubstituta->ativo==1) break;
        else listaSubstituta=listaSubstituta->proximo;
    }
    if(sentido==1){
        for(verticeSubstituto=listaSubstituta->vertices;;){
            verticeSubstituto->x=verticeSubstituto->x*(0.99);
            verticeSubstituto->y=verticeSubstituto->y*(0.99);
            verticeSubstituto->z=verticeSubstituto->z*(0.99);
            if(verticeSubstituto->proximo==NULL) break;
            else verticeSubstituto=verticeSubstituto->proximo;
        }
    }
    if(sentido==0){
        for(verticeSubstituto=listaSubstituta->vertices;;){
            verticeSubstituto->x=verticeSubstituto->x*(1.01);
            verticeSubstituto->y=verticeSubstituto->y*(1.01);
            verticeSubstituto->z=verticeSubstituto->z*(1.01);
            if(verticeSubstituto->proximo==NULL) break;
            else verticeSubstituto=verticeSubstituto->proximo;
        }
    }
}
```

FUNÇÃO TRANSLADAR

```
void transladar(forma *listaDeFormas, int sentido){
    forma *listaSubstituta;
    vertice *verticeSubstituto;
    for(listaSubstituta=listaDeFormas;;){
        if(listaSubstituta->ativo==1) break;
        else listaSubstituta=listaSubstituta->proximo;
    }
    if(sentido==1){
        for(verticeSubstituto=listaSubstituta->vertices;;){
            verticeSubstituto->x=verticeSubstituto->x+(2.0*xativo);
            verticeSubstituto->y=verticeSubstituto->y+(2.0*yativo);
            verticeSubstituto->z=verticeSubstituto->z+(2.0*zativo);
            if(verticeSubstituto->proximo==NULL) break;
            else verticeSubstituto=verticeSubstituto->proximo;
        }
    }
    if(sentido==0){
        for(verticeSubstituto=listaSubstituta->vertices;;){
```

```
            verticeSubstituto->x=verticeSubstituto->x-(2.0*xativo);
            verticeSubstituto->y=verticeSubstituto->y-(2.0*yativo);
            verticeSubstituto->z=verticeSubstituto->z-(2.0*zativo);
            if(verticeSubstituto->proximo==NULL) break;
            else verticeSubstituto=verticeSubstituto->proximo;
        }
    }
}
```

FUNÇÃO DESENHAR

```
void desenha(forma *listaDeFormas) {
    if(listaDeFormas->proximo!=NULL){
        if(listaDeFormas->proximo->tipo==8) desenhaCubo(listaDeFormas);
        else if (listaDeFormas->proximo->tipo==1) desenhaPonto(listaDeForma
                                                                    s);
        else if (listaDeFormas->proximo->tipo==2) desenhaLinha(listaDeForma
                                                                    s);
        else if (listaDeFormas->proximo->tipo==3) desenhaTriangulo(listaDeF
                                                                  ormas);
        else if (listaDeFormas->proximo->tipo==4)desenhaQuadrado(listaDeFor
                                                                   mas);
        desenha(listaDeFormas->proximo);
    }
}
```

FUNÇÃO DESENHAR PONTO

```
void desenhaPonto(forma *listaDeFormas) {
    glBegin(GL_POINTS);
    glColor3f(0+(0.5*listaDeFormas->proximo->ativo), 1+(0.5*listaDeFormas->
           proximo->ativo), 0+(0.5*listaDeFormas->proximo->ativo)); //Verde
    glVertex3f(listaDeFormas->proximo->vertices->x, listaDeFormas->proximo-
                        >vertices->y, listaDeFormas->proximo->vertices->z);
    glEnd();
}
```

FUNÇÃO DESENHAR QUADRO

```
void desenhaQuadrado(forma *listaDeFormas) {
    glBegin(GL_QUADS);
    glColor3f(0+(0.5*listaDeFormas->proximo->ativo), 0+(0.5*listaDeFormas->
            proximo->ativo), 1+(0.5*listaDeFormas->proximo->ativo)); //Azul
    glVertex3f(listaDeFormas->proximo->vertices->x, listaDeFormas->proximo-
        >vertices->y, listaDeFormas->proximo->vertices->z);
    glVertex3f(listaDeFormas->proximo->vertices->proximo->x, listaDeFormas-
        >proximo->vertices->proximo->y,listaDeFormas->proximo->vertices->
        proximo->z);
    glVertex3f(listaDeFormas->proximo->vertices->proximo->proximo->x, lista
        DeFormas->proximo->vertices->proximo->proximo->y,listaDeFormas->pro
        ximo->vertices->proximo->proximo->z);
    glVertex3f(listaDeFormas->proximo->vertices->proximo->proximo->proximo-
        >x, listaDeFormas->proximo->vertices->proximo->proximo->proximo->y,
        listaDeFormas->proximo->vertices->proximo->proximo->proximo->z);
    glEnd();
}
```

APÊNDICE II – LINK COM CÓDIGO FONTE COMPLETO, ARQUIVOS DE COMPILAÇÃO E EXECUTÁVEIS.

https://drive.google.com/folderview?id=0B-XjI7XufUKJZERzaWFMYXZHVlU

ANEXO – MARCADOR PADRÃO UTILIZADO PELO SISTEMA