Centro de Informática UFPE

Universidade Federal de Pernambuco
Graduação em Ciência da Computação

HARDWIRE: um módulo em *hardware* para a visualização em *wireframe* de objetos tridimensionais

Trabalho de Graduação

Aluno: João Marcelo Xavier Natário Teixeira
Orientadora: Prof. Judith Kelner
Co-orientadora: Prof. Veronica Teichrieb

Recife, outubro de 2006.

Resumo

Desde seu surgimento, há 16 anos atrás, FPGAs têm ganho bastante espaço na área de *hardware* reprogramável. À medida que são aperfeiçoados, eles se tornam mais velozes, menores e poderosos. FPGAs estão sendo cada vez mais utilizados na área de renderização gráfica 3D, através do manuseio de algoritmos complexos em *hardware* reconfigurável e de baixo custo. Um exemplo que comprova esse uso está no desenvolvimento de placas de processamento de comportamentos físicos, para auxiliar a CPU em simulações 3D. O desenvolvimento desse tipo de *hardware* certamente passa pela fase de prototipação em FPGA, e posterior migração para uma placa fechada e definitiva. Por outro lado, grupos de pesquisa em Realidade Aumentada atuam fortemente no desenvolvimento de aplicações que combinam ambos aspectos do mundo virtual e do real, e que exigem processamento em tempo real. Este Trabalho de Graduação propõe a implementação de um modelo sintetizável de um sistema de renderização 3D em *wireframe*, denominado *Hardwire*, como parte de um sistema maior de Realidade Aumentada embarcada. Esse desenvolvimento é possível através do uso de uma linguagem de descrição de *hardware* em conjunto com algumas técnicas de visualização 3D.

Agradecimentos

```
library LIFE, WORK;
use LIFE.time;
use WORK.knowledge;

entity JM is
	port(
		food : inout STD_LOGIC
	);
end JM;

architecture everything of JM is
	signal God : GOD;
	signal Father, Mother, relatives : family;
	signal Alline : LIFE.love.girlfriend;
	signal GRVM, GPRT : group;
	signal Veronica, Judith, Jamel : LIFE.guiders;
	signal Edna, Manoel : LIFE.professors;

	variable money : Real;

begin
	process(food)
	begin
		-- wait until LIFE.ends;
	end process;

end everything;
```

O tempo necessário para o Quartus sintetizar esse código é completamente não-determinístico.

Índice

1. INTRODUÇÃO **7**

2. CONTEXTO **8**

2.1. REALIDADE AUMENTADA 8
2.1.1. Aplicações de Realidade Aumentada *8*
2.1.2. Dispositivos de Realidade Aumentada *13*
2.2. SISTEMAS EMBARCADOS 15
2.2.1. FPGAs *15*
2.3. O PROJETO ARCAM 18
2.4. TRABALHOS RELACIONADOS 19

3. CONCEITOS BÁSICOS RELACIONADOS **21**

3.1. TRANSFORMAÇÕES DE VISUALIZAÇÃO 21
3.2. ROTAÇÃO 3D 23
3.3. PROJEÇÕES 3D 24
3.4. ALGORITMO DE DESENHO DE LINHAS 25
3.5. REPRESENTAÇÃO DE OBJETOS 3D 28
3.5.1. Representação Aramada ou Por "Wireframe" *29*
3.5.2. Representação Por Faces (ou Superfícies Limitantes) *29*
3.5.3. Representação Por Enumeração de Ocupação Espacial *30*
3.5.4. Representação por Octrees e Quadtrees *31*

4. HARDWIRE **32**

4.1. A PLATAFORMA ARCAM 32
4.2. METODOLOGIA UTILIZADA 33
4.3. ARQUITETURA 34
4.4. AMBIENTE DE DESENVOLVIMENTO 35
4.4.1. Hardware Utilizado *35*
4.4.2. NIOS *37*
4.4.3. VHDL *38*
4.4.4. Java e C *38*
4.5. TIPO DE FORMATO DE DADOS 40
4.5.1. Ponto-Flutuante *40*
4.5.2. Ponto-Fixo *45*
4.6. MÓDULOS UTILIZADOS 47
4.6.1. Módulos Fornecidos *47*
4.6.2. Módulos Implementados *50*
4.7. RESULTADOS OBTIDOS 76
4.8. DIFICULDADES ENCONTRADAS 77

5. CONCLUSÕES E TRABALHOS FUTUROS **78**

6. REFERÊNCIAS **79**

Índice de Figuras

FIGURA 1. IMAGEM SOBREPOSTA AO PACIENTE. 9
FIGURA 2. ENCANAMENTO INDUSTRIAL. 9
FIGURA 3. *SIGNPOST* NO MODO *STAND-ALONE* (ESQUERDA) E *SIGNPOST* MOSTRANDO O MAPA DA ESTRUTURA DO PRÉDIO (DIREITA). 10
FIGURA 4. A PROPAGANDA *PACIFIC BELL* É VIRTUAL. 10
FIGURA 5. VISÃO DO USUÁRIO JOGANDO *ARQUAKE*. 11
FIGURA 6. DOIS PDAS RODANDO O *INVISIBLE TRAIN* (ACIMA) E A INTERFACE DO JOGO (ABAIXO). 11
FIGURA 7. APLICAÇÃO *MAGIC CUBES* USADA COMO CONTADOR DE HISTÓRIA. 12
FIGURA 8. *MAGIC TABLE* SENDO USADO POR DOIS USUÁRIOS. 12
FIGURA 9. *WORKSPACE* PARA PLANEJAMENTO URBANO. 13
FIGURA 10. EXEMPLOS DE HMDS. 13
FIGURA 11. EXEMPLOS DE *TRACKERS* USADOS EM RA E RV. 14
FIGURA 12. UM FPGA DA ALTERA COM 20.000 PORTAS LÓGICAS. 16
FIGURA 13. BLOCO LÓGICO. 17
FIGURA 14. POSIÇÕES DOS PINOS DO BLOCO LÓGICO DO FPGA. 17
FIGURA 15. MULTIPLICADOR BINÁRIO DESENVOLVIDO POR MC KEON. 20
FIGURA 16. SIMULAÇÃO DAS FUNÇÕES IMPLEMENTADAS POR MC KEON NO MODELSIM. 20
FIGURA 17. CONCATENAÇÃO DE TRANSFORMAÇÕES GEOMÉTRICAS. 21
FIGURA 18. SISTEMA DE COORDENADAS 3D. 21
FIGURA 19. REPRESENTAÇÃO DA CÂMERA VIRTUAL. 22
FIGURA 20. SISTEMA DE COORDENADAS DE TELA. 22
FIGURA 21. MODELO DE CÂMERA VIRTUAL. 22
FIGURA 22. ETAPAS DA RENDERIZAÇÃO. 23
FIGURA 23. MATRIZES DE TRANSLAÇÃO E ROTAÇÃO. 23
FIGURA 24. ROTAÇÕES SUCESSIVAS APLICADAS A UM OBJETO 3D. 24
FIGURA 25. PROPORÇÃO DAS DIMENSÕES DO OBJETO MANTIDAS NOS PLANOS DE PROJEÇÃO. 25
FIGURA 26. EXEMPLO DO RESULTADO DO ALGORITMO DE BRESENHAM. 26
FIGURA 27. EQUAÇÃO GENÉRICA DA RETA. 26
FIGURA 28. PSEUDO-CÓDIGO DE UMA IMPLEMENTAÇÃO NÃO OTIMIZADA DO ALGORITMO DE BRESENHAM. 27
FIGURA 29. VERSÃO DO ALGORITMO DE BRESENHAM QUE SUPORTA LINHAS EM QUALQUER DIREÇÃO. 27
FIGURA 30. PSEUDO-CÓDIGO DA VERSÃO OTIMIZADA DO ALGORITMO DE BRESENHAM. 28
FIGURA 31. REPRESENTAÇÃO EM ARAMADO E POR FACES POLIGONAIS. 29
FIGURA 32. AMBIGÜIDADES DA REPRESENTAÇÃO ARAMADA. 29
FIGURA 33. EXEMPLO DE OBJETO DESCRITO POR *VOXELS*. 30
FIGURA 34. EXEMPLO DE OBJETO DESCRITO POR *QUADTREES*. 31
FIGURA 35. *HARDWARE* DE DESENVOLVIMENTO (FPGA + SENSOR DE IMAGEM). 32
FIGURA 36. ARQUITETURA DO SISTEMA ARCAM. 33
FIGURA 37. METODOLOGIA UTILIZADA. 34
FIGURA 38. ARQUITETURA DO *HARDWIRE*. 35
FIGURA 39. PLACA QUE ACOMODA O SENSOR DE IMAGEM. 35
FIGURA 40. PLACA DE PROTOTIPAÇÃO. 36
FIGURA 41. BLOCO ESQUEMÁTICO DO NIOS, COM ALGUMAS DE SUAS ENTRADAS E SAÍDAS. 37
FIGURA 42. MAPEAMENTOS REALIZADOS DA LINGUAGEM JAVA PARA VHDL (IMPLEMENTAÇÃO EM *HARDWARE*). 39
FIGURA 43. GERAÇÃO DO MÓDULO *ANGLE* A PARTIR DA APLICAÇÃO EM JAVA. 39
FIGURA 44. TRADUÇÃO DO FORMATO DE PONTO-FIXO PARA VALOR CORRESPONDENTE EM DECIMAL. 40
FIGURA 45. FORMATO DE PONTO-FLUTUANTE DE 32 *BITS*. 41
FIGURA 46. FORMATO DE PONTO-FLUTUANTE DE 64 *BITS*. 42
FIGURA 47. FORMATO DE PONTO-FIXO (32 *BITS*) ADOTADO NO DESENVOLVIMENTO DO *HARDWIRE*. 46
FIGURA 48. SÍMBOLO QUE REPRESENTA O MÓDULO *ADDER32*. 47
FIGURA 49. SÍMBOLO QUE REPRESENTA O MÓDULO *MULT32*. 48
FIGURA 50. SÍMBOLO QUE REPRESENTA O MÓDULO *DIV32*. 48
FIGURA 51. SÍMBOLO QUE REPRESENTA O MÓDULO *SQRT32*. 49
FIGURA 52. SÍMBOLO QUE REPRESENTA O MÓDULO *SQUARE32*. 49
FIGURA 53. SÍMBOLO QUE REPRESENTA O MÓDULO *SUB32*. 50
FIGURA 54. SÍMBOLO QUE REPRESENTA O MÓDULO *ADDER32_INST*. 50
FIGURA 55. EXEMPLO DE USO DO MÓDULO: 1 + 2. 51

FIGURA 56. EXEMPLO DE USO DO MÓDULO: -1 + 1. 51
FIGURA 57. EXEMPLO DE USO DO MÓDULO: 4.75 + 6.25. 51
FIGURA 58. EXEMPLO DE USO DO MÓDULO: -0.75 + 10.345. 51
FIGURA 59. SÍMBOLO QUE REPRESENTA O MÓDULO *ANGLE*. 52
FIGURA 60. SÍMBOLO QUE REPRESENTA O MÓDULO *BRESENHAM*. 53
FIGURA 61. TRANSFORMAÇÃO DO FORMATO PONTO-FIXO 32 *BITS* PARA INTEIRO 18 *BITS*. 53
FIGURA 62. SÍMBOLO QUE REPRESENTA O MÓDULO *EYE2SCREEN*. 54
FIGURA 63. ESQUEMÁTICO DO MÓDULO *EYE2SCREEN*. 55
FIGURA 64. SÍMBOLO QUE REPRESENTA O MÓDULO *HARDWIRE*. 55
FIGURA 65. INTERLIGAÇÕES ENTRE *HARDWIRE* E ARCAM. 56
FIGURA 66. SÍMBOLO QUE REPRESENTA O MÓDULO *INPUT_GEN*. 57
FIGURA 67. ESTRUTURA DO CUBO 3D REPRESENTADO PELO MÓDULO *INPUT_GEN*. 57
FIGURA 68. LISTA COM AS COORDENADAS DOS VÉRTICES DO OBJETO 3D. 58
FIGURA 69. LISTA COM AS ARESTAS DO OBJETO 3D. 58
FIGURA 70. TRECHO DE CÓDIGO RESPONSÁVEL PELA ROTAÇÃO 3D. 58
FIGURA 71. SÍMBOLO QUE REPRESENTA O MÓDULO *INV32_INST*. 60
FIGURA 72. MANIPULAÇÃO DOS *BITS* REALIZADA NA OPERAÇÃO DE INVERSÃO. 60
FIGURA 73. SÍMBOLO QUE REPRESENTA O MÓDULO *JOINER*. 61
FIGURA 74. SÍMBOLO QUE REPRESENTA O MÓDULO *MULT32_INST*. 62
FIGURA 75. MANIPULAÇÃO DOS *BITS* NA MULTIPLICAÇÃO. 62
FIGURA 76. SÍMBOLO QUE REPRESENTA O MÓDULO *NORMALIZE*. 63
FIGURA 77. EQUAÇÕES PARA NORMALIZAÇÃO DE UM VETOR. 63
FIGURA 78. VETORES REPRESENTADOS NA NOTAÇÃO DE PONTO-FIXO
ANTES E DEPOIS DO PROCESSAMENTO, E SEUS RESPECTIVOS VALORES EM DECIMAL. 64
FIGURA 79. SÍMBOLO QUE REPRESENTA O MÓDULO *ORTHOGONALIZE*. 64
FIGURA 80. FÓRMULA DA ORTOGONALIZAÇÃO DE VETORES. 65
FIGURA 81. SÍMBOLO QUE REPRESENTA O MÓDULO *PRODESC*. 66
FIGURA 82. FÓRMULA DO PRODUTO ESCALAR ENTRE DOIS VETORES. 66
FIGURA 83. SÍMBOLO QUE REPRESENTA O MÓDULO *PRODK*. 67
FIGURA 84. FÓRMULA DO PRODUTO DE UM VETOR POR UMA CONSTANTE. 67
FIGURA 85. SÍMBOLO QUE REPRESENTA O MÓDULO *PRODVET*. 68
FIGURA 86. FÓRMULA DO PRODUTO VETORIAL ENTRE DOIS VETORES. 68
FIGURA 87. SÍMBOLO QUE REPRESENTA O MÓDULO *SQRT32_INST*. 69
FIGURA 88. MANIPULAÇÃO DOS *BITS* NA OPERAÇÃO DE RADICIAÇÃO. 69
FIGURA 89. EXEMPLO: RADICIAÇÃO DE 627.0016. 70
FIGURA 90. EXEMPLO: RADICIAÇÃO DE 15241.383936. 70
FIGURA 91. SÍMBOLO QUE REPRESENTA O MÓDULO *SQUARE32_INST*. 70
FIGURA 92. MANIPULAÇÃO DOS *BITS* NA OPERAÇÃO DE QUADRADO DE UM NÚMERO. 71
FIGURA 93. EXEMPLO: QUADRADO DO VALOR 25.04. 71
FIGURA 94. EXEMPLO: QUADRADO DO VALOR (-2.75). 71
FIGURA 95. SÍMBOLO QUE REPRESENTA O MÓDULO *SUB32_INST*. 72
FIGURA 96. EXEMPLO: 1 – 2. 72
FIGURA 97. EXEMPLO: -1 – 1. 72
FIGURA 98. EXEMPLO: 4.75 – 6.25. 72
FIGURA 99. EXEMPLO: -0.75 – 10.345. 73
FIGURA 100. SÍMBOLO QUE REPRESENTA O MÓDULO *WORLD2EYE*. 73
FIGURA 101. ESQUEMÁTICO DO MÓDULO *WORLD2EYE*. 74
FIGURA 102. SÍMBOLO QUE REPRESENTA O MÓDULO *BIT_RAM*. 75
FIGURA 103. PRIMEIRO OBJETO VISUALIZADO USANDO O *HARDWIRE*. 76
FIGURA 104. CUBO ANIMADO ATRAVÉS DE ROTAÇÃO. 76

1. Introdução

Com a evolução dos dispositivos embarcados e das aplicações utilizando a tecnologia de Realidade Virtual (RV), torna-se possível a criação de sistemas que amplifiquem a visão da realidade observada pelo usuário. Dá-se o nome de Realidade Aumentada (RA) aos sistemas computacionais que promovem a coexistência do mundo virtual com o real [1].

Idealmente, em RA, objetos virtuais e reais coexistem de forma natural. Em um ambiente industrial, a RA pode auxiliar na identificação de problemas e apontar soluções, desde um simples auxílio na seqüência de um procedimento a ser seguido, até a simulação de situações do futuro baseando-se em informações do presente. Por exemplo, a RA pode, através da verificação de um aumento gradual na temperatura de um equipamento, prever um incêndio e alertar o usuário. Muitas outras áreas podem utilizar (e já estão utilizando) as ferramentas de RA, tais como: medicina, linhas de produção e reparos, robótica, entretenimento [2], aplicações militares e na indexação de objetos de prateleira [3].

Propor e construir um sistema nessa linha de aplicações é o grande desafio do projeto ARCam (*Augmented Reality Camera*, descrito com mais detalhes nas Seções 2.3 e 4.1), que propõe tornar ubíquo o uso da tecnologia de RA, tornando-a disponível facilmente, de maneira mais adequada e a preços acessíveis [4]. O objetivo principal do ARCam é construir um arcabouço para desenvolvimento de soluções embarcadas para RA, criando um sistema flexível que facilite o desenvolvimento de novas aplicações através da infra-estrutura de *hardware* disponível, juntamente com uma biblioteca de funções comuns a este tipo de aplicação. A partir desse arcabouço, será possível a criação de diferentes tipos de soluções, como, por exemplo, câmeras inteligentes programadas para realizar inspeção de equipamentos, completamente desenvolvidas em *hardware*.

Reconhecer padrões e visualizar objetos tridimensionais (3D) com informações associadas ao ambiente real são alguns dos objetivos das aplicações atuais de RA [1]. Para atingir estes objetivos, são utilizadas soluções em *software* que processam dados de entrada provenientes de câmeras e/ou *trackers* e retornam para o usuário do sistema a identificação e/ou a localização dos elementos encontrados. Com base nessas informações, a aplicação exibe ao usuário uma mistura do ambiente real (geralmente a imagem capturada pela câmera) e de modelos 3D realistas.

Um grande gargalo existente ocorre devido ao processamento das informações adquiridas pela câmera, que é realizado por bibliotecas de *software*, geralmente rodando sobre um sistema operacional, em conjunto com vários outros processos. O tempo gasto nesse processamento impossibilita aplicações desse tipo em tempo real, uma vez que o retorno para o usuário é não-imediato.

O objetivo deste Trabalho de Graduação é propor um módulo de renderização em *wireframe* (aramado), totalmente implementado em *hardware*, responsável por executar a fase final do processamento das aplicações de RA, ou seja, a visualização dos objetos 3D. Este módulo, denominado *Hardwire*, em conjunto com outros capazes de reconhecer marcadores e realizar algum processamento associado (como os módulos do projeto ARCam, por exemplo), irá compor uma plataforma embarcada e independente que permitirá a utilização de aplicações de RA de tempo real.

O Capítulo 2 descreve o contexto no qual o *Hardwire* está inserido. O Capítulo 3 apresenta os principais conceitos envolvidos na implementação do módulo de renderização proposto. O Capítulo 4 detalha o projeto *Hardwire*, assim como as dificuldades que surgiram ao longo de seu desenvolvimento, e ilustra as funcionalidades do protótipo implementado. O Capítulo 5 traz algumas considerações finais sobre o trabalho e propõe uma continuidade para esse projeto.

2. Contexto

A utilização de *hardware* embarcado na área de RA tem trazido grandes benefícios não só para o desenvolvimento de novas aplicações multimídia, como também possibilita novas formas de utilização dos dispositivos embarcados existentes.

2.1. Realidade Aumentada

Grupos de pesquisa em RA atuam fortemente no desenvolvimento de aplicações que combinam ambos, aspectos do mundo virtual e do real. Uma das dificuldades impostas a essas aplicações é a necessidade de adquirir informações do ambiente físico (real), processá-las e retornar para o usuário alguma informação associada, todo esse processo ocorrendo em tempo real. Geralmente, a aquisição das características do mundo é realizada através de câmeras e outros tipos de sensores, que repassam a imagem capturada a alguma biblioteca de reconhecimento de imagens (ou bibliotecas de reconhecimento de padrões), como o ARToolkit [5], o MXRToolkit [6] e o OpenCV [7]. Em RA, é muito comum o uso de marcadores (identificadores que são capturados do mundo real e que representam padrões reconhecíveis) para conseguir se capturar a posição espacial de determinados objetos no mundo. Alguns exemplos típicos de marcadores são regiões com bordas retangulares, geralmente em preto, com uma figura central. A borda indica para a biblioteca a existência de um marcador, tornando possível determinar através de cálculos sua posição e orientação no espaço, e a figura interna é usada para diferenciá-lo. Na maior parte das aplicações de RA, ocorre uma sobreposição das imagens do mundo real por elementos virtuais inseridos artificialmente.

O tempo desde a captura da imagem do mundo real até o fim do processamento é bastante crítico, de forma que o usuário não perceba atrasos ou que a execução da aplicação seja comprometida. Para que esse requisito seja alcançado com mais eficiência, uma solução seria utilizar *hardware* dedicado para o processamento das informações do ambiente. Uma vez que grande parte das bibliotecas de reconhecimento de padrões, assim como outras mais genéricas de RA, são implementadas exclusivamente em *software*, esse "tempo mínimo" só é atingido com o uso de máquinas de última geração, de custo bastante elevado. A implementação em *software* muitas vezes é considerada ineficiente pelo fato do código implementado, por mais otimizado que esteja, ser executado em conjunto com o sistema operacional ou com outras aplicações.

2.1.1. Aplicações de Realidade Aumentada

Nesta seção serão apresentadas algumas áreas nas quais RA tem se destacado. As aplicações em RA são inúmeras e abrangem diversas áreas desde aplicações na área de entretenimento passando por aplicações médicas e mais recentemente aplicações móveis e comerciais [8].

Tecnologias que provêem o uso eficiente de imagens são de extrema importância para a medicina. Essa é uma das razões da existência de inúmeras pesquisas que desenvolvem RA para esta área. A maioria das aplicações médicas visa orientar procedimentos através de imagens cirúrgicas. Estudos de imagens no pré-operatório como, por exemplo, tomografia computadorizada, ressonância magnética e ultra-som (sensores não-invasivos), provêem ao cirurgião a visão necessária da anatomia interna do paciente, e é através do estudo dessas imagens que a cirurgia é planejada. Os médicos podem usar a tecnologia de RA para a visualização dessas cirurgias, por exemplo. Com o conjunto dos dados coletados através dos sensores não-invasivos, esses dados podem ser renderizados e combinados em tempo real, dando ao médico uma espécie de visão de raio-X dos órgãos do paciente, resultando com isso em uma visão do interior do paciente sem a necessidade de grandes incisões [9], conforme ilustrado na Figura 1.

Além das cirurgias, uma outra grande aplicação são as imagens de ultra-som. Usando um *display* o médico pode ver a imagem do feto renderizada e sobreposta ao abdômen da paciente grávida, parecendo assim que a imagem está dentro da barriga, já que a mesma é renderizada em tempo real à medida que o feto se move [10]. RA também pode ser usada para o treinamento de cirurgiões principiantes. Instruções virtuais poderiam orientar o médico nos passos requeridos sem a necessidade de que o mesmo desvie sua

atenção do paciente para ler um manual [9].

Entre as inúmeras vantagens das aplicações de RA na medicina, as mais visíveis dizem respeito à fidelidade das imagens coletadas, já que as mesmas são capturadas em tempo real na sala de cirurgia, aumentando com isso o desempenho de toda a equipe cirúrgica e também propiciando a eliminação da necessidade de alguns procedimentos dolorosos e enfadonhos.

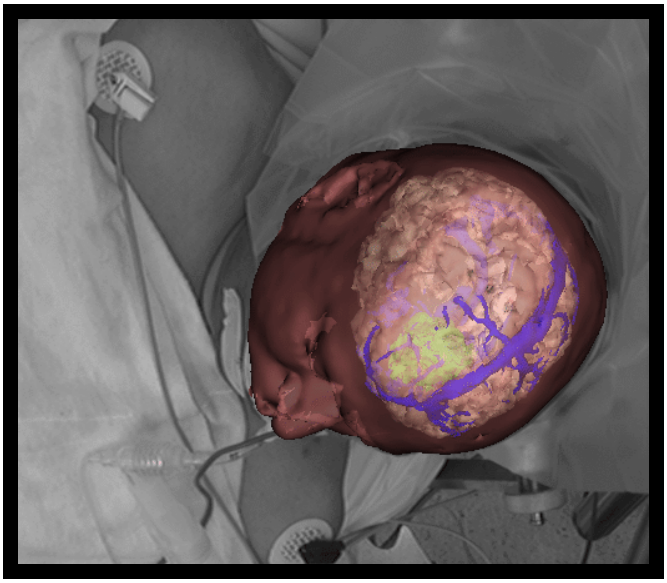

Figura 1. Imagem sobreposta ao paciente.

Com o intuito de ajudar em reparos e manutenções de um modo geral, pesquisas têm sido realizadas para o desenvolvimento de aplicações em RA. Para facilitar o entendimento das instruções, ao invés de ler manuais e observar figuras nestes, objetos 3D podem ser sobrepostos a um equipamento qualquer mostrando passo a passo as tarefas que devem ser realizadas e como fazê-las. Estes objetos podem ainda ser animados, para que as instruções sejam mostradas de uma maneira mais explícita.

Algumas aplicações já existentes consistem na manutenção de uma impressora a laser [1], e ainda de um encanamento industrial [9], onde são visualizados um mapa bidimensional (2D) da instalação e um modelo 3D das partes de interesse do equipamento real, ambos ilustrados na Figura 2.

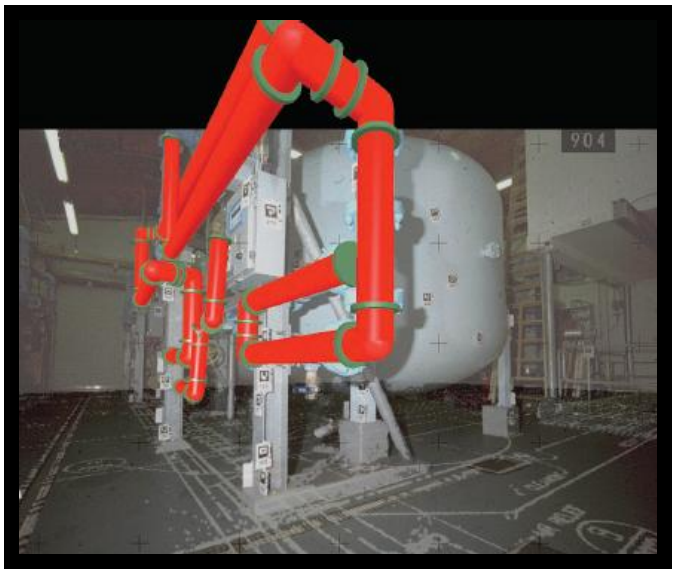

Figura 2. Encanamento industrial.

Uma aplicação muito comum em RA consiste em colocar pequenas notas em objetos e ambientes, notas estas que contém informações públicas e/ou particulares. Caso a aplicação venha a ter informações públicas, é necessário que haja uma disponibilidade de bases de dados; caso as informações tenham um caráter privado, as mesmas serão anexadas a objetos específicos [1]. O interessante nesse tipo de aplicação é a ajuda em tarefas cotidianas. Como exemplo, a aplicação *SignPost* [11] guia o usuário através de um prédio desconhecido mostrando uma variedade de sugestões para a navegação, com a possibilidade de prover uma visualização da estrutura do prédio através do destaque de elementos relevantes e a próxima saída a tomar, conforme mostrado na Figura 3. Uma outra aplicação similar a essa é um guia para museus, que mostra várias informações à medida que o usuário caminha por ele [12]. Ambas aplicações são também móveis, ou seja, utilizam PDAs (*Personal Digital Assistants*).

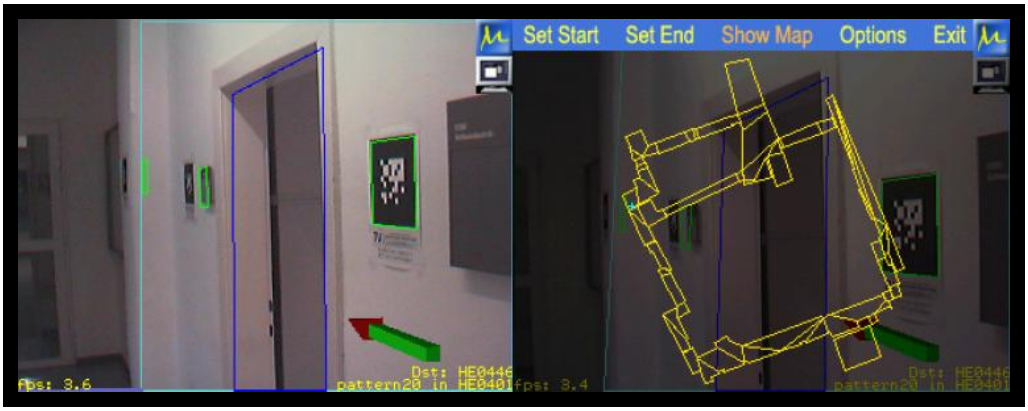

Figura 3. *SignPost* no modo *stand-alone* (esquerda) e *SignPost* mostrando o mapa da estrutura do prédio (direita).

Outra aplicação que vem se tornando usual tem relação com a área de publicidade, mais explicitamente a utilização de um vídeo em tempo real usado como propaganda virtual em um *outdoor* [9], conforme mostrado na Figura 4.

Figura 4. A propaganda *Pacific Bell* é virtual.

As aplicações na área de entretenimento são as mais diversas possíveis, desde jogos dos mais variados tipos até "contadores" de histórias. Entre os jogos criados com RA pode-se destacar o *ARQuake* [13], que foi desenvolvido baseado no jogo *Quake*, originalmente implementado para a plataforma *desktop*. O *ARQuake* é jogado no mundo real, o que dá ao usuário a mobilidade para ir onde desejar. Tudo que é visto é determinado exclusivamente pela orientação e posição da cabeça do usuário que está usando um HMD (*Head Mounted Display*). A aplicação *ARQuake* é mostrada na Figura 5.

Outra aplicação que mistura entretenimento e aplicação móvel é o *Invisible Train* [14]. Esse jogo foi desenvolvido inicialmente para crianças de ensino fundamental, e é um jogo *multiplayer* no qual os jogadores guiam um trem virtual sobre um trilho real em miniatura. Este trem só é visível para os jogadores através de PDAs, e para esses usuários são permitidas duas ações: operar as junções entre os trilhos e

mudar a velocidade do trem. Toda a interação ocorre através das telas sensíveis a toque dos próprios PDAs. A Figura 6 fornece uma idéia da aplicação *Invisible Train*.

Figura 5. Visão do usuário jogando *ARQuake*.

Figura 6. Dois PDAs rodando o *Invisible Train* (acima) e a interface do jogo (abaixo).

Outra aplicação bastante interessante na área de jogos é o *CamBall* [15]. Este jogo é um simples jogo de tênis, tendo como diferencial o fato de que os dois jogadores interagem entre si através de uma *Local Area Network* (LAN) ou Internet e com raquetes reais. Os jogadores se vêem através de computadores e nas raquetes são colocados marcadores cujas posições são capturadas por *webcams* instaladas em cada computador.

Saindo um pouco da área de jogos, mas ainda como entretenimento, foi desenvolvido o projeto *Magic Cubes* [16], ilustrado na Figura 7. Este foi desenvolvido para promover interações físicas e sociais pelos membros das famílias. O *Magic Cubes* consiste basicamente em marcadores no formato de cubo, que manipulados contam histórias, simulando um livro de história infantil.

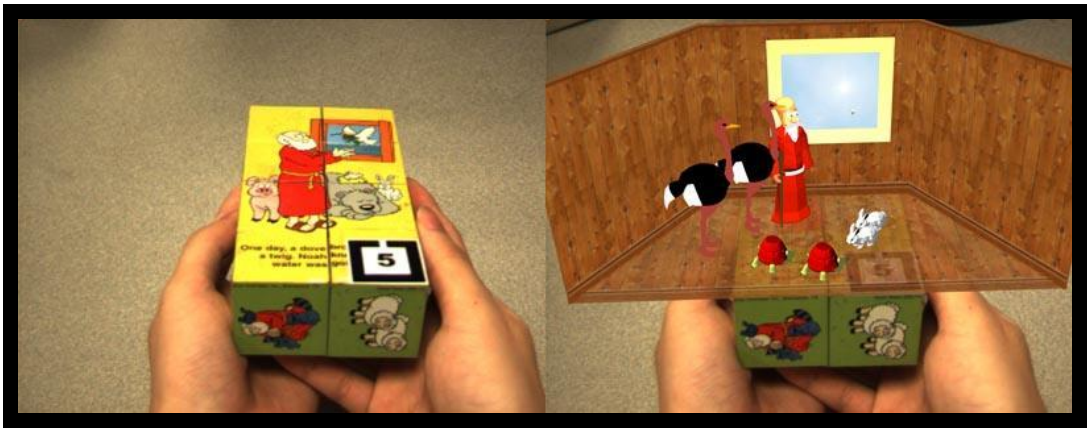

Figura 7. Aplicação *Magic Cubes* usada como contador de história.

Por fim, tem-se o *Magic Table* [17], que vem a ser um quadro branco onde se escreve, desenha e apaga. A diferença entre um quadro branco comum e o *Magic Table* consiste em *scanners* que capturam o que é escrito no quadro; o texto escrito então é colocado em retângulos (*patches*), que são manipulados através de pequenos círculos vermelhos (*tokens*) sobre o quadro. Uma ilustração do *Magic Table* pode ser vista na Figura 8.

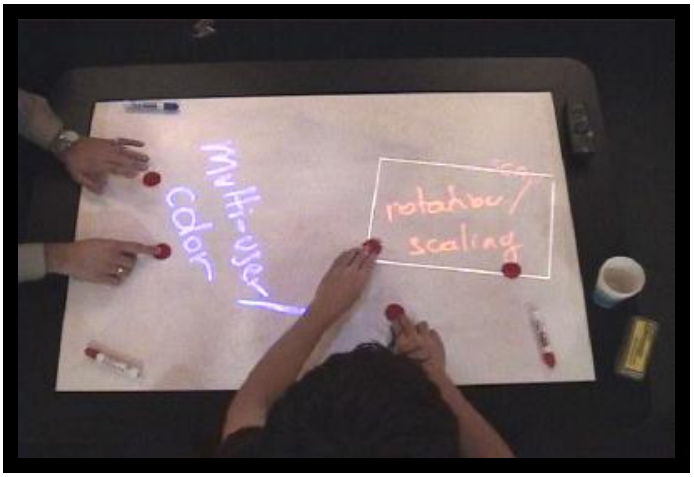

Figura 8. *Magic Table* sendo usado por dois usuários.

Arquitetos, membros de conselhos de cidade e grupos de interesse são alguns dos muitos tipos de usuários que se beneficiam da RA no planejamento urbano, podendo discutir alternativas à medida que visualizam a cidade virtual a sua frente. Dentro dessa perspectiva existe o *Urp* [18], que simula prédios e ventos soprando neste ambiente. Existe também um modelo mais simples [19] que possui prédios que podem ser facilmente movidos e animados, conforme mostrado na Figura 9. A grande vantagem em usar RA no planejamento de cidades é a facilidade de interação e visualização da visão de cada usuário, bastando para isso apenas manipular os prédios como se os mesmos fossem simples caixas.

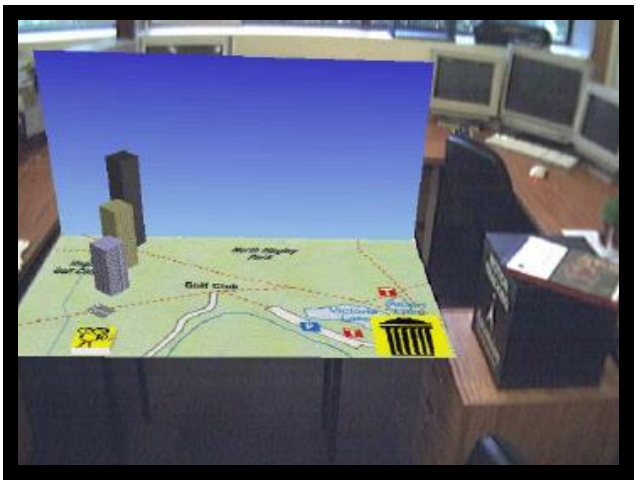

Figura 9. *Workspace* para planejamento urbano.

2.1.2. Dispositivos de Realidade Aumentada

Para que as aplicações de RA funcionem conforme o desejado é necessário o uso de uma série de dispositivos, tanto de entrada quanto de saída, que forneçam informações sobre o ambiente para a aplicação e o retorno do processamento realizado para o usuário. Os principais dispositivos utilizados são aqueles de visualização (HMDs *see-through*, HMDs opacos com uma câmera acoplada, telas de projeção, monitores de vídeo comuns) e de *tracking* (que detectam a posição do usuário ou de alguma parte do corpo do mesmo) [9]. Existem inúmeras variações desses dispositivos, e um breve resumo sobre eles será descrito a seguir.

A tecnologia dos *displays* continua a ser um fator limitante no desenvolvimento de sistemas de RA. Ainda não existem *displays* translúcidos que possuam brilho, resolução, campo de visão e contraste suficientes, capazes de sobrepor completamente a grande maioria das imagens reais por objetos virtuais. Além do mais, muitas tecnologias que começaram a alcançar esses objetivos ainda são volumosas, pesadas e de custo elevado. Apesar de tudo, nos últimos anos houve um grande avanço na tecnologia de *displays* translúcidos. Alguns exemplos de HMDs são mostrados na Figura 10.

Figura 10. Exemplos de HMDs.

Fabricantes bem estabelecidos de dispositivos ópticos e eletrônicos, como a Sony e a Olympus, produzem atualmente *displays* opacos, baseados em LCD (*Liquid Crystal Display*), voltados principalmente para assistir vídeos ou jogar *videogames*. Esses sistemas possuem uma resolução relativamente baixa (entre 180K e 240K *pixels*), campo de visão pequeno (cerca de apenas 30° na horizontal) e não suportam estéreo, mas são relativamente leves (abaixo de 120 gramas) e oferecem uma opção barata para pesquisas com *HMDs see-through* através de câmeras. A Sony introduziu a resolução SVGA (*Super Video Graphics Array*) em *displays* translúcidos ópticos, incluindo modelos estéreo (posteriormente descontinuados), os quais foram extensivamente utilizados em pesquisas na área de RA.

Um dos desafios encontrados no projeto de *HMDs see-through* através de câmeras é garantir que os olhos do usuário, assim como as câmeras, compartilhem efetivamente o mesmo caminho ótico, eliminando erros de *paralax* (diferença entre o que o observador vê e o que é capturado pela câmera) que afetam o desempenho de tarefas de curto alcance.

Uma abordagem alternativa para RA é projetar a informação virtual desejada (RA projetiva), diretamente, nos objetos do mundo físico que devem ser aumentados. No caso mais simples, as informações mostradas devem ser co-planares com a superfície na qual elas são projetadas e podem ser projetadas monoscopicamente por meio de um projetor comum, sem necessidade de usar óculos especiais.

Outra alternativa para RA projetiva recai nos projetores oculares, cujas imagens são projetadas através da linha de visão do observador pelo mundo. Os objetos focados são cobertos por um material reflexivo que reflete a luz de acordo com o ângulo de incidência. Múltiplos usuários podem visualizar diferentes imagens focando em um mesmo objeto, uma vez que elas são geradas por seus próprios sistemas de visualização.

Realizar com precisão o rastreamento da orientação da visão do usuário e sua localização é crucial para o registro de posicionamento em RA. Para ambientes internos específicos, muitos sistemas têm apresentado um bom resultado na aquisição do registro espacial dos objetos. Tipicamente tais sistemas empregam soluções híbridas (por exemplo, sensores magnéticos e de vídeo), para explorar as vantagens e minimizar as desvantagens de cada tecnologia de *tracking*. Sistemas que combinam acelerômetros e rastreamento por vídeo geralmente fornecem bons resultados, mesmo quando ocorrem movimentos rápidos durante o uso do sistema. Alguns exemplos de *trackers* são mostrados na Figura 11.

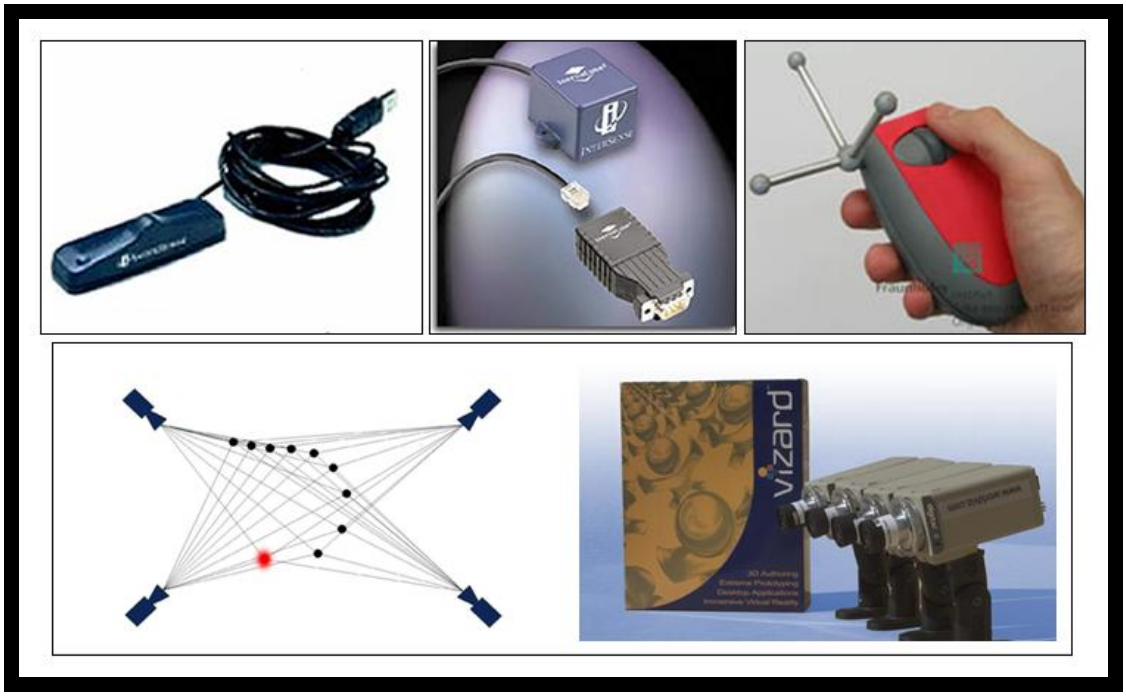

Figura 11. Exemplos de *trackers* usados em RA e RV.

Embora sistemas recentes de RA têm demonstrado registro eficiente em ambientes internos, ainda resta

muita pesquisa a ser feita em relação à calibração. RA eficiente requer conhecimento não só da posição do usuário, como também da posição de todos os objetos de interesse presentes no ambiente. Por exemplo, um mapa de profundidade da cena real é necessário quando se deseja oferecer suporte à oclusão na renderização. O rastreamento em ambientes externos e não conhecidos depende fortemente das mudanças que podem ocorrer no ambiente visualizado, monitorado através da colocação de marcadores fiduciais em posições conhecidas. Tais marcadores podem variar de tamanho e forma, e grande parte das técnicas de visão computacional consegue fornecer informação de rastreamento ainda com uma baixa taxa de atualização (cerca de 30Hz). Considerando que preencher todo um ambiente externo com marcadores é inviável, geralmente se utiliza uma abordagem baseada em localizador (GPS - *Global Positioning System)* em conjunto com marcadores ou outros detalhes específicos do ambiente. Esta é uma área de pesquisa recente, com muitos pontos ainda em aberto.

2.2. Sistemas Embarcados

Uma das principais ferramentas utilizadas na prototipação de módulos embarcados é o FPGA (*Field Programmable Gate Array*. Através dele é possível se projetar todo o circuito em detalhes, e verificar seu funcionamento antes da versão final do dispositivo. Esse tipo de dispositivo será descrito detalhadamente a seguir.

2.2.1. FPGAs

Um FPGAé um dispositivo semicondutor que contém componentes lógicos e interconexões programáveis. Os componentes lógicos programáveis podem ser programados para funcionar como portas lógicas básicas, como ANDs, ORs, XORs e NOTs, por exemplo, ou até mesmo como algumas funções combinacionais mais complexas, como decodificadores ou funções matemáticas simples. Na maioria dos FPGAs, esses componentes lógicos programáveis (ou blocos lógicos) também incluem elementos de memória, os quais podem ser simples *flip-flops* ou blocos de memória mais complexos [20].

Uma hierarquia de interconexões programáveis permite que os blocos lógicos de um FPGA sejam conectados à medida que são requeridos pelo *designer* do sistema, de forma similar a uma *protoboard*. Esses blocos lógicos e interconexões podem ser programados após o processo de fabricação pelo usuário/*designer* (como diz o termo, “programável em campo”), de forma que o FPGA possa realizar a função lógica que se deseja.

Os FPGAs são, geralmente, mais lentos do que seus concorrentes implementados por ASICs (*Application-Specific Integrated Circuits*), não suportam um *design* tão complexo quanto os suportados pelos ASICs e necessitam de mais potência. Todavia, eles apresentam várias vantagens, como um menor *time to market*, uma capacidade de ser reprogramado em campo com o objetivo de corrigir erros, e um menor custo de engenharia não-recorrente (custo necessário para se refazer partes do projeto do *hardware*, uma vez que o mesmo já foi finalizado). Existem alguns fabricantes que comercializam versões de FPGAs menos onerosas e sem muita flexibilidade, as quais não podem ser modificadas depois que o *design* é concluído. O desenvolvimento desses projetos é realizado em FPGAs comuns e depois migrado para uma versão fixa similar a um ASIC. CPLDs (*Complex Programmable Logic Device*), ou dispositivos lógicos complexos reprogramáveis, são uma outra alternativa.

As origens históricas dos FPGAs iniciaram-se com os CPLDs, em meados da década de 80. CPLDs e FPGAs incluem um número relativamente grande de elementos lógicos reprogramáveis. A densidade das portas lógicas dos CPLDs varia entre cerca de alguns milhares até 10 mil portas lógicas, enquanto os FPGAs tipicamente variam de dezenas de milhares a alguns milhões de portas lógicas. Um exemplo de FPGA com 20 mil portas lógicas é o *chip* FLEX, da Altera [21], mostrado na Figura 12.

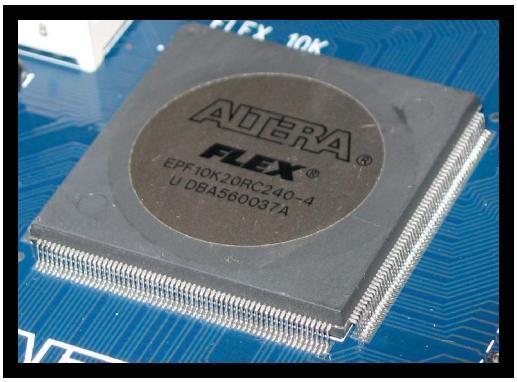

Figura 12. Um FPGA da Altera com 20.000 portas lógicas.

As diferenças primárias entre CPLDs e FPGAs estão em suas arquiteturas. Um CPLD apresenta uma arquitetura restrita que consiste de um ou mais *arrays* lógicos de somadores-multiplicadores, alimentando um número relativamente pequeno de registradores. Como conseqüência, eles possuem menos flexibilidade, mas têm a vantagem de atrasos mais previsíveis e uma proporção lógica/interconexão maior. A arquitetura dos FPGAs, por outro lado, é completamente baseada em interconexões. Isso possibilita que eles sejam mais flexíveis (em termos do número de projetos que são possíveis de se implementar com seu uso), mas também muito mais complexos de se programar.

Outra diferença notável entre CPLDs e FPGAs é a presença nos FPGAs de funções embarcadas de alto nível (como adicionadores e multiplicadores) e memórias embutidas. Uma diferença importante é que muitos FPGAs modernos oferecem suporte para reconfiguração completa ou parcial no próprio sistema, permitindo que os *designs* sejam modificados *on the fly*, tanto para atualizações do sistema quanto para reconfiguração dinâmica, como uma parte da operação normal do sistema. Alguns FPGAs possuem a capacidade de reconfiguração parcial, que permite que uma porção do dispositivo seja reprogramada enquanto a outra continua executando normalmente.

Recentemente, existe uma tendência em se utilizar uma abordagem de arquiteturas de grandes blocos misturados, através da combinação de blocos lógicos e interconexões de FPGAs tradicionais com microprocessadores embarcados e periféricos relacionados. Dessa forma, consegue-se construir um sistema completo em um *chip* programável. Exemplos de tais tecnologias híbridas podem ser encontrados nos dispositivos Xilinx Virtex-II PRO e Virtex-4 [22], os quais incluem um ou mais processadores PowerPC embarcados de fábrica no FPGA. O Atmel FPSLIC é outro exemplo de dispositivo, o qual utiliza um processador AVR em conjunto com a arquitetura lógica programável da Atmel. Uma abordagem alternativa é fazer uso de *cores* de processadores *soft*, que são implementados dentro da lógica do próprio FPGA. Entre esses *cores* estão o MicroBlaze e o PicoBlaze da Xilinx, os processadores NIOS e NIOS II da Altera, e o LatticeMico8 (código-aberto), assim como outros *cores* (comerciais ou livres) de processadores de terceiros.

Conforme mencionado anteriormente, muitos FPGAs modernos possuem a habilidade de serem reprogramados em tempo de execução, e isso leva à idéia de computação reprogramável ou sistemas reconfiguráveis – CPUs (*Central Processing Units*) que podem se reconfigurar para suportar uma tarefa específica. Ferramentas atuais de FPGA, todavia, não suportam completamente essa metodologia.

Deve-se perceber que novas arquiteturas não baseadas em FPGA estão começando a emergir. Microprocessadores configuráveis por *software*, como o Strech S5000, adotam uma abordagem híbrida fornecendo um *array* de *cores* de processadores e *cores* programáveis como FPGAs no mesmo *chip*. Outros dispositivos (como o *Mathstar's Field Programmable Object Array*, ou FPOA) fornecem *arrays* de objetos de alto nível programáveis que se enquadram entre os blocos lógicos de um FPGA e um processador mais complexo.

Aplicações de FPGAs incluem DSPs (Digital Signal Processors), *software-defined radios* (rádios implementados em *software*), sistemas de naves espaciais e de defesa, prototipação de ASICs, imagem médica, visão computacional, reconhecimento de fala, criptografia, bioinformática, emulação de *hardware*

computacional e uma crescente quantidade de outras áreas. Os FPGAs originalmente começaram como competidores dos CPLDs. À medida que seu tamanho, funcionalidades e velocidade aumentaram, eles começaram a dominar funcionalidades mais complexas, de forma que hoje em dia são comercializados como sistemas completos em *chips* (SOCs (*System-on-Chip*)). Os FPGAs podem ser utilizados em aplicações de qualquer área, e especialmente com algoritmos que possam fazer uso do paralelismo massivo oferecido por sua arquitetura. No escopo deste trabalho, o foco é dado ao suporte fornecido por FPGAs na área de computação gráfica.

A arquitetura básica típica consiste de um *array* de blocos lógicos configuráveis (CLBs - *Configurable Logic Block*s) e canais de roteamento. Um bloco lógico típico de FPGA consiste em uma *lookup table* (LUT) de 4 entradas, e um *flip-flop*, como mostrado na Figura 13.

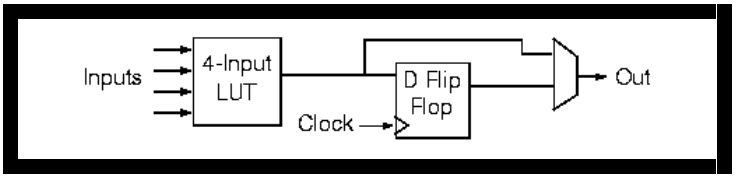

Figura 13. Bloco lógico.

Existe apenas uma saída, que pode ser a registrada (saída do *flip-flop*) ou a saída não-registrada da LUT. O bloco lógico possui quatro entradas para a *lookup table* e uma entrada de *clock*. Uma vez que os sinais de *clock* são geralmente roteados por redes dedicadas de propósito especial em FPGAs comerciais, eles são contabilizados separadamente dos outros sinais.

As posições dos pinos do bloco lógico do FPGA, seguindo a arquitetura básica típica mencionada como exemplo anteriormente, são mostradas na Figura 14.

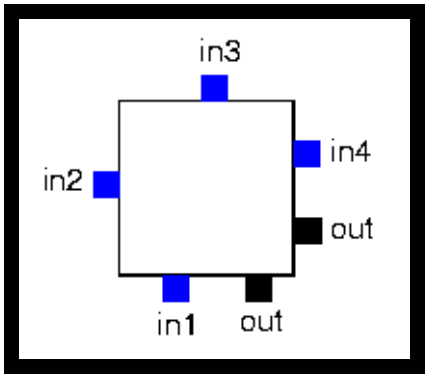

Figura 14. Posições dos pinos do bloco lógico do FPGA.

Cada pino é acessível por um lado do bloco lógico, enquanto o pino de saída pode ser conectado a fios em ambos os canais à direita e abaixo do bloco lógico. Cada pino de saída do bloco lógico pode se conectar a qualquer um dos segmentos dos canais adjacentes a ele.

De maneira similar, um canal de entrada e saída pode se conectar a qualquer segmento adjacente a ele. Por exemplo, um canal de entrada e saída localizado no topo do *chip* pode se conectar a qualquer uma das W conexões (sendo W o número de conexões do canal) do canal horizontal diretamente abaixo dele.

De forma genérica, o roteamento no FPGA não é segmentado. Ou seja, cada segmento de conexão pode se estender por apenas um bloco lógico, antes que ele termine em um *switch*. Através da união de vários *switches*, caminhos mais longos podem ser construídos. Para interconexões de maior velocidade, algumas arquiteturas de FPGA usam canais de roteamento mais longos que se estendem por múltiplos blocos.

Famílias modernas de FPGAs se aproveitam das características citadas anteriormente para incluir funcionalidades de alto nível diretamente no silício. Ter essas funções comuns embutidas no silício reduz a área de elementos lógicos necessária e dá a essas funções uma velocidade maior, se comparado à

construção das mesmas funcionalidades a partir de primitivas. Exemplos dessas funcionalidades incluem multiplicadores, blocos genéricos de DSPs, processadores embarcados, lógica de entrada e saída de alta velocidade e memórias embutidas.

Os FPGAs são também amplamente utilizados na validação de sistemas, incluindo validação pré-silício, validação pós-silício e desenvolvimento de *firmwares*. Isso permite que companhias produtoras de *chips* validem seus *designs* antes do *chip* ser produzido na fábrica, reduzindo ainda mais o *time to market*.

Com o objetivo de definir o comportamento do FPGA, o usuário fornece um *design* construído em uma linguagem de descrição de *hardware* (HDL – *Hardware Description Language*) ou um esquemático. VHDL e Verilog são exemplos de linguagens de descrição de *hardware* bastante utilizadas. Depois da definição do comportamento, através do uso de uma ferramenta de automação de *design* eletrônico, uma *netlist* é gerada e mapeada de acordo com a tecnologia presente no FPGA. A *netlist* pode ser inserida na arquitetura do FPGA em uso com o processo de *place-and-route*, geralmente realizado por algum *software* proprietário de *place-and-route* da empresa fabricante do FPGA. O usuário irá validar o mapeamento, assim como os resultados do *place-and-route* através de análise de tempo, simulação e outras metodologias de verificação. Uma vez que o *design* e o processo de validação estão finalizados, o arquivo binário gerado é usado para configurar o FPGA.

Como tentativa de reduzir a complexidade de desenvolvimento em linguagens de descrição de *hardware*, que são consideradas muitas vezes equivalentes em complexidade à linguagem assembly, existem medidas para aumentar o nível de abstração do *design*. Companhias como a Cadence, a Synopsys e a Celoxica utilizam SystemC como forma de combinar linguagens de alto nível com modelos de concorrência para permitir ciclos de desenvolvimento mais rápidos para FPGA do que quando se usa linguagens de descrição de *hardware* tradicionais. Abordagens baseadas em C ou C++ (em conjunto com bibliotecas ou extensões que permitam programação paralela) podem ser encontradas na ferramenta Catapult C da Mentor Graphics, assim como na ferramenta Impulse C da Impulse Accelerated Technologies. A Annapolis Micro Systems fornece uma abordagem gráfica do fluxo de dados de alto nível para *design* dos módulos de *hardware*. Linguagens como SystemVerilog, SystemVHDL e Handel-C (da Celoxica) procuram atingir o mesmo objetivo, mas são voltadas a tornar os engenheiros de *hardware* mais produtivos ao invés de tornar FPGAs mais acessíveis para engenheiros de *software*.

Com o objetivo de simplificar o desenvolvimento de sistemas complexos em FPGAs, existem bibliotecas de funções complexas pré-definidas e circuitos que foram testados e otimizados para acelerar o processo de *design*. Esses circuitos pré-definidos são comumente chamados de *IP cores* (termo utilizado para definir um módulo ou função a ser adicionada em um projeto num FPGA), e são disponibilizados por empresas de FPGA e outros fornecedores (raramente sem custo, e tipicamente liberadas sob licenças proprietárias). Outros circuitos pré-definidos são disponibilizados em comunidades de desenvolvedores, como a OpenCores.org (tipicamente gratuitos, e liberados sob GPL (*General Public License*), BSD (*Berkeley Software Distribution*) ou licenças similares) e outras fontes.

Em um típico fluxo de desenvolvimento, um desenvolvedor de aplicações para FPGA irá simular o *design*, em múltiplos estágios, durante o processo de produção da aplicação. Inicialmente a descrição RTL (*Register Transfer Level*) em VHDL ou Verilog é simulada através da criação de *testbenches* para simular o sistema e observar os resultados. Então, depois do sintetizador ter mapeado o *design* para uma *netlist*, a *netlist* é traduzida para um nível de descrição de portas lógicas onde a simulação é repetida para garantir que a síntese ocorreu sem erros. Finalmente, o *design* é enviado ao FPGA, e neste ponto atrasos de propagação são adicionados e a simulação é executada novamente, com os valores obtidos armazenados na *netlist*.

2.3. O Projeto ARCam

Tendo como objetivo a melhora no desempenho no processamento de aplicações de RA devido à utilização de *hardware* dedicado, o Grupo de Pesquisa em Realidade Virtual e Multimídia (GRVM) do CIn - UFPE começou a desenvolver o projeto ARCam [4]. Tal projeto consiste na utilização de um FPGA de porte médio, com capacidade equivalente a 60 mil elementos lógicos, para a criação de um sistema de RA *standalone*. O FPGA é ligado a uma câmera para efetuar a captura das imagens do ambiente, assim como

a um monitor, para mostrar o resultado do processamento, apresentando como resultado uma imagem "aumentada", ou seja, a mistura da imagem virtual com a real.

Esse projeto foi dividido em dois grandes módulos. O primeiro funciona como as tradicionais bibliotecas de detecção de padrões, responsável por capturar elementos conhecidos do ambiente, inferindo informações sobre localização e outras características relacionadas aos mesmos. Já o segundo recebe, dentre outras, informações sobre o posicionamento e a orientação dos objetos (marcadores) detectados e é responsável por renderizá-los junto à imagem real adquirida. Este Trabalho de Graduação visa a implementação inicial deste segundo módulo.

2.4. Trabalhos Relacionados

Diversas aplicações de RA têm sido desenvolvidas, mas voltadas para execução em dispositivos cujos processadores são de propósito geral, como computadores [3] e PDAs [23]. Nestas condições, todo o processamento é realizado em *software*, o que implica muitas vezes na perda de desempenho das aplicações, ou redução da qualidade da imagem quando as aplicações seguem as características de tempo real. Conciliar o uso destes tipos de processadores com aplicações de bom desempenho em tempo real, invariavelmente, acarreta altos custos pela exigência, por exemplo, de uma maior freqüência de *clock* e potência necessárias. É comum ver soluções onde o usuário necessita carregar consigo um *notebook* para rodar a aplicação, tornando a solução pouco versátil, de alto custo, principalmente no tocante ao peso e ao consumo de energia [3].

Na pesquisa preliminar realizada para a produção do projeto ARCam não foi identificada nenhuma solução flexível do ponto de vista de *hardware* e *software* para aplicações em RA. As soluções existentes para RA ainda não são, na sua grande maioria, acessíveis ao público em geral, por estarem em fase de pesquisa e serem sistemas dedicados a uma aplicação específica [3], [24], [25].

Enquanto todos os sistemas pesquisados implementam aplicações de RA usando uma abordagem de *hardware* e *software*, o ARCam foi desenvolvido de modo a ser constituído apenas por processadores de uso específico. Todas as funcionalidades foram implementadas em linguagem de descrição de *hardware*, resultando num sistema de *hardware* dedicado.

O *Hardwire* foi desenvolvido através do uso de linguagem de descrição de *hardware*, tendo em vista que um dos seus objetivos é oferecer suporte de renderização a uma aplicação de RA embarcada implementada com o ARCam. Diferentemente do Manticore [26], que é um projeto *open-source* de placa aceleradora 3D, o *Hardwire* funciona como um módulo interno da aplicação embarcada, com a função específica de renderização em *wireframe*.

O Manticore é completamente escrito em VHDL e atualmente é capaz de renderizar triângulos em um monitor VGA. O projeto inclui um módulo de saída VGA (também presente na placa de prototipação usada no ARCam), um controlador SDRAM (*Synchronous Dynamic Random Access Memory*) *open-source* (também desenvolvido completamente pelos autores do Manticore) e um rasterizador de triângulos (módulo responsável por desenhar os triângulos fornecidos em regiões da tela do usuário). Eventualmente esse projeto *open-source* irá incorporar primitivas padrões de gráficos 2D, múltiplas resoluções e número de cores, suporte à iluminação via *hardware* e uma interface PCI (*Peripheral Component Interconnect*) (talvez AGP (*Accelered Graphics Port*)) para sua conexão com um computador comum. Todo o projeto foi originalmente desenvolvido em um FPGA APEX20K200E da Altera, inserido na placa de prototipação do NIOS. A freqüência de operação conseguida com o uso dessa placa foi de 50MHz. Os autores do Manticore também pretendem construir um *design* próprio para a placa, criando assim um acelerador 3D completo.

A funcionalidade do *Hardwire* será validada na prática, em conjunto com o ARCam, através da visualização de objetos 3D (inicialmente um cubo) em um monitor conectado ao FPGA, diferentemente do trabalho realizado por Daniel Mc Keon [27], onde a verificação das transformações gráficas sintetizáveis implementadas ocorreram apenas através de simulação.

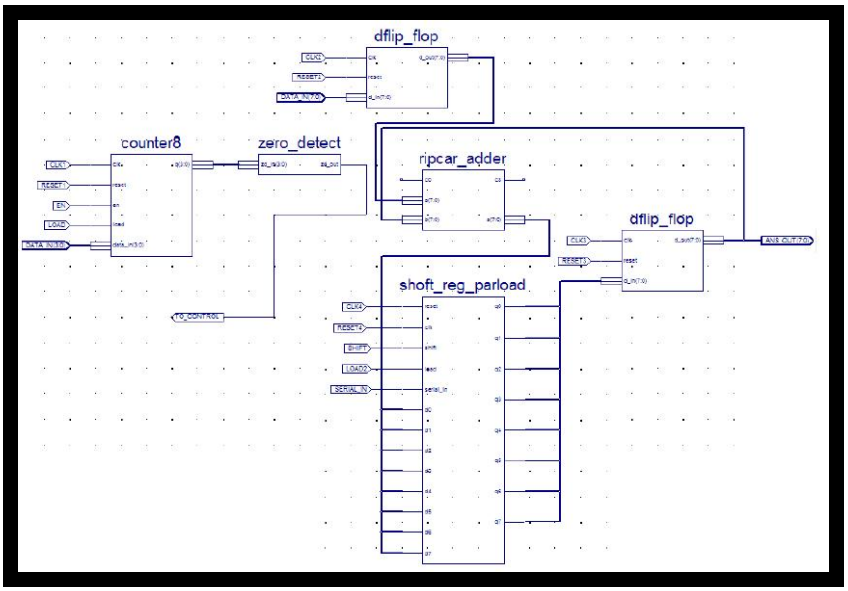

Figura 15. Multiplicador binário desenvolvido por Mc Keon.

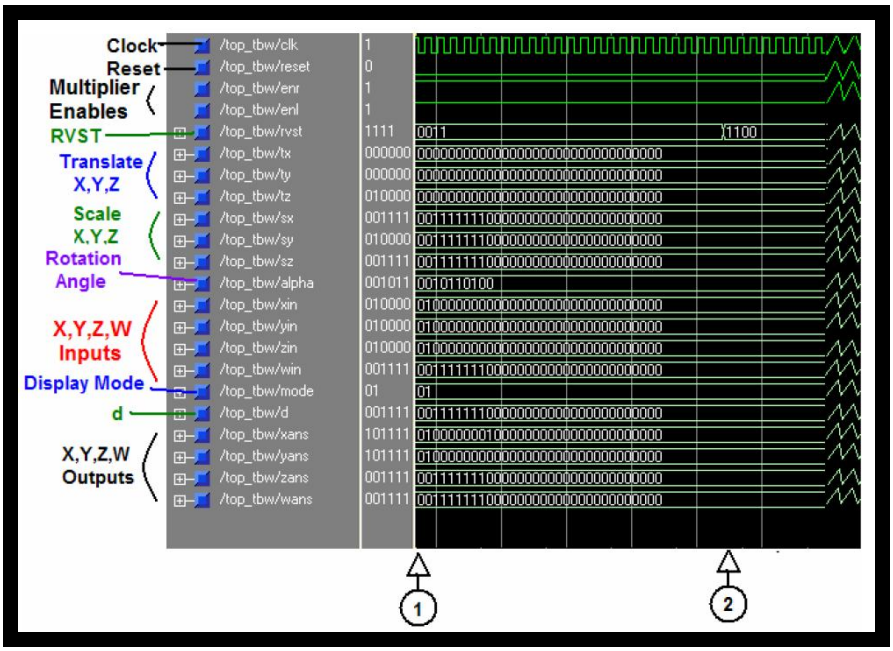

Figura 16. Simulação das funções implementadas por Mc Keon no ModelSim.

O trabalho desenvolvido por Mc Keon consistiu na implementação de um modelo sintetizável de transformações gráficas 3D. Dessa forma, ele possuía como objetivo criar a base, ou seja, implementar transformações vetoriais e projeções através do uso da linguagem VHDL. Essa camada de suporte permitiria um posterior estudo sobre processamento gráfico baseado em *clusters*, do Trinity College. As funções implementadas compreendiam principalmente operações manipuladas através de matrizes, como rotações, escalas e translações. O desenvolvimento do trabalho de Mc Keon começou a partir dos módulos mais básicos, ou seja, até os módulos mais simples (multiplicadores, por exemplo), foram desenvolvidos por ele, conforme mostrado na Figura 15. Após toda a implementação ter sido concluída, o funcionamento dos módulos criados foi validado através de simulações realizadas na ferramenta ModelSim, da Mentor Graphics, conforme ilustrado na Figura 16. Com esse trabalho, foi possível provar que os FPGAs são capazes de realizar cálculos complexos, como transformações vetoriais e projeções. Como conseqüência, eles podem ser usados como auxílio à CPU ou à GPU, ou até mesmo em soluções completamente embarcadas, como no caso do projeto ARCam.

3. Conceitos Básicos Relacionados

3.1. Transformações de Visualização

Uma vez criada a descrição de um objeto 3D, são executadas transformações geométricas que permitem definir condições particulares de visualização, ou seja, posição, tamanho e orientação do objeto. A cada posição particular do objeto, ou a cada posição particular do observador, corresponde uma diferente visualização do objeto. Quando se movimenta o objeto, ou o observador, relativamente ao sistema de coordenadas, estão sendo realizadas operações de transformação sobre o objeto, ou, sobre o sistema de coordenadas. As transformações se baseiam nas operações de translação, escala e rotação. Todas elas podem ser expressas como uma única matriz de transformação (resultado da concatenação de matrizes de transformações elementares) [28], como mostrado na Figura 17.

$$\begin{bmatrix} x' \\ y' \\ z' \\ \omega' \end{bmatrix} = \begin{bmatrix} \text{Transformação} \\ \text{de perspectiva} \end{bmatrix} \times \begin{bmatrix} \text{Transformação} \\ \text{de Câmera} \end{bmatrix} \times \begin{bmatrix} \text{Transformação} \\ \text{de Mundo} \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Figura 17. Concatenação de transformações geométricas.

Na implementação do *Hardwire*, as operações entre matrizes e vetores de coordenadas estão implícitas nas operações elementares presentes dentro dos módulos básicos criados (descritos posteriormente na Seção 4.6.1).

O objetivo das transformações de visualização 3D neste trabalho é, em resumo, traduzir as coordenadas de um ponto no sistema de coordenadas globais para uma posição na tela do monitor, caso o mesmo esteja visível. A posição dos pontos de entrada que compõem o objeto 3D é indicada através de três valores (x,y,z), responsáveis pela localização no sistema de coordenadas 3D, conforme ilustrado na Figura 18.

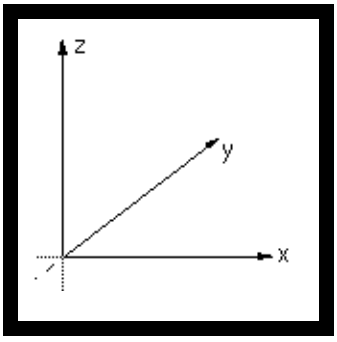

Figura 18. Sistema de coordenadas 3D.

O ponto fornecido define a localização fixa do vértice no mundo 3D. A posição do mesmo pode variar relativamente ao ponto de observação, caso seja modificada a posição da câmera de visualização, ou caso sejam aplicadas transformações sobre o objeto no espaço (translações ou rotações, por exemplo).

A visualização ocorre de acordo com o ponto de vista do observador, ou seja, a posição e orientação da câmera virtual. Ela é representada de acordo com o esquema apresentado na Figura 19. O ponto *C* indica a posição da câmera no espaço. Sua direção (para onde a câmera está apontada) é indicada pelo vetor *v*. O vetor *n* é sempre normal ao vetor *v* e é utilizado, juntamente com um terceiro vetor gerado *u*, para criar o sistema de coordenadas de câmera (baseado nesses três vetores, *u*, *v* e *n*).

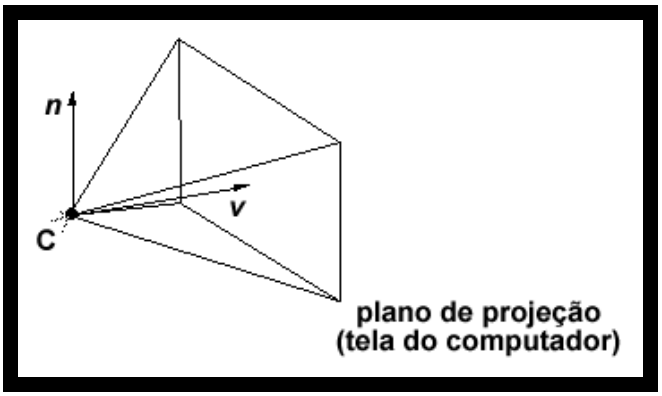

Figura 19. Representação da câmera virtual.

A toda posição do ponto no espaço 3D corresponde uma posição 2D no plano de projeção. Pode acontecer do ponto não se encontrar dentro dos limites da tela do computador, e nesse caso ele não é mostrado. O sistema de coordenadas de tela possui a sua origem no centro da tela e tem como eixo X (horizontal) seu comprimento e eixo Y (vertical) sua altura, conforme ilustrado na Figura 20.

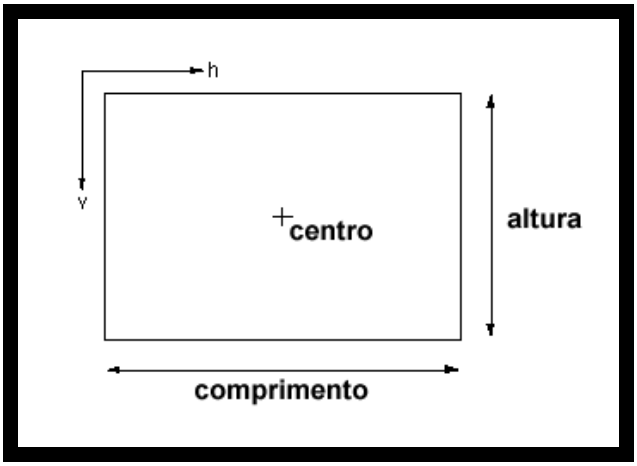

Figura 20. Sistema de coordenadas de tela.

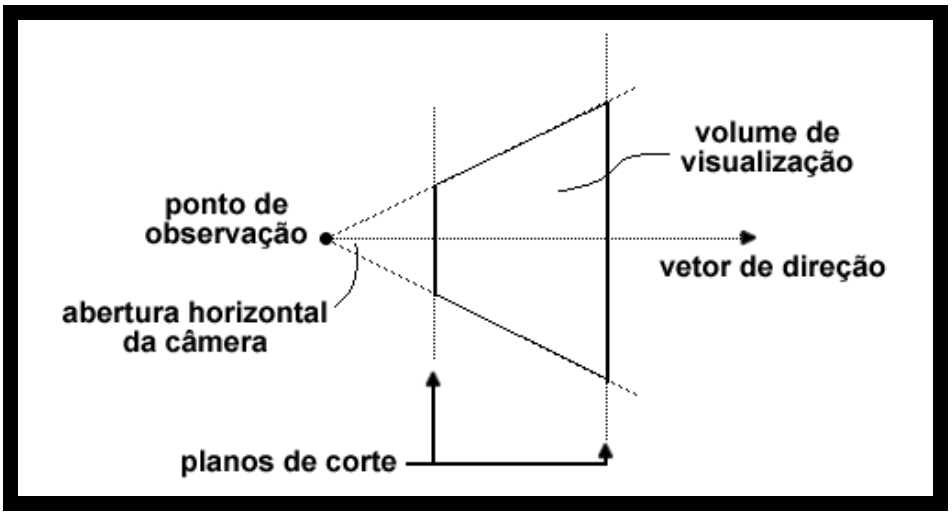

Figura 21. Modelo de câmera virtual.

O modelo de câmera adotado utiliza um ponto de origem (posição da câmera virtual), um ângulo de abertura horizontal, dois planos de corte para limitar a visualização e um vetor de direção, conforme

mostrado na Figura 21. Os dois planos de corte determinam uma região volumétrica de visualização, ou seja, todo ponto localizado dentro deste volume estará visível na tela de projeção.

O processo de renderização (desenho do objeto 3D na tela) obedece às etapas mostradas na Figura 22.

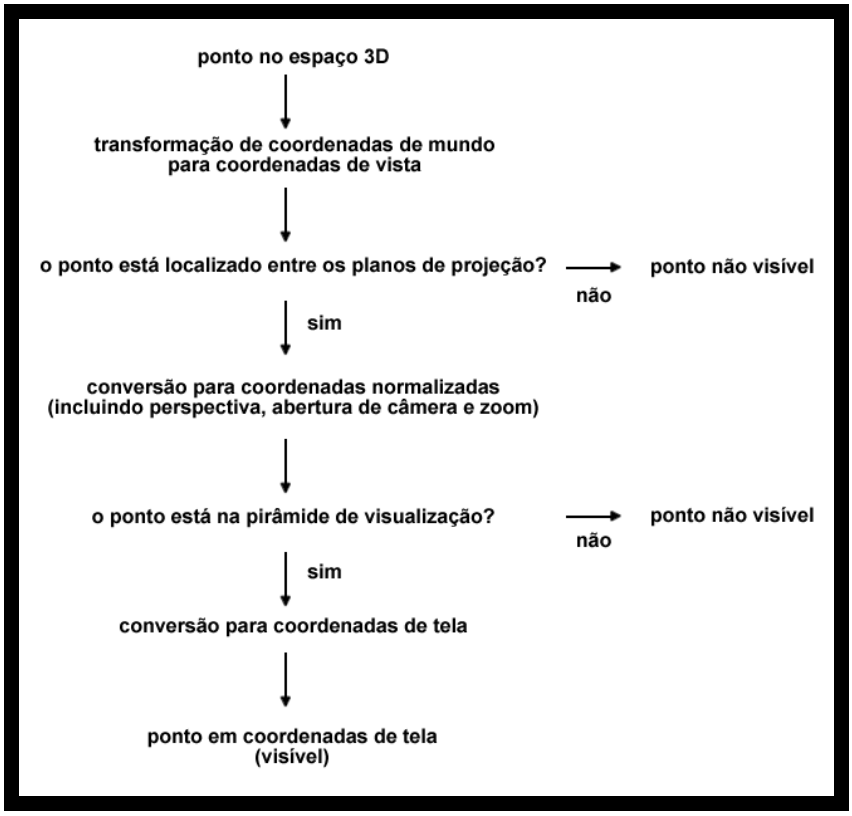

Figura 22. Etapas da renderização.

3.2. Rotação 3D

A matemática presente nas rotações 3D é mais complexa do que nas 2D, uma vez que um eixo de rotação deve ser especificado. Em duas dimensões, o eixo de rotação é sempre perpendicular ao plano XY, mas quando se trabalha com três dimensões o eixo de rotação pode ter qualquer orientação espacial [28].

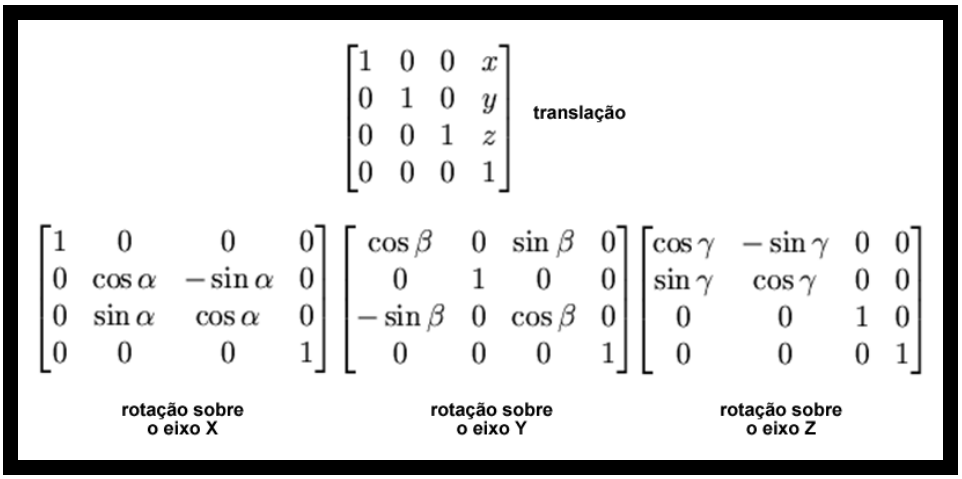

Figura 23. Matrizes de translação e rotação.

Operações com matrizes são utilizadas na matemática para realizar rotações. Assim como a operação de translação, uma rotação em torno de um eixo pode ser representada através de uma matriz, conforme mostrado na Figura 23. Durante a implementação do *Hardwire*, as matrizes foram abstraídas e apenas a seqüência das operações que ocorrem internamente a elas é utilizada.

Ao invés de uma única e complexa rotação sobre o eixo deslocado nas três coordenadas, o processo de rotação pode ser simplificado aplicando-se um conjunto de rotações sucessivas aos três eixos principais (X, Y e Z), conforme ilustrado na Figura 24.

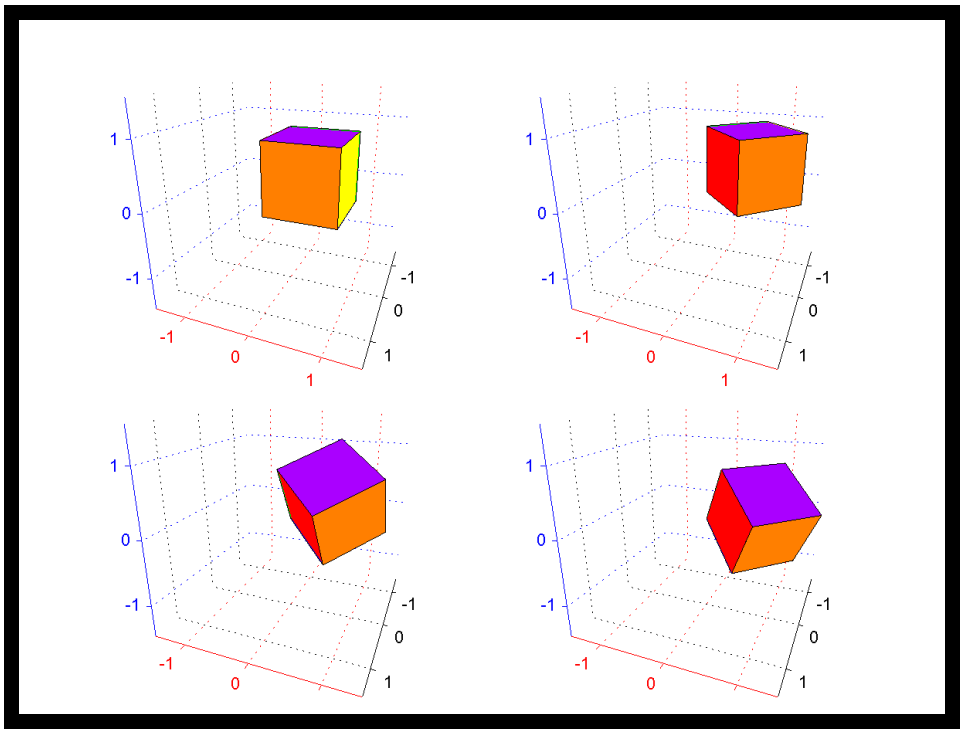

Figura 24. Rotações sucessivas aplicadas a um objeto 3D.

A versão corrente do protótipo do *Hardwire* já implementa a rotação em torno do eixo X. Foi criado um módulo que fornece os valores dos senos e cossenos necessários à rotação, e posteriormente esse processo será estendido para todos os três eixos.

3.3. Projeções 3D

Da mesma maneira que um desenhista, quando quer representar no papel a imagem de um objeto 3D, no computador também é preciso gerar uma projeção do objeto que se deseja exibir. Uma projeção 3D é simplesmente uma representação 2D de um objeto 3D. Existem várias técnicas e tipos de projeção, cada uma delas adequada a um tipo de aplicação. A mais simples é a projeção ortogonal, e a mais utilizada, a projeção em perspectiva. Essa última possui a capacidade de simular a projeção feita pelo olho humano, quando este capta a imagem de um objeto [28]. Para o primeiro protótipo do *Hardwire* foi implementado um esquema de projeção ortogonal bastante simples, apenas ignorando a coordenada *Z* do vértice (no sistema de coordenadas de vista), como pode ser visto na Figura 38 (a saída *Z* do módulo *world2eye* (explicado com mais detalhes na Seção 4.6.2) encontra-se desconectada). Dessa forma, o objeto 3D é projetado diretamente no plano de projeção, sem apresentar distorções, como exemplificado na Figura 25.

Geralmente, projeções transformam pontos de um sistema de coordenadas com dimensão *n* para pontos em um sistema de coordenadas com dimensão menor do que *n*. A computação gráfica utiliza há bastante tempo projeção em duas dimensões de objetos originalmente *n*-dimensionais. A projeção de um objeto 3D é definida pela projeção de retas originadas de um centro de projeção, passando por cada ponto do objeto e intersectando o plano de projeção. Uma vez que a projeção de uma linha é também uma linha, é

necessário apenas projetar os dois pontos extremos (origem e destino) da mesma e depois traçar uma reta entre eles.

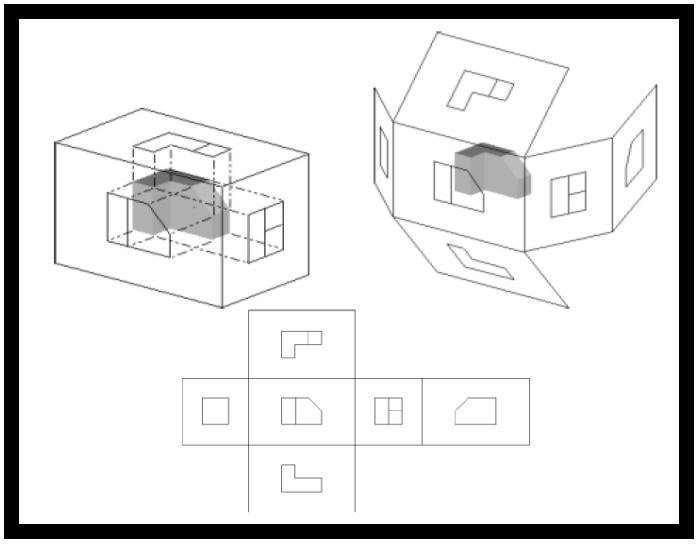

Figura 25. Proporção das dimensões do objeto mantidas nos planos de projeção.

As projeções mais utilizadas são as geométricas planares, pois a projeção é realizada sobre um plano, ao invés de uma superfície curva, e utiliza linhas retas, ao invés de curvas. Muitas projeções cartográficas não são nem planares nem geométricas. As projeções geométricas planares podem ser divididas em duas classes básicas de projeções: perspectiva e paralela (ortogonal). A diferença entre as duas está na relação entre o centro e o plano de projeção. Se a distância entre o centro e o plano de projeção for finita, a projeção é em perspectiva. Caso a distância seja infinita, a projeção é em paralelo.

O efeito visual de uma projeção em perspectiva é parecido com sistemas fotográficos e com o sistema visual humano, conhecido como encurtamento de perspectiva: o tamanho da projeção em perspectiva de um objeto varia inversamente com a distância do objeto em relação ao centro de projeção. Dessa forma, apesar da projeção dos objetos parecer realística, não é particularmente usada para armazenar formas exatas e medidas de objetos; distâncias não podem ser calculadas a partir da projeção, os ângulos são preservados apenas nas faces do objeto que são paralelas ao plano de projeção, e linhas em paralelo geralmente não são projetadas como linhas paralelas.

A projeção em paralelo é considerada uma visualização menos realística pela falta do encurtamento de perspectiva. Esse tipo de projeção pode ser utilizado para medidas exatas e de forma que linhas em paralelo permaneçam em paralelo após serem projetadas. Assim como na projeção em perspectiva, apenas os ângulos das faces paralelas ao plano de projeção são preservados.

3.4. Algoritmo de Desenho de Linhas

Continuando a descrição do processo de renderização em *wireframe*, um dos passos finais pode ser considerado a etapa de desenho das linhas originadas pela união de dois vértices, mais conhecidas por arestas.

O objetivo de todo e qualquer algoritmo de desenho de linhas é construir a melhor aproximação possível de uma linha ideal, levando em consideração as limitações presentes no dispositivo de saída. O algoritmo deve procurar satisfazer as seguintes características:

- a linha deve possuir uma aparência contínua, além de espessura e brilho uniformes;
- deve usar os *pixels* próximos à linha ideal, de forma que quanto maior a resolução, mais próximo da linha ideal ficará o resultado;

- deve gerar a linha rapidamente.

Algumas técnicas capazes de realizar esse tipo de tarefa foram estudadas. Foram analisadas as seguintes características: simplicidade de implementação dos algoritmos, requisitos de *hardware* e desempenho. A simplicidade faz-se necessário devido ao pouco tempo de implementação disponível, além de possibilitar um fácil entendimento do algoritmo. Apesar da placa de prototipação do projeto possuir muitos recursos, deve-se buscar utilizar o *hardware* fornecido da melhor forma possível, uma vez que outros módulos também serão implementados na própria placa e compartilharão as funcionalidades da mesma entre si. O desempenho do algoritmo de desenho de linhas também é essencial, pois implica diretamente na redução da taxa de quadros por segundo (*fps*) da aplicação.

A técnica que obteve os melhores resultados, levando em conta as três características listadas acima, foi aquela desenvolvida por Jack E. Bresenham [28], [29]. O algoritmo, denominado "algoritmo de desenho de linhas de Bresenham", apesar de não tratar o efeito de serrilhado, foi escolhido para ser usado no módulo de desenho de linhas do projeto *Hardwire*. Decidiu-se por não utilizar um mecanismo de *anti-aliasing* na concepção do protótipo desse projeto pela simplificação do mesmo e por essa característica apresentar uma prioridade menor de implementação, principalmente quando comparada com outros módulos que também fariam parte do sistema. Esse algoritmo determina quais pontos devem ser plotados em um *display* de duas dimensões (X e Y) com o objetivo de se obter uma representação aproximada de uma linha reta, entre os dois pontos fornecidos inicialmente como parâmetro. A técnica é geralmente usada para desenhar linhas em uma tela de computador, uma vez que utiliza apenas operações de soma/subtração de números inteiros e *shifting* (deslocamento) de *bits*. A escolha desse algoritmo justifica-se pelo fato dessas operações serem facilmente implementadas em *hardware* e apresentarem desempenho satisfatório, comparado a outras operações mais complexas como multiplicações ou divisões, por exemplo. O algoritmo de desenho de linhas de Bresenham é uma das mais antigas técnicas utilizadas no campo da computação gráfica.

Usando o algoritmo de Bresenham, a reta é desenhada entre dois pontos (*x0*, *y0*) e (*x1*, *y1*), nos quais X e Y indicam, respectivamente, coluna e linha, aumentando da esquerda para a direita e de cima para baixo, conforme mostrado na Figura 26. Inicialmente assume-se que a reta vai nessa direção, e que a distância horizontal *x1-x0* é maior que a distância vertical *y1-y0*. O objetivo do algoritmo é identificar, para cada coluna X entre *x0* e *x1*, a linha Y, nesta coluna X, que é mais próxima da reta e desenhar o *pixel* em (*x*, *y*).

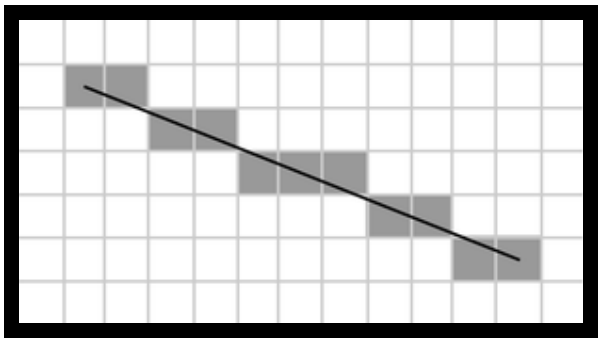

Figura 26. Exemplo do resultado do algoritmo de Bresenham.

O problema principal é descobrir qual o *pixel* mais próximo da reta, dada uma coluna qualquer. A equação genérica da reta formada por dois pontos é mostrada na Figura 27.

$$y - y_0 = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0).$$

Figura 27. Equação genérica da reta.

Uma vez que se sabe o valor da coluna, X, a linha do *pixel* é dada pelo arredondamento (teto) de Y para o número inteiro mais próximo. Todavia, calcular explicitamente esse valor para cada coluna X não é a forma mais otimizada. Percebe-se que Y começa em *y0*, e cada vez que se adiciona 1 ao valor de X, o valor (*y1-y0*)/(*x1-x0*) é adicionado a Y. Além do mais, uma vez que esse valor corresponde à inclinação da reta, por definição está entre 0 e 1. Em outras palavras, após o arredondamento, em cada coluna é utilizado o mesmo Y da coluna anterior ou o Y adicionado de 1.

É possível decidir qual o próximo valor de Y através de um monitoramento de um valor de erro, que indica a distância vertical entre o valor atual do Y e o valor exato do Y na reta para o X desejado. Cada vez que o valor de X é incrementado, o erro é aumentado com o valor da inclinação da reta. Sempre que o erro ultrapassar 0.5, a reta se torna mais próxima do próximo valor de Y, então o valor deve ser adicionado de 1, decrementando simultaneamente 1 do valor do erro acumulado. O procedimento funciona de acordo com o pseudo-código mostrado na Figura 28, assumindo que *plot(x,y)* desenha um ponto na tela e que a função *abs* retorna o valor absoluto.

```
function line(x0, x1, y0, y1)
     int deltax := abs(x1 - x0)
     int deltay := abs(y1 - y0)
     real error := 0
     real deltaerr := deltay / deltax
     int y := y0
     for x from x0 to x1
         plot(x,y)
         error := error + deltaerr
         if error ≥ 0.5
             y := y + 1
             error := error - 1.0
```

Figura 28. Pseudo-código de uma implementação não otimizada do algoritmo de Bresenham.

Essa primeira versão do algoritmo apenas trata de retas desenhadas da esquerda para a direita, e de cima para baixo. O objetivo da versão final do algoritmo é desenhar retas em qualquer direção. A versão mostrada na Figura 29 suporta retas na direção oposta, bastando apenas que se inverta os pontos iniciais, caso *x0 > x1*. Para determinar se a reta segue para cima, basta checar se *y0 >= y1*. Em caso positivo, decrementa-se 1 de Y, ao invés de incrementá-lo. Por último, deseja-se generalizar o algoritmo para desenhar retas em todas as direções. Até o momento está se utilizando o valor de X como base e o valor de Y como variação através da inclinação da reta. Ao trocar a coordenada X pela Y, dá-se suporte a retas que apresentam variação maior no outro eixo, diferente do X.

```
function line(x0, x1, y0, y1)
    boolean steep := abs(y1 - y0) > abs(x1 - x0)
    if steep then
       swap(x0, y0)
       swap(x1, y1)
    if x0 > x1 then
       swap(x0, x1)
       swap(y0, y1)
    int deltax := x1 - x0
    int deltay := abs(y1 - y0)
    real error := 0
    real deltaerr := deltay / deltax
    int y := y0
    if y0 < y1 then ystep := 1 else ystep := -1
    for x from x0 to x1
       if steep then plot(y,x) else plot(x,y)
       error := error + deltaerr
       if error ≥ 0.5
          y := y + ystep
          error := error - 1.0
```

Figura 29. Versão do algoritmo de Bresenham que suporta linhas em qualquer direção.

O problema presente nessa abordagem é que computadores geralmente operam com menos velocidade

sobre números fracionários, como *error* e *deltaerr*. Além do mais, o erro pode ser acumulado através de muitas operações de ponto-flutuante. Trabalhar com números inteiros seria mais rápido e preciso. Caso se multiplicassem todos os números fracionários anteriormente por *deltax*, teria-se todos os números expressos em formato inteiro. Após essa operação, o único problema restante é a constante 0.5. Para lidar com essa última questão, ambos os lados da comparação são multiplicados por 2. A multiplicação resultante por 2 pode ser implementada através de uma operação de deslocamento de *bits*, ao invés de uma multiplicação convencional, o que aumenta significativamente a velocidade do algoritmo. A nova versão do algoritmo é apresentada na Figura 30.

```
function line(x0, x1, y0, y1)
    boolean steep := abs(y1 - y0) > abs(x1 - x0)
    if steep then
        swap(x0, y0)
        swap(x1, y1)
    if x0 > x1 then
        swap(x0, x1)
        swap(y0, y1)
    int deltax := x1 - x0
    int deltay := abs(y1 - y0)
    int error := 0
    int ystep
    int y := y0
    if y0 < y1 then ystep := 1 else ystep := -1
    for x from x0 to x1
        if steep then plot(y,x) else plot(x,y)
        error := error + deltay
        if 2×error ≥ deltax
            y := y + ystep
            error := error - deltax
```

Figura 30. Pseudo-código da versão otimizada do algoritmo de Bresenham.

Essa última versão foi a escolhida e implementada em *hardware*, no projeto *Hardwire*. Foram feitos apenas alguns ajustes no algoritmo, de forma que ele também suportasse retas paralelas aos eixos cartesianos.

3.5. Representação de Objetos 3D

Existem inúmeras formas de representação de objetos. A escolha do tipo de representação varia de acordo com o objetivo de cada aplicação específica. A seguir é mostrado um quadro com alguns exemplos de formas de representação de objetos [28].

Quadro 1
Formas de representação de objetos 3D

Representação Aramada
Representação por Faces (ou superfícies limitantes)
Representação por Faces Poligonais
Representação por Enumeração da Ocupação Espacial
Representação por Decomposição do Espaço em *Octrees*
Representação por Decomposição do Espaço em *Quadtrees*
Quadtrees e *Octrees* Lineares
Quadtrees e *Octrees* Híbridas
Representação por Partição Binária do Espaço (BSP - *Binary Space Partition*)
Representação Implícita

Como o objetivo do *Hardwire* é a renderização de objetos 3D em *wireframe*, optou-se por utilizar a representação aramada, usando como parâmetro de entrada as coordenadas dos vértices e uma lista de arestas formada pelos mesmos. A Figura 31 ilustra a representação aramada, na esquerda, ao contrário da representação por faces poligonais, na qual existe preenchimento, na direita.

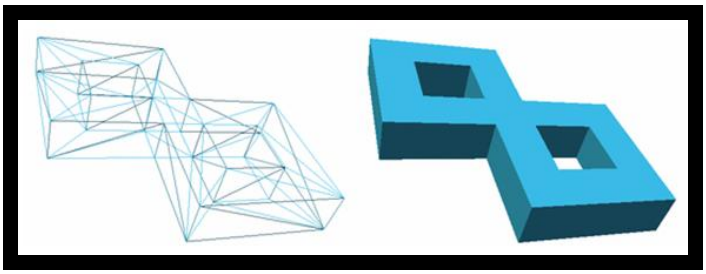

Figura 31. Representação em aramado e por faces poligonais.

3.5.1. Representação Aramada ou Por "*Wireframe*"

Nesta forma de representação os objetos são descritos por um conjunto de arestas que definem as bordas do objeto. O método é simplesmente uma extensão 3D do método de representação de objetos 2D por arestas. A principal vantagem desta técnica é a sua velocidade na exibição dos modelos, pois é necessário apenas exibir um conjunto de linhas. Como desvantagens da representação aramada, podem ser citados:

- geração de uma representação "ambígua". Ou seja, quando o modelo é exibido, pode dar margem a mais de uma interpretação. A Figura 32 exemplifica o problema. O primeiro cubo encontra-se na representação aramada e os dois seguintes, preenchidos com duas possíveis interpretações. O problema não reside propriamente no fato de que a simples exibição das linhas gera ambigüidades, mas sim, na constatação de que o modelo não fornece informações suficientes para que estas sejam eliminadas (no exemplo seria necessário remover as arestas da parte traseira do objeto);

- é bastante difícil, e em alguns casos impossível, realizar certas operações como a determinação de massa, volume, inclusão ou não de pontos, entre outras.

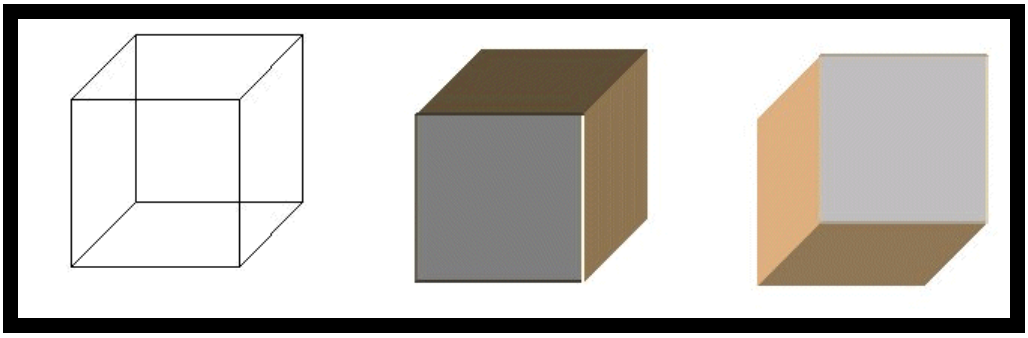

Figura 32. Ambigüidades da representação aramada.

3.5.2. Representação Por Faces (ou Superfícies Limitantes)

Esta técnica consiste em definir um modelo através de um conjunto de polígonos que delimitam uma região fechada do espaço. Esta região define o interior do modelo. Aos polígonos que limitam a região dá-se o nome de FACES. O objeto formado por esta técnica é normalmente chamado de POLIEDRO, ou seja, composto de muitos DIEDROS (diedro = semi-espaço).

Esta técnica também pode ser considerada uma extensão (mais inteligente) da modelagem 2D por arestas. Como naquele caso, onde há, pelo menos, duas formas de armazenamento das arestas, neste, é possível fazer uma analogia com as faces, que podem ser representadas de duas formas básicas:

- através de uma lista de vértices explícitos, na forma

$$\text{FACE: } (x_1, y_1, z_1)\text{-}(x_2, y_2, z_2)\text{- .... - } (x_n, y_n, z_n);$$

- através de duas listas, onde na primeira é dada a topologia da face (apenas o número dos vértices

que a compõem), na forma

FACE: v_1, v_2, v_3, ..., v_n

- e na segunda, onde é definida a geometria da face, com a explicitação das coordenadas de cada vértice do modelo, na forma

VÉRTICES:

1 - (x_1, y_1, z_1)

2 - (x_2, y_2, z_2)

....

n - (x_n, y_n, z_n).

3.5.3. Representação Por Enumeração de Ocupação Espacial

Também conhecidos como Modelos de Subdivisão (ou Decomposição) do Espaço, esta classe de formas de armazenamento decompõe o sólido em "pedaços".

Nessa representação, um sólido é visto como uma coleção de partes mais simples. O problema existente é saber o que significa uma parte mais simples. Na modelagem de um computador, por exemplo, o teclado pode ser uma das partes e o monitor outra.

Na Enumeração Exaustiva/Enumeração de Ocupação Espacial, a idéia é dividir o espaço em regiões e definir o objeto através das regiões que ele ocupa neste espaço. Esta técnica é bastante útil quando se deseja calcular propriedades de massa, pois basta saber, por exemplo, o volume de uma das partes em que o espaço foi dividido e multiplicar este valor pelo número de divisões ocupadas pelo objeto.

O espaço é subdividido em cubos formando uma “grade 3D”, como mostrado na Figura 33. A cada um destes cubos dá-se o nome de *voxel*. Codifica-se um sólido determinando quais *voxels* pertencem a ele.

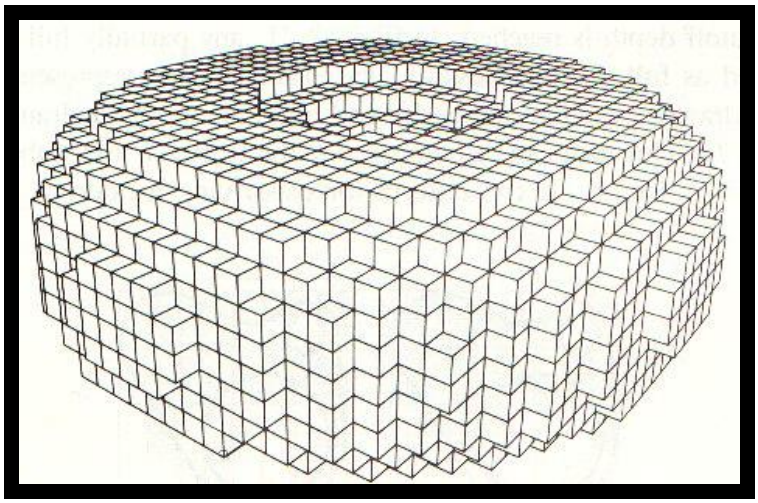

Figura 33. Exemplo de objeto descrito por *voxels*.

Como vantagem desse tipo de representação, é fácil determinar se um dado ponto pertence ou não ao sólido: basta verificar se esse ponto pertence a algum dos *voxels*. Também é fácil determinar se dois objetos se intersectam. Operações de união, interseção e diferença entre sólidos são facilitadas com essa representação, uma vez que se podem aplicar as operações às unidades menores (discretas) que compõem o objeto. Uma desvantagem visível é a grande quantidade de memória necessária para uma

representação detalhada de um objeto 3D.

3.5.4. Representação por *Octrees* e *Quadtrees*

Considerada um caso particular da Enumeração de Ocupação Espacial, a técnica "representação por *octree*" (ou árvore com 8 filhos) envolve o objeto, que em seguida é dividido em 8 cubos menores com o mesmo tamanho (octantes), conforme ilustrado na Figura 34. Cada um destes é então classificado em:

- cheio, caso o objeto ocupe todo o cubo;
- vazio, caso o objeto não ocupe nenhuma parte do cubo;
- cheio-vazio, caso o objeto ocupe parte do cubo.

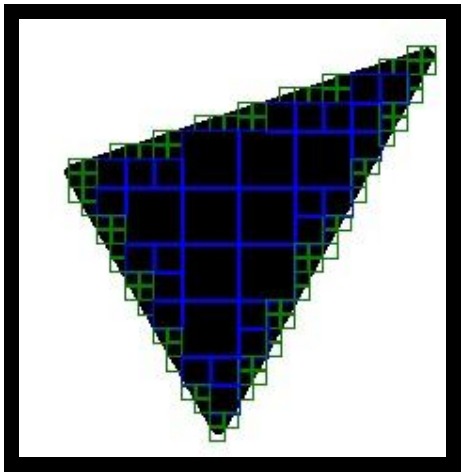

Figura 34. Exemplo de objeto descrito por *Quadtrees*.

Quando um octante for classificado como "cheio-vazio" ele é novamente dividido em 8 partes iguais e o processo de classificação é refeito para as novas partes. Este algoritmo repete-se até que só existam cubos com as duas primeiras classes.

Essa forma de representação é um caso especial de Enumeração Espacial. Neste caso, os *voxels* passam a ser cubos de dimensões variáveis. Para o armazenamento de objetos 2D, usam-se as *quadtrees*. Nelas divide-se o plano onde está o objeto em 4 partes iguais e classifica-se cada parte da mesma forma das *octrees*. Em geral, armazena-se a estrutura de *octrees* e *quadtrees* em forma de árvore.

A representação por *octrees* apresenta as mesmas vantagens da Enumeração Espacial, além de permitir uma representação mais detalhada com um gasto menor de memória. Como desvantagem, a manipulação desta técnica é mais árdua, ou seja, deve varrer toda a estrutura em árvore para obter a representação final do objeto 3D.

4. Hardwire

Esse capítulo descreve a plataforma utilizada no desenvolvimento do *Hardwire*, a metodogolia adotada durante a execução deste trabalho e a arquitetura construída. Além disso, é realizada uma comparação entre os formatos ponto-flutuante e ponto-fixo, de forma a esclarecer a escolha do formato usado na implementação. Os módulos de *hardware* implementados, dificuldades encontradas e resultados obtidos com o protótipo são, por fim, apresentados.

4.1. A Plataforma ARCam

Pesquisas relacionadas às arquiteturas de câmeras inteligentes normalmente são direcionadas aos problemas de processamento de imagem, como reconhecimento de padrões, que possam, por exemplo, identificar gestos [30], falhas de fabricação e problemas de perseguição de objetos em movimento. O ARCam pretende implementar módulos de processamento de imagem em *hardware*, assim como prover a infra-estrutura necessária para a sobreposição de elementos virtuais na imagem do mundo real, como uma forma de aperfeiçoar a interface com o usuário da aplicação.

A solução proposta no ARCam utiliza uma plataforma de desenvolvimento composta por uma placa Altera com o FPGA Stratix II, um sensor de imagem, além de um monitor de vídeo VGA.

A grande flexibilidade na implementação de *firmware* através de uma linguagem de descrição de *hardware*, como VHDL, possibilita escalabilidade na hora de duplicar um componente dentro do FPGA para melhorar o desempenho do processamento. A este tipo de sistema implementado em um FPGA, integrando vários módulos, dá-se o nome SoC. A Figura 35 mostra o ambiente de desenvolvimento utilizado no projeto, onde foi utilizado um sensor de imagem conectado à placa de desenvolvimento baseada em FPGA.

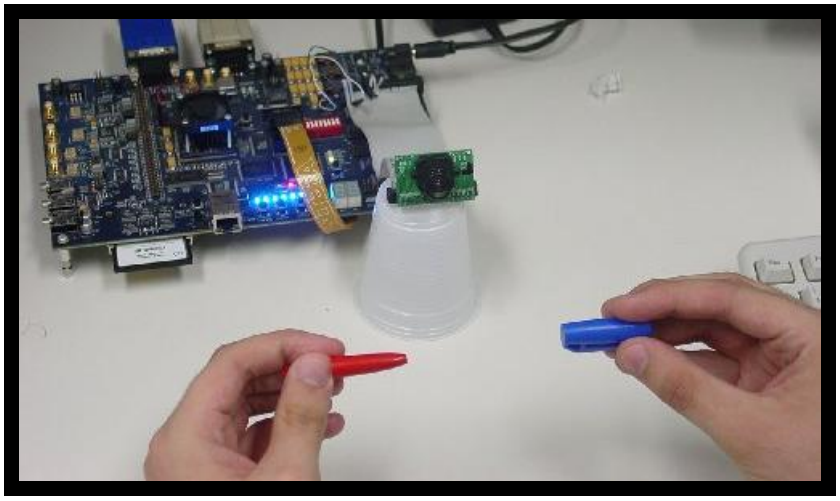

Figura 35. *Hardware* de desenvolvimento (FPGA + sensor de imagem).

O sistema é dividido em módulos, que são responsáveis pela aquisição, armazenamento, processamento e projeção de imagens. Em sua arquitetura, ilustrada na Figura 36, estão presentes o sensor de imagens, uma memória, o módulo de processamento PONG (criado como prova de conceito do funcionamento do arcabouço desenvolvido), um multiplexador e o vídeo VGA.

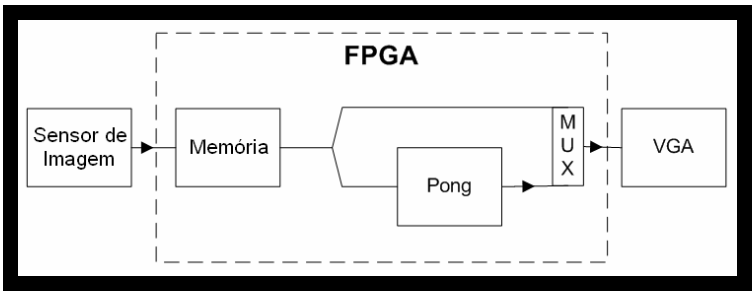

Figura 36. Arquitetura do sistema ARCam.

O sensor de imagem é conectado pino a pino com o módulo responsável pelo armazenamento na memória dos *frames* capturados. Os *pixels* que compõem cada *frame* são armazenados um a um, de forma seqüenciada, usando endereçamento consecutivo. Da mesma forma eles são lidos pelo módulo PONG para que seja feito o processamento devido em cada um deles.

A capacidade de armazenamento da memória corresponde a um *frame* de resolução 320x240 e 24 *bits* (8 por cor RGB), sendo os *pixels* armazenados numa lista do tipo FIFO (*First In First Out*). Cada *pixel* que é guardado será lido pelo módulo PONG e processado.

Uma solução embarcada para aplicações em RA necessita primeiramente do subsistema de aquisição de imagem do ambiente observado. Em sistemas convencionais, geralmente, é utilizada uma câmera com saída digital. Ela fornece os *frames* para a aplicação que fará a inserção dos componentes virtuais no ambiente observado. Já em um sistema embarcado este subsistema pode se restringir apenas ao sensor de imagem, que é o componente básico de uma câmera. Neste, a luz proveniente do ambiente sensibiliza uma matriz de sensores de luz (fotodiodos, transistores ou capacitores), onde cada ponto passa por um conversor analógico/digital quantificando os valores de níveis de tensão da matriz. A saída de um sensor de imagem é basicamente composta pelos sinais elétricos de sincronismo vertical, horizontal e de *pixels* que informam, respectivamente, quando um *frame* é finalizado, quando uma linha termina e quando o *pixel* está com seu valor estável no barramento.

Para realizar o processamento e tratamento de entrada e saída, um FPGA é utilizado pela sua característica principal de ser um *hardware* configurável. Ele pode implementar em seu interior os módulos necessários para interação com os sinais elétricos do sensor de imagem, já que possui um grande número de pinos de propósito geral, diferentemente de outras tecnologias.

Por fim, a arquitetura possui um subsistema responsável por exibir a imagem do mundo real, capturada pelo sensor de imagem, "aumentada" com as informações virtuais agregadas no subsistema de processamento. Numa primeira etapa, a saída do sistema será VGA, podendo ser ligada a um HMD ou a um monitor de vídeo comum. Em uma segunda etapa, quando todos os outros subsistemas estiverem concluídos, uma tela de LCD embarcada será incorporada ao sistema.

O Hardwire encontra-se inserido dentro da arquitetura do ARCam, localizado dentro do bloco de processamento (mesmo local onde se encontra o módulo PONG, mostrado na Figura 36).

4.2. Metodologia Utilizada

Para o desenvolvimento do projeto *Hardwire* foram utilizadas três linguagens computacionais: C, Java e VHDL. As duas primeiras tiveram papel fundamental na fase de criação do protótipo em *software* contendo as funcionalidades desejadas, além de algumas ferramentas criadas para dar suporte aos testes da versão final implementada. A última linguagem citada encontra-se presente na implementação em *hardware* de todo o projeto. Através dela foi possível criar todo um sistema sintetizável com as mesmas funcionalidades do protótipo desenvolvido em *software*. A metodologia utilizada na elaboração do projeto seguiu as fases especificadas na Figura 37.

A primeira etapa, a pesquisa bibliográfica, compreendeu a pesquisa por trabalhos na área de implementação de algoritmos relacionados ao processamento de imagens em *hardware*, assim como

novas idéias que contribuíssem com o desenvolvimento do projeto. Durante essa fase também foi definido o escopo e o cronograma do trabalho.

A segunda etapa, responsável pela elaboração do protótipo do *Hardwire* em *software*, também compreendeu a criação de ferramentas de suporte (geradores de módulos em VHDL e interpretadores de números no formato de ponto-fixo, por exemplo) para a validação dos módulos em VHDL. A maior parte das funcionalidades almejadas no FPGA foi implementada primeiramente usando a linguagem Java, de forma que se pudesse escolher a melhor forma de organização do fluxo de processamento dos algoritmos e definir uma arquitetura eficiente para o projeto.

A terceira etapa compreendeu a implementação e verificação funcional dos componentes que fazem parte do projeto *Hardwire*. Nessa etapa, todos os algoritmos anteriormente implementados em *software* necessários ao módulo em *hardware* foram traduzidos para a linguagem VHDL.

A quarta etapa compreendeu a elaboração do relatório final deste Trabalho de Graduação, assim como a revisão do mesmo.

Em seqüência, uma série de testes foi realizada para checar a corretude do *Hardwire*, e com os resultados obtidos pôde-se elaborar uma lista de ajustes, que foram solucionados.

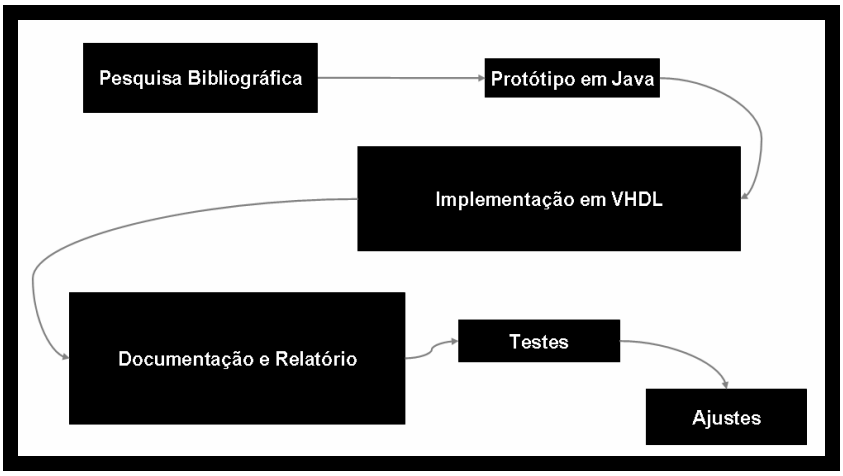

Figura 37. Metodologia utilizada.

As etapas da metodologia são recorrentes, ou seja, algumas atividades que ficaram pendentes foram resolvidas no decorrer do desenvolvimento de outras etapas.

4.3. Arquitetura

O mapeamento da funcionalidade implementada em *software* para as máquinas de estado em *hardware* implicou diretamente na divisão do *Hardwire* nas seguintes seções: criação dos *pixels* de entrada para o sistema (geração das entradas), transformação de representação em coordenadas mundiais para coordenadas de câmera, traçado das linhas que formam as arestas do objeto 3D e transformação das coordenadas de câmera para coordenadas de tela, como mostrado na Figura 38.

Todo o fluxo de processamento funciona de forma combinacional, dando a resposta logo que o sinal de entrada varia. Dessa forma, não foi necessário implementar um conjunto de registradores de borda (que estariam localizados nas fronteiras entre cada seção), de forma a guardar os dados produzidos por cada módulo, como ocorre num funcionamento em *pipeline*. O primeiro módulo, *input_gen* (descrito na Seção 4.6), além de gerar as entradas, funciona como controlador de todo o sistema, tendo conhecimento sobre quando as arestas do objeto terminam de ser desenhadas e sincronizando a alimentação de acordo com a máquina de estados de cada módulo subseqüente.

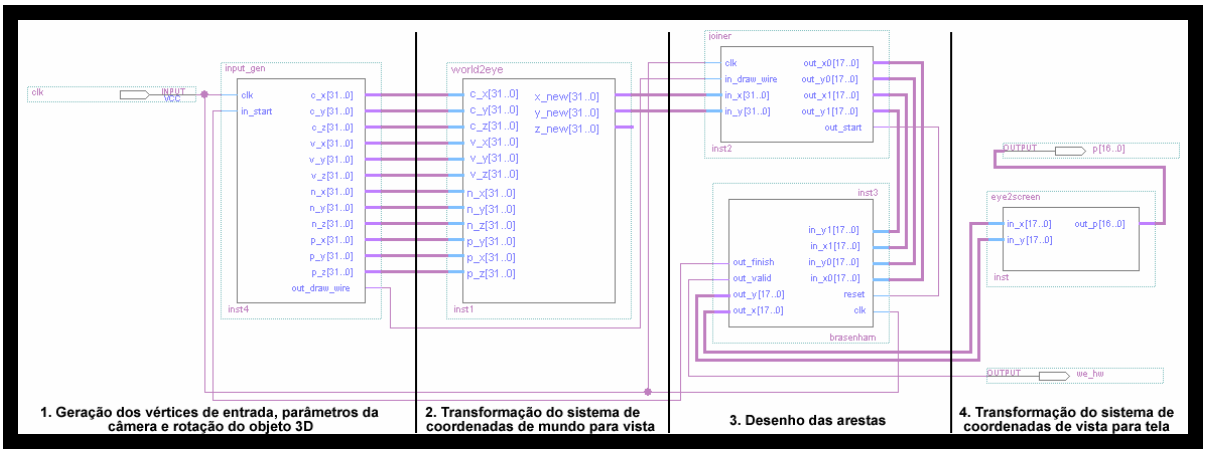

Figura 38. Arquitetura do *Hardwire*.

4.4. Ambiente de Desenvolvimento

4.4.1. *Hardware* Utilizado

A solução ARCam é composta por três dispositivos interconectados: sensor de imagem, FPGA e monitor VGA. Os dois primeiros são ligados diretamente, pino a pino com terra comum. Já entre o FPGA e o monitor de vídeo existe um DAC (*Digital-to-Analog Converter*) para converter as cores digitais em um nível de tensão analógico nos três canais de cores do monitor (vermelho, verde e azul). A utilização de um quarto dispositivo ainda está em fase de estudos: uma memória externa. O ideal seria restringir a quantidade de memória utilizada para no máximo os 2.4MB de blocos de memória internos ao FPGA, para evitar o uso de componentes adicionais.

Figura 39. Placa que acomoda o sensor de imagem.

O sensor de imagem, modelo Omnivison-OV7620, é formado por uma matriz de pontos sensíveis à luz visível colorida de 640x480 que lê 30 quadros por segundo. Uma placa de circuito impresso acomoda o *chip* sensor de imagem junto com a lente e um cristal para fornecer o *clock*, como exibido na Figura 39. Esta placa está sendo alimentada pela placa do FPGA e ligada fisicamente por um cabo IDE (*Integrated Drive Eletronics*) modificado no mapeamento dos pinos, uma vez que o barramento com os pinos do FPGA não possui somente pinos de propósito geral.

Os principais sinais fornecidos pelo sensor de imagem são:

- sincronismo horizontal: informa quando uma linha acaba de ser lida;
- sincronismo vertical: informa quando um quadro é concluído;

- sincronismo de *pixel*: informa que o valor do *pixel* está disponível no barramento com suas cores;
- paridade do quadro: informa se a linha é par ou impar;
- barramento Y: contém 8 *bits* para representar uma cor;
- barramento UV: contém 8 *bits* para representar duas cores. Este barramento é compartilhado, e em cada pulso de *clock* é disponibilizada uma cor.

O sensor de imagem possui um banco de registradores internos que possibilita a configuração de um grande número de parâmetros de operação, tais como resolução, número de quadros por segundo, modo entrelaçado e não entrelaçado, espelho da imagem, padrão de cores, exposição, gama, brilho, etc. Os registradores são configurados via protocolo I2C.

No ARCam, o processamento dos quadros provenientes do sensor de imagem será realizado, inicialmente, apenas por um FPGA da família Stratix II da Altera. O modelo selecionado é o EPS260, composto por 60.440 elementos lógicos (LEs (*Logical Elements*)), 2.544.192 *bits* de memória RAM, 36 blocos para processamento de sinais e 144 multiplicadores embarcados, necessários nas aplicações de processamento de matrizes (imagem). Esses dados são apresentados na Tabela 1.

Tabela 1
Características do Stratix II EP2S60.

ALMs	24,176
Adaptive look-up tables (ALUTs)	48,352
Equivalent Les	60,440
M512 RAM *blocks*	329
M4K RAM *blocks*	255
M-RAM *blocks*	2
Total RAM *bits*	2,544,192
DSP *blocks*	36
18-*bit* x 18-*bit multipliers*	144
Enhanced PLLs (*Phase Locked Loops*)	4
Fast PLLs	8
User I/O *pins*	492

O *hardware* do *kit* de desenvolvimento utilizado para este projeto é exibido na Figura 40. Vale ressaltar que, por ser um *kit*, este traz um grande número de periféricos externos para outras finalidades que não fazem parte do sistema proposto neste Trabalho de Graduação. Isso merece destaque devido ao fato do ARCam visar ser um sistema leve, pequeno, de baixo consumo e custo.

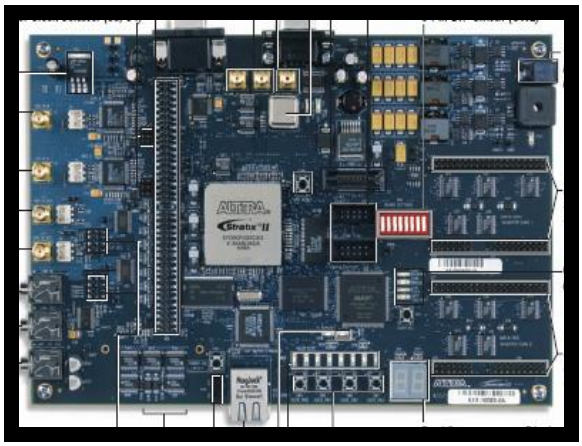

Figura 40. Placa de prototipação.

O subsistema de saída VGA da placa de prototipação, cujo modelo é o FMS3818KRC Triple Video D/A Converter, é responsável por exibir a imagem do mundo real "aumentada" com as informações virtuais agregadas no subsistema de processamento. A visualização pode ser feita através de qualquer dispositivo de saída que possa ser conectado à saída VGA da placa de prototipação. Na maior parte dos testes realizados baseados no estado atual do projeto, foram utilizados um monitor DELL CRT de 15 polegadas e outro, HP, também CRT, de 19 polegadas.

4.4.2. NIOS

O NIOS é um processador de propósito geral criado para ser embarcado nos FPGAs da Altera. Ele pode ser configurado (suporte a diversos tipos de periféricos podem ser adicionados ou retirados) e suporta aplicações escritas na linguagem C e compiladas para o mesmo. Através dele é possível também acessar o barramento de dados (Avalon), assim como os outros componentes externos a ele, implementados em VHDL. O símbolo do processador, assim como algumas de suas entradas e saídas, são mostrados na Figura 41.

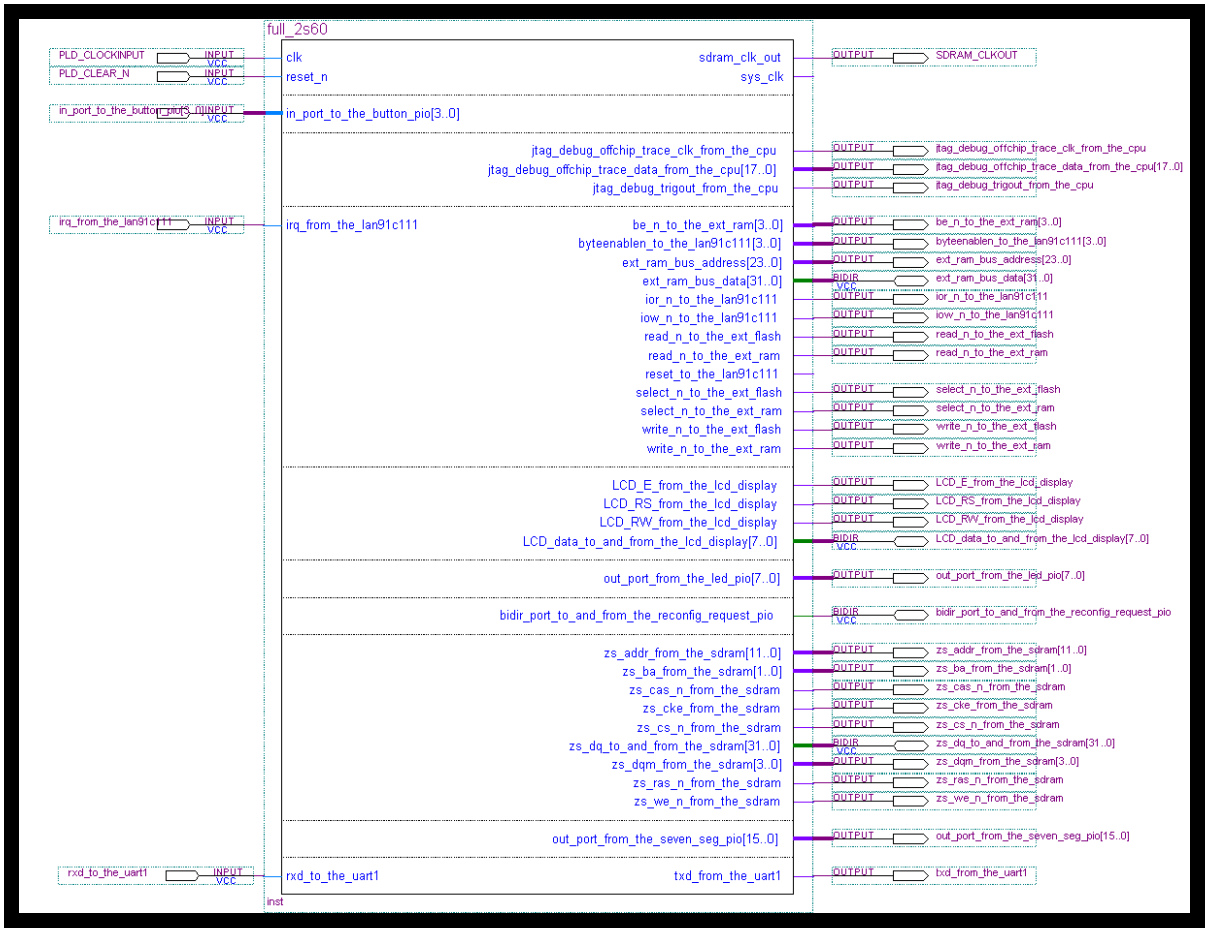

Figura 41. Bloco esquemático do NIOS, com algumas de suas entradas e saídas.

O ideal seria utilizar o processador apenas como coordenador dos dispositivos, de forma que o processamento real ocorresse no *hardware* dedicado desenvolvido. Após os primeiros testes, tendo em vista as dificuldades encontradas na integração NIOS/memória externa e os resultados obtidos (velocidade de acesso à memória através do NIOS), além dessa ser uma solução mista (*hardware* + *software*), optou-se por não utilizar o processador. Então, todo o processamento e comunicação serão realizados exclusivamente pelos módulos criados, sendo esta uma solução completamente implementada em *hardware*. O comportamento que antes seria realizado pelo código em C que executaria sobre o processador NIOS, agora será dado em função da junção dos blocos de *hardware* que estão em desenvolvimento.

4.4.3. VHDL

VHDL é uma linguagem de descrição de sistemas eletrônicos digitais [31]. Ela surgiu a partir de um programa do governo norte americano chamado de "*Very High Speed Integrated Circuits* (VHSIC), iniciado em 1980. No início do programa, ficou evidente a necessidade de uma linguagem padrão para descrição da estrutura e funcionalidade de circuitos integrados. Logo, a VHSIC *Hardware Description Language* foi desenvolvida, e subseqüentemente adotada como um padrão pelo Instituto de Engenheiros Elétricos e Eletrônicos (IEEE) nos Estados Unidos.

A linguagem foi criada com o objetivo de suprir uma série de necessidades do processo de desenvolvimento de *hardware*. Primeiramente, ela permite descrever a estrutura do *design*, ou seja, como o módulo será decomposto em submódulos e como estes estão conectados entre si. Em segundo lugar, ela permite a especificação das funcionalidades dos módulos através do uso de métodos familiares de programação. Por último, ela permite que um sistema seja simulado antes do mesmo ser fabricado, de forma que os desenvolvedores possam rapidamente comparar alternativas e testá-las em relação à corretude sem o tempo de espera e alto custo provenientes da prototipação em *hardware*.

Por se tratar de um sistema completamente desenvolvido em *hardware*, todos os módulos do *Hardwire* foram implementados usando a linguagem VHDL. Através dela, foi possível simular o comportamento funcional de cada sub-módulo criado e verificar sua corretude, antes de realizar os testes diretamente no FPGA. As simulações facilitaram a visualização dos resultados obtidos a partir do processamento isolado de cada módulo, característica essa difícil de ser analisada quando o sistema está rodando no FPGA programado. Para a simulação do sistema diretamente na placa de prototipação, a ferramenta SignalTap Logic Analyser foi utilizada. Através dela, é possível adquirir os valores de sinais específicos, enquanto o sistema está em funcionamento. A capacidade de armazenamento desses valores está restrita a capacidade dos blocos de memória interna da placa, e dessa forma não é possível visualizar uma grande quantidade de sinais ao mesmo tempo (o que não acontece na simulação via *software*).

4.4.4. Java e C

As linguagens de programação Java e C foram utilizadas no desenvolvimento de aplicações de suporte para teste e geração dos módulos em VHDL.Uma vez que a proposta do *Hardwire* é uma implementação completamente em *hardware*, o uso de uma linguagem de alto nível na fase de testes e como suporte não compromete o desempenho do sistema. Java, em específico, foi a linguagem escolhida para ser utilizada no desenvolvimento do protótipo em *software* do *Hardwire*. A aplicação criada aplicava transformações de visualização sucessivas em coordenadas de pontos e retornava a posição final em coordenadas de tela para todos eles.

Após o protótipo atingir o comportamento esperado, seu código foi reformulado com o objetivo de facilitar a tradução entre *software* desenvolvido e VHDL sintetizável. Foram realizados mapeamentos diretos entre trechos do código escrito em Java e módulos e conexões em *hardware*. A primeira grande tradução foi o mapeamento do tipo *double* em Java para o formato de ponto-fixo utilizado no *Hardwire*. Após isso, os valores das coordenadas dos vértices do objeto 3D, declarados no início do código, foram inseridos dentro do módulo *input_gen*, que serviu como gerador de entradas e controle do processamento do *Hardwire*. Todos os métodos implementados em Java foram mapeados em módulos, ou seja, para cada um existia um módulo em VHDL correspondente, com o mesmo tipo e número de entradas e saídas. A conexão entre os módulos seguiu a seqüência do processamento realizado no código em Java. O resultado gerado por cada módulo servia de entrada para os módulos subseqüentes. Todo o mapeamento realizado pode ser visto na Figura 42.

Além do protótipo em *software* (Java), foi criada uma aplicação responsável por gerar o módulo *angle*. Esse módulo funciona como um gerador de senos e cossenos em *hardware*, a partir de uma entrada específica. O programa recebia o tamanho da tabela de senos/cossenos como entrada e o arquivo *.vhd* era gerado como saída, conforme ilustrado na Figura 43.

A linguagem C foi utilizada na verificação dos resultados das simulações realizadas. Uma aplicação foi criada com o objetivo de traduzir os números da representação binária de ponto-fixo para seu valor decimal

correspondente. Dessa forma, foi possível comparar os valores de saída da simulação dos módulos em VHDL com os resultantes dos métodos implementados em Java, conforme mostrado na Figura 44.

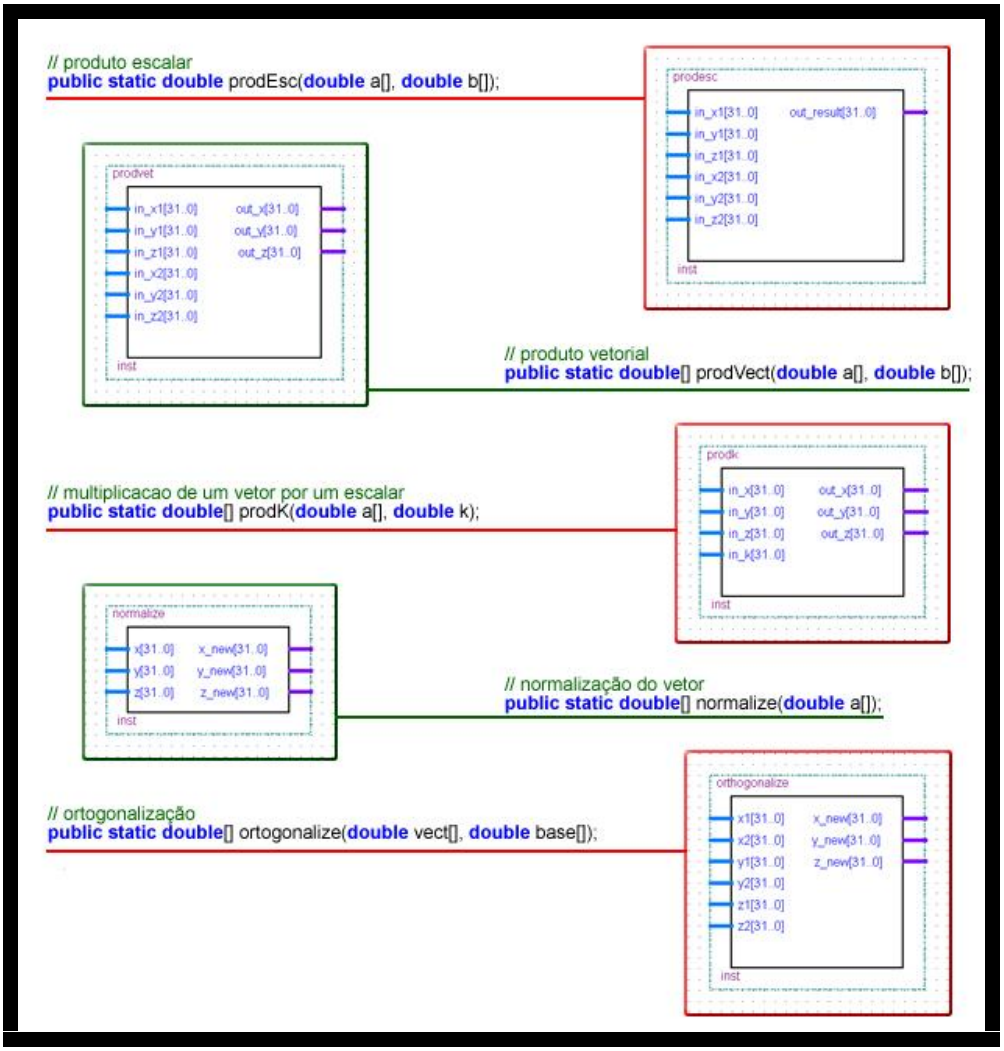

Figura 42. Mapeamentos realizados da linguagem Java para VHDL (implementação em *hardware*).

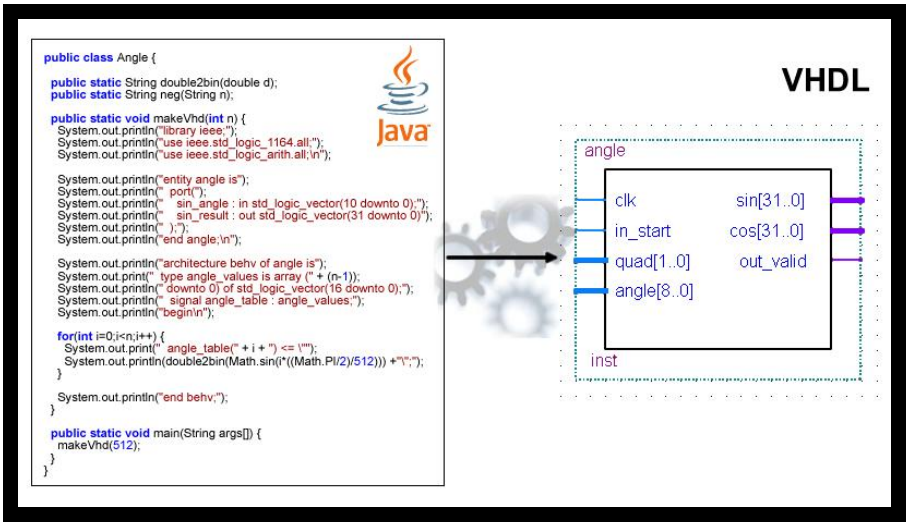

Figura 43. Geração do módulo *angle* a partir da aplicação em Java.

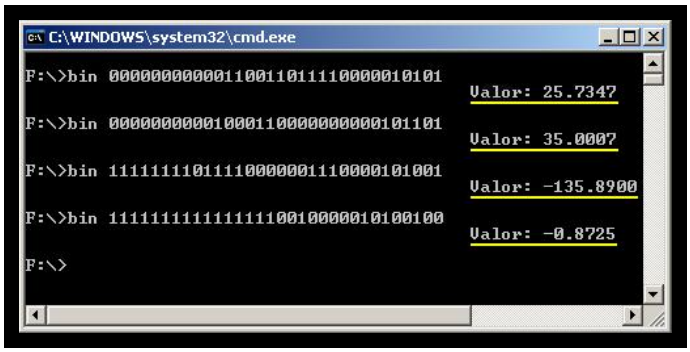

Figura 44. Tradução do formato de ponto-fixo para valor correspondente em decimal.

4.5. Tipo de Formato de Dados

Esta seção visa comparar dois tipos de formato numérico que foram analisados antes da implementação do *Hardwire*. Através da comparação de vantagens e desvantagens encontradas em ambos, optou-se pela implementação dos módulos utilizando a notação de ponto-fixo. A justificativa para uso dessa notação pode ser encontrada a seguir.

4.5.1. Ponto-Flutuante

Ponto-flutuante é uma maneira de se representar números reais através de dígitos ou *bits* em um computador ou calculadora, da mesma maneira que a notação científica é utilizada para representar valores com exatidão [32]. Um número na notação de ponto-flutuante é geralmente armazenado como três partes:

- Um significante (indicando os dígitos que definem a magnitude do número);
- Um expoente ou escala (que indica a posição do ponto de separação entre a parte inteira e a parte fracionária);
- Um sinal (que indica quando o número é positivo ou negativo).

A computação de ponto-flutuante possui um papel importante em uma variedade enorme de aplicações científicas, de engenharia e da indústria. A capacidade de realizar operações de ponto-flutuante é uma importante medida de desempenho para computadores destinados a esses tipos de aplicações. O valor dessa capacidade é medido em "FLOPS" (*Floating-point Operations Per Second*).

Números em ponto-flutuante são destinados a representar o modelo matemático dos números reais. Todavia, enquanto os números reais formam uma continuidade que pode ser subdividida sem limites, números na notação de ponto-flutuante apresentam uma resolução finita – eles apenas podem representar pontos discretos de uma reta numérica real. Com a representação em precisão dupla (*double precision*), pontos consecutivos diferem de 1 em 10^{16}. Ou seja, eles só podem representar um subconjunto dos números reais. Por causa disso, um número na notação de ponto-flutuante é, às vezes, considerado uma aproximação de um número real, ou a representação do número real com alguma tolerância, o que não é correto. Um número na notação de ponto-flutuante (isto é, uma *string* de *bits* armazenada em um computador) representa exatamente um número real. O valor pode não ser o número real pretendido de acordo com a situação, caso ele não esteja no subconjunto representável pelos *bits* disponíveis.

Uma representação de ponto-flutuante requer, antes de tudo, a escolha de uma base (*b*), assim como do número de *bits* (precisão - *p*) que formarão o significante. O significante (característica + mantissa) é um número que consiste de *p* dígitos na base *b*, de forma que cada dígito varia entre 0 e *b* – 1. A base 2 (ou seja, representação binária) é utilizada pela grande totalidade dos computadores.

Uma das vantagens da notação científica é que números muito pequenos (ou muito grandes) podem ser representados sem a necessidade de *strings* gigantescas com inúmeros zeros em seu começo ou fim, cuja contagem pode ser facilmente confundida. A notação científica determina uma posição específica para o

ponto – logo depois do primeiro dígito diferente de zero, e o expoente é responsável por determinar o valor do número.

Algumas pessoas (e algumas representações de computador) optam por uma convenção diferente para a localização presumida do ponto, como na parte esquerda ou no *bit* mais à esquerda do número. Isto simplesmente adiciona um endereço constante ao expoente. Quando um número em ponto-flutuante é normalizado, o seu dígito mais à esquerda deve ser não-nulo. Dessa forma, o valor zero não pode ser representado em uma notação normalizada de ponto-flutuante, uma vez que não se teria nenhum dígito diferente de zero na parte mais à esquerda do número. Com base nisso, ou o número zero não seria representado (poderia possivelmente ser aproximado através da representação de um valor suficientemente pequeno), ou seria representado violando as regras estabelecidas, e tratado como um padrão reconhecido e manipulado de forma especial. Por exemplo, a notação científica para o valor zero é simplesmente 0, enquanto que no IBM1620 (um computador decimal) ela aparece como *e* = -99 (expoente) e *s* = 0000000... (significante) – o significante não é normalizado, e o valor máximo do expoente negativo facilita a operação do processamento aritmético do *hardware*.

O valor matemático de um ponto-flutuante é geralmente *s.sssssssssss...sss* x *be*. Quando se trabalha com a base binária, o significante é uma *string* de *bits* (1s e 0s) de tamanho *p*, dos quais o *bit* mais à esquerda possui valor 1. Quando o número em ponto-flutuante não corresponde exatamente ao valor do número real desejado, realiza-se uma aproximação de forma a encontrar o valor mais próximo do número real em questão.

Com o objetivo de representar um ponto-flutuante como um dado armazenável em computador, o expoente deve ser codificado em um campo de *bits*. Uma vez que o expoente pode ser negativo, poderia ser usada a representação de complemento a dois. Ao invés disso, uma constante fixa é adicionada ao expoente, com a intenção de tornar o resultado um número positivo capaz de ser compactado em um campo de *bits* de tamanho fixo. Para a representação comum de 32 *bits* (*single precision*), ou formato *float* definido como um dos padrões da IEEE, essa constante possui o valor 127, de forma que o expoente é dito como representado no formato de "127 em excesso". O resultado dessa adição é armazenado num espaço de 8 *bits*. Como o significante mais à esquerda de um número em ponto-flutuante (normalizado) é sempre 1, esse *bit* não precisa ser armazenado. O *hardware* do computador funciona como se o valor 1 que foi suprimido tivesse sido fornecido. Esse é o "*bit* implícito" ou "*bit* escondido" definido no padrão da IEEE, que permite que um campo de 23 *bits* represente um significante de 24 *bits*, mas também implica que o número zero não possa ser representado com todos os 23 *bits* iguais a zero, já que seria interpretado como um significante iniciado por 1. Uma codificação especial é requerida para o valor zero. Por último, um *bit* de sinal é necessário. Este é definido com o valor 1 para indicar que todo o número em formato de ponto-flutuante é negativo, ou 0 para indicar que o mesmo é positivo.

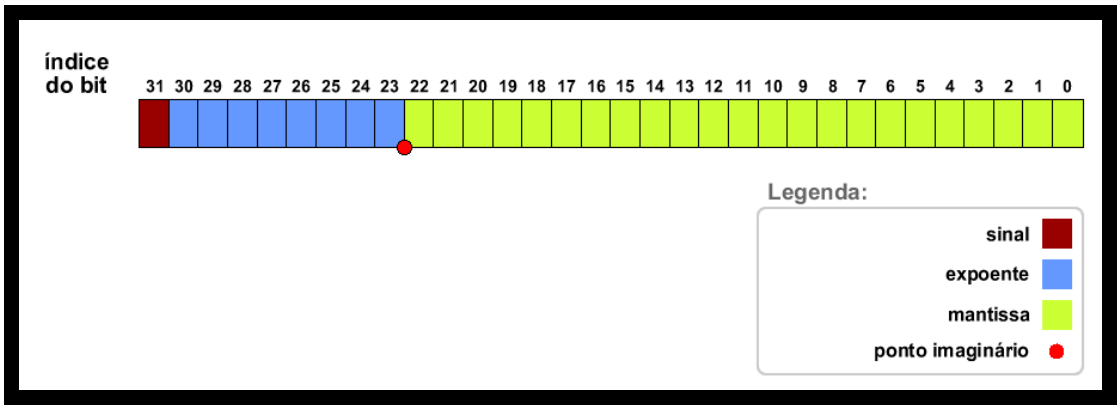

Figura 45. Formato de ponto-flutuante de 32 *bits*.

No passado, alguns computadores utilizavam um tipo de codificação em complemento a dois para todo o número, ao invés do formato sinal/magnitude. O número em ponto-flutuante é armazenado por completo em uma palavra de 32 *bits*, com o sinal localizado na parte mais à esquerda, seguido do expoente no formato de "127 em excesso" nos próximos 8 *bits*, e posteriormente o significante (sem o "*bit* implícito") nos 23 *bits* mais à direita. Para a representação comum de 64 *bits* (*double precision*), ou formato *double* foi

definido como um dos padrões da IEEE, o deslocamento adicionado ao expoente possui valor 1023, e o resultado é posto em um campo de 11 *bits*. A precisão é composta por 53 *bits*. Após a remoção do "*bit* implícito", restam 52 *bits*. O resultado compreende *1 + 11 + 52 = 64 bits*. Ambos os formatos (32 e 64 *bits*) são mostrados na Figura 45 e na Figura 46.

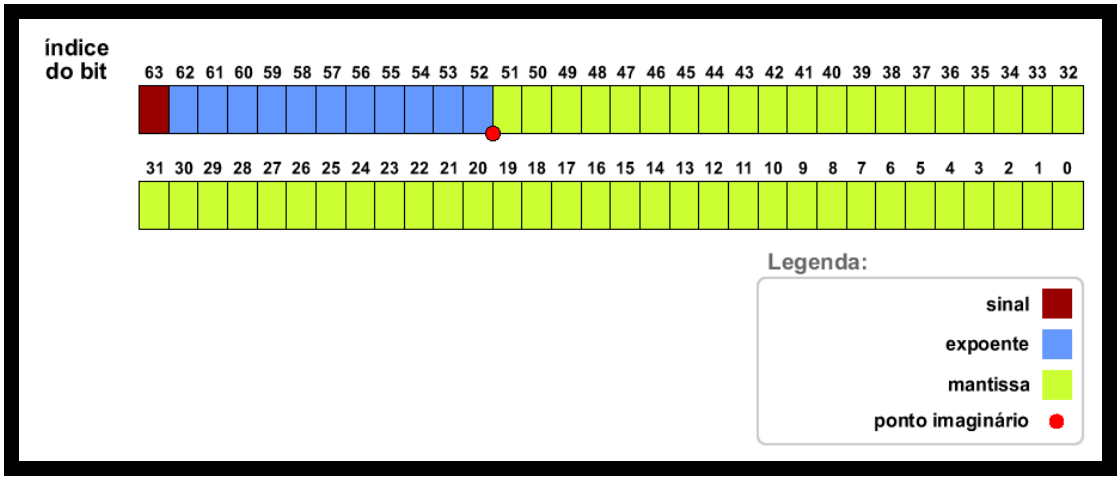

Figura 46. Formato de ponto-flutuante de 64 *bits*.

A necessidade de compactar o expoente em um campo de *bits* de tamanho fixo impõe limites sobre o expoente. Para o padrão de 32 *bits*, e^{+127} deve ocupar um espaço de 8 *bits*, de forma que $-127 \leq e \leq 128$. Os valores -127 e 128 são reservados para representações especiais, de forma que o valor de *e* varia entre -126 e 127. Isso significa que o menor número positivo normalizado possui o valor de $e = -126$; $s = 10000000000000000000000$, com o valor aproximado de 1.18×10^{-38}, e é representado em hexadecimal por 00800000. O maior número representável possui o valor de $e = 127$; $s = 11111111111111111111111$, com o valor aproximado de 3.4×10^{38}, e é representado em hexadecimal por 7F7FFFFF. No caso da precisão dupla (64 *bits*), os valores variam de 2.2×10^{-308} até 1.8×10^{308}.

Quando há um processamento sobre um número no formato de ponto-flutuante que origina um valor (após realizado o arredondamento necessário) acima do limite superior, diz-se que houve um caso de *overflow*. De acordo com o padrão da IEEE, o resultado é modificado para um valor especial que corresponde a "infinito", o qual possui um *bit* de sinal apropriado, o expoente reservado com valor de +128 e um padrão de *bits* na parte significante (tipicamente zero) indicando o "infinito". Esses números geralmente são impressos como "+INF" ou "-INF". *Hardwares* de manipulação numérica no formato de ponto-flutuante são desenvolvidos para suportar operações com a representação de "infinito" de acordo com o esperado, como por exemplo, (+INF) + (+7) = (+INF) e (+INF) x (-2) = (-INF).

Quando há um processamento sobre um número no formato de ponto-flutuante que origina um valor (após realizado o arredondamento necessário) não-nulo abaixo do limite inferior, diz-se que houve um caso de *underflow*. De acordo com o padrão definido pela IEEE, o expoente com o valor reservado de -127 é utilizado e o valor varia conforme descrito a seguir. Caso o valor do número seja zero, ele é representado por um expoente com valor -127 e todo o significante com valor 0. Isso significa que o valor 0 é representado em hexadecimal por 00000000. Caso o valor apresente expoente menor que -126, o significante é denormalizado, de forma a representar valores menores sem a notação de "*bit* implícito". Desta maneira, por exemplo, o valor aproximado de 7.3×10^{-40} seria representado por 0 00000000 00010000000000000000000, que corresponde a 00080000 em hexadecimal. A criação de números denormalizados é geralmente chamada de *underflow* gradual. À medida que os números ficam extremamente pequenos, os *bits* do significante são gradualmente sacrificados. Uma alternativa é o "*underflow* repentino", no qual qualquer número que não pode ser normalizado é simplesmente transformado em zero pelo *hardware*. Existe um alto grau de complexidade na manipulação de *underflow* gradual em *hardware*. Geralmente é usado *software* como suporte, através de interrupções. Isso acarreta uma perda de desempenho considerável, e nesses casos o "*underflow* repentino" deve ser implementado.

O comportamento padrão do *hardware* dos computadores é aproximar o resultado ideal (infinitamente

preciso) de uma operação aritmética para o valor representável mais próximo, e fornecer essa representação como resultado. Na prática, existem outras opções. *Hardwares* compatíveis com a especificação IEEE-754 permitem a escolha de um dos seguintes tipos de arredondamento:

- aproximar para o valor mais próximo (decisão padrão, utilizada na maior parte das vezes);
- arredondar para cima (até o limite de máximo infinito; valores negativos tendem a zero);
- arredondar para baixo (até o limite de mínimo infinito);
- arredondar para zero (funciona de forma similar às conversões de *float* para inteiro, que transformam, por exemplo, -3.9 para -3).

O padrão de arredondamento especificado pela IEEE 754 indica o uso da primeira opção (valor mais próximo) em todas as operações algébricas fundamentais, incluindo a operação de raiz quadrada. O uso não é obrigatório em funções de bibliotecas como seno e cosseno. Isto significa que o comportamento do *hardware* é completamente determinado para situações com 32 e 64 *bits*. O comportamento indicado para tratamento de *overflow* e *underflow* é que o resultado apropriado seja computado, levando em consideração o tipo de arredondamento, como se o intervalo (*range*) do expoente fosse infinitamente grande. Se o resultado do expoente não puder ser compactado no campo de *bits* corretamente, o *overflow*/*underflow* é tratado conforme descrito anteriormente.

A distância aritmética entre dois números representados em notação de ponto-flutuante é chamada de "ULP" (*Unit in the Last Place*). Por exemplo, os números representados por 45670123 e 45670124 em hexadecimal estão distantes em 1 ULP. Essa medida corresponde a aproximadamente 10^{-7} em 32 *bits* (*single precision*), e 10^{-16} em 64 *bits* (*double precision*). A IEEE especifica que o resultado de qualquer operação esteja distante do resultado ideal em no máximo metade de uma ULP.

Em adição ao valor especial de "infinito" que é produzido na ocorrência de um *overflow*, também existe um outro valor especial chamado de "NaN" (*Not a Number*), que é produzido por uma operação de raiz quadrada de número negativo, por exemplo. O valor NaN é codificado com o expoente reservado de 128 (ou 1024), e um significante que o diferencie do valor especial de infinito. O objetivo das representações INF e NaN é que, em circunstâncias normais, esses valores podem ser propagados para as próximas operações (qualquer operação com um valor NaN retorna como resultado NaN), e o resultado só precisa ser tratado quando for conveniente.

Além da criação de valores especiais, existem "eventos" que podem acontecer:

- um *overflow* ocorre, conforme descrito anteriormente, produzindo um infinito;
- um *underflow* ocorre, conforme descrito anteriormente, produzindo uma denormalização;
- uma divisão por zero ocorre quando o divisor da operação possui valor zero, produzindo um infinito com o sinal apropriado (apesar do sinal do zero não fazer diferença significativa). Destaca-se que um valor muito pequeno, porém diferente de zero, pode causar um *overflow* e resultar em infinito;
- um erro de operando acontece quando um NaN é criado. Isso ocorre quando um dos operandos é um NaN, ou qualquer outro fator dá origem ao erro, como é o caso da radiciação ou *log* de números negativos;
- um evento inexato acontece quando o arredondamento mudou o resultado do valor matemático verdadeiro. Todas essas exceções são postas em máscaras (não habilitadas). Algumas vezes *overflow*, divisão por zero e erro de operando estão habilitadas.

Uma vez que os números no formato de ponto-flutuante não representam exatamente os números reais, e as operações entre eles não representam o comportamento exato das operações com números reais, existem muitos problemas que surgem quando se escreve *software* que utiliza o formato de ponto-flutuante. Enquanto a adição e a multiplicação são ambas operações comutativas ($a + b = b + a$ e $a \times b = b \times a$), elas

não são associativas. De acordo com a ordem que as operações são realizadas, o arredondamento do ponto-flutuante pode ocasionar diferenças nos resultados. As operações também não são distributivas, pelo mesmo problema de aproximação. O arredondamento é realizado ao fim de cada operação algébrica, o que leva a imprecisões que podem acarretar resultados inesperados.

Além da perda de precisão, a incapacidade de representar exatamente números como *Π* (pi) ou 0.1, e outras pequenas imprecisões, os seguintes fenômenos podem acontecer:

- cancelamento pela subtração de operandos com valores aproximadamente iguais, podendo causar uma imprecisão grande. Esse é talvez o mais comum e sério problema de precisão;

- conversões ineficazes para inteiros, como por exemplo a conversão de (63.0/9.0) para inteiro resultando em 7, enquanto a conversão de (0.63/0.09) resultando em 6. Isso acontece porque operações de conversão geralmente truncam o número, ao invés de arredondá-lo;

- tamanho do expoente limitado, onde os resultados podem gerar *overflow*, tendendo para o infinito;

- o teste de divisão por zero é problemático, pois checar se o divisor é diferente de zero não garante que a divisão não irá gerar *overflow* ou infinito como resultado;

- comparação de igualdade entre números, onde programadores geralmente realizam comparações com níveis de tolerância, mas isso não necessariamente resolve o problema.

Por causa dos problemas citados acima, o uso ingênuo de números no formato de ponto-flutuante pode levar a diversos problemas. Um bom entendimento de análise numérica é essencial para a criação de *softwares* robustos de manipulação de ponto-flutuante. Além do cuidado que se deve tomar na criação dos programas, também é necessário cuidado ao se projetar o compilador que será utilizado. Algumas otimizações realizadas pelos compiladores (por exemplo, reordenação de instruções) podem ir contra o comportamento esperado do *software*.

O maior potencial da aritmética de ponto-flutuante é aproveitado quando ela é utilizada para medir quantidades do mundo real baseado em uma grande quantidade de escalas (como por exemplo, o período orbital de um planeta ou a massa de um próton), e seu pior funcionamento ocorre quando se pretende modelar interações de quantidades expressas como *strings* decimais que devem ser exatas. Como exemplo desse último caso têm-se os cálculos financeiros. Por esse motivo, *softwares* financeiros tendem a não utilizar a representação numérica binária de ponto-flutuante. O tipo "*decimal*", de linguagens como C# e Java, assim como o padrão IEEE 854, foi criado como solução para os problemas que existem na notação de ponto-flutuante binária e faz com que a aritmética sempre se comporte conforme esperado, com os números impressos em decimal.

O formato de 64 *bits* (*double precision*) é mais preciso do que qualquer medida física que pode ser realizada. Por exemplo, ele pode ser usado para indicar a distância da Terra até a Lua, com uma precisão de 50 nanômetros. O que torna a aritmética de ponto-flutuante problemática é o fato de ser realizada uma enorme quantidade de operações, responsáveis por fazer os erros mínimos de precisão crescerem. Alguns exemplos são operações de inversão de matrizes, vetores característicos e ainda resolução de equações diferenciais. Esses algoritmos devem ser bem projetados para que funcionem corretamente.

As principais vantagens de algumas propriedades da notação de ponto-flutuante são:

- qualquer número inteiro estritamente inferior a 224 pode ser exatamente representado na notação de 32 *bits* (*single precision*), assim como qualquer número estritamente inferior a 253 pode ser exatamente representado em 64 *bits* (*double precision*). Além do mais, o resultado da multiplicação de qualquer potência de 2 por um número com as características citadas acima pode também ser representado. Essa propriedade é muitas vezes utilizada em aplicações puramente inteiras, com o objetivo de se conseguir inteiros de 53 *bits* em máquinas que não possuem o formato de ponto-flutuante de 64 *bits*, mas apenas inteiros com 32 *bits*;

- a representação dos *bits* é monotônica, contanto que os valores especiais sejam evitados e os

sinais sejam manipulados cuidadosamente. Números no formato de ponto-flutuante são considerados iguais se e somente se as suas representações inteiras são iguais. Comparações de maior e menor podem ser realizadas através de comparações inteiras aplicadas ao conjunto de *bits*, desde que os sinais dos valores sejam iguais. Todavia, as comparações entre números no formato de ponto-flutuante atuais apresentam tipicamente mais sofisticação no tratamento de valores especiais (como INF e NaN, por exemplo);

- fazendo uma aproximação, tem-se que a representação em *bits* de um número da notação de ponto-flutuante é proporcional ao seu logaritmo na base 2, com um erro médio de 3% (isso pelo fato do expoente estar na parte mais significante dos *bits*). Este fato pode ser explorado em algumas aplicações, como em operações com volume em processamento de som digital.

A IEEE padronizou a representação binária de ponto-flutuante para computadores com o IEEE 754. Esse padrão é seguido por praticamente todas as máquinas atuais. O padrão permite que se use diferentes níveis de precisão, dos quais os de 32 e 64 *bits* são os mais comuns, uma vez que são suportados por linguagens de programação convencionais. Alguns *hardwares* (por exemplo, da série Pentium e Motorola 68000) também fornecem um formato de precisão estendido com 80 *bits*, sendo 15 *bits* para expoente e 64 para significante, sem *bit* implícito.

4.5.2. Ponto-Fixo

Em computação, o formato de ponto-fixo representa um tipo de número real que possui um número fixo de casas decimais antes e depois do ponto decimal [33]. Números na notação de ponto-fixo são bastante úteis para representar valores fracionários nativamente em complemento a dois, caso o processador não possua uma unidade de processamento de ponto-flutuante (FPU – *Floating-Point Unit*), ou se o formato de ponto-fixo oferecer maior desempenho ou precisão. Grande parte dos processadores embarcados de baixo custo não possui uma FPU.

Os *bits* à esquerda do ponto decimal representam a parte inteira (magnitude), enquanto os *bits* à direita do ponto representam a parte fracionária. Cada *bit* fracionário representa uma potência inversa de 2. Dessa forma, o primeiro *bit* possui o valor ½, o segundo ¼, o terceiro ⅛, e assim por diante. Para números no formato de ponto-fixo na notação de complemento a dois, o limite superior é dado por $2^{m-1} - \frac{1}{2^f}$, enquanto o limite inferior é dado por -2^{m-1}, onde *m* representa o número de *bits* da parte inteira e *f* representa o número de *bits* da parte fracionária. A assimetria entre limites superior e inferior ocorre devido à notação de complemento a dois. Por exemplo, um número binário de 16 *bits* com sinal no formato de ponto-fixo, com 4 *bits* após o ponto decimal, apresenta 12 *bits* de magnitude e 4 *bits* fracionários. Ele pode representar números entre 2047.9375 e -2048. Um número binário de 16 *bits* sem sinal no formato de ponto-fixo com 4 *bits* fracionários varia entre 4095.9375 e 0. Números em ponto-fixo conseguem representar potências fracionais de 2 com exatidão, mas, assim como números em ponto-flutuante, não conseguem representar exatamente potências fracionárias de 10. Se potências exatas de 10 forem desejadas, o formato BCD (*Binary-Coded Decimal*) deve ser utilizado. Todavia, BCD não torna eficiente o uso dos *bits* como faz a notação de complemento a dois, e nem é computacionalmente rápido.

Por exemplo, um décimo (0,1) e um centésimo (0,01) apenas podem ser representados aproximadamente na notação de ponto-fixo em complemento a dois ou representação de ponto-flutuante, enquanto a representação BCD consegue armazená-los com exatidão.

Valores em ponto-fixo são sempre representados exatamente contanto que eles estejam na faixa determinada pelos *bits* de magnitude. Essa é uma das diferenças para o formato de representação de ponto-flutuante, que possui campos de valor e expoente separados, o que permite especificar valores inteiros maiores do que o número de *bits* no campo significante, causando ao valor representado perda de precisão.

Um uso comum para números no formato de ponto-fixo BCD é o armazenamento de valores monetários, tendo em vista que nesse caso os valores inexatos dos números no formato de ponto-flutuante são sempre um problema. Historicamente, representações de ponto-fixo eram o padrão para tipos de dados decimais

(por exemplo, em PL/I ou COBOL). A linguagem de programação ADA inclui suporte para ambas as representações em ponto-fixo (ordinário e decimal) e ponto-flutuante. A linguagem de programação JOVIAL também fornece tipos de dados em ambos os formatos.

Poucas linguagens de programação incluem suporte para valores em ponto-fixo, porque na maior parte das aplicações as representações em ponto-flutuante são rápidas e precisas o suficiente. A representação de ponto-flutuante é considerada mais "fácil" para o programador do que a representação de ponto-fixo, uma vez que ela pode lidar com uma variação dinâmica maior e não requer que os programadores especifiquem o número de dígitos existentes após o ponto decimal.

Todavia, caso haja necessidade, os números no formato de ponto-fixo podem ser implementados até mesmo em linguagens como C e C++, que não incluem tal suporte. Com a publicação da ISO/IEC TR 18037:2004, tipos de dados no formato de ponto-fixo foram especificados para a linguagem de programação C.

Existem várias notações usadas para representar o tamanho da palavra e o ponto decimal em um número no formato de ponto-fixo. Uma notação comum é o prefixo "Q", onde o número que segue o Q especifica o número de *bits* fracionários à direita do ponto decimal. Por exemplo, Q15 representa um número com 15 *bits* fracionários. Essa notação é ambígua, uma vez que ela não especifica o tamanho da palavra, apesar de geralmente se assumir 16 ou 32 *bits*, dependendo do processador a ser utilizado. A forma não ambígua para a notação "Q" é representada por um Q seguido de um par de números, separados por um ponto, que indicam o número de *bits* de magnitude à esquerda do ponto decimal, seguido pelo número de *bits* fracionários. Por exemplo, Q1.15 descreve um número com 1 *bit* de magnitude e 15 *bits* fracionários em uma palavra de 16 *bits*. Outra notação, o prefixo "fx", funciona de forma similar e usa o comprimento da palavra como segundo item do par numérico. Por exemplo, fx1.16 descreve um número com 1 *bit* de magnitude e 15 *bits* fracionários, em uma palavra de 16 *bits*. O formato de ponto-fixo adotado no desenvolvimento do *Hardwire* é mostrado na Figura 47.

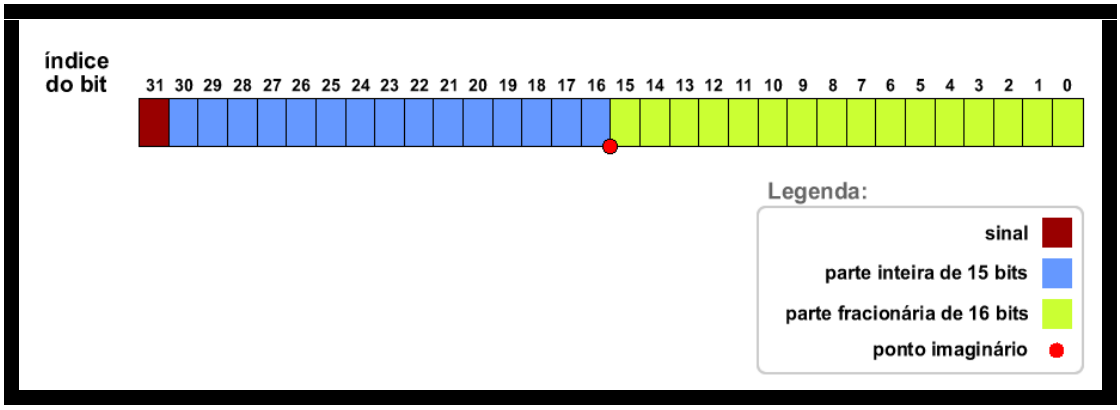

Figura 47. Formato de ponto-fixo (32 *bits*) adotado no desenvolvimento do *Hardwire*.

Uma vez que operações com números no formato de ponto-fixo podem produzir resultados que possuam mais *bits* do que os operandos envolvidos, existe a possibilidade de perda de informação. Como exemplo, o resultado de uma multiplicação de números no formato de ponto-fixo pode potencialmente ter tantos *bits* quanto a soma do número de *bits* dos operandos. Com o objetivo de tornar o resultado representável com o mesmo número de *bits* dos operandos, o resultado da operação deve ser arredondado ou truncado. Se este for o caso, a escolha de quais *bits* devem ser mantidos é muito importante. Quando se multiplica dois números com o mesmo formato de ponto-fixo, por exemplo com I *bits* inteiros, e Q *bits* fracionários, a resposta pode possuir até 2*I *bits* inteiros, e 2*Q *bits* fracionários.

Por simplicidade, muitos programadores usam o mesmo formato dos operandos para o resultado. Como conseqüência, devem-se manter os *bits* intermediários; I *bits* para a parte inteira menos significativa, e Q *bits* para a parte fracionária mais significativa. Os *bits* fracionários que são perdidos representam uma perda de precisão que é comum em multiplicação de números fracionários. Se for perdido algum número diferente de zero na parte inteira, o valor estará radicalmente impreciso. Esse caso é considerado um *overflow*, e deve ser evitado em cálculos embarcados.

Recomenda-se utilizar um modelo baseado em simulação de operadores como o VisSim, para detectar e evitar tais *overflows* com o uso de uma configuração apropriada do tamanho da palavra do resultado e da posição do ponto decimal, ganhos escalares adequados e limitação de magnitude de valores intermediários. Algumas operações, como a divisão, geralmente utilizam limitadores no resultado, fazendo com que qualquer *overflow* positivo resulte no maior número possível que pode ser representado no formato corrente. De forma similar, resultados com *overflow* negativo resultam na maior representação negativa suportada pelo formato adotado. Esse limitante é geralmente chamado de saturação. Alguns processadores suportam um *flag* de *overflow* que pode gerar uma interrupção na ocorrência de um *overflow*, mas é geralmente tarde para recuperar os resultados apropriados a essa altura.

4.6. Módulos Utilizados

A seguir, todos os módulos em hardware que foram utilizados na implementação do *Hardwire* são descritos. Uma pequena parte desses módulos é fornecida pela Altera, enquanto a maioria teve que ser desenvolvida com base no fluxo implementado no protótipo em *sotfware*.

4.6.1. Módulos Fornecidos

Esta subseção descreve o uso das megafunções LPM (*Library of Parameterized Modules*) (biblioteca de módulos parametrizáveis) que foram aproveitadas, uma vez que já são fornecidas pela ferramenta Quartus, da Altera.

adder32

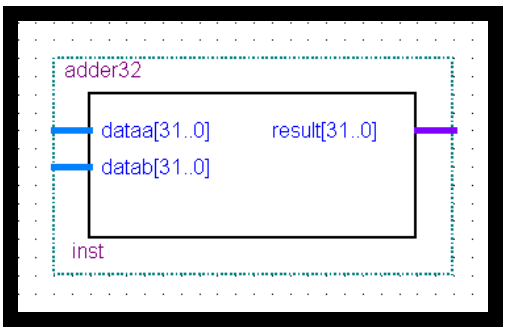

Figura 48. Símbolo que representa o módulo *adder32*.

Esse módulo, cujo símbolo é mostrado na Figura 48, faz uso do componente *lpm_add_sub*, e foi criado a partir dos seguintes parâmetros principais:

Parâmetro	Valor	Descrição
lpm_direction	"ADD"	Indica que será utilizada exclusivamente a função de soma.
lpm_width	32	Número de *bits* referente aos dados manipulados pelo módulo.

Todo o circuito fornecido é combinacional, não sendo necessária uma entrada para sinal de *clock*. Esse módulo realiza operação de soma entre dois números inteiros.

mult32

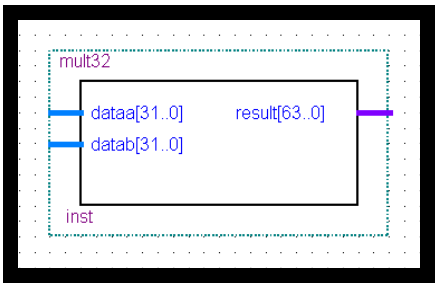

Figura 49. Símbolo que representa o módulo *mult32*.

Esse módulo, cujo símbolo é mostrado na Figura 49, faz uso do componente *lpm_mult*, e foi criado a partir dos seguintes parâmetros principais:

Parâmetro	Valor	Descrição
lpm_hint	"DEDICATED_MULTIPLIER_CIRCUITRY=YES" "MAXIMIZE_SPEED=5"	Indica que o módulo utilizará circuito dedicado da placa para realizar a multiplicação. O parâmetro de velocidade indica que o módulo deve ser otimizado para dar a resposta de forma combinacional.
lpm_representation	"SIGNED"	Parâmetro indica que serão manipulados números com sinal.
lpm_widtha	32	Número de *bits* pertencentes à primeira entrada do módulo.
lpm_widthb	32	Número de *bits* pertencentes à segunda entrada do módulo.
lpm_widthp	64	Tamanho da saída.

Este módulo também é combinacional e realiza operação de multiplicação entre dois números no formato inteiro com sinal.

div32

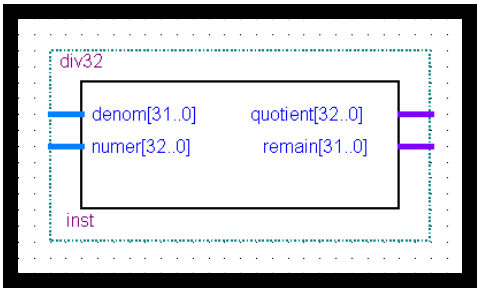

Figura 50. Símbolo que representa o módulo *div32*.

Esse módulo, cujo símbolo é mostrado na Figura 50, faz uso do componente *lpm_divide*, e foi criado a partir dos seguintes parâmetros principais:

Parâmetro	Valor	Descrição
lpm_drepresentation	"SIGNED"	Indica que se está trabalhando com números com sinal no denominador.
lpm_hint	"LPM_REMAINDER POSITIVE=TRUE"	Indica que o resto da divisão deve apresentar sinal positivo.
lpm_nrepresentation	"UNSIGNED"	Indica que se está trabalhando com números sem sinal no numerador.
lpm_widthd	32	Número de *bits* do denominador.
lpm_widthn	33	Número de *bits* do numerador.

A escolha do número de *bits* do numerador e denominador, assim como o formato (com ou sem sinal) serão justificados na descrição do módulo *inv32_inst*.

sqrt32

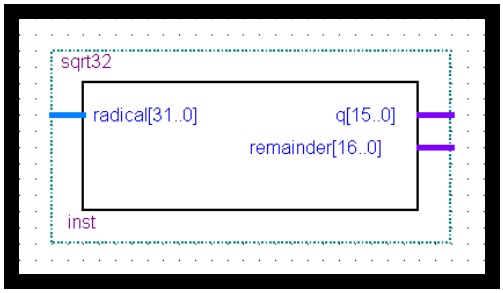

Figura 51. Símbolo que representa o módulo *sqrt32*.

Esse módulo, cujo símbolo é mostrado na Figura 51, faz uso do componente *altsqrt*, e foi criado a partir dos seguintes parâmetros principais:

Parâmetro	Valor	Descrição
pipeline	0	Indica que o módulo não apresenta níveis de *pipeline*, ou seja, a implementação do mesmo é puramente combinacional.
q_port_width	16	Indica que a saída do módulo que representa o resultado contém o total de 16 *bits*.
r_port_width	17	Indica que a saída do módulo que representa o resto contém o total de 17 *bits*.
width	32	Indica que o número de entrada possui 32 *bits*.

Este módulo realiza operação de radiciação de números inteiros.

square32

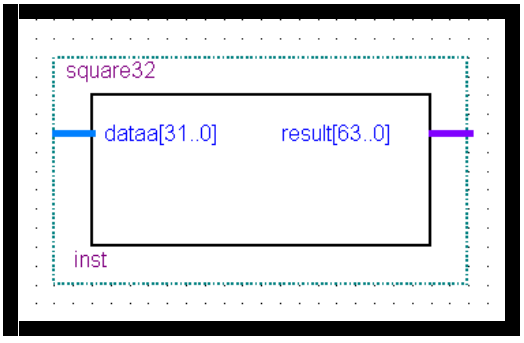

Figura 52. Símbolo que representa o módulo *square32*.

Esse módulo, cujo símbolo é mostrado na Figura 52, faz uso do componente *altsquare*, e foi criado a partir dos seguintes parâmetros principais:

Parâmetro	Valor	Descrição
data_width	32	Indica que o valor de entrada está representado através de 32 *bits*.
pipeline	0	Indica que o módulo não apresenta níveis de *pipeline*, ou seja, a implementação do mesmo é puramente combinacional.
representation	"SIGNED"	Indica que o módulo deve trabalhar com números com sinal.
result_width	64	Indica que a saída resultante do módulo deve possuir 64 *bits*.

Este módulo realiza operação de quadrado de números inteiros.

sub32

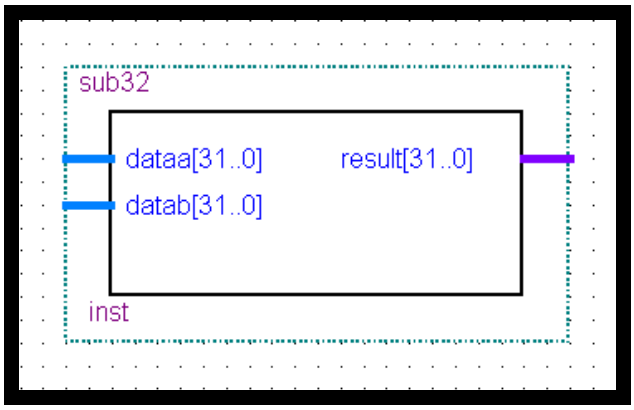

Figura 53. Símbolo que representa o módulo *sub32*.

Esse módulo, cujo símbolo é mostrado na Figura 53, faz uso do componente *lpm_add_sub*, e foi criado a partir dos seguintes parâmetros principais:

Parâmetro	**Valor**	**Descrição**
lpm_direction	"SUB"	Indica que será utilizada exclusivamente a função de subtração.
lpm_width	32	Número de *bits* referente aos dados manipulados pelo módulo.

Todo o circuito fornecido é combinacional, não sendo necessária uma entrada para sinal de *clock*. Esse módulo realiza operação de subtração entre dois números inteiros.

4.6.2. Módulos Implementados

A seguir serão apresentados todos os módulos que foram implementados, mostrando detalhes de suas respectivas funcionalidades, entradas e saídas.

adder32_inst

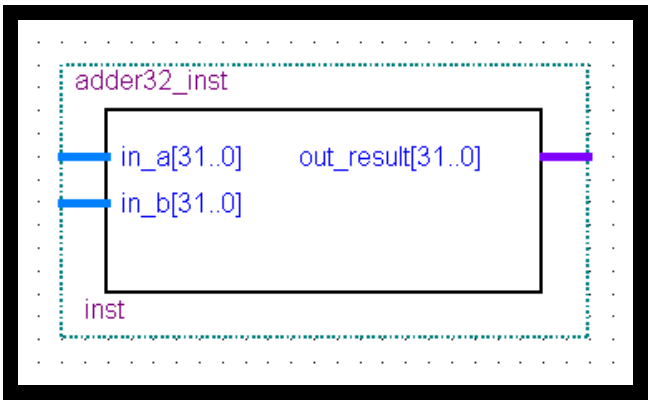

Figura 54. Símbolo que representa o módulo *adder32_inst*.

Este módulo, cujo símbolo é mostrado na Figura 54, funciona como um mapeamento do módulo *adder32*. Optou-se pelo uso dessa "interface" pelo fato de tornar independente qual módulo de soma está sendo utilizado. Conforme visto na seção que detalha as características comuns entre números inteiros e números no formato de ponto-fixo, a operação de soma sobre números inteiros funciona de forma semelhante à operação de soma no formato decimal utilizado. Portanto, esse módulo somente repassa os valores de entrada e saída para o módulo mais interno, responsável por realizar a operação. A Figura 55, a Figura 56, a Figura 57 e a Figura 58 ilustram alguns exemplos do uso deste módulo.

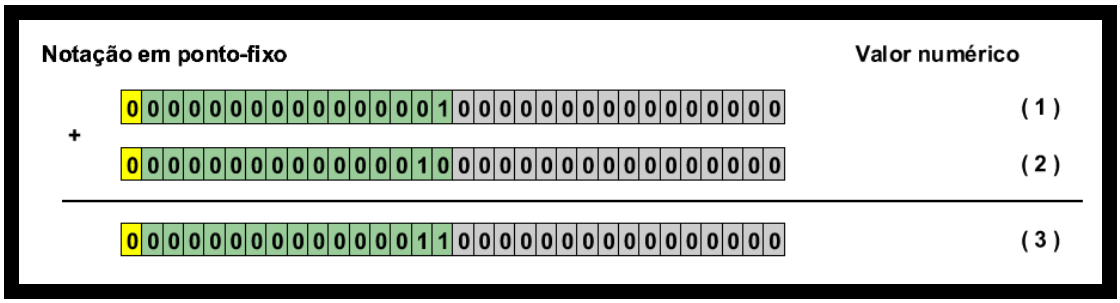

Figura 55. Exemplo de uso do módulo: 1 + 2.

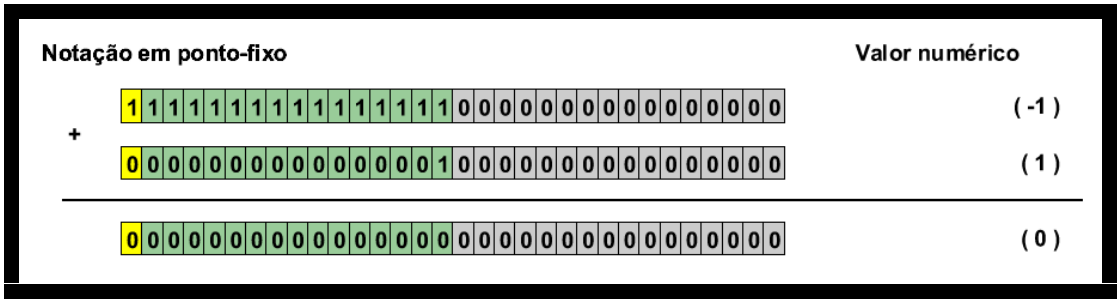

Figura 56. Exemplo de uso do módulo: -1 + 1.

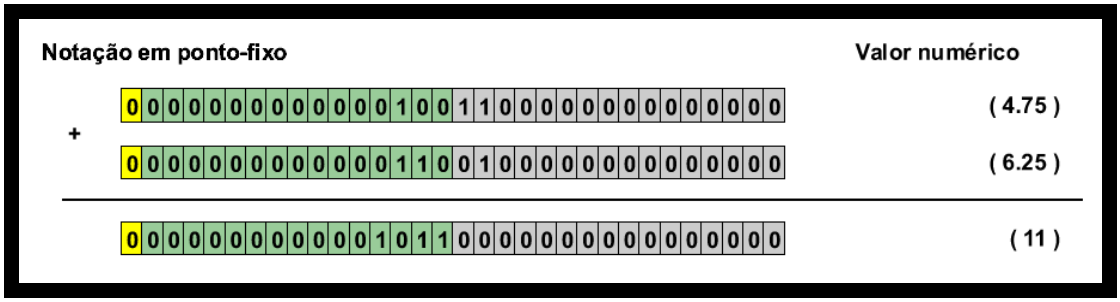

Figura 57. Exemplo de uso do módulo: 4.75 + 6.25.

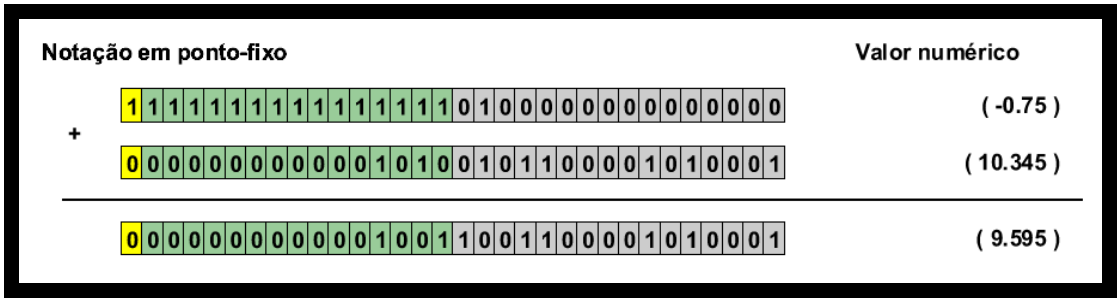

Figura 58. Exemplo de uso do módulo: -0.75 + 10.345.

Descrição dos pinos de entrada e saída:

Nome	Tamanho	Descrição
Entrada		
In_a	32 *bits*	Parâmetro que representa um dos dois valores a serem somados.
In_b	32 *bits*	Parâmetro que representa um dos dois valores a serem somados.
Saída		
out_result	32 *bits*	Saída contendo o valor do resultado da operação de soma realizada.

angle

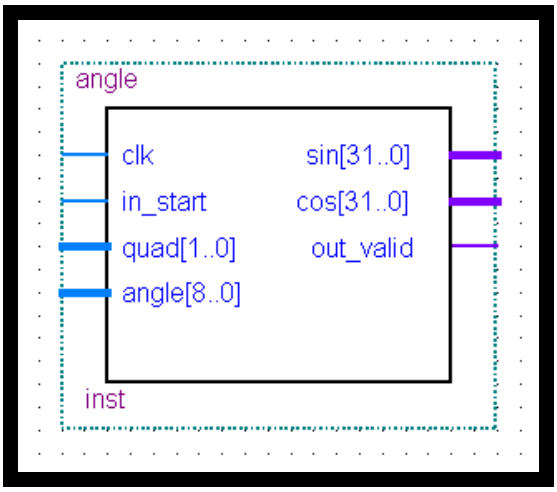

Figura 59. Símbolo que representa o módulo *angle*.

Esse módulo, cujo símbolo é mostrado na Figura 59, foi criado a partir da necessidade de adquirir valores de seno e cosseno de ângulos variados, os quais seriam utilizados para realizar a rotação do objeto dentro do módulo *input_gen*. Foram pesquisadas implementações já existentes e algoritmos específicos para uso em FPGA, como por exemplo o CORDIC [34]. Esse algoritmo consegue calcular em tempo real o valor do seno correspondente de determinado ângulo. Devido à complexidade do algoritmo, optou-se por implementar um módulo que contém uma tabela dos valores que serão retornados como saída. A tabela de 512 entradas é responsável pelo mapeamento dos valores de seno compreendidos no intervalo fechado que vai de 0 a 90 graus. Os valores de seno nos outros intervalos, assim como os valores de cosseno do ângulo são criados dentro do módulo a partir de operações de complemento e simetria angular.

Descrição dos pinos de entrada e saída:

Nome	**Tamanho**	**Descrição**
Entrada		
Clk	1 *bit*	Utilizado como sinal de *clock* do módulo.
in_start	1 *bit*	Essa entrada indica que o módulo pode começar a realizar os cálculos sobre os valores *quad* e *angle* de entrada para encontrar seno e cosseno correspondentes.
Quad	2 *bits*	Indica a que quadrante pertence o ângulo. Exemplo: 00 => Primeiro quadrante 01 => Segundo quadrante 10 => Terceiro quadrante 11 => Quarto quadrante
Angle	9 *bits*	Esse valor indica um ângulo proporcional ao intervalo compreendido entre 0 e 90 graus. Exemplo: 000000000 => 0 grau 010101011 => 30 graus 100000000 => 45 graus 110101011 => 75 graus
Saída		
sin	32 *bits*	Seno do ângulo passado como entrada, em formato ponto-fixo.
cos	32 *bits*	Cosseno do ângulo passado como entrada, em formato ponto-fixo.
out_valid	1 *bit*	Sinal que indica que as saídas são válidas (seno e cosseno já foram calculados com sucesso).

bresenham

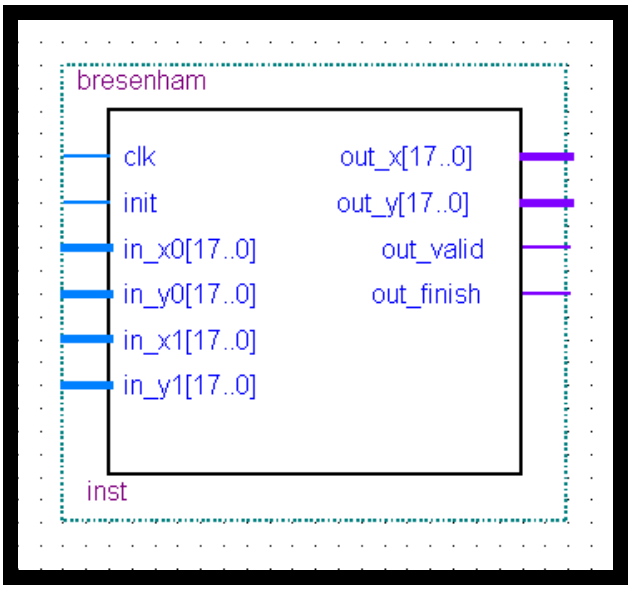

Figura 60. Símbolo que representa o módulo *bresenham*.

Este módulo, cujo símbolo é mostrado na Figura 60, baseou-se na implementação da versão otimizada para *hardware* do algoritmo de desenho de linhas de Bresenham [29]. Tal algoritmo, conforme mostrado na seção de conceitos básicos, apenas utiliza operações simples como somas, subtrações e deslocamento de *bits* (operações com baixo teor de complexidade de se implementar em *hardware*). Dessa forma, foi mapeado o pseudo-código mostrado na Seção 3.4 para VHDL, com algumas modificações que se fizeram necessárias. Como exemplo das modificações realizadas, pode-se citar o suporte a retas completamente na horizontal ou retas completamente na vertical, que são casos particulares onde os pontos coincidem com uma das coordenadas (*X* ou *Y*). O tempo de finalização do desenho da reta desejada (número de ciclos necessários) depende da distância das coordenadas dos dois pontos (origem e destino) fornecidos como parâmetro. Quanto mais próximos os pontos estiverem, menor será a reta e mais rapidamente será finalizada.

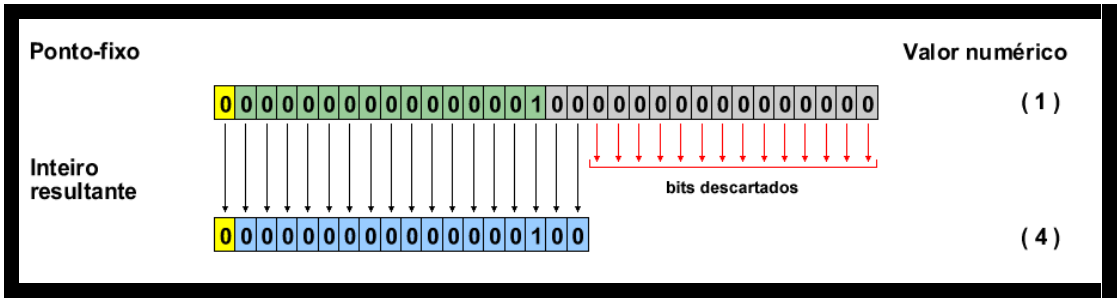

Figura 61. Transformação do formato ponto-fixo 32 *bits* para inteiro 18 *bits*.

O algoritmo de Bresenham funciona para desenho de retas entre números inteiros, ou seja, a linha é criada a partir de um deslocamento incremental com base nos *pixels* da tela. Portanto, decidiu-se que as coordenadas de entrada do módulo possuiriam 18 *bits* no total, ao invés dos 32 *bits* do formato de ponto-fixo. Esses 18 *bits* são originados a partir dos 16 *bits*, que representam a parte inteira do número em formato de ponto-fixo, juntamente com os dois primeiros *bits* que representam a parte fracionária do número. Sendo assim, um número que antes se encontrava na notação ponto-fixo é transformado para inteiro com uma operação de deslocamento de dois *bits* para esquerda. Todos os *bits* restantes da parte fracionária são desconsiderados. Poderiam-se considerar apenas os *bits* da parte inteira e desconsiderar a parte fracionária, mas por questões de precisão optou-se por considerar os dois primeiros *bits* fracionários. Essa transformação de ponto-fixo com 32 *bits* para número inteiro de 18 *bits* pode ser vista na Figura 61.

Descrição dos pinos de entrada e saída:

Nome	Tamanho	Descrição
Entrada		
clk	1 *bit*	Utilizado como sinal de *clock* do módulo.
init	1 *bit*	Esse *bit* é utilizado para indicar ao módulo quando deve começar o processamento do desenho das linhas. O módulo considera que as entradas dos dois pontos (origem e destino) já apresentam os valores que serão usados no cálculo da linha.
in_x0	18 *bits*	Parâmetro no formato inteiro com sinal que representa a coordenada *X* do primeiro ponto (origem).
in_y0	18 *bits*	Parâmetro no formato inteiro com sinal que representa a coordenada *Y* do primeiro ponto (origem).
in_x1	18 *bits*	Parâmetro no formato inteiro com sinal que representa a coordenada *X* do segundo ponto (destino).
in_y1	18 *bits*	Parâmetro no formato inteiro com sinal que representa a coordenada *Y* do segundo ponto (destino).
Saída		
out_x	18 *bits*	Coordenada *X* gerada no formato inteiro com sinal de um ponto que fará parte da linha gerada.
out_y	18 *bits*	Coordenada *Y* gerada no formato inteiro com sinal de um ponto que fará parte da linha gerada.
out_valid	1 *bit*	Esse *bit* indica se as coordenadas representadas pelas saídas *out_x* e *out_y* são válidas, ou seja, se elas podem ser escritas na memória de vídeo.
out_finish	1 *bit*	Esse *bit* indica para o mecanismo de controle de entradas (nesse caso, o módulo *input_gen*) que o desenho da linha foi concluído e que os *pixels* em seqüência podem ser processados.

Pode-se perceber que ambos os pontos passados como parâmetro de entrada só apresentam valores de coordenadas *X* e *Y*, ou seja, apenas duas dimensões. Isso é justificado pelo fato dos *pixels* já se encontrarem em coordenadas de tela, ou seja, uma vez que a tela só apresenta duas dimensões (largura e altura), esse valor já é resultado da projeção 3D realizada sobre a tela de visualização.

Os nomes "origem" e "destino" foram adotados na descrição desse módulo e de seus respectivos parâmetros apenas com o objetivo de facilitar o entendimento da aplicação, uma vez que o algoritmo de Bresenham, de acordo com os parâmetros de entrada, troca os pontos de posição (de acordo com a diferença das coordenadas *X* e *Y* dos pontos, o próprio algoritmo se encarrega de trocar origem e destino). Além disso, o algoritmo verifica em qual coordenada há um maior deslocamento e com base nessa informação pode realizar uma troca entre as coordenadas *X* e *Y* dos pontos. Nesse caso, a etapa final do desenho de linhas deve ser responsável por entender se ocorreu alguma troca e fornecer a saída correta do módulo.

eye2screen

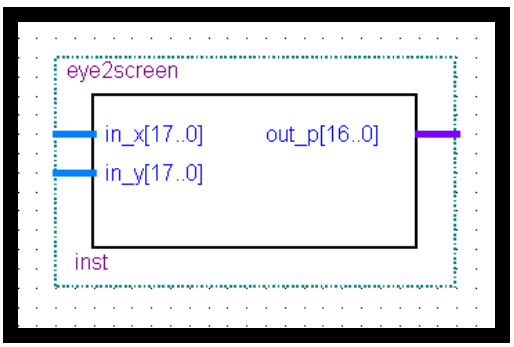

Figura 62. Símbolo que representa o módulo *eye2screen*.

Este módulo, cujo símbolo é mostrado na Figura 62, é responsável por realizar a transformação de coordenadas de vista para coordenadas de tela. Como a memória utilizada é representada por um *array* de apenas uma dimensão, que é mapeado em toda a extensão da tela, a saída desse módulo indica a posição de memória que representa o *pixel* a ser escrito na tela.

A Figura 63 ilustra o esquema de processamento que ocorre no interior do módulo. Inicialmente os valores de *X* e *Y* do parâmetro de entrada são transladados para a origem (centro da tela) e após esse cálculo o endereço linear da memória é encontrado.

A saída é calculada de acordo com a seguinte função: *out_p* = (160 + *in_x*) + (100 – *in_y*)*320.

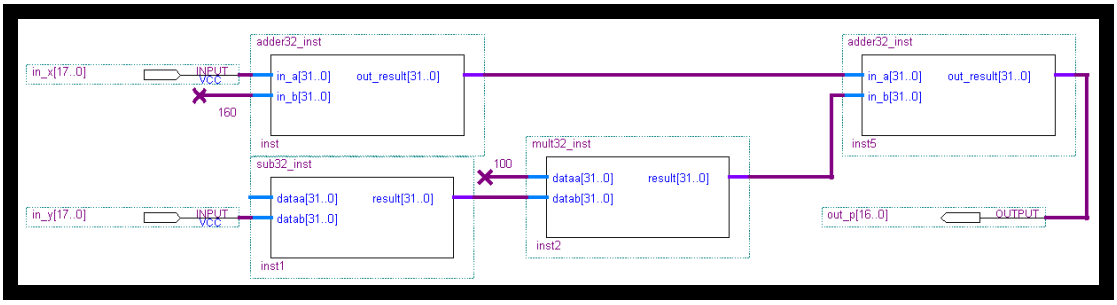

Figura 63. Esquemático do módulo *eye2screen*.

Descrição dos pinos de entrada e saída:

Nome	Tamanho	Descrição
Entrada		
in_x	18 *bits*	Parâmetro que corresponde à coordenada *X* do ponto, no formato de número inteiro com sinal.
in_y	18 *bits*	Parâmetro que corresponde à coordenada *Y* do ponto, no formato de número inteiro com sinal.
Saída		
out_p	18 *bits*	Valor de saída do módulo, representando o endereço de memória relativo aos parâmetros de entrada que foram fornecidos. Este valor de saída pode variar de 0 a 63999, referente à resolução de vídeo representada pela memória, que é de 320x200 *pixels*.

Esse módulo funciona de forma combinacional, ou seja, a saída é disponibilizada logo após a entrada ser fornecida. Dessa forma, não há necessidade de um sinal de *clock* presente no módulo.

hardwire

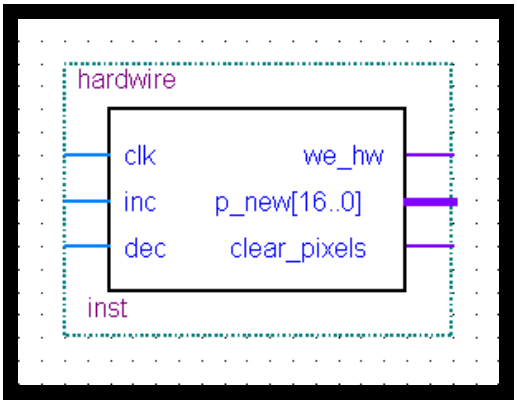

Figura 64. Símbolo que representa o módulo *hardwire*.

Este módulo, cujo símbolo é mostrado na Figura 64, encapsula todo o *Hardwire*, ou seja, representa a unidade *top* implementada. Como no momento as informações de entrada não estão sendo geradas em tempo real, pelos módulos externos ao *Hardwire* (parte do projeto ARCam), o módulo gera internamente tais parâmetros através do módulo *input_gen*. Todos os pinos presentes nesse encapsulamento são ligados aos módulos externos, e juntos dão origem à aplicação que mostra um cubo em 3D rotacionando na tela. As interligações podem ser observadas na Figura 65.

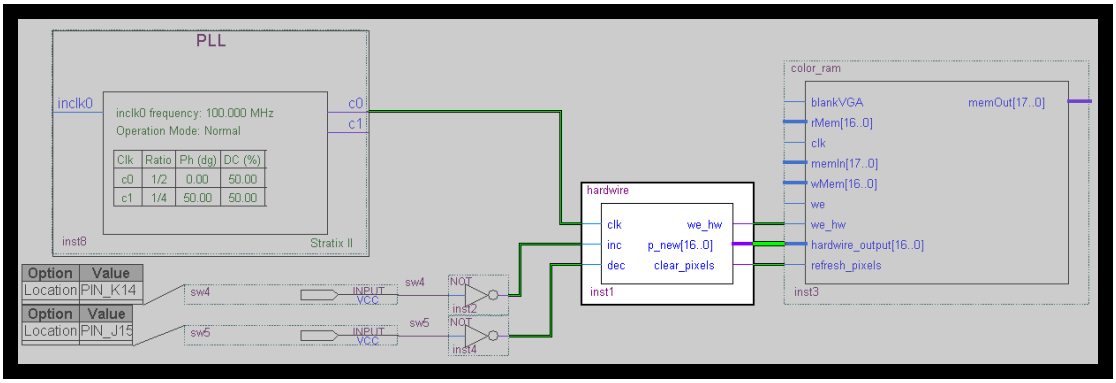

Figura 65. Interligações entre *Hardwire* e ARCam.

O *clock* de entrada é originado por um PLL do sistema, e as entradas *inc* e *dec* são mapeadas diretamente em dois botões distintos da placa. O *we_hw* funciona como um sinal de *write enable*, responsável por indicar à memória externa que o *pixel* pode ser escrito. O valor de *p_new* indica a posição do *pixel* na memória externa, enquanto que o sinal de saída *clear_pixels* indica que deve ocorrer um *refresh* na memória, ou seja, todos os *pixels* já escritos devem ser apagados. Quando um *refresh* acontece, a aplicação mostra apenas a entrada que vem da câmera, até que todos os *pixels* da memória sejam limpos. Uma vez terminada essa operação, os *pixels* do cubo projetado voltam a aparecer na tela.

Descrição dos pinos de entrada e saída:

Nome	**Tamanho**	**Descrição**
Entrada		
clk	1 *bit*	Utilizado como sinal de *clock* do módulo.
inc	1 *bit*	Esse parâmetro foi utilizado para fins de teste. Como exemplo, ele funcionava para controlar o sentido de rotação do cubo na tela, assim como parar/habilitar a animação, ou incrementando o ângulo de rotação do cubo.
dec	1 *bit*	Esse parâmetro também foi utilizado como controle externo para testes, assim como a entrada *inc*. Nos testes realizados geralmente apresentava a funcionalidade inversa à entrada anterior, decrementando o ângulo de rotação ao qual o cubo estava submetido, por exemplo.
Saída		
we_hw	1 *bit*	Esse sinal de saída funciona como um *write enable* para memória externa, indicando que deve ser escrito um *pixel* preto na memória, no endereço gerado também pelo módulo. Tal *pixel* fará parte de uma das arestas do objeto 3D projetado.
p_new	17 *bits*	Esse sinal de saída corresponde ao endereço linear da memória externa, onde deve ser colocado o *pixel* preto, referente à representação em *wireframe* do objeto 3D.
clear_pixels	1 *bit*	Esse sinal indica que a memória deve ser reiniciada, tendo todos os seus antigos valores atualizados. O objetivo desse sinal é limpar a memória para que se possa desenhar o objeto em outra posição, ou em outro ângulo de visão. Dessa forma, o desenho sucessivo de um objeto com pequenas variações dos parâmetros de entrada pode originar animações, como é o caso do exemplo do cubo em *wireframe* rotacionando no eixo *X*.

input_gen

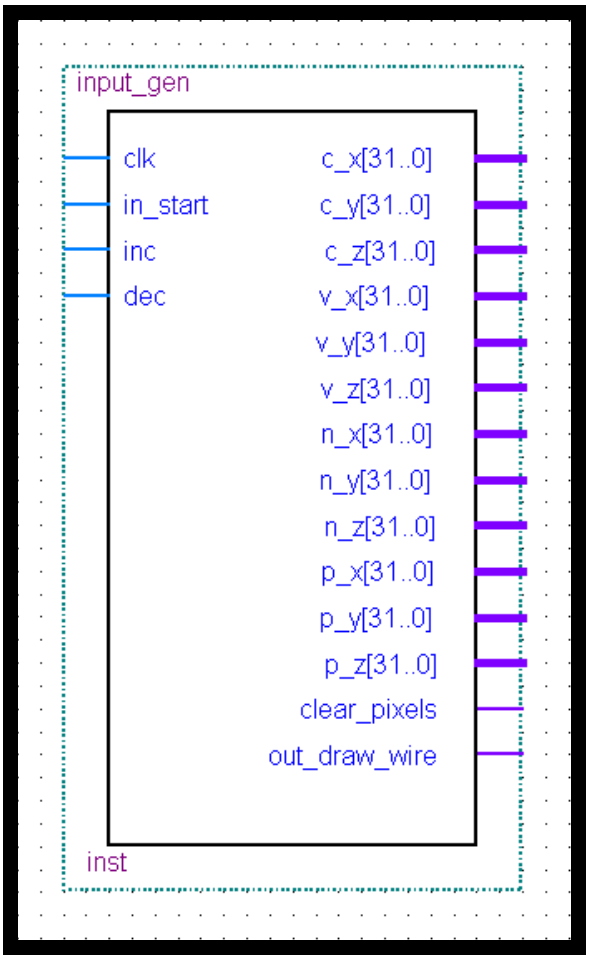

Figura 66. Símbolo que representa o módulo *input_gen*.

Esse módulo, cujo símbolo é mostrado na Figura 66, funciona como gerador de entradas interno ao *Hardwire*. As entradas geradas pelo módulo representam a estrutura do cubo mostrado na Figura 67. A idéia é que em sua versão final, ele passe a funcionar apenas como um controlador, recebendo as entradas como parâmetros externos do *Hardwire* e apenas repassando-as para os outros módulos em seqüência. Este módulo contém uma lista de *pixels*, com informações sobre suas respectivas coordenadas, assim como uma lista de arestas formada por eles, conforme ilustrado na Figura 68 e na Figura 69, respectivamente.

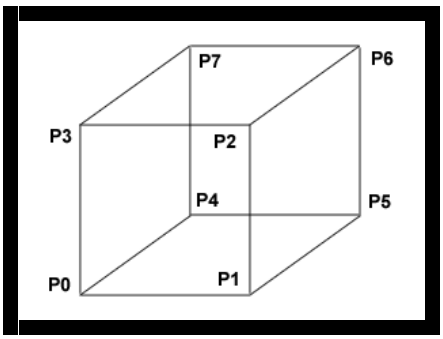

Figura 67. Estrutura do cubo 3D representado pelo módulo *input_gen*.

```
-- Lista com coordenadas
-- P0 : (1,-3,5)                                          -- P1 : (11,-3,5)
memx(0) <= "00000000000000010000000000000000";            memx(1) <= "00000000000001011000000000000000";
memy(0) <= "11111111111111010000000000000000";            memy(1) <= "11111111111111010000000000000000";
memz(0) <= "00000000000001010000000000000000";            memz(1) <= "00000000000001010000000000000000";

-- P2 : (11,7,5)                                          -- P3 : (1,7,5)
memx(2) <= "00000000000001011000000000000000";            memx(3) <= "00000000000000010000000000000000";
memy(2) <= "00000000000001110000000000000000";            memy(3) <= "00000000000001110000000000000000";
memz(2) <= "00000000000001010000000000000000";            memz(3) <= "00000000000001010000000000000000";

-- P4 : (1,-3,-5)                                         -- P5 : (11,-3,-5)
memx(4) <= "00000000000000010000000000000000";            memx(5) <= "00000000000001011000000000000000";
memy(4) <= "11111111111111010000000000000000";            memy(5) <= "11111111111111010000000000000000";
memz(4) <= "11111111111111011000000000000000";            memz(5) <= "11111111111111011000000000000000";

-- P6 : (11,7,-5)                                         -- P7 : (1,7,-5)
memx(6) <= "00000000000001011000000000000000";            memx(7) <= "00000000000000010000000000000000";
memy(6) <= "00000000000001110000000000000000";            memy(7) <= "00000000000001110000000000000000";
memz(6) <= "11111111111111011000000000000000";            memz(7) <= "11111111111111011000000000000000";
```

Figura 68. Lista com as coordenadas dos vértices do objeto 3D.

```
-- Lista com as arestas
-- 0 <-> 1          -- 1 <-> 2          -- 2 <-> 3          -- 3 <-> 0
wire1(0) <= 0;      wire1(1) <= 1;      wire1(2) <= 2;      wire1(3) <= 3;
wire2(0) <= 1;      wire2(1) <= 2;      wire2(2) <= 3;      wire2(3) <= 0;

-- 4 <-> 5          -- 5 <-> 6          -- 6 <-> 7          -- 7 <-> 4
wire1(4) <= 4;      wire1(5) <= 5;      wire1(6) <= 6;      wire1(7) <= 7;
wire2(4) <= 5;      wire2(5) <= 6;      wire2(6) <= 7;      wire2(7) <= 4;

-- 0 <-> 4          -- 3 <-> 7          -- 1 <-> 5          -- 2 <-> 6
wire1(8) <= 0;      wire1(9) <= 3;      wire1(10) <= 1;     wire1(11) <= 2;
wire2(8) <= 4;      wire2(9) <= 7;      wire2(10) <= 5;     wire2(11) <= 6;
```

Figura 69. Lista com as arestas do objeto 3D.

Além de gerar as entradas do objeto 3D a ser visualizado, este módulo é responsável por sincronizar a maior parte dos sinais de controle dos módulos seqüenciais internos ao *Hardwire*. A máquina de estados presente neste módulo tem conhecimento quando uma aresta acabou de ser desenhada, quando deve iniciar o desenho de uma nova aresta, quais *pixels* carregar de acordo com a seqüência de arestas armazenada e quando deve realizar uma operação de *refresh* na memória externa. Além disso, as operações de rotação sobre os pontos do objeto 3D acontecem no interior desse módulo, conforme visto na Figura 70.

```
when "0111" =>                          when "1000" =>

    ycos <= temp_y * usedCos;               roguesquad1 <= ycos + zsin;
    zsin <= temp_z * usedSin;               roguesquad2 <= zcos - ysin;
    zcos <= temp_z * usedCos;
    ysin <= temp_y * usedSin;               state <= "1001";

    state <= "1000";
```

Figura 70. Trecho de código responsável pela rotação 3D.

Os valores dos senos e cossenos dos ângulos gerados como entrada são obtidos a partir de uma instância do módulo *angle*, descrito anteriormente. As coordenadas de saída deste módulo representam, respectivamente, a posição da câmera no espaço, o vetor de direção da câmera virtual (para que direção a câmera aponta), um vetor normal à direção da câmera, assim como a posição do ponto pertencente ao objeto. Atualmente, apenas os valores relativos ao posicionamento dos objetos no mundo virtual são modificados; conseqüentemente, os demais parâmetros são todos constantes. Com as informações relativas à câmera como parâmetro de saída, é possível realizar operações de mudança de posição da câmera, assim como mudar a direção para onde a mesma está apontando.

Descrição dos pinos de entrada e saída:

Nome	Tamanho	Descrição
Entrada		
clk	1 *bit*	Utilizado como sinal de *clock* do módulo.
in_start	1 *bit*	Esse parâmetro indica que uma aresta acabou de ser desenhada, permitindo que se dê início ao processo de desenho da próxima aresta pertencente ao objeto 3D.
inc	1 *bit*	Esse parâmetro foi utilizado para fins de teste. Como exemplo, ele funcionava para controlar o sentido de rotação do cubo na tela, assim como parar/habilitar a animação, ou incrementando o ângulo de rotação do cubo.
dec	1 *bit*	Esse parâmetro também foi utilizado como controle externo para testes, assim como a entrada *inc*. Nos testes realizados geralmente apresentava a funcionalidade inversa à entrada anterior, decrementando o ângulo de rotação ao qual o cubo estava submetido, por exemplo.
Saída		
c_x	32 *bits*	Esse valor representa a coordenada *X* da posição da câmera virtual.
c_y	32 *bits*	Esse valor representa a coordenada *Y* da posição da câmera virtual.
c_z	32 *bits*	Esse valor representa a coordenada *Z* da posição da câmera virtual.
v_x	32 *bits*	Esse valor representa a coordenada *X* do vetor de direção da câmera virtual.
v_y	32 *bits*	Esse valor representa a coordenada *Y* do vetor de direção da câmera virtual.
v_z	32 *bits*	Esse valor representa a coordenada *Z* do vetor de direção da câmera virtual.
n_x	32 *bits*	Esse valor representa a componente *X* de um vetor normal ao vetor direção da câmera virtual.
n_y	32 *bits*	Esse valor representa a componente *Y* de um vetor normal ao vetor direção da câmera virtual.
n_z	32 *bits*	Esse valor representa a componente *Z* de um vetor normal ao vetor direção da câmera virtual.
p_x	32 *bits*	Esse valor corresponde à coordenada *X* da posição de um dos vértices do objeto 3D.
p_y	32 *bits*	Esse valor corresponde à coordenada *Y* da posição de um dos vértices do objeto 3D.
p_z	32 *bits*	Esse valor corresponde à coordenada *Z* da posição de um dos vértices do objeto 3D.
clear_pixels	1 *bit*	Esse sinal indica que a memória deve ser reiniciada, tendo todos os seus antigos valores atualizados. O objetivo desse sinal é limpar a memória para que se possa desenhar o objeto em outra posição, ou em outro ângulo de visão. Dessa forma, o desenho sucessivo de um objeto com pequenas variações dos parâmetros de entrada pode originar animações, como é o caso do exemplo do cubo em *wireframe* rotacionando no eixo *X*.
out_draw_wire	1 *bit*	Esse sinal indica para o módulo *joiner* que o mesmo pode começar a armazenar o valor resultante das transformações aplicadas aos dois vértices da aresta (origem e destino).

Todas as saídas que representam coordenadas, tanto da câmera quanto do ponto pertencente ao objeto 3D, já se encontram no formato de ponto-fixo adotado (32 *bits*). As informações dos quatro vetores gerados, *X*, *Y* e *Z* estão representadas em coordenadas de mundo, ou seja, sem nenhuma transformação aplicada, todas elas relativas à origem do espaço 3D.

inv32_inst

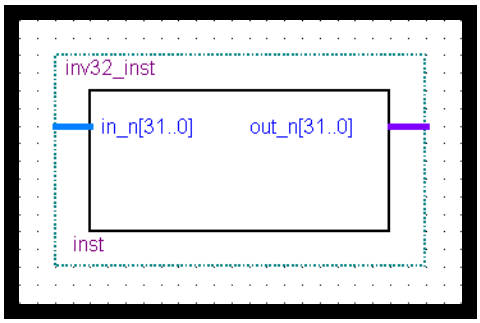

Figura 71. Símbolo que representa o módulo *inv32_inst*.

Este módulo, cujo símbolo é mostrado na Figura 71, faz uso de uma instância do *div32*. Apesar da entrada e saída apresentarem 32 *bits*, internamente é realizada uma manipulação numérica e um numerador com 33 *bits* é utilizado no módulo de divisão instanciado. Inicialmente procurou-se utilizar um módulo de divisão com 32 *bits* tanto para o numerador quanto para o denominador, assim como para a saída. A partir deste módulo, foi possível criar um módulo capaz de inverter um número em ponto-fixo apenas utilizando a constante "1" em formato de ponto-fixo no lugar do denominador e tornando o denominador variável. Percebeu-se que o resultado obtido não era satisfatório, uma vez que os *bits* estavam deslocados e não se encaixavam no "*range*" dos *bits* utilizados. Em resumo, para transformar um módulo de divisão inteira em um que realize divisão de números em formato de ponto-fixo, é necessário apenas passar os números (numerador e denominador, já no formato de ponto-fixo) como entrada e realizar uma operação de deslocamento de *bits* ao término da operação. Verificou-se que a operação de deslocamento poderia ser suprimida utilizando-se o valor "1" como constante no formato de ponto-fixo, só que já previamente deslocado dezesseis vezes para a esquerda, como mostra a Figura 72.

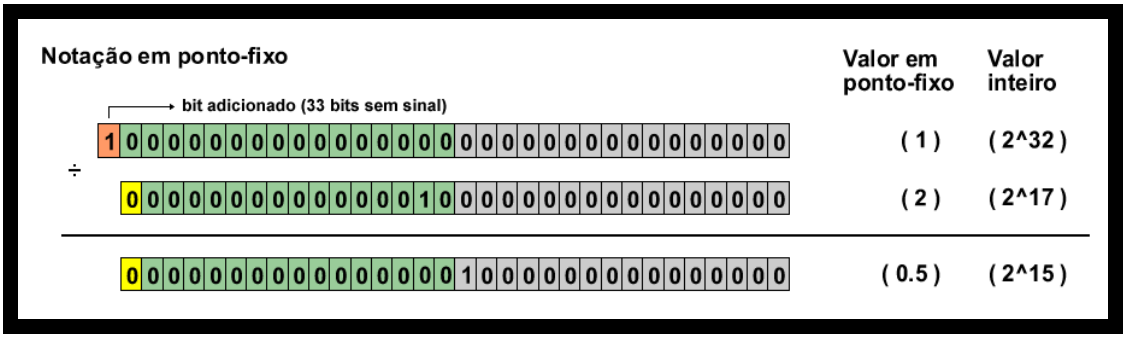

Figura 72. Manipulação dos *bits* realizada na operação de inversão.

Dessa forma, não é preciso realizar nenhuma operação de deslocamento sobre o resultado obtido com a divisão inteira do número, e a saída representa o resultado no formato adequado.

Descrição dos pinos de entrada e saída:

Nome	Tamanho	Descrição
Entrada		
in_n	32 *bits*	Parâmetro que corresponde ao número de entrada que será invertido, no formato de ponto-fixo com sinal.
Saída		
out_n	32 *bits*	Esse sinal de saída corresponde ao valor invertido do número passado como entrada.

Esse módulo funciona de forma combinacional, ou seja, não é necessário um sinal de *clock* para o processamento. O resultado encontra-se disponível logo após a alimentação dos parâmetros de entrada.

joiner

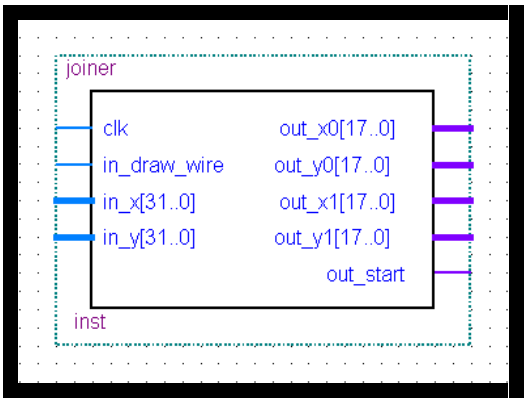

Figura 73. Símbolo que representa o módulo *joiner*.

Apesar deste módulo, cujo símbolo é mostrado na Figura 73, apresentar um baixo grau de complexidade, possuindo apenas uma máquina de estados simples, é extremamente importante para o processo de desenho de linhas. Ele é responsável por reter os valores das coordenadas dos dois pontos, que juntos formam a aresta a ser desenhada pelo módulo *bresenham*. Assim que o sinal de entrada *in_draw_wire* é ativado, os valores das coordenadas do ponto de origem da aresta são armazenados em um sinal interno. No próximo ciclo de *clock*, o módulo reconhece os novos valores de entrada como sendo relacionados ao segundo ponto. Tendo em vista essa seqüência de passagem de entradas entre os módulos, é de extrema importância a sincronização entre todos os módulos, para que os valores que trafegam durante todo o caminho de processamento estejam consistentes desde a origem (saída do *input_gen*) até seu destino (exibição da aresta na tela). Após carregar as coordenadas dos dois pontos, o sinal *out_start* é ligado para indicar ao módulo *bresenham* que o processamento da aresta pode ser iniciado.

Descrição dos pinos de entrada e saída:

Nome	Tamanho	Descrição
Entrada		
clk	1 *bit*	Utilizado como sinal de *clock* do módulo.
in_draw_wire	1 *bit*	Esse parâmetro indica para o módulo, quando habilitado, que ele pode começar a armazenar as coordenadas dos pontos recebidos como parâmetro.
in_x	32 *bits*	Esse parâmetro é utilizado na captura da coordenada *X* dos dois pontos dados como entrada em intervalos diferentes de *clock*.
in_y	32 *bits*	Esse parâmetro é utilizado na captura da coordenada *Y* dos dois pontos dados como entrada em intervalos diferentes de *clock*.
Saída		
out_x0	18 *bits*	Esse sinal de saída representa o valor da coordenada *X* do primeiro ponto, que dará origem ao desenho da aresta do objeto 3D.
out_y0	18 *bits*	Esse sinal de saída representa o valor da coordenada *Y* do primeiro ponto, que dará origem ao desenho da aresta do objeto 3D.
out_x1	18 *bits*	Esse sinal de saída representa o valor da coordenada *X* do segundo ponto, que juntamente com o primeiro irão formar a aresta do objeto 3D.
out_y1	18 *bits*	Esse sinal de saída representa o valor da coordenada *Y* do segundo ponto, que juntamente com o primeiro irão formar a aresta do objeto 3D.
out_start	1 *bit*	Esse sinal de saída indica para o módulo *bresenham* que as coordenadas geradas são válidas e que o processo de desenho das arestas pode ter início.

Esse módulo também se encarrega de transformar o número do formato de ponto-fixo para o formato de inteiro de 18 *bits* que é requerido pelo módulo *bresenham*. O módulo mantém a saída até que o sinal de entrada *in_draw_wire* seja novamente ativado e as entradas sejam modificadas.

mult32_inst

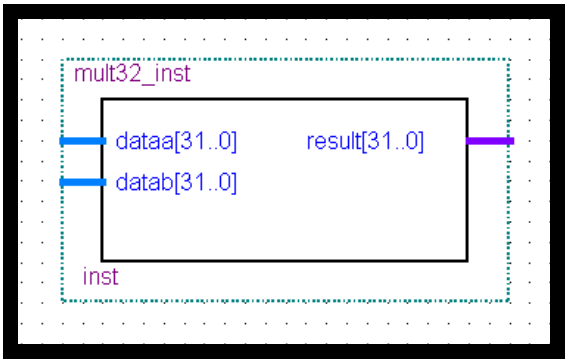

Figura 74. Símbolo que representa o módulo *mult32_inst*.

Este módulo, cujo símbolo é mostrado na Figura 74, faz uso de uma instância do módulo *mult32* para realizar a operação de multiplicação de números no formato de ponto-fixo. Como o módulo instanciado foi criado com o objetivo de operar com números inteiros, deve-se considerar apenas a parte intermediária (32) dos 64 *bits* do resultado. Dessa forma, tem-se um multiplicador que recebe duas entradas de 32 *bits* no formato ponto-fixo e retorna como saída um número também em ponto-fixo com a mesma quantidade de *bits* da entrada. Esse módulo não trata multiplicações que possam causar *overflow* numérico, e em situação onde os números extrapolam o limite dos 16 *bits* para a parte inteira, o resultado final é considerado inconsistente, já que apenas a parte intermediária do resultado é considerada, como mostra a Figura 75.

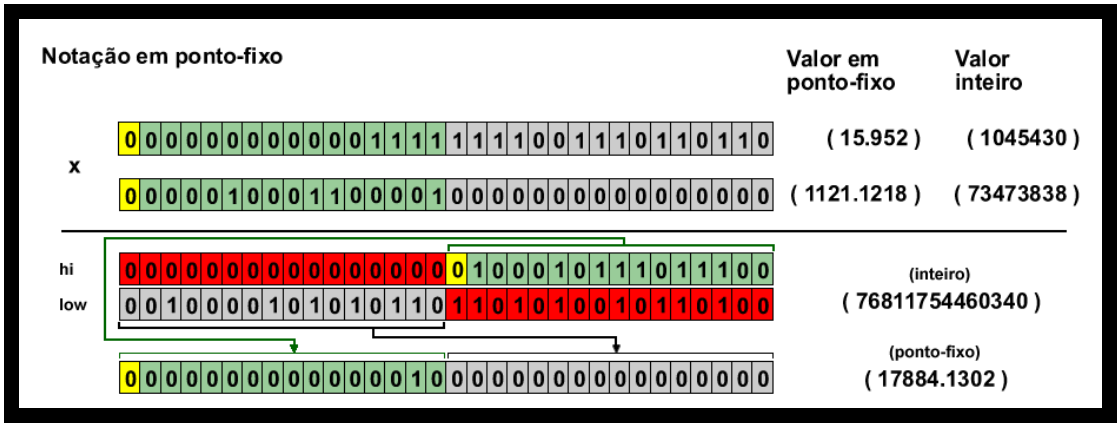

Figura 75. Manipulação dos *bits* na multiplicação.

Optou-se pela saída com tamanho de 32 *bits*, por todos os outros módulos do sistema que trabalham com o formato de ponto-fixo apresentarem o formato de 32 *bits*. Ou seja, de uma forma ou de outra um truncamento seria realizado no número de 64 *bits* resultante do módulo de multiplicação inteira instanciado. Da maneira implementada, perde-se 16 *bits* na precisão do número e 16 *bits* na parte inteira, o que pode ocasionar *overflows*, como já explicado anteriormente.

Descrição dos pinos de entrada e saída:

Nome	Tamanho	Descrição
Entrada		
dataa	32 *bits*	Parâmetro que corresponde a um dos números que fará parte da operação de multiplicação a ser realizada.
datab	32 *bits*	Parâmetro que corresponde a um dos números que fará parte da operação de multiplicação a ser realizada.
Saída		
result	32 *bits*	Esse sinal de saída corresponde ao resultado da multiplicação dos dois parâmetros de entrada, já no formato de ponto-fixo escolhido.

Esse módulo não depende de sinal de *clock*, uma vez que foi implementado de forma combinacional. O resultado da multiplicação é disponibilizado logo após o fornecimento das entradas. O módulo de multiplicação inteira que foi instanciado é mapeado em um dos multiplicadores do FPGA, utilizando dessa forma circuito dedicado para realizar a multiplicação.

normalize

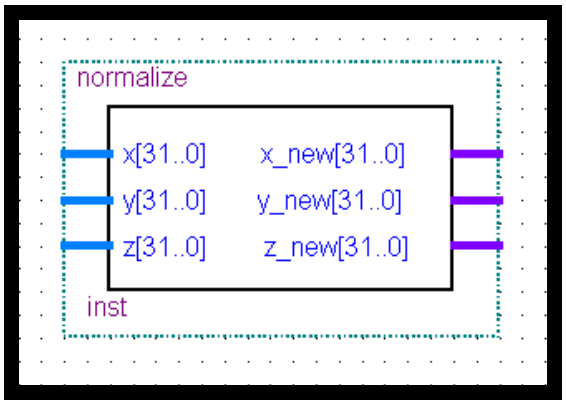

Figura 76. Símbolo que representa o módulo *normalize*.

Esse módulo, cujo símbolo é mostrado na Figura 76, é responsável pela operação de normalização de vetores. Alguns vetores que representam direção de câmera, ou até mesmo para cálculos de transformação de matriz de visualização devem estar normalizados para que as operações aconteçam de forma correta. A operação de normalização obedece às equações mostradas na Figura 77.

$$V = (x, y, z)$$
$$\|V\| = \sqrt{x^2 + y^2 + z^2}$$
$$V' = \frac{V}{\|V\|} = \frac{(x, y, z)}{\sqrt{x^2 + y^2 + z^2}}$$

Figura 77. Equações para normalização de um vetor.

Como resultado do processamento desse módulo, têm-se as três coordenadas de um vetor com a mesma direção do vetor de origem e com tamanho unitário, conforme mostrado na Figura 78.

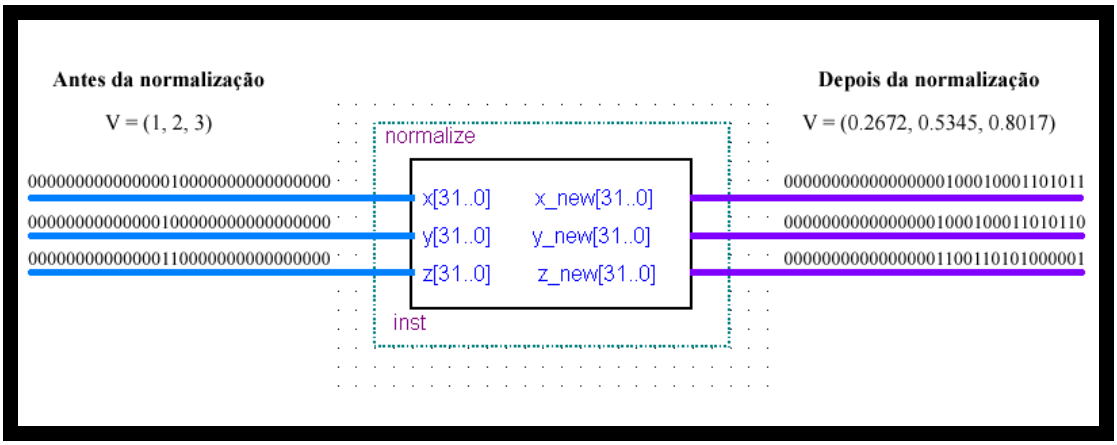

Figura 78. Vetores representados na notação de ponto-fixo antes e depois do processamento, e seus respectivos valores em decimal.

Descrição dos pinos de entrada e saída:

Nome	Tamanho	Descrição
Entrada		
x	32 *bits*	Parâmetro que representa a coordenada *X* do vetor a ser normalizado.
y	32 *bits*	Parâmetro que representa a coordenada *Y* do vetor a ser normalizado.
z	32 *bits*	Parâmetro que representa a coordenada *Z* do vetor a ser normalizado.
Saída		
x_new	32 *bits*	Sinal de saída com o valor da coordenada *X* do vetor normalizado.
y_new	32 *bits*	Sinal de saída com o valor da coordenada *Y* do vetor normalizado.
z_new	32 *bits*	Sinal de saída com o valor da coordenada *Z* do vetor normalizado.

Todos os números manipulados por este módulo encontram-se no formato de ponto-fixo, assim como as saídas geradas pelo mesmo.

orthogonalize

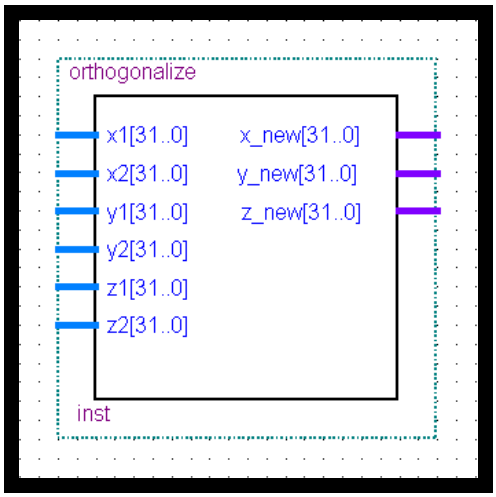

Figura 79. Símbolo que representa o módulo *orthogonalize*.

Este módulo, cujo símbolo é mostrado na Figura 79, é responsável por encontrar um vetor ortogonal a outro. Ele realiza operações aritméticas sobre os vetores de entrada e gera como saída um vetor ortogonal ao primeiro, baseado no segundo vetor fornecido como parâmetro, conforme mostrado na Figura 80.

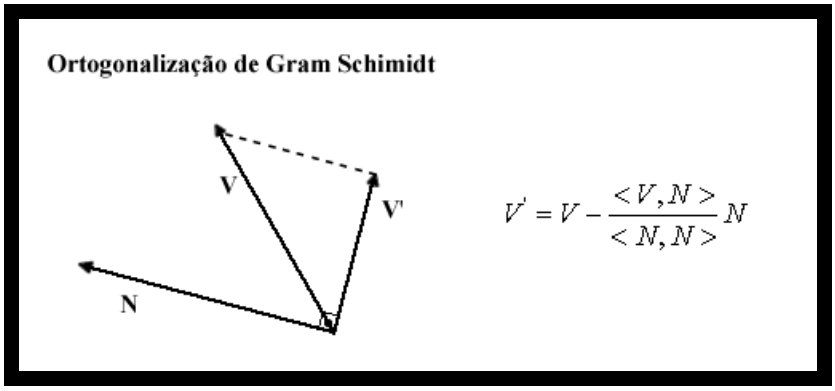

Figura 80. Fórmula da ortogonalização de vetores.

Esse módulo faz uso de instâncias dos seguintes módulos criados: *prodk*, *prodesc*, *sub32_inst*, *inv32_inst*, *mult32_inst* e *adder32_inst*.

Descrição dos pinos de entrada e saída:

Nome	**Tamanho**	**Descrição**
Entrada		
x1	32 *bits*	Parâmetro que representa o valor da coordenada *X* do vetor considerado como base do processo de ortogonalização.
y1	32 *bits*	Parâmetro que representa o valor da coordenada *Y* do vetor considerado como base do processo de ortogonalização.
z1	32 *bits*	Parâmetro que representa o valor da coordenada *Z* do vetor considerado como base do processo de ortogonalização.
x2	32 *bits*	Parâmetro que representa o valor da coordenada *X* do vetor que será utilizado como referência na ortogonalização.
y2	32 *bits*	Parâmetro que representa o valor da coordenada *Y* do vetor que será utilizado como referência na ortogonalização.
z2	32 *bits*	Parâmetro que representa o valor da coordenada *Z* do vetor que será utilizado como referência na ortogonalização.
Saída		
x_new	32 *bits*	Sinal de saída que representa a coordenada *X* de um vetor ortogonal ao primeiro vetor fornecido como parâmetro.
y_new	32 *bits*	Sinal de saída que representa a coordenada *Y* de um vetor ortogonal ao primeiro vetor fornecido como parâmetro.
z_new	32 *bits*	Sinal de saída que representa a coordenada *Z* de um vetor ortogonal ao primeiro vetor fornecido como parâmetro.

Toda a operação é realizada de forma combinacional, sem necessitar de sinal de *clock* para o processamento. A saída é disponibilizada assim que as entradas são modificadas. Todos os números manipulados por este módulo encontram-se no formato de ponto-fixo, assim como as saídas geradas pelo mesmo.

prodesc

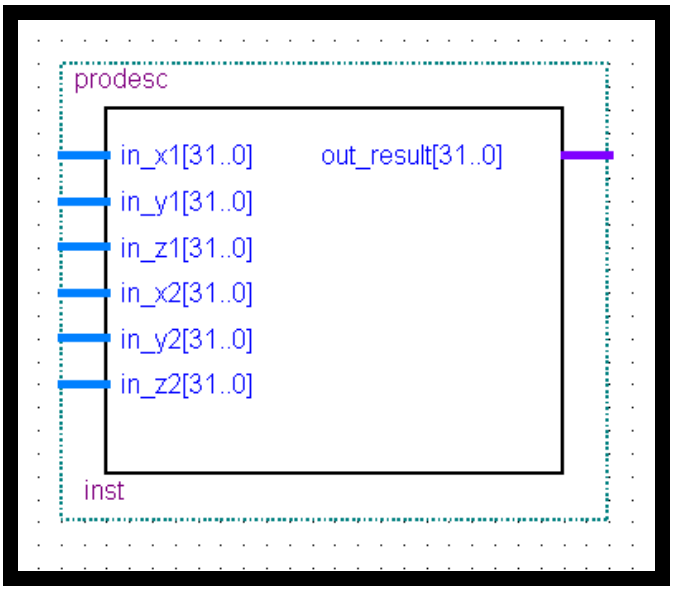

Figura 81. Símbolo que representa o módulo *prodesc*.

Esse módulo, cujo símbolo é mostrado na Figura 81, é responsável por realizar a operação de produto escalar entre dois vetores. O processamento que ocorre no interior do módulo corresponde ao esquema mostrado na Figura 82.

$$V_1 = (x_1, y_1, z_1)$$
$$V_2 = (x_2, y_2, z_2)$$
$$< V_1, V_2 > = x_1 \times x_2 + y_1 \times y_2 + z_1 \times z_2$$

Figura 82. Fórmula do produto escalar entre dois vetores.

O módulo *prodesc* apenas utiliza instâncias dos módulos *adder32* e *mult32_inst* em seu processamento.

Descrição dos pinos de entrada e saída:

Nome	Tamanho	Descrição
Entrada		
in_x1	32 *bits*	Parâmetro que representa o valor da coordenada *X* do primeiro vetor.
in_y1	32 *bits*	Parâmetro que representa o valor da coordenada *Y* do primeiro vetor.
in_z1	32 *bits*	Parâmetro que representa o valor da coordenada *Z* do primeiro vetor.
in_x2	32 *bits*	Parâmetro que representa o valor da coordenada *X* do segundo vetor.
in_y2	32 *bits*	Parâmetro que representa o valor da coordenada *Y* do segundo vetor.
in_z2	32 *bits*	Parâmetro que representa o valor da coordenada *Z* do segundo vetor.
Saída		
out_result	32 *bits*	Sinal de saída que corresponde ao resultado do produto escalar dos dois vetores passados como parâmetro.

Toda a operação é realizada de forma combinacional, sem necessitar de sinal de *clock* para o processamento. A saída é disponibilizada assim que as entradas são modificadas. Todos os números manipulados por este módulo encontram-se no formato de ponto-fixo, assim como as saídas geradas pelo mesmo.

prodk

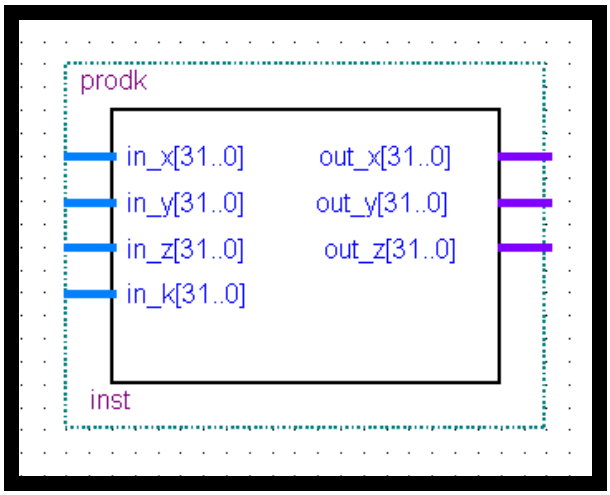

Figura 83. Símbolo que representa o módulo *prodk*.

Esse módulo, cujo símbolo é mostrado na Figura 83, é responsável por realizar o produto de um vetor por um escalar, ou seja, multiplicar o valor de todas as coordenadas de um vetor por um escalar escolhido. O processamento que ocorre no interior do módulo corresponde ao esquema mostrado na Figura 84.

$$V = (x, y, z)$$
$$V' = kV = k \times (x, y, z) = (kx, ky, kz)$$

Figura 84. Fórmula do produto de um vetor por uma constante.

Esse módulo apenas utiliza instâncias do componente *mult32_inst* no processamento realizado.

Descrição dos pinos de entrada e saída:

Nome	Tamanho	Descrição
Entrada		
in_x	32 *bits*	Parâmetro que representa o valor da coordenada *X* do vetor.
in_y	32 *bits*	Parâmetro que representa o valor da coordenada *Y* do vetor.
in_z	32 *bits*	Parâmetro que representa o valor da coordenada *Z* do vetor.
in_k	32 *bits*	Parâmetro que representa o valor do número (escalar) envolvido na operação de produto escalar.
Saída		
out_x	32 *bits*	Sinal de saída que corresponde ao resultado da multiplicação da coordenada *X* pelo escalar fornecido como parâmetro.
out_y	32 *bits*	Sinal de saída que corresponde ao resultado da multiplicação da coordenada *Y* pelo escalar fornecido como parâmetro.
out_z	32 *bits*	Sinal de saída que corresponde ao resultado da multiplicação da coordenada Z pelo escalar fornecido como parâmetro.

Toda a operação é realizada de forma combinacional, sem necessitar de sinal de *clock* para o processamento. A saída é disponibilizada assim que as entradas são modificadas. Todos os números manipulados por este módulo encontram-se no formato de ponto-fixo, assim como as saídas geradas pelo mesmo.

prodvet

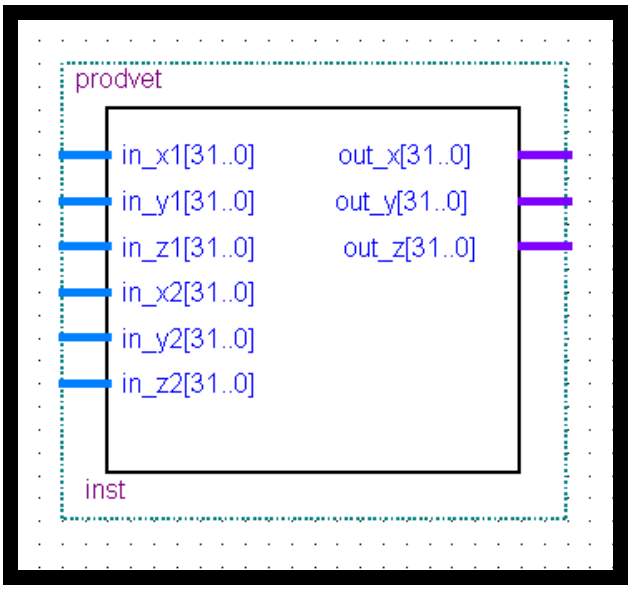

Figura 85. Símbolo que representa o módulo *prodvet*.

Este módulo, cujo símbolo é mostrado na Figura 85, é responsável por realizar o produto vetorial entre dois vetores. O processamento que ocorre no interior do módulo pode ser visualizado na Figura 86.

$$V_1 = (x_1, y_1, z_1)$$
$$V_2 = (x_2, y_2, z_2)$$
$$V_1 \times V_2 = \begin{vmatrix} i & j & k \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix} = (y_1 \times z_2 - y_2 \times z_1)i + (x_2 \times z_1 - x_1 \times z_2)j + (x_1 \times y_2 - x_2 \times y_1)k =$$
$$= (y_1 \times z_2 - y_2 \times z_1, x_2 \times z_1 - x_1 \times z_2, x_1 \times y_2 - x_2 \times y_1)$$

Figura 86. Fórmula do produto vetorial entre dois vetores.

Esse módulo apenas utiliza instâncias dos componentes *mult32_inst* e *sub32_inst* no processamento realizado.

Descrição dos pinos de entrada e saída:

Nome	Tamanho	Descrição
Entrada		
in_x1	32 *bits*	Parâmetro que representa o valor da coordenada *X* do primeiro vetor.
in_y1	32 *bits*	Parâmetro que representa o valor da coordenada *Y* do primeiro vetor.
in_z1	32 *bits*	Parâmetro que representa o valor da coordenada *Z* do primeiro vetor.
in_x2	32 *bits*	Parâmetro que representa o valor da coordenada *X* do segundo vetor.
in_y2	32 *bits*	Parâmetro que representa o valor da coordenada *Y* do segundo vetor.
in_z2	32 *bits*	Parâmetro que representa o valor da coordenada *Z* do segundo vetor.
Saída		
out_x	32 *bits*	Sinal que representa o valor da coordenada *X* do vetor resultante do produto vetorial realizado.
out_y	32 *bits*	Sinal que representa o valor da coordenada *Y* do vetor resultante do produto vetorial realizado.
out_z	32 *bits*	Sinal que representa o valor da coordenada *Z* do vetor resultante do produto vetorial realizado.

Toda a operação é realizada de forma combinacional, sem necessitar de sinal de *clock* para o processamento. A saída é disponibilizada assim que as entradas são modificadas. Todos os números manipulados por este módulo encontram-se no formato de ponto-fixo, assim como as saídas geradas pelo mesmo.

sqrt32_inst

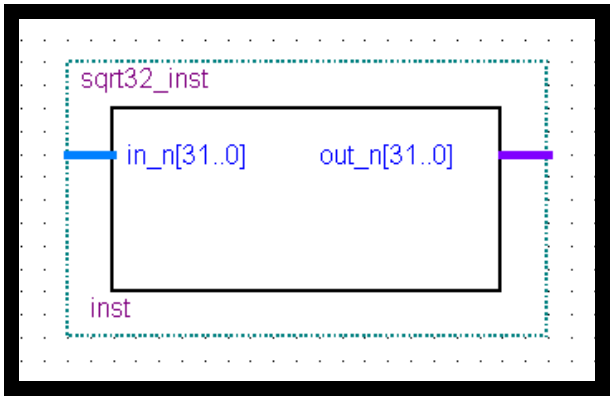

Figura 87. Símbolo que representa o módulo *sqrt32_inst*.

Esse módulo, cujo símbolo é mostrado na Figura 87, é responsável por retornar a raiz quadrada do número passado como parâmetro de entrada. Nesse processamento, apenas uma instância do componente *sqrt32* é utilizada. Como o módulo instanciado realiza a operação de radiciação de números inteiros, deve-se fazer uma manipulação nos *bits* de saída, uma vez que o parâmetro de entrada é passado no formato de ponto-fixo. A saída do módulo *sqrt32_inst* baseia-se nos 16 *bits* resultantes do componente instanciado, posicionados na parte intermediária dos 32 *bits* de saída em ponto-fixo, conforme ilustrado na Figura 88. Dessa forma, os oito primeiros *bits*, assim como os oito últimos, sempre apresentam o valor zero.

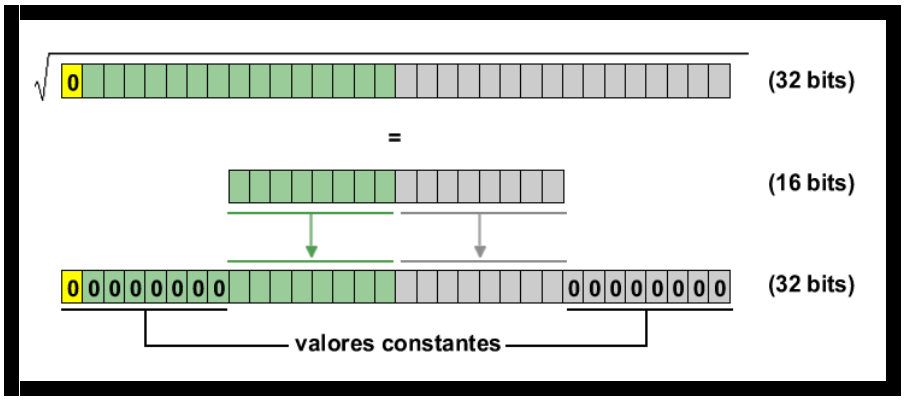

Figura 88. Manipulação dos *bits* na operação de radiciação.

A manipulação realizada nos *bits* de saída mostra que o resultado só pode possuir valores positivos (conforme esperado, por ser resultado de uma operação de radiciação). Ocorre perda de precisão com a manipulação de *bits* realizada, já que os últimos oito *bits* (os que indicam as partes fracionárias menores do número) são todos preenchidos com o valor constante e igual a zero.

Descrição dos pinos de entrada e saída:

Nome	**Tamanho**	**Descrição**
Entrada		
in_n	32 *bits*	Parâmetro que representa o valor sobre o qual a operação de radiciação será realizada.
Saída		
out_n	32 *bits*	Saída contendo o valor aproximado do resultado da operação de radiciação realizada.

Toda a operação é realizada de forma combinacional, sem necessitar de sinal de *clock* para o processamento. A saída é disponibilizada assim que as entradas são modificadas. Todos os números manipulados por este módulo encontram-se no formato de ponto-fixo, assim como as saídas geradas pelo mesmo. A Figura 89 e a Figura 90 ilustram dois exemplos do uso desse módulo.

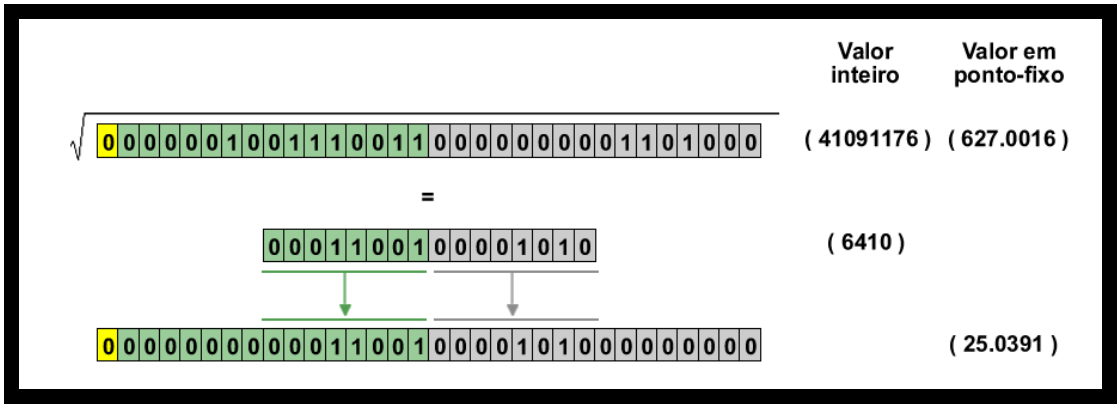

Figura 89. Exemplo: radiciação de 627.0016.

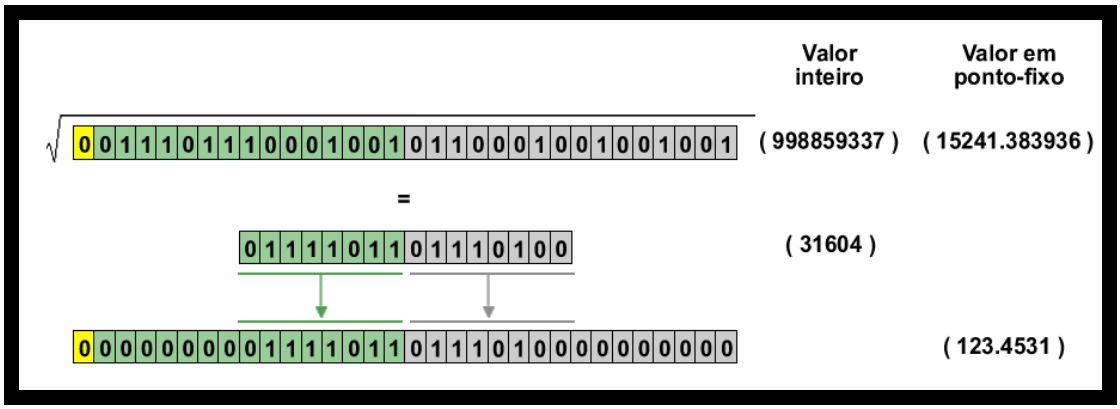

Figura 90. Exemplo: radiciação de 15241.383936.

square32_inst

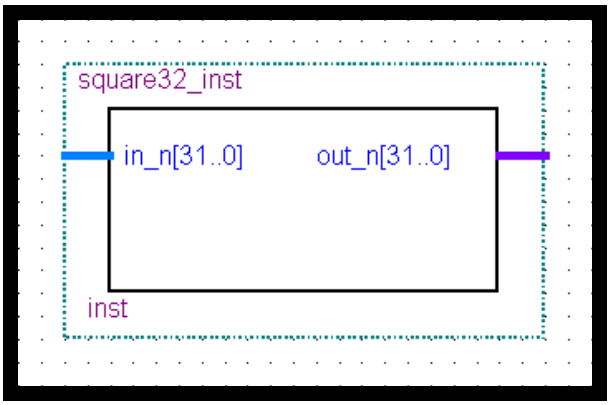

Figura 91. Símbolo que representa o módulo *square32_inst*.

Esse módulo, cujo símbolo é mostrado na Figura 91, é responsável por realizar a operação de quadrado do parâmetro de entrada (multiplicação dele por ele mesmo). Nesse processamento, apenas uma instância do componente *square32* é utilizada. Como o módulo instanciado realiza a operação de quadrado de números inteiros, deve-se fazer uma manipulação nos *bits* de saída, uma vez que o parâmetro de entrada é passado no formato de ponto-fixo. A saída do módulo *square32_inst* baseia-se na parte intermediária de 32 *bits* dos

64 *bits* resultantes do componente instanciado. Esses 32 *bits* adquiridos dão origem ao número em formato de ponto-fixo, de acordo com o processamento ilustrado na Figura 92.

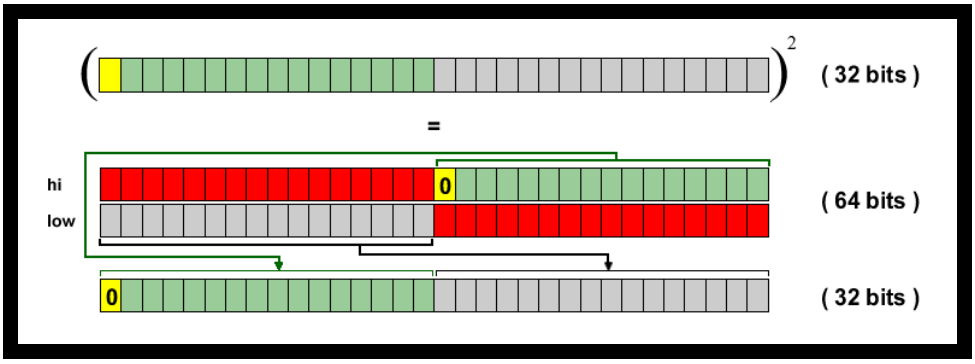

Figura 92. Manipulação dos *bits* na operação de quadrado de um número.

Descrição dos pinos de entrada e saída:

Nome	Tamanho	Descrição
Entrada		
in_n	32 *bits*	Parâmetro que representa o valor sobre o qual a operação de quadrado será realizada.
Saída		
out_n	32 *bits*	Saída contendo o valor aproximado do resultado da operação de quadrado realizada.

Toda a operação é realizada de forma combinacional, sem necessitar de sinal de *clock* para o processamento. A saída é disponibilizada assim que as entradas são modificadas. Todos os números manipulados por este módulo encontram-se no formato de ponto-fixo, assim como as saídas geradas pelo mesmo. A Figura 93 e a Figura 94 ilustram dois exemplos do uso desse módulo.

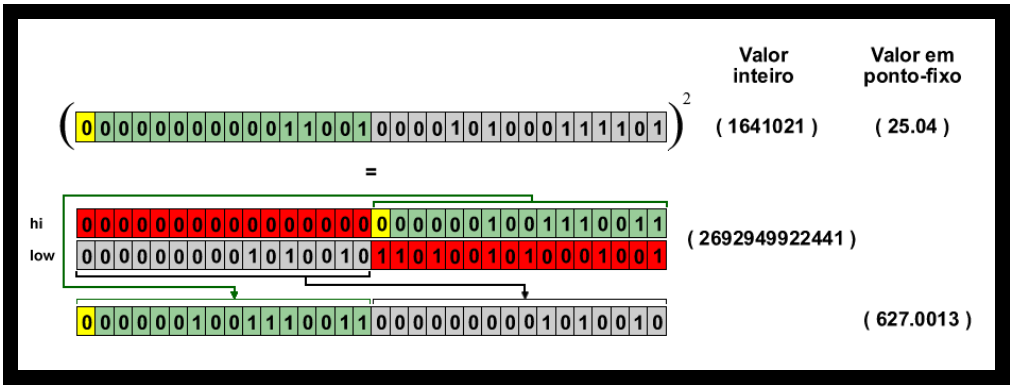

Figura 93. Exemplo: quadrado do valor 25.04.

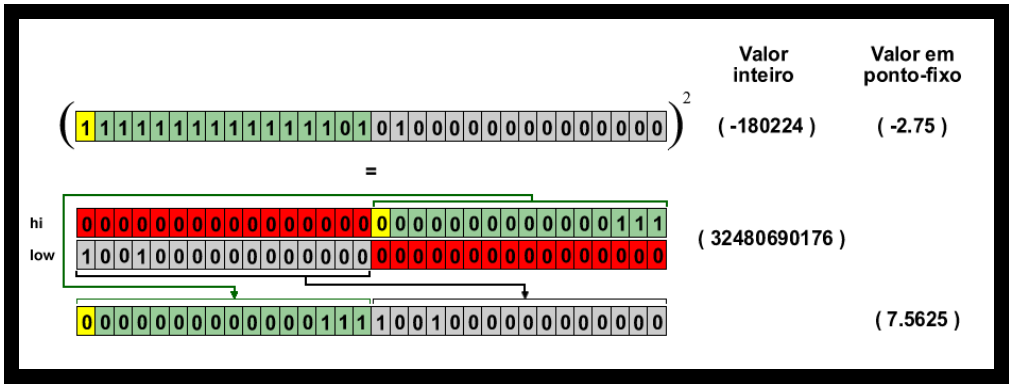

Figura 94. Exemplo: quadrado do valor (-2.75).

sub32_inst

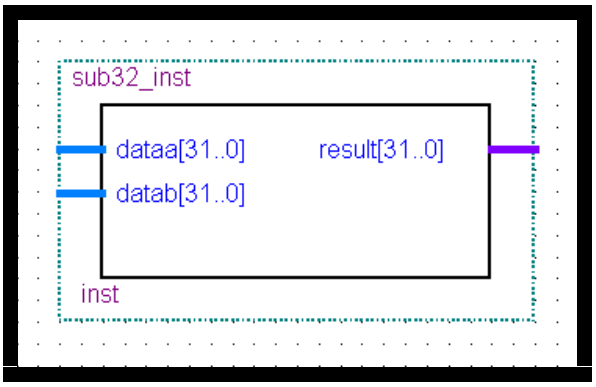

Figura 95. Símbolo que representa o módulo *sub32_inst*.

Este módulo, cujo símbolo é mostrado na Figura 95, funciona como um mapeamento do módulo *sub32*. Optou-se pelo uso dessa "interface" pelo fato de tornar independente qual módulo de subtração está sendo utilizado. Conforme visto na seção que detalha as características comuns entre números inteiros e números no formato de ponto-fixo, a operação de subtração sobre números inteiros funciona de forma semelhante à operação de subtração no formato decimal utilizado. Portanto, esse módulo somente repassa os valores de entrada e saída para o módulo mais interno, responsável por realizar a operação.

A Figura 96, a Figura 97, a Figura 98 e a Figura 99 ilustram exemplos de uso do módulo com o formato de ponto-fixo.

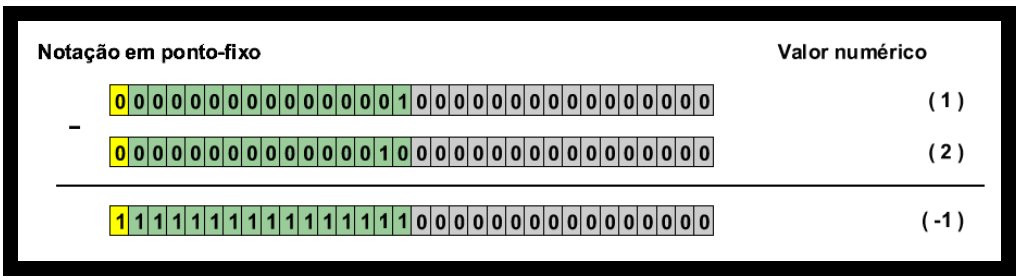

Figura 96. Exemplo: 1 – 2.

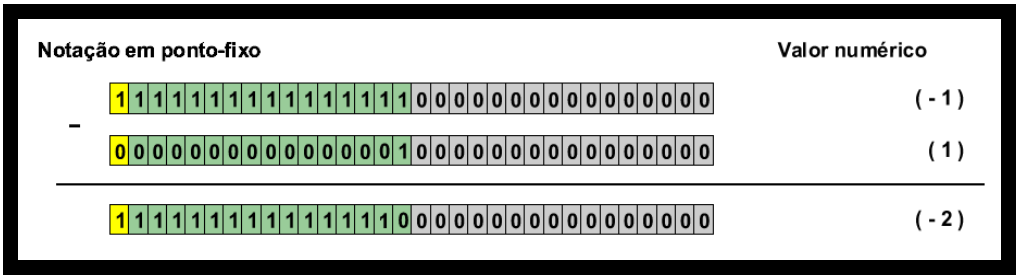

Figura 97. Exemplo: -1 – 1.

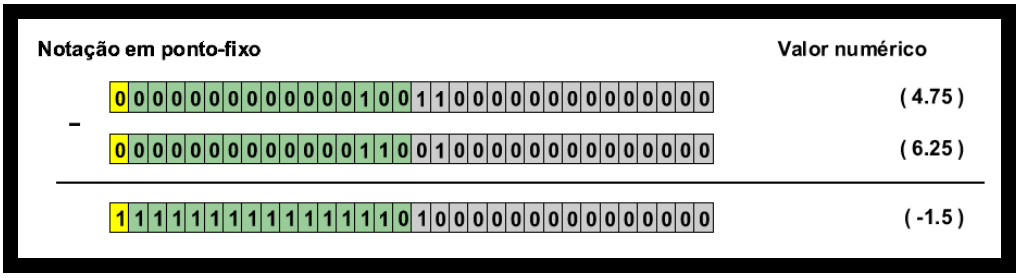

Figura 98. Exemplo: 4.75 – 6.25.

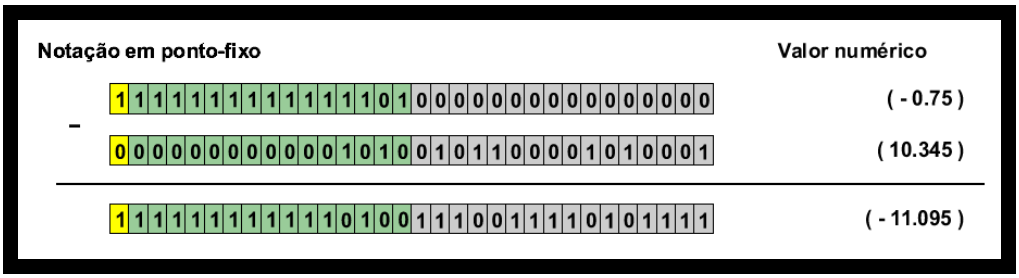

Figura 99. Exemplo: -0.75 – 10.345.

Descrição dos pinos de entrada e saída:

Nome	Tamanho	Descrição
Entrada		
dataa	32 *bits*	Parâmetro que representa um dos dois valores a serem subtraídos.
datab	32 *bits*	Parâmetro que representa um dos dois valores a serem subtraídos.
Saída		
result	32 *bits*	Saída contendo o valor do resultado da operação de subtração realizada.

Toda a operação é realizada de forma combinacional, sem necessitar de sinal de *clock* para o processamento. A saída é disponibilizada assim que as entradas são modificadas. Todos os números manipulados por este módulo encontram-se no formato de ponto-fixo, assim como as saídas geradas pelo mesmo.

world2eye

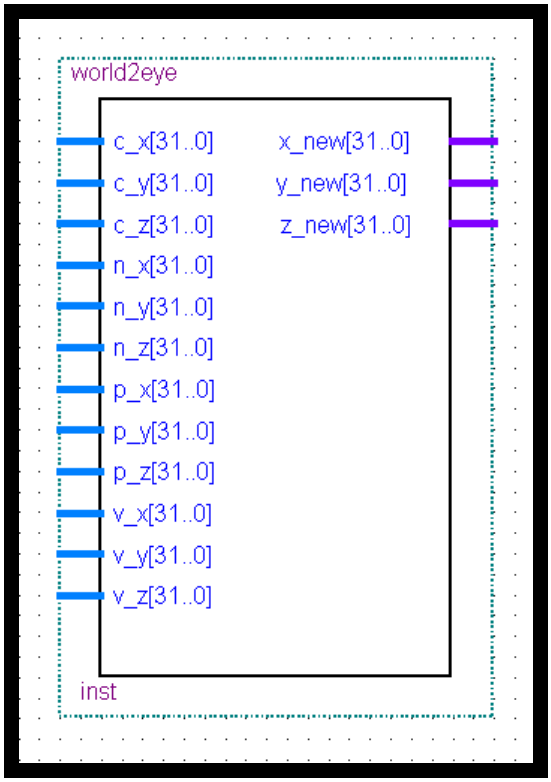

Figura 100. Símbolo que representa o módulo *world2eye*.

Apesar deste ser um dos módulos mais extensos do *Hardwire*, seu desenvolvimento ocorreu normalmente. O módulo, cujo símbolo é mostrado na Figura 100, recebe os parâmetros de visualização de um ponto em coordenadas globais, e juntamente com os parâmetros da câmera fornecidos, calcula as novas

coordenadas do ponto no sistema de coordenadas de vista (as novas coordenadas são dadas em função da câmera). O processamento foi implementado de acordo com o esquema mostrado na Figura 101.

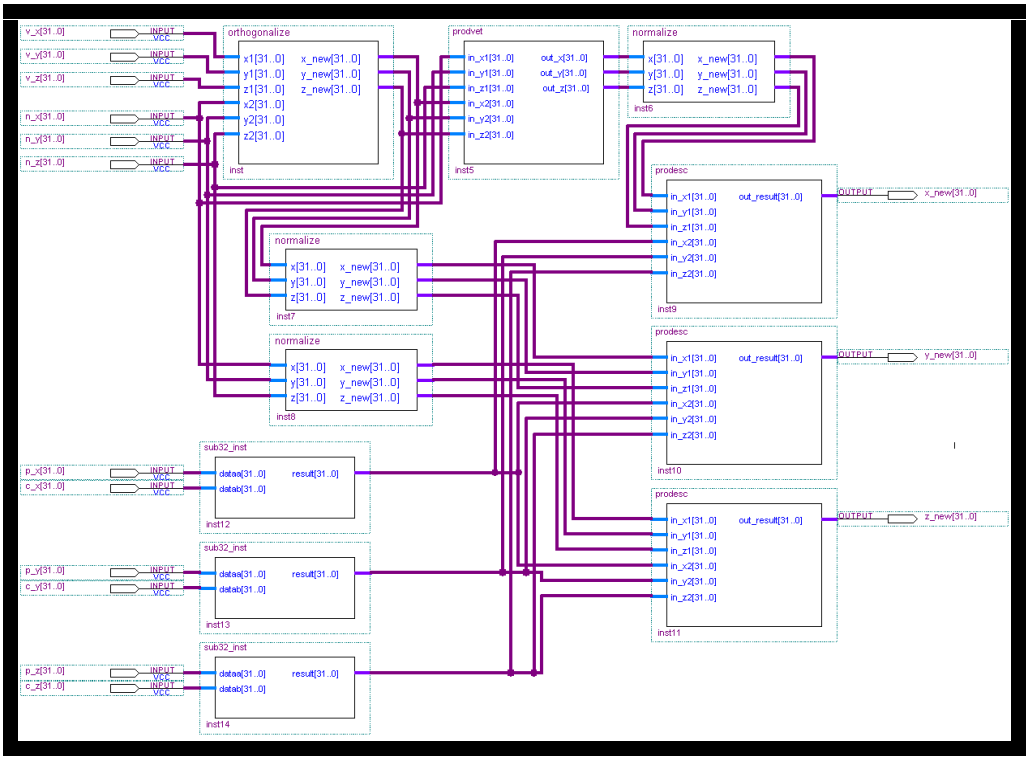

Figura 101. Esquemático do módulo *world2eye*.

Descrição dos pinos de entrada e saída:

Nome	Tamanho	Descrição
Entrada		
c_x	32 *bits*	Parâmetro que representa o valor da coordenada *X* de posição da câmera virtual.
c_y	32 *bits*	Parâmetro que representa o valor da coordenada *Y* de posição da câmera virtual.
c_z	32 *bits*	Parâmetro que representa o valor da coordenada *Z* de posição da câmera virtual.
n_x	32 *bits*	Parâmetro que representa o valor da coordenada *X* do vetor normal à direção da câmera virtual.
n_y	32 *bits*	Parâmetro que representa o valor da coordenada *Y* do vetor normal à direção da câmera virtual.
n_z	32 *bits*	Parâmetro que representa o valor da coordenada *Z* do vetor normal à direção da câmera virtual.
p_x	32 *bits*	Parâmetro que representa o valor da coordenada *X* de posição do ponto no sistema de coordenadas mundiais.
p_y	32 *bits*	Parâmetro que representa o valor da coordenada *Y* de posição do ponto no sistema de coordenadas mundiais.
p_z	32 *bits*	Parâmetro que representa o valor da coordenada *Z* de posição do ponto no sistema de coordenadas mundiais.
v_x	32 *bits*	Parâmetro que representa o valor da coordenada *X* do vetor de direção da câmera virtual.
v_y	32 *bits*	Parâmetro que representa o valor da coordenada *Y* do vetor de direção da câmera virtual.
v_z	32 *bits*	Parâmetro que representa o valor da coordenada *Z* do vetor de direção da câmera virtual.
Saída		
x_new	32 *bits*	Sinal de saída que representa o valor da coordenada *X* em coordenadas de vista.
y_new	32 *bits*	Sinal de saída que representa o valor da coordenada *Y* em coordenadas de vista.
z_new	32 *bits*	Sinal de saída que representa o valor da coordenada *Z* em coordenadas de vista.

Cada operação realizada no interior do *world2eye* representa um ou mais módulos implementados, de forma que foi necessário apenas realizar a conexão correta entre eles para se obter o resultado desejado. Considerando que cada submódulo funciona corretamente, a união de alguns deles também funcionaria como esperado. O único fator limitante, nesse caso, seria a freqüência de operação do circuito. Como tudo ocorre em apenas um ciclo de *clock*, o período deve ser grande o suficiente para que a corrente elétrica flua do início ao fim do circuito. Como não se utiliza uma freqüência muito alta (a utilizada pelo ARCam possui o valor de 50MHz), não foi necessário realizar qualquer modificação no esquema implementado, visto que as restrições de freqüência satisfaziam aquela desejada.

O módulo faz uso de instâncias dos seguintes componentes criados: *orthogonalize*, *prodesc*, *sub32_inst*, *prodvet* e *normalize*.

Toda a operação é realizada de forma combinacional, sem necessitar de sinal de *clock* para o processamento. A saída é disponibilizada assim que as entradas são modificadas. Todos os números manipulados por este módulo encontram-se no formato de ponto-fixo, assim como as saídas geradas pelo mesmo.

bit_ram

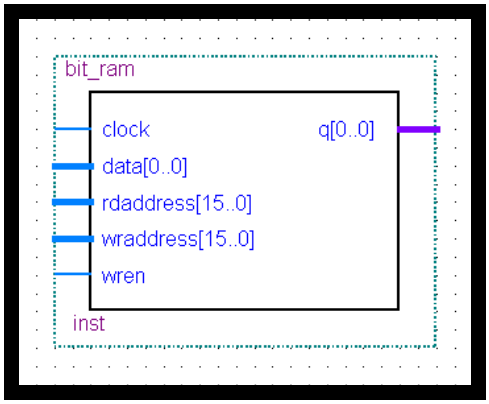

Figura 102. Símbolo que representa o módulo *bit_ram*.

Este módulo, cujo símbolo é mostrado na Figura 102, funciona como uma memória que armazena valores de tamanho unitário (1 *bit*). Ele indica quais as posições da tela que possuem um *pixel* preenchido (que faça parte de pelo menos uma das arestas do objeto 3D desenhado na tela). O *Hardwire* apenas manipula diretamente essa "memória". O processo de escrita da imagem resultante na saída VGA acessa simultaneamente as memórias de captura da câmera e o *bit_ram*. Caso haja um *bit* preenchido (com valor 1) na posição de memória correspondente no *bit_ram*, um *pixel* preto é pintado na tela. Em caso negativo, é pintado na tela o *pixel* com a cor capturada pela câmera.

Descrição dos pinos de entrada e saída:

Nome	**Tamanho**	**Descrição**
Entrada		
clock	1 *bit*	Sinal de *clock* utilizado pela memória.
data	1 *bit*	Representa o dado a ser gravado em alguma posição de memória específica.
rdaddress	16 *bits*	Sinal que indica a posição de memória da qual o dado deve ser lido.
wraddress	16 *bits*	Sinal que indica a posição de memória na qual o dado deve ser escrito.
wren	1 *bit*	Sinal que habilita a escrita na posição de memória especificada por *wraddress*.
Saída		
q	1 *bit*	Saída da memória que representa o conteúdo do espaço endereçado pelo parâmetro *rdaddress*.

Esse módulo funciona com base em um sinal de *clock*, responsável por indicar quando um dado qualquer

deve ser lido ou escrito no espaço de memória reservado.

4.7. Resultados Obtidos

Após a finalização do primeiro protótipo do *Hardwire*, é possível visualizar um cubo (objeto 3D simples utilizado para os testes iniciais) na tela do monitor ligado diretamente à placa de prototipação, como ilustrado na Figura 103.

Foram realizadas algumas alterações a partir dessa primeira versão e inseriu-se a funcionalidade de rotação do objeto 3D, como pode ser visto na Figura 104. Isso permitiu que se mostrasse o objeto 3D de forma animada na tela, aperfeiçoando a visualização do mesmo.

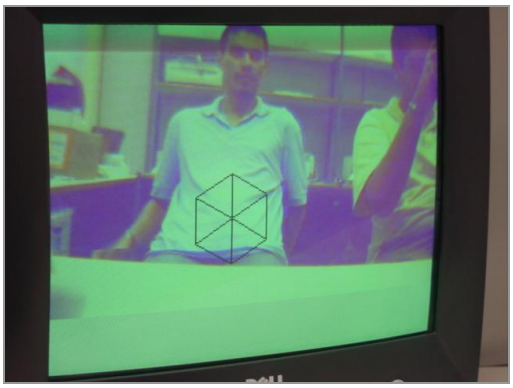

Figura 103. Primeiro objeto visualizado usando o *Hardwire*.

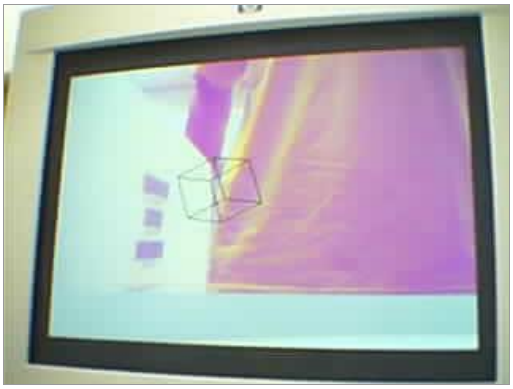

Figura 104. Cubo animado através de rotação.

Ao final da implementação em *hardware*, verificou-se através do analizador presente no Quartus que a freqüência de operação obtida com o *Hardwire* atendeu ao *clock* de 50MHz utilizado no projeto ARCam. A arquitetura pode ser modificada para suportar *clocks* superiores, bastando para isso particionar os componentes e transformar boa parte do circuito combinacional em seqüencial, visto que esse é um dos maiores limitantes da freqüência de operação final.

Constatou-se que a quantidade de pulsos de *clock* necessária para o desenho por completo do objeto virtual testado (cubo 3D) é inversamente proporcional à sua distância para o observador. Ou seja, quanto mais próximo o objeto se encontra da câmera virtual, maior ele será visualizado e, conseqüentemente, mais *pixels* deverão ser desenhados na tela, através do algoritmo de Bresenham.

O circuito é sempre retro-alimentado pelo módulo *input_gen*, de forma que uma vez que o objeto é completamente desenhado na tela, o módulo reinicia o envio dos dados dos vértices e arestas para a renderização. Caso não se aplique nenhuma transformação sobre os vértices do cubo, um novo objeto 3D será desenhado por cima do anterior, sem diferenças perceptíveis. Para os testes realizados, optou-se por aplicar sucessivas transformações de rotação sobre o objeto em um dos eixos cartesianos. Dessa forma, o

objeto renderizado no ciclo de renderização anterior seria diferente do atual e com o passar do tempo uma região da tela seria preenchida pela ocupação do objeto em seus diversos estados de rotação. De forma a possibilitar a visualização da animação de rotação do objeto 3D, e outras modificações subseqüentes na entrada, foi implementada uma operação de *refresh* (atualização da memória) na memória que armazena os *pixels* renderizados. Sempre que a entrada era modificada, um sinal de *refresh* era ativado e a memória passava um período de 64000 ciclos de *clock* (a memória armazenava um *bit* para cada *pixel*, sendo utilizada uma resolução de 320x200) para limpar seu conteúdo por completo. Com a finalização da limpeza da memória, um sinal era enviado para o módulo *input_gen*, de forma que a renderização pudesse ser reiniciada.

4.8. Dificuldades Encontradas

O desenvolvimento de artefatos em *hardware* acrescenta uma série de dificuldades e desafios àqueles já presentes no desenvolvimento de *software*. Um deles é o espaço (número de elementos lógicos disponíveis), além da freqüência de operação do circuito e da velocidade do mesmo. Essas preocupações devem ser levadas em consideração durante todo o projeto de um circuito embarcado.

As principais dificuldades encontradas na implementação do módulo *Hardwire* aconteceram na fase de definição das funcionalidades do projeto, na qual foram definidas com detalhes as etapas de processamento que deveriam ser implementadas. Além disso, aconteceram alguns erros devido ao uso inadequado de algumas bibliotecas fornecidas, assim como também ocorreram problemas com versões mais recentes do Quartus [21] (programa responsável pela síntese de todo o módulo), que foram solucionados ao longo da implementação.

A velocidade de compilação de projetos de *hardware* também influi diretamente no desenvolvimento dos mesmos, quando comparada à velocidade de compiladores e geradores de *software*. Muitas vezes não foi possível trabalhar com opções de otimização de área e velocidade, pois à medida que os módulos iam sendo desenvolvidos e agrupados, o tempo de compilação de todo o projeto crescia vertiginosamente.

Apesar de existir uma ferramenta de simulação do circuito funcionando em tempo real na placa, o Quartus Signal Tap Logical Analyser não permite uma boa visualização dos sinais capturados pelo circuito adicional que é posto no FPGA. Freqüentemente, o resultado das simulações foi exportado e utilizou-se um interpretador externo para agrupar os valores dos sinais simulados, sendo assim possível realizar uma comparação efetiva com valores provenientes dos protótipos e modelos funcionais criados.

5. Conclusões e Trabalhos Futuros

Este documento apresentou e descreveu o *Hardwire*, uma solução de renderização para aplicações de RA embarcadas, capaz de prover a visualização de objetos 3D em *wireframe*. Foram apresentados os módulos implementados, assim como a metodologia utilizada.

Foram estudados métodos e técnicas de implementação de algoritmos de renderização gráfica em *hardware*, definida uma arquitetura capaz de exercer tal tarefa e implementado um módulo de renderização usando uma linguagem de descrição de *hardware* (VHDL). A corretude funcional do módulo desenvolvido foi comprovada através de testes que ocorreram diretamente na placa de prototipação, exibindo um cubo 3D em *wireframe*, o que compreende uma aplicação simples de RA.

Constatou-se que o desempenho obtido pela implementação em *hardware* foi satisfatório, conforme o esperado, uma vez que existe a possibilidade de paralelismo real no FPGA. Em alguns testes realizados com o protótipo, a renderização muitas vezes acontecia de forma mais rápida do que o tempo necessário para realizar uma operação de *refresh* da memória. Por causa disso, às vezes não era possível visualizar por completo uma aresta do objeto 3D, a menos que se diminuísse o número de quadros renderizados por segundo. É fato que a implementação em *hardware* de algoritmos de computação gráfica só vem trazer benefícios para as aplicações que precisam desse tipo de suporte, uma vez que a manipulação de cálculos complexos ocupa uma grande fatia do tempo de processamento das CPUs.

Como trabalhos futuros, pretende-se implementar a projeção em perspectiva, conforme mencionado anteriormente, assim como finalizar o módulo de rotação em torno dos outros eixos. Caso se deseje renderizar o objeto 3D com preenchimento, deve-se pensar em soluções que suportem o uso de *Z-buffer* e utilização de texturas para melhor visualização.

O Trabalho de Graduação descrito neste documento contribuiu significativamente com o desenvolvimento do projeto ARCam. Espera-se, com este trabalho, ter impulsionado novos esforços de projetos na área de desenvolvimento de *hardware* do Centro de Informática, assim como na Academia de forma geral.

6. Referências

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

[2] THOMAS, B., CLOSE, B., et al., *ARQuake: An Outdoor/Indoor Augmented Reality First Person Application*. Proceedings of the ISWC, pp. 139-146, 2000.

[3] UMLAUF, E., PIRINGER, H., et al., *ARLib: the Augmented Library*. Proceedings of the IEEE International ART Workshop, 2p., 2002.

[4] GUIMARÃES, G. F., SILVA, S. M., SILVA, G. D., LIMA, J. P. S. M., TEICHRIEB, V., KELNER, J., *ARCam: solução embarcada para aplicações em realidade aumentada*. III Workshop de Realidade Aumentada (WRA), Rio de Janeiro, 2006.

[5] ARToolkit. Disponível em: Human Interface Technology Lab site.
URL: http://www.hitl.washington.edu/artoolkit. Visitado em outubro, 2006.

[6] BATH, W., PAXMAN, J., *UAV Localisation & Control Through Computer Vision*. Australian Robotics & Automation Association, 2005.

[7] Open Source Computer Vision Library. Disponível em: Intel Corporation site.
URL: http://www.intel.com/technology/computing/opencv. Visitado em outubro, 2006.

[8] BASTOS, N., *Arquitetura para dispositivos não-convencionais de interação utilizando realidade aumentada: um estudo de caso*. Recife: CIn-UFPE, 2006. Trabalho de Graduação.

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

[10] Introduction to Augmented Reality. Disponível em: Jim Vallino's RIT SE Department Home Page site.
URL: http://www.se.rit.edu/~jrv/. Visitado em outubro, 2006.

[11] WAGNER, D.; SCHMALSTIEG, D., *First Steps Towards Handheld Augmented Reality*. International Symposium on Wearable Computers (ISWC), 7., 2003, USA, Washington: IEEE Computer Society, 2003. p. 127-137.

[12] WAGNER, D.; SCHMALSTIEG, D., *A Handheld Augmented Reality Museum Guide*. IADIS International Conference on Mobile Learning, 2005.

[13] PIEKARSKI, W.; THOMAS, B., *ARQuake: The Outdoor Augmented Reality Gaming System*. Communications of the ACM, v. 45, n. 1, p. 36-38, janeiro 2002.

[14] WAGNER, D.; PINTARIC, T.; LEDERMANN, F.; SCHMALSTIEG, D., *Towards Massively Multi-User Augmented Reality on Handheld Devices*. International Conference on Pervasive Computing, 3., 2005, Germany, 2005. p. 208-219.

[15] WOODWARD, C.; HONKAMAA, P.; JÄPPINEN, J.; PYÖKKIMIES, E., *CamBall – Augmented Networked Table Tennis Played with Real Rackets*. International Conference on Advances in Computer Entertainment Technology, 2004, Singapore, New York: ACM Press, 2004. p. 275-276.

[16] ZHOU, Z.; CHEOK, A. D.; LI, Y.; KATO, H., *Magic Cubes for Social and Physical Family Entertainment*. Conference on Human Factors in Computing Systems, 2005, USA, New York: ACM Press, 2005. p. 1156-1157.

[17] BÉRARD, F., *The Magic Table: Computer-Vision Based Augmentation of a Whiteboard for Creative Meetings*. Workshop on Projector-Camera Systems, 2003, France, Washington: IEEE Computer Society Press, 2003.

[18] ULLMER, B.; ISHII, H., *Emerging Frameworks for Tangible User Interfaces*. IBM Systems Journal, v. 39, n. 3/4, p. 915-931, 2000.

[19] BUCHMANN, V.; VIOLICH, S.; BILLINGHURST, M.; COCKBURN, A., *FingARtips – Gesture Based Direct Manipulation in Augmented Reality*. International Conference on Computer Graphics and Interactive Techniques (GRAPHITE), 2., 2004, Singapore, New York: ACM Press, 2004. p. 212-221.

[20] Field-programmable gate array. Disponível em: Wikipedia site. URL: http://en.wikipedia.org/wiki/FPGA. Visitado em outubro, 2006.

[21] FPGAs, CPLDs, & Structured ASICs: Altera, the Leader in Programmable Logic. Disponível em: Altera site. URL: http://www.altera.com. Visitado em outubro, 2006.

[22] Xilinx: The Programmable Logic Company. Disponível em: Xilinx site. URL: http://www.xilinx.com. Visitado em outubro, 2006.

[23] PASMAN, W., WOODWARD, C., *Implementation of an Augmented Reality System on a PDA*. Procedings of the IEEE and ACM ISMAR, pp. 276-277, 2003.

[24] WAGNER, D., PINTARIC, T., et al., *Towards Massively Multi-user Augmented Reality on Handheld Devices*. Procedings of the Pervasive, pp. 208-219, 2005.

[25] MATSUSHITA, N., HIHARA, D., et al., *ID CAM: A Smart Camera for Scene Capturing and ID Recognition*. Procedings of the IEEE and ACM ISMAR, pp. 227-236, 2003.

[26] Manticore - open source 3D graphics accelerator. Disponível em: Manticore site. URL: http://icculus.org/manticore. Visitado em outubro, 2006.

[27] McKEON, D., *Synthesizable VHDL Model of 3D Graphics Transformations*. Trinity College Dublin, 2005. Trabalho de Graduação.

[28] FOLEY, J. D., VAN DAM, A., FEINER, S. K., HUGHES, J. F., *Computer Graphics: Principles and Practice. Second Edition in C*. Addison Wesley, USA, 2005.

[29] Bresenham's line algorithm. Disponível em: Wikipedia site. URL: http://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm. Visitado em outubro, 2006.

[30] BONATO, V., SANCHES, A., et al., *A Real Time Gesture Recognition System for Mobile Robots*. Procedings of the ICINCO, pp. 207-214, 2004.

[31] VHSIC Hardware Description Language. Disponível em: Wikipedia site. URL: http://en.wikipedia.org/wiki/Vhdl. Visitado em outubro, 2006.

[32] Floating-point. Disponível em: Wikipedia site. URL: http://en.wikipedia.org/wiki/Floating-point. Visitado em outubro, 2006.

[33] Fixed-point arithmetic. Disponível em: Wikipedia site. URL: http://en.wikipedia.org/wiki/Fixed-point. Visitado em outubro, 2006.

[34] ANDRAKA R., *A survey of CORDIC algorithms for FPGA based computers*. International Symposium on Field Programmable Gate Arrays, 1998.

Assinaturas

Judith Kelner

Veronica Teichrieb

João Marcelo Xavier Natário Teixeira