UNIVERSIDADE FEDERAL DE PERNAMBUCO

GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

CENTRO DE INFORMÁTICA

QuadDetector: um módulo em *hardware* para detecção de quadrados em tempo real

TRABALHO DE GRADUAÇÃO

Aluna: Guilherme Dias da Silva (gds@cin.ufpe.br)
Orientadora: Judith Kelner (jk@cin.ufpe.br)
Co-orientadora: Veronica Teichrieb (vt@cin.ufpe.br)

Março de 2007

RESUMO

No campo de desenvolvimento de aplicações em hardware, os FPGAs são plataformas atraentes pelo alto grau de versatilidade. Sendo um chip re-programável, o FPGA é capaz de se transformar em um *hardware* desenvolvido pelo projetista através de uma linguagem de descrição de hardware. A aplicação de FPGAs no desenvolvimento de sistemas embarcados com soluções para realidade aumentada configura-se uma área de pesquisa recente, cujo objetivo é possibilitar a criação de um sistema dedicado para processamento de imagens com alta taxa de quadros por segundo.

O objetivo deste Trabalho de Graduação é construir um módulo em *hardware*, denominado QuadDetector, usando a linguagem para descrição de hardware VHDL, que seja capaz de fazer o processamento em tempo real de imagens e detectar quadrados na cena capturada. Essa detecção faz parte de uma primeira etapa de identificação de marcadores para aplicações de realidade aumentada utilizando uma biblioteca de desevolvimento para tais aplicações em software, conhecida como ARToolKit.

O QuadDetector é um subprojeto do ARCam (*Augmented Reality Câmera)*, financiado pelo CNPq, que visa embarcar os passos implementados em *software* pelo ARToolKit, permitindo então o desenvolvimento de aplicações de realidade aumentada em *hardware*.

AGRADECIMENTOS

Antes de tudo agradeço a Deus pelo dom da vida.

Agradeço a meus pais pelo apoio de sempre em qualquer momento, mesmo fisicamente longes. Minha linda mãe que sempre esteve ao meu lado, com toda sua fé em Deus, e meu fiel conselheiro pai. Agradeço pelo apoio nos momentos mais difíceis do curso, quando a dúvida sobre a desistência era recorrente e suas palavras aconchegantes. Sempre ao meu lado, em qualquer momento, para qualquer coisa. Incluo aqui também a participação fundamental dos meus irmãos, não só no curso como na minha vida.

Agradeço à minha companheira de mais de 7 anos Cellinha pelo apoio, sempre ao meu lado durante todo esse curso. Nos momentos mais difíceis sempre é meu refúgio, meu porto seguro.

Agradeço à meus avós pela acolhida aqui em Recife e todo o apoio me dado.

Agradeço aos amigos de trabalho pela força, especialmente João Marcelo (Joma) e João Paulo (Jonga) por estarem sempre esclarecendo algumas dúvidas e acompanhando o projeto. À minha co-orientadora Veronica (vt) por estar sempre disposta a ajudar com sua paciência e sorriso no rosto.

Agradeço à professora Judith pela oportunidade de, junto ao Grupo de Pesquisas em Realidade Virtual e Multimídia de sua coordenação, construir esse trabalho.

Índice

Índice de figuras

Índice de tabelas

INTRODUÇÃO

Inicialmente as pesquisas tecnológicas eram tradicionalmente feitas em áreas distintas promovendo o grande conhecimento de grupos de pesquisadores sobre um determinado assunto estudado e gerando, muitas vezes, ilhas de conhecimento que pouco interagem para gerar novos conhecimentos. Hoje, há uma tendência mundial de interdisciplinaridade nas pesquisas gerando um conhecimento misto com impacto nas mais diversas áreas.

Nesse Trabalho de Graduação será utilizada uma abordagem mista, envolvendo prototipação de Sistemas Embarcados[22] com aplicação direta em Realidade Aumentada[1]. Essa área de pesquisa é conhecida como RA, e relacionada ao campo da Realidade Virtual (RV). O intuito desse trabalho é associar aspectos relevantes das aplicações embarcadas, como velocidade de processamento, tamanho do dispositivo e versatilidade de desenvolvimento com demandas latentes das aplicações de RA que, na sua grande maioria, são desenvolvidas em *software* utilizando bibliotecas de desenvolvimento. O uso dessas bibliotecas está, necessariamente, atrelado ao uso de um PC convencional que possui poder computacional muitas vezes não utilizado na aplicação desenvolvida, gerando ociosidade computacional. Além, é claro, do uso de um sistema operacional, que facilita o desenvolvimento, mas gera concorrência pela CPU com outras aplicações.

Em RA, uma importante biblioteca para desenvolvimento de aplicações é o ARToolKit[2] que serviu como base para a consolidação desse trabalho.

1. Objetivos

O objetivo desse Trabalho de Graduação é estudar, definir e desenvolver uma arquitetura em hardware, o QuadDetector, que incorpore módulos necessários para a execução de todos os passos fundamentais da detecção de quadrados, tornando capaz a inserção desse módulo maior no projeto ARCam, apresentado na seção [7], incorporando maiores funcionalidades ao mesmo e contribuindo para o objetivo final desse projeto, embarcar o ARToolKit.

Como resultados desse trabalho serão apresentados módulos responsáveis por:

1) Reconhecer bordas no *frame* capturado do mundo real retirando um contorno fechado da cena;

2) Aplicar algoritmos de redução de vértices para determinar o número de *pixels* a serem analisados no contorno;

3) Aplicar algoritmos de aproximação poligonal para verificar se o contorno é um polígono;

4) Identificar se o polígono obtido é um quadrado;

5) Destacar na cena o quadrado identificado;

2. Estrutura do Documento

Este documento está estruturado em seis capítulos que descrevem desde a problemática identificada até a solução adotada e o material de referência usado.

O capítulo um identifica a problemática atacada, bem como apresenta a solução proposta de uma maneira geral, destacando aspectos importantes das áreas de pesquisa envolvidas.

O capítulo dois contextualiza o trabalho, conceituando as principais temáticas usadas e destacando o contexto ao qual ele está inserido dentro do projeto ARCam. Aspectos importantes como a RA e os Sistemas Embarcados e suas características são mostrados e ressaltados com aplicações envolvendo essas áreas.

O capítulo três apresenta o processo de detecção de quadrados e suas minúcias. Os passos que são necessários para realizar essa atividade e os algoritmos relativos a cada um desses passos. Serão apresentados todos os aspectos relevantes desse processo e alguns algoritmos em pseudocódigo que foram implementados. São apresentadas quatro subseções, começando com a detecção de um contorno, seguido pela redução de vértices, depois a aproximação poligonal até a apresentação na cena do quadrado identificado, caso exista.

O capítulo quatro é a consolidação do trabalho, apresentando o protótipo desenvolvido e como este trabalho foi realizado. O ambiente utilizado, a infra-estrutura de *hardware* necessária, os módulos desenvolvidos, a inserção do QuadDetector no projeto ARCam e, por último, os resultados obtidos com essa nova funcionalidade.

O capítulo cinco apresenta a conclusão desse trabalho. As lições aprendidas, além de trabalhos futuros para o projeto em desenvolvimento.

CONTEXTO

O presente capítulo tem por objetivo contextualizar o trabalho com relação às tecnologias envolvidas. Conceitos importantes serão apresentados para facilitar a compreensão do trabalho aqui desenvolvido.

1. Realidade Aumentada

1.1. Conceitos Básicos

A Realidade Aumentada, conhecida como RA [1], é uma área de estudo correlata ao campo da Realidade Virtual (RV) que tem como objetivo aumentar o mundo real adicionando elementos virtuais ao mesmo, em tempo real. Essa melhoria pode ser alcançada com objetos visuais, sonoros ou qualquer outro tipo que ative os sentidos do ser humano. Para isso, informações do mundo real precisam ser capturadas e processadas em tempo real para serem adicionadas informações sensíveis à percepção humana.

Entende-se que a RA é também uma evolução da RV, onde o usuário interage com o ambiente misto, ou seja, o mundo real com objetos virtuais inseridos neste ou ainda ocultando objetos da cena real. Assim a RA complementa o mundo real ao invés de sobrepô-lo. Como o próprio nome sugere, a RA aumenta o ambiente para o usuário provendo informações que este por si só não é capaz de perceber apenas com seus próprios sentidos.

A Figura 1 abaixo ilustra uma aplicação *table-top* em RA que consiste em prédios virtuais do centro de Augsburg - Alemanha projetados sobre o mapa real da cidade.

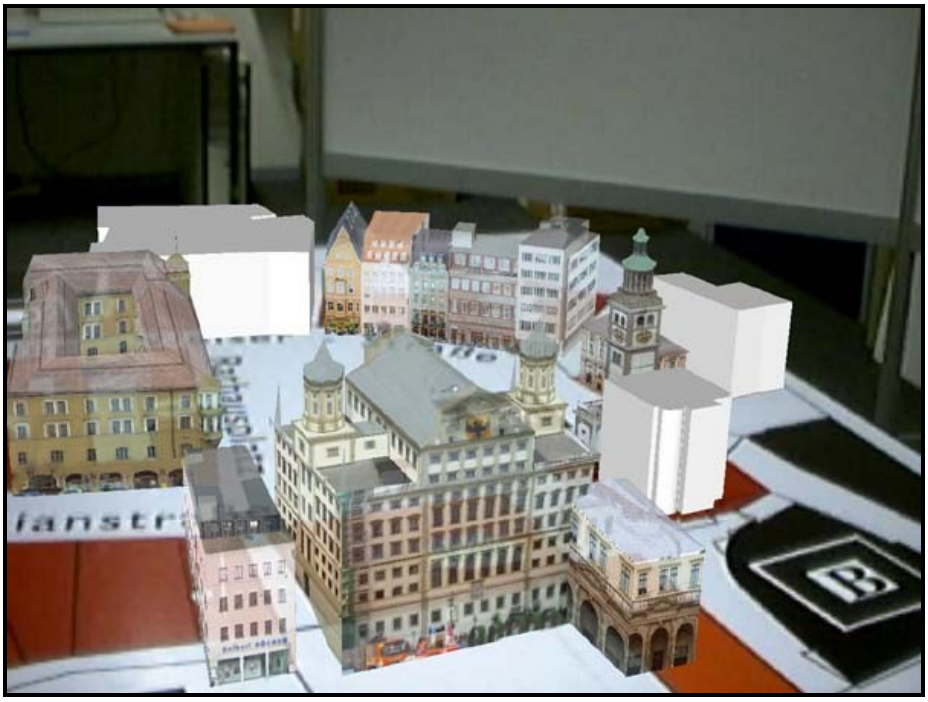


Figura 1: Exemplo de aplicação RA[23]

1.2. Bibliotecas e Ferramentas

Para desenvolvimento de aplicações de RA, várias bibliotecas e ferramentas já estão disponíveis para auxiliar o desenvolvedor. As de maior relevância para esse trabalho serão aqui apresentadas.

ARToolKit

O ARToolKit [2] foi originalmente desenvolvido para servir de apoio na concepção de interfaces colaborativas pelo Dr. Hirokazu Kato, na Universidade de Osaka. E, desde então, tem sido mantido pelo *Human Interface Technology Laboratory* (HIT Lab) da Universidade de Washington e pelo HIT Lab NZ da Universidade de Canterbury, em Christchurch.

O ARToolKit é uma biblioteca de software open source, escrita na linguagem C, para concepção de aplicações de RA. Um exemplo de uma possível aplicação pode ser um personagem virtual tridimensional (3D) que aparece em cima de um marcador, como pode

ser visto na Figura 2. O marcador é um cartão de papel onde é impresso um padrão, no interior de um quadrado de bordas pretas, que corresponde a uma imagem específica previamente conhecida pela biblioteca. Assim, rastreando-se a posição e orientação desse marcador na cena, o ARToolKit é capaz de posicionar sobre o marcador um objeto virtual predefinido.

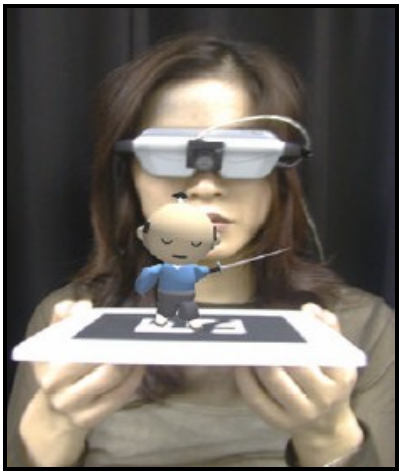


Figura 2: Exemplo de aplicação de RA usando ARToolKit [2]

As principais características do ARToolKit são:

i. *Tracking* para posicionamento e orientação de uma única câmera;

ii. Padrão de marcador utilizando um quadrado de bordas pretas e um padrão desenhado no interior do quadrado, tudo em duas cores (facilita o processamento);

iii. Possibilidade de uso de qualquer marcador, desde que o mesmo obedeça ao padrão;

iv. Distribuição multi-plataforma (por exemplo, SGI IRIX, Linux, MacOS e Windows) com distribuição do código fonte completo.

O funcionamento do ARToolKit é simples. Primeiramente, a câmera posicionada no display do usuário captura as imagens do mundo real e as envia para um computador, onde uma busca nas imagens é feita à procura do padrão do quadrado de bordas pretas, identificado como marcador. Se um marcador for identificado na cena, a partir de uma

série de cálculos matemáticos é possível determinar a posição e orientação relativa do marcador em relação à câmera e, portanto, à percepção do usuário sobre a cena. De posse desse posicionamento, torna-se possível a inserção de um objeto virtual nessas coordenadas, situando-o sobre o marcador e misturando o mundo real com aspectos virtuais. Essa imagem pode então ser visualizada pelo usuário no display.

A Figura 3, especifica o fluxo de atividades do ARToolKit desde a captura da cena até a renderização dos objetos virtuais e da imagem capturada do mundo real (real + virtual).

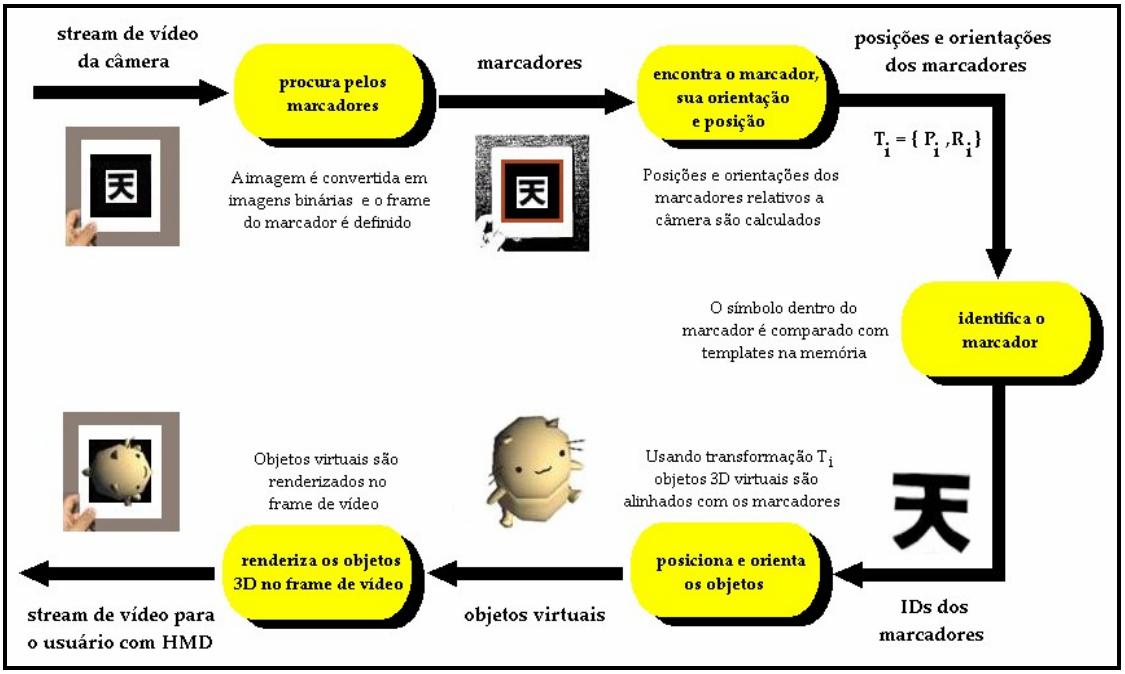


Figura 3: Fluxo de atividades do ARToolKit [2]

ARToolKit Plus

O ARToolKitPlus [24] foi desenvolvido na Graz University Technology dentro do projeto Studierstube, e é uma biblioteca *open source* de RA baseada no ARToolKit, embora

a primeira se preocupe exclusivamente com a questão da detecção de marcadores, não oferecendo funções para captura de vídeo ou renderização de objetos 3D.

Seu código fonte conta com várias otimizações, tais como, utilização de matemática de ponto fixo com o intuito de gerar aplicações eficientes para os dispositivos móveis como, por exemplo, *Personal Digital Assistants* (PDAs) e *smartphones*. Os marcadores utilizados por esta biblioteca são semelhantes aos do ARToolKit, com a diferença que o desenho no interior do quadrado de bordas pretas consiste em uma codificação do identificador do marcador.

ARTag

O ARTag[3] foi desenvolvido pelo *National Research Council of Canada* e inspirado no ARToolKit. O ARTag, tal como o ARToolKit, também consiste em uma biblioteca de padrões. A grande diferença entre essas bibliotecas é o objeto comparado; enquanto o ARToolKit compara imagens, o ARTag compara códigos digitais de 0 (zeros) e 1 (um). O marcador do ARTag também é um quadrado de bordas pretas, mas seu interior é preenchido com uma malha 6x6 de quadrados pretos e brancos que representam os códigos digitais. A Figura 4 ilustra uma aplicação de RA utilizando o ARTag.

Figura 4: Exemplo de aplicação de RA usando ARTag [3]

Desenvolvido baseado no ARToolKit, o ARTag resolveu alguns problemas encontrados nele, como por exemplo:

i. *False positive* - acontece quando a presença de um marcador é reportada, mas não existe nenhum no ambiente;

ii. *Inter-marker confusion* - acontece quando a presença de um marcador é reportada, mas com o ID errado;

iii. *False negative* - acontece quando existe um marcador no ambiente, mas sua presença não é reportada.

Algumas outras vantagens ainda são relevantes no ARTag. Ele possui mais marcadores que o ARToolKit, um total de 2002, dentre eles, 1001 têm o mesmo padrão do ARToolKit, um quadrado de bordas pretas e interior branco, enquanto que os restantes 1001 têm o padrão inverso, um quadrado de bordas brancas e interior preto. Uma outra vantagem sobre o ARToolKit é que ao contrário desse, no ARTag não é necessário que um arquivo contendo todos os padrões seja previamente carregado. Além disso, os marcadores do ARTag podem ser reconhecidos mesmo que uma parte deles esteja oculta, o que não acontece com os marcadores do ARToolKit. A Figura 5 ilustra alguns marcadores do ARTag

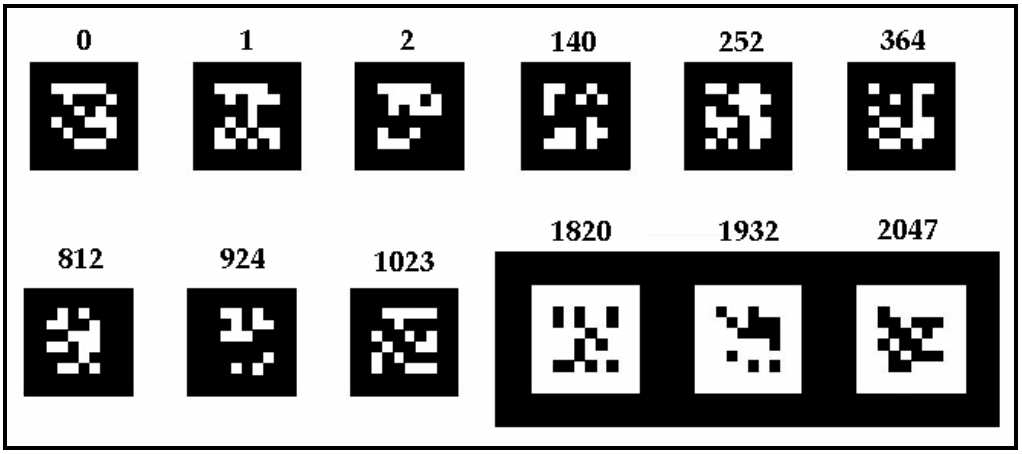


Figura 5: Exemplo de marcadores do ARTag [3]

O processo para o reconhecimento de um marcador no ARTag começa basicamente do mesmo modo que no ARToolKit. Primeiramente, deve ser localizado o contorno da borda do quadrado. Após isso, a região interna que contém a codificação é retirada e são determinados os códigos de 0 e 1 contidos. Todo o processamento subseqüente de identificar e verificar o marcador são feitos digitalmente. Quatro seqüências binárias de 36 bits são obtidas do *array* de códigos binários originários da codificação, uma para cada das quatro possíveis posições rotacionadas; dentre essas, apenas uma será validada pelo processo de decodificação. A seqüência binária codificada no marcador encapsula um ID de 10 *bits*, e os 26 *bits* extras provêem redundância para reduzir as chances de detecção e identificação falsas e também para prover unicamente uma das quatro possíveis rotações.

1.3. Aplicações

Embora recente, a RA apresenta uma vasta gama de aplicações potenciais. Presente em pesquisas espalhadas pelo mundo, sua aplicações atingem as mais diversas áreas. Por essência, a RA tende a melhorar a experiência do usuário interagindo com o mundo real, utilizando aplicações de entretenimento, manutenção de equipamentos, projetos de engenharia, robótica, educação e medicina. À medida que as pesquisas avançam novas aplicações surgem e consolidam seu potencial.

Algumas aplicações foram escolhidas para ilustrarem e enriquecerem o conteúdo apresentado.

Entretenimento

As aplicações no campo do entretenimento são as mais diversas possíveis, porém os jogos ganham destaque. Em RA, por exemplo, já foi desenvolvido o jogo ARQuake [4] baseado no jogo Quake, desenvolvido originalmente para *desktop*. O ARQuake é jogado no

mundo real, fornecendo ao usuário mobilidade para ir onde desejar. A visão do usuário é determinada pela orientação e posição da cabeça do mesmo que usa um HMD (*Head Mounted Display*). A Figura 6 ilustra o ARQuake.

Figura 6: ARQuake: a) Interação do usuário como o jogo; b) *screenshot* do jogo

Uma aplicação interessante que mistura entretenimento e aplicação móvel é o Invisible Train [16], que consiste em um jogo desenvolvido inicialmente para crianças de nível primário. Trata-se de um jogo *multi-player* no qual os jogadores guiam um trem virtual através de um trilho real em miniatura. O diferencial dessa aplicação é que o 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, tudo isto através das telas *touch screen* dos próprios PDAs. A Figura 7 abaixo ilustra a aplicação.

Figura 7: *Invisible Train*, um exemplo de aplicação de RA com *handhelds*

Ainda no campo do entretenimento, foi desenvolvido o projeto Magic Cubes [5]. Este foi desenvolvido para promover interações físicas e sociais entre os membros de famílias. Consiste basicamente em marcadores em formato de cubo que contam histórias, como se fossem um livro de histórias infantis, servindo também como brinquedo que auxilia na construção de casas, montando a mobília para quartos e salas. A Figura 8 ilustra a aplicação.

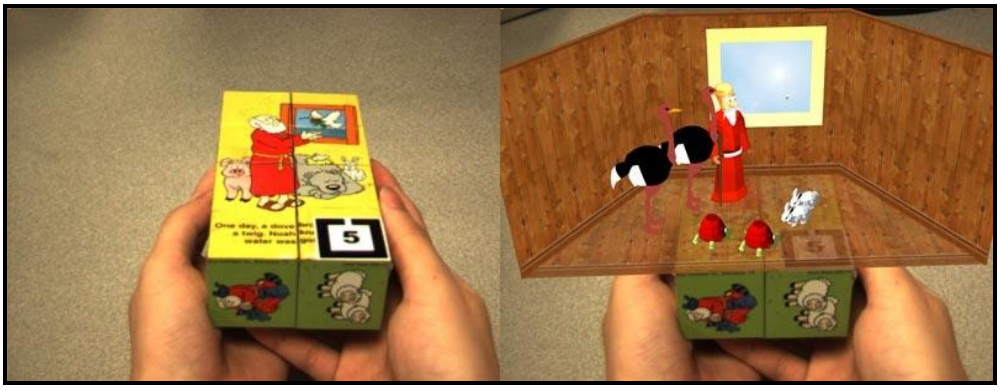


Figura 8: Aplicação *Magic Cubes*

Manutenção de Equipamentos

Ainda como aplicações de RA, surgem ferramentas que auxiliam na manutenção de equipamentos, sejam de escritórios como uma impressora, ou de indústrias como tubulações. A idéia desse tipo de aplicação é fornecer ao usuário as informações de um manual de instruções virtual, enquanto manuseia o equipamento.

Uma das aplicações desenvolvidas auxilia na montagem de uma tubulação industrial, onde são visualizados um mapa bidimensional (2D) da instalação e um modelo 3D das partes de interesse do equipamento real. A Figura 9 ilustra o caso.

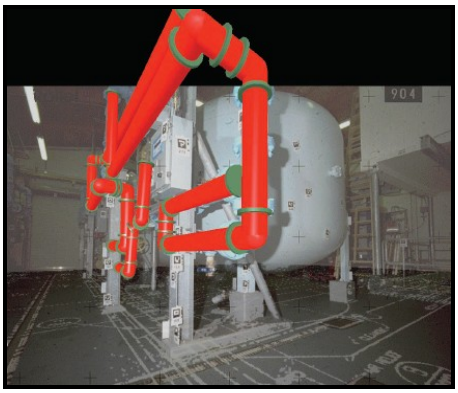


Figura 9: Aplicação de RA na manutenção de tubulações industriais

No caso das impressoras, a aplicação mostra o passo a passo de como reparar o equipamento, como mostra a Figura 10 abaixo.

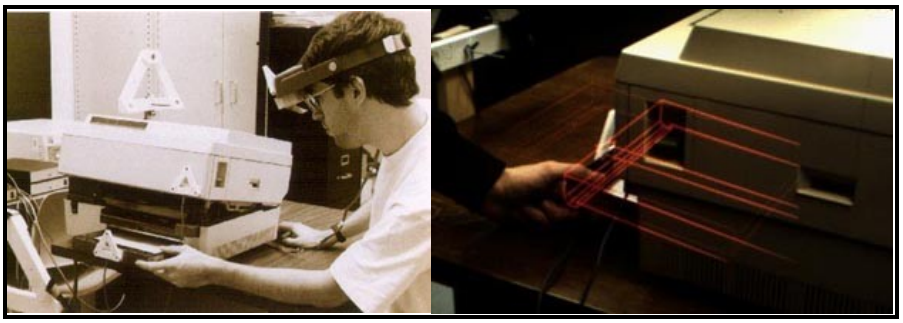


Figura 10: Aplicação de RA na manutenção de impressoras

Projetos de Engenharia e Robótica

Aplicações para auxílio em projetos de engenharia, como mostra a Figura 11 ilustrando uma aplicação que auxilia na visualização em *wireframe* da tubulação, facilitam a percepção do ambiente.

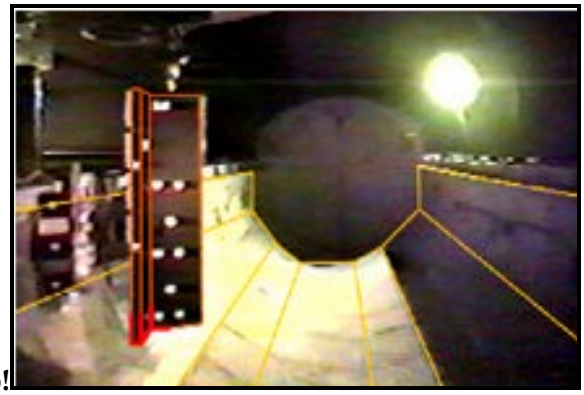
Erro! 

Figura 11: Aplicação de RA em projetos de engenharia

Aplicações na área de robótica e telerobótica como estão ilustradas na Figura 12 abaixo, ajudam na percepção do movimento. Nesse caso, em especial, é possível o operador visualizar a movimentação do robô antes da execução da tarefa e julgar sua viabilidade.

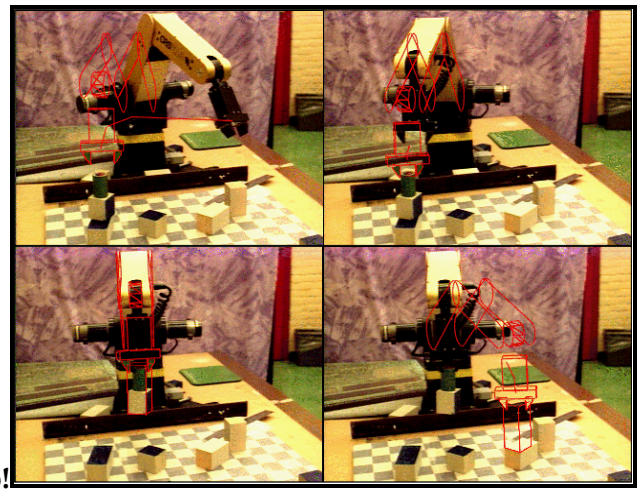
Erro! 

Figura 12: Aplicação de RA em robótica

Medicina

Uma área bem interessante para aplicações de RA é relativa à saúde. A relevância das aplicações nessa área de conhecimento pode significar a diferença entre a vida e a morte para algumas pessoas. O uso eficiente de imagens é extremamente importante para a medicina, e essa é uma das razões da existência de pesquisas que envolvem RA nessa área.

A maioria dessas 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 (com sensores não-invasivos), provêem ao cirurgião a visão necessária da anatomia interna do paciente, e é através do estudo dessas imagens que a cirurgia poderá ser planejada. Os médicos podem usar a tecnologia de RA para a visualização de informações adicionais durante as 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, conforme ilustrado na Figura 13.

A RA também pode ser usada para o treinamento de cirurgiões. 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.

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.

A Figura 13 ilustra uma aplicação de RA na medicina onde a imagem interior do paciente é projetada sobre seu corpo.

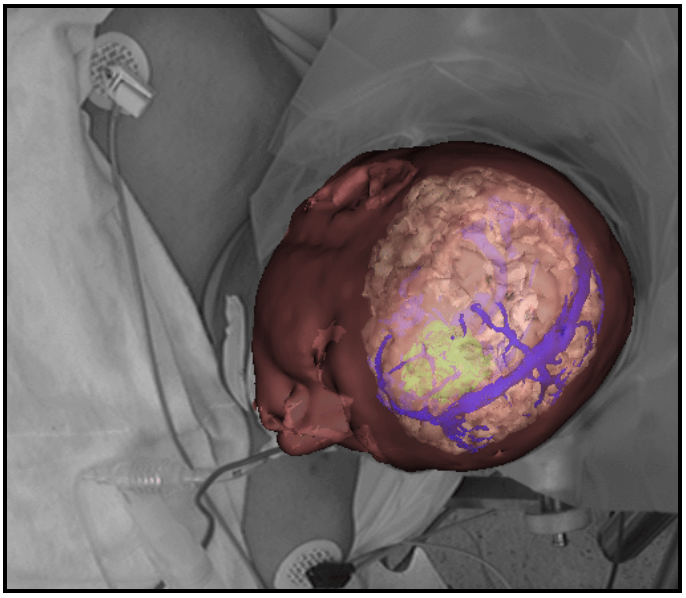


Figura 13: Aplicação de RA em medicina

2. Sistemas Embarcados

Após uma breve descrição de uma das duas áreas de pesquisas que fundamentam esse trabalho, a segunda delas será aqui apresentada. Os sistemas embarcados [22] têm papel fundamental no momento tecnológico que o mundo está passando pelo seu grande poder de aplicação. Sua principal característica é a de sistema dedicado. Assim, ele é construído sobre medida para uma aplicação desejada, reduzindo a ociosidade computacional e o alto consumo de potência e espaço. Em geral, são sistemas pequenos, velozes e confiáveis.

Em muitos casos, são sistemas críticos, construídos para operar em locais com grandes restrições de tempo de processamento, espaço e escassez de potência.

Embora, não necessariamente, um sistema embarcado seja um *hardware* específico desenvolvido para uma aplicação, na maioria dos casos isso acontece, ou seja,

o sistema é projetado em nível de desenvolvimento de *hardware* fazendo uso de uma linguagem para esse propósito. Existem no mercado microprocessadores e microcontroladores com perfis específicos que também são usados para a construção de sistemas embarcados através de linguagens de baixo nível como C; não faz párte do escopo desse trabalho propor uma discussão sobre esses sistemas, visto que o objetivo aqui é construir um *hardware* utilizando uma linguagem de descrição de *hardware* como VHDL (*Very High Speed Integrated Circuits Description Language*).

2.1. FPGAs

Como elementos fundamentais no processo de desenvolvimento de sistemas embarcados estão os FPGAs (*Field Programmable Gate Array*). Por FPGA pode-se entender um *hardware* de propósito geral que têm a capacidade de ser configurado e se transformar em um *hardware* desenvolvido pelo usuário.

Um FPGA é um dispositivo semicondutor que contém componentes lógicos e interconexões programáveis. Os componentes lógicos programáveis podem ser programados para funcionarem 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 [18].

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 *proto-board*. Esses blocos lógicos e interconexões podem ser programados após o processo de fabricação pelo usuário/*designer*, de forma que o FPGA possa realizar a função lógica desejada.

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 Devices*), 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. A Figura 14 abaixo ilustra um dos FPGA`s mais recentes da fabricante Altera, um Stratix II GX com pouco mais que 132 mil elementos lógicos.

Figura 14: FPGA Stratix II GX da Altera

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 apresentar 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 [19], 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.

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,* ou SOCs (*System-on-a-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 RA.

A arquitetura básica típica consiste de um *array* de blocos lógicos configuráveis (CLBs - *Configurable Logic Blocks*) e canais de roteamento. Um bloco lógico típico de FPGA consiste em uma *lookup table* (LUT) de 4 (quatro) entradas, e um *flip-flop*, como mostrado na Figura 15.

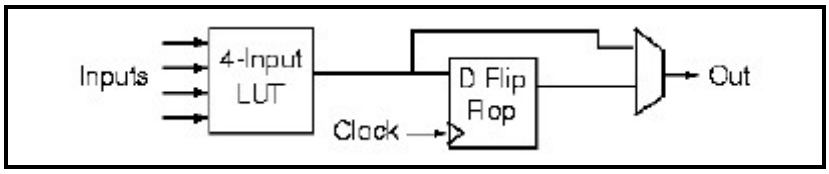


Figura 15: Esquemático básico de um *flip flop*

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 16

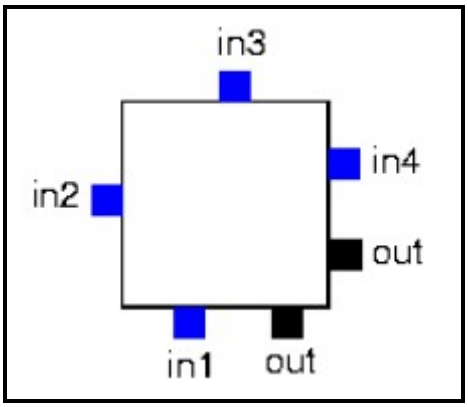


Figura 16: Bloco lógico de um FPGA

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 exploram as 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. 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 às linguagens *assemblers*, 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 mesma 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*.

3. Sistemas Embarcados e o uso de Realidade Aumentada

Dado o panorama geral sobre sistemas embarcados e RA apresentado nas duas seções anteriores, aqui vamos fazer um*a* interseção entre elas e abordar as vantagens e desvantagens de se desenvolver *hardware* embarcado para aplicações de RA.

Por se tratarem de aplicações usadas em tempo real, os sistemas de RA demandam muita capacidade de processamento e uma taxa de quadros/segundo relativamente alta (O mínimo aceitável para uma aplicação de RA é 15 quadros/segundo). Algumas bibliotecas de *software* usadas para desenvolvimento desse tipo de aplicação possuem um nível de abstração de *hardware* muito alto, o que tende a causar um impacto direto no desempenho da aplicação. Assim, torna-se necessário um poder computacional muito superior ao requerido apenas pelos algoritmos da aplicação. O uso de partes de um sistema operacional, como gerenciamento de memória e chaveamento de processos entre aplicações distintas tende a diminuir o desempenho da aplicação. O *hardware* que roda essa aplicação precisa rodar também vários outros processos em paralelo, não sendo assim dedicado.

Por outro lado, aplicações de RA desenvolvidas em *software* possuem vantagens quanto à simplicidade de implementação, se comparada ao desenvolvimento em *hardware*, agilizando o tempo de projeto. Além disso, há bibliotecas prontas e gratuitas, como mostrou a seção 1.2 desse trabalho que auxiliam o projetista. A infra-estrutura necessária para esse tipo de desenvolvimento na grande maioria das vezes não vai além de um PC de porte médio equipado com alguma câmera (*webcam*, por exemplo) e ferramentas de desenvolvimento pagas ou gratuitas, como o ambiente Eclipse [20] .

Já o desenvolvimento de aplicações em *hardware* possui um custo alto, pois envolve a aquisição de placas para a prototipação com FPGA, como um kit de desenvolvimento da Altera, por exemplo, além da aquisição de licenças para ferramentas de desenvolvimento, simulação e validação que, em geral, são caras. O ritmo de pesquisas e desenvolvimento na área de *hardware* não acompanha o ritmo da

área de *softwares*, tendo, portanto uma comunidade mais reduzida e por vezes com escassez de fóruns e listas de discussões.

Mas a grande vantagem dessa forma de desenvolvimento é a capacidade de se criar uma aplicação otimizada e customizada para o propósito desejado, sem ociosidade computacional ou dissipação elevada de potência. São alocados apenas os recursos necessários para implementar o sistema desejado, aqui um detector de quadrados. Vantagens quanto ao tamanho do *hardware* desenvolvido, velocidade de processamento e dedicação total são determinantes ao desempenho do projeto.

De forma sucinta, sabe-se que desenvolver *hardware* não é uma tarefa trivial como pode ser o desenvolvimento de um *software*. Uma cultura tecnológica mínima torna-se necessária como gerenciamento de memória ou uma simples manipulação de *bits* e *bytes*. O uso de conhecimentos sobre álgebra booleana, conversões dos sistemas numéricos binário-hexadecimal-decimal são praxe nesse tipo de desenvolvimento.

Com uma chamada programação de baixo nível é possível obter resultados de operações matemáticas extremamente mais rápido do que quando se realizam as mesmas operações através de bibliotecas de *software* que rodam por cima de um sistema operacional. É exatamente nesse ponto que se justifica uma aplicação de RA (tipicamente processamento de imagens) ser totalmente embarcada.

Processamento de imagens é, por essência, um conjunto de operações matemáticas realizadas sobre matrizes, onde uma ou a associação de várias dessas operações gera um resultado esperado. De forma digital, uma imagem é uma matriz de elementos fundamentais (*pixels*) que carregam informações sobre a tonalidade (cor) de cada um desses pontos da imagem. A escala de cor representada por cada um desses pontos vai depender do número de *bits* destinado à representação de cada um deles. Uma representação bastante interessante é a de um *byte* por cor do padrão RGB (Red, Green, Blue) sendo assim um *pixel* representado por três bytes ou vinte e quatro *bits*.

Ressalta-se aqui, portanto, a grande motivação para o desenvolvimento desse trabalho; aproveitar todo o poder operacional de um *hardware* dedicado em operações matemáticas sobre matrizes. A otimização da matemática através de operações lógicas com vetores binários (*bits*) é feita de forma direta, sem existirem camadas de abstração que separem a captura da imagem do processamento em si. A conexão é feita de forma direta contribuindo para o alto desempenho dos sistemas.

4. O Projeto ARCam

O QuadDetector está sendo desenvolvido como um subprojeto do ARCam.

O projeto ARCam [7] (*Augmented Reality Camera*) está sendo desenvolvido no Grupo de Pesquisas em Realidade Virtual e Multimídia (GRVM) do Centro de Informática da Universidade Federal de Pernambuco (Cin-UFPE), financiado pelo CNPq, com o intuito de prover uma infra-estrutura básica para aplicações embarcadas de RA.

É comum ver soluções de RA onde o usuário necessita carregar consigo um *notebook* para rodar a aplicação, tornando a solução pouco versátil e de alto custo, principalmente no tocante ao peso e ao consumo de energia. Essas aplicações usam uma abordagem de *hardware* e *software*, enquanto que o ARCam está sendo desenvolvido de modo a ser constituído apenas por processadores de uso específico. Todas as funcionalidades são implementadas em linguagem de descrição de *hardware*, resultando num sistema de *hardware* dedicado.

O projeto incorpora um FPGA de médio porte (60 mil elementos lógicos), um sensor de imagens e um monitor de vídeo VGA. O sensor de imagens é responsável pela captura do mundo real. Conectado a ele está o FPGA que realiza o processamento dessas imagens e utiliza o monitor de vídeo, também conectado a ele, para exibir a RA, com informações mistas, do mundo real e as virtuais inseridas no mundo real.

4.1. Objetivo

O objetivo principal do ARCam é construir um arcabouço para desenvolvimento de soluções embarcadas de 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 neste tipo de aplicação. O projeto se propõe a embarcar o ARToolKit descrito na seção 1.2 que é uma biblioteca vastamente utilizada para desenvolver aplicações de RA usando marcadores. A partir desse arcabouço, será possível a criação de diferentes tipos de soluções, como, por exemplo, câmeras inteligentes programadas para realizar inspeção de equipamentos.

4.2. Arquitetura

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, 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 sobre a imagem do mundo real, como uma forma de melhorar a interface com o usuário da aplicação.

A grande flexibilidade na implementação de *firmware* através de uma linguagem de descrição de *hardware* 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 17 mostra o ambiente de desenvolvimento utilizado no projeto, onde foi conectado o sensor de imagem (canto inferior direito) a uma placa de desenvolvimento baseada em FPGA.

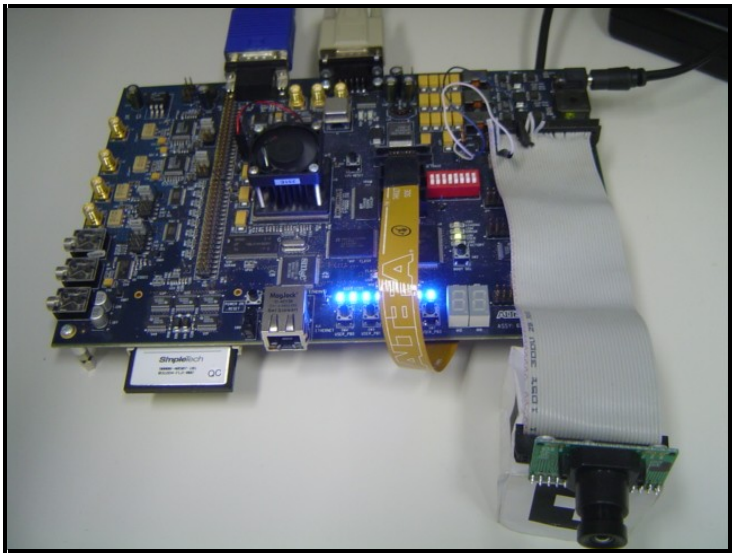


Figura 17: Infra-estrutura do ARCam

O sistema é dividido em módulos que são responsáveis pela aquisição, armazenamento, processamento e projeção de imagens. Em sua arquitetura, ilustrada na Figura 18, o sistema possui o sensor de imagens, uma memória, o módulo de processamento que consiste na aplicação desenvolvida pelo projetista, um multiplexador e o vídeo VGA.

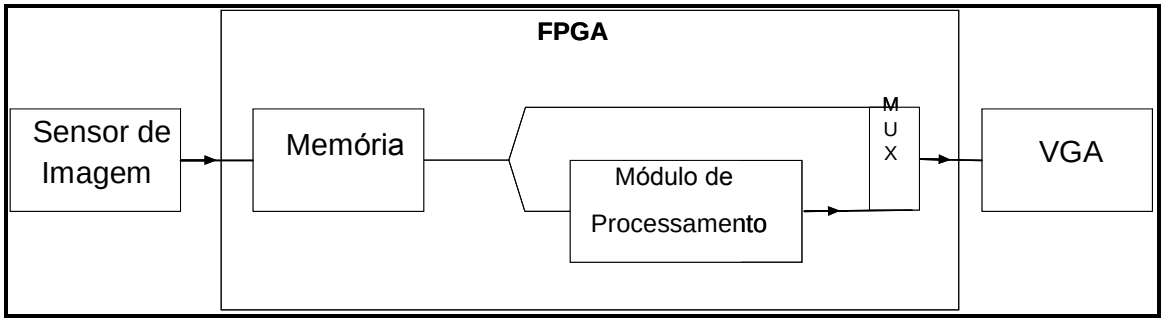


Figura 18: Arquitetura do ARCam

O módulo de processamento é responsável pela versatilidade do projeto. Fornecida essa infra-estrutura básica, o projetista pode implementar seu módulo de processamento representando qualquer operação sobre um *frame* e simplesmente acoplá-lo ao projeto (inseri-lo no lugar do módulo de processamento mostrado na Figura 18). Assim sendo, o módulo de processamento é a aplicação. Também é característico do

ARCam prover algumas bibliotecas básicas de processamento de imagens como binarização, filtros, média de imagens e algumas outras, incluindo a biblioteca desenvolvida neste Trabalho de Graduação, o QuadDetector, que é um módulo para detecção de quadrados.

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 de processamento (aplicação) 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 em forma de FIFO (*First In First Out*). Cada *pixel* que guardado será lido pela aplicação (módulo de processamento) e processado.

A atual arquitetura 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). Um quarto dispositivo, uma memória externa ao FPGA, ainda encontra-se em fase de estudo, para verificar se será viável a sua utilização no projeto de *hardware*.

DETECÇÃO DE QUADRADOS

O objetivo deste trabalho é embarcar o *pipeline* de detecção de marcadores (com exceção da identificação do padrão no interior do marcador), como os utilizados pelas bibliotecas de RA apresentadas no capítulo 2, em *hardware*, constituindo um conjunto de componentes a serem adicionados ao ARCam. Este *pipeline* visa, basicamente, detectar quadrados de bordas pretas.

O processo de detecção de quadrados em uma imagem, como esperado, é o resultado de uma série de operações matemáticas realizados sobre a mesma. São determinantes basicamente quatro passos: a detecção de um contorno (*border tracing* [25]), a redução de vértices (*vertex reduction* [26]), a aproximação poligonal [26] e, por último, a classificação de um polígono.

As seções seguintes descrevem cada um dos quatro passos supracitados, que foram desenvolvidos nesse trabalho.

1. Detecção de Contorno (Border Tracing)

A detecção de contorno, ou *border tracing*, consiste na etapa preliminar ao objetivo final da detecção de quadrados, isso porque um quadrado representa um contorno fechado. Assim, essa etapa é responsável por vasculhar a imagem em busca desses contornos. Para isso, antes do algoritmo ser executado a imagem precisa ser binarizada (transformada para preto e branco) de modo que a identificação de *pixels* vizinhos e iguais se torne mais fácil, trabalhando com a possibilidade de um *pixel* ser preto (nível lógico 0) ou branco (nível lógico 1).

O primeiro passo é identificar um *pixel* preto e o segundo passo é saber se o *pixel* vizinho a ele também é da mesma cor. Para isso, podem ser usadas duas

abordagens de busca, uma conhecida como 4-*connectivity* [27] e a outra 8-*connectivity* [27].

Na abordagem 4-*connectivity* apenas quatro posições, daí o deriva o nome, vizinhas são checadas, mais especificamente a superior e a inferior, além das posições à direita e à esquerda do *pixel* em análise. A Figura 19 evidencia o processo de busca 4-*connectivity*. O *pixel* em análise está posicionado na origem do sistema de coordenadas apresentado.

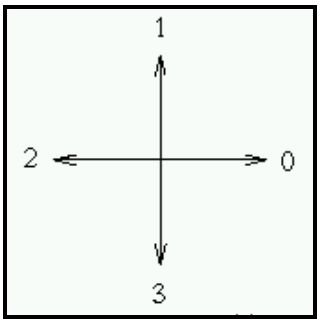


Figura 19: Processo de busca de vizinhos 4-*connectivity*

Na abordagem 8-*connectivity* oito *pixels* vizinhos são analisados, levando em consideração os quatro propostos pela abordagem 4-*connectivity*, mais quatro vizinhos das diagonais em relação ao pixel analisado. A Figura 20 mostra o processo de busca dos oito vizinhos ao *pixel* analisado.

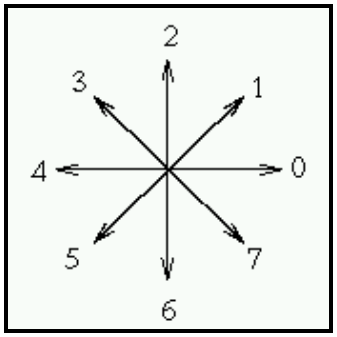


Figura 20: Processo de busca de vizinhos 8-*connectivity*

Nesse trabalho, foi usada a abordagem 8-*connectivity* por ser mais completa e tornar o processo de detecção mais fiel à realidade. A título de ilustração, podemos considerar um objeto como o da Figura 21 . O mesmo se apresenta como um quadrado, mas uma implementação utilizando 4-*connectivity* não seria capaz de rastrear esse contorno e, por isso, o objeto não seria classificado como um quadrado. Já o modelo 8-*connectivity* seria capaz de identificar o objeto na cena e prosseguir com a execução do algoritmo até a classificação dos polígonos. O motivo dessa identificação ser feita apenas utilizando uma abordagem 8-*connectivity* se deve ao fato de todos os vizinhos à qualquer um dos *pixels* da figura se situarem em suas diagonais, condições não checadas na abordagem 4-*connectivity*

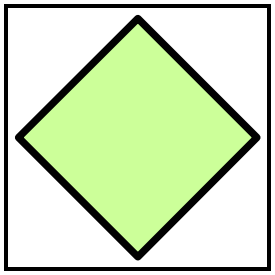


Figura 21: Quadrado rotacionado

Identificado o primeiro *pixel* preto na cena, as coordenadas do mesmo são salvas e inicia-se a busca por outros pontos vizinhos e pretos também. Assim, vai-se percorrendo todo um contorno e sempre comparando as coordenadas de cada ponto preto encontrado com as coordenadas do primeiro *pixel* preto identificado na cena. Quando isso acontece, o algoritmo conseguiu fechar um contorno e então pode-se salvar esse caminho percorrido como uma borda fechada.

Nessa etapa de processamento, o interesse do algoritmo é identificar contornos na cena, não importando a forma do traçado identificado. A partir desse ponto, algumas outras etapas do processamento se encarregarão de trabalhar esse contorno até a classificação final do mesmo, sendo considerado um quadrado ou não.

A Figura 22 abaixo mostra um contorno presente na cena e o resultado do processamento do mesmo na detecção de contorno.

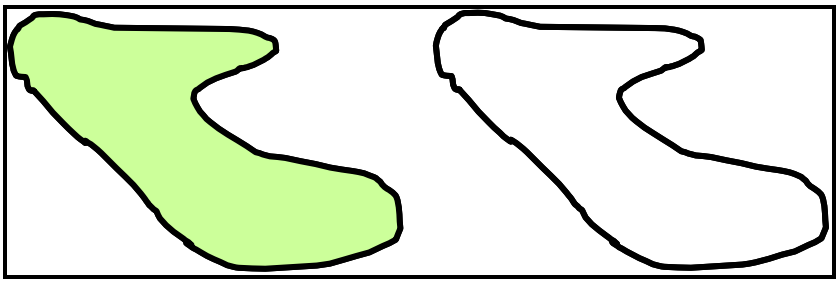


Figura 22: Detecção de contorno: a) Imagem de entrada; b) Resultado do processamento de detecção de contorno

2. Redução de Vértices (Vertex Reduction)

A redução de vértices tem papel importante na detecção de contorno, pois é responsável por simplificar o contorno. Sua função é diminuir o número de *pixels* a serem processados a fim de reduzir o custo computacional.

Dado um determinado contorno, o mesmo se apresenta como um conjunto de *pixels* que, numa visão macro, são vizinhos que carregam basicamente a mesma informação. A redução de vértices serve para simplificar o contorno, descartando *pixels* muito próximos uns dos outros. A quantidade de *pixels* a descartar é determinada pela implementação do projetista.

De maneira geral, o que a redução de vértices faz é descartar pontos cuja distância até um ponto inicial seja menor que uma tolerância ε determinada. Assim, o descarte de *pixels* determinado pelo projetista é dado ajustando essa tolerância ε. Quanto maior for o valor de ε, menos pontos terá o resultado da redução e menos fiel ao objeto estudado também será o resultado. Por conseguinte, quanto menor for o valor de ε, mais pontos terá o contorno resultante, sendo mais fiel ao objeto estudado. A Figura 23

mostra a imagem entrada para a redução de vértices e três resultados desse algoritmo para três valores de ε, onde ε1 > ε2 > ε3 respectivamente.

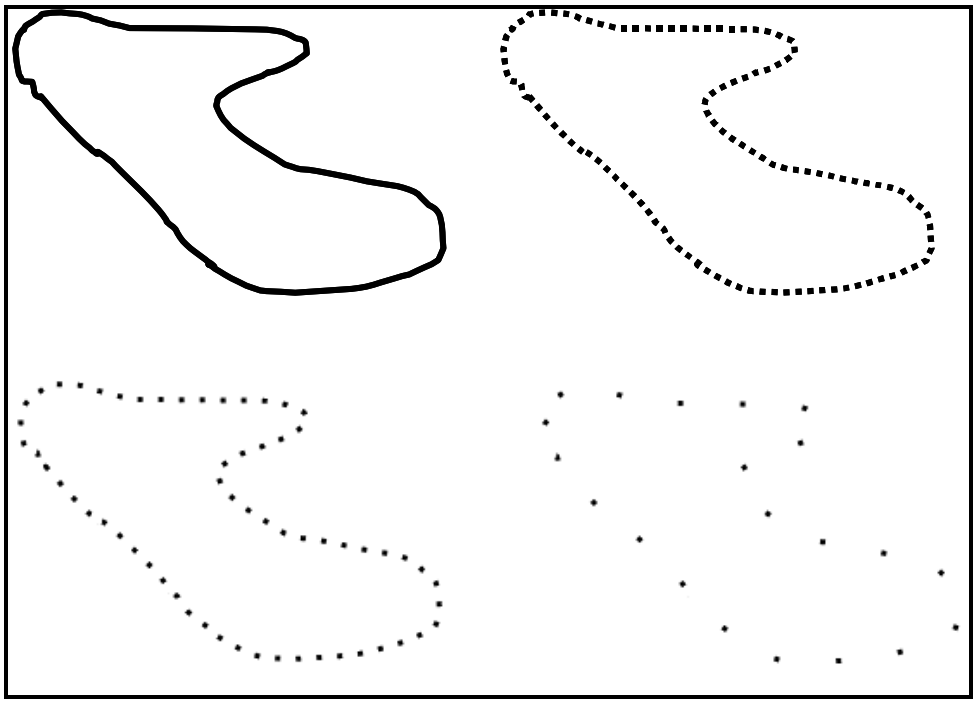


Figura 23: Redução de Vértices: a) Imagem de Entrada; b) Resultado da redução de vértices para ε1 ; c) Resultado da redução de vértices para ε2; d) Resultado da redução de vértices para ε

O algoritmo de redução de vértices descrito é conhecido como algoritmo força-bruta para redução de um polígono. Existem também outras formas de se simplificar um polígono que não serão discutidas neste trabalho, pois não é o seu objetivo. O método de redução de vértices apresentado foi implementado nesse trabalho e se mostrou eficiente (ver capítulo QuadDetector).

3. Aproximação Poligonal

O algoritmo de aproximação poligonal é fundamental para a detecção de quadrados. O objetivo dessa etapa é colher os pontos do polígono suficientes para

aproximá-lo. De antemão, para um quadrado, sabe-se que apenas os quatro vértices são suficientes para construí-lo.

A aproximação poligonal trabalha de forma iterativa, construindo segmentos de retas com pontos do polígono resultantes da redução de vértices e calculando a distância desses segmentos a todos os outros pontos do polígono. Considere um segmento S1 formado pelos pontos P1 e P2. Para cada ponto Pi do polígono, se a distância de S1 à Pi for maior que um ψ (tolerância no contexto da aproximação poligonal), dois novos segmentos S2 e S3 são criados onde S2 é formado pelos pontos P1 e Pi e S3 pelos pontos Pi e P2, e o ponto Pi é adicionado ao vetor resultante da aproximação. Caso a distância seja menor que ψ, o ponto Pi é descartado.

A Figura 24 ilustra os passos do algoritmo da aproximação poligonal

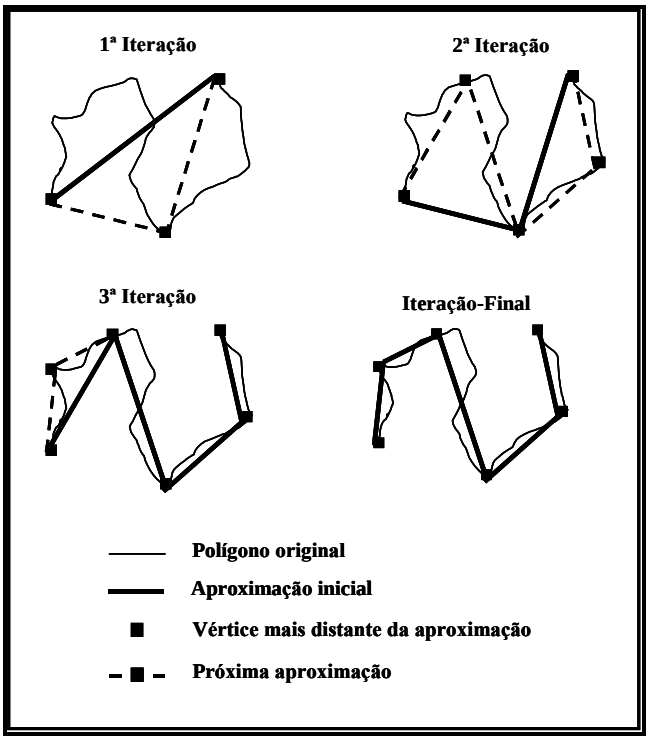


Figura 24: Passos da aproximação poligonal

Mais especificamente, no algoritmo de aproximação poligonal, os pontos extremos de um polígono são conectados com uma linha reta como uma aproximação

grosseira e inicial do polígono. Assim o polígono inteiro é determinado computando as distâncias de todos os vértices intermediários ao segmento usado. Sendo todas as distâncias menores que a tolerância ψ determinada, o algoritmo pára e o vetor resultante é composto pelos pontos usados no traçado dos segmentos, como mostrou a Figura 24 .

Vale destacar que, para pontos situados sobre o segmento analisado, a distância dele ao segmento será zero. Assim, pode-se concluir que para um quadrado, o resultado da aproximação poligonal resultaria em apenas quatro vértices resultantes, pois traçadas as quatro arestas ligando os quatro vértices, os outros pontos estariam sobre as arestas com distância para elas igual a zero.

A Figura 25 abaixo indica os passos do algoritmo de aproximação poligonal para um quadrado resultante da redução de vértices.

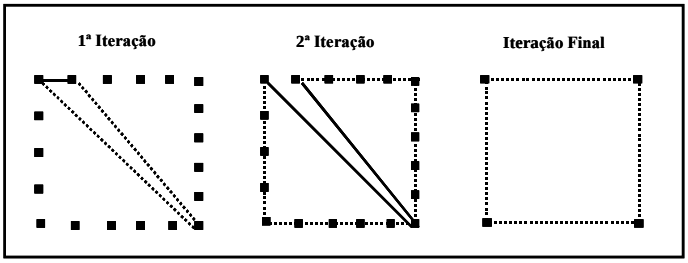


Figura 25: Passos da aproximação poligonal para um quadrado

4. Classificação dos Polígonos

A etapa de classificação de polígonos é o último passo para a detecção de um quadrado. A base para a classificação consiste em analisar a convexidade do polígono, bem como os ângulos formados pelas arestas que ligam os pontos da aproximação, ou seja, os vértices do quadrado.

Em se tratando de um quadrado, o primeiro teste a fazer é checar se o mesmo possui apenas quatro vértices. Feito isso, parte-se para a classificação quanto à convexidade ou concavidade do polígono. Para analisar se o polígono é convexo,

constroem-se quatro triângulos a partir da permutação três a três dos quatro vértices do mesmo e computam-se as áreas desses triângulos a partir do produto vetorial dos três vetores associados que desenham a figura, vide Figura 26 abaixo.

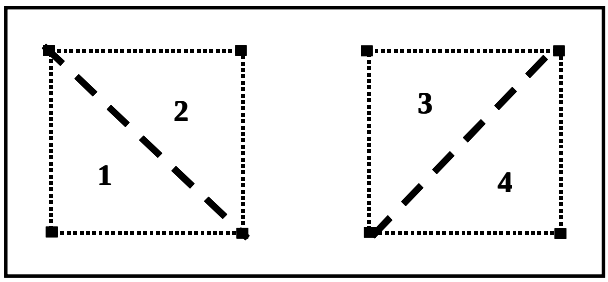


Figura 26: Passos da classificação de polígonos

Caso todos os quatro resultados possuam o mesmo sinal (positivo ou negativo) o polígono em questão é convexo, caso contrário é côncavo. Não cabe aqui discutir o porquê dessa classificação, visto que esse processo já é provado e amplamente utilizado no processo de classificação de polígonos. Referências [28] [29] de processamento de imagens e álgebra linear abordam vastamente esse conceito.

Após a classificação quanto à convexidade, o próximo passo a ser analisado são os ângulos formados pelas arestas da figura. Aqui cabe a introdução de uma tolerância ζ para o ângulo em questão, pois pode-se, e deseja-se considerar um quadrado como um polígono convexo de quatro vértices cujos ângulos internos sejam 80°, por exemplo, e não necessariamente 90°. Assim, é o valor de ζ que determina a margem de tolerância a que pode-se submeter o algoritmo de classificação quanto aos ângulos internos do polígono.

Para a obtenção dos respectivos ângulos, calculam-se os produtos escalares dos dois vetores arestas, associados a cada vértice. Sabe-se de antemão, também do estudo de álgebra linear e processamento de imagens, que a definição de produto escalar é a mostrada na Figura 27:

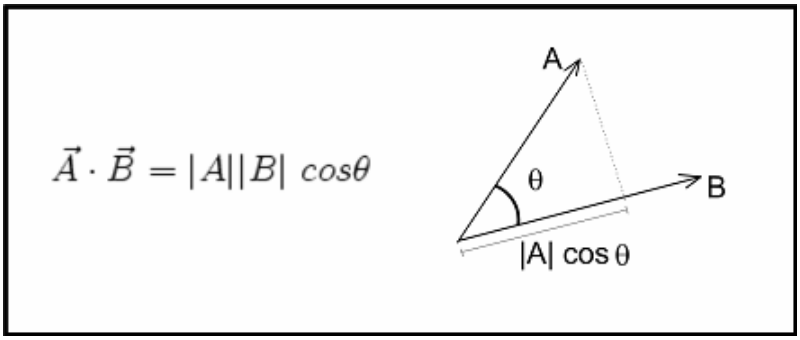


Figura 27: Classificação de polígonos: a) Expressão do produto escalar entre dois vetores A e B; b) Ângulo formado por dois vetores A e B

Assim, de posse dos vetores e do valor de seus produtos escalares, podemos extrair o ângulo formado entre os mesmos e classificar o polígono como sendo ou não um quadrado.

QUADDETECTOR

O presente capítulo deste trabalho tem por finalidade apresentar o QuadDetector, descrevendo sua implementação e testes. Serão descritos a metodologia adotada, bem como os ambientes de desenvolvimento utilizados e a arquitetura construída. Os módulos implementados, as dificuldades encontradas e os resultados obtidos com o *hardware* são, por fim, apresentados.

1. Metodologia adotada

Para o desenvolvimento desse trabalho duas linguagens computacionais foram utilizadas, VHDL e Java. A primeira delas foi utilizada para implementação do *core* do projeto. Através dela foi possível desenvolver todo o sistema e sintetizá-lo em ambiente adequado, que será detalhado na próxima seção, a fim de testá-lo em FPGA. A segunda foi utilizada para implementação de alguns algoritmos em *software* a fim de se obterem testes comparativos de resultados para verificação funcional do protótipo desenvolvido. A Figura 28 ilustra a metodologia aplicada durante o desenvolvimento

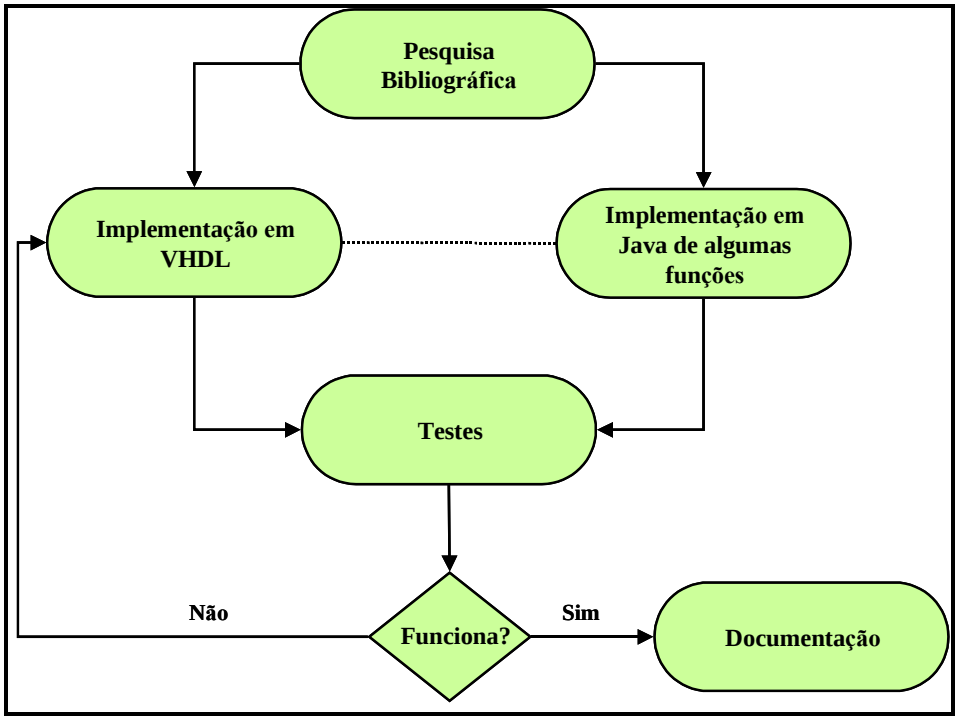


Figura 28: Metodologia de desenvolvimento

A pesquisa bibliográfica compreendeu a busca 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 a mesma, também foi definido o escopo e o cronograma do trabalho.

A segunda etapa foi responsável pela elaboração do protótipo do QuadDetector, além de alguns algoritmos em *software* para a validação dos módulos em VHDL.

A terceira etapa compreendeu a fase de testes de todo o protótipo, muito embora durante o desenvolvimento modularizado de cada uma das partes testes tenham sido efetuados para que se pudesse dar prosseguimento ao desenvolvimento do módulo subseqüente. Nessa fase de testes, resultados obtidos através da execução dos algoritmos implementados em *software* foram utilizados como modelo comparativo para avaliação dos resultados do protótipo. A passagem para a próxima etapa só foi realizada a partir do momento que os resultados esperados foram atingidos.

Por fim, a quarta e última etapa correspondeu a documentação do projeto, bem como a escrita do Trabalho de Graduação aqui apresentado, incluindo revisões e correções.

2. Ambiente de Desenvolvimento

Os ambientes utilizados para o desenvolvimento desse projeto podem ser divididos em três aspectos: o *hardware* (FPGA + câmera de captura) usado; o ambiente para o desenvolvimento em VHDL e o ambiente para o desenvolvimento em Java.

2.1. Hardware

A solução ARCam, que incorpora os resultados produzidos neste Trabalho de Graduação, 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 para converter as cores digitais em um nível de tensão analógico nos três canais de cores do monitor (vermelho, verde e azul).

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 29 . 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.

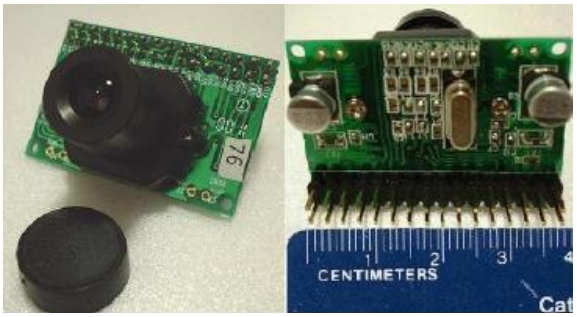


Figura 29: Sensor de imagem utilizado

Os principais sinais fornecidos pelo sensor de imagem são:

i. sincronismo horizontal: informa quando uma linha acaba de ser lida;

ii. sincronismo vertical: informa quando um quadro é concluído;

iii. sincronismo de *pixel*: informa que o valor do *pixel* está disponível no barramento com suas cores;

iv. paridade do quadro: informa se a linha é par ou impar;

v. barramento Y: contém 8 bits para representar uma cor;

vi. 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), 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 e outros dados que caracterizam o FPGA utilizado são apresentados na Tabela 1 .

Tabela 1: Características do FPGA utilizado

ALMs	**24,176**
Adaptative *look-up* tables (ALUTs)	**48,352**
Equivalent Les	**60,44**
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* para prototipação do circuito, incluindo o FPGA, faz parte de um *kit* de desenvolvimento fornecido pela Altera. A Figura 30 mostra a placa utilizada nesse projeto já com a câmera acoplada; neste cenário o usuário interage com o sistema utilizando dois objetos, um azul e outro vermelho, para jogar um jogo embarcado, o Pong [7], desenvolvido para validação da arquitetura do ARCam . 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 baixo custo.

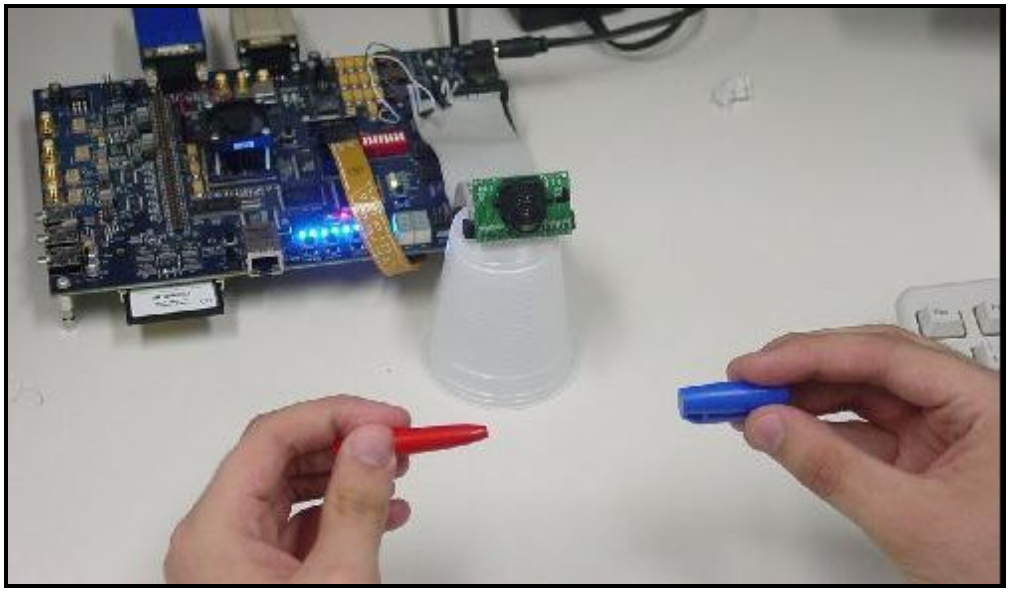


Figura 30: *Kit* Altera utilizado

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 na atual versão do projeto, foram utilizados um monitor DELL CRT de 15 polegadas e outro, HP, também CRT, este com 19 polegadas.

2.2. Ambiente VHDL

VHDL é uma linguagem de descrição de sistemas eletrônicos digitais. Ela surgiu a partir de um programa do governo norte americano chamado *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 o alto custo provenientes da prototipação em *hardware*.

Por se tratar de um sistema completamente desenvolvido em *hardware*, todos os módulos do QuadDetector foram implementados usando a linguagem VHDL. Através dela, foi possível simular o comportamento funcional de cada submódulo criado e verificar sua corretude, antes de realizar os testes diretamente no FPGA.

A implementação dos módulos foi realizada no ambiente Altera Quartus II versão 5.2. O ambiente faz parte do *kit* para desenvolvimento adquirido que conta com a placa de prototipação mais o ambiente acima citado.

As simulações foram realizadas em dois cenários: O primeiro deles utilizando a ferramenta de simulação funcional do Quartus II, que permite a simulação através de arquivos que contém os vetores de entrada do sistema, nestes mesmos arquivos são salvos os resultados. A visualização dos resultados obtidos foi feita a partir do processamento isolado de cada módulo, característica essa difícil de ser analisada quando o sistema está rodando no FPGA programado. O outro cenário adotado foi o da simulação no ambiente real do projeto, o FPGA. Para isso, a ferramenta SignalTap Logic Analyser, que vem associada ao ambiente Quartus II, 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*).

A Figura 31 ilustra quatro cenários diferentes do ambiente Quartus II: O desenvolvimento de *hardware* através de modelagem esquemática, depois em linguagem VHDL, no terceiro momento uma simulação usando a ferramenta de simulação do Quartus e, por último, uma simulação utilizando o SignalTap Logic Analyser.

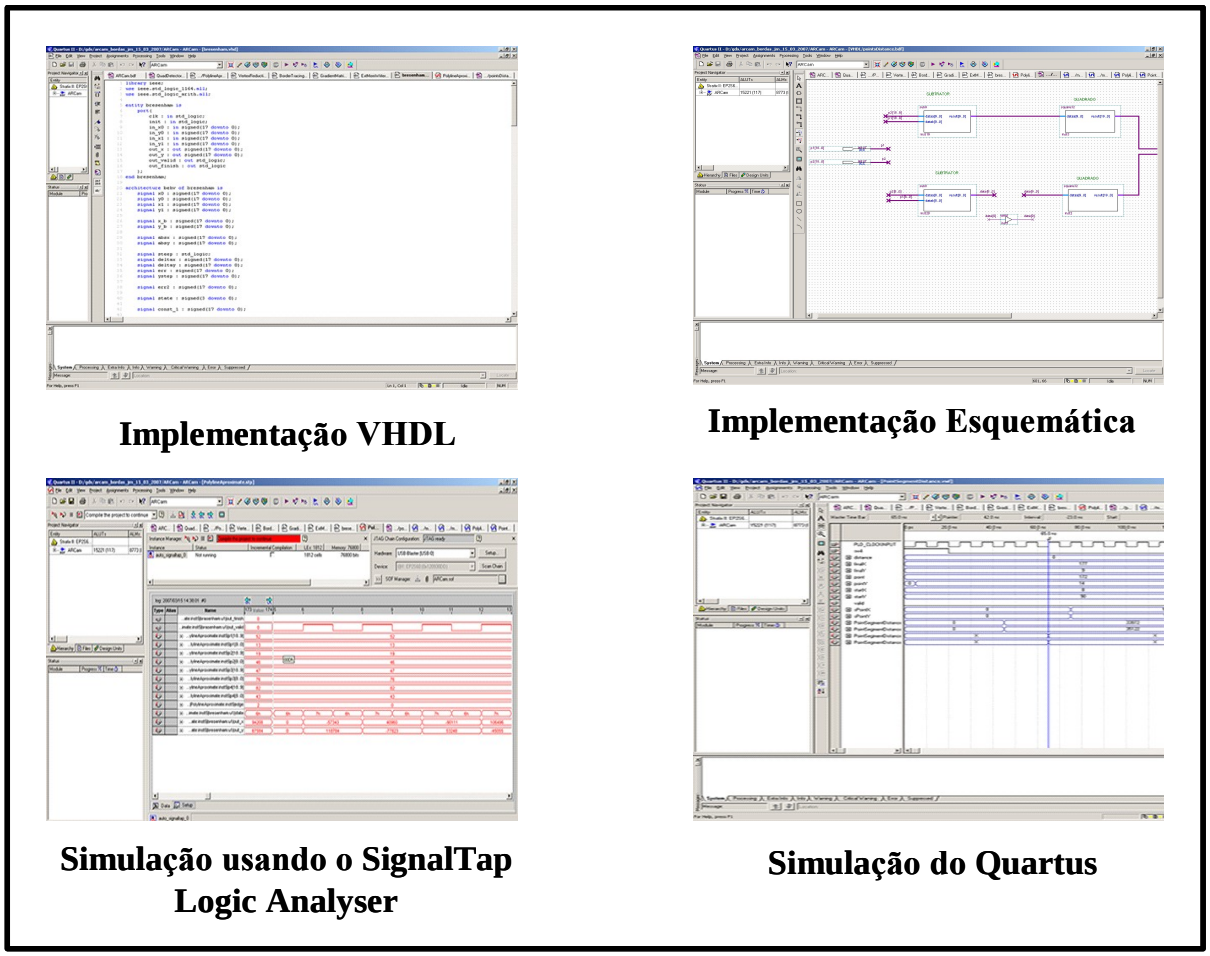


Figura 31: Ambiente Quartus

3. Módulos do Projeto

Esta seção é responsável por apresentar ao leitor não só o módulo QuadDetector, como todos os submódulos que o compõem. São apresentados os módulos mais importantes desenvolvidos durante este trabalho e alguns forencidos pela própria Altera, como memória e operações aritméticas fundamentais tais como soma/subtração, multiplicação/divisão, raiz quadrada e segunda potência.

A descrição dos módulos se dará em duas etapas, apresentando primeiramente os módulos utilizados pelo QuadDetector mas que não foram implementados pelo autor e, por conseguinte, os módulos implementados durante o trabalho para viabilizar o QuadDetector.

3.1. Módulos fornecidos

Os módulos utilizados no QuadDetector que não foram implementados durante esse Trabalho de Graduação, são, na sua maioria, fornecidos pela Altera, e são disponibilizados dentro do próprio ambiente do Quartus II. O único que não pertence ao repertório de funções da Altera, que foi utilizado dentro do módulo QuadDetector, é o Hardwire [21]. Na sequência são apresentados os módulos `memContour`, `sqrt`, `square32`, `mult9`, `add9`, `sub9` e `breseham`.

memContour

Esse módulo corresponde ao espaço de memória onde é armazenado o resultado do *border tracing*, ou seja, as bordas rastreadas no *frame*. O conteúdo armazenado indica as coordenadas no plano xy do *pixel* encontrado. Um mesmo bloco foi também utilizado para armazenar os vértices resultantes do segundo passo da detecção, a redução de vértices. A Figura 32 ilustra o módulo.

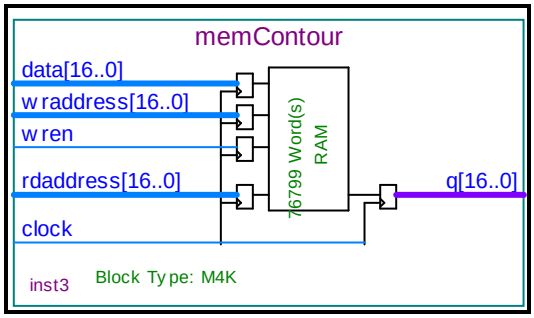


Figura 32: Módulo `memContour`

A Tabela 2 mostra os sinais de entrada/saída do módulo especificado.

Tabela 2: Sinais de entrada/saída do módulo `memContour`

Nome	Tamanho	Tipo	Descrição
data	17	Entrada	É o conteúdo que se deseja armazenar na memória
wraddress	17	Entrada	Endereço do conteúdo a ser armazenado
wren	1	Entrada	Sinal de controle que indica o momento que a memória pode ser escrita
rdaddress	17	Entrada	Endereço do conteúdo a ser lido
clock	1	Entrada	Relógio de sincronismo do sistema
q	17	Saída	Conteúdo lido da memória

O vetor de dados armazenado carrega nos 9 *bits* mais significativos a coordenada x do ponto estudado, enquanto que os outros 8 *bits* indicam a coordenada y do ponto. A escolha desse tamanho se dá de acordo com o número necessário de *bits* para representar um ponto em um sistema de coordenadas cartesianas que mapeiem uma resolução 320x240. Por isso, os 9 *bits* para a coordenada x conseguem representar números inteiros positivos em um intervalo de 0 a 511 (o que satisfaz a condição da resolução utilizada de 320 colunas). Já os últimos 8 *bits* chegam a representar valores

inteiros positivos no intervalo de 0 a 255 (também suficientes para o plano de no máximo 240 linhas).

Vale lembrar que um mesmo componente `memContour` também é utilizado para armazenar o resultado da redução de vértices, sendo o QuadDetector composto por dois módulos `memContour`.

sqrt

Esse módulo calcula a raiz quadrada de um número representado por um vetor de 32 *bits*. Sua construção foi baseada na função `altsqrt`, disponível no ambiente Quartus.

A Figura 33 ilustra o módulo

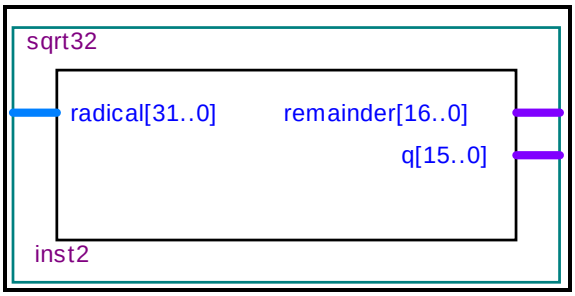


Figura 33: Módulo `sqrt`

A Tabela 3 mostra os sinais de entrada/saída do módulo especificado.

Tabela 3: Sinais de entrada/saída do módulo `sqrt`

Nome	Tamanho	Tipo	Descrição
radical	32	Entrada	Corresponde ao valor de entrada que se deseja calcular a raíz quadrada
remainder	17	Saída	Corresponde ao resto da divisão, corresponderia a parte decimal, caso fosse usada
q	16	Saída	É o resultado da radiciação

`square32`

Esse módulo calcula a potência de dois de um número representado por um vetor de 10 *bits*. Sua construção foi baseada na função `altsquare`, disponível no ambiente Quartus.

A Figura 34 ilustra o módulo.

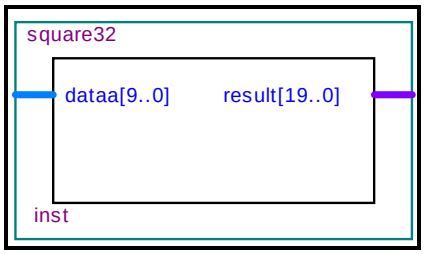


Figura 34: Módulo `square32`

A Tabela 4 mostra os sinais de entrada/saída do módulo especificado.

Tabela 4: Sinais de entrada/saída do módulo `Square32`

Nome	Tamanho	Tipo	Descrição
dataa	10	Entrada	Corresponde ao valor de entrada que se deseja calcular a segunda potência
result	20	Saída	Corresponde ao resultado da potência de dois do vetor de entrada

`mult9`

Esse módulo calcula a multiplicação de dois números representados por vetores de 10 *bits*. Sua construção foi baseada na função `lpm_mult`, disponível no ambiente Quartus.

A Figura 35 ilustra o módulo.

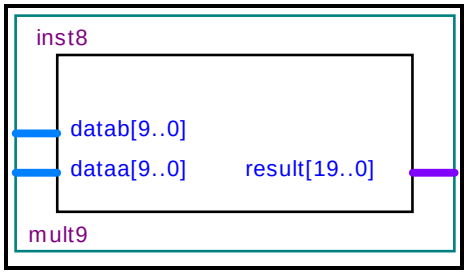


Figura 35: Módulo `mult9`

A Tabela 5 mostra os sinais de entrada/saída do módulo especificado.

Tabela 5: Sinais de entrada/saída do módulo `mult9`

Nome	Tamanho	Tipo	Descrição
dataa	10	Entrada	Corresponde ao primeiro termo da multiplicação
datab	10	Entrada	Corresponde ao segundo termo da multiplicação
result	20	Saída	Representa o resultado da multiplicação

`add9`

Esse módulo calcula a soma de dois números representados por vetores de 10 bits. Sua construção foi baseada na função `lpm_add_sub`, disponível no ambiente Quartus.

A Figura 36 ilustra o módulo.

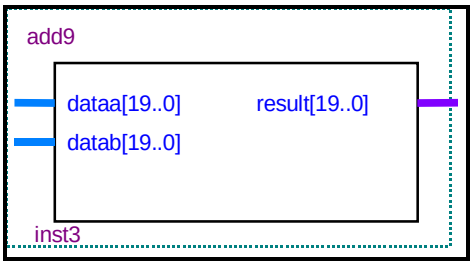


Figura 36: Módulo `add9`

A Tabela 6 mostra os sinais de entrada/saída do módulo especificado.

Tabela 6: Sinais de entrada/saída do módulo `add9`

Nome	Tamanho	Tipo	Descrição
dataa	10	Entrada	Corresponde ao primeiro termo da adição
datab	10	Entrada	Corresponde ao segundo termo da adição
result	10	Saída	Representa o resultado da adição

sub9

Esse módulo calcula a subtração de dois números representados por vetores de 10 *bits*. Sua construção foi baseada na função `lpm_add_sub`, disponível no ambiente Quartus.

A Figura 37 ilustra o módulo.

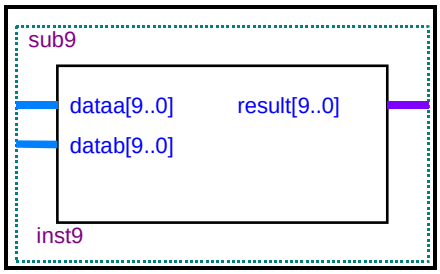


Figura 37: Módulo `sub9`

A Tabela 7 mostra os sinais de entrada/saída do módulo especificado.

Tabela 7: Sinais de entrada/saída do módulo `sub9`

Nome	Tamanho	Tipo	Descrição
dataa	10	Entrada	Corresponde ao primeiro termo da subtração
datab	10	Entrada	Corresponde ao segundo termo da subtração
result	10	Saída	Representa o resultado da subtração

`breseham`

Esse módulo foi implementado em um módulo de processamento do ARCam, o Hardwire, que é usado para gerar arestas ligando pontos passados como parâmetro. No escopo do QuadDetector, esse módulo foi utilizado ao fim da classificação de polígonos, pois se o contorno se classificar como quadrado, os vértices do mesmo são passados para o `bresenham`, que liga os pontos pintando na tela o quadrado detectado.

A Figura 38 ilustra o módulo.

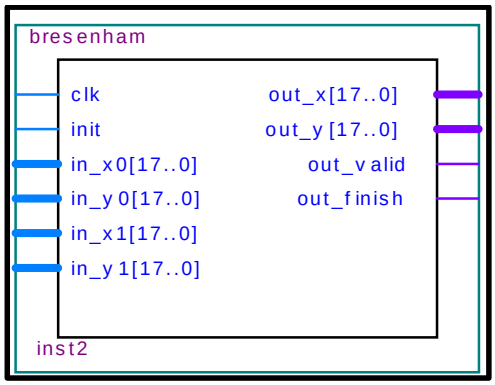


Figura 38: Módulo `bresenham`

A Tabela 8 mostra os sinais de entrada/saída do módulo especificado.

Tabela 8: Sinais de entrada/saída do módulo `breseham`

Nome	Tamanho	Tipo	Descrição
clk	1	Entrada	relógio de sincronismo do sistema
init	1	Entrada	sinal que indica começo de processamento
in_x0	18	Entrada	Coordenada x do primeiro vértice
in_x1	18	Entrada	Coordenada y do primeiro vértice
in_y0	18	Entrada	Coordenada x do segundo vértice
in_y1	18	Entrada	Coordenada y do segundo vértice
out_x	18	Saída	Coordenada x de um ponto xy pertencente à aresta que une os dois vértices passados como parâmetro
out_y	18	Saída	Coordenada y de um ponto xy pertencente à aresta que une os dois vértices passados como parâmetro

out_valid	1	Saída	Sinal que indica o ponto xy válido pertencente à aresta
out_finish	1	Saída	Sinal que indica fim de processamento de aresta para os pontos da aresta em processamento

O funcionamento do bresenham é simples: ao se passar dois pontos de coordenadas (x0,y0) e (x1,y1) ele calcula todos os pontos da aresta que une esses pontos e para cada ponto calculado, a *flag out_valid* é levantada. Ao se fecharem todos os pontos da aresta a *flag out_finish* é também levantada.

3.2. Módulos Implementados

Essa seção tem por objetivo mostrar todos os módulos implementados para a construção do QuadDetector. Todos esses módulos são de autoria do autor e sua concepção baseia-se nos conceitos descritos no capítulo sobre a detecção de quadrados. A ordem de apresentação dos mesmos será conforme os passos já discutidos anteriormente para realização da detecção de quadrados, a saber os módulos `BordeTracing, VertexReduciont e PolylineAproximate`.

`BorderTracing`

O módulo `BorderTracing` foi totalmente implementado em VHDL. Seu papel é rastrear um contorno na imagem, passada como parâmetro.

A Figura 39 ilustra o módulo.

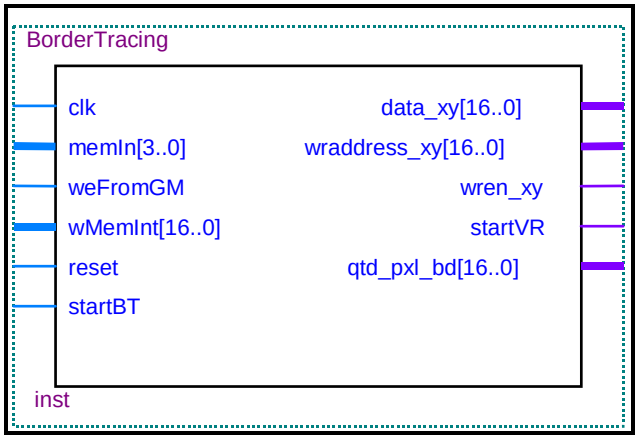


Figura 39: Módulo `Border Tracing`

A Tabela 9 mostra os sinais de entrada/saída do módulo especificado.

Tabela 9: Sinais de entrada/saída do módulo `BorderTracing`

Nome	Tamanho	Tipo	Descrição
clk	1	Entrada	Relógio de sincronismo do sistema
memIn	4	Entrada	Sinal que indica a cor do pixel
weFromGM	1	Entrada	Sinal que indica *pixel* válido
wMemInt	17	Entrada	Coordenadas do *pixel* lido
reset	1	Entrada	Reset do sistema
startBT	1	Entrada	Sinal que inicia o processamento
data_xy	17	Saída	Coordenadas do pixel do contorno detectado
wraddress_xy	17	Saída	Endereço de escrita na memória de contorno
wren_xy	1	Saída	Sinal que habilita escrita na memória de contorno
startVR	1	Saída	Sinal que indica fim do Border Tracing e inicializa a redução de vértices
qtd_pxl_bd	17	Saída	Quantidade de *pixels* encontrados no contorno

O funcionamento desse módulo baseia-se na descrição da detecção de contorno apresentada no capítulo anterior. A implementação de cada passo foi feita através da utilização de uma FSM (*Finite State Machine*), ou máquina de estados finita. Foram necessários vinte e seis estados, divididos em três blocos principais. O primeiro bloco é composto pelos estados de inicialização de variáveis e fim de processamento. O segundo bloco é composto pelos estados que procuram um *pixel* preto na imagem. O terceiro e último bloco é responsável por verificar os vizinhos do *pixel* encontrado, e garantir a conectividade entre o pixel analisado e seu vizinho preto, caso seja encontrado.

A Tabela 10 mostra os estados utilizados nesse módulo e o respectivo bloco de estados do qual fazem parte.

Tabela 10: Tabela de estados do módulo `BorderTracing` e respectivos blocos

Estados	Blocos
resetState	Inicialização / Fim
endBorderTracingProcess	
lookingForBlackPixel_fetch	Procura *pixel* preto
lookingForBlackPixel_fetch2	
lookingForBlackPixel_fetch3	
lookingForBlackPixel	
blackPixelNotFound	
validBlackPixel	
verifyPrevious_fetch	Verifica vizinhos
verifyPrevious_fetch2	
verifyPrevious_fetch3	
verifyPrevious	
verifyNext_fetch	
verifyNext_fetch2	
verifyNext	
lookingFirstNeighbor	
lookingFirstNeighbor_fetch	
lookingFirstNeighbor_fetch2	
lookingFirstNeighbor_fetch3	
lookingFirstNeighbor_final	
lookingForBlackNeighbor	
lookingForBlackNeighbor_fetch	

lookingForBlackNeighbor_fetch2	
lookingForBlackNeighbor_fetch3	
lookingForBlackNeighbor_final	
lookingForBlackNeighbor_final2	

A Figura 40 mostra uma abstração da máquina de estados implementada nesse módulo.

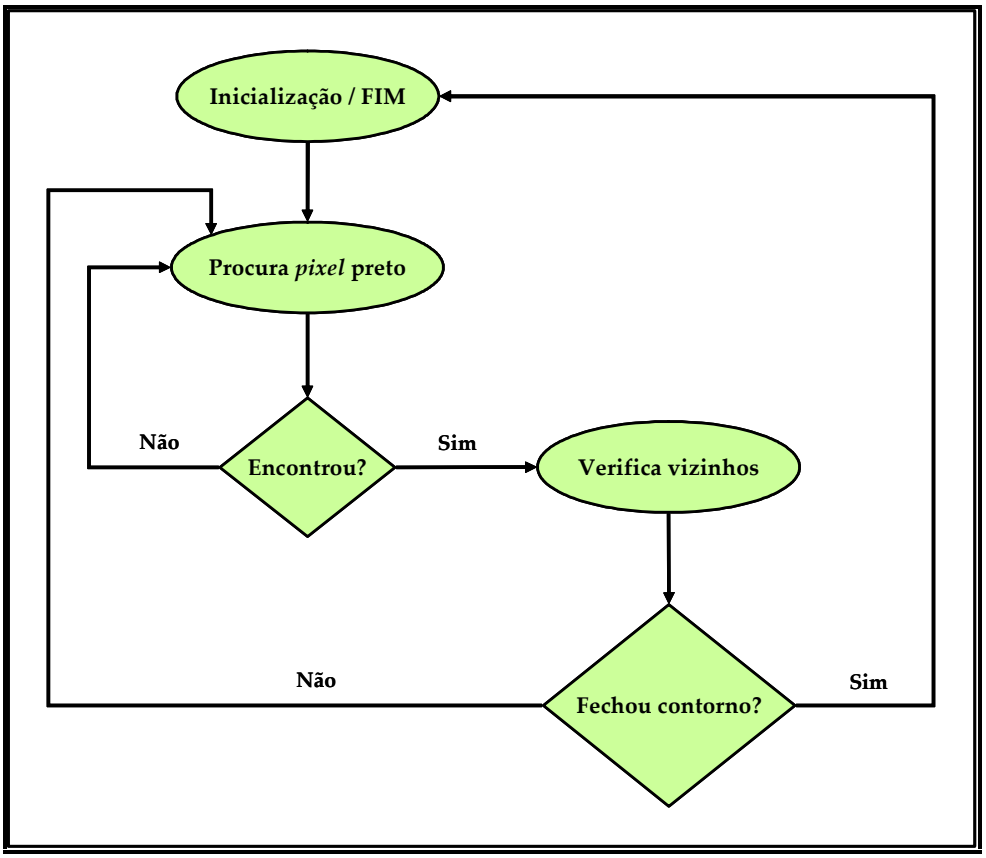


Figura 40: Máquina de estados do `Border Tracing`

Quando um contorno é detectado, o mesmo é salvo na memória de contorno mostrada na seção 3.1 desse capítulo e o processamento do `BorderTracing` levanta a *flag* startVR para indicar que a redução de vértices pode então ser iniciada.

`VertexReduction`

Assim como o `BorderTracing`, o módulo `VertexReduction` foi totalmente implementado em VHDL. Sua função é aplicar o algoritmo de redução de vértices ao contorno armazenado na memória de contorno. Como mostrado no capítulo 3 que trata a detecção de quadrados, faz-se necessário adotar um valor ε relativo a tolerância que se deseja dar entre os *pixels* do contorno. Essa implementação usa um valor de tolerância variável de acordo com o tamanho do contorno (número de *pixels*). O valor de ε será obtido através de uma operação de *shift right* de cinco bits (corresponde a uma divisão por trina e dois) no valor que indica a quantidade de pontos do contorno. A adoção desse parâmetro se baseia em um ajuste de 2% de tolerância.

A Figura 41 ilustra o módulo.

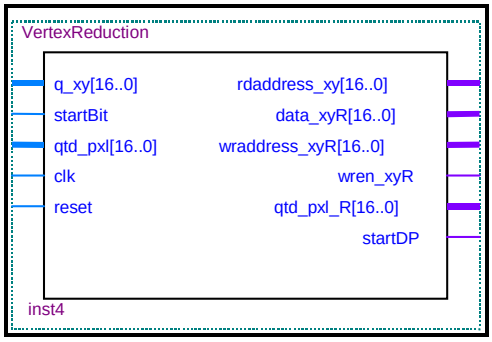


Figura 41: Módulo `VerteReduction`

A Tabela 11 mostra os sinais de entrada/saída do módulo especificado.

Tabela 11: Sinais de entrada/saída do módulo `VertexReduction`

Nome	Tamanho	Tipo	Descrição
q_xy	17	Entrada	Conteúdo lido da memória de contorno
startBit	1	Entrada	Sinal que inicia o processamento
qtd_pxl	17	Entrada	Sinal que indica a quantidade de *pixels* encontrados no contorno, usado para cálculo da tolerância
clk	1	Entrada	Relógio de sincronismo do distema
reset	1	Entrada	Reset de sistema

rdaddress_xy	17	Saída	Endereço de leitura na memória de contorno
data_xyR	17	Saída	Dado armazenado na memória de contorno reduzido
wraddress_xyR	17	Saída	Endereço de escrita na memória de contorno reduzido
wren_xyR	1	Saída	Sinal que habilita escrita na memória de contorno reduzido
qtd_pxl_R	17	Saída	Sinal que indica a nova quantidade de *pixels* do contorno já reduzido
startDP	1	Saída	Sinal que indica fim da redução de vértices e inicializa a aproximação poligonal

O funcionamento desse módulo baseia-se na descrição da redução de vértices apresentada no capítulo anterior. A implementação de cada passo foi também feita através da utilização de uma máquina de estados finita. Foram necessários nove estados, divididos em três blocos principais. O primeiro bloco é composto pelos estados de inicialização de variáveis e fim de processamento. O segundo bloco é composto pelos estados que buscam os pontos na memória de contorno. O terceiro e último bloco é responsável por computar as distâncias do *pixel* analisado para os próximos, até que uma dessas distâncias seja maior que a tolerância. Nesse momento, o *pixel* cuja distância tenha satisfeito a condição de tolerância passa a ser o *pixel* analisado e suas distâncias para os próximos serão computadas até que a condição seja satisfeita novamente. Percebe-se aqui que, dessa forma, todo o contorno é percorrido e os pontos que não satisfizerem a condição de tolerância em relação à distância são simplesmente descartados.

A Tabela 12 mostra os estados utilizados nesse módulo e o respectivo bloco de estados do qual fazem parte.

Tabela 12: Tabela de estados do módulo `VertexReduction` e respectivos blocos

Estados	Blocos
startContour	Inicialização / Fim
vertexReduction_fetch	Carrega o ponto analisado
vertexReduction_fetch2	
vertexReduction_fetch3	
vertexReduction	
pointsDistanceSum	Verifica distância para o próximo
pointsDistanceMod2	
pointsDistance	
checkTolerance	

A Figura 42 mostra uma abstração da máquina de estados implementada nesse módulo.

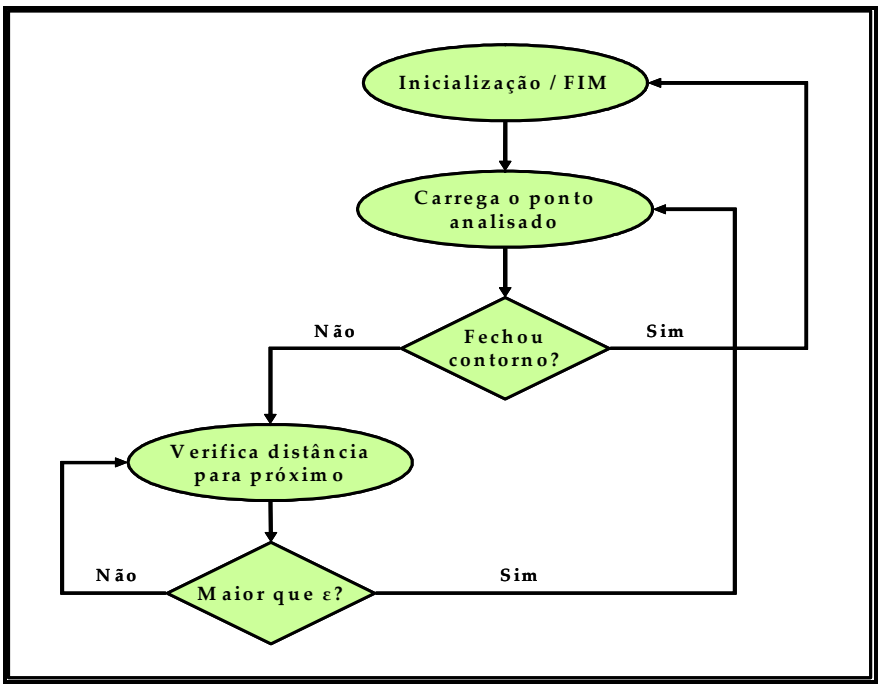


Figura 42: Máquina de estados do `VertexReducion`

Ao fim da redução de vértices, a memória de contorno reduzido implementada a partir do módulo `memContour`, conterá apenas os pontos relativos ao contorno já reduzido. A *flag* `startDP` é levantada para indicar que a aproximação poligonal pode ser iniciada.

`PolylineAproximate`

Assim como os dois últimos módulos citados, o módulo `PolylineAproximate` foi totalmente implementado em VHDL. Sua função é aplicar o algoritmo de aproximação poligonal. Dentro desse módulo também foi implementada a classificação poligonal, para identificar o contorno como sendo um quadrado ou não. A justificativa para essa junção de dois algoritmos em um mesmo módulo se baseia na capacidade de processamento do sistema. Como o resultado da aproximação poligonal é composto por um número pequeno de pontos (vértices do polígono), não se faz necessária à escrita dos mesmos em uma outra memória externa ao módulo para iniciar a classificação. A vantagem dessa abordagem é que ela evita desperdício de tempo com acessos à memória. O resultado da aproximação é salvo em um vetor dentro do próprio módulo e a classificação é feita com base nos pontos desse vetor. Como citado na seção 4 do capítulo anterior, um primeiro teste é verificar o tamanho desse vetor para saber se apenas quatro pontos foram detectados. Caso isso aconteça, os próximos passos da classificação são então executados.

Para o caso da aproximação poligonal, uma outra tolerância ψ precisa ser introduzida, como mostrado na seção 4 do capítulo anterior. O valor da mesma é obtido através de uma operação de *shift rigth* de dois *bits* (divisão por quatro) no sinal que indica a quantidade de *pixels* do contorno reduzido. Esse valor corresponde a 25% do número de pontos do contorno reduzido, o que para um quadrado representaria a quantidade de pontos de uma de suas quatro arestas. A idéia se baseia no fato de que cada aresta do quadrado contém cerca de 25% dos pontos de todo o contorno.

A Figura 43 ilustra o módulo.

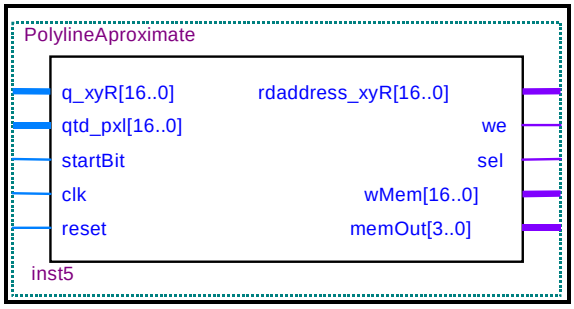


Figura 43: Módulo `PolylineAproximate`

A Tabela 13 mostra os sinais de entrada/saída do módulo especificado.

Tabela 13: Sinais de entrada/saída do módulo `PolylineAproximate`

Nome	Tamanho	Tipo	Descrição
q_xyR	17	Entrada	Conteúdo lido da memória de contorno reduzido
qtd_pxl	17	Entrada	Sinal que indica a quantidade de *pixels* encontrados no contorno já reduzido, usado para cálculo da tolerância
startBit	1	Entrada	Sinal que indica a quantidade de *pixels* encontrados no contorno, usado para cálculo da tolerância
clk	1	Entrada	Relógio de sincronismo do sistema
reset	1	Entrada	Reset do sistema
rdaddress_xyR	17	Saída	Endereço de leitura na memória de contorno reduzido
we	1	Saída	Sinal que habilita escrita na memória externa usada para plotar pontos no monitor VGA
sel	1	Saída	Sinal usado para plotar pontos no monitor VGA. Usado para selecionar o ponto que será plotado, virtual ou real
wMem	4	Saída	Cor do *pixel* salvo na memória VGA.
memOut	17	Saída	Coordenadas do *pixel* para que seja plotado na tela

O funcionamento desse módulo baseia-se na descrição da aproximação poligonal e da classificação de polígonos, descritas no capítulo anterior. A implementação de cada passo também foi feita através da utilização de uma máquina de

estados finita. Foram necessários vinte e seis estados, divididos em seis blocos principais. O primeiro bloco é composto pelos estados de inicialização de variáveis e fim de processamento. O segundo bloco é composto pelos estados que controlam os ponteiros e a pilha do sistema. O terceiro bloco é responsável por montar os segmentos e buscar os pontos cujas distâncias ao segmento serão computadas. O quarto bloco é responsável por calcular as distâncias do segmento aos pontos. O quinto bloco verifica a convexidade do polígono depois de aproximado (é o algoritmo de classificação do polígono) e classifica o mesmo como sendo um quadrado ou não. O sexto e último bloco de estados dessa FSM é responsável por plotar na tela o quadrado identificado a partir da utilização da interface VGA disponível no ARCam.

Como o algoritmo da aproximação poligonal funciona de forma iterativa, para sua implementação fez-se necessário o uso de pilhas e ponteiros que armazenam o estado do sistema, de modo que segmentos e pontos distintos possam ser processados em momentos diferentes. A forma de iteração desse algoritmo foi detalhada na seção 3 do capítulo anterior e corresponde aos passos de montar segmentos de reta entre os pontos do contorno de forma que computadas as distâncias desses segmentos para os outros pontos do contorno, pontos pertencentes às arestas do polígono sejam descartados e os vértices do mesmo sejam capturados.

A Tabela 14 mostra os estados utilizados nesse módulo e o respectivo bloco de estados do qual fazem parte.

Tabela 14: Tabela de estados do módulo `PolylineAproximate` e respectivos blocos

Estados	Blocos
init	Inicialização / Fim
approximateStart	Checa ponteiros
approximateComplete	
approximate	
segmentTake	Monta segmento
readPoint_fetch1	
readPoint_fetch2	
readPointer	

calculateDistancesWait	Calcula distâncias
calculate	
calculateDistances	
checkTolerance	
poligonFlush	Verifica convexidade
isQuadStart	
rdPointFetch	
rdPointFetch1	
takePoints	
calculateAreas	
isQuadContinue	
makeVectors	
calculateNorms2	
calculateNormsProd	
calculateAngles	
isQuadEnd	
notQuad	
drawEdges	Desenha na tela

A Figura 44 mostra uma abstração da máquina de estados implementada nesse módulo.

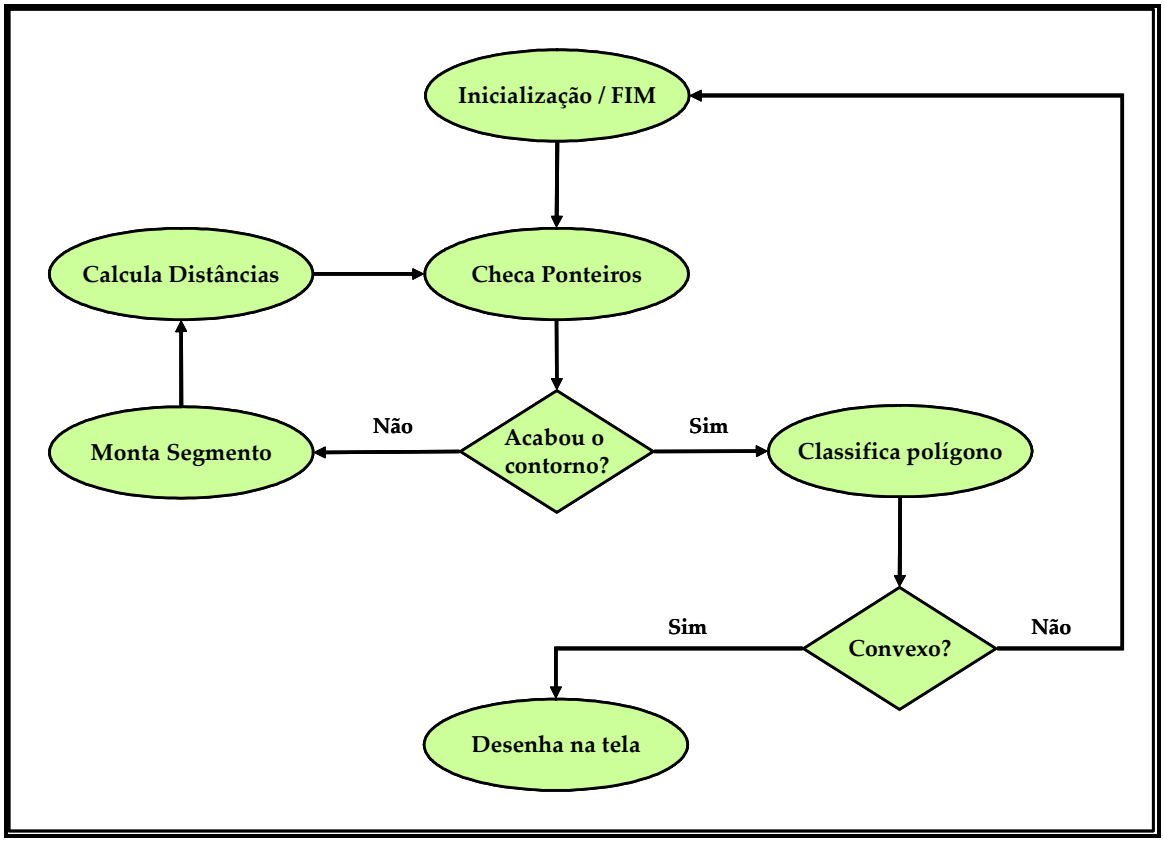


Figura 44: Máquina de estados do `PolylineAproximate`

Como adendo dessa seção, alguns trechos de código são colocados na seqüência a fim de mostrar como foram efetuadas algumas operações matemáticas como produto escalar, produto vetorial, cálculos de norma de vetores, além dos cálculos de área dos triângulos e dos ângulos formados pelas arestas dos quadrados.

De posse dos quatro pontos p1, p2, p3 e p4 que resultaram da aproximação poligonal para a construção um quadrado (vértices), a Figura 45 mostra os cálculos das áreas dos quatro triângulos internos ao quadrado introduzidos na seção 4 do capítulo 3.

```
t1Area <= SHR(((p1(18 downto 9)*p3(8 downto 0))-(p1(8 downto 0)*p3(18 downto 9))+
               (p1(8 downto 0)*p2(18 downto 9))-(p1(18 downto 9)*p2(8 downto 0))+
               (p3(18 downto 9)*p2(8 downto 0))-(p3(8 downto 0)*p2(18 downto 9))),"1");

t2Area <= SHR(((p2(18 downto 9)*p4(8 downto 0))-(p2(8 downto 0)*p4(18 downto 9))+
               (p2(8 downto 0)*p3(18 downto 9))-(p2(18 downto 9)*p3(8 downto 0))+
               (p4(18 downto 9)*p3(8 downto 0))-(p4(8 downto 0)*p3(18 downto 9))),"1");

t3Area <= SHR(((p3(18 downto 9)*p1(8 downto 0))-(p3(8 downto 0)*p1(18 downto 9))+
               (p3(8 downto 0)*p4(18 downto 9))-(p3(18 downto 9)*p4(8 downto 0))+
               (p1(18 downto 9)*p4(8 downto 0))-(p1(8 downto 0)*p4(18 downto 9))),"1");

t4Area <= SHR(((p4(18 downto 9)*p2(8 downto 0))-(p4(8 downto 0)*p2(18 downto 9))+
               (p4(8 downto 0)*p1(18 downto 9))-(p4(18 downto 9)*p1(8 downto 0))+
               (p2(18 downto 9)*p1(8 downto 0))-(p2(8 downto 0)*p1(18 downto 9))),"1");
```

Figura 45: Código para cálculos de área de triângulos

A Figura 46 mostra a formação dos vetores v1, v2, v3 e v4 a partir dos quatro pontos supracitados, além do cálculo das normas (módulos) desses vetores e o produto escalar deles, dois a dois, a fim de checarem os ângulos formados nos vértices.

```
WHEN makeVectors =>
    v1(18 downto 9) <= p2(18 downto 9) -  p1(18 downto 9);
    v1(8 downto 0)  <= p2(8 downto 0)  -  p1(8 downto 0);
    v2(18 downto 9) <= p3(18 downto 9) -  p2(18 downto 9);
    v2(8 downto 0)  <= p3(8 downto 0)  -  p2(8 downto 0);
    v3(18 downto 9) <= p4(18 downto 9) -  p3(18 downto 9);
    v3(8 downto 0)  <= p4(8 downto 0)  -  p3(8 downto 0);
    v4(18 downto 9) <= p1(18 downto 9) -  p4(18 downto 9);
    v4(8 downto 0)  <= p1(8 downto 0)  -  p4(8 downto 0);
    state <= calculateNorms2;

WHEN calculateNorms2 =>
    v1Norm2 <= (v1(18 downto 9))*(v1(18 downto 9)) + (v1(8 downto 0))*(v1(8 downto 0));
    v2Norm2 <= (v2(18 downto 9))*(v2(18 downto 9)) + (v2(8 downto 0))*(v2(8 downto 0));
    v3Norm2 <= (v3(18 downto 9))*(v3(18 downto 9)) + (v3(8 downto 0))*(v3(8 downto 0));
    v4Norm2 <= (v4(18 downto 9))*(v4(18 downto 9)) + (v4(8 downto 0))*(v4(8 downto 0));
    state <= calculateNormsProd;

WHEN calculateNormsProd =>
    multNorm12(27 downto 8) <= v1Norm * v2Norm;
    multNorm23(27 downto 8) <= v2Norm * v3Norm;
    multNorm34(27 downto 8) <= v3Norm * v4Norm;
    multNorm41(27 downto 8) <= v4Norm * v1Norm;

    prod12(35 downto 16) <= (v1(18 downto 9)*v2(18 downto 9)) + (v1(8 downto 0)*v2(8 downto 0));
    prod23(35 downto 16) <= (v2(18 downto 9)*v3(18 downto 9)) + (v2(8 downto 0)*v3(8 downto 0));
    prod34(35 downto 16) <= (v3(18 downto 9)*v4(18 downto 9)) + (v3(8 downto 0)*v4(8 downto 0));
    prod41(35 downto 16) <= (v4(18 downto 9)*v1(18 downto 9)) + (v4(8 downto 0)*v1(8 downto 0));
```

Figura 46: Código para operações com vetores

A Figura 47 mostra o cálculo dos cossenos formados pelos ângulos internos do quadrado. Essa verificação é o último passo para a detecção de quadrados.

```
WHEN calculateAngles =>
    cos1 <= conv_signed(abs(conv_integer(prod12) / conv_integer(multNorm12)),9);
    cos2 <= conv_signed(abs(conv_integer(prod23) / conv_integer(multNorm23)),9);
    cos3 <= conv_signed(abs(conv_integer(prod34) / conv_integer(multNorm34)),9);
    cos4 <= conv_signed(abs(conv_integer(prod41) / conv_integer(multNorm41)),9);
```

Figura 47: Código para cálculo dos ângulos internos de um quadrado

Finalizando a execução do módulo `PolylineAproximate`, chega-se ao fim do processamento necessário para identificar um quadrado em uma imagem, seja ela simulada ou capturada de um ambiente real a partir de um sensor de imagem, como acontece nesse trabalho. Nesse ponto, o quadrado identificado é então desenhado na tela e sobreposto ao contorno que o originou. Na seção de resultados, a seguir, serão mostrados resultados colhidos durante a utilização do QuadDetector, validando o mesmo.

4. Resultados

Esta seção tem por objetivo apresentar os resultados obtidos com a implementação do QuadDetector.

A Figura 48 mostra uma cena capturada pela câmera, contendo um marcador do tipo suportada pelo ARToolKit. Na figura, o usuário está posicionando o marcador no campo de visão da câmera, para que seja identificado pelo sistema. O destaque preto na imagem visualizada no monitor representa a borda que está sendo rastreada dessa cena, ainda sem característica de forma, representando simplesmente os contornos.

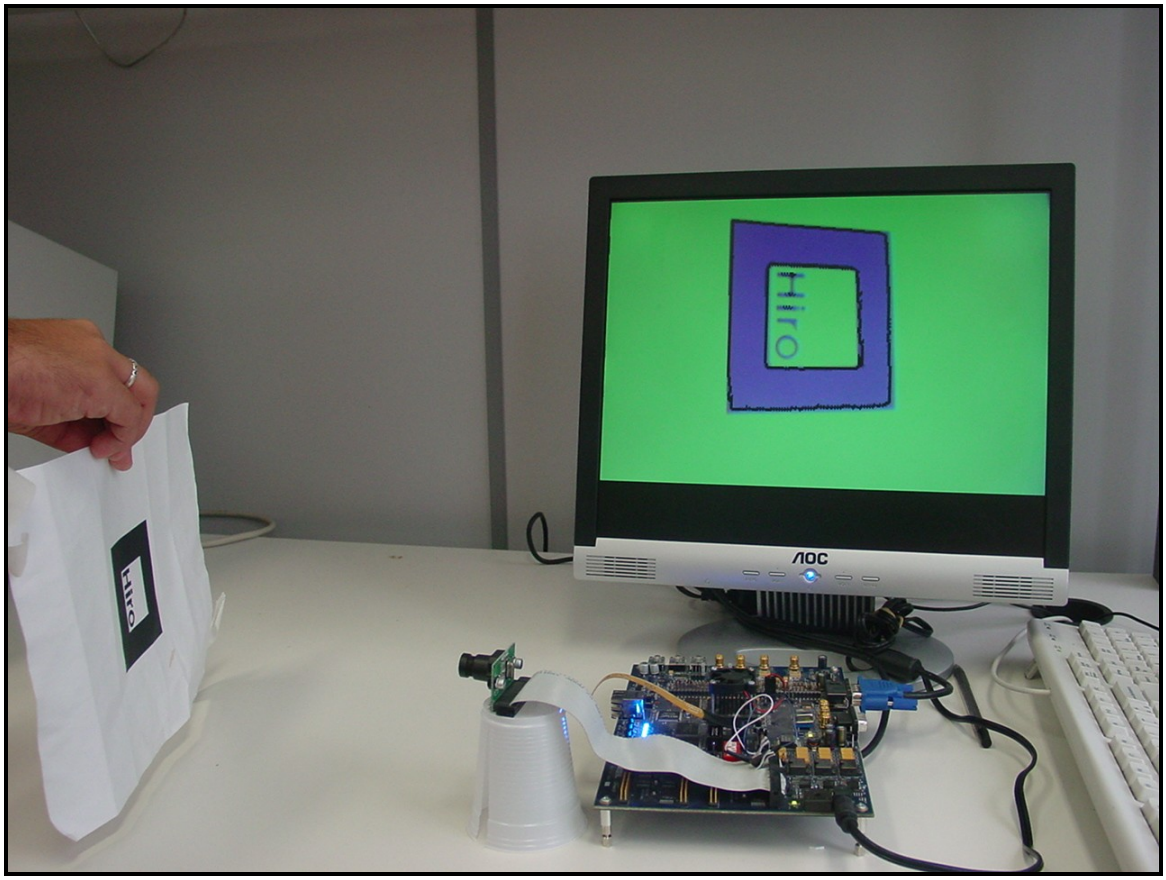


Figura 48: Quadrado sendo posicionado à frente da câmera para detecção

A Figura 49 mostra destacada não só a borda rastreada (em preto), como também o quadrado detectado (destacado em branco). Vale ressaltar aqui que as bordas internas que compõem o marcador e que são quadradas, não estão detectadas devido ao fato da detecção ainda ser feita apenas no primeiro contorno carregado na memória. Do ponto de vista de aplicações para RA, a parte de detecção do marcador na cena já está resolvida, faltando apenas reconhecer o padrão do marcador. A limitação quanto a detecção de apenas um quadrado na cena se dá por conta do espaço de memória disponível para implementação. O uso de uma memória externa já está sendo analisado, o que permitirá não só o processamento de mais de uma borda por *frame* como também o uso de uma imagem de resolução mais alta (640x480). A seção de trabalhos futuros detalha melhor esse ponto.

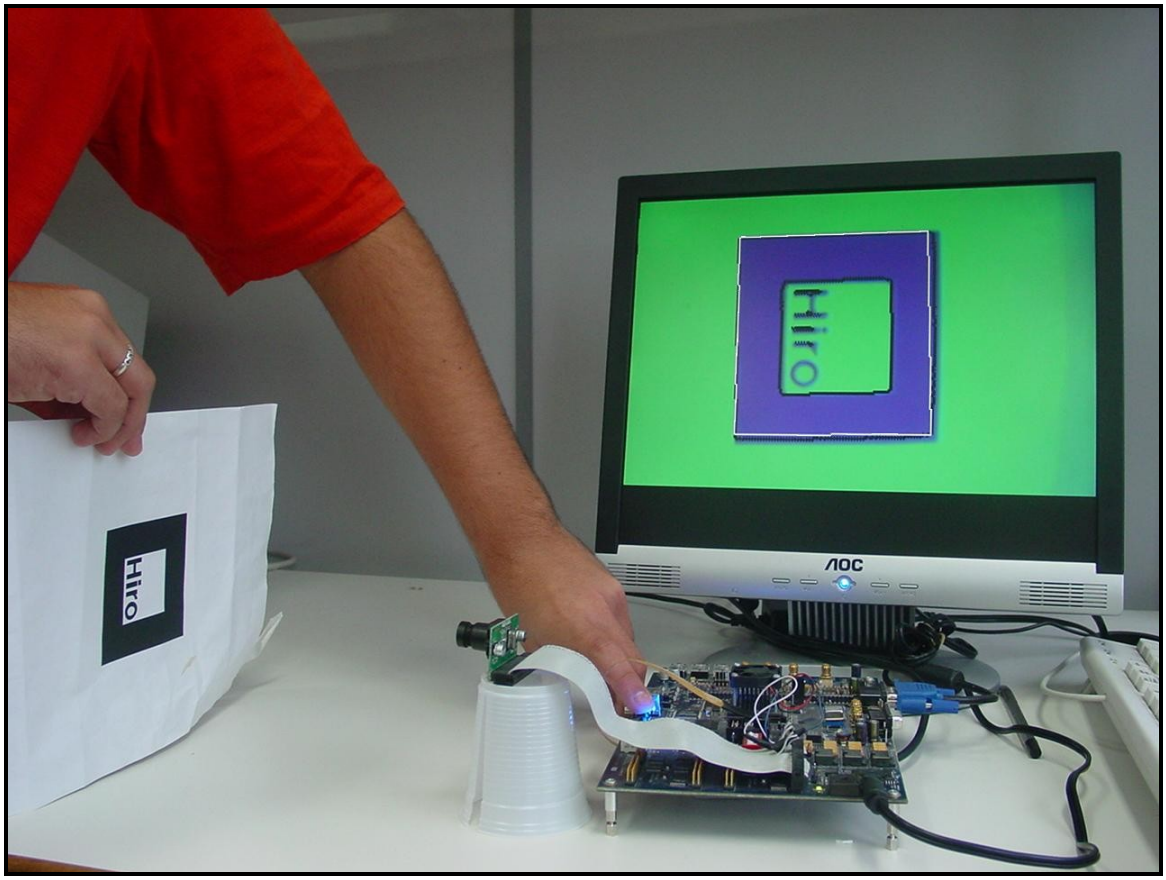


Figura 49: Resultado da detecção de quadrados

A Figura 50 mostra que a imagem detectada e renderizada virtualmente pode ser congelada, enquanto a câmera continua a capturar novas imagens e as mesmas podem ser mostradas na tela. Essa abordagem pode se mostrar interessante em aplicações onde detectado o posicionamento de um objeto na cena tenta-se posicionar a câmera novamente conforme o posicionamento do objeto na imagem.

Figura 50: Quadrado detectado e congelado na cena

O *clock* de operação do sistema se divide em dois. Apenas para o hardware da câmera o clock utilizado foi de 27Mhz, especificado em seu *datasheet* como *clock* máximo que garante o funcionamento da mesma à uma taxa de 30 quadros/segundo. No resto do ARCam, incluindo aqui o QuadDetector, o *clock* utilizado foi de 25 MHz, garantindo o funcionamento ideal do módulo. O uso de um clock maior poderia gerar problemas ao sistema, que tenderia a processar imagens antes mesmo da câmera disponibilizá-las. A título de explicação, o uso de *clocks* variados foi possível através da utilização de um PLL (*Phase-locked-loop*) integrado à placa de prototipação utilizada. O PLL permite a multiplicação do clock original da placa (100MHz) por uma constante estabelecida pelo projetista, podendo a mesma ser maior (multiplica) ou menor (divide) que um.

Foram realizados testes comparativos entre a execução do mesmo algoritmo para detecção de quadrados implementado em *hardware* através do QuadDetector e sua implementação em C/C++. O PC usado para comparação é da família Pentium da Intel e roda a uma freqüência de 2.0GHz. Os resultados da comparação estão apresentados na Tabela 15, abaixo. Para a comparação, os resultados obtidos a partir da execução normal de cada algoritmo foram normalizados a uma freqüência comum de 100MHz.

Tabela 15: Comparação entre os resultados das implementações *hardware*/*software*

Implementação	**Freqüência de operação**	**Ciclos necessários**	**Tempo de processamento à mesma freqüência**
Hardware	25 MHz	47.023	470 us
Software	2.0 Ghz	1.299.604	12.996 us

É significativo o ganho que se obteve com a implementação em *hardware* do algoritmo de detecção de quadrados. São resultados como esse que comprovam a característica principal dos sistemas embarcados (velocidade de processamento), como também motivam pesquisadores a continuar desenvolvendo esses sistemas para áreas como a RA.

Vale salientar que o comparativo foi feito com o *hardware* do modo como foi implementado, sem serem realizadas otimizações de qualquer natureza como eliminação de estados e diminuição de acessos à memória, como proposto nos trabalhos futuros. É certo que um mínimo de otimização ainda é possível de ser feito para o QuadDetector, tornando-o ainda mais eficiente, e aumentando seus benefícios em relação à sua implementação em *software.*

5. 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 LEs) disponível no FPGA, 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 sistema embarcado, e serem analisadas ainda na fase de estudo de viabilidade, para saber se o elemento disponível é compatível com o tamanho do projeto almejado.

As principais dificuldades encontradas na implementação do módulo QuadDetector aconteceram na fase de definição das funcionalidades do projeto, na qual foram definidas com detalhes as etapas de processamento que deveriam ser implementadas, além do padrão de tamanho do vetor usado para a representação de um *pixel*. Além disso, aconteceram alguns erros devido ao uso inadequado de algumas bibliotecas fornecidas, assim como problemas com versões mais recentes do Quartus II que foram solucionados ao longo do desenvolvimento desse subprojeto.

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*. Durante todo o desenvolvimento do projeto foi inviável o uso de otimizações de compilação, uma vez que apenas o ARCam com a funcionalidade de capturar a imagem e projetá-la na tela já demora cerca de 5 minutos para ser sintetizado; o ARCam com a funcionalidade do QuadDetector leva cerca de 16 minutos. À medida que os módulos iam sendo desenvolvidos e agrupados, o tempo de compilação de todo o projeto crescia vertiginosamente.

Problemas relativos ao uso de memória interna também foram encontrados e devido a isso, optou-se por utilizar uma resolução pequena de 320x240, embora a câmera utilizada seja capaz de fornecer imagens no tamanho VGA de 640x480.

Problemas com a ferramenta, quanto à síntese, foram detectados. Durante o processo de compilação ela não acusou erros quanto ao tamanho do *hardware* implementado e ao baixá-lo no FPGA, o mesmo acendeu o *Led* vermelho indicando erro no *hardware* baixado. Por muitas vezes a simples re-compilação do projeto resolveu o problema. Um outro problema relativo à ferramenta foi detectado no que tange o processo de compilação. Uma opção de otimização da compilação através do *Smart Compilation*, que deveria não sintetizar os módulos sintetizados já em um momento anterior que não sofreram alteração se mostrou instável. Devido à essa instabilidade, muitas vezes ela chegou a não re-sintetizar módulos alterados, o que acarretou na falta de confiabilidade e, portanto, não utilização da mesma.

CONCLUSÃO

O presente documento apresentou e detalhou todo a implementação do QuadDetector, um *hardware* capaz de detectar quadrados em uma imagem real, em tempo real, capturada a partir de uma câmera associada a um FPGA.

Técnicas e algoritmos para esse tipo de processamento foram estudados à nível de *software* e a viabilidade de implementação dos mesmos em *hardware* foi avaliada. Constatada a viabilidade, os algoritmos foram implementados usando VHDL.

A contribuição desse Trabalho de Graduação para o projeto ARCam foi bastante relevante, uma vez que o mesmo almeja embarcar o fluxo de atividades do ARToollKit, cuja principal característica é a utilização de marcadores na cena para sobrepor aos mesmos objetos virtuais. O fato dos marcadores seguirem o padrão quadrado de bordas pretas leva a concluir que o primeiro passo a ser dado é a detecção de um quadrado.

Além da contribuição já destacada na identificação de marcadores do ARToollKit, o QuadDetector pode ser expandido para qualquer tipo de aplicação que busque reconhecer um padrão quadrado na tela, ou retângulos, como uma placa de carro, peças industriais ou outra aplicação com tal característica.

É importante ressaltar que atualmente ainda existem poucos trabalhos na literatura que visam o desenvolvimento de sistemas embarcados para aplicações de RA. Espera-se também que esse trabalho fique como legado e referência para pesquisas futuras envolvendo essa linha de pesquisa.

1. Trabalhos Futuros

Os trabalhos futuros relativos ao QuadDetector são tornar seu processamento contínuo, ou seja, a cada *frame* armazenado o processamento é executado, permitindo que o mesmo não seja feito apenas ao pressionar um botão de *start*, como acontece atualmente.

Uma outra perspectiva é a de unir os algoritmos de detecção de contorno e redução de vértices em um único módulo, a fim de evitar muitos acessos à memória. Além disso, possíveis otimizações relativas à eliminação de estados são totalmente viáveis.

A possível aplicação de uma memória externa já está em fase de estudo para que sejam liberados LEs do FPGA para processamento e não para armazenamento, como ocorre na atual versão. Dessa forma, espera-se também poder trabalhar com uma imagem de resolução maior, 640x480, já que a câmera pode fornecer imagens com essa resolução.

REFERÊNCIAS BIBLIOGRÁFICAS

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

[2] ARToollKit. Disponível em: Human Interface Technology Lab site. URL: http://www.hitl.washington.edu/artoolkit. Visitado em Março de 2007.

[3] ARTag. Disponível em: ARTag site. URL: http://www.cv.iit.nrc.ca/research/ar/artag/. Visitado em Março de 2007.

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

[5] 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.

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

[7] 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.

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

[9] Open Source Computer Vision Library. Disponível em: Intel Corporation site. URL: http://www.intel.com/technology/computing/opencv. Visitado março de 2007.

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

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

[18] Field-programmable gate array. Disponível em: Wikipedia site. URL: http://en.wikipedia.org/wiki/FPGA. Visitado em Março de 2007.

[19] Xilinx: The Programmable Logic Company. Disponível em: Xilinx site. URL: http://www.xilinx.com. Visitado em Março de 2007.

[20] Eclipse: an open development platform. Disponível em: Eclipse site. URL: http://www.eclipse.org. Visitado em Março de 2007.

[21] TEIXEIRA, J M. Hardwire: um módulo em hardware para a visualização em wireframe de objetos tridimensionais Trabalhos de conclusão de curso. Ano de conclusão: 2006.

[22] Sistemas embarcados. Disponível em: Wikipedia site. URL: http://pt.wikipedia.org/wiki/Sistemas_embarcados. Visitado em Março de 2007.

[23] A Table-Top Augmented Reality Application. Disponível: Multimedia Concepts and Applications site. URL: http://mm-werkstatt.informatik.uni-augsburg.de/project_details.php?id=1. Consultado em 26 dez. 2005.

[24] ARToolKitPlus. Disponível em: HandHeld Augmeted Reality site. URL: http://studierstube.icg.tu-graz.ac.at/handheld_ar/artoolkitplus.php. Visitado em Março de 2007.

[25] Countour fitting. Disponível em: Cantag/algorithms, Univesity of Cambridge site. URL: http://www.cl.cam.ac.uk/research/dtg/research/wiki/Cantag/algorithms. Visitado em Março de 2007.

[26] Polyline Simplification. Disponível em: SoftSurfer site. URL: http://geometryalgorithms.com/Archive/algorithm_0205/#Douglas-Peucker. Visitado em Março de 2007.

[27] Border Tracing. Disponível em: Digital Image Processing site. URL: http://www.icaen.uiowa.edu/~dip/LECTURE/Segmentation2.html#location. Visitado em Março de 2007.

[28] Convex Polygon. Disponível em: Wikipedia site. URL: http://en.wikipedia.org/wiki/Convex_polygon. Visitado em Março de 2007.

[29] Convex Polygon. Disponível em: The Three-Coins Algorithm for Convex Hulls of polygons site. URL: http://cgm.cs.mcgill.ca/~beezer/cs507/main.html. Visitado em Março de 2007.

DATAS E ASSINATURAS

Recife, 29 de Março de 2007

Orientadora

_____

Judith Kelner

Co-orientadora

_____

Veronica Teichrieb

Aluno

_____

Guilherme Dias da Silva