UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

VGLGUI: Uma interface gráfica de programação visual para a biblioteca VisionGL

Dissertação de Mestrado

Roberto Wagner Santos Maciel

São Cristóvão – Sergipe

2022

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Roberto Wagner Santos Maciel

VGLGUI: Uma interface gráfica de programação visual para a biblioteca VisionGL

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Sergipe como requisito parcial para a obtenção do título de mestre em Ciência da Computação.

Orientador(a): Daniel Oliveira Dantas

São Cristóvão – Sergipe

2022

Este trabalho é dedicado aos profissionais de saúde
que lutam diariamente pela vida.

Agradecimentos

Agradeço a Deus pela vida e força para continuar lutando, a minha esposa, Alexsandra Carmo, pelo amor, compreensão, parceria, incentivo a nunca desistir e as minhas filhas Ana Beatriz e Maria Eduarda, que chegaram juntas como uma benção e transformaram minha visão de mundo. Agradeço aos meus pais pela educação, carinho e aos meus familiares pelo apoio de sempre. Amo vocês!

Ao meu orientador, Daniel Dantas, que desde o primeiro contato foi solícito e conduziu toda a pesquisa com muita dedicação em fazer sempre o melhor, meu muito obrigado por todos os ensinamentos! Com o Sr. aprendi coisas muito além do aspecto acadêmico, lições que me transformaram durante toda esta jornada e levo para a vida pessoal e profissional.

Ao meu parceiro de pesquisa, João Gabriel, que sempre esteve presente nesta etapa final da pesquisa, meu muito obrigado. Espero que você tenha sucesso nos seus empreendimentos. Pode sempre contar comigo.

Aos meus colegas de trabalho, que compartilham comigo a luta do dia a dia por uma tecnologia da informação de qualidade e inclusiva, a competência de vocês impulsiona-me a sempre buscar crescer.

Aos meus alunos de graduação e pós-graduação, que me inspiram a estudar todos os dias por acreditar que a edução transforma realidades, meus agradecimentos.

Este passo é a realização de um sonho de 25 anos atrás de ter uma formatura pela Universidade Federal de Sergipe, que bom que veio em um mestrado. Tudo ao seu tempo!!!

Consciência transforma a realidade!
Rogério Chér

Resumo

Imagens médicas são usadas em clínicas para apoiar o diagnóstico e o tratamento de doenças. O desenvolvimento de algoritmos de visão computacional eficazes para o processamento de imagens é uma tarefa desafiadora que requer uma quantidade significativa de tempo investido na fase de prototipagem. Existem sistemas de programação visual que buscam facilitar a prototipagem. Outros sistemas que permitem o processamento paralelo tentam possibilitar o tratamento de conjuntos de dados de imagens muito grandes que demandam um alto tempo de execução. Os sistemas de *workflow*, por outro lado, tornaram-se ferramentas populares, pois permitem desenvolver algoritmos como uma coleção de blocos de função, que podem ser vinculados graficamente a *pipelines* de entrada e saída. Isso ajuda a reduzir a curva de aprendizado para programadores iniciantes. Por fim, existem sistemas que facilitam a programação e aumentam a produtividade por meio da geração automática de código. VisionGL é uma biblioteca de código aberto que facilita a programação por meio da geração automática de código *wrapper* C++. O código *wrapper* é responsável por chamar funções de processamento paralelo de imagens ou *shaders* em CPUs usando OpenCL e em GPUs usando OpenCL, GLSL e CUDA. VGLGUI é uma interface gráfica de usuário para processamento de imagem que permitirá a programação visual de *workflow* para processamento paralelo de imagens, por meio de funções VisionGL para geração automática de código *wrapper* e otimização de transferências de imagem entre RAM e GPU. Esta pesquisa tem por objetivo apresentar a descrição da arquitetura da VGLGUI em múltiplas visualizações, utilizando o padrão arquitetural ISO/IEC/IEEE 42010:2011, o 4 + 1 View Model of Software Architecture e a Unified Modeling Language (UML). Também tem como objetivo a descrição e criação do interpretador de *workflow* da VGLGUI, e demonstração dos resultados de dois *pipelines* de processamento de imagem em duas plataformas diferentes: com a linguagem Python usando a biblioteca OpenCV executando na CPU, e; com o interpretador da VGLGUI executando na GPU.

Palavras-chave: Arquitetura de Software. Interpretador. *Workflow*. Processamento de imagem

Abstract

Medical imaging is used in clinics to support the diagnosis and treatment of disease. Developing effective computer vision algorithms for image processing is a challenging task that requires a significant amount of time invested in the prototyping phase. There are visual programming systems that seek to facilitate prototyping. Other systems that allow parallel processing try to make it possible to handle very large image datasets that demand a high execution time. *Workflow* systems, on the other hand, have become popular tools because they allow you to develop algorithms as a collection of function blocks that can be graphically linked to input and output *pipelines*. This helps to reduce the learning curve for beginning programmers. Finally, there are systems that make programming easier and increase productivity through automatic code generation. VisionGL is an open source library that facilitates programming through automatic generation of C++ *wrapper* code. The *wrapper* code is responsible for calling parallel image processing functions or *shaders* on CPUs using OpenCL and on GPUs using OpenCL, GLSL and CUDA. VGLGUI is a graphical user interface for image processing that will allow visual *workflow* programming for parallel image processing, through VisionGL functions for automatic *wrapper* code generation and optimization of image transfers between RAM and GPU. This research aims to present the architecture description of VGLGUI in multiple views, using the ISO / IEC / IEEE 42010: 2011 architectural standard, the 4 + 1 View Model of Software Architecture and the Unified Modeling Language (UML). It also aims to describe and create the VGLGUI *workflow* interpreter, and demonstrate the results of two image processing *pipelines* on two different platforms: with the Python language using the OpenCV library running on the CPU, and; with the VGLGUI interpreter running on the GPU.

Keywords: Software Architecture. Interpreter. *Workflow*. Image processing

Lista de ilustrações

Lista de tabelas

Lista de códigos

Lista de abreviaturas e siglas

2D	Duas dimensões
3D	Três dimensões
ADL	Architecture description language
ACM	Association for Computing Machinery
API	Application Programming Interface
ASP	Active Server Pages
AST	Abstract syntax tree
CPU	Central process unit
CWM	Common warehouse metamodel
CUDA	Compute Unified Device Architecture
SGBD	Sistema gerenciador de banco de dados
DICOM	Digital Imaging and Communications
DSL	Domain-specific languages
DST	Medical diagnosis support tool
EC2	Amazon Elastic Compute Cloud
FCFS	First-come first-served
GRS	General Running Service
GPU	Graphics processing unit
GRS	General Running Service
GUI	Graphical user interface
HTGS	Hybrid Task Graph Scheduler
IDL	Interactive Data Language
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers

ISO	International Organization for Standardization
JIST	Java Image Science Toolkit
MIC	Model-Integrated Computing
MOF	Meta Object Facility
NFS	Network File System
NIST	National Institute of Standards and Technology
NPP	NVIDIA Performance Primitives
OpenCV	Open Source Computer Vision Library
PACS	Picture archiving and communication system
RAM	Random access memory
SOA	Service-oriented architecture
UML	Unified Modeling Language
WebGL	Web Graphics Library
XSS	Cross-site scripting

Sumário

1

Introdução

Uma imagem médica é um componente importante nas instituições de saúde atuais, usada principalmente como ferramenta de apoio ao diagnóstico médico (*diagnostic support tool*, DST) para melhorar a qualidade de cuidados médicos (MARWAN; KARTIT; OUAHMANE, 2017). A imagem médica é uma das ferramentas de diagnóstico médico mais básicas e comuns. Para olhos treinados, ela pode descrever com precisão os órgãos internos de um ser humano e indicar a presença de patologias. O primeiro passo em qualquer interpretação de imagens médicas é a segmentação da imagem. O ser humano pode segmentar e analisar a imagem no nível cognitivo. Em contraste, os computadores precisam de algoritmos específicos para esta tarefa (GAL; STOICU-TIVADAR, 2011).

Imagens médicas são usadas em clínicas para apoiar o diagnóstico e tratamento de doenças (QUEIROS et al., 2018). Há um aumento recente nas técnicas de imagem não invasivas para triagem de pacientes quanto a anormalidades, incluindo câncer (PARK et al., 2016).

O reconhecimento de processos patológicos é um dos problemas mais importantes no processamento da imagem médica: melhoria da qualidade da imagem, recuperação da imagem, reconhecimento de elementos. O padrão Digital Imaging and Communications in Medicine (DICOM), permite armazenar diversos dados médicos em um formato versátil, bem como conectar vários dispositivos compatíveis (computadores pessoais, impressoras, scanners) entre si, formando assim uma rede *picture archiving and communication system* (PACS). A rede PACS garante todo o trabalho necessário com imagens médicas digitais, aumenta a velocidade e a qualidade dos diagnósticos. O equipamento médico digital é considerado o fornecedor de todas as imagens para o sistema PACS. O mais importante é que o PACS oferece uma oportunidade de aumento da velocidade de acesso a imagens médicas, fornece o trabalho simultâneo de especialistas de diferentes hospitais e aumenta a capacidade de atendimento dos dispositivos médicos. A qualidade do diagnóstico médico é melhorada por meio da introdução de tecnologias digitais especiais (DORONICHEVA; SAVIN, 2018).

O tamanho das coleções de imagens aumentou e atingiu petabytes de dados. Esses volumes não podem ser processados em um computador dentro de um tempo razoável. Portanto, as tarefas contemporâneas de processamento de imagens requerem paralelismo (SOZYKIN; EPANCHINTSEV, 2015). Uma solução avançada de visualização e processamento de imagens médicas permite a leitura de grandes quantidades de imagens, capacidade de avaliar imagens em várias modalidades, ferramentas de *workflow* relevantes, vários algoritmos de diagnóstico auxiliados por computador e uma ampla gama de aplicações clínicas (UKIS et al., 2013).

É comum ter instituições de saúde fornecendo assistência remota, como diagnóstico remoto (telediagnóstico) e opinião de um médico especializado (teleconsulta) a outras instituições médicas que não possuem recursos humanos especializados ou mesmo para fins educacionais. A evolução da tecnologia cria oportunidades para melhorar serviços de telemedicina em um paradigma "a qualquer momento e em qualquer lugar" (MONTEIRO; SILVA; COSTA, 2012).

A comunidade científica tem testemunhado o surgimento de métodos de aprendizagem profunda (deep learning) aplicados no processamento de linguagem natural, genoma, visão computacional e informática em saúde. Da mesma forma, a análise de imagens médicas tem sido revolucionada por técnicas de aprendizagem profunda para tarefas como segmentação, registro, classificação e detecção, com desempenho que às vezes supera o de especialistas humanos. Contudo, apesar da popularidade e do desempenho atuais, nem sempre é possível acessar as informações dos algoritmos de aprendizagem projetados para tarefas médicas. Isso torna difícil comparar soluções, integrá-las aos trabalhos existentes ou usá-los para experimentação própria ou para desenvolvimento. Pesquisadores, no entanto, ainda enfrentam o desafio de criar um ambiente de desenvolvimento e obter familiaridade com APIs para adaptar modelos pré-treinados em seus próprios dados (MILLETARI et al., 2019).

O desenvolvimento de ferramentas de processamento de imagem tornou-se popular, com aplicações biológicas e médicas. Ferramentas computacionais úteis foram desenvolvidas para análise biomédica de imagens, incluindo ImageJ, CellProfiler e Icy. ImageJ é uma ferramenta desenvolvida em Java, comum para análise de imagens em biologia. CellProfiler é uma ferramenta especializada para segmentação 2D sua análise quantitativa e sua extensão, é capaz de manipular e analisar imagens multidimensionais. Icy inclui capacidade de processamento de imagem multidimensional, e otimiza seu processamento por múltiplos núcleos com a implementação em OpenCL. A maioria das ferramentas de processamento de imagem existentes são sistemas independentes que não estão equipados com uma plataforma de comunicação para permitir colaboração entre usuários ou o compartilhamento de imagens, software, ou recursos de CPU e GPU. O desenvolvimento de sistemas de processamento de imagem próprios tornou-se difícil desde que hardware caro, como um *cluster* de GPUs ou um supercomputador, e seu software são necessários para análise de imagens 3D em larga escala (MORITA et al., 2013).

No campo da saúde, processamento de imagens médicas por meio da computação em nuvem é uma tecnologia promissora. De fato, permite que os profissionais de saúde processem

e analisem imagens usando recursos remotos da nuvem. Além disso, fornece software como serviço, que realiza importantes reduções de custos associados à exploração do *data center* local e contratação de equipe de TI (MARWAN; KARTIT; OUAHMANE, 2017). A melhoria da qualidade da imagem, a recuperação das imagens e o reconhecimento dos elementos, são áreas de desenvolvimento no processamento de imagens médicas na nuvem, fundamentais para o reconhecimento de processos patológicos, por meio do fornecimento de informações visuais sobre a estrutura interna e funções do corpo humano (DORONICHEVA; SAVIN, 2018).

As tecnologias avançadas de imagem atraíram profissionais de saúde a adotarem essas ferramentas para melhorar a qualidade dos cuidados médicos. O processamento de imagens médicas na nuvem é uma solução adequada para atender às demandas de assistência médica e implementações foram recentemente propostas. Destaco: proposta de um método seguro para efetuar o processamento da imagem sobre uma imagem criptografada; estrutura para terceirização da visualização de dados médicos na nuvem; plataforma para processamento de imagens médicas com base na infraestrutura de Eucalyptus e software ImageJ; proposta de uma solução baseada em nuvem para realizar com segurança a operação de projeção de raios de volume; estrutura que permite processamento de imagens médicas sobre registros digitais criptografados; solução para gerenciar informações médicas baseadas na estrutura do Hadoop e API para desenvolver aplicativos; *framework* para análise e processamento de imagens médicas na nuvem; *framework* que fornece processamento de imagens médicas como serviço, que visa promover a colaboração entre profissionais de saúde; *framework* baseado em *service-oriented architecture* (SOA) para processar imagens médicas (MARWAN; KARTIT; OUAHMANE, 2017).

Apesar das múltiplas vantagens desta tecnologia, ao migrar para a nuvem surgem inúmeros desafios. Segurança e privacidade são os principais fatores que dificultam a ampla aceitação do processamento de imagens médicas na nuvem. Em geral, esses desafios são divididos em três categorias principais: técnicos, gerenciais e jurídicos (MARWAN; KARTIT; OUAHMANE, 2017).

Os desafios gerenciais são interoperabilidade e portabilidade. Os provedores de nuvem oferecem aos clientes acesso onipresente a serviços remotos por meio da Internet. Os clientes têm acesso a esses recursos entregues via aplicativos de serviço da web e *application programming interface* (API) (MARWAN; KARTIT; OUAHMANE, 2017). Para resolver problemas de interoperabilidade em serviços de processamento de imagens médicas baseados em nuvem, o protocolo DICOM é proposto para proteger informações médicas relacionadas aos pacientes. Este padrão sugerido garante a interoperabilidade entre organizações de saúde e centros de imagem. Além disso, o DICOM permite a integração de dispositivos de imagem de diferentes fabricantes em um arquivo de imagens e sistema de comunicação PACS (MARWAN; KARTIT; OUAHMANE, 2017). DICOM provê formato de arquivo, armazenamento, confirmação de armazenamento, busca e recuperação, lista de tarefas, procedimento realizado por equipamento e impressão (DICOM, 2008).

Os desafios jurídicos envolvem conformidade e questões legais. As plataformas em nuvem estão espalhadas por vários servidores localizados em diferentes países. Com isto, problemas legais e de conformidade devem ser abordados antes da mudança para a nuvem. As leis atuais não cobrem esses novos desafios. Além disso, os atos de regulamentação da saúde não são os mesmos em todos os países (MARWAN; KARTIT; OUAHMANE, 2017).

Vários são os desafios na criação de sistemas de processamento de imagens médicas, como o tamanho das imagens e a necessidade de integração com outras soluções no ambiente clínico e hospitalar (UKIS et al., 2013). Pesquisadores e programadores têm dificuldade em construir soluções de segmentação, filtragem e visualização de imagens médicas, devido à curva de aprendizado, complexidade de instalação e uso dessas ferramentas (MORITA et al., 2013).

VisionGL é uma biblioteca de código aberto que facilita a programação por meio da geração automática de código *wrapper* C++. O código *wrapper* é responsável por chamar funções de processamento paralelo de imagens ou *shaders* em CPUs usando OpenCL e em GPUs usando OpenCL, GLSL e CUDA. VGLGUI é uma interface gráfica de usuário para processamento de imagem que permitirá a programação visual de *workflow* para processamento paralelo de imagens, por meio de funções da biblioteca VisionGL (MACIEL; SOARES; DANTAS, 2021).

Esta pesquisa tem por objetivo apresentar a descrição da arquitetura da VGLGUI em múltiplas visões, utilizando o padrão arquitetural ISO/IEC/IEEE 42010:2011, 4+1 View Model of Software Architecture e a Unified Modeling Language (UML). Também tem como objetivo a descrição e criação do interpretador de *workflow* da VGLGUI, e demonstração dos resultados de dois *pipelines* de processamento de imagem em execução na CPU e na GPU. Os *workflows* foram portados para Python com OpenCV executado na CPU e VisionGL com C++ executado na CPU e GPU. Os experimentos foram executados 10 ou 1000 vezes cada, respeitando a performance viável para apurar a média de tempo de execução. Os experimentos foram realizados em cinco plataformas, Python na CPU, Interpreter na CPU, Interpreter na GPU, VisionGL na CPU e VisionGL na GPU.

A descrição da arquitetura de um sistema, auxilia a compreensão da essência do sistema e das principais propriedades relacionadas ao seu comportamento, composição e evolução, que por sua vez afeta preocupações como a viabilidade, utilidade e manutenção. As descrições de arquitetura são usadas pelas partes que criam, utilizam e gerenciam sistemas para melhorar a comunicação e cooperação, permitindo que trabalhem de maneira integrada e coerente. Linguagens de descrição de arquitetura são usadas para codificar as práticas comuns de arquitetura (ISO, 2011).

2

Referencial Teórico

2.1 Descrição da arquitetura - ISO/IEC/IEEE 42010:2011

A complexidade dos sistemas de informação é a cada dia maior. Isso levou a novas oportunidades, mas também a maiores desafios para as organizações que criam e utilizam sistemas. Os princípios e procedimentos de arquitetura são cada vez mais aplicados para ajudar a gerenciar a complexidade enfrentada por partes interessadas dos sistemas. A descrição da arquitetura auxilia a compreensão da essência do sistema e das principais propriedades relacionadas ao seu comportamento, composição e evolução. A evolução afeta preocupações como a viabilidade, utilidade e manutenção do sistema (ISO, 2011).

A Norma Internacional ISO/IEC/IEEE 42010:2011 trata da criação, análise e manutenção de arquiteturas de sistemas através do uso de descrições de arquitetura (ISO, 2011). As descrições de arquitetura são usadas pelas partes que criam, utilizam e gerenciam sistemas de informação para melhorar comunicação e cooperação, permitindo que trabalhem de maneira integrada e coerente. Estruturas e linguagens de descrição de arquitetura estão sendo criadas como padrões que codificam as convenções comuns de arquitetura e a descrição de arquitetura em diferentes comunidades e domínios de aplicação. A descrição da arquitetura de um sistema faz uso de uma linguagem de descrição arquitetural para organizar, expressar pontos de vista e visões arquiteturais que tratam das preocupações de seus *stakeholders* (ISO, 2011).

As partes interessadas, *stakeholders* de um sistema, têm preocupações com relação ao sistema considerado. Uma preocupação pode ser mantida por uma ou mais partes interessadas. Preocupações surgem ao longo do ciclo de vida das necessidades e requisitos do sistema, das escolhas de design e da implementação e operação. Uma preocupação pode se manifestar de várias formas, como em relação a uma ou mais partes interessadas, necessidades, objetivos, expectativas, responsabilidades, requisitos, restrições de design, suposições, dependências, atributos de qualidade, decisões de arquitetura, riscos ou outros problemas relacionados ao

sistema (ISO, 2011).

Podem ser feitas perguntas para entender e documentar o contexto de um software. As respostas são capturadas modelando a descrição da arquitetura como um conjunto de cenários, alternativas e decisões. É possível ajudar os *stakeholders* a explorarem informalmente o contexto de design de um sistema proposto e seu ambiente e, no processo, determinar as preocupações que levam às decisões arquiteturais (HARPER; ZHENG, 2015).

Uma visão é governada por seu ponto de vista. O ponto de vista estabelece as convenções para construir, interpretar e analisar a visão para tratar as preocupações enquadradas por esse ponto de vista. As convenções do ponto de vista podem incluir linguagens, notações, tipos de modelo, regras de projeto e métodos de modelagem, técnicas de análise e outras operações em vista. Uma visualização de arquitetura é composta por um ou mais modelos de arquitetura. Um modelo de arquitetura usa modelagens adequadas às preocupações a serem abordadas (ISO, 2011).

2.1.1 Uso das descrições da arquitetura

As descrições de arquitetura têm muitos usos por uma variedade de partes interessadas ao longo do ciclo de vida do sistema. Descrições da arquitetura são usadas como base para as atividades de design e desenvolvimento de sistemas; como base para analisar e avaliar implementações alternativas de uma arquitetura; como documentação de desenvolvimento e manutenção; para documentar aspectos essenciais de um sistema. As decisões de arquitetura, suas razões e implicações servem como entrada para ferramentas automatizadas para simulação, geração e análise de sistemas; para especificar um grupo de sistemas que compartilham recursos comuns; para comunicação entre as partes envolvidas no desenvolvimento, produção, implantação, operação e manutenção de um sistema e; planejamento de transição de uma arquitetura herdada para uma nova arquitetura (ISO, 2011).

2.1.2 Identificação das partes interessadas e preocupações

Uma descrição da arquitetura deve identificar as partes interessadas que tenham preocupações consideradas fundamentais para a arquitetura do sistema. Os seguintes interessados devem ser considerados e, quando aplicável, identificados na descrição da arquitetura: usuários; operadores; compradores; proprietários; fornecedores; desenvolvedores; construtores e; mantenedores do sistema. Uma descrição da arquitetura deve identificar as preocupações consideradas fundamentais para a arquitetura do sistema em questão (ISO, 2011).

Uma descrição da arquitetura deve identificar as preocupações consideradas fundamentais para a arquitetura do sistema de interesse. As seguintes preocupações devem ser consideradas e, quando aplicável, identificadas na descrição da arquitetura: os objetivos do sistema; a adequação da arquitetura para alcançar os objetivos do sistema; a viabilidade de construir e implantar o

sistema; os riscos e impactos potenciais do sistema para as partes interessadas ao longo do seu ciclo de vida e; capacidade de manutenção e evolução do sistema. Uma descrição da arquitetura deve associar cada preocupação às partes interessadas identificadas (ISO, 2011).

2.1.3 Visões e decisões da arquitetura

Uma descrição da arquitetura deve incluir exatamente uma visão da arquitetura para cada ponto de vista da arquitetura usado. Cada visão de arquitetura deve aderir às convenções de seu ponto de vista de arquitetura governante. Cada visualização da arquitetura deve incluir identificação e informações suplementares, conforme especificado pela organização ou projeto; identificação de seu ponto de vista governante; modelos de arquitetura que abordam todas as preocupações formuladas por seu ponto de vista de governo e cobrem as sistema inteiro desse ponto de vista e; registro de quaisquer problemas conhecidos dentro de uma visão em relação ao seu ponto de vista governante (ISO, 2011).

Uma descrição da arquitetura deve registrar as decisões de arquitetura consideradas essenciais para a arquitetura do sistema de interesse. A estratégia do projeto deve ser aplicada para estabelecer critérios para selecionar as principais decisões a serem tomadas. Os critérios a considerar são decisões sobre requisitos arquiteturalmente significativos; decisões que necessitam de um grande investimento de esforço ou tempo para realizar, implementar ou aplicar; decisões que afetam as principais partes interessadas ou várias partes interessadas; decisões que exigem raciocínio intrincado ou não óbvio; decisões altamente sensíveis a mudanças; decisões que podem custar caro mudar e decisões que formam uma base para o planejamento e gerenciamento do projeto (ISO, 2011).

No Capítulo 4 será descrita a arquitetura da VGLGUI. Serão utilizados elementos da ISO/IEC/IEEE 42010:2011: identificação de partes interessadas e suas preocupações; decisões de arquitetura e suas consequências e; visões da arquitetura.

2.2 Unified Modeling Language (UML)

Uma *architecture description language* (ADL) é qualquer forma de expressão para uso em descrições de arquitetura. Uma ADL fornece um ou mais tipos de modelo como forma de enquadrar algumas preocupações para seus *stakeholders*. Uma ADL pode ser focada de maneira restrita, definindo um tipo de modelo único ou amplamente focada para fornecer vários tipos de modelo, opcionalmente organizados em pontos de vista. Muitas vezes, uma ADL é suportada por ferramentas automatizadas para auxiliar na criação, uso e análise de seus modelos (ISO, 2011).

A Unified Modeling Language (UML) é uma linguagem visual para especificar, construir e documentar os artefatos de sistemas. É uma linguagem de modelagem de propósito geral que pode ser usada com todos os principais métodos de objetos e componentes, e que pode ser aplicada a todos os domínios como saúde, finanças, telecomunicações, aeroespacial etc. Pode

ser usada em plataformas de implementação como J2EE, .NET etc (Object Management Group, 2011).

2.2.1 Arquitetura da linguagem UML

A especificação da UML é definida usando uma abordagem de metamodelagem, ou seja, um metamodelo é usado para especificar o modelo que compreende a UML, que adapta técnicas de especificação formal. Embora esta abordagem careça de um pouco do rigor de um método de especificação, a UML oferece as vantagens de ser mais intuitiva e pragmática para a maioria dos implementadores e praticantes. O metamodelo UML foi desenvolvido com princípios de design a seguir (Object Management Group, 2011):

- Modularidade — Este princípio de forte coesão e baixo acoplamento é aplicado para agrupar construções em pacotes e organizar recursos em metaclassses.
- Camadas — As camadas são aplicadas de duas maneiras ao metamodelo UML. Primeiro, a estrutura do pacote é em camadas para separar as construções centrais da metalinguagem das construções de nível superior que as utilizam. Em segundo lugar, um metamodelo de quatro camadas padrão arquitetural é aplicado de forma consistente para separar preocupações (especialmente em relação à instanciação) entre camadas de abstração.
- Particionamento — O particionamento é usado para organizar áreas conceituais dentro da mesma camada. A biblioteca de infraestrutura da UML tem particionamento de baixa granularidade, usado para fornecer a flexibilidade exigida pelos atuais e futuros padrões de metamodelagem. No caso do metamodelo UML, o particionamento é mais grosseiro para aumentar a coesão dentro dos pacotes e flexibilizar o acoplamento entre pacotes.
- Extensibilidade — A UML pode ser estendida através de um novo dialeto ou de uma nova linguagem. O novo dialeto pode ser definido usando perfis para personalizar o idioma para plataformas específicas (por exemplo, J2EE/EJB e .NET/COM+) e domínios (por exemplo, finanças, telecomunicações e aeroespacial). Uma nova linguagem relacionada à UML pode ser especificada reutilizando parte do pacote da biblioteca de infraestrutura da UML, aumentando com metaclasses e metarrelações apropriadas.
- Reutilização — Uma biblioteca de metamodelo flexível e refinada é fornecida para definir o metamodelo UML e é reutilizada para definir outros metamodelos arquiteturalmente relacionados, como o Meta Object Facility (MOF) e o Common Warehouse Metamodel (CWM).

2.2.2 Diagramas da UML

A UML oferece um conjunto de tipos de diagramas usados para documentar visualizações dos requisitos funcionais do usuário, estrutura do sistema e comportamento dinâmico dos

elementos de software (SELLAMI; HAOUES; BEN-ABDALLAH, 2013). O objetivo da UML é fornecer múltiplas visões do sistema. Cada diagrama da UML analisa o sistema, ou parte dele, sob uma determinada perspectiva (GUEDES, 2011).

Linguagens de modelagem orientada a objetos, como UML, têm sido usadas para descrever arquiteturas de software há décadas (BUCHMANN; DOTOR; WESTFECHTEL, 2014). A estrutura de um software pode ser descrita, por exemplo, usando diagramas de pacotes, diagramas de classes ou diagramas de componentes, e a interação entre os componentes do sistema pode ser descrita por diagramas de sequência (BUCHMANN; DOTOR; WESTFECHTEL, 2014).

O **diagrama de casos de uso** é o diagrama mais geral e informal da UML. Usado nas fases de levantamento e análise de requisitos, utiliza-se de linguagem simples e de fácil compreensão. O diagrama é usado para dar aos *stakeholders* uma ideia do sistema. É estabelecida a relação entre atores, usuários, outros sistemas, algum hardware e casos de uso, serviços do sistema e funcionalidades (GUEDES, 2011).

O **diagrama de classes** é o mais utilizado e um dos mais importantes da UML. Serve de apoio para a maioria demais diagramas. Define a estrutura de classes utilizadas pelo sistema, os atributos e métodos que cada classe tem. Estabelece como as classes se relacionam e como as classes trocam informações (GUEDES, 2011). Os diagramas de classes UML são usados por engenheiros de software para modelar conceitualmente a estrutura de um sistema em termos de classes, atributos e operações, e para expressar restrições que devem ser aplicadas a cada instância do sistema (CALI et al., 2012). Os diagramas de classes são amplamente usados na indústria de software para criar uma visão de baixo nível do desenvolvimento de sistemas (FAHRENBERG et al., 2014).

O **diagrama de objetos** está amplamente associado ao diagrama de classes. É um complemento do diagrama de classes. Fornece uma visão dos valores armazenados pelos objetos do diagrama de classes em um momento da execução do sistema (GUEDES, 2011).

O **diagrama de pacotes** é um diagrama estrutural que representa os subsistemas ou submódulos de uma solução. Pode ser usado independente ou associado a outros diagramas. Pode ser usado para ilustrar a arquitetura de uma linguagem de programação, ou para definir camadas de um software ou de um processo de desenvolvimento (GUEDES, 2011). O diagrama de pacote mostra a decomposição de um determinado sistema em unidades coesas que são fracamente acopladas umas às outras. Cada pacote pode conter as entidades atômicas finais, consistindo em classes e seus relacionamentos mútuos (TONELLA; POTRICH, 2005).

O **diagrama de sequência** é um diagrama comportamental que se preocupa com a ordem temporal em que as mensagens são trocadas entre os objetos envolvidos num processo. Baseia-se em um caso de uso de mesmo nome. Apoia-se no diagrama de classes para determinar os objetos envolvidos em um processo. Identifica o evento gerador do processo, o ator responsável por esse evento e determina como o processo será executado e concluído. Define como as mensagens

são trocadas entre os métodos (GUEDES, 2011). O diagrama de sequência requer a análise simultânea de restrições e regras de outros diagramas que dificultam seu aprendizado (ALHAZMI; THEVATHAYAN; HAMILTON, 2020). Os diagramas de sequência descrevem um conjunto de processos parcialmente ordenados em eventos `send` e `receive` (ALVIN; PETERSON; MUKHOPADHYAY, 2019).

O **diagrama de comunicação** é associado ao diagrama de sequência. Um diagrama complementa o outro. As informações mostradas no diagrama de comunicação com frequência são praticamente as mesmas apresentadas no diagrama de sequência. Porém, o de comunicação não se preocupa com a temporalidade do processo. Preocupa-se com a forma como os elementos do diagrama estão vinculados e quais mensagens trocam entre si durante processo (GUEDES, 2011).

O **diagrama de máquina de estados** demonstra o comportamento de um elemento por meio de um conjunto finito de transições de estado. Pode ser utilizado para expressar o comportamento de uma parte do sistema ou pode ser utilizado para expressar o protocolo de uso de parte de um sistema. Baseia-se em um diagrama de casos de uso. Pode ser utilizado para acompanhar os estados de uma instância de uma classe (GUEDES, 2011).

O **diagrama de atividade** preocupa-se em descrever os passos a serem percorridos para conclusão de uma atividades específica. A atividade pode ser representada por um método com certo grau de complexidade, um algoritmo, ou um processo completo. Concentra-se no controle de fluxo de uma atividade (GUEDES, 2011).

O **diagrama de visão geral de interação** é uma variação do diagrama de atividade que fornece uma visão geral dentro de um sistema ou processo de negócio (GUEDES, 2011).

O **diagrama de componentes** está amplamente associado à linguagem de programação que será utilizada para desenvolver o sistema modelado. Representa os componentes do sistema quando ele for ser implementado em termos de módulos de código-fonte, bibliotecas, formulários, arquivos de ajuda e módulos executáveis. Determina como tais componentes estarão estruturados e irão interagir para que o sistema funcione de maneira adequada (GUEDES, 2011). Embora os componentes de software compreendam partes relativamente grandes de sistemas, os aplicativos podem facilmente consistir em centenas de componentes fortemente interconectados (HOLY; BRADA, 2011). O diagrama de componentes é um modelo de abstração de alto nível com informações úteis para o entendimento da arquitetura de um sistema, pois descreve as unidades de execução, armazenamento de dados e mecanismos de interação (HAITZER; ZDUN, 2013).

O **diagrama de implantação** determina todo o aparato físico sobre o qual o sistema deverá ser executado. Necessidades de hardware do sistema, as características físicas como servidores, estações, topologias e protocolos de comunicação. Permite demonstrar como se dará a distribuição dos módulos do sistema, em situações em que estes forem ser executados em mais de um servidor (GUEDES, 2011). O diagrama de implantação UML é usado para modelar a

implantação estática de um sistema, mostrando uma configuração dos componentes constituintes de um portfólio de infraestrutura (INGALSBE, 2005).

O **diagrama de estrutura composta** descreve a estrutura interna de uma classe ou componente. Detalhando partes internas que o compõem, como estas se comunicam e colaboraram entre si. Descreve a colaboração em que um conjunto de instâncias cooperam entre si para realizar uma tarefa (GUEDES, 2011).

O **diagrama de tempo** descreve a mudança no estado ou condição de uma instância de uma classe ou papel durante um período. Tipicamente utilizado para demonstrar a mudança no estado de um objeto no tempo em resposta a eventos externos (GUEDES, 2011).

O **diagrama de perfil** possibilita a definição de novos elementos UML. Permite assim estender os diagramas existentes com a inclusão de estruturas customizadas para uma determinada necessidade (GUEDES, 2011).

No Capítulo 4, para a descrição da arquitetura da VGLGUI serão utilizados diagramas de caso de uso, de classes, de pacotes, de sequência, de componentes e de implantação.

2.3 4+1 View Model of Software Architecture

O uso de múltiplas visões tem sido um princípio importante na descrição da arquitetura desde os primeiros trabalhos em arquitetura de software (HILLIARD et al., 2012). A documentação de múltiplas visões ajuda os *stakeholders* a entenderem melhor a arquitetura do sistema, e permite gerenciar a complexidade e os riscos durante o seu desenvolvimento (RIBEIRO; RIBEIRO; SOARES, 2017; FRANCA; LIMA; SOARES, 2017). Modelar todas as preocupações dos *stakeholders* de um sistema em uma única visão é uma tarefa difícil. As equipes geralmente lidam apenas com visualizações parciais e incompletas de um sistema, que são mais fáceis de gerenciar (REINEKE; TRIPAKIS, 2014). O 4+1 View Model of Software Architecture usa múltiplas visões para abordar separadamente as preocupações das várias partes interessadas na arquitetura do softwasre e atender aos requisitos funcionais e não funcionais (KRUCHTEN, 1995). Um exemplo de arquitetura que usa este modelo é encontrado no estudo de Vidoni (VIDONI; VECCHIETTI, 2016).

A descrição de uma arquitetura de software usando o 4+1 View Model consiste em cinco visualizações principais. A visão lógica, Logical View, é o modelo de objetos do design, quando um método de design orientado a objetos é usado. A visualização do processo, Process View, captura os aspectos de simultaneidade e sincronização do design. A visão física, Physical View, descreve os mapeamentos do software no hardware e seu aspecto distribuído. A visão de desenvolvimento, Development View, descreve a organização estática do software em seu ambiente de desenvolvimento, e uma visão complementar de cenários, Scenarios, são uma abstração dos requisitos mais importantes (KRUCHTEN, 1995).

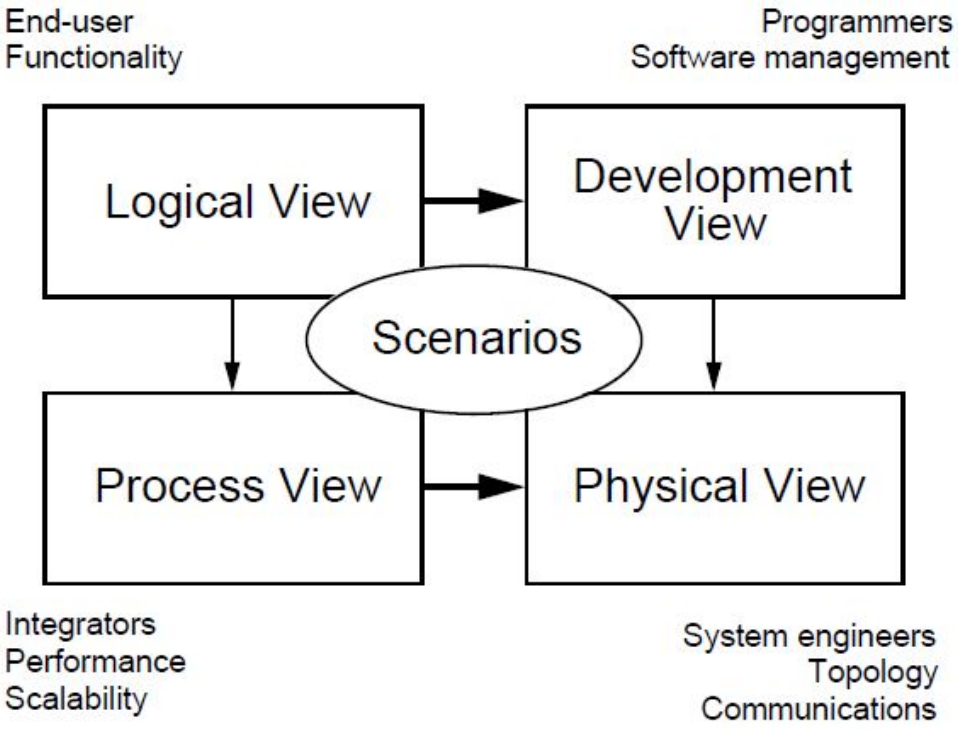


Figura 1 – 4+1 View Model of Software Architecture

A Figura 1 mostra as visões do 4+1 View Model of Software Architecture.

A **visão lógica** oferece suporte principalmente aos requisitos funcionais, o que o sistema deve fornecer em termos de serviços aos seus usuários. O sistema é decomposto em um conjunto de abstrações-chave, tomadas principalmente do domínio do problema, na forma de objetos ou classes de objetos. Eles exploram os princípios da abstração, encapsulamento e herança. Esta decomposição não é apenas para fins de análise funcional, mas também serve para identificar mecanismos comuns e elementos de design nas várias partes do sistema (KRUCHTEN, 1995). O ponto de vista lógico descreve a estrutura lógica do sistema em termos de uma rede de componentes interagindo independentemente da realização técnica, a fim de atingir o máximo de reutilização e atender aos requisitos não funcionais (EDER et al., 2017).

A visão lógica pode ser representada por meio de diagramas de classes. Um diagrama de classes mostra um conjunto de classes e seus relacionamentos lógicos: associação, uso, composição e herança. Conjuntos de classes relacionadas podem ser agrupados em categorias de classe. Se for importante definir o comportamento interno de um objeto, pode ser usado com o diagrama de máquina de estado. Como alternativa para uma abordagem orientada a objetos, um aplicativo que é muito orientado a dados pode ter sua visão lógica representada com um diagrama de entidades e relacionamentos (KRUCHTEN, 1995).

A **visão do processo** leva em consideração alguns requisitos não funcionais, como desempenho e disponibilidade. Aborda questões de concorrência e distribuição, integridade do sistema, tolerância a falhas e como as principais abstrações da visão lógica se encaixam na

arquitetura do processo, em qual segmento de controle é uma operação para um objeto realmente executado. A arquitetura do processo pode ser descrita em vários níveis de abstração, cada nível abordando diferentes preocupações. Um processo é um grupo de tarefas que formam uma unidade executável. Os processos representam o nível no qual a arquitetura do processo pode ser controlada taticamente, ou seja, iniciada, recuperada, reconfigurada e desligada. O software é dividido em um conjunto de tarefas independentes. Uma tarefa é uma *thread* de controle separada que pode ser agendada individualmente em um nó de processamento. A arquitetura do processo pode ser descrita em vários níveis de abstração, cada nível abordando diferentes questões. No nível mais alto, a arquitetura do processo pode ser vista como um conjunto de redes lógicas de execução independente de programas de comunicação denominados processos (KRUCHTEN, 1995).

A **visão de desenvolvimento** da arquitetura concentra-se na organização no ambiente de desenvolvimento de software. O software é empacotado em pequenos blocos, bibliotecas de programas ou subsistemas, que podem ser desenvolvidos por um ou um pequeno número de desenvolvedores. Os subsistemas são organizados em uma hierarquia de camadas, cada camada fornecendo uma interface estreita e bem definida para as camadas acima dela. A visão de desenvolvimento serve como base para a alocação de requisitos, para alocação de trabalho para equipe, ou mesmo para organização de equipe, para avaliação de custos e planejamento, para acompanhar o andamento do projeto, para raciocinar sobre reutilização de software, portabilidade e segurança. É a base para o estabelecimento de uma linha de produtos (KRUCHTEN, 1995).

A visão de desenvolvimento do sistema é representada por diagramas de módulo e subsistema, mostrando o relacionamentos de exportação e importação (KRUCHTEN, 1995).

A **visão física** leva em consideração principalmente os requisitos não funcionais do sistema, como disponibilidade, confiabilidade, tolerância a falhas, desempenho, taxa de transferência e escalabilidade. O software executa em uma rede de computadores ou nós de processamento. Os vários elementos identificados — redes, processos, tarefas e objetos — precisam ser mapeados nos vários nós. Várias diferentes configurações físicas são usadas: algumas para desenvolvimento e teste, outras para implantação do sistema para vários sites ou para clientes diferentes. O mapeamento do software para os nós, portanto, precisa ser altamente flexível e ter um impacto mínimo no próprio código-fonte (KRUCHTEN, 1995).

Os elementos nas quatro visualizações são mostrados para trabalhar juntos perfeitamente pelo uso de um conjunto de importantes **cenários**, instâncias de casos de uso mais gerais, para os quais descrevem-se os scripts correspondentes. Scripts são sequências de interações entre objetos e entre processos. Os cenários são uma abstração dos requisitos mais importantes. Seu design é expresso usando diagramas de cenário de objeto e diagramas de interação de objeto. Esta visão é redundante com as outras, mas serve a dois propósitos principais: como um condutor para descobrir os elementos arquiteturais durante o projeto de arquitetura e; como uma função de validação e ilustração após a conclusão do projeto de arquitetura, tanto no papel quanto no ponto

de partida para os testes de um protótipo arquitetural (KRUCHTEN, 1995).

No Capítulo 4, para a descrição da arquitetura da VGLGUI será utilizado o modelo de Visão 4+1. Na visão lógica serão usados os diagramas de pacotes e diagrama de classes. Na visão de processo será usado o diagrama de sequência. Na visão de desenvolvimento serão usados o diagrama de pacotes e diagrama de componentes. Na visão física será usado o diagrama de implantação.

3

Trabalhos relacionados

O desenvolvimento de algoritmos eficazes de visão computacional e processamento de imagens é uma tarefa desafiadora que requer uma quantidade significativa de tempo investido na fase de prototipagem. Os não programadores são mal servidos, pois não há uma interface de alto nível fácil de usar (WANG; HOGUE, 2020).

Existem sistemas de programação visual que buscam facilitar a prototipagem. Outros sistemas que permitem o processamento paralelo possibilitam o tratamento de conjuntos de dados de imagens muito grandes que demandam um alto tempo de execução. Os sistemas de *workflow*, por outro lado, tornaram-se ferramentas populares, pois permitem desenvolver algoritmos como uma coleção de blocos de função, que podem ser vinculados graficamente a *pipelines* de entrada e saída. Isso ajuda a reduzir a curva de aprendizado para programadores iniciantes. Por fim, existem sistemas que facilitam a programação e aumentam a produtividade por meio da geração automática de código.

3.1 Sistemas de programação visual

São exemplos de sistemas de programação visual Khoros-Cantata, Interactive Data Language (IDL), Processing, Pure Data e CVNodes.

3.1.1 Khoros-Cantata

Khoros é um ambiente de desenvolvimento de software usado para processamento de imagens. Cantata é uma linguagem de programação visual construída dentro do sistema Khoros (YOUNG; ARGIRO; WORLEY, 1995). Ao fornecer um ambiente visual para a resolução de problemas, o Cantata aumenta a produtividade de pesquisadores e desenvolvedores de aplicativos, independentemente de sua experiência em programação (YOUNG; ARGIRO;

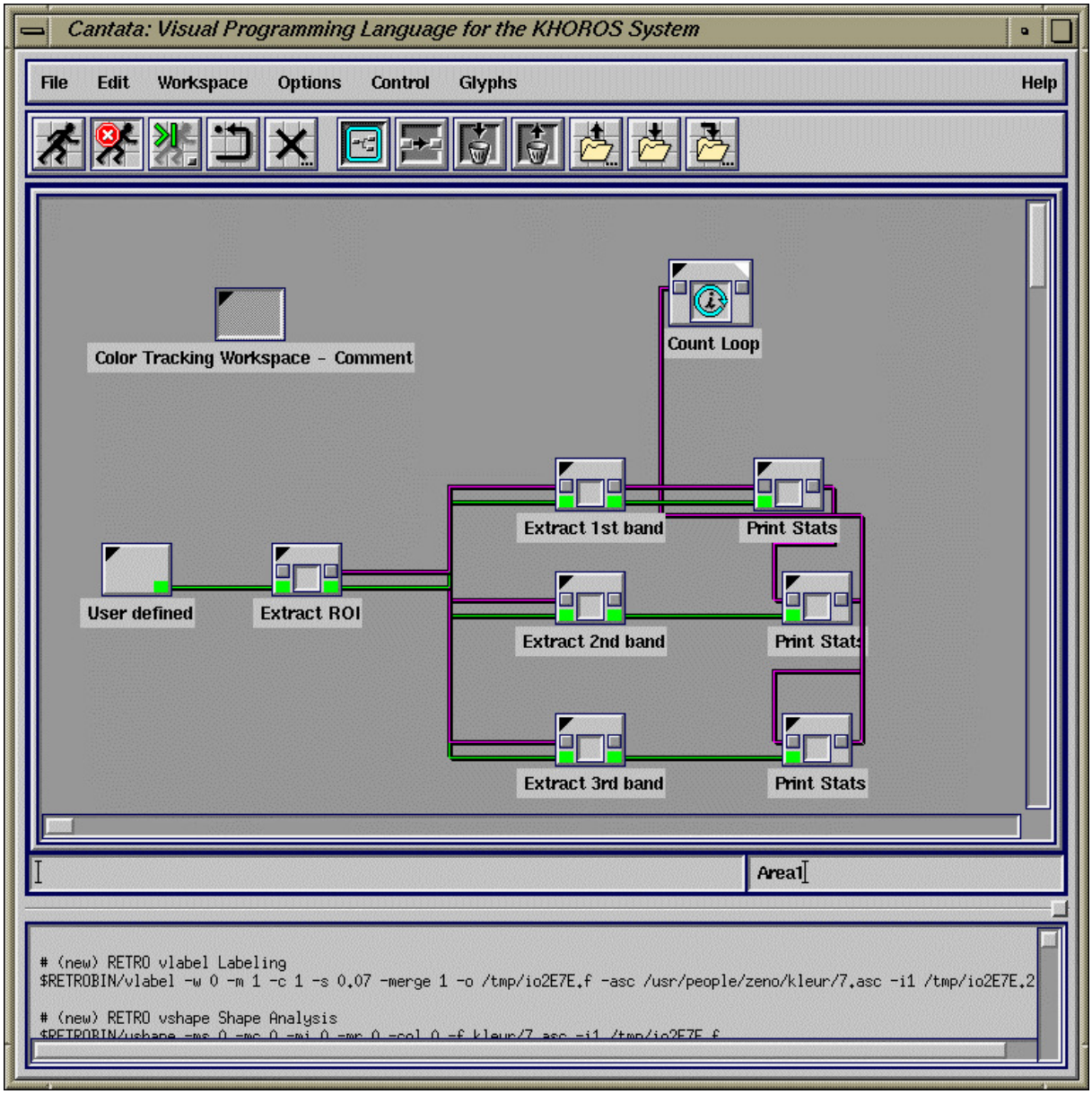


Figura 2 – Ambiente de desenvolvimento Khoros-Cantata (GERADTS; BIJHOLD, 1999)

KUBICA, 1995). O Cantata possui glifos de controle de fluxo simples em termos de sintaxe visual (JOHNSTON; HANNA; MILLAR, 2004).

No Khoros-Cantata, cada glifo é um elemento icônico que representa um programa. Cada conexão dirigida representa um caminho pelo qual os dados fluem mais naturalmente para corresponder a uma representação mental do problema. Os glifos fornecem uma representação icônica de um operador, que pode ser conectado a outro em um fluxo de dados visual, com a possibilidade de atrasar a execução de um glifo até que outro seja executado (YOUNG; ARGIRO; WORLEY, 1995). Para criar um programa visual, o usuário seleciona os programas desejados, coloca os glifos correspondentes no espaço de trabalho e conecta os glifos formando uma rede que indica o fluxo de dados para o processamento da imagem. Os programas criados no espaço de trabalho podem ser executados, salvos e restaurados para serem usados novamente ou modificados

mais tarde (YOUNG; ARGIRO; KUBICA, 1995).

Na área de trabalho do Khoros-Cantata o objeto `ToolboxMenu` é um menu com ferramentas que permitem ao usuário selecionar um operador e criar um glifo. O objeto `CommandBar` é uma barra de comandos que fornece acesso rápido aos recursos mais usados. Para executar uma ação específica, o usuário pode clicar em um botão de comando que representa a ação. São ações suportadas a execução da rede, redefinição da rede, limpeza da área de trabalho, verificação da rede, obtenção de informações sobre a rede, acesso aos mecanismos de recortar, colar, salvar e restaurar uma área de trabalho (YOUNG; ARGIRO; WORLEY, 1995).

Um dos principais problemas do projeto Khoros é a união de dois paradigmas bastante independentes de programação visual: desenvolvimento direto na interface gráfica do usuário e programação visual baseada no conceito de fluxo de dados (GUREVICH et al., 2006).

A Figura 2 ilustra o ambiente de desenvolvimento Khoros-Cantata.

3.1.2 Interactive Data Language (IDL)

Interactive Data Language (IDL) é um instrumento que facilita o aprendizado dos desenvolvedores e integra um grande número de funções adequadas para o tratamento de imagens médicas em 3D. IDL é multiplataforma, oferece ferramenta visual com padrão de dados, leitura, gravação e processamento de imagens digitais. Essas características aumentam a eficiência dos programadores. IDL é adequada para fazer operações massivas de dados e processamento de imagens (XIAOQI; XIN; DONGZHENG, 2012).

3.1.3 Processing

O Processing foi criado para ensinar os fundamentos de programação de computadores dentro de um contexto visual. Pode ser usado como uma ferramenta para escrever esboços de software, incluindo um mecanismo de renderização 2D/3D personalizado. Processing é escrito em Java com recursos extraídos do PostScript e OpenGL. A linguagem é facilmente estendida escrevendo código adicional ou integrando bibliotecas Java existentes. É uma linguagem de programação de texto para processar imagens de maneira responsiva (REAS; FRY, 2004). Não é uma ferramenta de produção comercial, mas é construído especificamente para aprendizado e prototipagem (REAS; FRY, 2003).

Processing fornece três modos diferentes. No modo mais básico, os programas são comandos de linha única para desenhar formas primitivas na tela. No modo mais complexo, o código Java pode ser escrito dentro do ambiente. O modo intermediário permite a criação de software dinâmico em uma estrutura híbrida. O paradigma de programação é o procedural. Há um equilíbrio entre funcionalidades e clareza, o que incentiva o processo de experimentação e reduz a curva de aprendizado (REAS; FRY, 2003).

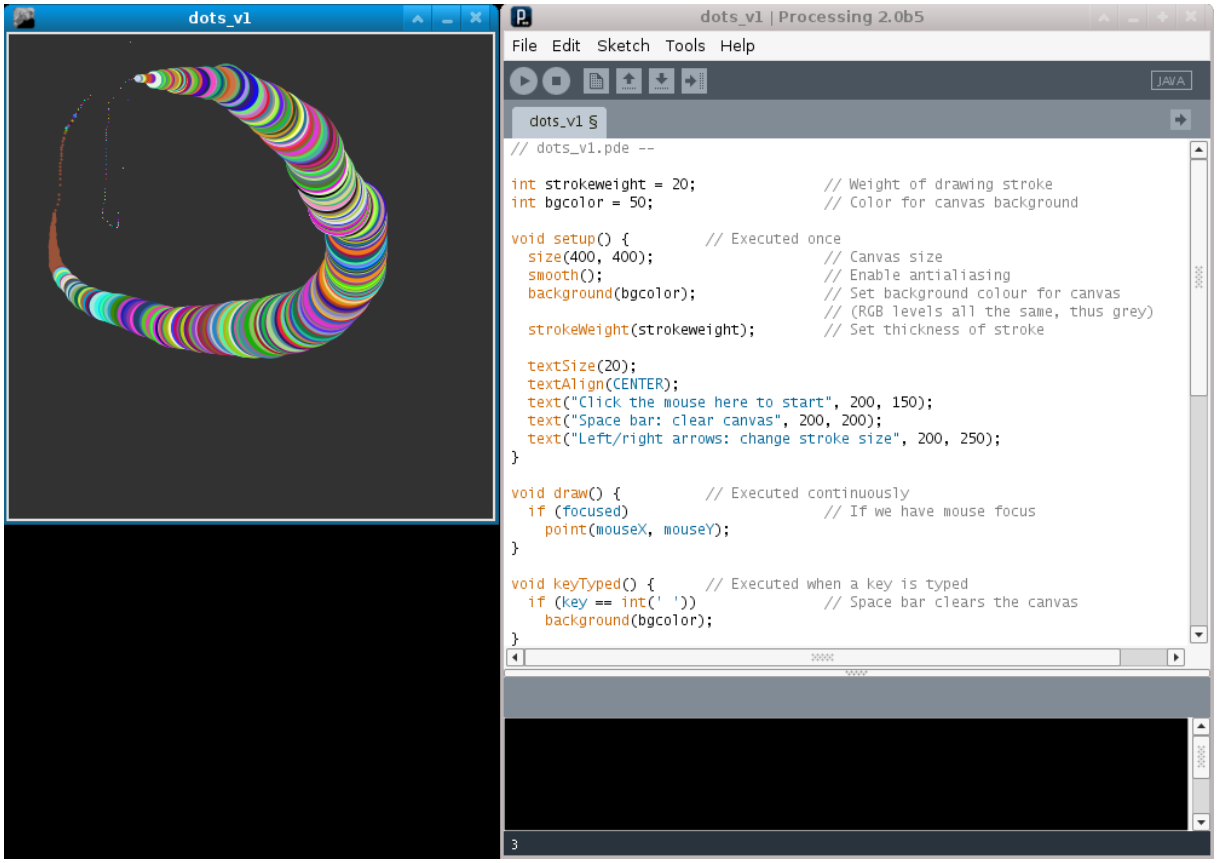


Figura 3 – Ambiente de desenvolvimento Processing (REAS; FRY, 2003)

Processing tem comunidades baseadas na web, o que permite que alunos, educadores, profissionais e operadores estejam envolvidos no uso do software, publicando seus trabalhos e compartilhando conhecimento (REAS; FRY, 2004).

A Figura 3 ilustra o ambiente de desenvolvimento Processing.

3.1.4 Pure Data

Pure Data é uma linguagem de programação visual de código aberto usada para criar programas por meio de uma interface gráfica. Uma caixa representa uma determinada função. As caixas são conectadas com linhas que representam o fluxo de dados (DRYMONITIS, 2015).

Permite que pesquisadores e desenvolvedores criem software graficamente sem escrever linhas de código. Pode ser usado para processar e gerar gráficos 2D/3D. O Pure Data pode trabalhar em redes locais e remotas. É adequada para aprender métodos básicos de processamento multimídia e programação visual, bem como para desenvolver sistemas complexos para projetos de grande escala. As funções algorítmicas são representadas em Pure Data por caixas visuais chamadas objetos, colocados dentro de um área de trabalho chamada canvas. O fluxo de dados entre objetos é obtido por meio de conexões visuais chamadas *patch cords*. Cada objeto executa uma tarefa específica, que pode variar em complexidade, desde operações matemáticas de nível

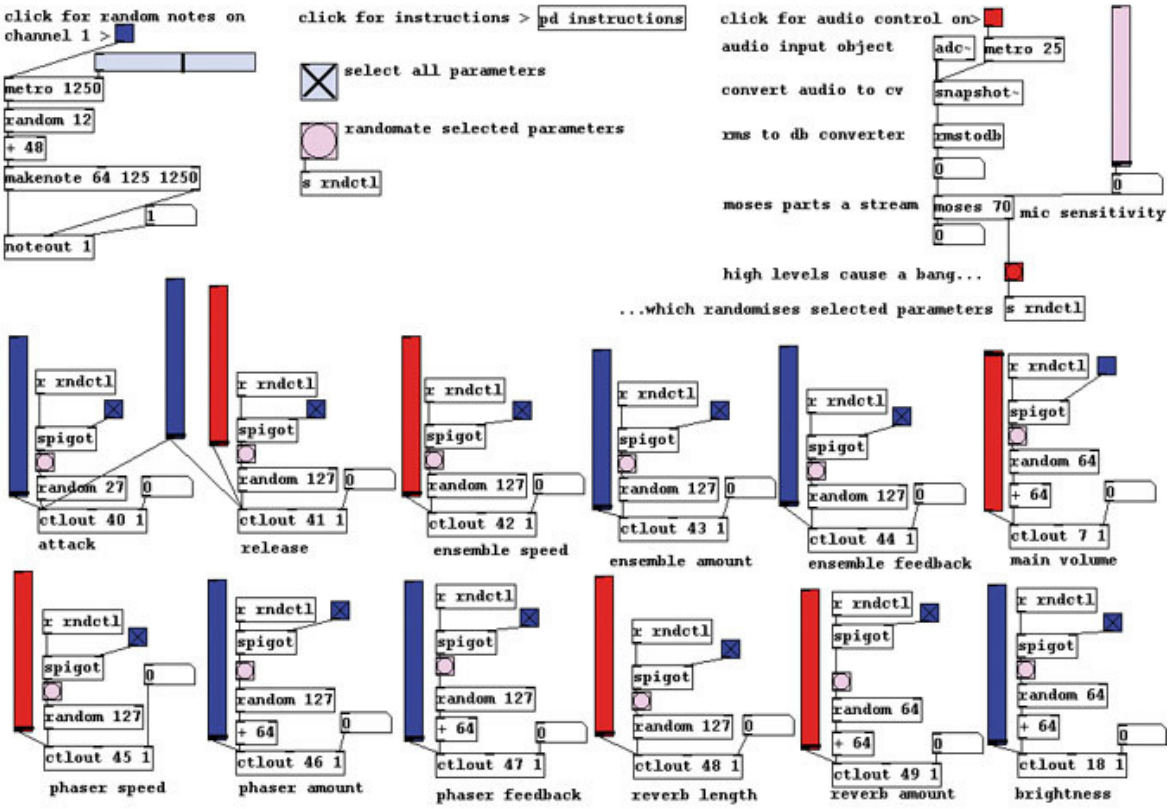


Figura 4 – Ambiente de desenvolvimento Pure Data

(PUREDATA. . . , 2022)

muito baixo até funções complicadas de áudio ou vídeo. Os objetos incluem elementos básicos, objetos externos, compilados de C ou C++ (PUREDATA, 2022).

A Figura 4 ilustra o ambiente de desenvolvimento Pure Data.

3.1.5 CVNodes

Open Source Computer Vision Library (OpenCV) é uma biblioteca de software de visão computacional e aprendizado de máquina de código aberto. O OpenCV foi construído para fornecer uma infraestrutura comum para aplicativos de visão computacional e acelerar o uso da máquina nos produtos comerciais. A biblioteca possui mais de 2.500 algoritmos otimizados. OpenCV tem uma comunidade com mais de 47 mil pessoas e 18 milhões de downloads. A biblioteca é amplamente utilizada em empresas, grupos de pesquisas e por órgãos governamentais. Possui interfaces C++, Python, Java e MATLAB e suporta Windows, Linux, Android e Mac OS. Uma interface CUDA e OpenCL com todos os recursos está sendo desenvolvida. Existem mais de 500 algoritmos e cerca de 10 vezes mais funções que compõem ou suportam esses algoritmos (OPENCV, 2022).

O OpenCV é uma biblioteca muito popular, abrangente, com funções para segmentação de imagens, reconhecimento de objetos, análise de formas, detecção de recursos e rastreamento e

reconstrução tridimensional. Executa na CPU, aproveitando as instruções MMX/SSE disponíveis nos processadores Intel, mas a paralelização é menor que a alcançável com a GPU (BRADSKI; KAEHLER, 2008).

Embora o OpenCV contenha muitos recursos, ainda é fundamentalmente um programa de baixo nível, uma biblioteca voltada para estudantes de visão computacional e pesquisadores com experiência significativa em programação. CVNodes é uma ferramenta de programação visual que visa resolver esse problema e permitir que não-programadores criem, depurem, e iterem algoritmos de visão computacional facilmente. CVNodes é uma interface de alto nível que alavanca o poder de baixo nível do OpenCV e, portanto, fornece acesso ao processamento de imagem geral e algoritmos de visão que são aplicáveis a muitos domínios (WANG; HOGUE, 2020).

O CVNodes permite que o usuário identifique problemas de forma rápida e fácil em cada estágio do *pipeline* de desenvolvimento, permitindo a inspeção dos dados de entrada e saída. O paradigma de interface desenvolvido usa uma linguagem visual baseada em nós hierárquicos que permite aos usuários criar, arrastar, soltar e conectar nós que implementam vários métodos de pré-processamento para formar um *pipeline*. Sua usabilidade, entretanto, ainda precisa ser melhorada para programadores novatos (WANG; HOGUE, 2020).

O sistema fornece uma visão em tempo real das etapas de processamento e da saída resultante. Cada nó pode ser executado independentemente ou de forma automática seguindo o fluxo de dados. Isso permite que desenvolvedores inspecionem a operação e produção de cada nó e determinem o impacto que parâmetros específicos têm nos resultados finais. Cada *pipeline* baseado em nó pode ser salvo, carregado e reutilizado como uma caixa preta. A reutilização do programa produzido permite compartilhar *pipelines* salvos para serem estendidos por vários desenvolvedores (WANG; HOGUE, 2020).

A Figura 5 ilustra o ambiente de desenvolvimento CVNodes.

3.2 Sistemas de processamento paralelo

O uso de imagens médicas é comum no diagnóstico e pesquisa científica. Muitas imagens são convencionais ou bidimensionais, mas as imagens obtidas pela ressonância magnética, tomografia por emissão de pósitrons e tomografia computadorizada são tridimensionais e geralmente ocupam dezenas de megabytes. Essas imagens tridimensionais são armazenadas de duas maneiras diferentes: como pilhas de imagens convencionais, ou seja, um conjunto de arquivos numerados sequencialmente; ou como arquivos que suportam representação de imagem tridimensional, ou seja, arquivos no formato DICOM (DANTAS; LEAL; SOUSA, 2015).

Processamento de vídeo em tempo real e processamento rápido de imagem 3D podem ser bastantes desafiadores por causa do poder de processamento necessário. Para aplicações

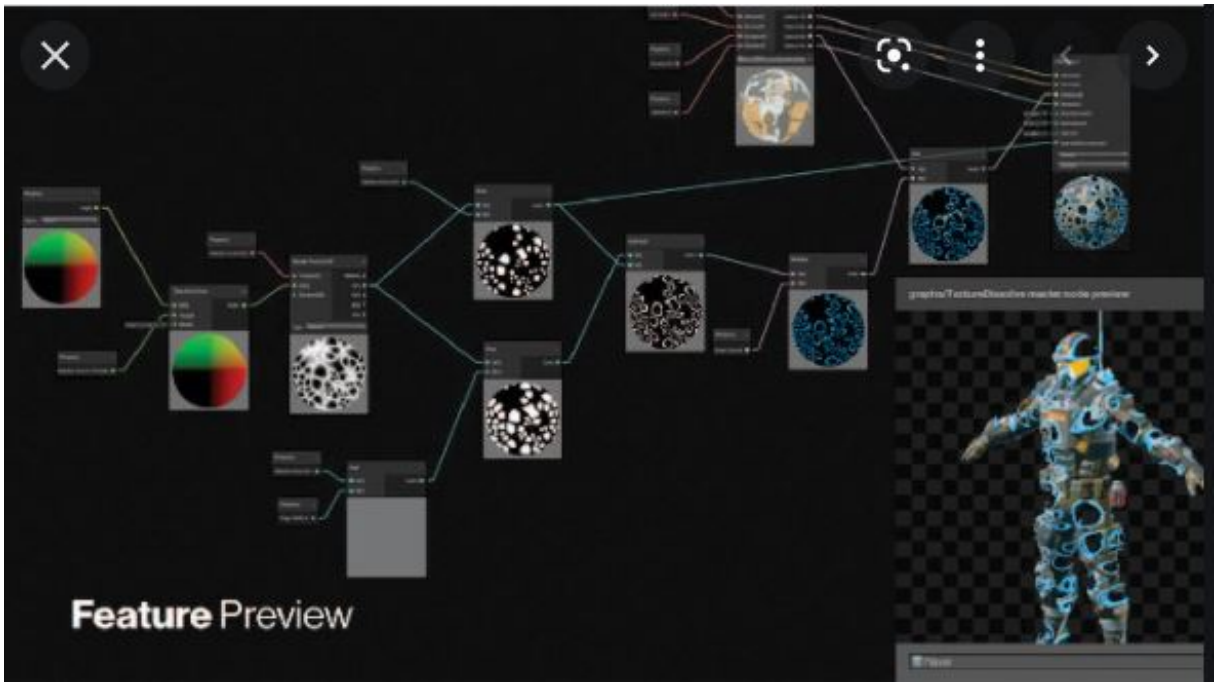


Figura 5 – Ambiente de desenvolvimento CVNode
(WANG; HOGUE, 2020)

específicas, como compactação e descompactação *MPEG*, existem circuitos integrados capazes de executá-los. Mas para aplicações não específicas, há CPUs genéricas ou GPUs programáveis modernas. A vantagem das GPUs é sua capacidade de processamento paralelo, útil para processamento de *pixels* e *voxels* (DANTAS; LEAL; SOUSA, 2015).

Existem sistemas que oferecem processamento paralelo para permitir o tratamento de grandes conjuntos de dados de imagens que demandam um longo tempo de execução. O código de programação para usar o paralelismo é complexo, particularmente ao lidar com dependências de dados, gerenciamento de memória, movimentação de dados e ocupação do processador (BLATTNER et al., 2015).

Existem pelo menos três linguagens que podem ser usadas para programar GPUs: GLSL, CUDA e OpenCL. A linguagem OpenCL, além de ser projetada para abordar problemas genéricos de programação, é compatível com a maioria das marcas de GPUs e por seu design e compatibilidade, é uma linguagem adequada para o processamento rápido de imagens (DANTAS; LEAL; SOUSA, 2015).

O uso do OpenCL não é tão simples quanto às linguagens de alto nível para programar CPUs. São necessárias chamadas de APIs para executar tarefas indiretamente relacionadas ao trabalho. Antes de realmente processar a imagem, é necessário detectar dispositivos compatíveis com o OpenCL no computador, inicializar o contexto e a fila de execução do OpenCL, compilar e vincular o código que será executado na GPU (*shader*) e transferir a imagem e os parâmetros para a GPU (DANTAS; LEAL; SOUSA, 2015).

São sistemas que oferecem processamento paralelo a VisionGL e *Hybrid Task Graph Scheduler* (HTGS). Apesar da biblioteca VisionGL ser um sistema para processamento paralelo, suas características serão detalhadas na seção 3.4.1 juntamente com os sistemas de geração automática de código.

3.2.1 *Hybrid Task Graph Scheduler* (HTGS)

HTGS é um *framework* e sistema de tempo de execução que oculta o movimento dos dados, maximiza a ocupação do processador quando executado em computadores híbridos (com CPU e GPU) e gerencia o uso da memória para permanecer dentro das limitações do sistema. O HTGS aumenta a produtividade do programador ao implementar *wokflows* híbridos que se adaptam a sistemas multi-core e multi-GPU. Fluxos de trabalho híbridos são eficazes em paralelizar um algoritmo, ocultando o movimento dos dados e mantendo os processadores ocupados (BLATTNER et al., 2015).

HTGS é implementado em Java. O desenvolvimento de fluxos de trabalho híbridos é complexo e demorado. HTGS auxilia na implementação de fluxos de trabalho híbridos usando gráficos de tarefas e incluindo um sistema de tempo de execução para agendar as tarefas em coleções híbridas de recursos de computação, CPUs e GPUs. HTGS ajuda a construir gráficos de tarefas que lidam com dependências, gerencia a memória, CPU e GPU, escala para sistemas multi-GPU por meio de execução de *pipelines* de processamento e grava as informações processadas. Cada tarefa criada por meio do HTGS expõe o recursos e vincula tarefas ao hardware físico (BLATTNER et al., 2015).

3.3 Sistemas de *workflow*

Existem os sistemas de *workflow*, que permitem ao usuário desenvolver algoritmos como uma coleção de blocos de funções, que podem ser vinculados graficamente a *pipelines* de entrada e saída (HAMIDIAN et al., 2014). Esses sistemas que diminuem a curva de aprendizado para programadores iniciantes. Os sistemas que usam *workflow* são Taverna Workbench, MATLAB e ImageFlow.

3.3.1 Taverna Workbench

Taverna Workbench é um sistema de fluxo de trabalho para bioinformática, para auxiliar programadores na análise de algoritmos de processamento de imagens visualmente e sem código. Programadores não especialistas podem criar um fluxo de trabalho prático. Os programadores podem controlar graficamente as variáveis de entrada, saída e realizar simulações através do fluxo de dados. Eles não precisam estar familiarizados com os detalhes do sistema *back-end* (KAEWKEEREE; TANDAYYA, 2012).

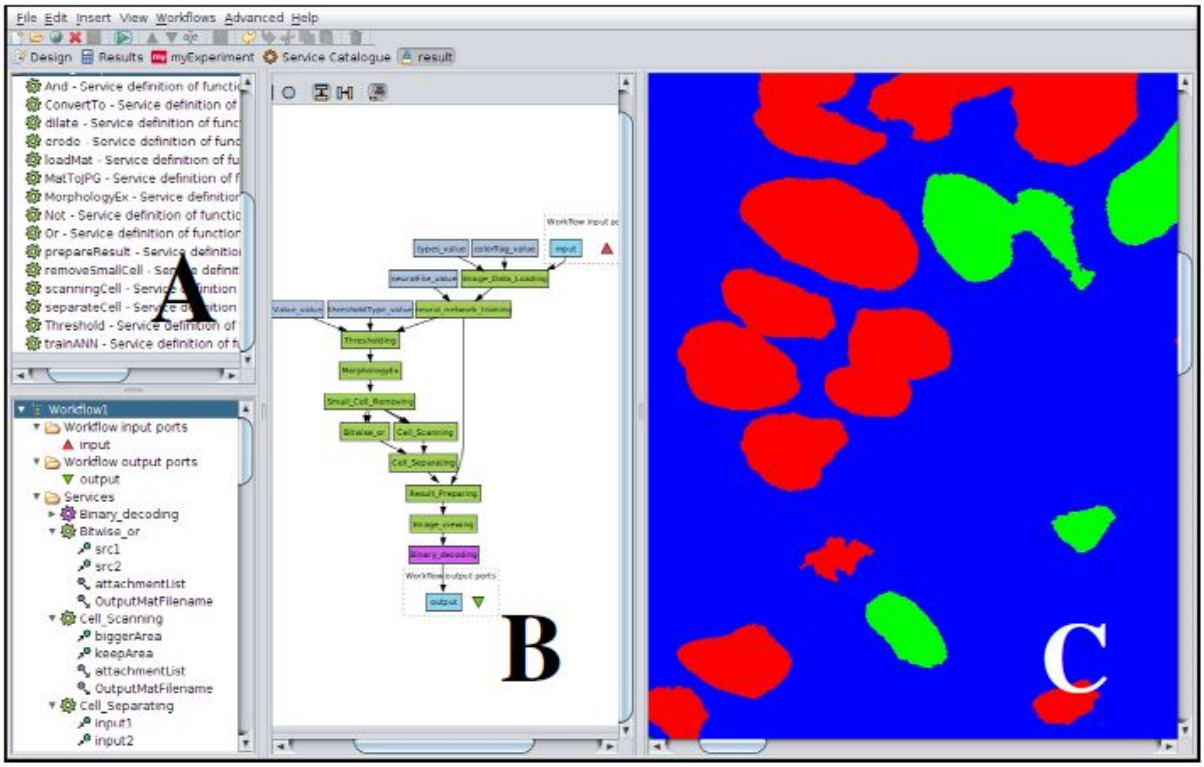


Figura 6 – Ambiente de desenvolvimento Taverna
(KAEWKEEREE; TANDAYYA, 2012)

O Taverna Workbench é um sistema de código aberto, que possui um editor com interface gráfica do usuário e um mecanismo executor de *workflow*. Para controlar o *workflow*, os processadores são conectados através dos links de dados e trocam dados entre si. Taverna Workbench tem a vantagem de permitir a visualização dos processos de trabalho, principalmente os serviços baseados na Internet e componentes mais reutilizáveis. Existe uma comunidade online para usuários que compartilham *workflows* científicos e recursos computacionais (KAEWKEEREE; TANDAYYA, 2012).

Taverna Workbench é um serviço web com duas funções representativas: servidor e cliente. A função servidor provê serviços de processamento de imagem. O servidor web pode estar na mesma máquina que o cliente ou em um servidor remoto que pode fornecer melhor desempenho. A função cliente oferece suporte ao design de *workflow* para as sequências operacionais e algoritmos de processamento de imagem. As portas de entrada e saída do processador correspondem à interface de serviço (KAEWKEEREE; TANDAYYA, 2012).

A Figura 6 ilustra o ambiente de desenvolvimento Taverna.

3.3.2 MATLAB

MATLAB é uma ferramenta poderosa para processamento de imagens. Iniciantes em programação podem resolver problemas técnicos mais rapidamente com *workflows* do que

programando em linguagens tradicionais como C e C++. Porém, sua arquitetura é muito complexa e torna o aprendizado difícil para iniciantes, pois seu processo de algoritmo fixo não permite flexibilidade de adaptação (TONG; CHENG-DONG; DONG-YUE, 2011).

3.3.3 ImageFlow

ImageFlow permite que as imagens sejam processadas por software legado distribuído em conjunto com sistemas de destino interconectados. Os principais recursos do ImageFlow são processamento de imagem baseado em *workflow*, General Running Service (GRS) para programas legados, mecanismo de transferência de dados adaptável e implantação de software baseada em *workflow* (CAO et al., 2009).

O processamento de imagem baseado em *workflow* é realizado com lógicas como sequência, condição, iteração, simultaneidade e sincronização. O serviço de execução geral é usado para reutilizar software legado sem reengenharia de seus códigos-fonte, usando General Running Service (GRS) como uma estrutura de execução do programa. O mecanismo de transferência de dados adaptável considera o volume de dados a transferir entre a entrada e a saída dos serviços. A implantação de software baseada em *workflow* possibilita a execução de sistemas distribuídos em escala de forma eficiente. Assim, componentes de software podem ser transferidos automaticamente para uma série de *hosts* em paralelo com o objetivo de executar o fluxo de trabalho (CAO et al., 2009).

3.4 Sistemas de geração automática de código

Por fim, existem os sistemas que facilitam a programação e aumentam a produtividade por meio da geração automática de código. Exemplos de sistemas que geram código automaticamente são VisionGL, *NVIDIA Performance Primitives* (NPP) e *Model-Integrated Computing* (MIC).

3.4.1 VisionGL

As unidades de processamento gráfico, também conhecidas como GPU's são construídas para renderizar cenas sintéticas mais rapidamente que a CPU. O processamento gráfico requer cálculos sobre grandes conjuntos de dados de vértices e fragmentos. Os vértices são os elementos fundamentais do poliedro que compõem uma cena sintética. Fragmentos são como pixels da imagem final, mas com diferentes profundidades (DANTAS; BARRERA, 2011).

É natural usar o poder de processamento das GPU's para acelerar cálculos paralelizáveis, como simulações baseadas em equações diferenciais, operações com matrizes, processamento de imagem e sinal (OWENS et al., 2005).

As unidades de processamento gráfico de vários núcleos (GPU) fornecem uma plataforma de hardware paralela de alto desempenho no desktop com um custo baixo. No entanto, o uso

generalizado desta capacidade computacional é prejudicado pelo fato de que a programação de GPUs é difícil (LI et al., 2012).

VisionGL é uma biblioteca de código aberto que fornece um conjunto de funções de processamento de imagem acelerado, por exemplo, operações *pixelwise*, convolução, morfologia matemática clássica e difusa, com dilatação, erosão, abertura, fechamento, dilatação condicional e reconstrução (DANTAS; LEAL; SOUSA, 2016). A VisionGL ajuda a criar operadores de processamento de imagem, gerando automaticamente o código *wrapper* e otimizando as transferências de imagem entre *random access memory* (RAM) e GPU (DANTAS; OLIVEIRA; LEAL, 2017).

VisionGL cria *shaders* ocultando as complexidades do OpenGL. Encadeia códigos de *shaders* para tornar possível a criação de um *pipeline* de processamento de alto desempenho. Esse encadeamento é possível através do uso de um código wrapper, que prepara os dados e os envia à GPU para que o usuário não se preocupe com o OpenGL. O código do wrapper é gerado, em tempo de compilação, a partir do código-fonte dos *shaders*. *Shader* é o nome das funções executadas na GPU (DANTAS; BARRERA, 2011). Os shaders mais comuns são os de fragmentos, vértices e computação (SELLERS; WRIGHT; JR, 2016).

3.4.2 NVIDIA Performance Primitives (NPP)

O desenvolvimento de código eficiente com base na computação em GPUs requer bastante cuidado devido às várias restrições no acesso à memória, cache e sincronização inerente à complexa arquitetura da GPU. Um dado programa (kernel) com uma *thread* de ajuste fino e alocação de memória pode dar excelentes resultados em uma GPU particular, mas pode ter um desempenho insatisfatório em um dispositivo diferente de outro fornecedor ou geração de tecnologia. Compute Unified Device Architecture (CUDA) e *frameworks* OpenCL tentam ocultar os detalhes específicos da plataforma, mas fornecem apenas informações básicas para obter o desempenho máximo em qualquer GPU (LI et al., 2012).

Reconhecendo a necessidade de apoiar usuários que não são especialistas em processamento paralelo e não querem se tornar programadores de sistemas, a NVIDIA Performance Primitivos (NPP), uma biblioteca de imagem, vídeo e sinal funções de processamento foi criada com base no CUDA (HALFHIL, 2008; BOYER; SKADRON; WEIMER, 2008). A NPP tem mais de 2.000 comandos primitivos de processamento para melhorar o desempenho do código executado exclusivamente na CPU, dependendo do tipo de GPU e CPU envolvidos (WANG; HUANG, 2008).

A biblioteca NPP facilita a geração automática de código CUDA, mas a utilização da NPP ainda requer que se aprenda CUDA (LI et al., 2012).

3.4.3 Model-Integrated Computing (MIC)

O desenvolvimento de algoritmos de processamento de imagens digitais envolve experimentação, durante a qual os usuários criam protótipos de implementações e ajustam os parâmetros até o efeito desejado ser obtido. A melhoria de desempenho em dez vezes fornecida por GPUs pode aumentar a velocidade de processamento, mas apenas se o código desenvolvido não retardar o processo. Para a maioria dos usuários, isso só é possível com suporte de ferramentas de alto nível (LI et al., 2012).

MIC é um ambiente gráfico para projetar *workflows* de processamento de imagem que geram automaticamente todo o código CUDA, incluindo chamadas NPP necessárias para executar o aplicativo em uma GPU. Os usuários podem arrastar e soltar componentes e conectá-los para criar seus *workflows*. As imagens são trocadas entre nós consecutivos usando ponteiros para evitar a cópia desnecessária de dados. O interpretador analisa automaticamente toda a hierarquia do modelo e sintetiza o código CUDA. O tempo de execução com o código NPP se mostra mais rápido do que um código implementado sequencialmente e executado na CPU (LI et al., 2012).

3.5 Tecnologias utilizadas no processamento de imagens médicas na nuvem

O surgimento do conceito de computação em nuvem permitiu que empresas implantassem serviços baseados em infraestruturas em nuvem sem precisar gerenciar e manter infraestrutura local. Algumas empresas terceirizam seus serviços e aplicativos. Existem alguns provedores de nuvem, como, Amazon AWS, Google e Rackspace, que oferecem muitos serviços em nuvem, como, por exemplo, computação elástica, armazenamento, bancos de dados e sistemas de notificação (MONTEIRO; SILVA; COSTA, 2012).

A demanda por serviços de saúde tem aumentado devido a altas taxas de crescimento populacional e maior expectativa de vida. No entanto, o setor da saúde ainda sofre com a escassez de pessoal e fundos para atender à enorme demanda. Construir e manter *data centers* internos levaria a um aumento de despesas. Por esse motivo, a computação em nuvem é uma solução adequada para promover a utilização de tecnologias de informação e comunicação em toda a área médica. Atualmente, existem diferentes definições de computação em nuvem, mas a mais conhecida é fornecida pelo National Institute of Standards and Technology (NIST), que define a nuvem como um modelo para fornecer sob demanda recursos computacionais pela Internet. Esses recursos são oferecidos como um serviço e cobrados apenas pelo tempo e espaço realmente usado. Nesse modelo, os provedores de nuvem garantem a manutenção e segurança destes serviços (MARWAN; KARTIT; OUAHMANE, 2017).

Os fornecedores destes serviços na nuvem oferecem software como serviço, plataforma como serviço e infraestrutura como serviço.

3.5.1 Software como serviço

A oferta de software como serviço é muito comum hoje em dia e é possível encontrar muitos serviços na Internet, assim como aplicativos do Google, aplicativos de redes sociais, como Facebook e SalesForce.com. Essas aplicações geralmente têm uma interface baseada na web (MONTEIRO; SILVA; COSTA, 2012).

Na saúde há ferramentas para processar imagens médicas, disponíveis em qualquer horário e lugar. Usando essas ferramentas, os profissionais de saúde enviam uma imagem a estudar para o provedor do serviço na nuvem. Em seguida, os provedores de nuvem usam ferramentas avançadas para analisar esta imagem. Então, o resultado da análise é devolvido ao profissional de saúde (MARWAN; KARTIT; OUAHMANE, 2017). São exemplos de softwares como serviço para processamento de imagens médicas CloudMed e Java Image Science Toolkit (JIST).

3.5.1.1 CloudMed

CloudMed é um software como serviço que pode ser usado por instituições médicas para fornecer um serviço confiável para teleconsulta e telediagnóstico. Um radiologista pode interagir em tempo real com outras pessoas por meio do compartilhamento de arquivos, mensagens, visualização e análise de imagens médicas usando o serviço. Um usuário que tem acesso ao serviço pode fazer consultas de imagens DICOM, recuperar imagens médicas, armazenar essas imagens na nuvem e compartilhar com outros radiologistas. Um mecanismo automático de anonimização também é fornecido para permitir compartilhamento não identificado de exames. Usando o paradigma Web 2.0 de programação e o padrão HTML5 foi possível criar uma interface fácil de usar, onde o usuário interage com o aplicativo da web muito semelhante ao aplicativo de desktop (MONTEIRO; SILVA; COSTA, 2012).

O CloudMed é composto por uma camada de banco de dados, um gateway do sistema de informações de gerenciamento médico e um aplicativo cliente da web. O núcleo é dividido em três principais componentes: o servidor Openfire, o Remote Archive, o Core Business Database and Protocol. Cada um desses componentes possui uma finalidade no sistema. O servidor Openfire suporta o núcleo do serviço. O Remote Archive, suporta o banco de dados e o protocolo implementados como plug-ins Openfire. O servidor Openfire XMPP é responsável por gerenciar as mensagens e notificações. Um radiologista ou médico deve ser registrado como colaborador no servidor para acessar as funcionalidades disponíveis (MONTEIRO; SILVA; COSTA, 2012).

Através servidor Openfire, o usuário pode gerenciar amigos e grupos de amigos. Um radiologista pode adicionar outro radiologista à sua lista e criar grupos, associando radiologistas a cada grupo. Core Business Database and Protocol é um importante componente implementado como um plug-in Openfire. Este componente traz toda a semântica para o sistema, permitindo que o servidor interprete as funcionalidades personalizadas construídas sobre o protocolo. O Remote

Archive é o componente responsável por gerenciar a área de armazenamento (MONTEIRO; SILVA; COSTA, 2012).

3.5.1.2 Java Image Science Toolkit (JIST)

O Java Image Science Toolkit (JIST) é uma plataforma de processamento de imagens integrada à nuvem Amazon Elastic Compute Cloud (EC2) que combina uma solução para o processamento interativo de dados com uma infraestrutura eficiente de processamento em lote em grande escala. Tem como objetivo aumentar a diversidade de pesquisadores que podem usar ferramentas modernas de processamento de imagens e análises genéticas. Os desenvolvedores podem construir seus algoritmos usando as funcionalidades de análise de imagem da JIST. Centenas de módulos estão atualmente disponíveis para entrada e saída de imagem, manipulação, segmentação e registro (MIRARAB; FARD; SHAMSI, 2014).

O JIST garante a interoperabilidade fornecendo interfaces gráficas de usuário e ferramentas de processamento em lote. O processamento de imagens médicas envolvendo grandes conjuntos de dados multimodais e dados de imagem requer o gerenciamento de uma grande quantidade de dados e requer infraestrutura de computação de alto desempenho (MIRARAB; FARD; SHAMSI, 2014).

3.5.2 Plataforma como serviço

Plataforma como serviço disponibiliza ferramentas para desenvolvimento, linguagens de programação e bancos de dados. Instituições médicas usam essa plataforma para desenvolver seus próprios sistemas de processamento de imagem (MARWAN; KARTIT; OUAHMANE, 2017).

Plataforma como serviço permite que os consumidores de aplicativos não se preocupem com as limitações tradicionais de hardware. Os desenvolvedores concentram-se na construção de aplicativos e não com sistemas operacionais, escala de infraestrutura, balanceamento de carga e tarefa de administração do sistema (MONTEIRO; SILVA; COSTA, 2012). TOMAAT é um exemplo de plataforma como serviço.

3.5.2.1 TOMAAT

TOMAAT é um programa de código aberto escrito em Python que permite desenvolver algoritmos complexos de análise de imagens médicas. Permite que os pesquisadores tornem seus algoritmos acessíveis como um serviço, melhorando seus modelos e *pipelines* de processamento de dados. Os usuários das clínicas acessam os serviços através de uma interfaces simples de adotar e operar. Os pesquisadores publicam seus algoritmos colaborando com a comunidade (MILLETARI et al., 2019).

A arquitetura TOMAAT pode ser dividida em três partes: servidor, cliente e serviços. **Servidor** é uma máquina equipada com hardware adequado, que executa vários algoritmos de

análise de imagem como um serviço. Os servidores são hospedados e mantidos por desenvolvedores de algoritmos que estão interessados em disseminar suas soluções para os usuários finais, os clientes, por meio de uma conexão de rede local ou remota. O TOMAAT fornece uma interface de comunicação, realizada através do protocolo HTTP. Os desenvolvedores podem escolher se desejam tornar seus servidores públicos ou não, registrando-os em um diretório de índice global conhecido como serviço de anúncio. **Clientes** possibilitam aos usuários finais acessar e utilizar soluções oferecidas pelos servidores de maneira direta através de uma interface do usuário. Como algoritmos e hardware são hospedados e mantidos por desenvolvedores no lado do servidor, As interfaces dos usuários do cliente podem ser reduzidas, solicitando apenas aos usuários que definam as entradas e saídas de algoritmos necessários. Resultados recebidos dos servidores podem ser processados ou armazenados posteriormente. **Serviço de anúncio** é um sistema que mantém uma lista atualizada de todos os servidores públicos e soluções que os desenvolvedores se registraram como pontos válidos (MILLETARI et al., 2019).

Com TOMAAT as clínicas não precisam investir na criação e suporte de infraestrutura tecnológica, mantendo as soluções dos clientes limitadas e deixando os desenvolvedores cuidarem das configurações necessárias. Pesquisadores podem comparar métodos de análise de imagens e publicar seus resultados. Os equipamentos dos usuários não requerem GPUs para executar grandes modelos. Os modelos podem ser atualizados no servidor de forma transparente para o usuário. Todos os componentes do TOMAAT, incluindo o serviço de anúncios, pode ser configurado em um ambiente de rede seguro (MILLETARI et al., 2019).

3.5.3 Infraestrutura como serviço

Os provedores de nuvem dependem de tecnologia de virtualização para oferecer recursos para a organização de saúde implantar aplicativos de imagens médicas de maneira econômica. Estes recursos por demanda são escaláveis para atender às necessidades dos consumidores. A computação em nuvem depende de sistemas distribuídos para melhorar a disponibilidade e desempenho. Este modelo de entrega é usado principalmente para armazenar imagens médicas para promover o intercâmbio de dados entre o ecossistema de saúde (MARWAN; KARTIT; OUAHMANE, 2017).

A infraestrutura como serviço consiste em uma grade de servidores virtualizados para armazenamento, *firewalls*, balanceadores de carga e redes. Os consumidores podem ter acesso total ao sistema operacional, *firewalls*, roteadores e balanceamento de carga (MONTEIRO; SILVA; COSTA, 2012).

3.6 VGLGUI

VGLGUI é uma interface gráfica de programação visual que expõe as funcionalidades da VisionGL. Foi especificada em um artigo de conferência (MACIEL; SOARES; DANTAS, 2021).

A arquitetura da VGLGUI permite a programação visual através de relações entre glifos e suas conexões. Esta arquitetura tem como referência o Khoros-Cantata.

VGLGUI não permitirá a criação de novas funções diretamente na interface, mas será possível agregar funcionalidade através da inclusão de novas bibliotecas, isto evitará a geração de erros difíceis de identificar visualmente, o que ocorre no Khoros-Cantata.

A especificação da VGLGUI prevê a geração automática do código do interpretador de *workflow*.

4

Descrição da arquitetura VGLGUI

4.1 Arquitetura do sistema VGLGUI

Este capítulo descreve a arquitetura da VGLGUI, em múltiplas visões, utilizando o padrão arquitetural ISO/IEC/IEEE 42010:2011, 4+1 View Model of Software Architecture e da UML.

VGLGUI fornece funcionalidades para criar, editar, executar e interromper a execução de arquivos de *workflow* por meio de glifos. Um glifo é uma representação icônica de uma função conectada a outras, formando um fluxo de processamento de dados. Essas funções chamam as funções da VisionGL. A subseção 4.1.5 ilustra as seis camadas da VGLGUI. Cada camada fornece uma interface estrita e bem definida com as camadas acima dela. O interpretador de *workflow*, detalhado no Capítulo 5, faz parte do protótipo implementado.

Os *stakeholders* da VGLGUI são estudantes de pós-graduação, administradores de sistema, gerentes de projeto e especialistas em processamento de imagem. As preocupações dos alunos de pós-graduação são a fácil implementação de *workflow*, a persistência de *workflows* entre as sessões, o compartilhamento de *workflow*, o salvamento das imagens resultantes no final, o uso de seus conjuntos de dados privados e a privacidade dos dados do usuário. As preocupações dos administradores de sistema são a facilidade de manutenção e implantação de servidores e a disponibilidade de servidores de processamento de imagem na Internet. É uma preocupação dos gerentes de projeto manter o código-fonte com facilidade e dos especialistas em processamento de imagem adicionar novas funções com facilidade.

4.1.1 Decisões de arquitetura

A descrição arquitetural deve registrar as decisões arquiteturais consideradas essenciais para o sistema de interesse (ISO, 2011). A Tabela 1 lista as preocupações dos *stakeholders* da VGLGUI com as decisões arquiteturais feitas para o desenvolvimento da interface gráfica e as consequências dessas decisões. As decisões arquiteturais mais importantes da VGLGUI são as

Tabela 1 – Registros de decisão de arquitetura

Stakeholder	Concerne	Id Decisão	Consequência
estudantes pós-graduação	1 - Fácil implementação de *workflow*	j, k	Maior dificuldade de implementação
	2 - Persistência de *workflows* entre as sessões	a	Maior dificuldade na implementação
	3 - Compartilhamento de *workflows*	b	Maior dificuldade na implementação
	4 - Salvando as imagens resultantes no final	c	Maior dificuldade de implementação e segurança
	5 - Usando seus conjuntos de dados privados	d	Maior dificuldade na implementação e segurança
	6 - Privacidade dos dados do usuário.	f	Maior dificuldade em segurança
administradores sistemas	7 - Facilidade de manutenção	e, i	Manutenção e implementação mais fáceis
	8 - Facilidade de implantação de clientes e servidores	g, l	Manutenção e implantação mais fáceis
	9 - Disponibilidade de servidores na Internet	f	Maior dificuldade em segurança
gerentes projetos	10 - Facilidade de manutenção	e, i	Manutenção e implementação mais fáceis
	11 - Facilidade de implementação	e, h, i	Manutenção e implementação mais fáceis
especialistas processamento imagens	12 - Adicionar novas funções com facilidade	e	Manutenção e implementação mais fáceis

seguintes:

a) Deve ser usado um sistema de gerenciamento de banco de dados (SGBD) ou técnica de serialização de dados.

b) O *workflow* deve ter uma representação externa em formato de arquivo para que possa ser baixado e compartilhado.

c) O servidor deve ter acesso de gravação ao sistema de arquivos.

d) O sistema deve permitir o upload ou leitura de arquivos de qualquer sistema de arquivos na rede.

e) Parte do código fonte será gerado automaticamente.

f) Segurança por meio da autenticação do usuário.

g) Arquitetura da solução com cliente, editor de *workflow* e servidor.

h) O agendamento é executado usando um algoritmo de ordem de chegada *first-come first-served* (FCFS).

i) Apenas um nível de controle de agendamento no cliente é necessário e o servidor não precisa controlar a alternância entre CPU e GPU.

j) Haverá um editor visual de *workflow*, que será desenvolvido utilizando a API WebGL (Web Graphics Library), que é uma API JavaScript.

k) Uso do elemento canvas HTML5, que suporta renderização de gráficos 2D e 3D.

l) Desenvolvimento de uma aplicação web sem a necessidade de *plug-ins* nos navegadores.

4.1.2 Cenários de uso

Nesta seção, descrevemos dois cenários de uso típico da VGLGUI. O usuário da VGLGUI deve conhecer minimamente as operações de processamento de imagens, o que fazem e para

que servem. Não é necessário conhecer detalhes de suas implementações, pois a programação é visual.

Cenário 1. Um estudante de pós-graduação que trabalha com processamento de imagens médicas está iniciando seu trabalho diário. Ontem ele trabalhou em um *workflow* de processamento de imagens médicas e hoje seu plano é finalizá-lo. Seu objetivo é pré-processar um lote de alguns milhares de imagens e gravar as imagens resultantes para usar depois. Ele carrega o *workflow* inacabado e adiciona algumas operações. Ele o executa em um subconjunto de algumas imagens e visualiza o resultado na tela. Mais tarde, satisfeito com os resultados, decide aplicar o *workflow* a todo o conjunto de dados, salvando as imagens resultantes em uma pasta enquanto visualiza o resultado na tela.

Cenário 2. Um cenário complementar para descrever a arquitetura considera que o administrador do sistema é o responsável pela manutenção de um servidor com uma CPU potente. Hoje, ele adicionará uma nova GPU ao servidor e configurará o sistema para usá-la. Ele reconfigura o sistema e rapidamente os usuários podem usar a nova GPU para executar seus *workflows*. Mais tarde, ele recebe um e-mail de um usuário com o código-fonte de um *shader* OpenCL. Ele grava o arquivo na pasta apropriada e executa um script que atualiza os binários do servidor e do cliente. Os novos binários têm um novo glifo e uma nova função de *wrapper*, permitindo o uso do novo *shader*. Ele responde ao email informando que a nova função já está disponível no sistema.

4.1.3 Visão lógica

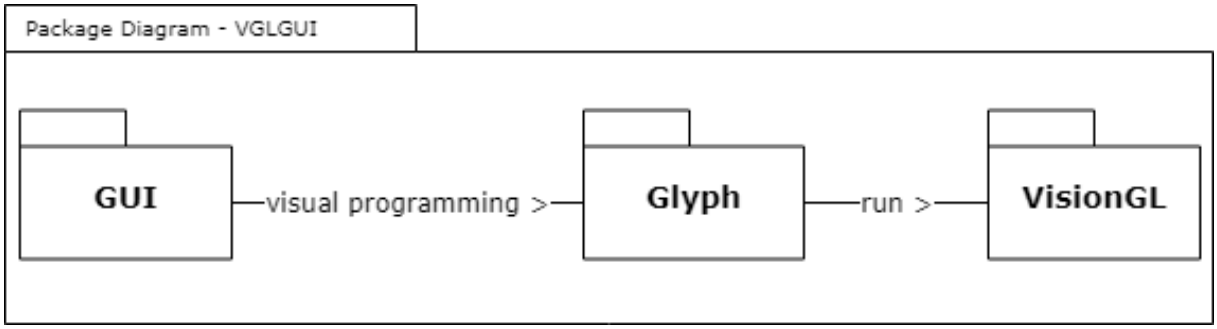


Figura 7 – Diagrama de pacotes VGLGUI

A Figura 7 mostra como os elementos da interface VGLGUI são estruturados e conectados. A interface gráfica do usuário fornece a funcionalidade para criar, editar, executar e interromper a execução de arquivos de processamento de imagem visual por meio de glifos. O glifo é uma representação icônica de uma função que pode ser conectada a outras formando um fluxo de processamento de dados. Essas funções chamam as funções da VisionGL.

As classes, com seus atributos e métodos, representam a interface gráfica para a biblioteca VisionGL e são ilustradas na Figura 8.

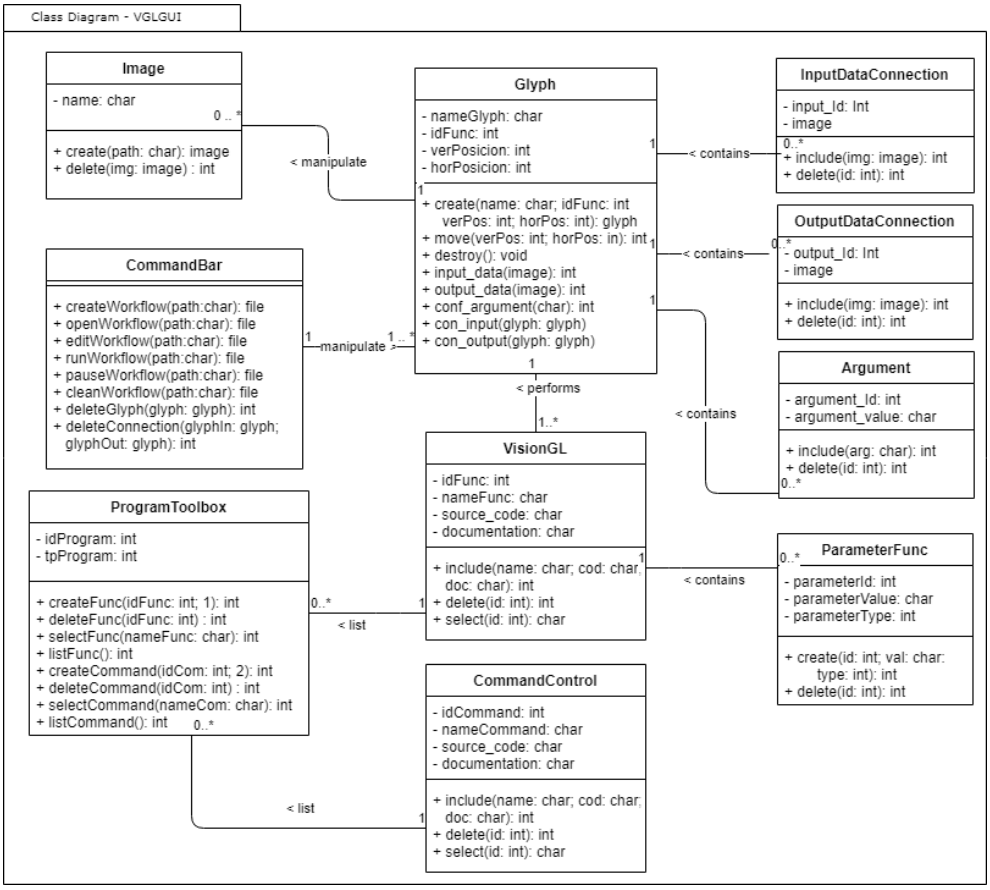


Figura 8 – Diagrama de classes

A classe `Glyph` representa uma funcionalidade da biblioteca VisionGL a ser executada no *workflow* de processamento de imagem. É possível configurar os parâmetros e atribuir as imagens de entrada e saída. A classe `CommandBar` possui os métodos para criar, editar, copiar, excluir e mover o glifo no espaço de trabalho. É possível executar e pausar o glifo e conectá-lo a outro glifo.

A classe `ProgramToolbox` lista as operações de imagem da VisionGL, como dilatação e erosão; controlar comandos de fluxo, como criação de procedimento, estrutura condicional, criação de loops para contar execuções, criação de loops para realizar atividades enquanto uma condição é atendida, estabelecendo o caminho de fusão, interrompendo a execução e estabelecendo expressões; e funções de visualização para imagens de pré-processamento e pós-processamento. O usuário pode desenvolver seu *workflow* selecionando os comandos disponíveis no `ProgramTollBox`.

4.1.4 Visão do processo

O diagrama de sequência ilustrado na Figura 9 mostra as mensagens trocadas entre as classes VGLGUI durante a prototipagem de um *pipeline* de processamento de imagem. No cenário descrito, considera-se que o usuário abriu a interface VGLGUI e criou um arquivo de

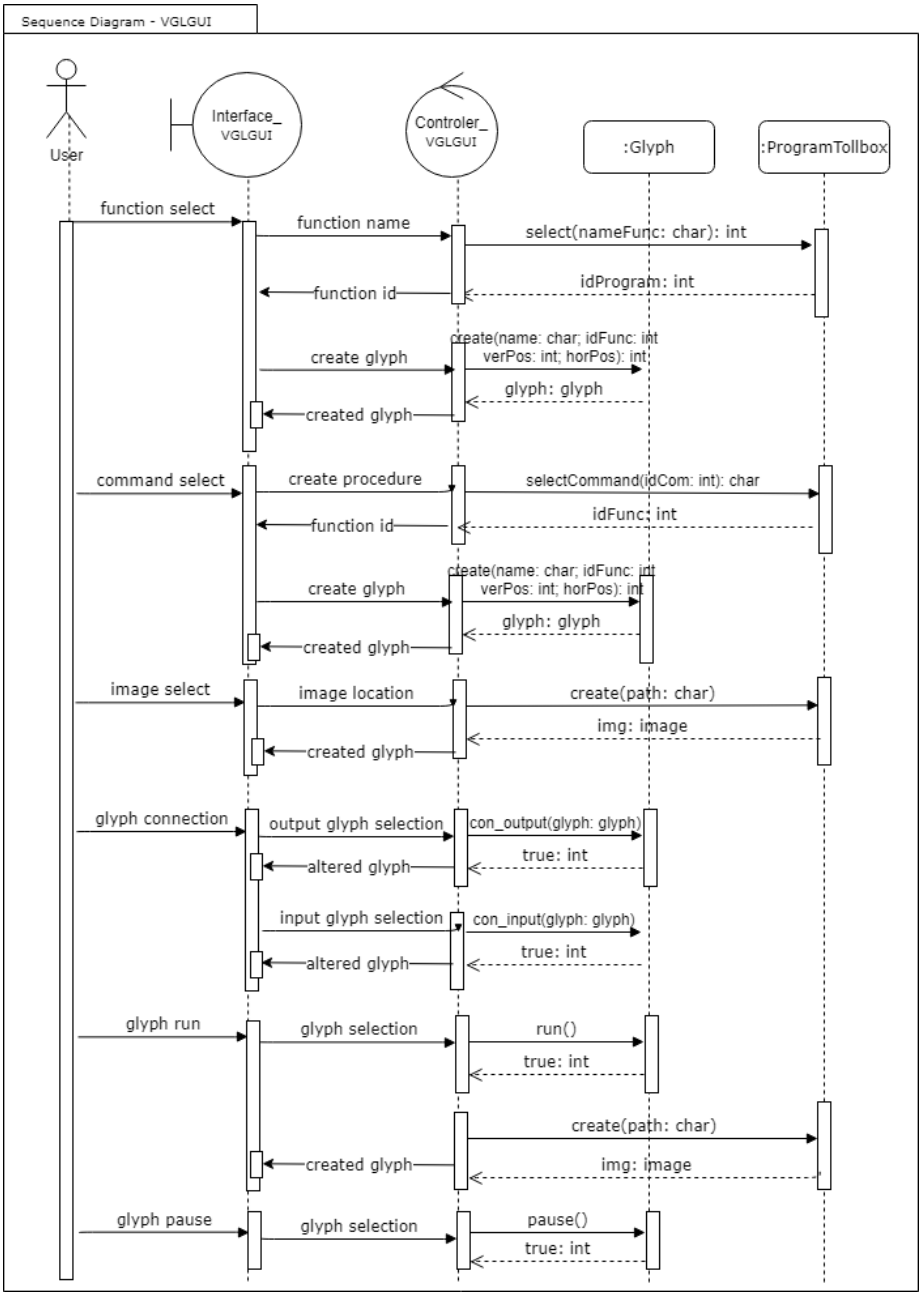


Figura 9 – Diagrama de sequência

workflow. Um *workflow* é formado por um conjunto de glifos conectados em um formato de rede para representar o fluxo de dados a ser executado. O *workflow* é persistido em um arquivo de disco ou em um SGBD.

A GUI possui funções para criar, editar, copiar e excluir um arquivo de *workflow*. Métodos são implementados para execução, pausa, reinicialização da execução e limpeza completa do conteúdo do espaço de trabalho.

Inicialmente, o usuário seleciona a função VisionGL listada por `ProgramToolbox` que executa a primeira etapa de seu programa. Uma representação gráfica desse operador é criada na GUI em formato de glifo. É possível definir parâmetros para o operador realizar sua ação.

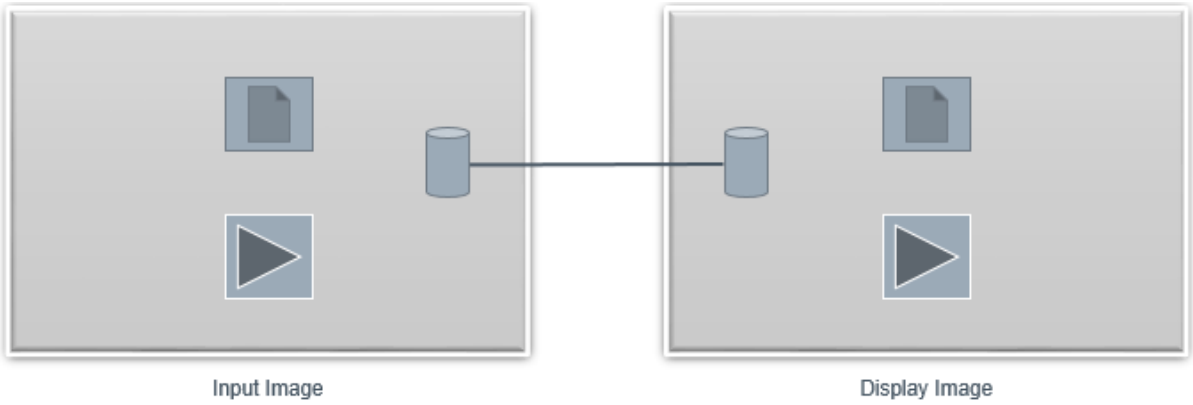


Figura 10 – Conexões entre glifos

Para fazer isso, o usuário deve clicar no botão *Argument*, e uma caixa de diálogo se abre para a edição do parâmetro. Em seguida, o usuário seleciona em `ProgramToolBox` os comandos de controle de fluxo necessários para montar a lógica de programação. Uma representação gráfica do comando é criada na GUI em formato de glifo.

O usuário indica o caminho da imagem para processamento. Um glifo que representa a imagem é criado na GUI sem a conexão de entrada porque apenas fornece informações apenas para outros glifos.

O desenvolvedor realiza sucessivamente a seleção e conexão dos glifos para montar sua lógica de programação. Para a conexão de dois glifos é necessário clicar na conexão de dados do primeiro glifo e depois clicar na conexão do segundo glifo, uma linha de conexão é estabelecida. O evento `glyph connection` é mostrado na Figura 9. A conexão entre os glifos é ilustrada na Figura 10.

A execução do operador é disparada clicando em seu botão *Run*. O evento `glyph run`, mostrado na Figura 9, retrata a execução de um único glifo ou um programa inteiro. É possível pausar a execução de um glifo. O evento `glyph pause` mostra na Figura 9 este comando. Uma mensagem de erro é exibida no console quando há um problema com a execução de um operador.

No final da execução do *workflow*, as imagens resultantes podem ser salvas em arquivos.

4.1.5 Visão de desenvolvimento

A Figura 11 representa as 6 camadas necessárias para a operação da VGLGUI. As camadas 4, 5 e 6 são executadas no cliente e as camadas 1, 2 e 3 são executadas no servidor.

- Camada 6 — *Client*, comunica-se com a web para permitir a edição e execução do *workflow*.
- Camada 5 — *Framework*, fornece a funcionalidade da interface.

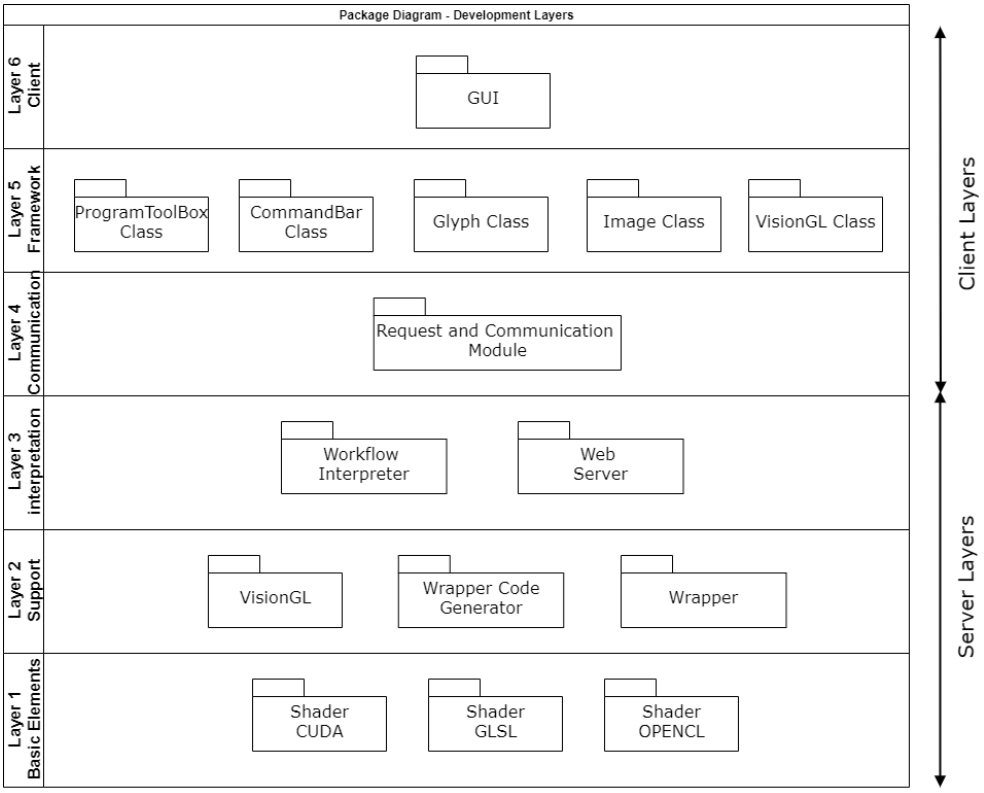


Figura 11 – Diagrama de pacotes - camadas desenvolvimento

- Camada 4 — *Communicator*, é responsável por solicitações e outras comunicações com o servidor.
- Camada 3 — *Interpretation*, possui um interpretador de *workflow* e um servidor web.
- Camada 2 — *Support*, fornece recursos para a criação de operadores.
- Camada 1 — *Basic Elements*, contém os operadores e *shaders* que executam o *workflow*.

A VGLGUI possui os componentes GUI, Glyph e VisionGL, ilustrados na Figura 12.

O componente GUI consiste em um espaço de trabalho, um menu principal, uma barra de comandos, uma caixa de ferramentas do programa e um console. O menu principal está localizado na parte superior da tela e agrupa funções para manipular arquivos, editar *workflows* e configurar a GUI. A barra de comandos aparece logo abaixo do Menu e oferece acesso fácil aos recursos usados com mais frequência. A caixa de ferramentas do programa possui uma coleção de *shaders*, funções para o programador usar na construção do *workflow* de processamento de imagens. O console exibe mensagens sobre o processamento do *workflow*.

Um glifo é uma representação visual de uma função disponível na VGLGUI para o desenvolvimento de *workflow* de processamento de imagem. Os componentes do glifo são o nome do operador; a conexão de dados de entrada, usada para a entrada de dados; conexão de dados de saída, usada para saída de dados; botão *Argument*, usado para especificar valores de

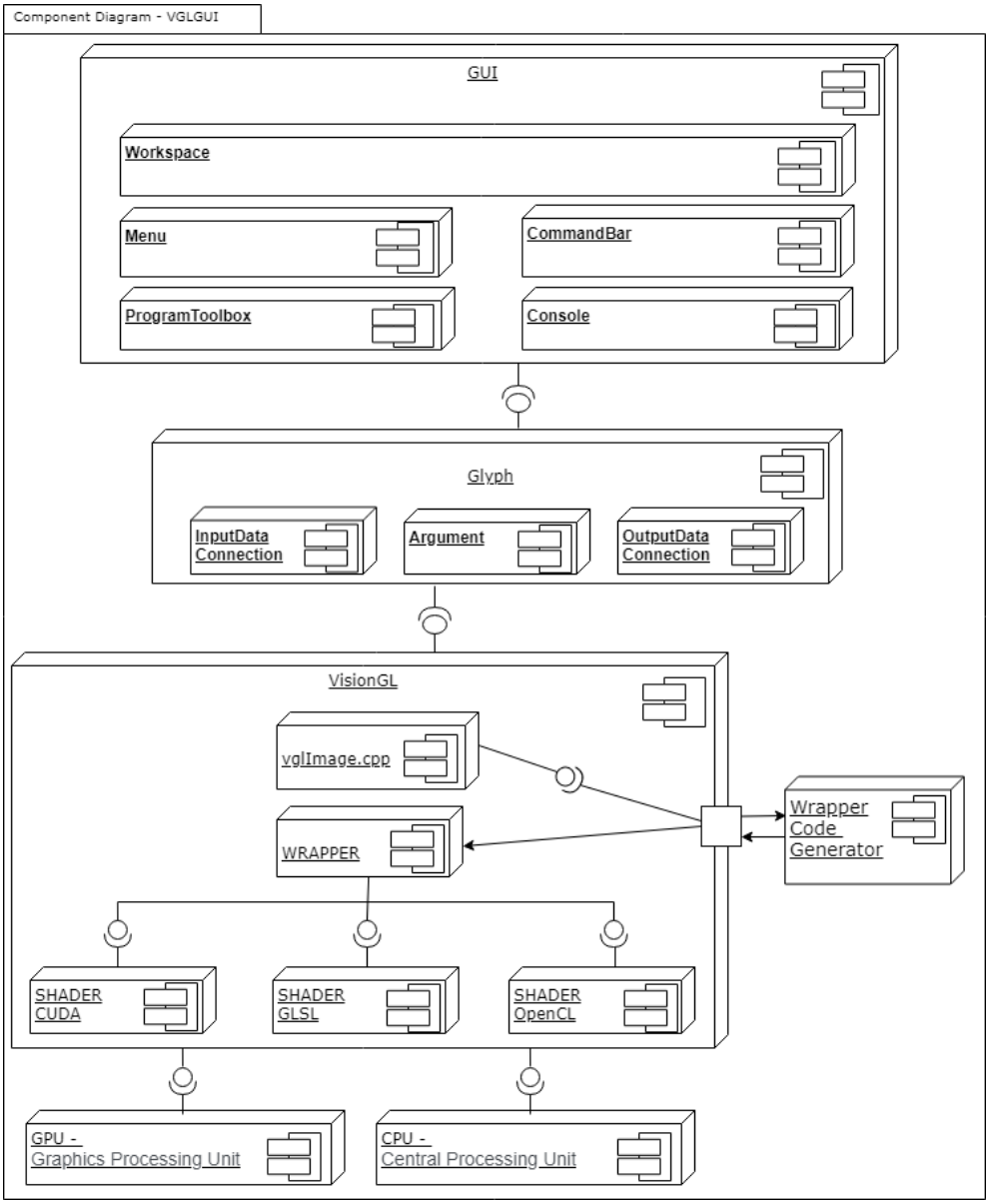


Figura 12 – Diagrama de componentes

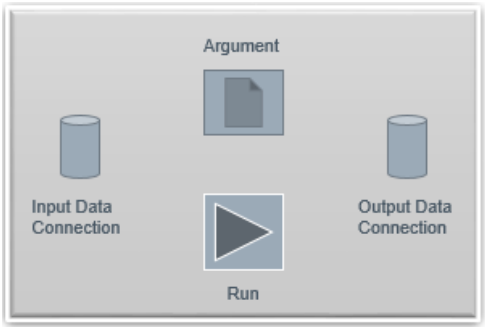


Figura 13 – Componentes do glifo

parâmetros para o operador; e o botão de *Run*, usado para executar a função. Os componentes do glifo são ilustrados na Figura 13.

A biblioteca VisionGL possui uma estrutura de dados chamada `vglImage` que possui ponteiros para as imagens armazenadas em cada contexto suportado, que são RAM, OpenCL, CUDA e GLSL (DANTAS; LEAL; SOUSA, 2015). As imagens são suportadas com duas, três e mais dimensões, com um, três e quatro canais (DANTAS; LEAL; SOUSA, 2016). A biblioteca possui um script que gera o código do *wrapper* com as chamadas a APIs necessárias antes de chamar o *shader*. Este código de *wrapper* é escrito em C++.

4.1.6 Visão física

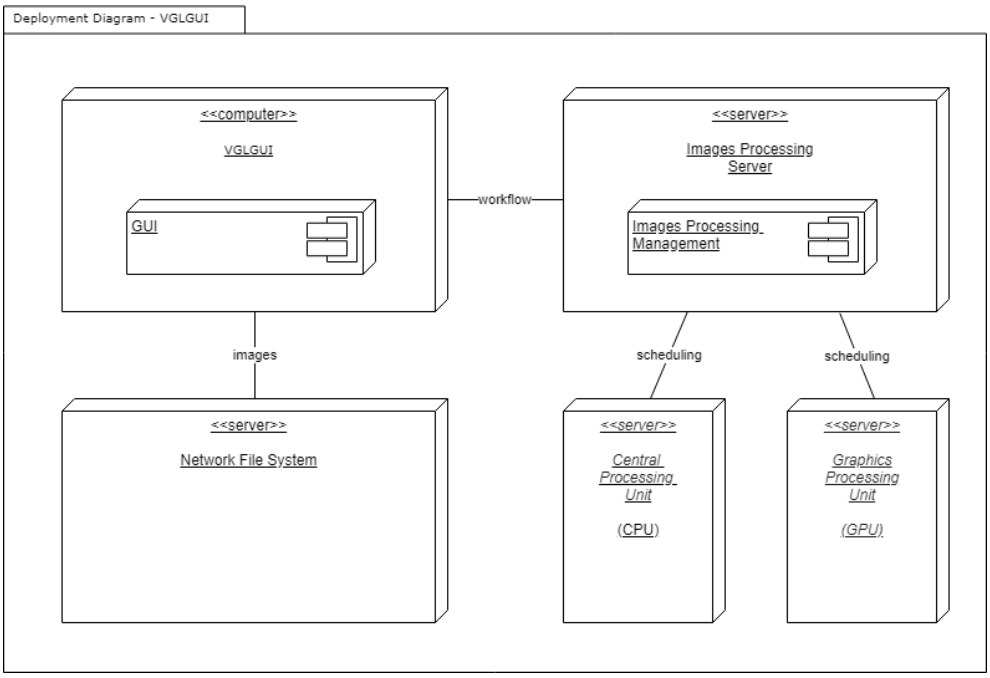


Figura 14 – Diagrama de implantação

A arquitetura física na qual a VGLGUI é implementada e executada em termos de tecnologias para conectar seus elementos e tráfego de dados é descrita nesta seção e ilustrada na Figura 14.

A biblioteca VisionGL suporta processamento paralelo em CPUs usando OpenCL e em GPUs usando OpenCL, GLSL e CUDA. Uma extensão para suportar processamento distribuído chamada VGLGUI envolve a criação de um cliente e editor de *workflow* e um servidor, capaz de executar esse *workflow*.

O código-fonte necessário na criação de novas funções de processamento de imagens é gerado automaticamente, tanto no cliente quanto no servidor. Para executar o *workflow*, o cliente envia uma lista de nomes de arquivos de entrada disponíveis em algum caminho de um *network file system* (NFS) para um servidor. Se o número de imagens for muito grande, o cliente aloca um subconjunto de imagens para cada servidor.

O agendamento é executado usando um algoritmo FCFS. Os servidores são configuráveis, podendo usar apenas a CPU ou GPU. Máquinas com GPU discreta (separada do processador) executam duas instâncias do servidor, uma associada à CPU e outra à GPU. Portanto, apenas um nível de controle de agendamento no cliente é necessário e o servidor não precisa controlar a comutação entre CPU e GPU.

O desenvolvimento é feito com a API WebGL, que é uma API JavaScript, disponível a partir do elemento `canvas` do HTML5, que oferece suporte para renderização de gráficos 2D e 3D, implementados em uma aplicação web sem a necessidade de *plug-ins* nos navegadores.

No Capítulo 5 será apresentada a arquitetura do interpretador de *workflow*, que foi o módulo escolhido para ser implementado no protótipo do sistema. Será detalhada a estrutura de classes que armazena os dados em execução, o modelo de funcionamento do interpretador e o padrão de arquivo de *workflow*.

5

Interpretador de *workflow* VGLGUI

5.1 Interpretador de *workflow*

Um interpretador é um programa que analisa o código-fonte em tempo real sem compilá-lo primeiro. Os elementos de um interpretador são o **leitor**, que transforma uma string de texto simples em uma sequência de *tokens*; o **analisador**, que pega uma sequência de tokens e produz uma **árvore de sintaxe abstrata** (AST) de uma linguagem e; o **avaliador**, que é um programa que executa o AST (Sakib Hadziavdic, 2020).

Um *workflow* consiste em um fluxo orquestrado de componentes de processamento de dados. Um *workflow* pode ser descrito como uma sequência de operações de trabalho que podem ser simples ou complexas (TRAISUWAN; TANDAYYA; LIMNA, 2015). *Workflow* significa que o trabalho é feito por diferentes componentes em uma sequência fixa (TONG; CHENG-DONG; DONG-YUE, 2011).

Este capítulo apresenta a descrição do protótipo do módulo VGLGUI Interpreter, implementado para execução em *localhost*. O protótipo contempla a leitura e execução de arquivos de *workflow* para processamento do *pipeline* de processamento de imagens. A execução ocorre por meio dos operadores da camada 1, Basic Elements, que são chamados pela camada 2, Support. Serão implementados no futuro a interface gráfica para edição e execução do *workflow* (camada 6, Client), o *framework* que fornece as funcionalidades da interface (camada 5, Framework), a comunicação com o servidor (camada 4, Communication), e o servidor web (camada 3, Interpretation). A Figura 11 ilustra as camadas descritas anteriormente.

5.2 Interpretador de *workflow* VGLGUI

O interpretador de *workflow* da interface VGLGUI é um módulo da camada 3, Interpretation, da visão de desenvolvimento descrita na Seção 4.1.5, como mostra a Figura 11. Possui uma

arquitetura baseada em glifos e conexões. As regras do relacionamento entre glifos e conexões são as seguintes:

1. Glifos correspondem a vértices na teoria dos grafos e representam uma função da biblioteca VisionGL.
2. As arestas são conexões entre glifos e representam a imagem a ser processada.
3. Cada conexão possui uma imagem armazenada.
4. Um glifo do tipo fonte carrega ou cria uma imagem e só pode ter conexões de saída.
5. Um glifo do tipo sumidouro mostra ou salva uma imagem e só pode ter conexões de entrada.
6. As conexões cujos glifos de origem já foram executados, e que portanto já tiveram suas imagens de entrada geradas, têm o status READY = TRUE, o que significa que a imagem está pronta para ser processada.
7. Os glifos possuem um status READY que informa se ele está pronto para executar, e um status DONE que informa se já foi executado. Ambos os status começam como FALSE.
8. Glifos têm uma lista de conexões de entrada. Quando todas as conexões de entrada têm o status READY = TRUE, o glifo muda seu status para READY = TRUE.
9. Glifos cujo status é READY = TRUE são executados.
10. O status do glifo torna-se DONE = TRUE após sua execução.
11. Glifos do tipo fonte são criados com o status READY = TRUE.

5.3 Funcionamento do interpretador de *workflow* VGLGUI

Domain-specific languages (DSLs) são adaptadas a domínios específicos de engenharia e oferecem facilidade de uso em comparação com as linguagens de programação gerais. As DSLs podem fornecer níveis mais altos de abstração em ambientes próximos ao domínio do especialista, ajudando cientistas e analistas a lidarem com códigos e dados complexos (SHEN et al., 2020).

O Código 1 mostra os dois tipos de linhas de comando usadas para criar glifos e conexões, com base no formato de arquivo de *workflow* Khoros. Podemos dizer que este formato de arquivo é uma espécie de DSL porque sua sintaxe foi adaptada para aplicação exclusiva em processamento de imagem através de glifos e conexões.

O programador usa a VGLGUI para desenvolver seu *pipeline* de processamento de imagens, e a interface gera o arquivo de *workflow* correspondente em disco com a extensão `wksp`. O código fonte do *workflow* Demo está listado no Apêndice A e o do código fonte do *workflow* Fundus está listado no Apêndice B.

Código 1 – Linhas de comando do arquivo *workflow* VGLGUI

```
1  #  Glyph '[GlyphName]'
2  Glyph:[Library]:comment::localhost:[Glyph_ID]:[Glyph_X]:[Glyph_Y]::
       ↪ -[var_str] '[var_str_value]'
3  Glyph:[Library]:comment::localhost:[Glyph_ID]:[Glyph_X]:[Glyph_Y]::
       ↪ -[var_num] [var_num_value]
4
5  #  Connections '[GlyphName]'
6  NodeConnection:data:[output_Glyph_ID]:[output_varname]:
       ↪ [input_Glyph_ID]:[input_varname]
```

Para editar o *workflow* de processamento de imagens no VGLGUI, o usuário escolhe a opção *File - Open workflow file* no menu, indica o caminho do arquivo e clica em *Open*. O VGLGUI Interpreter lê as linhas de comando do tipo `Glyph`, identifica as informações sobre a posição da tela, função a ser executada, parâmetros de execução e cria a representação gráfica do glifo no espaço de trabalho VGLGUI.

Em seguida, o interpretador lê as linhas de comando do tipo `NodeConnection`, cria a estrutura que armazena os nós e arestas do *workflow* na memória e desenha a representação gráfica das conexões no espaço de trabalho VGLGUI.

Cada entrada está ligada a uma única saída de outro glifo, seu predecessor imediato na sequência de processamento da imagem. Cada saída pode ser conectada a mais de uma entrada de outros glifos.

O VGLGUI Interpreter gera uma AST e armazena as informações dos glifos e suas conexões na memória. A Figura 15 mostra as classes usadas na representação interna do *workflow* na memória. O VGLGUI Interpreter foi desenvolvido em Python. O usuário edita o *workflow* graficamente e, ao clicar no menu *File - Write file workflow*, as alterações são salvas no disco. O tratamento de erros é realizado durante a leitura e execução do arquivo de *workflow* para evitar falhas básicas do programa. As mensagens de erro mostram a linha do *workflow* com o erro e a regra não atendida.

No Capítulo 6 será demonstrado o uso de dois *workflows* com VGLGUI: Demo e Fundus.

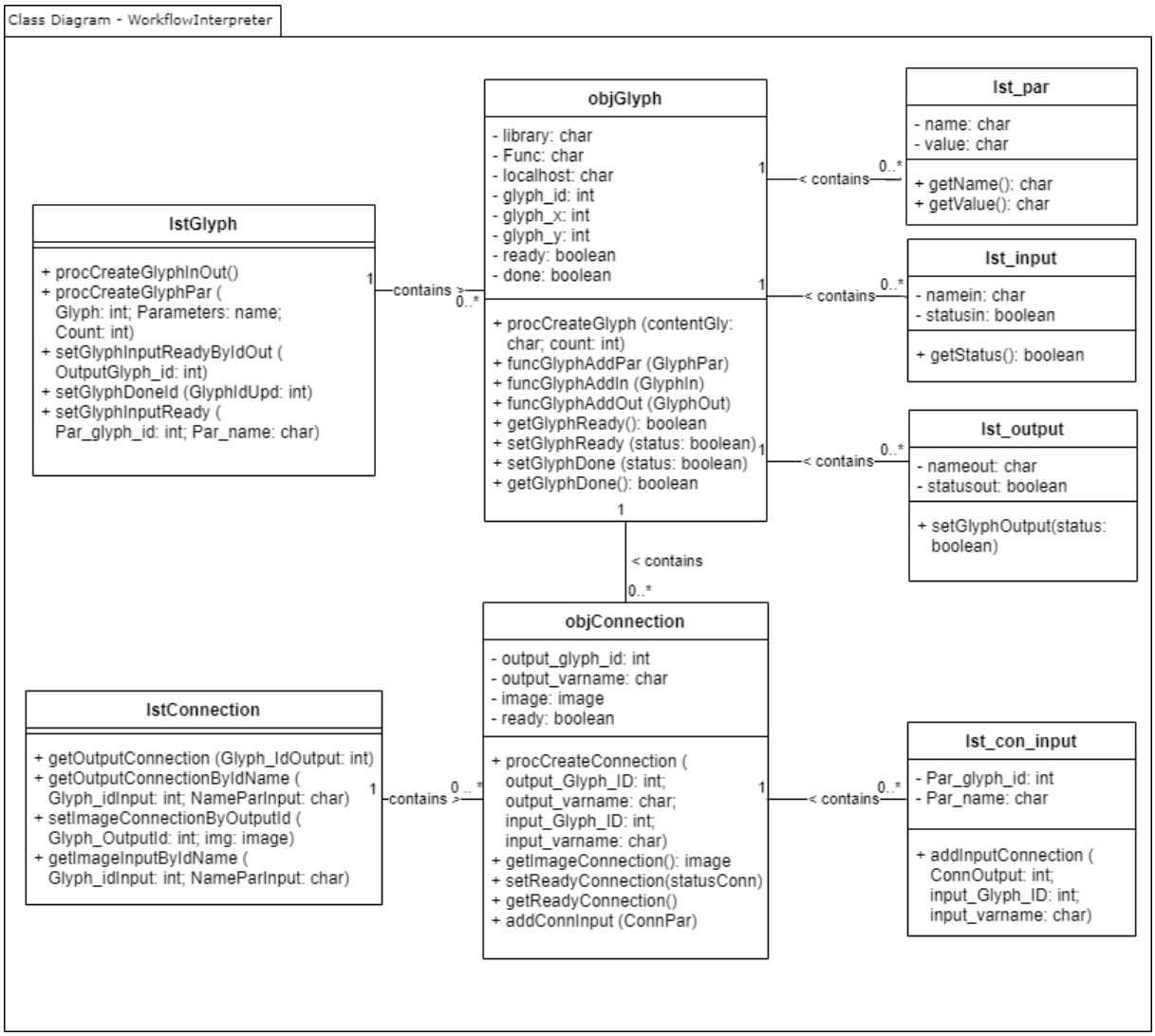


Figura 15 – Diagrama classe: VGLGUI interpreter.

6

Resultados e discussão

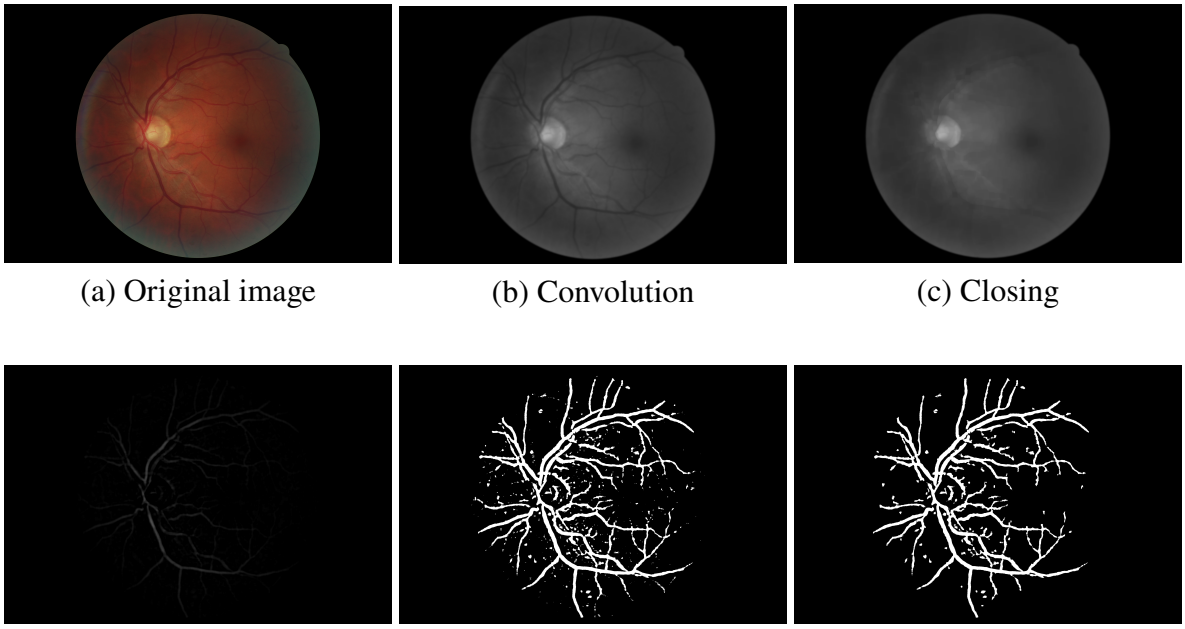
Neste estudo, demonstramos o uso de dois *workflows* com VGLGUI: Demo e Fundus. O *workflow* Demo mostra um uso básico do interpretador. O *workflow* Fundus mostra um cenário da vida real, segmentando uma imagem de retinografia do conjunto de dados HRF (BUDAI et al., 2013). Ambos os *workflows* foram executados em um computador desktop com CPU Intel Core i3-4130 3,40 GHz com 12 GB de RAM e GPU NVIDIA GeForce GTX 1070 com 8 GB de RAM. Os *workflows* usaram operações básicas, como carregamento de imagem, salvamento de imagem, convolução e operações de morfologia matemática, como dilatação e erosão. O *workflow* Fundus também usa operações de subtração, fechamento e reconstrução.

O interpretador da VGLGUI foi testado com ambos os *workflows* em execução na CPU e na GPU. Ambos os *workflows* também foram portados para Python com OpenCV executado na CPU e VisionGL com C++ executado na CPU e GPU. Os experimentos foram executados 10 ou 1000 vezes cada, respeitando o desempenho viável para apurar a média de tempo de execução.

6.1 *Workflow* Demo

No *workflow* Demo, o objetivo é demonstrar o uso de funções básicas, como convolução, dilatação e erosão. A figura 17 é a representação visual do *workflow* Demo. O Apêndice A mostra a listagem do arquivo de *workflow* Demo.

O Glyph 1 carrega a imagem a ser processada. O Glyph 2 aloca uma imagem com o mesmo tamanho da imagem de entrada exigida por `vglClRgb2Gray`. O Glyph 3 aplica a função `vglClRgb2Gray` para converter a imagem de entrada em tons de cinza. A função VisionGL `vglClConvolution` aplica um desfoque gaussiano para reduzir o ruído. O Glyph 10 salva o resultado da convolução em um arquivo. O Glyph 7 aplica a função `vglCLDilate` para aumentar as áreas claras. O Glyph 8 aloca uma imagem do mesmo tamanho que a imagem de entrada exigida por `vglClErode`. O Glyph 9 executa o `vglClErode` para erodir áreas claras e eliminar



(a) Original image (b) Convolution (c) Closing

(d) Subtraction (e) Threshold (f) Reconstruction

Figura 16 – Resultados do *workflow* de processamento de imagens Fundus.

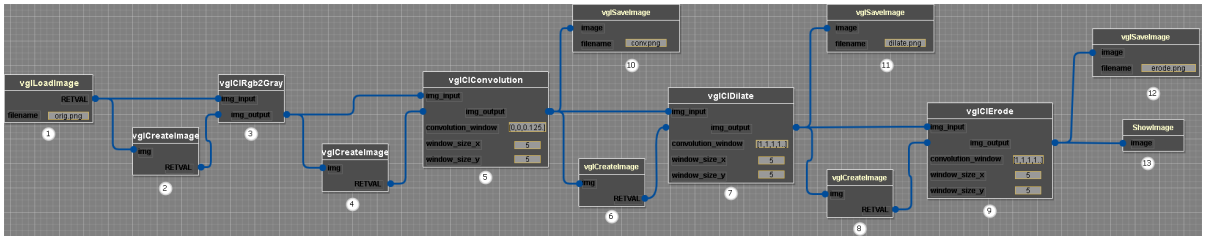


Figura 17 – Representação visual do *workflow* Demo.

Tabela 2 – Tempo médio de execução do *workflow* Demo, em milissegundos, em uma imagem com 5184×3456 pixels

Operations	Python CPU	Interpreter CPU	Interpreter GPU	VisionGL CPU	VisionGL GPU
Rgb2Gray	4.89	296.95	1.38	1.61	1.36
Convolution 51×51	5.80	4269.16	4.12	4.64	3.87
Dilation 51×51	10.29	4979.74	3.87	4.42	3.70
Erosion 51×51	6.67	5021.59	3.89	4.37	3.55
TOTAL	27.65	14567.44	13.26	15.04	12.48

áreas brilhantes menores que o elemento estruturante. O Glyph 12 salva o resultado da erosão em um arquivo. O Glyph 13 mostra a imagem final resultante na tela.

A Tabela 2 mostra os tempos de execução do *workflow* Demo em milissegundos.

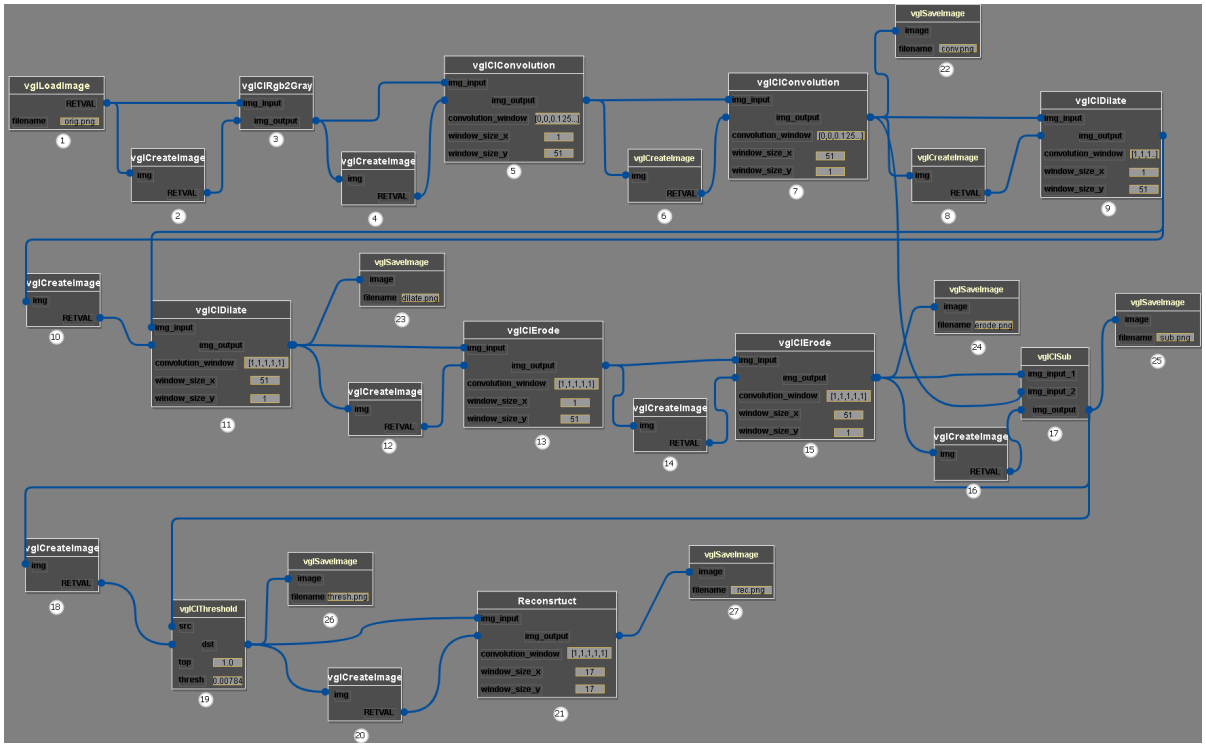


Figura 18 – Representação visual do *workflow* Fundus.

6.2 *Workflow* Fundus

No *workflow* Fundus, o objetivo é a aplicação de um *pipeline* para segmentação de vasos sanguíneos em imagens da retina (DANTAS; OLIVEIRA; LEAL, 2017). As etapas gerais do *pipeline* são: convolução (desfoque gaussiano), fechamento seguido de subtração (*black-hat*), limiarização e abertura por reconstrução. Os filtros e elementos estruturantes foram separados em um filtro de coluna e um filtro de linha. Os resultados são idênticos, mas a complexidade de tempo é muito menor.

A Figura 16 mostra as imagens resultantes do *workflow* Fundus de processamento de imagens. A figura 18 é a representação visual do *workflow* Fundus. A Tabela 3 mostra os tempos de processamento em milissegundos. O Apêndice B mostra a listagem do *workflow* Fundus.

6.3 Discussão

O interpretador de *workflow*, cuja estrutura foi detalhada neste estudo, é uma camada da arquitetura VGLGUI que une as funções do VisionGL e a interface gráfica para facilitar o aprendizado da programação do *pipeline* de processamento de imagens. VGLGUI usa OpenCL para paralelizar o processamento na GPU.

Na VGLGUI, o desenvolvedor elabora seu *pipeline* de processamento de imagens sem a necessidade de código. As funções são escolhidas a partir de um menu gerado com as funções

Tabela 3 – Tempo médio de execução *workflow* Fundus, em milessegundos, em uma imagem com 5184×3456 pixels

Operations	Python CPU	Interpreter CPU	Interpreter GPU	VisionGL CPU	VisionGL GPU
Rgb2Gray	7.55	301.89	1.36	269.87	1.33
Convolution 51×51	708.22	17861.99	21.98	18109.69	20.55
Closing 51×51	71.16	38960.01	43.26	25830.05	38.76
Subtraction	5.11	475.56	1.03	455.64	1.03
Threshold	45.86	475.56	1.03	270.66	0.71
Reconstruction 17×17	1107.72	47952.16	32.09	88195.03	59.12
TOTAL	1945.62	106027.17	100.75	133130.94	121.50

disponíveis na VisionGL. O usuário pode criar suas próprias funções em OpenCL e a biblioteca irá gerar automaticamente suas funções *wrapper* e entradas de menu. Esse recurso diminui a curva de aprendizado de programadores novatos e acelera o desenvolvimento.

Dois *workflows* foram criados para demonstrar como o VGLGUI Interpreter funciona. O *workflow* Demo demonstra o uso de funções básicas. O *workflow* Fundus demonstra o uso em um cenário da vida real, segmentando uma imagem de retinografia. Ambos os *workflows* foram testados com uma imagem com cerca de 18 megapixels.

O tempo médio de processamento do *workflow* Demo pelo interpretador em execução na GPU foi de 13,26 ms e pela versão Python em execução na CPU foi de 27,65 ms. Isso significa que o interpretador pode executar um *workflow* simples pelo menos duas vezes mais rápido que o Python. A ocupação da GPU foi baixa neste *workflow*, o que significa que as chamadas Python feitas pelo interpretador e que são executadas na CPU causam um gargalo quando as operações da GPU são muito rápidas.

Por outro lado, o tempo médio de processamento do interpretador para o *workflow* Fundus em execução na GPU foi de 100,75 ms, e pela versão Python em execução na CPU foi de 1945,62 ms. O *workflow* Fundus é executado na GPU cerca de 20 vezes mais rápido que sua versão Python. Como as operações são mais complexas, usando janelas maiores do que no *workflow* Demo, a ocupação da GPU é maior e o gargalo da CPU prejudica menos a velocidade de processamento. As transferências frequentes de dados entre a RAM e a memória GPU podem causar um gargalo. Como a imagem é transferida apenas uma vez da RAM para a memória da GPU e na GPU permanece durante todo o processamento do *workflow*, não há latência associada a isso. O tempo médio de processamento do Interpretador para o *workflow* Fundus em execução na GPU foi de 100.75 ms, 17,1% mais rápido que a execução da biblioteca VisionGL na GPU, que totalizou 121.50 ms.

O tempo médio de processamento do Interpretador para o *workflow* Fundus em execução na CPU foi de 106027.17 ms, 20,4% mais rápido que a execução da VisionGL na CPU, que

totalizou 133130,94 ms. Foi necessário reduzir o número de passos na execução da Visiongl e do Interpretador em CPU de 1000 para 10 vezes devido à lentidão ao executar as funções, mesmo com a biblioteca fazendo o paralelismo utilizando todos núcleos da CPU.

7

Conclusão

Imagens médicas são uma das ferramentas de diagnóstico médico mais básicas e comuns. Para olhos treinados, elas podem descrever com precisão os órgãos internos de um ser humano e indicar a presença de patologias. O primeiro passo em qualquer interpretação de imagens médicas é a segmentação da imagem. O olho treinado pode segmentar e analisar a imagem no nível cognitivo. Em contraste, os computadores precisam de algoritmos específicos para esta tarefa. O tamanho das coleções de imagens aumentou drasticamente e atingiu petabytes de dados. Esses volumes não podem ser processados por um computador dentro de um tempo razoável. Portanto, as tarefas contemporâneas de processamento de imagens requerem paralelismo.

No campo da saúde, o processamento de imagens médicas por meio da computação em nuvem permite que os profissionais de saúde processem e analisem imagens usando recursos remotos da nuvem. A computação em nuvem fornece software como serviço, que permite importantes reduções de custos associados à exploração do *data center* local e contratação de equipe de TI.

Vários são os desafios na criação de sistemas de processamento de imagens médicas, como o tamanho das imagens e a necessidade de integração com outras soluções no ambiente clínico e hospitalar. VisionGL é uma biblioteca de código aberto que facilita a programação por meio da geração automática de código *wrapper* C++. O código *wrapper* é responsável por chamar funções de processamento paralelo de imagens ou *shaders* em CPUs usando OpenCL e em GPUs usando OpenCL, GLSL e CUDA.

VGLGUI é uma interface gráfica de usuário para processamento de imagem que permitirá a programação visual de *workflow* para processamento paralelo de imagens, por meio de funções VisionGL para geração automática de código *wrapper* e otimização de transferências de imagem entre RAM e GPU.

Este estudo apresenta a descrição da arquitetura da VGLGUI e do protótipo do módulo VGLGUI Interpreter implementado para execução em *localhost*. O protótipo contempla a leitura

e execução de arquivos de *workflow* para processamento de *pipeline* de imagens. A execução ocorre por meio dos operadores da camada 1, Basic Elements, que são chamados pelos *wrappers* da camada 2, Support.

Foram criados dois *workflows* para demonstrar as funcionalidades do interpretador: Demo e Fundus. Cada *workflow* foi implementado e executado em duas plataformas diferentes: Python executando na CPU e VGLGUI Interpreter executando na GPU. O *speedup* varia entre cerca de dois e 20. Essa variação se deve ao gargalo da CPU, que é relativamente mais estreito quando as funções que rodam na GPU são muito rápidas. Quando as operações executadas na GPU são mais exigentes, por exemplo, com grandes *kernels* e elementos estruturantes, a ocupação da GPU aumenta e a aceleração é maior.

O sistema proposto também pode facilitar a implementação de *pipelines* de processamento de imagens, fornecendo um editor de *workflow* visual. A programação visual pode ser mais fácil e intuitiva de usar do que as linguagens de programação padrão e também pode acelerar o desenvolvimento, fornecendo um ambiente de teste pronto para uso em experimentos de processamento de imagens.

Como trabalhos futuros serão implementados a interface gráfica para edição e execução do *workflow* (camada 6, Client), o *framework*, que expõe as funcionalidades da interface (camada 5, Framework), a comunicação com o servidor (camada 4, Communication), e o servidor web (camada 3, Interpretation). Versões futuras da VGLGUI podem incluir geração automática de código do interpretador, reconhecimento de padrões, novas funções de processamento de imagens, seja aumentando a VisionGL ou expondo funções de processamento de imagens do OpenCV. Também estamos considerando a implementação de um *scheduler* para distribuir a carga entre os servidores.

Referências

ALHAZMI, S.; THEVATHAYAN, C.; HAMILTON, M. Interactive pedagogical agents for learning sequence diagrams. In: *Artificial Intelligence in Education*. [S.l.]: Springer, 2020. p. 10–14. Citado na página 23.

ALVIN, C.; PETERSON, B.; MUKHOPADHYAY, S. Static generation of UML sequence diagrams. *International Journal on Software Tools for Technology Transfer*, Springer, v. 23, p. 31–53, 2019. Citado na página 23.

BLATTNER, T. et al. A hybrid task graph scheduler for high performance image processing workflows. In: *2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. [S.l.]: IEEE, 2015. p. 634–637. Citado 2 vezes nas páginas 34 e 35.

BOYER, M.; SKADRON, K.; WEIMER, W. Automated dynamic analysis of cuda programs. In: *In Proceedings of STMCS*. [S.l.: s.n.], 2008. Citado na página 38.

BRADSKI, G.; KAEHLER, A. Learning OpenCV: Computer vision with the OpenCV library. In: *O Reilly*. [S.l.]: Cambridge, 2008. Citado na página 33.

BUCHMANN, T.; DOTOR, A.; WESTFECHTEL, B. Model-driven software engineering: concepts and tools for modeling-in-the-large with package diagrams. *Computer Science – Research and Development*, Springer, v. 29, n. 1, p. 73–93, 2014. ISSN 1865-2034. Citado na página 22.

BUDAI, A. et al. *Robust Vessel Segmentation in Fundus Images*. 2013. Disponível em: <https://www5.cs.fau.de/research/data/fundus-images/>. Citado na página 58.

CALI, A. et al. Querying UML class diagrams. In: *International Conference on Foundations of Software Science and Computational Structures FoSSaCS*. [S.l.]: Springer, 2012. p. 1–25. Citado na página 22.

CAO, H. et al. ImageFlow: Workflow based image processing with legacy program in grid. In: *2009 Second International Conference on Future Information Technology and Management Engineering*. [S.l.]: IEEE, 2009. p. 115–118. Citado na página 37.

DANTAS, D. O.; BARRERA, J. Automatic generation of wrapper code for video processing functions. In: *Learning and Nonlinear Models (LNLM) – Journal of the Brazilian Neural Network Society*. [S.l.]: Sociedade Brasileira de Redes Neurais (SBRN), 2011. p. 130–137. Citado 2 vezes nas páginas 37 e 38.

DANTAS, D. O.; LEAL, H. D. P.; SOUSA, D. O. B. Fast 2D and 3D image processing with OpenCL. In: *International Conference on Image Processing ICIP*. [S.l.]: IEEE, 2015. p. 4858–4862. Citado 3 vezes nas páginas 33, 34 e 52.

DANTAS, D. O.; LEAL, H. D. P.; SOUSA, D. O. B. Fast multidimensional image processing with OpenCL. In: *International Conference on Image Processing ICIP*. [S.l.]: IEEE, 2016. p. 1779–1783. Citado 2 vezes nas páginas 38 e 52.

DANTAS, D. O.; OLIVEIRA, D. D. S.; LEAL, H. D. P. Blood vessels extraction using fuzzy mathematical morphology. In: *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. [S.l.]: IEEE, 2017. p. 914–918. Citado 2 vezes nas páginas 38 e 60.

DICOM. 2008. Https://pt.wikipedia.org/wiki/DICOM/. Citado na página 16.

DORONICHEVA, A. V.; SAVIN, S. Z. Web-technology for medical images segmentation. *3rd Russian-Pacific Conference on Computer Technology and Applications (RPC)*, IEEE, https://ieeexplore-ieee-org.ez20.periodicos.capes.gov.br/document/8482234/, p. 1–5, 2018. Citado 2 vezes nas páginas 14 e 16.

DRYMONITIS, A. Introduction to PureData. In: *Digital Electronics for Musicians*. [S.l.]: Apress, 2015. p. 1–50. ISBN 978-1-4842-1584-5. Citado na página 31.

EDER, J. et al. Bringing DSE to life: exploring the design space of an industrial automotive use case. In: *20th International Conference on Model Driven Engineering Languages and Systems MODELS*. [S.l.]: IEEE, 2017. p. 270–280. Citado na página 25.

FAHRENBERG, U. et al. Sound merging and differencing for class diagrams. In: *International Conference on Fundamental Approaches to Software Engineering FASE*. [S.l.]: Springer, 2014. p. 63–78. Citado na página 22.

FRANCA, J. M. S.; LIMA, J. de S.; SOARES, M. S. Development of an Electronic Health Record Application using a Multiple View Service Oriented Architecture. In: INSTICC. *Proceedings of the 19th International Conference on Enterprise Information Systems - Volume 2, ICEIS*. [S.l.]: SciTePress, 2017. p. 308–315. ISBN 978-989-758-302-5. Citado na página 24.

GAL, N.; STOICU-TIVADAR, V. Simulation of medical image interpretation. In: *2011 15th IEEE International Conference on Intelligent Engineering Systems*. [S.l.: s.n.], 2011. p. 33–37. Citado na página 14.

GERADTS, Z.; BIJHOLD, J. Forensic video investigation with real-time digitized uncompressed video image sequences. In: *Investigation and Forensic Science Technologies*. [S.l.]: SPIE, 1999. p. 154–164. Citado 2 vezes nas páginas 7 e 29.

GUEDES, G. T. A. *UML 2: uma abordagem prática*. [S.l.]: Novatec Editora, 2011. ISBN 978-85-7522-281-2. Citado 3 vezes nas páginas 22, 23 e 24.

GUREVICH, I. B. et al. An open general-purposes research system for automating the development and application of information technologies in the area of image processing, analysis and evaluation. *Pattern Recognition and Image Analysis*, Springer, v. 16, n. 4, p. 530–563, 2006. ISSN 1054-6618. Citado na página 30.

HAITZER, T.; ZDUN, U. Controlled experiment on the supportive effect of architectural component diagrams for design understanding of novice architects. In: *European Conference on Software Architecture ECSA*. [S.l.]: Springer, 2013. p. 54–71. Citado na página 23.

HALFHIL, T. R. Parallel processing with CUDA. In: *Microprocessor Report*. [S.l.: s.n.], 2008. Citado na página 38.

HAMIDIAN, H. et al. Adapting medical image processing tasks to a scalable scientific workflow system. In: *2014 IEEE World Congress on Services*. [S.l.]: IEEE, 2014. p. 385–392. Citado na página 35.

HARPER, K. E.; ZHENG, J. Exploring software architecture context. In: *12th Working IEEE/IFIP Conference on Software Architecture WICSA*. [S.l.]: IEEE, 2015. p. 123–126. Citado na página 19.

HILLIARD, R. et al. On the composition and reuse of viewpoints across architecture frameworks. In: *Joint Working Conference on Software Architecture & 6th European Conference on Software Architecture*. [S.l.]: IEEE, 2012. p. 131–140. Citado na página 24.

HOLY, L.; BRADA, P. Viewport for component diagrams. In: *International Symposium on Graph Drawing GD*. [S.l.]: Springer, 2011. p. 443–444. Citado na página 23.

INGALSBE, J. A. Supporting the building and analysis of an infrastructure portfolio using UML deployment diagrams. In: *International Conference on the Unified Modeling Language UML*. [S.l.]: Springer, 2005. p. 105–117. Citado na página 24.

ISO. *42010-2011 - ISO/IEC/IEEE Systems and software engineering – Architecture description*. 2011. Disponível em: <https://ieeexplore.ieee.org/servlet/opac?punumber=6129465>. Citado 5 vezes nas páginas 17, 18, 19, 20 e 44.

JOHNSTON, W. M.; HANNA, J. R. P.; MILLAR, R. J. Advances in dataflow programming languages. *ACM Computing Surveys*, ACM, v. 36, n. 1, 2004. Citado na página 29.

KAEWKEEREE, S.; TANDAYYA, P. Enhancing the Taverna workflow system for executing and analyzing the performance of image processing algorithms. In: *2012 Ninth International Conference on Computer Science and Software Engineering (JCSSE)*. [S.l.]: IEEE, 2012. p. 328–333. Citado 2 vezes nas páginas 35 e 36.

KRUCHTEN, P. Architectural blueprints: The "4+1" View Model of Software Architecture. *IEEE Software*, IEEE, v. 12, n. 6, p. 42–50, 1995. Citado 4 vezes nas páginas 24, 25, 26 e 27.

LI, B. et al. Rapid prototyping of image processing workflows on massively parallel architectures. In: *Proceedings of the 10th International Workshop on Intelligent Solutions in Embedded Systems*. [S.l.]: IEEE, 2012. p. 15–20. Citado 2 vezes nas páginas 38 e 39.

MACIEL, R.; SOARES, M.; DANTAS, D. A system architecture in multiple views for an image processing graphical user interface. In: *In Proceedings of the 23rd International Conference on Enterprise Information Systems (ICEIS 2021)*. [S.l.]: SCITEPRESS, 2021. p. 213–223. Citado 2 vezes nas páginas 17 e 42.

MARWAN, M.; KARTIT, A.; OUAHMANE, H. Using cloud solution for medical image processing: Issues and implementation efforts. In: *3rd International Conference of Cloud Computing Technologies and Applications CloudTech*. [S.l.]: IEEE, 2017. p. 1–7. Citado 7 vezes nas páginas 14, 16, 17, 39, 40, 41 e 42.

MILLETARI, F. et al. Cloud deployment of high-resolution medical image analysis with TOMAAT. *Journal of Biomedical and Health Informatics*, IEEE, v. 23, n. 3, p. 969–977, 2019. ISSN 2168-2194. Citado 3 vezes nas páginas 15, 41 e 42.

MIRARAB, A.; FARD, N. G.; SHAMSI, M. A cloud solution for medical image processing. In: *Journal of Engineering Research and Applications*. [S.l.]: Ijera, 2014. p. 74–82. Citado na página 41.

MONTEIRO, E. J. M.; SILVA, L. A. B.; COSTA, C. CloudMed: Promoting telemedicine processes over the cloud. In: *7th Iberian Conference on Information Systems and Technologies*. [S.l.]: IEEE, 2012. p. 1–6. Citado 5 vezes nas páginas 15, 39, 40, 41 e 42.

MORITA, M. et al. Biomedical image communication platform. In: *First International Symposium on Computing and Networking*. [S.l.]: IEEE, 2013. p. 281–287. Citado 2 vezes nas páginas 15 e 17.

Object Management Group. *OMG Unified Modeling LanguageTM (OMG UML), Infrastructure*. 2011. Disponível em: <http://www.omg.org/spec/UML/2.4.1/Infrastructure>. Citado na página 21.

OPENCV. 2022. https://opencv.org/about/. Citado na página 32.

OWENS, J. D. et al. A survey of general-purpose computation on graphics hardware. In: *In Eurographics 2005, State of the Art Reports*. [S.l.: s.n.], 2005. p. 21–51. Citado na página 37.

PARK, J. H. et al. C2A: Crowd consensus analytics for virtual colonoscopy. *2016 IEEE Conference on Visual Analytics Science and Technology (VAST)*, IEEE, 2016. Citado na página 14.

PUREDATA. 2022. http://puredata.info/. Citado na página 32.

PUREDATA development environment. 2022. https://www.soundonsound.com/techniques/pure-data-introduction. Citado na página 32.

QUEIROS, S. et al. MITT: Medical Image Tracking Toolbox. *IEEE Transactions on Medical Imaging*, IEEE, https://ieeexplore.ieee.org/document/8365811, v. 37, n. 11, p. 2547–2557, 2018. Citado na página 14.

REAS, C.; FRY, B. Processing: A learning environment for creating interactive web graphics. In: *SIGGRAPH Web Graphics*. [S.l.]: ACM, 2003. Citado 3 vezes nas páginas 7, 30 e 31.

REAS, C.; FRY, B. Processing.org: Programming for artists and designers. In: *SIGGRAPH Web Graphics*. [S.l.]: ACM, 2004. Citado 2 vezes nas páginas 30 e 31.

REINEKE, J.; TRIPAKIS, S. Basic problems in multi-view modeling. *Tools and Algorithms for the Construction and Analysis of Systems TACAS*, Springer, v. 18, n. 3, p. 1577–1611, 2014. ISSN 1619-1366. Citado na página 24.

RIBEIRO, Q. A. D. S.; RIBEIRO, F. G. C.; SOARES, M. S. A technique to architect real-time embedded systems with SysML and UML through multiple views. In: INSTICC. *Proceedings of the 19th International Conference on Enterprise Information Systems - Volume 2, ICEIS*. [S.l.]: SciTePress, 2017. p. 287–294. ISBN 978-989-758-302-5. Citado na página 24.

Sakib Hadziavdic. *How to Approach Writing an Interpreter From Scratch*. 2020. Disponível em: <https://www.toptal.com/scala/writing-an-interpreter>. Citado na página 54.

SELLAMI, A.; HAOUES, M.; BEN-ABDALLAH, H. Automated COSMIC-based analysis and consistency verification of UML activity and component diagrams. In: *Evaluation of Novel Approaches to Software Engineering ENASE*. [S.l.]: Springer, 2013. v. 417, p. 48–63. Citado na página 22.

SELLERS, G.; WRIGHT, R. S.; JR, N. H. *OpenGL Super Bible: Comprehensive tutorial and reference. Seventh edition*. [S.l.]: Pearson Education Inc, 2016. ISBN 978-0-672-33747-5. Citado na página 38.

SHEN, L. et al. Domain-specific language techniques for visual computing: A comprehensive study. In: *Arch Computat Methods Eng 28*. [S.l.]: SPRINGER, 2020. p. 3113–3134. Citado na página 55.

SOZYKIN, A.; EPANCHINTSEV, T. MIPr: a framework for distributed image processing using hadoop. In: *2015 9th International Conference on Application of Information and Communication Technologies (AICT)*. [S.l.: s.n.], 2015. p. 35–39. Citado na página 15.

TONELLA, P.; POTRICH, A. Package diagram. In: *Reverse Engineering of Object Oriented Code*. [S.l.]: Springer, 2005. p. 133–154. ISBN 978-0-387-40295-6. Citado na página 22.

TONG, J.; CHENG-DONG, W.; DONG-YUE, C. Research and implementation of a digital image processing education platform. In: *2011 International Conference on Electrical and Control Engineering*. [S.l.]: IEEE, 2011. p. 6719–6722. Citado 2 vezes nas páginas 37 e 54.

TRAISUWAN, A.; TANDAYYA, P.; LIMNA, T. Workflow translation and dynamic invocation for image processing based on OpenCV. In: *2015 12th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. [S.l.]: IEEE, 2015. p. 319–324. Citado na página 54.

UKIS, V. et al. Architecture of cloud-based advanced medical image visualization solution. In: *International Conference on Cloud Computing in Emerging Markets*. [S.l.]: IEEE, 2013. p. 1–5. Citado 2 vezes nas páginas 15 e 17.

VIDONI, M.; VECCHIETTI, A. Towards a Reference Architecture for Advanced Planning Systems. In: INSTICC. *Proceedings of the 18th International Conference on Enterprise Information Systems - Volume 1: ICEIS,*. [S.l.]: SciTePress, 2016. p. 433–440. ISBN 978-989-758-187-8. Citado na página 24.

WANG, J.; HOGUE, A. CVNodes: A visual programming paradigm for developing computer vision algorithms. In: *17th Conference on Computer and Robot Vision (CRV)*. [S.l.]: IEEE, 2020. p. 174–181. Citado 3 vezes nas páginas 28, 33 e 34.

WANG, Y. K.; HUANG, W. B. Acceleration of an improved retinex algorithm. In: *In Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. [S.l.]: IEEE, 2008. p. 72–77. Citado na página 38.

XIAOQI, L.; XIN, L.; DONGZHENG, J. Research and implement of three-dimensional reconstruction technology for medical images based on IDL. In: *International Conference on Computer Science and Service System*. [S.l.]: IEEE, 2012. p. 2317–2321. Citado na página 30.

YOUNG, M.; ARGIRO, D.; KUBICA, S. Cantata: Visual programming environment for the Khoros system. *SIGGRAPH Computer Graphics*, ACM, v. 29, n. 2, p. 22–24, 1995. ISSN 0097-8930. Citado 2 vezes nas páginas 29 e 30.

YOUNG, M.; ARGIRO, D.; WORLEY, J. An object oriented visual programming language toolkit. *SIGGRAPH Computer Graphics*, ACM, v. 29, n. 2, p. 25–28, 1995. ISSN 0097-8930. Citado 3 vezes nas páginas 28, 29 e 30.

Apêndices

APÊNDICE A – Arquivo *workflow* Demo

```
#  Glyph 'Image load'
Glyph:VGL_CL:vglLoadImage:localhost:1:302:82: -filename
    'images/orig.png'

#  Glyph 'Create Image Rgb2Gray'
Glyph:VGL_CL:vglCreateImage::localhost:2:562:122::

#  Glyph 'vglClRgb2Gray'
Glyph:VGL_CL:vglClRgb2Gray::localhost:3:382:182::

#  Glyph 'Create Image Convolution'
Glyph:VGL_CL:vglCreateImage::localhost:4:562:122::

#  Glyph 'vglClConvolution'
Glyph:VGL_CL:vglClConvolution::localhost:5:382:182::
    -convolution_window [0.0030, 0.0133, 0.0219, 0.0133,
    0.0030, 0.0133, 0.0596, 0.0983, 0.0596, 0.0133, 0.0219,
    0.0983, 0.1621, 0.0983, 0.0219, 0.0133, 0.0596, 0.0983,
    0.0596, 0.0133, 0.0030, 0.0133, 0.0219, 0.0133, 0.0030]
    -window_size_x 5 -window_size_y 5

#  Glyph 'Create Image Dilate'
Glyph:VGL_CL:vglCreateImage::localhost:6:562:122::

#  Glyph 'vglClDilate'
Glyph:VGL_CL:vglClDilate::localhost:7:382:182:: -convolution_window
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
    -window_size_x 5 -window_size_y 5

#  Glyph 'Create Image Erode'
Glyph:VGL_CL:vglCreateImage::localhost:8:562:242::

#  Glyph 'vglClErode'
Glyph:VGL_CL:vglClErode::localhost:9:462:302:: -convolution_window
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
```

```
    -window_size_x 5 -window_size_y 5

#  Glyph 'Image save Convolution'
Glyph:VGL_CL:vglSaveImage:localhost:10:882:482:
    -filename 'conv.png'

#  Glyph 'Image save Dilate'
Glyph:VGL_CL:vglSaveImage:localhost:11:882:482:
    -filename 'dilate.png'

#  Glyph 'Image save Erode'
Glyph:VGL_CL:vglSaveImage:localhost:12:882:482:
    -filename 'erode.png'

#  Glyph 'Image Show Erode'
Glyph:VGL_CL:ShowImage:localhost:13:882:482:
    -winame 'erode.png'

#  Connections 'Applying Rgb2Gray'
NodeConnection:data:1:RETVAL:2:img
NodeConnection:data:1:RETVAL:3:img_input
NodeConnection:data:2:RETVAL:3:img_output

#  Connections 'Applying Convolution'
NodeConnection:data:3:img_output:4:img
NodeConnection:data:3:img_output:5:img_input
NodeConnection:data:4:RETVAL:5:img_output

#Connections 'Save Convolution'
NodeConnection:data:5:img_output:10:image

#Connections 'Applying Dilate'
NodeConnection:data:5:img_output:6:img
NodeConnection:data:5:img_output:7:img_input
NodeConnection:data:6:RETVAL:7:img_output

#Connections 'Save Dilate'
NodeConnection:data:7:img_output:11:image
```

```
#Connections 'Applying Erode'
NodeConnection:data:7:img_output:8:img
NodeConnection:data:7:img_output:9:img_input
NodeConnection:data:8:RETVAL:9:img_output

#Connections 'Save Erode'
NodeConnection:data:9:img_output:12:image
NodeConnection:data:9:img_output:13:image
```

APÊNDICE B – Arquivo *workflow* Fundus

```
#  Glyph 'Image load'
Glyph:VGL_CL:vglLoadImage:localhost:1:302:82: -filename
    'images/goodQuality/1_good.JPG' -iscolor 1 -has_mipmap 0

#  Glyph 'Create Image Rgb2Gray'
Glyph:VGL_CL:vglCreateImage::localhost:2:562:122::

#  Glyph 'vglClRgb2Gray'
Glyph:VGL_CL:vglClRgb2Gray::localhost:3:382:182::

#  Glyph 'Create Image Convolution'
Glyph:VGL_CL:vglCreateImage::localhost:4:562:242::

#  Glyph 'vglClConvolution'
Glyph:VGL_CL:vglClConvolution::localhost:5:462:302::
    -convolution_window
    ['0.00037832','0.00055477','0.00080091','0.00113832',
    '0.00159279','0.00219416','0.00297573','0.00397312',
    '0.00522256','0.0067585','0.00861055','0.01080005',
    '0.01333629','0.0162128','0.01940418','0.02286371',
    '0.02652237','0.0302895','0.0340554','0.03769589',
    '0.04107865','0.04407096','0.04654821','0.04840248',
    '0.04955031','0.04993894','0.04955031','0.04840248',
    '0.04654821','0.04407096','0.04107865','0.03769589',
    '0.0340554',0'0.0302895','0.02652237','0.02286371',
    '0.01940418','0.0162128','0.01333629','0.01080005',
    '0.00861055','0.0067585','0.00522256','0.00397312',
    '0.00297573','0.00219416','0.00159279','0.00113832',
    '0.00080091','0.00055477','0.00037832']
    -windows 51 -windows 1

#  Glyph 'Create Image Convolution'
Glyph:VGL_CL:vglCreateImage::localhost:6:562:242::

#  Glyph 'vglClConvolution'
```

```
Glyph:VGL_CL:vglClConvolution::localhost:7:462:302::
    -convolution_window
    ['0.00037832','0.00055477','0.00080091','0.00113832','
    0.00159279','0.00219416','0.00297573','0.00397312',
    '0.00522256','0.0067585','0.00861055','0.01080005',
    '0.01333629','0.0162128','0.01940418','0.02286371',
    '0.02652237','0.0302895','0.0340554','0.03769589',
    '0.04107865','0.04407096','0.04654821','0.04840248',
    '0.04955031','0.04993894','0.04955031','0.04840248',
    '0.04654821','0.04407096','0.04107865','0.03769589',
    '0.0340554',0'0.0302895','0.02652237','0.02286371',
    '0.01940418','0.0162128','0.01333629','0.01080005',
    '0.00861055','0.0067585','0.00522256','0.00397312',
    '0.00297573','0.00219416','0.00159279','0.00113832',
    '0.00080091','0.00055477','0.00037832']
    -windows 1 -windows 51

#  Glyph 'Create Image Dilate'
Glyph:VGL_CL:vglCreateImage::localhost:8:562:242::

#  Glyph 'Dilate'
Glyph:VGL_CL:vglClDilate::localhost:9:462:302::
    -convolution_window
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
    -window_size_x 51 -window_size_y 1

#  Glyph 'Create Image Dilate'
Glyph:VGL_CL:vglCreateImage::localhost:10:562:242::

#  Glyph 'vglClDilate'
Glyph:VGL_CL:vglClDilate::localhost:11:462:302::
    -convolution_window
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
    -window_size_x 1 -window_size_y 51

#  Glyph 'Create Image Erode'
Glyph:VGL_CL:vglCreateImage::localhost:12:562:242::
```

```
#  Glyph 'vglClErode'
Glyph:VGL_CL:vglClErode::localhost:13:462:302::
    -convolution_window
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
    -window_size_x 1 -window_size_y 51

#  Glyph 'Create Image Erode'
Glyph:VGL_CL:vglCreateImage::localhost:14:562:242::

#  Glyph 'Erode'
Glyph:VGL_CL:vglClErode::localhost:15:462:302::
    -convolution_window
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
    -window_size_x 51 -window_size_y 1

#  Glyph 'Create Image Sub'
Glyph:VGL_CL:vglCreateImage::localhost:16:562:242::

# Glyph 'Sub'
Glyph:VGL_CL:vglClSub::localhost:17:462:302::

#  Glyph 'Create Image Threshold'
Glyph:VGL_CL:vglCreateImage::localhost:18:562:242::

#  Glyph 'vglClThreshold'
Glyph:VGL_CL:vglClThreshold::localhost:19:462:302::
    -thresh 0.00784 -top 1

#  Glyph 'Create Image Reconstruct'
Glyph:VGL_CL:vglCreateImage::localhost:20:562:242::

#  Glyph 'Reconstruct'
Glyph:VGL_CL:Reconstruct::localhost:21:462:302::
    -convolution_window
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
```

```
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
    -window_size_x 17 -window_size_y 17

#  Glyph 'Image save Rbg2Gray'
Glyph:VGL_CL:vglSaveImage::localhost:22:882:362::
    -filename 'artigo/fundusRgb2gray.png'

#  Glyph 'Image save Convolution'
Glyph:VGL_CL:vglSaveImage::localhost:23:882:362::
    -filename 'artigo/fundusConvolution.png'

#  Glyph 'Image save Dilate'
Glyph:VGL_CL:vglSaveImage::localhost:24:882:362::
    -filename 'artigo/fundusDilate.png'

#  Glyph 'Image save Erode'
Glyph:VGL_CL:vglSaveImage::localhost:25:882:362::
    -filename 'artigo/fundusErode.png'

#  Glyph 'Image save Sub'
Glyph:VGL_CL:vglSaveImage::localhost:26:882:362::
    -filename 'artigo/fundusSub.png'

#  Glyph 'Image save Thresh'
Glyph:VGL_CL:vglSaveImage::localhost:27:882:362::
    -filename 'artigo/fundusThresh.png'

#  Glyph 'Image save Rec'
Glyph:VGL_CL:vglSaveImage::localhost:28:882:362::
    -filename 'artigo/fundusRec.png'
```

```
#  Connections 'Applying Rgb2Gray'
NodeConnection:data:1:RETVAL:2:img
NodeConnection:data:1:RETVAL:3:img_input
NodeConnection:data:2:RETVAL:3:img_output

# Connections 'Show in Rgb2Gray'
NodeConnection:data:3:img_output:22:image

#Connections 'Applying Convolution_1x51'
NodeConnection:data:3:img_output:4:img
NodeConnection:data:3:img_output:5:img_input
NodeConnection:data:4:RETVAL:5:img_output

#Connections 'Applying Convolution_51x1'
NodeConnection:data:5:img_output:6:img
NodeConnection:data:5:img_output:7:img_input
NodeConnection:data:6:RETVAL:7:img_output

#Connections 'Save Convolution'
NodeConnection:data:7:img_output:23:image
NodeConnection:data:7:img_output:17:img_input2

#Connections 'Applying Dilate 1x51'
NodeConnection:data:7:img_output:8:img
NodeConnection:data:7:img_output:9:img_input
NodeConnection:data:8:RETVAL:9:img_output

#Connections 'Applying Dilate 51x1'
NodeConnection:data:9:img_output:10:img
NodeConnection:data:9:img_output:11:img_input
NodeConnection:data:10:RETVAL:11:img_output

#Connections 'Save Dilate'
NodeConnection:data:11:img_output:24:image

#Connections 'Applying Erode 1x51'
NodeConnection:data:11:img_output:12:img
NodeConnection:data:11:img_output:13:img_input
NodeConnection:data:12:RETVAL:13:img_output
```

```
# Connections 'Applying Erode 51x1'
NodeConnection:data:13:img_output:14:img
NodeConnection:data:13:img_output:15:img_input
NodeConnection:data:14:RETVAL:15:img_output

#Connections 'Save Erode'
NodeConnection:data:15:img_output:25:image

# Connections 'Applying Sub'
NodeConnection:data:15:img_output:16:img
NodeConnection:data:15:img_output:17:img_input1
NodeConnection:data:16:RETVAL:17:img_output

#Connections 'Save Sub'
NodeConnection:data:17:img_output:26:image

# Connections 'Applying Thresh'
NodeConnection:data:17:img_output:18:img
NodeConnection:data:17:img_output:19:src
NodeConnection:data:18:RETVAL:19:dst

# Connections 'Save Thresh'
NodeConnection:data:19:dst:27:image

#Connections 'Applying Rec'
NodeConnection:data:19:dst:20:img
NodeConnection:data:19:dst:21:img_input
NodeConnection:data:20:RETVAL:21:img_output

#Connections 'Save Rec'
NodeConnection:data:21:img_output:28:image
```