PyImageVis: Processamento e Visualização de Imagens Médicas em Python

Diego A. T. Q. Leite[1,2], **Gilson A. Giraldi** [1], **Pedro H. M. Lira** [1], **Ricardo E. Kneipp**[2]

[1]Laboratório Nacional de Computação Científica
CEP 25651-075 – Petrópolis – RJ – Brazil

[2]Instituto Superior de Tecnologia em Ciência da Computação de Petrópolis
CEP 25651-070 – Petrópolis – RJ – Brazil

{diegoamim,ricardo.kneipp}@gmail.com,{pedrohml,gilson}@lncc.br

Abstract. *This paper describes the PyImageVis system, an open-source software implemented in Python, for processing and visualization of medical images. The main goal of PyImageVis is to provide a computational system for the research of new algorithms for medical image processing and visualization. In this work, we describe the capabilities of the PyImageVis and present details of the software engineering behind its development. Besides, we demonstrate its potential with the execution of processing tasks in dental radiographs as well as computed tomography. Finally, we compare PyImageVis with related systems and discuss its performance.*

Resumo. *Neste trabalho descrevemos o software* PyImageVis*, implementado em Python, utilizando recursos de bibliotecas de código livre. O principal objetivo do* PyImageVis *é proporcionar uma plataforma facilmente extensível para teste e desenvolvimento de algoritmos para processamento e visualização de imagens médicas. Neste artigo, descrevemos as principais características do* PyImageVis*, apresentamos detalhes de sua modelagem e implementação. Demonstramos sua utilidade com exemplos envolvendo radiografias dentárias e tomografias. Por fim, comparamos este software com outros do gênero e discutimos sua eficiência computacional.*

1. Introdução

Nas últimas décadas, os pesquisadores das áreas de computação gráfica e processamento de imagens desenvolveram técnicas de visualização e análise de imagens radiológicas permitindo a elaboração de novas metodologias para diagnosticar doenças [Bankman 2009]. Tais técnicas permitem que os médicos visualizem, analisem e manipulem tanto imagens bidimensionais como as representações tridimensionais dos órgãos, tecidos, ou outras estruturas do corpo do paciente [Suri et al. 2002]. Neste trabalho, as aplicações em foco envolvem tomografias computadorizadas e radiografias dentárias.

Um exame tomográfico é constituído por uma sequência de imagens bidimensionais da região de interesse, fornecendo informações detalhadas dos tecidos, o que proporciona ao profissional da área de saúde uma avaliação mais precisa das condições do paciente [Suri et al. 2002]. Uma vez realizado o exame tomográfico, o volume de dados gerado (imagem tridimensional) pode ser tratado via técnicas de processamento de

ISSN: 1519-8219

imagens e computação gráfica, com o objetivo de extrair informações ou gerar modelos tridimensionais dos órgãos de interesse [Bárbara and de Carvalho Zavaglia 2006]. Por outro lado, a identificação automática de lesões em radiografias dentárias é uma área de crescente interesse em função da grande aplicação deste tipo de imagem para diagnóstico odontológico [Zhou and Abdel-Mottaleb 2005].

No Brasil, a pesquisa de novas técnicas para visualização e processamento de imagens médicas está concentrada nas universidades e institutos de pesquisa. O Instituto Nacional de Ciência e Tecnologia em Medicina Assistida por Computação Científica (INCT-MACC) [1], coordenado pelo Laboratório Nacional de Computação Científica (LNCC), é um exemplo deste fato. Os profissionais envolvidos necessitam de uma plataforma de trabalho que facilite a validação de novos algoritmos. Softwares como o *MatLab* [Blanchet and Charbit 2006], ou *SciLab* [da Motta Pires 2004] podem ser usados para esta finalidade. Contudo, são limitados para a visualização de imagens tridimensionais. Visando atender esta carência, apresentamos neste trabalho o software *PyImageVis*. Este software foi desenvolvido principalmente para uso acadêmico, no intuito de proporcionar a pesquisadores da área de visualização e processamento de imagens, uma plataforma que aproveite bibliotecas de código livre já existentes, com interface amigável e, principalmente, que seja facilmente expansível.

Todo o sistema foi desenvolvido em *Python* [Hetland 2008], uma linguagem de programação de alto nível e interpretada; ou seja, o código fonte não é compilado e sim interpretado. A linguagem é orientada a objetos e de tipagem dinâmica e forte, tendo interpretadores para diferentes plataformas, como *Windows*, *MacOS* e *Linux*. A linguagem *Python* foi escolhida neste trabalho por apresentar uma sintaxe enxuta, legível e expressiva, o que facilita o raciocínio durante a implementação de protótipos mais complexos, além de aumentar a produtividade do programador. Além disso, os recursos da linguagem tornam muito mais simples incorporar novas atualizações no sistema. Por fim, a linguagem *Python* oferece pleno suporte para o desenvolvimento de classes para processamento/visualização de imagens em função das bibliotecas disponíveis para estas áreas [2].

O sistema utiliza interfaces que seguem uma metodologia análoga à do *MatLab*: o usuário pode acessar os recursos disponíveis via chamadas de funções em linha-de-comando (no *prompt* do interpretador) ou execução de *scripts* personalizados para uma aplicação específica. Além disto, novos operadores podem ser incluídos ao sistema via inserção de uma linha de código no arquivo *__init__.py* encontrado no diretório *lib* do sistema. Embora o caráter de pesquisa seja mais significativo, esta plataforma pode também ser utilizada para desenvolver aplicações que possam ser utilizadas diretamente pelo profissional médico. O software pode editar vários formatos de imagens existentes, tais como *TIFF*, *GIF*, *BMP*, *PICT*, *JPEG*, *PNG* e *DICOM*.

Este trabalho está estruturado da seguinte forma. A seção 2 fornece um estudo sobre trabalhos relacionados. Nas seções 3 e 4, são apresentados detalhes sobre o software desenvolvido, sua implementação, interface gráfica e modelagem. Os resultados experimentais demonstrando a utilização do *PyImageVis* serão discutidos na seção 5. Em

[1] INCT-MACC em http://macc.lncc.br/

[2] Porque usar Python em aplicações científicas em http://pyscience-brasil.wikidot.com/python:python-oq-e-pq/

seguida, na seção 6, comparamos o *PyImageVis* com outros sistemas do gênero, do ponto de vista do usuário final. Finalmente, apresentamos as conclusões e trabalhos futuros na seção 7.

2. Trabalhos Relacionados

Esta seção tem como objetivo apresentar os principais sistemas computacionais, desenvolvidos usando *Python*, na área de processamento e visualização de imagens médicas. Desta forma, será possível situar nosso trabalho no contexto em questão. Vale destacar que existe uma vasta lista de softwares desenvolvidos em *C/C++* nesta área [de Carvalho et al. 2002]. Contudo, uma vez que a linguagem de desenvolvimento é o *Python*, optamos por abordar apenas sistemas desenvolvidos nesta linguagem.

A seguir, serão discutidos os softwares *InVesalius* [Bárbara and de Carvalho Zavaglia 2006], *MayaVi* [3] e *BioImageXD* [4]. Estes aplicativos vêm sendo testados pela comunidade, apresentado desempenho satisfatório. O foco desta seção é destacar o principal diferencial do *PyImageVis* em relação a estes aplicativos: interface entre o usuário e os recursos do sistema.

Assim como no *PyImageVis*, estes sistemas fazem uso de bibliotecas gráficas já consagradas na literatura da área. Abaixo, listamos estas bibliotecas apresentando uma pequena descrição das mesmas.

1. *VTK* (*Visualization ToolKit*): Software gratuito que consiste em uma biblioteca de classes para computação gráfica, processamento de imagem e visualização [Kitware 2010].
2. *NumPy*: Biblioteca matemática que suporta matrizes multidimensionais, e possui diversas funções para a manipulação destas estruturas [5].
3. *Tkinter*: Biblioteca gráfica que acompanha a distribuição oficial do interpretador *Python* [Grayson 2000].
4. *wxPython*: *Wrapper* da biblioteca *wxWidgets* [Smart et al. 2011], para a linguagem *Python*, que traz ferramentas para a construção de interfaces gráficas.
5. *SciPy*: Foi desenvolvida para trabalhar com estruturas da biblioteca *NumPy*, adicionando funcionalidades para a computação científica em *Python* que incluem algoritmos para métodos numéricos e processamento de sinais [6].
6. *PIL* (*Python Imaging Library*): É uma biblioteca para a linguagem de programação *Python* que adiciona suporte à edição (conversão para tons-de-cinza, rotação, escala, zoom, etc.) e gravação de imagens $2D$ em diferentes formatos (*TIFF*, *GIF*, *BMP*, *PICT*, *JPEG*, *PNG*).
7. *PyDicom*: Como o próprio nome já diz, foi desenvolvido utilizando a linguagem *Python* com o objetivo de trabalhar com arquivos do tipo *DICOM* [7].
8. *PyQt4*: Biblioteca que permite ao desenvolvedor acessar facilmente o *Qt* [Blanchette and Summerfield 2008] através da linguagem de programação *Python* [Summerfield 2007]. *Qt* é um *toolkit* robusto e multiplataforma para desenvolver interfaces gráficas.

[3] MayaVi em http://mayavi.sourceforge.net/
[4] BioImageXD em http://bioimagexd.net/
[5] NumPy User Guide em http://docs.scipy.org/doc/numpy/user/
[6] SciPy Reference Guide disponível em http://www.scipy.org/
[7] DICOM em http://medical.nema.org/

9. *Matplotlib*: É uma biblioteca escrita em *Python* voltada para edição de gráficos bidimensionais [8], tais como gráficos de funções, histogramas e gráficos de barras.

O *InVesalius* é um software de domínio público, implementado em *Python* e *C++*, tendo como objetivo auxiliar o profissional médico na geração de modelos digitais de próteses e planejamento cirúrgico de seus pacientes [Bárbara and de Carvalho Zavaglia 2006]. O software permite a criação de modelos tridimensionais (3D), correspondentes às estruturas anatômicas dos pacientes, a partir de imagens tomográficas. A Figura 1 ilustra a interface gráfica deste software.

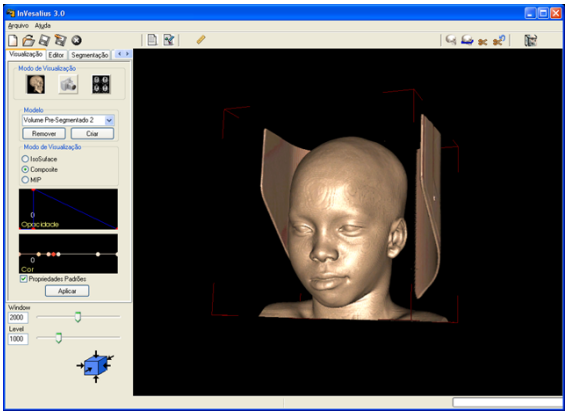

Figura 1. Visualização volumétrica gerada no software *InVesalius*.

Desenvolvido desde julho de 2001 no contexto do *ProMED* (Prototipagem Rápida na Medicina [9]), o programa tem sido amplamente empregado por cirurgiões e hospitais brasileiros desde 2004. A Prototipagem Rápida é uma tecnologia que utiliza modelos digitais projetados em computador para a produção de modelos reais. O projeto ProMED visa a aplicação da computação gráfica e prototipagem rápida no planejamento de cirurgias complexas. A prototipagem também permite uma melhor visão clínica e a construção de próteses específicas para cada caso.

O *MayaVi* é um software de código aberto para visualização de dados científicos. Foi implementado em *Python* utilizando recursos da biblioteca *VTK* [Kitware 2010] e *NumPy*. Sua interface gráfica foi desenvolvida usando a biblioteca *Tkinter* [Grayson 2000], sendo portável tanto para *Windows* quanto *Linux*.

O software *BioImageXD* possui recursos para análise e processamento de imagens de microscopia celular e permite a renderização de modelos digitais. É um software de código aberto, desenvolvido em *Python* e *C++*, usando a biblioteca *wxPython* [Smart et al. 2011] para a interface gráfica e a biblioteca *VTK* para processamento de imagens tridimensionais e renderização.

Todos estes sistemas (*InVesalius*, *MayaVi*, *BioImageXD*) possuem em comum a proposta de usar interfaces intuitivas para o usuário, baseadas em *mouse* e *menus*, facilitando assim sua utilização por profissionais da área de saúde. Porém, são limitados para o desenvolvimento e teste de programas para aplicações científicas. O *PyImageVis*,

[8] Matplotlib User Guide disponível em http://matplotlib.sourceforge.net/contents.html/
[9] ProMED em http://www.cti.gov.br/promed/

por outro lado, tem como público alvo pesquisadores em ciência da computação e/ou engenharia. O usuário acessa os recursos de processamento e visualização de imagens enviando comandos, digitados no *prompt* do interpretador. Essa característica, utilizada em programas consagrados como o *MatLab*, por exemplo, é uma vantagem do software pois facilita a integração entre diferentes algoritmos e reutilização de código-fonte. Assim, embora seja um software que exige familiaridade com programação, o *PyImageVis* tem recursos para ser um *framework* para prototipagem de aplicações complexas em análise e visualização de imagens médicas.

3. Modelagem e Interface do PyImageVis

O objetivo principal deste trabalho é criar um ambiente computacional eficiente e facilmente expansível para o desenvolvimento e teste de métodos em processamento/visualização de imagens médicas. Assim, optamos por utilizar uma interface gráfica análoga à do MatLab, aproveitando a experiência da comunidade na utilização deste sistema.

Desta forma, o software conta com a integração de um interpretador *Python* em sua interface, simulando um *prompt* de comando. Também conta com um editor *Python* para o desenvolvimento de novas funcionalidades no próprio software. A interface gráfica desenvolvida inclui um *Command Window*, uma janela (*History Window*) que contém o histórico dos comandos digitados no *Command Window*; um *Workspace*, onde o programador poderá visualizar o conjunto de variáveis armazenadas na memória. A Figura 2 mostra estes componentes, os quais estão presentes também no sistema *MatLab* [Blanchet and Charbit 2006].

O desenvolvimento do software foi direcionado pela ideia de utilizar algumas das bibliotecas de código aberto descritas na seção 2, possibilitando um desenvolvimento rápido e com baixo custo. Assim, optamos pela biblioteca *SciPy* para processamento de imagens e a biblioteca *VTK* para visualização de imagens $3D$, bem como o paradigma de orientação a objetos para a implementação da interface gráfica. Para usar os recursos da biblioteca *SciPy* precisamos converter imagem em matriz, o que é feito usando funções da biblioteca *NumPy*.

Antes de prosseguirmos com a descrição do sistema é oportuno ter em mente uma visão global da interação entre o usuário e o mesmo. Isto pode ser obtido pela descrição textual dos casos de uso, apresentada na Tabela 1.

Assim, seguindo a Tabela 1, no caso de uso "Ler Imagem", precisamos definir os formatos de interesse para as imagens. Além dos formatos padrões para imagens $2D$, necessitamos editar imagens *DICOM*. Desta forma, optamos por utilizar as bibliotecas *PIL* e *PyDICOM*. O caso de uso "Realizar Processamento", depende de funções de processamento de imagens, ficando portanto contemplado pelas funções da biblioteca *SciPy*.

O caso de uso seguinte, da Tabela 1, "Atualizar *Workspace* e *History Window*", depende da implementação de classes específicas para controle da interface gráfica mostrada na Figura 2. A seguir fornecemos uma breve descrição das classes utilizadas:

- **QMainWindow**: Classe herdada do *PyQt4* que fornece a janela principal do aplicativo.
- **Editor**: Classe responsável pelo editor do software.

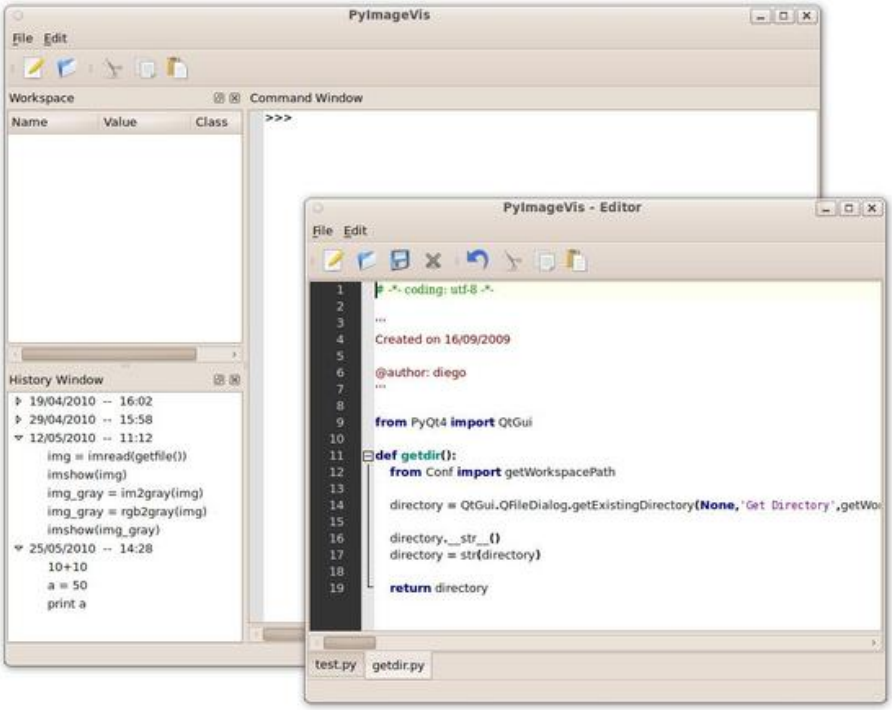

Figura 2. Componentes da interface gráfica do *PyImageVis*: *Command Window*, *History Window*, *Workspace* e Editor *Python*.

- **MainWindow**: Responsável por toda a estrutura e eventos do software.
- **SciShell**: Simula um interpretador *Python* dentro do aplicativo, sendo representada pela janela **Command Window** na interface gráfica.
- **InteractiveInterpreter**: classe responsável pela manipulação do interpretador da linguagem Python.
- **HistoryWindow**: Classe que controla o histórico, adicionado e removendo registros sempre que algum comando é inserido na janela **Command Window**.
- **Workspace**: Armazenadas as variáveis em uso pelo usuário. Está em constante comunicação com a classe **SciShell**.

Estas classes estão representadas no diagrama *UML* (*Unified Modeling Language* [Booch et al. 2000]) da Figura 3.

O último caso de uso da Tabela 1, "Realizar Visualização", necessita de funções de saída gráficas, que permitam visualização de gráficos $2D$ (biblioteca *Matplotlib*), imagens bidimensionais, bem como objetos $3D$ (biblioteca *VTK*). Na seção 4 vamos abordar este ítem.

4. Fluxo de Dados e Implementação

Considerando o objetivo fundamental do sistema e os casos de uso da Tabela 1, podemos afirmar que o fluxo de dados no *PyImageVis* envolve basicamente leitura, processamento e visualização de imagens. As bibliotecas *Python* escolhidas para estas tarefas (*SciPy*, *NumPy*, *PIL*, *PyDICOM*, *Matplotlib* e *VTK*) possuem formatos de dados próprios. Assim, um ponto fundamental no desenvolvimento do sistema é a escolha de um formato padrão

Caso de uso: Ler Imagem	
Ator	Usuário
Pré-Condição	Imagens médicas em formato compatível.
Fluxo Básico	1- O usuário fornece uma imagem ao sistema. 2- O sistema transforma a imagem em uma matriz.
Caso de uso: Realizar Processamento	
Ator	Usuário
Pré-Condição	Imagem em memória.
Fluxo Básico	1- O usuário chama a função a ser executada no software, tendo como entrada a matriz a ser processada. 2- O sistema realiza o processamento.
Caso de uso: Atualizar *Workspace* e *History Window*	
Ator	Sistema
Pré-Condição	Alguma função deve ter sido chamada no software pelo usuário.
Fluxo Básico	1- O sistema verifica se o comando realiza algum tipo de atualização no software. 2- O sistema verifica se o comando não é repetido. 3- O sistema realiza atualizações no *History Window* e/ou *Workspace*.
Caso de uso: Realizar Visualização	
Ator	Usuário
Pré-Condição	A matriz deve estar carregada no sistema e no formato correto.
Fluxo Básico	1- O usuário insere algum comando de visualização no software. 2- O sistema executa o comando. 3- O sistema exibe no *display* o resultado (imagem), permitindo interação com o usuário através do *mouse* e/ou teclado para objetos $3D$.

Tabela 1. Casos de uso textual

para representar as imagens. Foi escolhido o formato ”matriz” da biblioteca *NumPy* pois desta forma podemos utilizar diretamente os recursos de processamento e saída gráfica das biblioteca *SciPy* e *Matplotlib*, respectivamente. O esquema da Figura 4 mostra como as bibliotecas de leitura, processamento e visualização se integram com a biblioteca *NumPy*. Estas bibliotecas estão descritas na seção 2.

Para a utilização da biblioteca *SciPy* não é necessário nenhuma função especial, pois o formato *NumPy* é padrão desta biblioteca. Por outro lado, para as bibliotecas *PIL* e *VTK* foram desenvolvidas as seguintes funções para permitir sua integração no sistema:

- *pil2numpy*() - Recebe uma imagem no formato padrão da biblioteca *PIL* e retorna uma matriz bidimensional no formato *NumPy*.
- *numpy2pil*() - Inversa da função anterior.

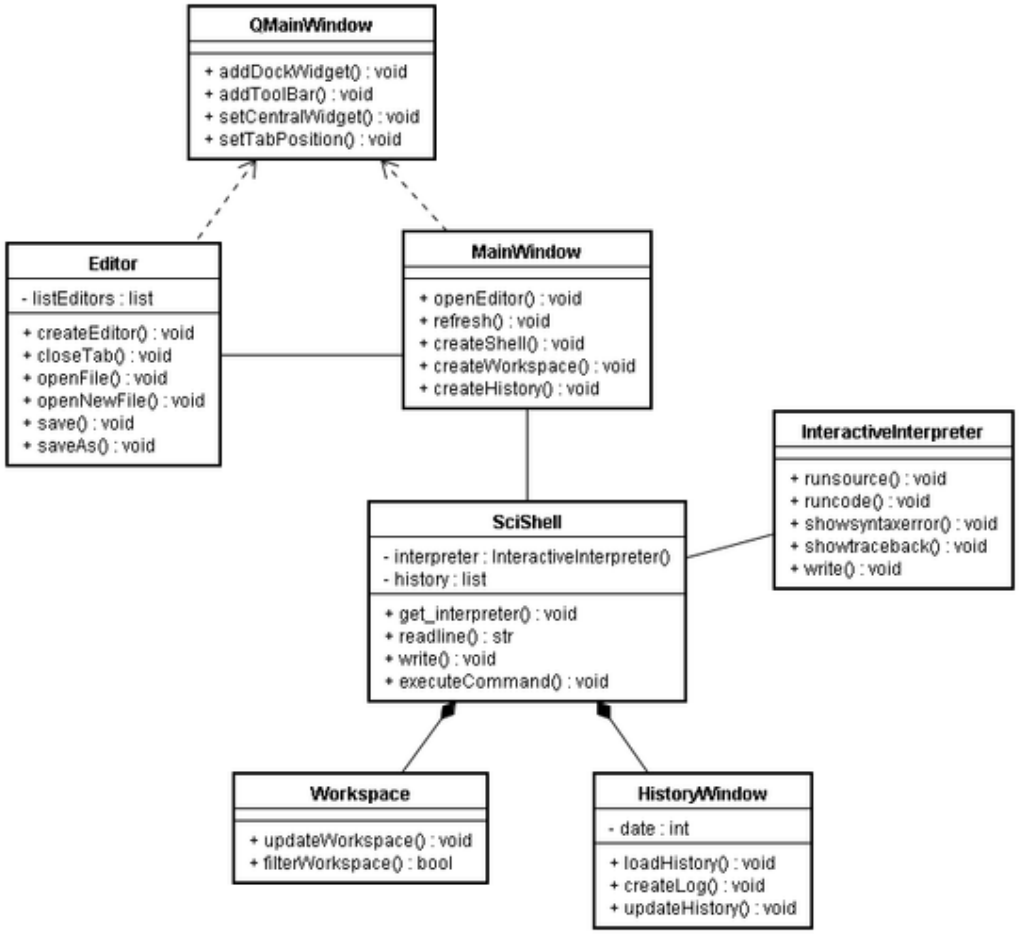

Figura 3. Diagrama de classe

- *numpy2vtk*() - Recebe uma matriz bidimensional ou multidimensional e retorna um objeto *VTK*.

Como pode ser visto na figura 4, as biblioteca responsáveis pela leitura de imagens são a *PIL* e *PyDICOM*. A primeira possui funções para edição de imagens $2D$ e foi escolhida devido a vasta gama de formatos suportados e a facilidade de integração com a biblioteca *NumPy*. Porém a biblioteca *PIL* não suporta o formato *DICOM*, desta forma foi necessário a utilização da biblioteca *PyDICOM*.

Sendo assim, quando um usuário carregar uma imagem $2D$ no *PyImageVis*, a imagem será lida diretamente pela função *imread*, descrita no Algoritmo 1, onde são chamadas funções da biblioteca *PIL* para a leitura da imagem e funções da biblioteca *NumPy* para a transformação da imagem em matriz. Neste processo é utilizada a função *pil2numpy*, citada acima. Se o formato for *DICOM*, inicialmente, a imagem passará pela *PyDICOM* para ser convertida no formato *PIL*. Em seguida, será convertida no formato *NumPy*. Este processo é executado pela função *dicomread* implementada segundo o Algoritmo 2.

No caso de imagens $3D$, foi implementada a função *loadvolume* que recebe

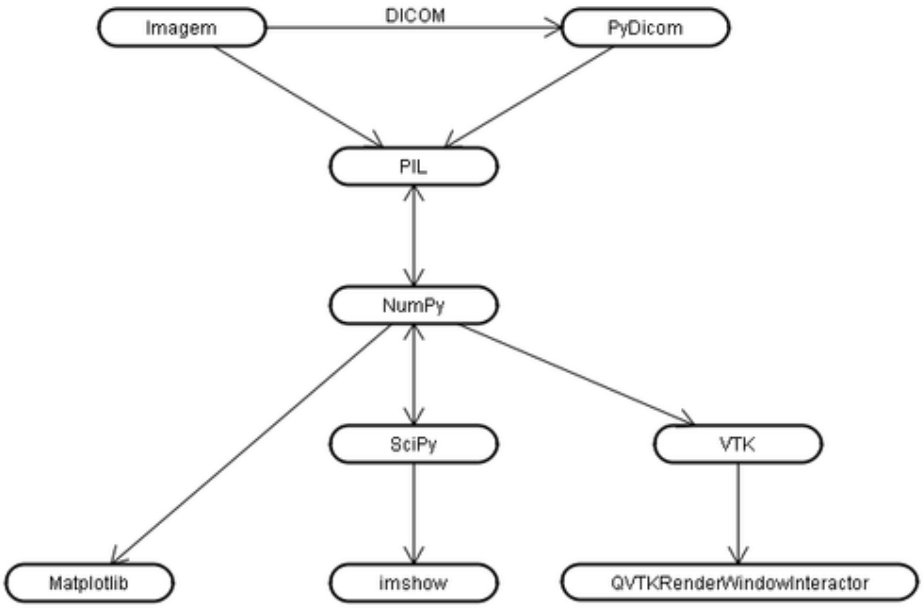

Figura 4. Diagrama mostrando a comunicação entre as bibliotecas no sistema *PyImageVis* e os recursos para saída gráfica (biblioteca *Matplotlib*, função *imshow* e *widget QVTKRenderWindowInteractor*).

o endereço (nome ou caminho) das imagens, executa a leitura, por um procedimento análogo àquele utilizado no Algoritmo 1, e retorna uma matriz multidimensional no formato *NumPy*.

```
image_pil = Image.open(path)
image_numpy = pil2numpy(image_pil)
return image_numpy
```

Algoritmo 1: Função *imread*.

```
image_dicom = dicom.ReadFile(path)
size = (image_dicom.Columns,image_dicom.Rows)
image_pil = Image.frombuffer(size, image_dicom.PixelData)
image_numpy = pil2numpy(image_pil, 'f')
return image_numpy
```

Algoritmo 2: Função *dicomread*.

Finalmente, o resultado gerado em um processamento deverá ser visualizado (nível inferior do fluxograma da Figura 4). Se o resultado for um gráfico $2D$ (histograma ou função $y = f(x)$), utilizamos as funções da biblioteca Matplotlib, a qual recebe dados no formato *NumPy*. Para imagens $2D$ implementamos a função *imshow*, descrita no Algoritmo 3. A *widget QVTKRenderWindowInteractor* (criada por Prabhu Ramachandran em 2002), desenvolvida para *PyQt4* (seção 2), é utilizada para a visualização de resultados $3D$ gerados pelos métodos do *VTK*.

```
image_pil = numpy2pil(image_numpy)
image_pil.show()
```

Algoritmo 3: Função *imshow*.

5. Resultados

Nesta seção, vamos demonstrar algumas funcionalidades da ferramenta desenvolvida, descrita nas seções 3 e 4. Inicialmente, vamos explorar edição e processamento de imagens bidimensionais. Em seguida, vamos demonstrar visualização volumétrica via *Marching Cubes* e *volume rendering*, aproveitando este último caso para demonstrar como incluir novas funções ao sistema.

Para a geração dos resultados contidos nesta seção, foi utilizada uma máquina equipada com processador Intel Core 2 Quad de 3.00 GHz com 4GB de memória RAM e contendo duas placas de vídeo GeForce 9800 GTX com 512 MB de memória. O sistema operacional utilizado foi o Ubuntu 9.10.

5.1. Processamento de Imagens

A Figura 5 mostra os componentes principais da interface gráfica, e sua aplicação na edição de uma radiografia dentária. A sequência de operações utilizadas está descrita no Algoritmo 4.

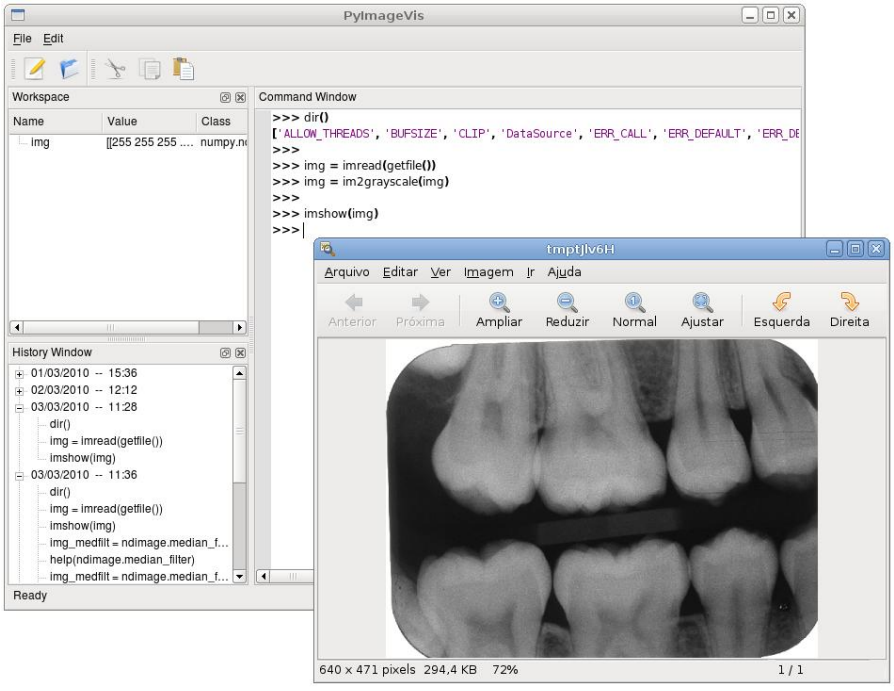

Figura 5. Visualização de uma radiografia dentária, do tipo *bite wing*, através do software *PyImageVis*. Resultado do Algoritmo 4.

As funções *imread* e *imshow* estão descritas nos Algoritmos 1 e 3, respectivamente. A função *im2grayscale* chama a função *grayscale()* da biblioteca *PIL* para converter imagens em tons-de-cinza (0 - 255). Por outro lado, a função *getfile()* permite

```
imagem = imread( getfile() )
imagem = im2grayscale( imagem )
imshow( imagem )
```

Algoritmo 4: Algoritmo para leitura e visualização de imagem bidimensional. Ver Figura 5.

ao usuário navegar pelos diretórios do computador, através de uma interface gráfica, até selecionar a imagem desejada. Desta forma, o retorno da função *getfile()* (endereço da imagem selecionada) é direcionado diretamente como argumento da função *imread()* para leitura da imagem. Finalmente, o comando *imshow()* executa a função responsável pela visualização (Figura 5).

Radiografias do tipo apresentada na Figura 5 (*bite wing*) são utilizadas em um projeto envolvendo pesquisadores da Faculdade de Odontologia da Universidade Estadual do Rio de Janeiro (FO-UERJ) e do LNCC, cujo objetivo é estudar a periodontite, uma inflamação crônica e destrutiva que leva à perda dos tecidos de sustentação dos dentes e, eventualmente, a perda dentária. O primeiro passo no processamento destas imagens é remover artefatos da região entre as arcadas dentárias. Embora esta região aparente ter intensidade homogênea, observamos a presença de artefatos oriundos do protocolo de aquisição das imagens. Isto fica evidenciado na Figura 6.(a) onde efetuamos a visualização da imagem após a binarização pelo limiar 25.

A Figura 6.(b) mostra o resultado da aplicação de três filtros sobre a imagem da Figura 5. A sequência de operações utilizadas está descrita no Algoritmo 5. Inicialmente, a imagem original é processada com o filtro da mediana [Gonzalez and Woods 1992, Vaseghi 2000]. Nesta operação, a intensidade de cada pixel (m, n) da imagem original I é substituída pela mediana das intensidades dos *pixels* no interior de uma janela W centrada em (m, n).

```
imagem_medfilt = ndimage.median_filter( imagem, (3,3) )
imshow( imagem_medfilt )
imagem_contrast = imcontrast( imagem_medfilt, 1.1 )
imagem_bin = imagem_contrast > 30
imshow( imagem_bin )
```

Algoritmo 5: Filtro da mediana, realce de contraste e visualização

Na primeira linha do algoritmo 5, encontra-se a palavra *ndimage* que representa o pacote de processamento de imagens da biblioteca científica *SciPy*. Dentro deste pacote encontra-se a função *median_filter()* que implementa o filtro da mediana. É passado como argumento para este método a imagem juntamente com uma *tupla* que definirá o tamanho da vizinhança. O próximo passo é a aplicação da função *imcontrast()* que tem o objetivo de realçar o contraste da imagem. Esta função tem como argumentos de entrada a imagem e um valor decimal que controla a porcentagem de contraste a ser aplicado na imagem (valores maiores que 1.0 melhoram o contraste). Na quarta linha do algoritmo 5, é realizada a binarização da imagem, onde foi utilizado como limiar o valor 30 (Figura 6(b)).

Em seguida, aplicamos a dilatação [Gonzalez and Woods 1992] do pacote

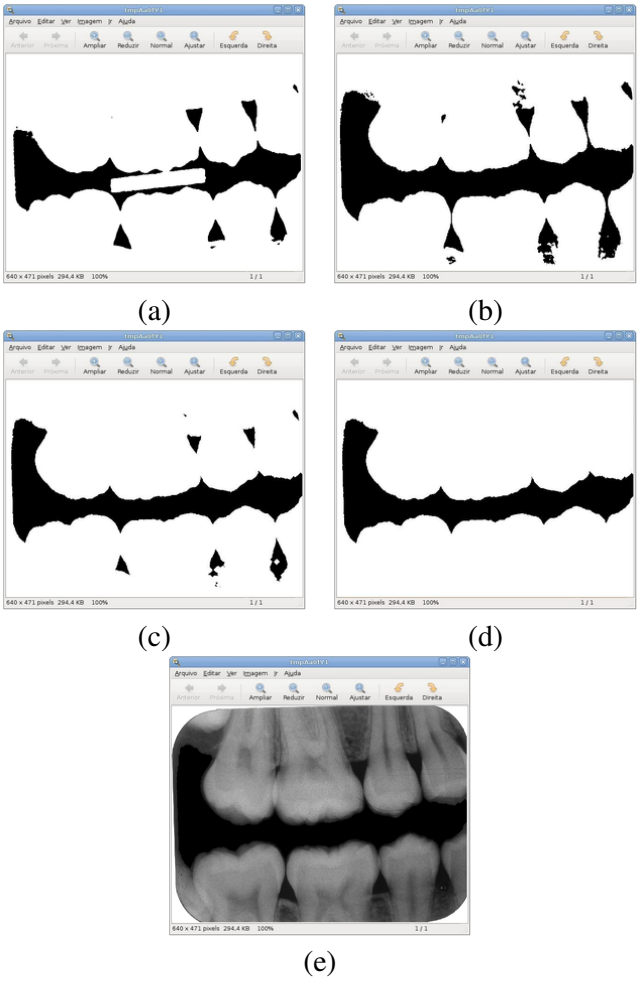

Figura 6. Resultados do Algoritmo 5 e 6. (a) Imagem original binarizada realçando artefato entre as arcadas. (b) Imagem binarizada após aplicação do filtro da mediana e realce de contraste. (c) Resultado da dilatação sobre a imagem binarizada. (d) Imagem gerada pelo operador *binary_fill_holes()*. (e) Visualização da radiografia dentária sem artefatos entre as arcadas.

ndimage, com o objetivo de separar a região entre as arcadas das demais regiões. Na primeira linha do algoritmo 6, existe uma variável chamada *'struct'* que recebe uma matriz 3×3 que é o elemento estruturante a ser passado para a operação morfológica.

Para realizar a tarefa de dilatação foi utilizada a função *binary_dilation()*, chamada na segunda linha. Além da imagem e do elemento estruturante, é passado também o número de interações do operador (5, no caso). A Figura 6.(c) mostra a imagem resultante. Observamos a separação da região de interesse das demais regiões, ficando "buracos" no fundo da imagem (regiões menores imersas no fundo branco). Assim, na quarta linha do algoritmo 6, é aplicado uma operação para preenchimento de regiões (comando *binary_fill_holes()* do pacote *ndimage*) com o objetivo de eliminar estas regiões (Figura 6.(d)).

Finalmente, basta inverter, a imagem gerada, utilizando a função *iminvert()* que

```
struct = array([ [0,1,0], [1,1,1], [0,1,0] ])
imagem_dilation = ndimage.binary_dilation( imagem_bin, struct, 5 )
imshow( imagem_dilation )
imagem_holes = ndimage.binary_fill_holes( imagem_dilation )
imshow( imagem_holes )
imagem_invert = iminvert( imagem_holes )
imagem_subtract = imsubtract( imagem, imagem_invert )
imshow( imagem_subtract )
```

Algoritmo 6: Eliminação do background

utiliza métodos da *PIL*, e subtrair a imagem original da imagem invertida através da função *imsubtract()* que também utiliza métodos da *PIL*. Nesta etapa, artefatos na região entre as arcadas são eliminados, ou seja, o campo de intensidades tornou-se homogêneo nesta região. O resultado está mostrado na Figura 6.(e).

5.2. Reconstrução de Superfícies via *Marching Cubes*

A ideia é gerar uma aproximação poligonal da superfície (isosuperfície) definida por uma expressão do tipo:

$$\{(x_1, x_2, x_3) \in \Re^3; I\,(x_1, x_2, x_3) = \lambda\}, \tag{1}$$

onde $I : \Delta \rightarrow \Re$ é a imagem, com $\Delta \subset \Re^3$, e λ é uma constante. Um algoritmo comumente usado na geração de isosuperfícies em imagens $3D$ é o algoritmo *Marching Cubes* [Lorensen and Cline 1987].

Para inserir o algoritmo *Marching Cubes* no software *PyImageVis* implementamos uma função *marchingcubes()* e uma classe denominada *MarchingCubes*, descritas nos Algoritmo 7 e 8, respectivamente. A função *marchingcubes()* recebe uma matriz *NumPy* multidimensional, criada através da função *loadvolume()*. Logo em seguida a classe *MarchingCubes* é instanciada, sendo passado como argumento a matriz *NumPy*. Na classe *MarchingCubes* é feita a conversão da matriz *NumPy* para um objeto *VTK* através da função *numpy2vtk()*. Em seguida, é instanciada a classe *vtkImageMarchingCubes*, do *VTK*, responsável pela computação da isosuperfície.

```
input : Matriz multidimensional NumPy (data_numpy)
mc = MarchingCubes(data_numpy)
mc.show()
return mc
```

Algoritmo 7: função marchingcubes.

Para disponibilizar a função *marchingcubes()* no *PyImageVis* é necessário que o arquivo onde se encontra esta função e também a classe *MarchingCubes* estejam dentro do diretório *'lib'* do sistema. E por último é necessário inserir a linha de código "from *file* import *function_name*", sendo *file* o nome do arquivo onde se encontra a função e *function_name* o nome da função a ser importada, no arquivo __init__.py, também encontrado no diretório *'lib'*. Qualquer outra classe do *VTK* poderá ser inserida no *PyImageVis* por um procedimento análogo.

```
     input  : Matriz multidimensional NumPy (dataImporter)
1    dataImporter = numpy2vtk(data_numpy)

2    Geração da isosuperfície
3    cubes = vtkImageMarchingCubes()
4    cubes.SetInputConnection(dataImporter.GetOutputPort())

5    Plano de Corte
6    planeWidget = vtkImagePlaneWidget()
7    planeWidget.SetInput(dataImporter.GetOutput())
8    planeWidget.SetPlaneOrientationToZAxes()
9    plane = vtkPlane()

10   Visualização
11   mapper = vtkPolyDataMapper()
12   mapper.SetInputConnection(cubes.GetOutputPort())
13   mapper.AddClippingPlane(plane)
14   actor = vtkActor()
15   actor.SetMapper(mapper)
16   renderer = vtkRenderer()
17   renderer.AddActor(actor)
18   renWin = vtkRenderWindow()
19   renWin.AddRenderer(renderer)
```

Algoritmo 8: classe MarchingCubes.

Veja no algoritmo 9 os comandos necessários para utilizar o algoritmo *Marching Cubes* no software *PyImageVis*. Na primeira linha do algoritmo 9 encontra-se a função *loadvolume()*, que tem o objetivo de ler o volume de imagens, localizado no diretório retornado pela função *getdir()*, e armazená-las em uma matriz multidimensional, no formato *NumPy*. Na segunda linha é feita a chamada à função *marchingcubes()*, descrita no Algoritmo 7.

```
v = loadvolume( getdir() )
cubes = marchingcubes( v )
```

Algoritmo 9: Gerando isosuperfície via *marching cubes*

O resultado do Algoritmo 9 pode ser visto na figura 7. Neste caso foi utilizado um volume formado por 108 imagens no formato *DICOM*, com resolução de 512×512 pixeis cada, obtidas no Portal do Software Público Brasileiro [10]. A exemplo do caso $2D$, é aberta uma nova janela para exibir o resultado do algoritmo. Contudo, no caso $3D$ o usuário pode interagir com o resultado gerado através do *mouse* e do teclado, podendo, por exemplo, efetuar rotação, escala (*zoom*) e translação.

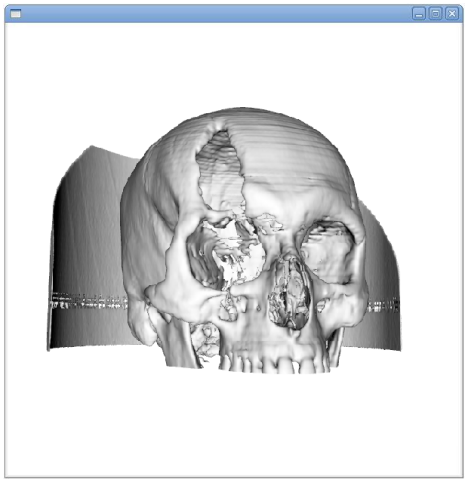

Figura 7. Visualização de isosuperfícies no software *PyImageVis*. Resultado do algoritmo 9

Outra possibilidade de interação com a superfície é usando um plano de corte. Para isso são utilizadas duas telas, uma onde é visualizado o objeto $3D$, o plano de corte e a porção da superfície acima do plano; e, uma segunda tela onde é visualizada uma imagem que representa a interseção do plano de corte com o volume (Figura 8). Esta funcionalidade foi implementada na classe *MarchingCubes* (linhas $6 - 9$). Para tornar esta funcionalidade ativa, basta que o usuário pressione a tecla *"i"* durante a visualização da superfície.

[10]Imagens disponíveis em http://www.softwarepublico.gov.br/dotlrn/clubs/invesalius/file-storage/view/dcm/cranium01.rar

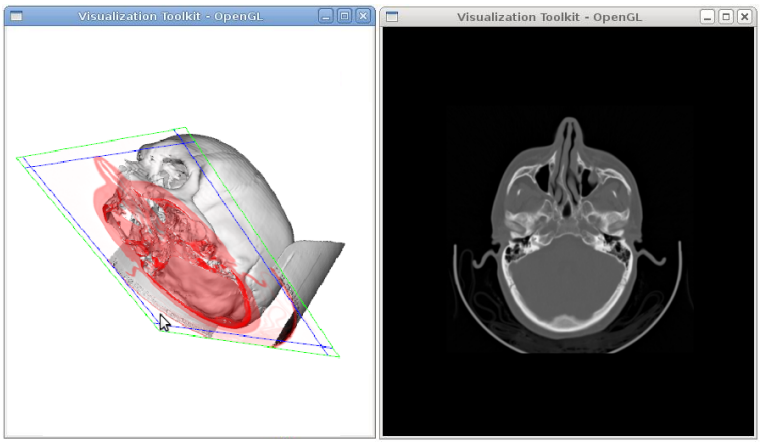

Figura 8. Plano de corte e imagem gerada pela interseção do plano de corte com o volume de imagens (direita).

5.3. *Volume rendering*

Uma outra possibilidade para a visualização de um campo escalar tridimensional é o *volume rendering* ou seja, produzir uma projeção plana diretamente a partir do volume de dados. Para a realização deste teste utilizaremos o mesmo volume de imagens utilizado na seção 5.2.

A exemplo da técnica de Marching Cubes, implementamos uma função *raycast()* e uma classe denominada *RayCast*. A função raycast(), descrita no Algoritmo 10, recebe como argumento uma matriz *NumPy* multidimensional, cria uma instância da classe *RayCast* passando para esta classe como argumento a matriz *NumPy*, e retorna o objeto instanciado. A classe *RayCast* recebe a matriz *NumPy* em sua inicialização, em seguida transforma esta matriz em um objeto *VTK* e executa a computação via técnica de *ray cast* (linhas $3-14$ do Algoritmo 11). As funções de transferência para cor e opacidade, utilizadas nas linhas $3-7$ do Algoritmo 11, foram definidas por tentativa e erro, utilizando as classes/métodos do *VTK*.

```
input : Matriz multidimensional NumPy (data_numpy)
rc = RayCast(data_numpy)
rc.show()
return rc
```

Algoritmo 10: função raycast.

O Algoritmo 12 mostra a sequência de comandos necessários para utilizar o *volume rendering* no *PyImageVis*. A primeira linha faz a leitura das imagens, via função *loadvolume()*, como já explicado acima. O resultado da execução da função *raycast()*, na segunda linha do Algoritmo 12, está mostrado na Figura 9. Para inserir o *volume rendering* no *PyImageVis*, usa-se o mesmo procedimento utilizado no *marching cubes* (seção 5.2).

```
      input : Matriz multidimensional NumPy (dataImporter)
1     dataImporter = numpy2vtk(data_numpy)

2     Volume rendering
3     alphaChannelFunc = vtkPiecewiseFunction()
4     colorFunc = vtkColorTransferFunction()
5     volumeProperty = vtkVolumeProperty()
6     volumeProperty.SetColor(colorFunc)
7     volumeProperty.SetScalarOpacity(alphaChannelFunc)
8     compositeFunction = vtkVolumeRayCastCompositeFunction()
9     mapper = vtkVolumeRayCastMapper()
10    mapper.SetVolumeRayCastFunction(compositeFunction)
11    mapper.SetInputConnection(dataImporter.GetOutputPort())
12    volume = vtkVolume()
13    volume.SetMapper(mapper)
14    volume.SetProperty(volumeProperty)

15    Visualização
16    renderer = vtkRenderer()
17    renderer.AddActor(volume)
18    renWin = vtkRenderWindow()
19    renWin.AddRenderer(renderer)
```

Algoritmo 11: classe RayCast.

```
v = loadvolume( getdir() )
ray = raycast( v )
```

Algoritmo 12: Ray casting

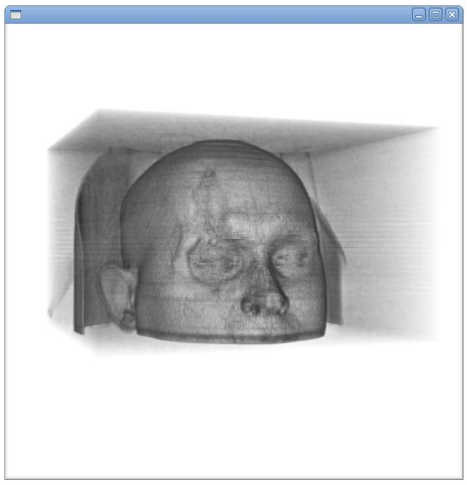

Figura 9. *Volume Rendering* para volume *DICOM* contendo 108 imagens.

6. Discussão

O principal trabalho relacionado ao *PyImageVis* é o software *InVesalius*, sendo também desenvolvido aqui no Brasil em linguagem de programação *Python* (vide seção 2). Estes softwares são relacionados, porém não são concorrentes; uma vez que, enquanto o *InVesalius* é focado para uso em hospitais, o *PyImageVis*, como já foi dito anteriormente na seção 3, foi projetado para ser utilizado em pesquisa. Desta forma, o modelo utilizado no desenvolvimento do *PyImageVis* é constituido por um *framework* que visa a facilidade de integração entre diferentes algoritmos e a reutilização de código.

Do ponto de vista do usuário final, o *InVesalius* tem como foco profissionais da área médica enquanto que o *PyImageVis* tem como objetivo atender pesquisadores em computação e engenharia. Assim, o primeiro vai optar por uma interface que utiliza o *mouse* como principal ferramenta de interação com o software, enquanto que o segundo utiliza chamadas de funções em linha-de-comando para ganhar flexibilidade. As principais vantagens do *PyImageVis*, em relação ao *InVesalius*, são a facilidade de inserção de novas funcionalidades e os recursos para prototipagem de algoritmos via *scripts*.

A Tabela 2 permite analisar o tempo de execução das principais funções do sistema. Nos cinco primeiros métodos citados nesta tabela (numpy2pil,pil2numpy,...,ndimage) foi usada como entrada a imagem em tons-de-cinza mostrada na Figura 5 (640×480 pixeis). Nos métodos para leitura e visualização de imagens $3D$ (loadvolume,marchingcubes,raycast) da Tabela 2, o volume de entrada é o mesmo utilizado para gerar a Figura 7 (108 imagens, cada uma com resolução 512×512, em tons-de-cinza). Destes métodos, a função *loadvolume()* apresentou o maior tempo de execução, 24.493 segundos, por envolver leitura em disco de um volume de dados. As funções de processamento de imagens são obviamente mais rápidas que as funções de visualização volumétrica uma vez que a quantidade de dados e o número de operações envolvidas é bem menor no primeiro caso.

Na versão atual do *PyImageVis*, as funções para edição/visualização de imagens $2D$ pertencem à biblioteca *PIL*. Contudo, escolhemos o formato *NumPy* como padrão para representar as imagens. Assim, quando utilizamos funções de processamento da *PIL*

Funções/Métodos e Tempo de execução	
Descrição	Tempo de Execução (em segundos)
numpy2pil	0.00018
pil2numpy	0.00022
imread	0.010
imshow	0.005
Métodos ndimage	0.08
loadvolume	24.493
marchingcubes	6.412
raycast	0.133

Tabela 2. Tabela com o tempo de CPU das principais funções e métodos do *PyImageVis*.

precisamos executar a função *numpy2pil*() inicialmente e, ao final, retornar ao formato *NumPy* executando *pil2numpy*(), o que aumentam o tempo de CPU. Contudo, pelas linhas $1-2$ da Tabela 2 vemos que este custo é pouco significativo na prática ($T = 0.00040$). Este problema não ocorre para as classes das bibliotecas *VTK*, *Matplotlib* e *Scipy*, pois a primeira é usada apenas para visualização (saída) de resultados e as duas últimas trabalham diretamente com o formato *NumPy*.

7. Conclusões e Trabalhos Futuros

O software *PyImageVis* apresentado neste trabalho, visa propiciar um ambiente de código aberto, na linguagem *Python*, para prototipagem de algoritmos nas áreas de processamento e visualização de imagens médicas. A interface gráfica escolhida, que segue a filosofia do *MatLab*, juntamente com as funcionalidades demonstradas e os respectivos tempos de execução, são indicativos do potencial do sistema desenvolvido.

Como trabalhos futuros, destacamos a necessidade de implementação de funções próprias para edição/visualização de imagens $2D$, bem como leitura de imagens $3D$, evitando assim a utilização da biblioteca *PIL*. A inclusão de novas funcionalidades e avaliação com usuários para verificar a usabilidade do aplicativo são também parte da lista de trabalhos futuros.

8. Agradecimentos

Gostaríamos de agradecer ao Dr. Marcelo Daniel B. Faria pelas radiografias dentárias utilizadas e pelas discussões que ajudaram a enriquecer este trabalho.

Referências

Bankman, I. N., editor (2009). *Handbook of medical imaging: processing and analysis*, volume 1. Academic Press, San Diego, USA, 2nd edition.

Blanchet, G. and Charbit, M. (2006). *Digital Signal and Image Processing Using MATLAB*. ISTE Ltd, London, UK.

Blanchette, J. and Summerfield, M. (2008). *C++ GUI Programming with Qt 4 - The official C++/Qt book* . Prentice Hall Press, USA, 2nd edition.

Booch, G., Rumbaugh, J., and Jacobson, I. (2000). *UML, Guia do Usuário*. Editora Campus, Rio de Janeiro, Brasil.

Bárbara, A. S. and de Carvalho Zavaglia, C. A. (2006). *Processamento de Imagens Médicas Tomográficas para Modelagem Virtual e Física - O software InVesalius*. Doutorado Engenharia Mecânica, Universidade Estadual de Campinas . Faculdade de Engenharia Mecânica.

da Motta Pires, P. S. (2004). *Introdução ao Scilab - Versão 3.0*. Departamento de Engenharia de Computação e Automação da UFRN, Natal, RN. Web: www.dca.ufrn.br/ pmotta/sciport-3.0.pdf.

de Carvalho, M. M., Czekster, R. M., and Manssour, I. H. (2002). *Uma Ferramenta Interativa para Visualização e Extração de Medidas de Imagens Médicas*. SBC - Concurso de Trabalhos de Iniciação Científica (CTIC) Edição 2003, Web: www.sbc.org.br/sbc2003/ctic.html.

Gonzalez, R. and Woods, R. (1992). *Digital Image Processing*. Addison-Wesley Publishing Company, Boston, USA, 2nd edition.

Grayson, J. E. (2000). *Python and Tkinter Programming*. Manning Publications Co., USA, 1st edition.

Hetland, M. L. (2008). *Beginning Python: From Novice to Professional*. Apress, Berkeley, USA, 2nd edition.

Kitware, I. (2010). *VTK User's Guide Version 5*. Kitware, Inc., Columbia, USA, 11th edition.

Lorensen, W. E. and Cline, H. E. (1987). Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21:163–169.

Smart, J., Roebling, R., Zeitlin, V., and Dunn, R. (2011). *wxWidgets 2.8.10: A portable C++ and Python GUI toolkit*. wxWidgets Team, Web: http://docs.wxwidgets.org/2.8/.

Summerfield, M. (2007). *Rapid GUI Programming with Python and Qt: the definitive guide to pyqt programming*. Prentice Hall Press, Upper Saddle River, USA.

Suri, J., Setarhedran, S. K., and Singh, S., editors (2002). *Advanced algorithm approaches to medical image segmentation: state of art applications in cardiology, neurology, mamography and pathology*. Springer-Verlag, London, UK.

Vaseghi, S. (2000). *Advanced signal processing and digital noise reduction*. Wiley Chichester, Welwyn, UK, 2nd edition.

Zhou, J. and Abdel-Mottaleb, M. (2005). A content-based system for human identification based on bitewing dental X-ray images. *Pattern Recognition*, 38(11):2132–2142.