UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

LUCAS MAGARIO HAYASHI
LUIZ HENRIQUE FERNANDES ESTEVOM

PROCESSAMENTO DE VISTAS ORTOGONAIS PARA MODELAGEM 3D AUTOMATIZADA COM PYTHON

PONTA GROSSA
2023

LUCAS MAGARIO HAYASHI

LUIZ HENRIQUE FERNANDES ESTEVOM

PROCESSAMENTO DE VISTAS ORTOGONAIS PARA MODELAGEM 3D AUTOMATIZADA COM PYTHON

Processing Orthogonal Views For Automated 3D Modeling With Python

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia Elétrica da Universidade Tecnológica Federal do Paraná (UTFPR).

Orientador: Prof. Dr. Hugo Valadares Siqueira.

PONTA GROSSA

2023

LUCAS MAGARIO HAYASHI
LUIZ HENRIQUE FERNANDES ESTEVOM

PROCESSAMENTO DE VISTAS ORTOGONAIS PARA MODELAGEM 3D AUTOMATIZADA COM PYTHON

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia Elétrica da Universidade Tecnológica Federal do Paraná (UTFPR).

Data de aprovação: 19/junho/2023

Hugo Valadares Siqueira
Doutorado
Universidade Tecnológica Federal do Paraná

Thiago Antonini Alves
Doutorado
Universidade Tecnológica Federal do Paraná

Murilo Oliveira Leme
Doutorado
Universidade Tecnológica Federal do Paraná

PONTA GROSSA
2023

Tudo que é preciso na vida é
ignorância e confiança; depois, o
seu sucesso será garantido.
- Mark Twain

AGRADECIMENTOS

Certamente estes parágrafos não irão atender a todas as pessoas que fizeram parte dessa importante fase de nossas vidas. Portanto, desde já pedimos desculpas àquelas que não estão presentes entre essas palavras, mas elas podem estar certas que fazem parte do nosso pensamento e de nossa gratidão.

Agradecemos ao nosso orientador Prof. Dr. Hugo Valadares Siqueira, pela sabedoria com que nos guiou nesta trajetória.

Aos nossos amigos que tornaram esta jornada inesquecível.

A Coordenação do Curso e todos os servidores da UTFPR, pela cooperação.

A Gabriela, pelo companheirismo, incentivo e por mostrar através de sua própria vida, que tudo é possível.

A Nataniely, por todo o apoio e incentivo na realização deste trabalho

Gostaríamos de deixar registrado também, o nosso reconhecimento à nossas famílias, pois acreditamos que sem todo o apoio deles, este momento seria impossível.

Enfim, a todos os que por algum motivo contribuíram para a realização desta pesquisa.

RESUMO

Através do estudo de conceitos fundamentais relacionados ao processamento de vistas ortogonais, algoritmos de detecção de borda e modelagem 3D, foi desenvolvido um projeto incipiente de um sistema automatizado em Python capaz de produzir modelos 3D a partir do processamento de vistas ortogonais com o auxílio do software Blender.
O objetivo do algoritmo desenvolvido é fazer a detecção de bordas das vistas ortogonais, o tratamento, a otimização e armazenamento dos dados, além de utilizar um software de computação gráfica para realizar a modelagem e visualização do sólido.
Os resultados obtidos confirmaram a viabilidade e eficácia do sistema para sólidos mais simples, se tornando uma solução eficiente para a modelagem de objetos 3D, tornando o processo mais rápido, fácil e com menor taxa de erros. Além disso, este projeto também visa facilitar a compreensão e visualização de conceitos de desenho técnico para ambientes acadêmicos.

Palavras-chave: Processamento de imagem; Modelagem 3D automatizada; Vistas ortogonais; Automatização com Python.

ABSTRACT

Through the study of fundamental concepts related to the processing of orthogonal views, edge detection algorithms, and 3D modeling, an incipient project was developed for an automated system in Python, capable of producing 3D models from the processing of orthogonal views with the help of the software Blender.
The objective of the developed algorithm is to detect edges of orthogonal views, treat, optimize, and store data, in addition to using computer graphics software to model and visualize the solid.
The results obtained confirmed the viability and effectiveness of the system for simple solids, becoming an efficient solution for modeling 3D objects, making the process faster, easier, and with a lower error rate. In addition, this project also aims to facilitate the understanding and visualization of technical drawing concepts for academic environments.

Keywords: Image processing; Automated 3D modeling; Orthogonal views; Automation with Python.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

CAD	Computer Aided Design
JPEG	Joint Photographic Experts Group
PDF	Portable Document Format
PNG	Portable Network Graphic
px.	Pixels
RGB	Red Green Blue
txt.	Text
UTFPR	Universidade Tecnológica Federal do Paraná

SUMÁRIO

1. INTRODUÇÃO

Não se pode considerar a tecnologia moderna um simples estudo da técnica, mas sim uma junção entre esta e a ciência, criando um conceito de teoria e praticidade. Deste modo, é possível dizer que a tecnologia surge para facilitar a vida humana, criando novos caminhos para processos já existentes ou inexistentes (MIRANDA, 2002).

Em 1962 a tese de Ivan Sutherland foi publicada, considerada como uma das mais importantes na história da computação gráfica: “Sketchpad – A Man-Machine Graphical Communication System”, a qual introduz uma nova área da tecnologia e estruturação de dados, a computação gráfica (AZEVEDO; CONCI, 2003).

Com o avanço da tecnologia, e o desenvolvimento dos softwares CAD, a computação gráfica evoluiu de forma exponencial, não se limitando somente ao campo de duas dimensões (2D) e iniciando um novo movimento dentro da computação gráfica tridimensional (CONRADO; 2019).

A modelagem em 3 dimensões (3D) é uma área que está se expandindo cada vez mais e com o avanço da tecnologia, a modelagem não atua mais somente na indústria, mas também em outros ambientes como no entretenimento, projetos arquitetônicos, na medicina e em representações e simulações para pesquisa. (AZEVEDO; CONCI, 2003).

Como acontece todos os dias, sempre que uma tecnologia se popularizar e se mostrar amplamente necessária, surge a necessidade por uma busca de torná-la cada vez mais prática, rápida e com menores custos (CARIAS, 2020).

Um modo popular e que consegue alcançar os três pontos descritos anteriormente é a automatização de processos. O termo Automação pode ser definido como uma série de ações que não necessitam de interferência humana A automação se baseia na dinamização e controle de processos, englobando uma verificação de seu próprio funcionamento (SILVA, 2007).

A geração automática de sólidos 3D tem sido um desafio importante na área da computação gráfica tendo em vista que, atualmente, a modelagem manual de sólidos 3D necessita de tempo e de habilidades especializadas.

Indra Nooyi mostra a importância de aceitar e incorporar o avanço da tecnologia ao trabalho realizado pelo ser humano, pois esta altera de forma drástica a maneira como se vive e trabalha (NOOYI, 2003).

Nos apoiando nessa ideia e pensando em uma possível facilitação do aprendizado na área, o estudo deste projeto será a automatização da modelagem 3D a partir de desenhos técnicos já existentes, visando primeiramente a modelagem de sólidos.O projeto seguirá com foco na automatização do processo.

1.1 TEMA DA PESQUISA

Automatização do processo de modelagem de um sólido 3D a partir de suas vistas ortogonais. Esse processo geralmente implica na necessidade de um operador humano para realizar a modelagem do sólido, seja este destinado à impressão, torneamento na indústria, ou até mesmo para ser utilizado em ambientes virtuais.

1.2 DELIMITAÇÃO DO TEMA

O tema central desta investigação é o estudo e criação de um código para modelagem de sólidos simples, a partir da interseção de suas extrusões, com possíveis aplicações para salas de aula, indústria e criação de ambientes virtuais (jogos, arquitetura, design de produtos etc…).

1.3 PROBLEMA

A modelagem 3D atualmente necessita de um operador humano para realizar a construção do sólido com base em uma série de desenhos 2D. Essa etapa acaba por custar dinheiro e tempo às empresas que necessitam de uma modelagem mais rápida de alguns sólidos mais simples. Em cursos de engenharia, em disciplinas como desenho técnico, existe uma certa dificuldade em compreender como se localizam os desenhos das vistas ortogonais para a modelagem dos sólidos 3D. Este projeto visa criar uma alternativa para uma visualização facilitada.

1.4 HIPÓTESE/PREMISSA

O estudo da automatização do processo de modelagem de sólidos 3D a partir de suas vistas ortogonais pode oferecer uma solução viável e eficiente para superar a dependência de operadores humanos dentro das metas alcançadas. Ao desenvolver um código capaz de gerar modelos 3D a partir da interseção de extrusões de sólidos simples, com aplicações em salas de aula, indústria e ambientes virtuais, é possível agilizar a criação de sólidos e reduzir custos para empresas, além de facilitar a compreensão e visualização dos conceitos de desenho técnico para estudantes. Porém, como será mostrado no decorrer do trabalho, este projeto tem como foco o início do estudo da área com uma solução inicial e limitada.

1.5 OBJETIVOS

1.5.1 Objetivo geral

O objetivo geral deste trabalho é o desenvolvimento de um código em Python que realize a modelagem automatizada de um sólido 3D a partir de desenhos técnicos de suas vistas ortogonais.

1.5.2 Objetivos específicos

O projeto teve como seus objetivos específicos:

- Conceber uma solução inicial para o problema, criando uma alternativa viável para parte dos sólidos;
- Iniciar um estudo na área para ser explorado futuramente;
- Criar um método que fosse de fácil manipulação e compreensão;

1.6 JUSTIFICATIVA

A justificativa do projeto é fundamentada no estudo inicial de duas premissas: a primeira é a de otimizar e agilizar o processo de criação de modelos tridimensionais a partir de desenhos em vistas ortogonais, de modo a diminuir a intervenção manual no processo dentro de empresas e indústrias, porém, tendo como um primeiro estudo de uma solução, a precisão das medidas não foi uma prioridade. Já a segunda se deve a facilitação, a compreensão da matéria e a localização das vistas ortogonais para alunos e professores dentro do ambiente educacional, visando principalmente a matéria de desenho técnico.

O projeto irá seguir como uma primeira alternativa para resolução do problema a ser estudado, podendo não alcançar uma total solução mas abrindo uma possibilidade para novas versões e melhorias futuras.

1.7 ORGANIZAÇÃO DO TRABALHO

O trabalho está dividido em cinco capítulos, cada qual abordando um tema necessário para o desenvolvimento e compreensão do projeto, sendo eles:

- O primeiro capítulo é uma introdução geral ao tema, apresentando o escopo geral do projeto e seus objetivos;
- O segundo capítulo apresenta uma fundamentação teórica com os principais conceitos utilizados no trabalho, como a linguagem de programação utilizada, as vistas ortogonais do desenho técnico, ferramentas de modelagem, e também, o programa de modelagem 3D utilizado.
- No terceiro capítulo será abordada a solução para o problema, como foi seu desenvolvimento e os processos que a formaram, como o pré-processamento das imagens, a coleta de dados, o desenvolvimento dos códigos e a modelagem automatizada;
- O quarto capítulo apresenta os resultados obtidos pelo projeto, os sólidos gerados, a avaliação crítica dos resultados, precisão, eficiência e eventuais limitações. Além de uma comparação com abordagens já existentes;
- O quinto capítulo será uma conclusão do trabalho, abordando sobre os objetivos, resultados e trabalhos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

Um modelo 3D é uma representação matemática de superfícies tridimensionais e sua modelação pode ser descrita como o processo de definição da forma de um objeto através da manipulação de sua geometria (LOPES, 2023).

A geometria 3D, é uma ramificação da matemática que estuda as propriedades e relações dos objetos em um espaço tridimensional. Ela abrange uma ampla gama de técnicas e tópicos essenciais para a modelagem, análise e visualização de objetos tridimensionais. Os principais conceitos que envolvem a área de estudo da geometria 3D são (CONRADO, 2019):

- Pontos, Linhas e planos: estes são elementos representados no espaço tridimensional. Um ponto é especificado pelas coordenadas *x,y* e *z*, uma linha 3D é especificado pela ligação entre dois pontos e um plano é determinado pela ligação de diversos pontos não colineares em um espaço;
- Vetores e operações vetoriais: são utilizados na geometria 3D para representar deslocamentos, direções, magnitudes e descrever movimentos e transformações geométricas;
- Poliedros e sólidos: São objetos tridimensionais limitados por faces planas como cubos, pirâmides e tetraedros;
- Curvas e superfícies: As curvas e as superfícies são definidas por um conjunto de equações paramétricas que definem a posição dos seus pontos;
- Transformações geométricas: São operações que modificam a posição, orientação e formato de objetos 3D.

Com diversas aplicações, a geometria 3D permite a representação e a análise de objetos tridimensionais, auxiliando na simulação de projetos e ambientes virtuais, criação de modelos físicos e digitais e na visualização de estruturas complexas (CONRADO, 2019).

2.1 CONCEITO DE MODELAGEM

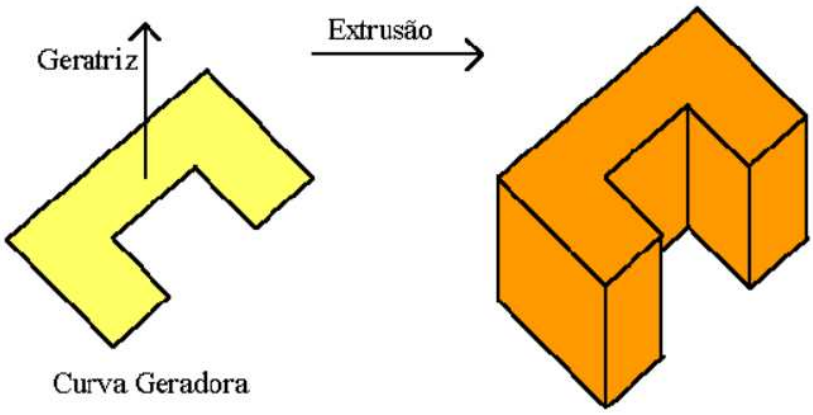
A modelagem pode ser considerada como o processo de criação ou representação de um modelo. Este pode simular um sistema real ou um objeto abstrato (LOPES, 2023).

A modelagem 3D é uma forma específica que se concentra na criação e representação de objetos tridimensionais. Diferentemente do caso 2D, em que os objetos são representados em duas dimensões (largura e altura), a 3D adiciona a dimensão de profundidade, permitindo a criação de sólidos com volume (LOPES, 2023).

Existem diversas técnicas que permitem modelar e criar modelos 3D a partir de vistas ortogonais, por exemplo:

- Modelagem por extrusão: esta técnica envolve a criação de uma forma 2D de uma vista ortogonal, e em seguida a sua extrusão ao longo do eixo Z, criando assim um objeto 3D como mostra a Figura 1;

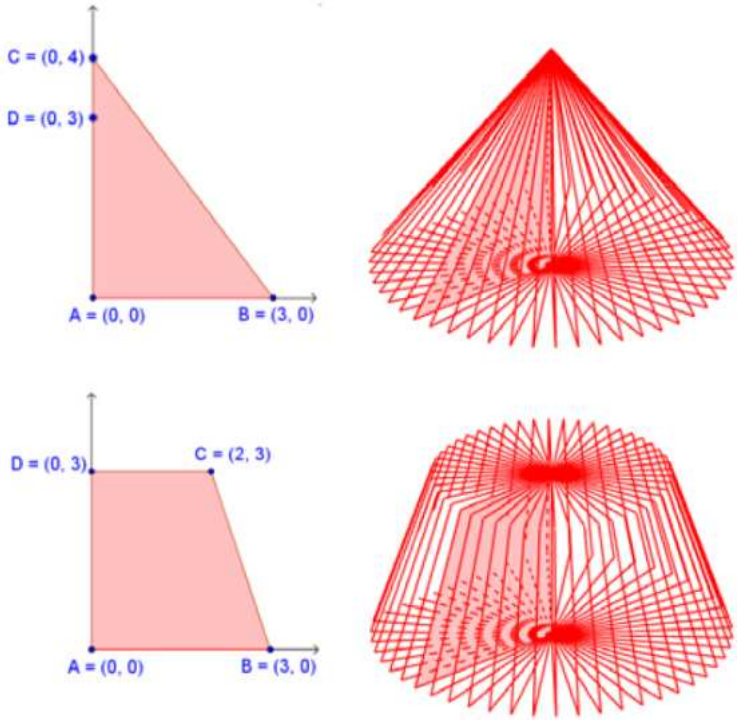
FIGURA 1 - EXEMPLO DE EXTRUSÃO



FONTE: AZEVEDO; CONCI, (2003)

- Modelagem por revolução: Muito parecido com a modelagem por extrusão, esta consiste na criação da forma 2D, em seguida, a sua rotação em torno de um eixo para criar o objeto 3D, como exemplificado na Figura 2;

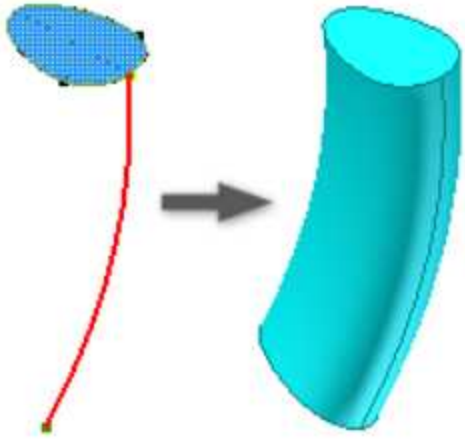
FIGURA 2 - MODELAGEM POR REVOLUÇÃO



FONTE: DANTAS; MATHIAS, (2016)

- Modelagem por varredura: assim como as citadas anteriormente, envolve a criação de uma forma 2D para, em seguida, varrer esta forma ao longo de um eixo para criar o objeto 3D, como exemplifica a Figura 3;

FIGURA 3 - MODELAGEM POR VARREDURA

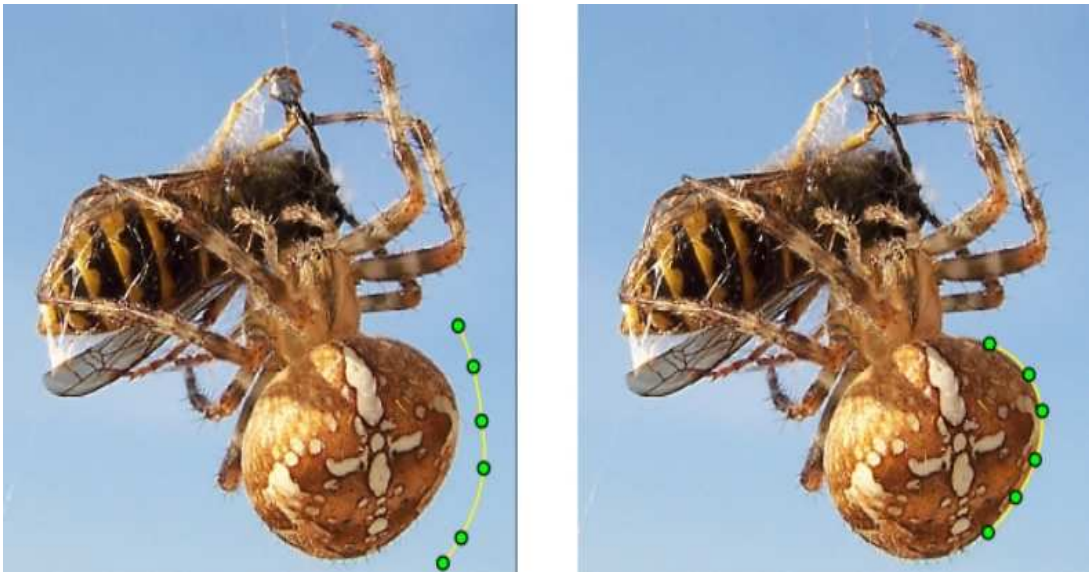


FONTE: AUTODESK (2023)

- Modelagem por contorno: A modelagem por contorno consiste na criação de vistas ortogonais em linhas e curvas. Estas curvas, posteriormente, são

conectadas para formar uma superfície 3D de um objeto. É frequentemente usada em modelagens que são baseadas em imagens, ao traçar contornos de um objeto em diferentes vistas e criar uma malha 3D com base nesses contornos;

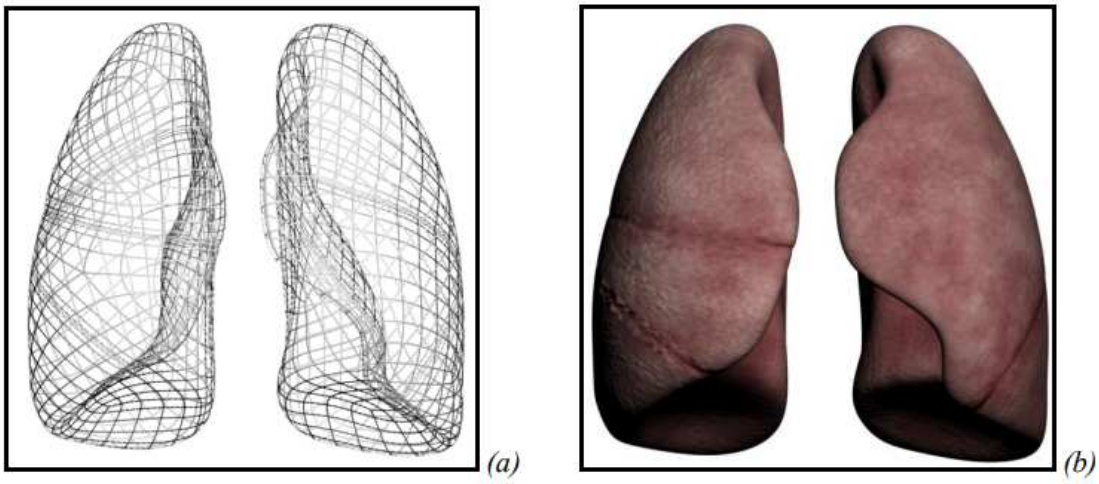
FIGURA 4 - MODELAGEM POR VOXELIZAÇÃO



FONTE: (SNAKE-CONTOUR-EXAMPLE, 2005)

- Modelagem por voxelização: trata-se de uma técnica que envolve a conversão de uma representação 2D em uma vista ortogonal em um objeto 3D. Cada elemento 3D, também chamado de voxel, corresponde a um elemento 2D na vista. Esta técnica cria uma grade tridimensional de voxels com base nas informações da vista ortogonal.

FIGURA 4 - MODELAGEM POR VOXELIZAÇÃO



FONTE: Andrade P. H. A., Vieira, J. W., Oliveira V. R. S., Veloso R. J. B., Lima F. R. A. (2020)

Após a criação do modelo em três dimensões, é possível texturizar a sua superfície ou até fazer animações com ele, para dar um aspecto mais realista ao objeto (LOPES, 2023).

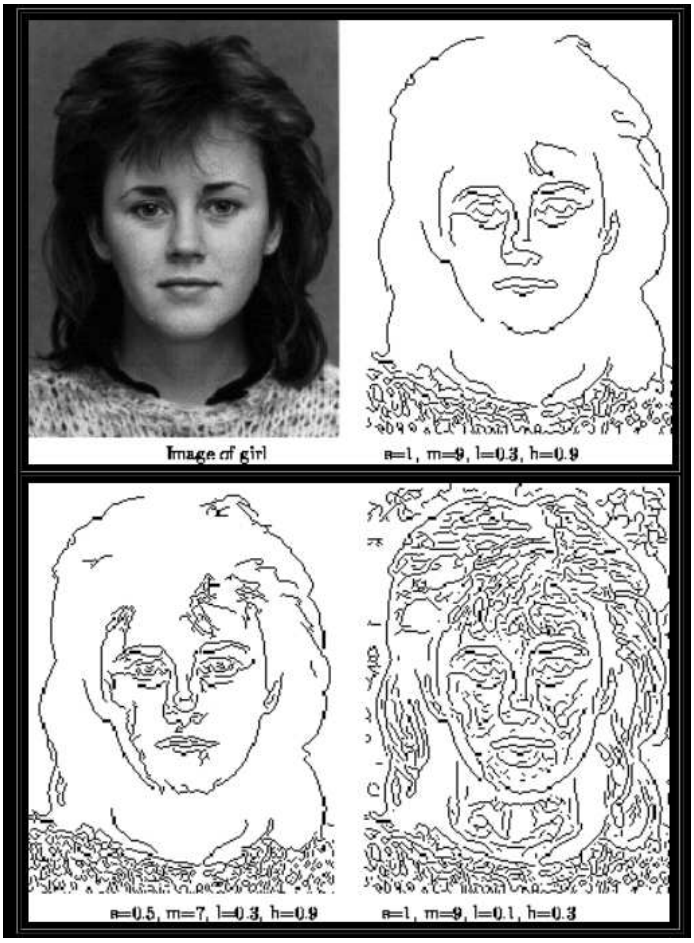
Para a realização deste trabalho foi utilizado o conceito da Modelação Poligonal, que consiste em pontos ligados entre si através de arestas, com o objetivo de delimitar faces e formar uma malha poligonal (NOVAIS, 2015)

A etapa de modelação ocorre depois do software realizar a detecção das coordenadas dos vértices das imagens das vistas ortogonais. Para este processo de detecção das bordas utilizamos o algoritmo de Canny (NOVAIS, 2015).

2.2 ALGORITMO DE CANNY

O algoritmo de Canny é um filtro de convolução que localiza as bordas e suaviza os ruídos ao combinar um operador diferencial com um filtro Gaussiano, como mostra a Figura 4, representada por imagens com diferentes escalas e limiarizações utilizando o operador de Canny (BUENO, 2000):

FIGURA 4 - DETECÇÃO DE BORDAS PELO ALGORITMO DE CANNY



FONTE: BUENO, (2000)

Em figuras 2D, as regiões de baixa frequência são definidas por áreas sem variações dos níveis de cinza, enquanto as áreas com maiores variações na cor são caracterizadas por maiores frequências. Uma borda ou aresta é definida por uma descontinuidade significativa (alta frequência) na intensidade dos tons de cinza da imagem (GONZALEZ; WOODS, 1992).

O algoritmo é um operador que detecta a variação nestas frequências, sendo definido pela soma de quatro termos exponenciais e aproximado pela primeira derivada de uma função Gaussiana ($G(x)$) expressa pela Equação 1:

Desta forma temos a função Gaussiana expressa pela função $G(x)$:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-x^2}{2\sigma^2}} \quad (1)$$

Sua primeira derivada é expressa pela Equação 2:

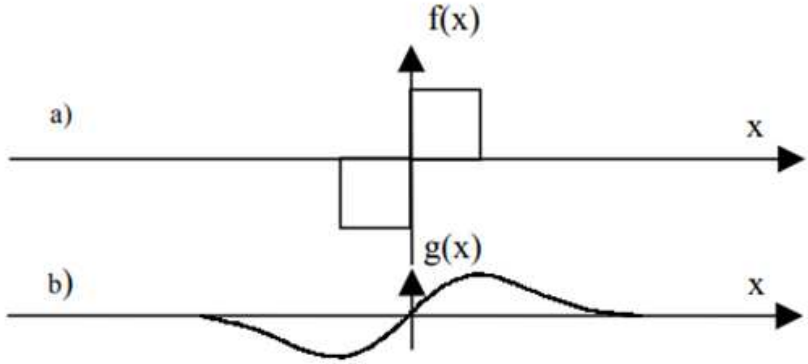
$$G'(x) = \frac{-x}{\sqrt{2\pi\sigma^3}} e^{\frac{-x^2}{2\sigma^2}} \quad (2)$$

De acordo com Canny, um bom operador deve atender a três requisitos:

- Taxa de erro: o operador deve possuir uma baixa taxa de erro ao tentar identificar as verdadeiras bordas da imagem, ou seja, ela deve maximizar a razão sinal/ruído;
- As distâncias entre os pontos da borda extraídos pelo operador e as bordas reais do objeto devem ser minimizadas;
- Resposta mínima: o operador não deve produzir respostas muito largas ou múltiplas respostas para uma borda. Isto ajuda a preservar a nitidez e consequentemente a identificação das bordas.

Canny diz em seu trabalho que as bordas devem ser consideradas como um máximo local. Por este motivo o filtro “a” da figura 5 possui um desempenho inferior ao da figura “b”, pois o seu resultado apresenta mais de um máximo na borda (CANNY, 1986)

FIGURA 5 - MÁXIMOS DE BORDA



BUENO, Marcelo Lemes (2000)

Um bom operador envolve a maximização dos 3 critérios citados anteriormente, sendo equivalente à maximização do produto do SNR (razão sinal/ruído) e a equação de localização da borda, como expressa a equação 3:

$$\left(\frac{\left|\int_{-W}^{W} G(-x)f(x)\,dx\right|}{n_0\sqrt{\int_{-W}^{W} f^2(x)\,dx}}\right)\cdot\left(\frac{\left|\int_{-W}^{W} G'(-x)f'(x)\,dx\right|}{n_0\sqrt{\int_{-W}^{W} f'^2(x)\,dx}}\right) \quad (3)$$

em que:

- $F(x)$ = Resposta de impulso do filtro definido no intervalo [-w; w];
- $G(x)$ = Borda unidimensional
- n_0 = Quantificação do ruído na imagem

A expressão usada para encontrar os máximos adjacentes na resposta do filtro $f(x)$ acima, é dada pela Equação 4:

$$x_{max} = 2\pi \left(\frac{\int_{-\infty}^{+\infty} f^2(x)\,dx}{\int_{-\infty}^{+\infty} f'^2(x)\,dx} \right)^{1/2} \quad (4)$$

Para encontrar resultados ainda mais assertivos, além de tentar maximizar o valor do filtro *f*(*x*), também deve-se garantir que o valor de X seja maior possível, para aumentar a chance de separar os máximos verdadeiros dos máximos falsos no filtro *f*(*x*).

O funcionamento do algoritmo de Canny pode ser descrito em 4 etapas:

- Redução de ruído: Com o objetivo de diminuir ruídos na imagem, é feito uma convolução na imagem através de uma máscara Gaussiana;
- Detecção das bordas: são usadas 4 máscaras para determinar a direção das bordas;
- Armazenamento de dois mapas de gradientes. O mapa de intensidade e o mapa de sentido do gradiente;
- Com a análise dos mapas são determinadas as bordas, em que os gradientes de maior intensidade são definidos como bordas.

2.3 BIBLIOTECAS DE ALGORITMOS

Após a etapa de leitura das imagens das vistas ortogonais, foi necessário fazer o tratamento dos vértices das imagens de forma a deixá-las mais nítidas e precisas, fazer o tratamento dos pontos de modo que os conjuntos representassem pelo menos um eixo de suas vistas ortogonais e armazenar as informações. Para todas essas etapas, foram utilizadas diferentes bibliotecas de algoritmos.

2.3.1 OpenCV ou CV2:

O OpenCV ou “Open Source Computer Vision Library” é uma das bibliotecas mais populares para o processamento de imagem e visão computacional. Ela foi criada inicialmente pela Intel no ano de 2000, desenvolvida na linguagem de programação C++ e possui diversos módulos de visão computacional como

reconhecimento de objetos, filtros de imagem e análise estrutural utilizados neste trabalho (ANTONELLO, 2017).

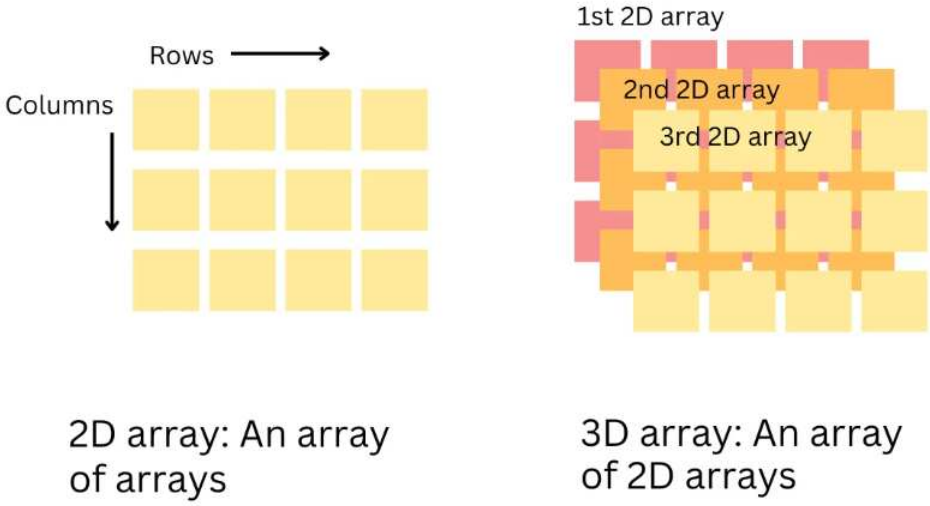
Esta linguagem de programação possui ligações com diversas outras linguagens, neste trabalho foi utilizada a conexão com o Python. O seu uso foi muito importante para o desenvolvimento deste trabalho pois:

- Esta biblioteca consegue ler e escrever imagens em diversos formatos, como JPEG, PNG e PDF, além de também oferece suporte a várias profundidades de bits e canais de cores;
- Ao possuir uma ampla gama de funções para processamento de imagens, nos permitiu realizar operações básicas e avançadas nesta área;
- Oferece algoritmos de detecção e rastreamento de objetos ao detectar bordas, contornos e entre outros.

2.3.2 Numpy:

A biblioteca Numpy ou Numerical Python tem como objetivo realizar operações em arrays multidimensionais. A Figura 6 exemplifica este processo.

FIGURA 6 - ARRAYS MULTIDIMENSIONAIS



FONTE: DHONDGE, (2023)

Um array é uma estrutura multidimensional que permite armazenar dados na memória e localizar por meio de um sistema de indexação. O NumPy Python

denomina o array como Ndarray. A Figura 7 apresenta um exemplo de uso da estrutura, onde “*arr”* é o nome do array e “*np.array”* é a função da biblioteca numpy.

FIGURA 7 - EXEMPLO DE ARRAY 3D

```
arr = numpy.array([2D_array1 values],[2D_array2 values]...)
```

```
array_2 = np.array([[[1,2,3],[3,4,5]],
                    [[6,7,8],[9,8,7]],
                    [[6,5,4],[3,2,1]]])

print("Output")
print(array_2)
```

FONTE: DHONDGE, (2023)

A construção do Numpy é baseada na estrutura de dados do Ndarray. Com isso, esta biblioteca oferece uma ampla gama de funcionalidades como manipulações de dados, geração de subconjuntos e filtros, tratamento de dados entre outros (MULINARI, 2018).

2.3.3 Matplotlib

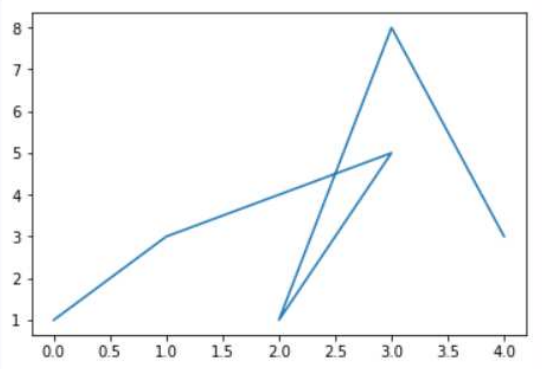
O Matplotlib é uma biblioteca de visualização de dados em Python. Esta biblioteca suporta uma grande variedade de gráficos interativos, animados, estáticos e também a personalização dos mesmos. Além disso, permite criar múltiplos gráficos em uma figura que podem ser exportados em vários formatos. Uma outra vantagem é a integração, pois ela pode ser combinada com outras bibliotecas do ecossistema Python, como o Numpy e o Pandas, para análise e visualização de dados de forma mais eficiente.

Neste trabalho foi utilizado o módulo Pyplot da biblioteca Matplotlib. O módulo Pyplot fornece uma interface semelhante ao software MatLab com o objetivo de criar gráficos. Cada função do Pyplot afeta a imagem de alguma maneira, como a função ‘plot()’, que é usada para criar gráficos de linha e de dispersão, a função ‘bar()’,

usada para criar gráficos de barras e a função ‘show()’, utilizada para exibir o gráfico criado. A Figura 8 mostra um exemplo de aplicação de tais funcionalidades.

FIGURA 8 - EXEMPLO DE GRÁFICO SIMPLES COM A BIBLIOTECA MATPLOTLIB

```
import matplotlib.pyplot as plt

x = [0, 1, 3, 2, 3, 4]
y = [1, 3, 5, 1, 8, 3]

plt.figure()

plt.plot(x, y)

plt.show()
```



FONTE: ELENO (2023)

2.3.4 Biblioteca Scipy.Spatial

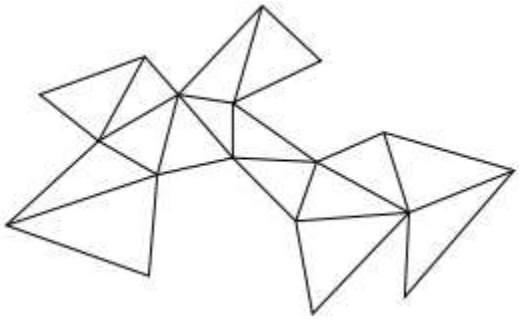
A biblioteca Scipy possui ferramentas dedicadas à computação científica. Podem ser aplicados em projetos que necessitem de interpolações, otimizações, funções especiais, processamentos de imagem, entre outros. Este é o principal pacote de rotinas científicas em Python e funciona em conjunto com a biblioteca Numpy ao trabalhar de forma eficiente com matrizes.

Spatial data, em português dados espaciais, se referem a dados que são representados em um espaço Geométrico. O módulo scipy.spatial contêm ferramentas para manipular estes dados

O módulo Spatial da biblioteca Scipy utilizada neste projeto é destinado ao cálculo de triangulações, diagramas de Voronoi e cascas convexas (Convex Hull) de um conjunto de pontos (SPATIAL, 2018)

A triangulação possui dois conceitos. A triangulação de um polígono é baseada no conceito de divisão de um polígono em vários triângulos para o cálculo de sua área. A triangulação por pontos se baseia na criação de uma superfície composta por triângulos, em que cada um de seus pontos corresponde a pelo menos um vértice de sua superfície, como mostra a Figura 9 (FERNANDES, 2023).

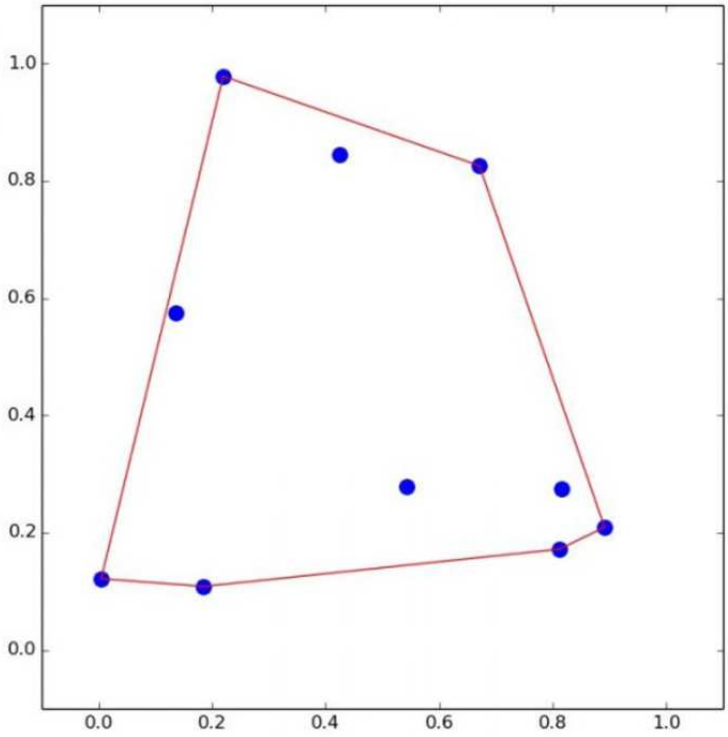
FIGURA 9 - TRIANGULAÇÃO DE UM POLÍGONO



FONTE: FERNANDES, (2023)

O convex Hull se baseia em um conceito da geometria computacional onde o objetivo é encontrar o menor conjunto de pontos em um espaço que abrange todos os outros pontos em um plano, conforme a Figura 10 (SCIPY, 2021):

FIGURA 10 - EXEMPLO DE CONVEX HULL



FONTE: (SCIPY, 2021)

O exemplo acima consiste em um plano 2D, porém neste projeto foi expandido para um plano 3D.

2.4 DIAGRAMAS DE VORONOI

O Diagrama de Voronoi é uma ferramenta utilizada para resolver problemas que envolvem um conceito de proximidade em um plano. Ela divide um espaço em regiões com base na proximidade de pontos de dados específicos (LIMA, 2021)

Eles são frequentemente utilizados para análise espacial, ao fornecer informações sobre a distribuição dos dados e são importantes para a visualização de ao ajudar a identificar agrupamentos, fronteiras e relacionamentos entre pontos.

Além disso, esta ferramenta possui aplicação em áreas de geometria computacional, roteamento de redes, design de jogos entre muitas outras aplicações ao fornecer uma abordagem eficiente e intuitiva para a análise e representação de dados baseada em proximidade (LIMA, 2021).

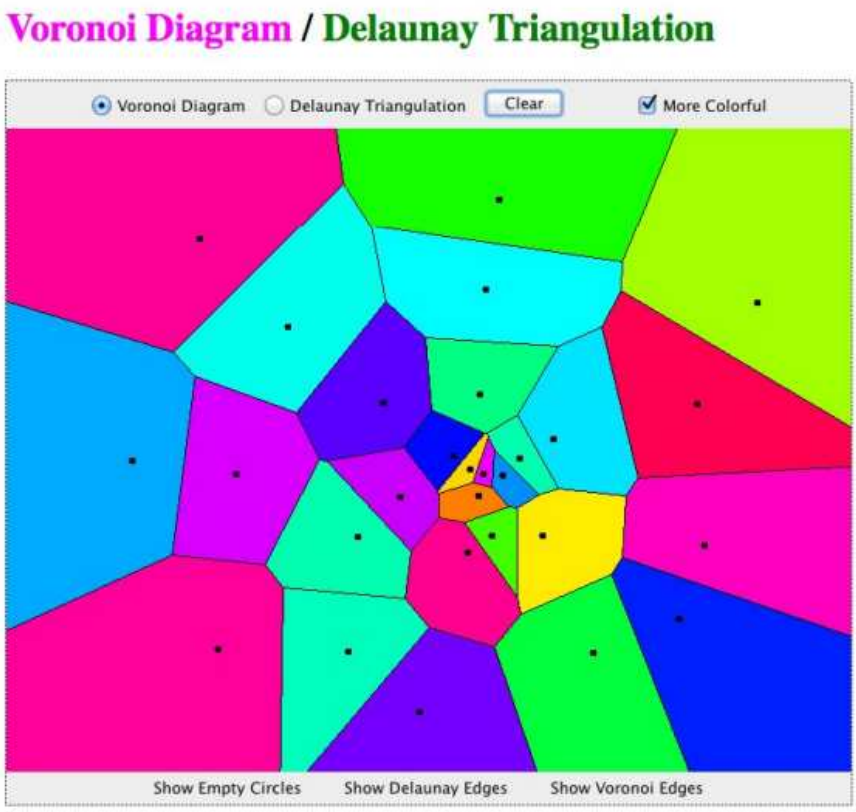
Em um diagrama de Voronoi, cada ponto de dados é chamado de gerador ou semente e é no entorno destes que os polígonos de Voronoi são construídos de

modo que cada um contenha todos os pontos do espaço mais próximos a um gerador do que outro gerador.

Para criar um diagrama de Voronoi são seguidos os seguintes passos (LIMA, 2021)

- Definição dos geradores: os pontos que podem representar desde coordenadas até um gráfico de dispersão, são posicionados em um espaço;
- Cálculo dos limites dos polígonos: ao calcular as linhas perpendiculares que dividem o espaço entre cada par de pontos adjacentes, um polígono é criado ao considerar que ele contém todos os pontos próximos a ele do que qualquer outro gerador;
- Preenchimento dos polígonos: ao definir os limites dos polígonos, é feito um preenchimento com uma cor para destacar a sua superfície. Um exemplo de sua aplicação está na Figura 11:

FIGURA 11 - DIAGRAMA DE VORONOI



FONTE: MONTENEGRO, (2023)

Após o armazenamento das informações, é necessário um software para visualizar e realizar a modelagem 3D a partir dos pontos definidos.

2.5 SOFTWARE BLENDER

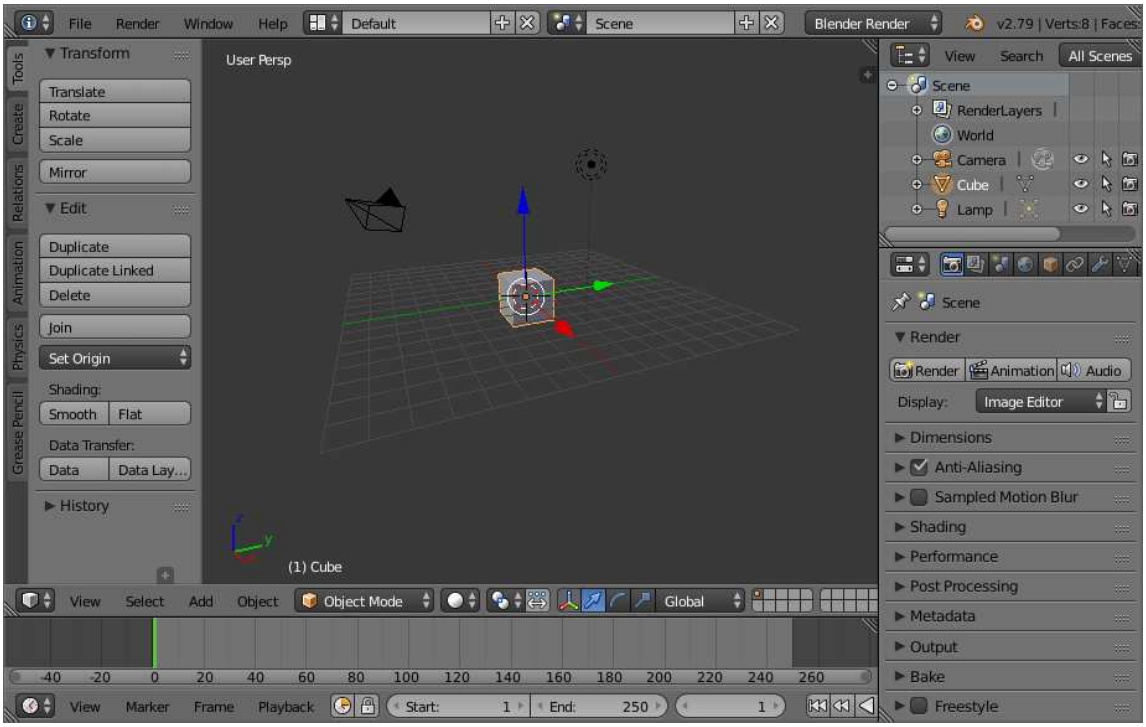
Para a visualização da figura 3D gerada pelo código e para a automatização da modelagem foi escolhido o software Blender. Um software de criação 3D de código aberto, gratuito e desenvolvido pela empresa Blender Foundation (BLENDER, 2023)

Tal escolha se baseou em alguns pontos, tais como:

- A possibilidade de uma integração e comunicação direta via Python;
- A facilidade da implementação de seus comandos de modelagem;
- Já possuir um meio de automatização;
- O software é gratuito.

Este programa possui um completo pacote de ferramentas para a criação de projetos 3D. O fato de sua interface ser bem flexível, baseada e controlada através do Python, facilitou a sua integração com o projeto desenvolvido neste trabalho, com a utilização de scripts na mesma linguagem de programação. A sua interface com o usuário é como na Figura 12:

FIGURA 12 - INTERFACE DO SOFTWARE BLENDER



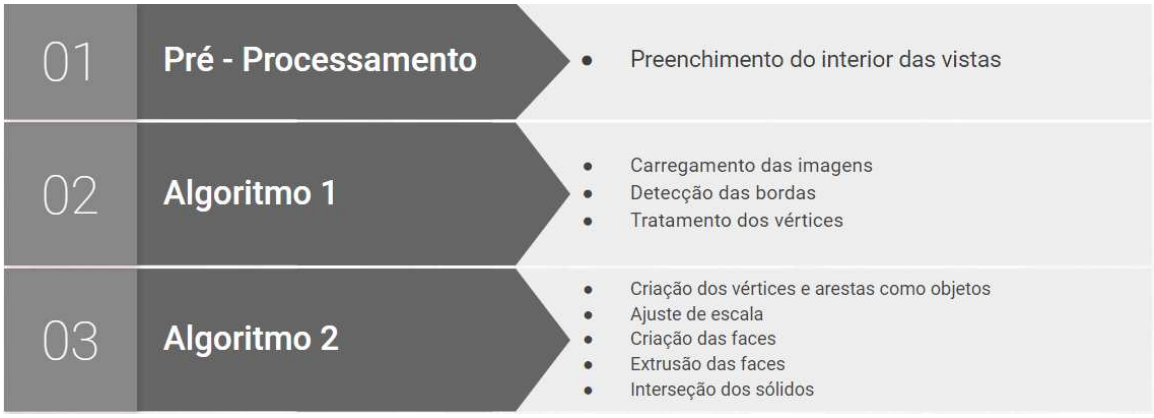
FONTE: BLENDER MANUAL, (2023)

O código escrito foi executado dentro da guia “Scripts” do Blender. Essa guia do programa serve justamente para a automatização de tarefas, mas também pode ser usada para a criação de ferramentas personalizadas, animações e interação com outros softwares. O programa já oferece uma série de algoritmos pré-definidos ao usuário para questões mais recorrentes como exportação de arquivos, mapeamento UV e operadores de interface (BLENDER, 2023).

3. METODOLOGIA

Para a automação do processo de modelagem 3D de um sólido através do processamento de vistas ortogonais, será necessária a criação de dois algoritmos, um para a identificação e tratamento das imagens e outro para a automatização da modelagem a partir das informações adquiridas pelo primeiro código, além de um pré - processamento das imagens. Como pode ser visto no diagrama a seguir:

FIGURA 13 - DIAGRAMA DE FUNCIONAMENTO DO PROJETO



FONTE: AUTORIA PRÓPRIA

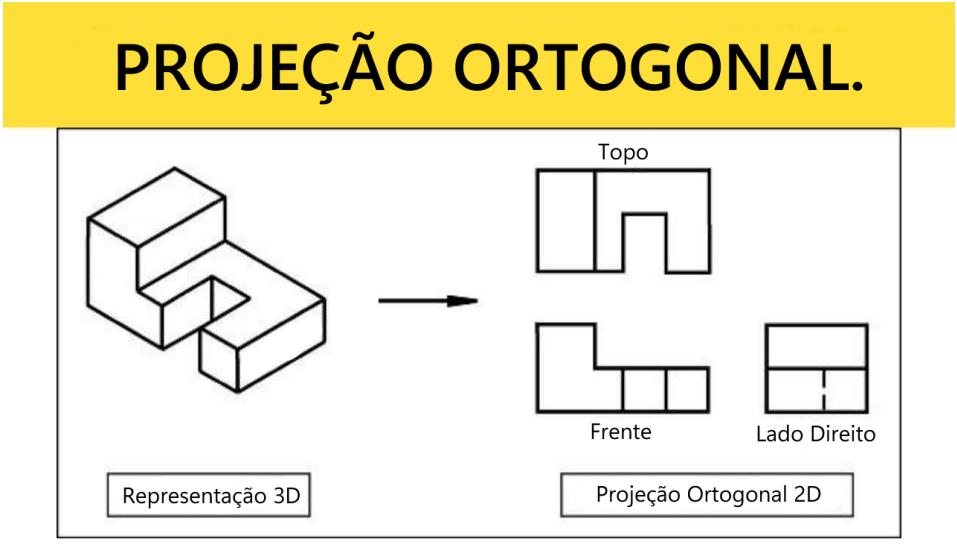
Os dois algoritmos foram programados em linguagem Python e são indispensáveis para o funcionamento da automatização. Enquanto o primeiro pode ser executado em qualquer ambiente Python, o segundo precisa ser executado dentro do ambiente do software Blender, ou executado de uma maneira em que o código consiga acessar o programa que esteja aberto em segundo plano. No caso deste projeto, foi escolhida a execução do código dentro do software Blender.

A análise dos algoritmos foi dividida por etapas de operação, como será abordado a seguir.

3.1 COLETA DE DADOS E PRÉ PROCESSAMENTO

Baseado no sólido de CIVIL SEEK, (2020). Que pode ser visto na Figura 13 :

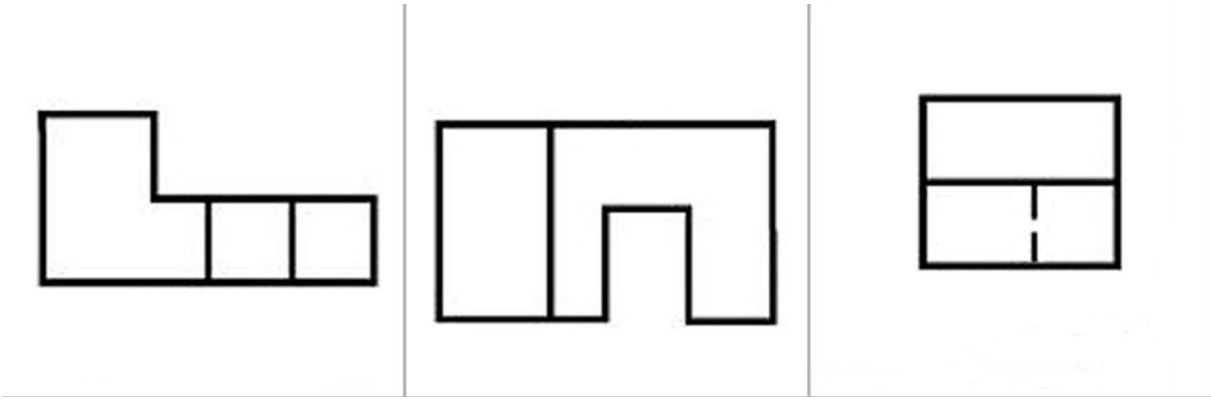
FIGURA 14 - SÓLIDO E VISTAS ORTOGONAIS



FONTE: CIVIL SEEK (2020)

As vistas foram redesenhadas, cada qual em uma imagem separada e de mesmas dimensões (1000x1000px). Foi necessário manter as vistas com a mesma centralização na medida do possível, como pode ser visto nas imagens da Figura 14:

FIGURA 15 - VISTAS REDESENHADAS



FONTE: AUTORIA PRÓPRIA

Porém, foi percebido que para a identificação das bordas mais externas da vista, o preenchimento do interior do desenho seria uma alternativa viável, visto que as bordas mais internas seriam excluídas durante a leitura e o código reconheceria apenas as bordas da silhueta do desenho.

Assim, o modelo das vistas finais será como a Figura 15:

FIGURA 16 - VISTAS PREENCHIDAS

FONTE: AUTORIA PRÓPRIA

3.2 CARREGAMENTO DAS IMAGENS E DETECÇÃO DAS BORDAS.

Iniciando o primeiro código e após a importação das bibliotecas necessárias, a primeira etapa foi o carregamento das imagens das vistas ortogonais previamente discutidas.

Para o carregamento das imagens no código, foi utilizado o comando `cv2.imread` da biblioteca OpenCV (cv2). Como mostrado a seguir:

FIGURA 17 - LINHAS DE CARREGAMENTO DE IMAGEM

```
frontal = cv2.imread("vista_frontal_full.png",0)
lateral = cv2.imread("vista_lateral_full.png",0)
superior = cv2.imread("vista_superior_full.png",0)
```

FONTE: AUTORIA PRÓPRIA

Este comando funciona seguindo dois argumentos principais: o primeiro indica o caminho do arquivo de imagem que deseja ser lido, (exemplo utilizado no código: “vista_frontal_full.png”); o segundo indica o modo de leitura da imagem, no modo utilizado acima (0), a imagem será lida em escala de cinza.

No projeto foram utilizadas imagens em formato .png pela facilidade de manipulação, porém, outros tipos de formatos também são aceitos dentro da biblioteca OpenCV.

O modo escala de cinza foi escolhido pois nenhuma cor seria de utilidade dentro do programa, além de garantir uma maior precisão na detecção das bordas.

Após a leitura da imagem em tons de cinza, a imagem será armazenada dentro da variável denominada em forma de uma matriz Numpy, também conhecida como *N*-dimensional array.

Neste projeto o NumPy foi utilizado para armazenar informações em três dimensões, pois é necessário guardar imagens com largura, altura e profundidade relacionadas aos canais de cores RGB.

Outros motivos para a utilização deste tipo de estrutura foi o fato de realizar processamentos complexos com maior velocidade do que estruturas nativas do Python, ocupar menos memória ao armazenar dados em um bloco contínuo de memória e facilidade em aplicações que exigem cálculos e operações de processamento de imagens.

Após o carregamento das imagens, se inicia a detecção das bordas de cada imagem através da função a seguir:

FIGURA 18 - DETECÇÃO DAS BORDAS COM O ALGORITMO DE CANNY

```
frontal_edges = cv2.Canny(frontal, 100, 200)
lateral_edges = cv2.Canny(lateral, 100, 200)
superior_edges = cv2.Canny(superior, 100, 200)
```

FONTE: AUTORIA PRÓPRIA

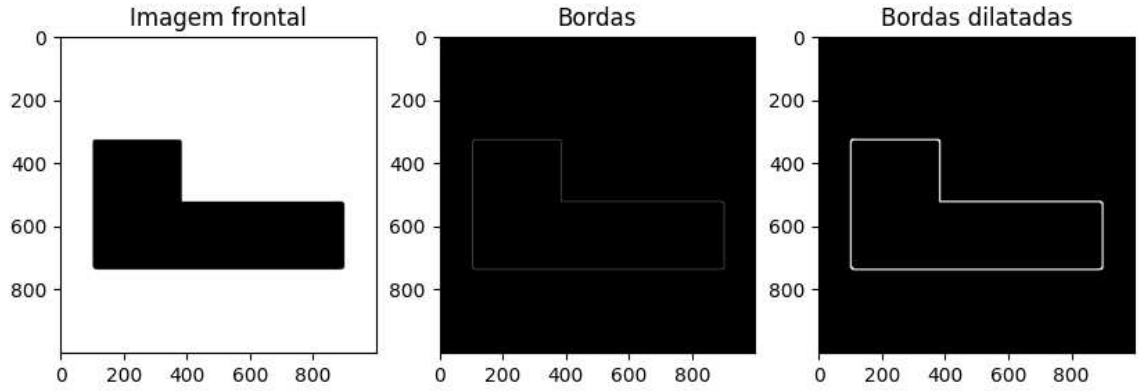
O funcionamento desta executa o algoritmo de Canny, utilizando como parâmetros a imagem de entrada convertida anteriormente e os limites de gradiente para a detecção das bordas.

No código acima, o limite de gradiente inferior foi de 100 e o superior de 200, esses limites foram escolhidos após testes com diferentes tipos de vistas ortogonais para determinar um intervalo que mantivesse a melhor forma das imagens.

Após esse reconhecimento inicial, foi aplicada uma dilatação em cada borda das imagens com finalidade de preencher lacunas e aumentar a espessura das bordas, além de garantir uma melhor visualização.

Para um maior entendimento das etapas descritas, foram geradas imagens para visualização das bordas, como é mostrado na Figura 16

FIGURA 19 - TRATAMENTO DAS BORDAS



FONTE: AUTORIA PRÓPRIA

A partir disso, é feita detecção das coordenadas das vértices das bordas dilatadas, a criação de uma lista para armazenar essas coordenadas e a conversão dessa lista em um array Numpy.

O método desenvolvido para a modelagem do projeto leva em conta apenas os vértices da silhueta das faces, não levando em conta linhas tracejadas e detalhes internos dos desenhos.

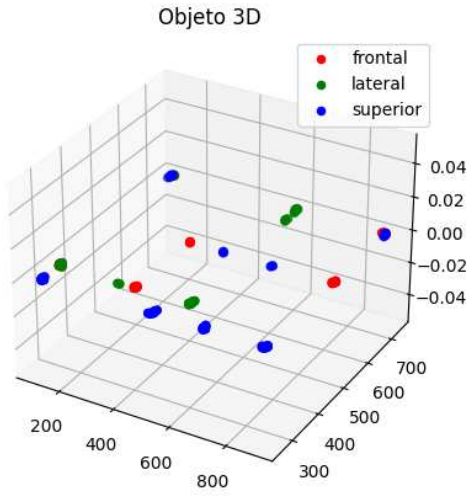
3.3 TRATAMENTO DOS VÉRTICES

Para dar prosseguimento ao projeto, foi necessário o tratamento dos vértices encontradas na etapa anterior. Visto que os vértices foram identificados a partir de um conjunto de bordas, o número era alto e impreciso, o que futuramente poderia causar apenas um aumento no número de polígonos na formação do sólido final.

Além de que o posicionamento de cada conjunto de vértices ainda não estava condizente com sua representação no mundo real (superior, lateral e lado direito).

Antes do tratamento dos vértices, os pontos foram gerados utilizando a biblioteca do Matplotlib da seguinte maneira (Figura 17):

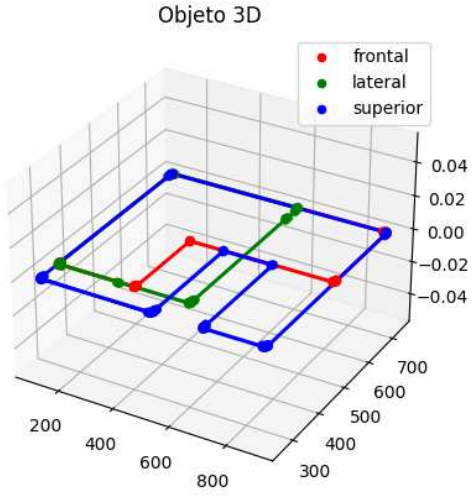
FIGURA 20 - DISPOSIÇÃO DOS PONTOS ANTES DO TRATAMENTO



FONTE: AUTORIA PRÓPRIA (2023)

Para uma melhor visualização, foram adicionadas as arestas seguindo os contornos, como na Figura 18:

FIGURA 21 - DISPOSIÇÃO DOS PONTOS ANTES DO TRATAMENTO COM ARESTAS



FONTE: AUTORIA PRÓPRIA (2023)

Como é possível perceber, a forma de cada desenho segue fiel com seus pontos projetados, porém, todos os pontos ainda são plotados no mesmo plano, o

que é contra a percepção no mundo real e posteriormente iria adicionar etapas na modelagem.

Como discutido anteriormente, foi necessária a diminuição na quantidade das vértices gerados, podendo diminuir a fidelidade das medidas, mas ainda sim mantendo a forma original com uma maior velocidade e praticidade.

Para diminuir o número de vértices foi necessário definir uma distância máxima para se considerar dois pontos como próximos, após alguns testes com diferentes tipos de imagens e faces, foi escolhido um número que era mais fiel ao desenho original de cada face, neste caso 20. Após, foi criada uma função para encontrar os pontos mais próximos e transformá-los em um único ponto médio. Em seguida foi necessário criar uma nova lista de pontos reduzida, como é possível ver a seguir:

FIGURA 22 - FUNÇÃO DE TRATAMENTO DAS ARESTAS

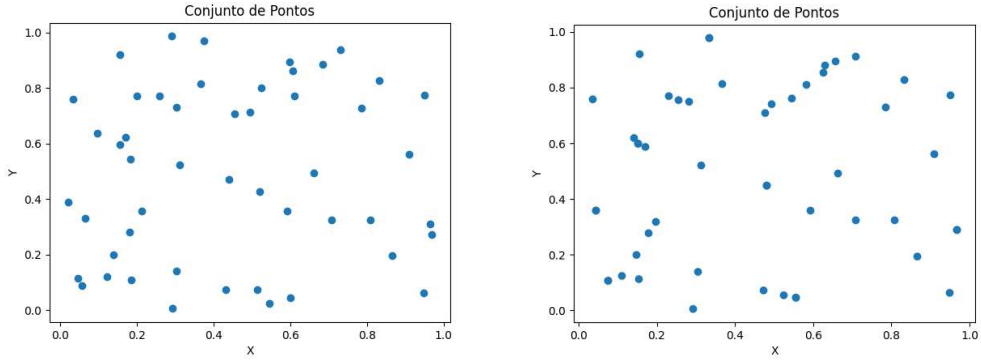
```
def find_average_coordinates(coordinates, max_distance):
    # Criar a árvore de busca k-dimensional
    tree = cKDTree(coordinates)

    # Encontrar os pontos próximos
    groups = tree.query_ball_tree(tree, r=max_distance)

    # Criar um novo array de coordenadas para armazenar os pontos médios
    new_coordinates = []

    # Iterar pelos grupos de pontos próximos
    for group in groups:
        if len(group) == 1:
              # Se o grupo tiver apenas um ponto, adicionar as coordenadas do ponto ao novo array
            new_coordinates.append(coordinates[group[0]])
        else:
                # Se o grupo tiver mais de um ponto, calcular a média das coordenadas
            mean_coord = np.mean(coordinates[group], axis=0)
            # Adicionar as coordenadas do ponto médio ao novo array
            new_coordinates.append(mean_coord)

    # Converter o novo array de coordenadas em um array NumPy
    new_coordinates_array = np.array(new_coordinates)
```

```
return new_coordinates_array
```

FONTE: AUTORIA PRÓPRIA (2023)

Esta função recebe a lista de coordenadas e a distância máxima definida e devolve um novo conjunto de pontos, com cerca de 20% a menos do que antes do tratamento, como ilustrado no exemplo da Figura 19 :

FIGURA 23 - PONTOS ANTES E DEPOIS DO TRATAMENTO, DEMONSTRAÇÃO

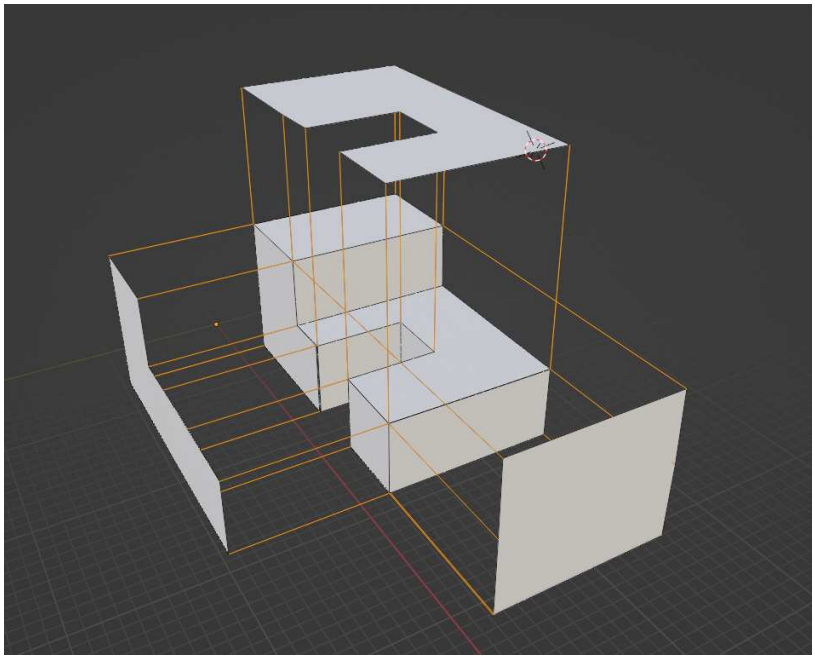


FONTE: AUTORIA PRÓPRIA (2023)

Assim que os pontos foram reduzidos, o foco do projeto foi direcionado para o deslocamento dos conjuntos de pontos em igualdade com suas projeções no mundo real.

Para melhor entendimento, foi feita a ilustração da Figura 20, em que é possível ver que cada vista devia ser deslocada para seu eixo “real” em relação a seu sólido. Deste modo o caminho encontrado para a modelagem automática seria mais prático no algoritmo que trabalha dentro do software.

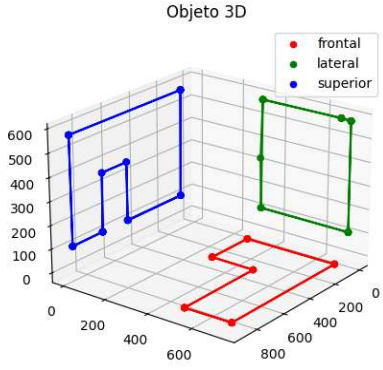
FIGURA 24 - VISTAS EM SUAS PROJEÇÕES DO OBJETO REAL



FONTE: AUTORIA PRÓPRIA (2023)

Assim, após o deslocamento das vistas a disposição segue a ideia proposta, como é possível ver na Figura 21:

FIGURA 25 - VISTAS DESLOCADAS EM RELAÇÃO A SUA POSIÇÃO REAL



FONTE: AUTORIA PRÓPRIA (2023)

Deste modo, os vértices estão prontos para a modelagem dentro do software Blender. Cada conjunto de vértices é salvo em um arquivo .txt.

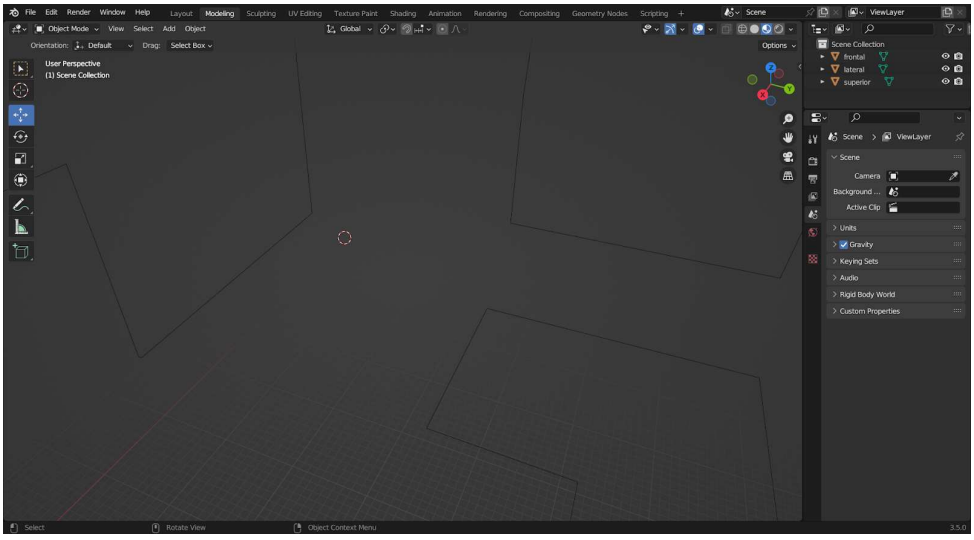
3.4 MODELAGEM AUTOMATIZADA UTILIZANDO O SOFTWARE BLENDER

Iniciando no código é feita a importação das bibliotecas `bpy`, `bmesh` e `math`, que já foram abordadas no capítulo anterior, seguindo para a leitura dos arquivos .txt.

Na primeira função do algoritmo, o objetivo final a ser atingido era a criação das 3 vistas com seus pontos e arestas já definidos, assim como foram na plotagem do primeiro código.

Começando pela leitura dos arquivos .txt, a função define os valores de cada ponto e calcula suas arestas e após isso, cria uma nova mesh e adiciona os vértices e arestas dentro desta. Dessa forma, cada vista seria um objeto tridimensional, como pode ser visto na Figura 22:

FIGURA 26 – VISTAS CRIADAS COMO OBJETOS DENTRO DO BLENDER

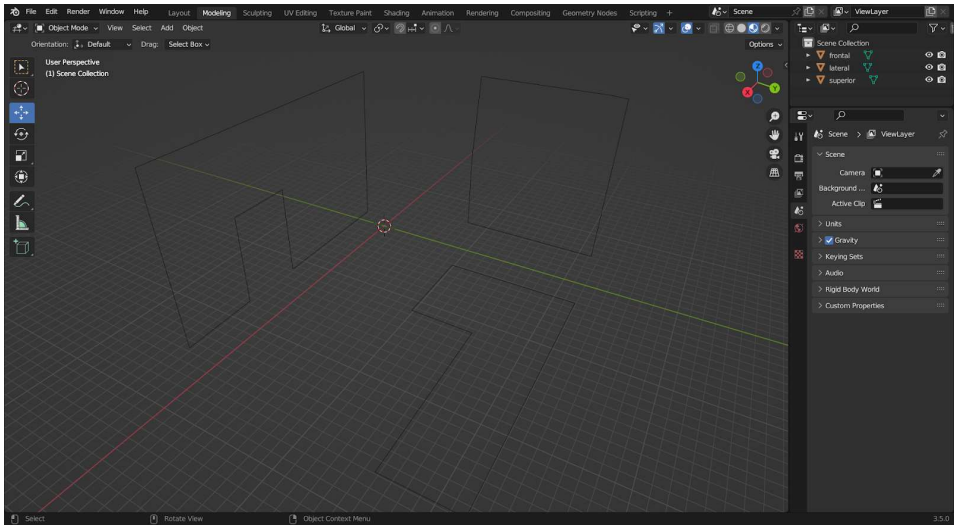


FONTE: AUTORIA PRÓPRIA (2023)

Porém, como é possível perceber, a escala dos pontos vindos do primeiro código está muito maior do que o espaço de trabalho que o Blender costuma utilizar.

Para corrigir isso, foi criada uma função que seleciona todos os objetos presentes dentro da cena e diminui sua escala em um terço da original. O resultado foi como mostra a Figura 23:

FIGURA 27 - VISTAS COMO OBJETOS DE ESCALA ALTERADA

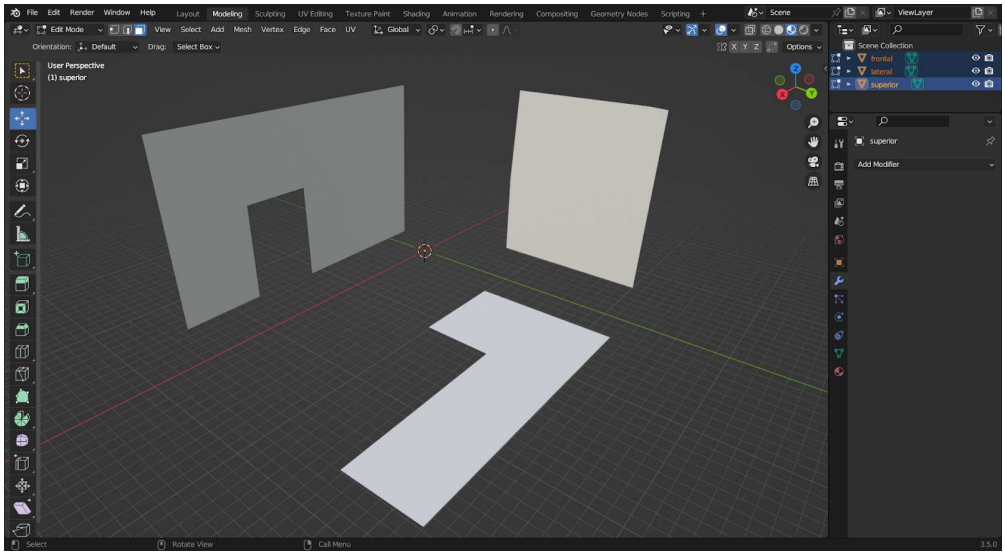


FONTE: AUTORIA PRÓPRIA (2023)

Assim, as meshs foram criadas como objetos dentro da cena, porém, para o prosseguimento do processo de modelagem, foi necessário que os objetos não fossem apenas “molduras” mas sim faces completas.

Para isso foi implementada uma ferramenta que permite preencher áreas fechadas de um objeto com faces, popularmente utilizada para “fechar buracos” nomeada como a ferramenta “fill” do software. Após a implementação nos objetos, as faces foram construídas como mostra a Figura 24:

FIGURA 28 - OBJETOS COMO FACES FECHADAS

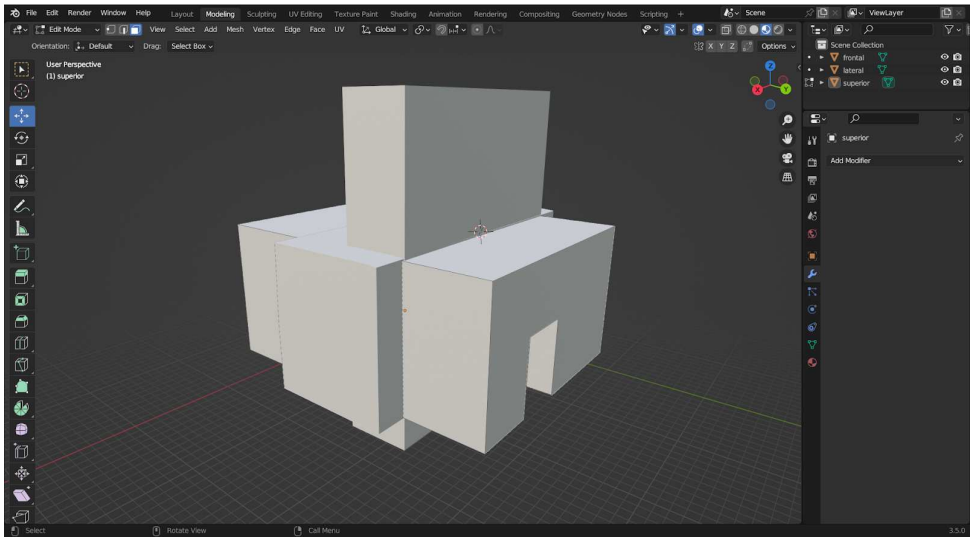


FONTE: AUTORIA PRÓPRIA (2023)

Após isso, foi possível seguir para a extrusão das meshs. Nesta etapa cada face selecionada é aumentada em um eixo selecionado, adicionando mais vértices, pontos ou arestas ao objeto. Essa ferramenta é comumente utilizada para criar volume ou adicionar detalhes.

Como dito anteriormente, as vistas foram posicionadas de acordo com sua projeção real no sólido. Isso foi feito para que nesta etapa de extrusão cada mesh extrudada pudesse ser direcionada ao centro, tendo que dentro desse volume o sólido compatível com o objeto real. Na imagem da Figura 25 é possível ver as meshs extrudadas, em que cada vista foi extrudada com seu eixo em direção ao centro do sólido. A vista frontal no eixo Z, a lateral no eixo X e a superior no eixo Y:

FIGURA 29 - RESULTADO APÓS A EXTRUSÃO DAS FACES



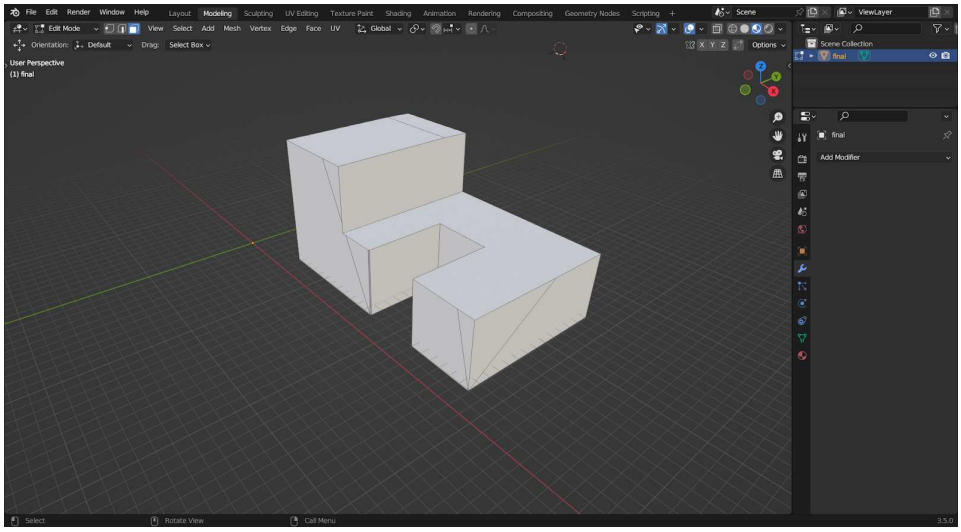
FONTE: AUTORIA PRÓPRIA (2023)

Para finalizar a modelagem, é possível perceber que o sólido final se encontra justamente na interseção dos sólidos extrudados, visto que cada um deles representa o aspecto de sua projeção alongada.

Para atingir o resultado pretendido, foi utilizada a ferramenta de interseção do software Blender, que gera um quarto objeto formado pela área de interseção dos objetos das vistas.

Após isso, temos o resultado final do algoritmo, o sólido 3d gerado a partir das vistas ortogonais na Figura 26:

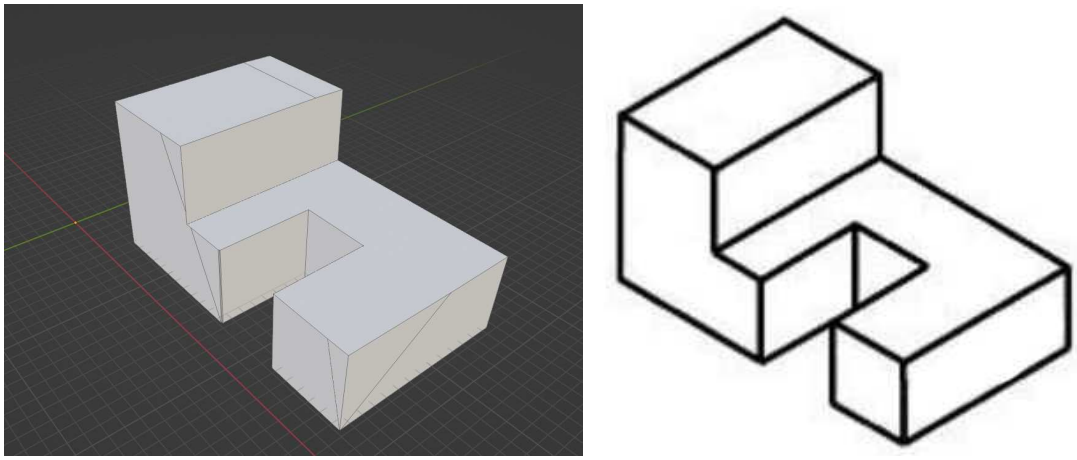
FIGURA 30 - SÓLIDO FINAL GERADO A PARTIR DAS VISTAS ORTOGONAIS



FONTE: AUTORIA PRÓPRIA (2023)

Para comparação com o sólido que originou as vistas ortogonais em seguida é apresentada a Figura 27:

FIGURA 31 - COMPARAÇÃO DO SÓLIDO GERADO COM O ORIGINAL



FONTE: AUTORIA PRÓPRIA (2023)

4. RESULTADOS

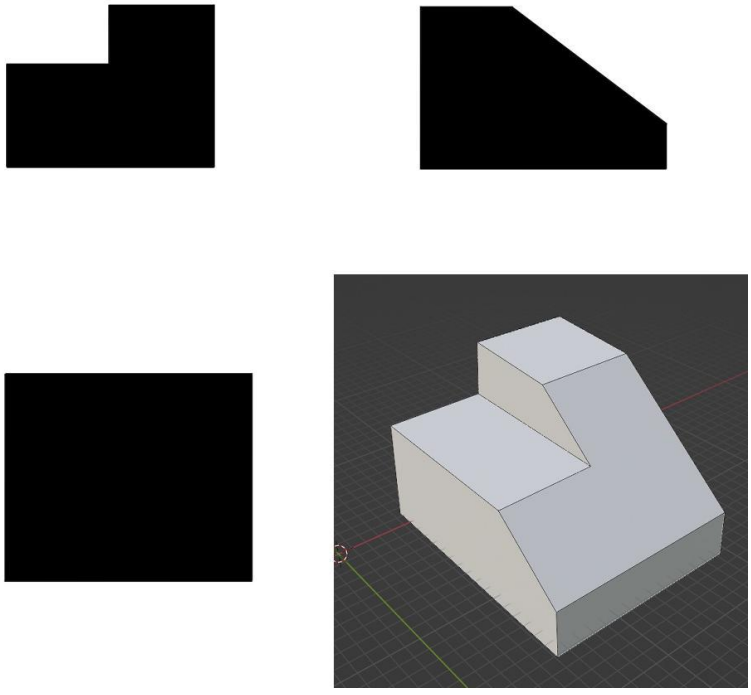
Nesta seção são apresentados os resultados alcançados pela implementação do projeto, a avaliação dos mesmos e uma comparação com outras abordagens já existentes. Todos os sólidos a seguir foram gerados da mesma maneira que o sólido do capítulo anterior.

4.1 APRESENTAÇÃO DOS RESULTADOS

Após a finalização dos códigos e da geração do primeiro sólido mostrado no capítulo anterior, foram gerados outros três para a avaliação dos resultados obtidos pelo projeto (apresentados a seguir). As vistas utilizadas para a construção desses sólidos foram retiradas da apostila didática do Professor Dr. eng. Márcio Fontana Catapan e redesenhadas e preenchidas posteriormente para o pré-processamento das imagens, como comentado anteriormente (CATAPAN, 2015).

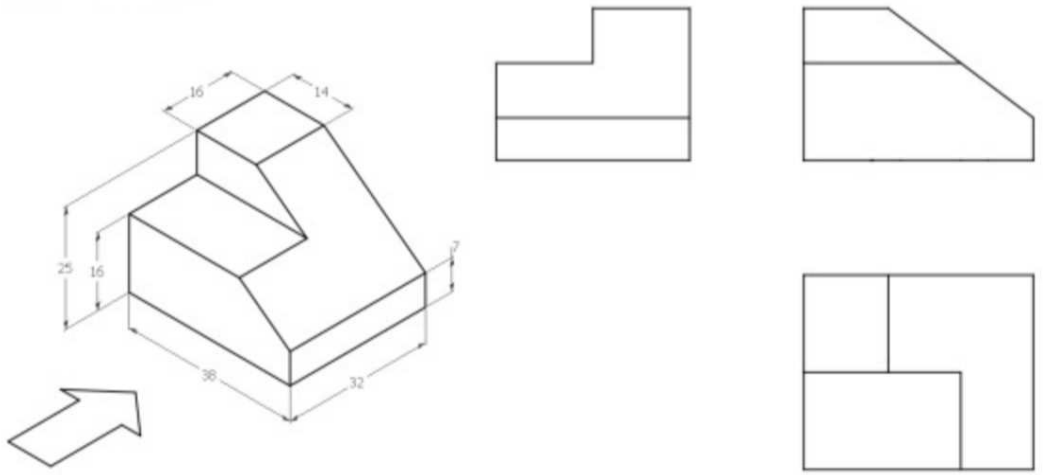
A seguir, os outros três sólidos gerados, suas comparações com os desenhos originais e suas vistas ortogonais já preenchidas como mostram as Figuras 28 a 33:

FIGURA 32 - SEGUNDO SÓLIDO GERADO A PARTIR DAS VISTAS ORTOGONAIS



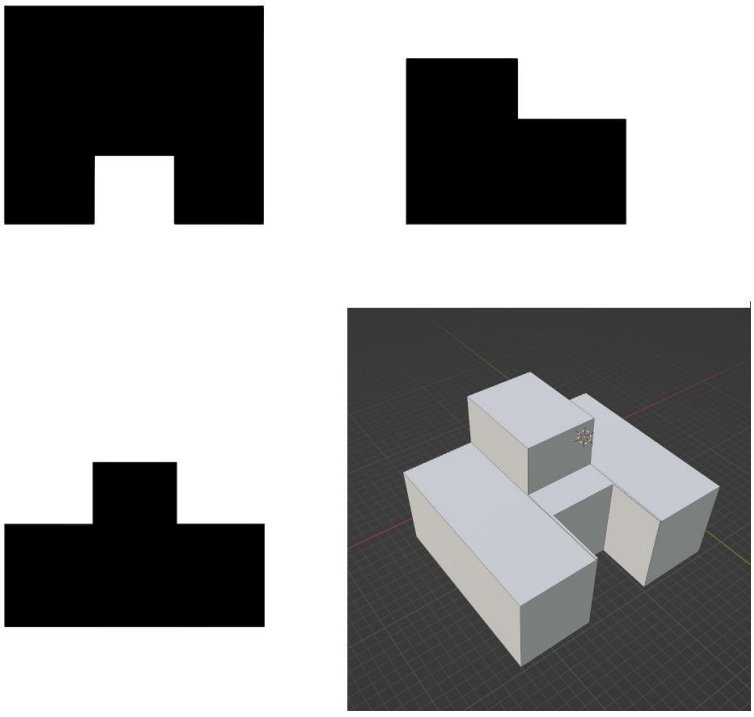
FONTE: AUTORIA PRÓPRIA (2023)

FIGURA 33 - DESENHO TÉCNICO DO SEGUNDO SÓLIDO RETIRADO DA APOSTILA



FONTE: CATAPAN, (2015)

FIGURA 34 - TERCEIRO SÓLIDO GERADO A PARTIR DAS VISTAS ORTOGONAIS

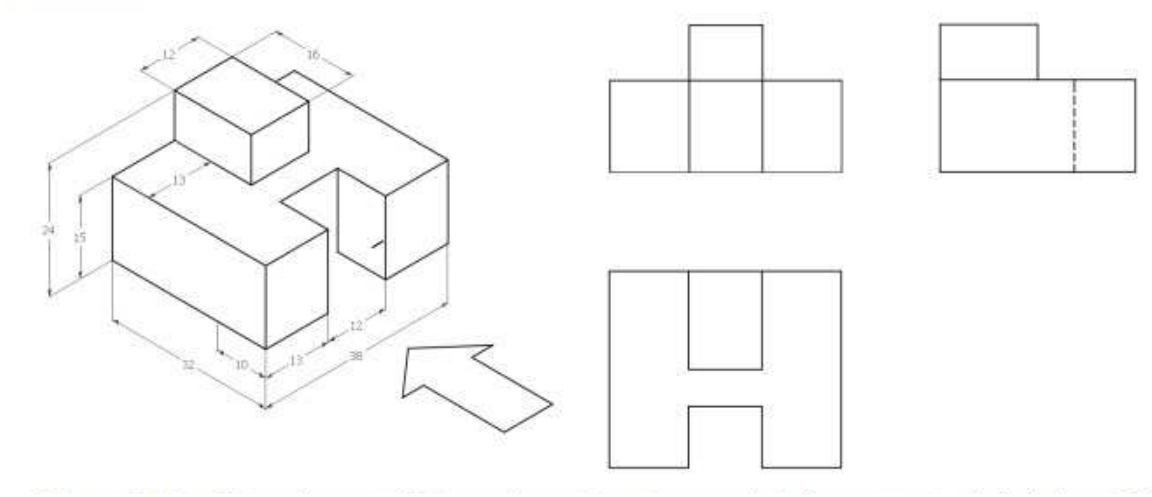


FONTE: AUTORIA PRÓPRIA (2023)

Um ponto a ser ressaltado em todas as imagens e, principalmente na figura anterior, é a diferença de perspectiva entre a visualização dentro do software Blender e a do desenho técnico. Isso se dá pela diferença do modo de perspectiva onde, no desenho técnico a perspectiva é isométrica, ou seja, não existem pontos de

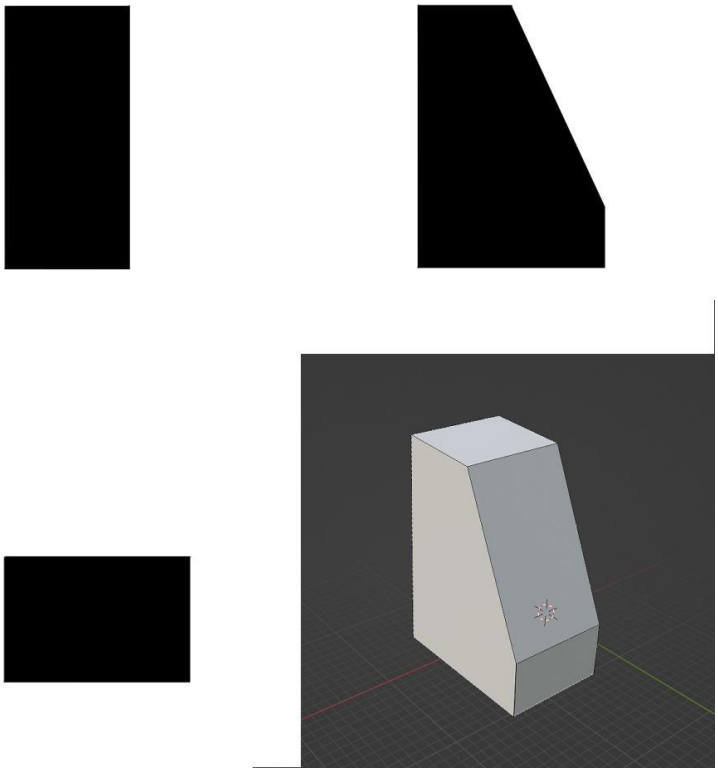
fuga para se distorcer a imagem. Porém na visualização 3D a perspectiva é mais próxima da visualização real, tendo pontos de fuga e “distorcendo” a imagem.

FIGURA 35 - DESENHO TÉCNICO DO TERCEIRO SÓLIDO RETIRADO DA APOSTILA



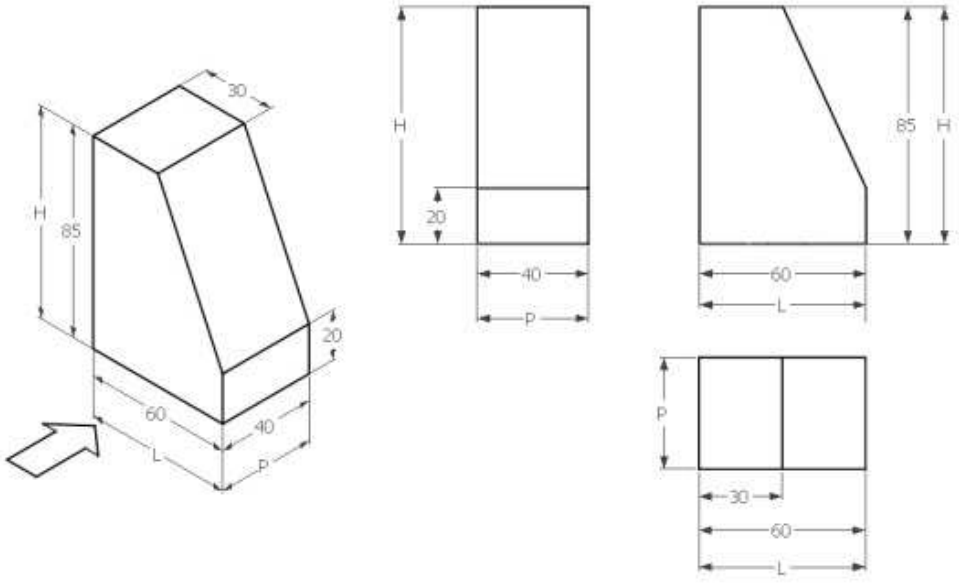
FONTE: CATAPAN, (2015)

FIGURA 36 - QUARTO SÓLIDO GERADO A PARTIR DAS VISTAS ORTOGONAIS



FONTE: AUTORIA PRÓPRIA (2023)

FIGURA 37 - DESENHO TÉCNICO DO QUARTO SÓLIDO RETIRADO DA APOSTILA



FONTE: CATAPAN, (2015)

Assim, é possível concluir que os resultados apresentados são próximos aos seus respectivos modelos, afirmando a eficácia do método para esses modelos de sólido.

4.2 AVALIAÇÃO DOS RESULTADOS

Após a avaliação dos sólidos gerados, foram identificadas possíveis melhorias para o projeto. Nota-se:

- Há interferência da qualidade da imagem no processamento dos pontos;
- Áreas em que o projeto se encaixava;
- Tipo específico de sólido que não se encaixa no projeto.

Começando pela interferência da qualidade da imagem no processamento, foi observado que em alguns sólidos específicos, como os que possuem algum tipo de curva ou cortes em diagonal, após o processamento das vistas, os vértices

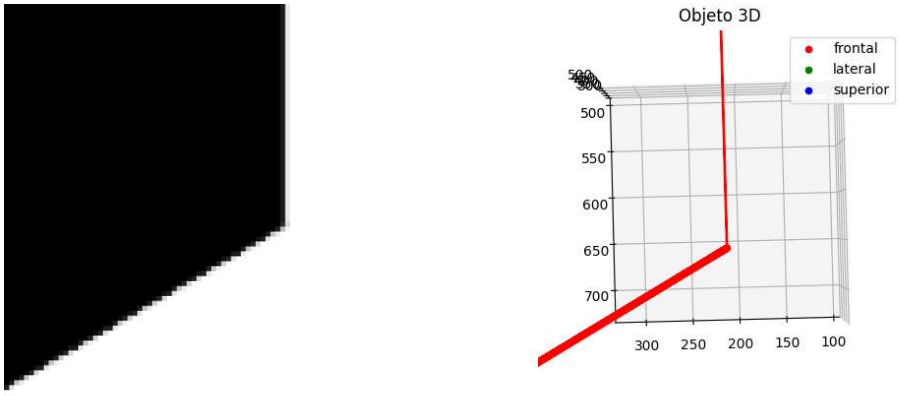
identificados após a dilatação das bordas apareciam numa quantidade muito maior do que deveriam. Por exemplo, uma diagonal que deveria possuir apenas dois vértices e uma aresta, gera inúmeros pontos.

Ao realizar uma análise do porquê esse problema ocorria foi tentado alterar o número do range máximo para pontos próximos, diminuindo assim o número de pontos. Porém, com essa abordagem a maioria das vistas perdia seu formato original, comprometendo o resultado final do sólido e gerando grandes diferenças entre seu desenho ortogonal e sua representação 3D.

Após isso, o problema foi analisado novamente e foi percebido que, quanto menor a resolução da imagem, mais pontos ela teria em áreas arredondadas ou diagonais já que o serrilhamento de pixels atrapalha na leitura da imagem, visto que cada pixel fora da média é visto como um ponto. A solução encontrada foi a de utilizar imagens vetoriais (ou de qualidade alta) ao invés de imagens em bitmap para os desenhos preenchidos.

A seguir, na Figura 34, é possível ver a demonstração do que o serrilhamento dos pixels por uma imagem de baixa qualidade pode causar na definição dos pontos:

FIGURA 38 - SERRILHAMENTO DE PIXELS CAUSANDO MAIS PONTOS NA LEITURA



FONTE: AUTORIA PRÓPRIA (2023)

Esse aumento no número de pontos é prejudicial tanto para a construção do sólido, quanto para o sólido renderizado primeiro porque ao aumentar o número de vértices dessa maneira, o segundo código que será executado dentro do Blender, terá dificuldades na etapa do preenchimento do plano. Também porque quanto maior

o número de pontos, mais “pesada” fica a renderização do sólido, podendo afetar o desempenho do código tanto em velocidade quanto em processamento, o que pode levar a parada do software. Por isso a necessidade de sempre utilizar imagens de qualidade mais alta, como mencionado anteriormente.

O segundo ponto analisado foi a identificação das áreas em que o projeto mais se encaixaria dentro do planejado. Na concepção do projeto, duas finalidades principais foram definidas: a primeira foi seu uso num ramo construtivo, seja no ambiente real ou virtual; a segunda foi para auxiliar na didática das matérias de desenho técnico para engenharia elétrica.

O terceiro ponto foi notado após finalizar o projeto e após a análise dos resultados, onde foi percebido que para um ambiente construtivo virtual, como em assets e props para jogos eletrônicos de baixo número de polígonos (low-poly), projeções arquitetônicas e simulações, bem como o auxílio na didática dentro e fora da sala de aula, o projeto cumpre com a sua função. Manteve-se a principal geometria do sólido e aumentou-se a velocidade do processo.

No ramo virtual o projeto contribui na diminuição de gastos e tempo, mas também existe a possibilidade do projeto ser levado como uma nova ferramenta a ser implementada em alguns softwares, como o Blender.

Porém, ao abordar o funcionamento do projeto dentro de um ambiente industrial, foi levado em consideração que a precisão das medidas é algo indispensável para o cumprimento da tarefa. Na impressão 3D de um componente, por exemplo, cada milímetro é essencial para o bom funcionamento e longevidade do equipamento.

Nesse caso específico, é possível dizer que o projeto não pode se enquadrar de forma direta, já que uma precisão tão alta não foi algo levado como prioridade. Entretanto, para implementações futuras, o ajuste de uma precisão mais acentuada do sólido foi definido como um dos pontos principais para uma nova versão.

Durante os testes também foi percebido que, para um tipo específico de sólido, a renderização não era perfeita. Uma característica que pode afetar a projeção final. Estes seriam os que possuem algum tipo de corte ou orifício que não pode ser identificado em nenhuma de suas silhuetas, visto que o primeiro código realiza a leitura de suas vértices a partir delas.

Essa característica faz com que o detalhe em questão não seja identificado e, posteriormente, também não seja gerado na modelagem do sólido final.

Contudo, ao avaliar que a proposta definida no início do projeto era a geração de sólidos simples e, considerando que uma grande parcela desses sólidos não possuem a característica citada, é possível afirmar que o projeto tem aplicabilidade dentro dos limites comentados anteriormente. Ressalta-se que o foco do trabalho era dar início ao estudo do tema e que melhorias que permitam acertar este código é um importante trabalho futuro.

5. CONCLUSÃO E SUGESTÕES PARA TRABALHOS FUTUROS

Este trabalho explorou o tema de processamento de vistas ortogonais para a modelagem 3D utilizando a linguagem em Python. A modelagem 3D desempenha um papel fundamental em diversas áreas, como engenharia, arquitetura, jogos e animação, permitindo a criação de objetos virtuais que se assemelham a objetos reais.

Os resultados obtidos comprovaram a viabilidade e eficácia do tema proposto dentro dos limites já comentados, demonstrando que a automação do processo de desenvolvimento de objetos 3D a partir de vistas ortogonais pode trazer benefícios significativos em termos de economia de tempo, redução de falhas e recursos. Além disso, o uso do Python como linguagem de programação possibilitou a criação de um sistema flexível e fácil manutenção, permitindo a extensão e adaptação para diferentes cenários e necessidades específicas.

Algumas dificuldades foram encontradas no desenvolvimento do projeto, como o método de modelagem a ser escolhido que abrangesse a maior parte dos sólidos, os ajustes para detecção dos vértices das imagens e o ajuste no tratamento desses vértices.

Para trabalhos futuros, uma ligação do projeto com uma impressora 3D através de uma interface poderia trazer um avanço ainda maior dentro da engenharia elétrica, bem como melhorias no código para atender a todos os tipos de sólidos existentes. É de grande importância que pesquisas continuem sendo feitas neste tema para um maior desenvolvimento da área.

Em suma, este trabalho demonstrou que a combinação de processamento de vistas ortogonais com a referida linguagem é uma abordagem promissora e eficaz para a modelagem 3D automatizada. A automação deste processo pode trazer benefícios significativos para as áreas de aplicação ao simplificar a criação de modelos 3D, impulsionando a eficiência e a produtividade no desenvolvimento de projetos que envolvem a visualização e análise de objetos tridimensionais.

APÊNDICE A - CÓDIGO 1

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import cKDTree

# Leitura da imagem
################################################################################

# Carregar as três imagens das vistas ortogonais em tons de cinza em uma lista
frontal = cv2.imread("vista_frontal_full.png",0)
lateral = cv2.imread("vista_lateral_full.png",0)
superior = cv2.imread("vista_superior_full.png",0)

# Cria uma imagem de referência usando a dimensão da imagem frontal
reference_image = np.zeros(frontal.shape, np.uint8)

# Realizar a detecção de bordas em cada imagem
frontal_edges = cv2.Canny(frontal, 100, 200)
lateral_edges = cv2.Canny(lateral, 100, 200)
superior_edges = cv2.Canny(superior, 100, 200)

# Realizar a dilatação nas bordas de cada face
kernel = np.ones((5,5), np.uint8)
dilated_edges_f = cv2.dilate(frontal_edges, kernel, iterations=1)
dilated_edges_l = cv2.dilate(lateral_edges, kernel, iterations=1)
dilated_edges_s = cv2.dilate(superior_edges, kernel, iterations=1)

# Exibir as imagens
fig, axs = plt.subplots(1, 3, figsize=(10, 4))
axs[0].imshow(frontal, cmap='gray')
axs[1].imshow(frontal_edges, cmap='gray')
axs[2].imshow(dilated_edges_f, cmap='gray')

axs[0].set_title('Imagem frontal')
axs[1].set_title('Bordas')
axs[2].set_title('Bordas dilatadas')

plt.show()
```

```
# Encontrar os contornos das bordas dilatadascv2.CHAIN_APPROX_SIMPLE)
contours_s, hierarchy_s = cv2.findContours(dilated_edges_s, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

# Imprime o número de contornos encontrados
print("Número de contornos encontrados frontal:", len(contours_f))
print("Número de contornos encontrados lateral:", len(contours_l))
print("Número de contornos encontrados superior:", len(contours_s))

def obter_coordenadas(contours):
    points = []
    for contour in contours:
contours_f, hierarchy_f = cv2.findContours(dilated_edges_f, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)
contours_l, hierarchy_l = cv2.findContours(dilated_edges_l, cv2.RETR_TREE,
        for point in contour:
            points.append(point[0])
    return points

# Chamar a função para obter as coordenadas dos contornos
points_f = obter_coordenadas(contours_f)
points_l = obter_coordenadas(contours_l)
points_s = obter_coordenadas(contours_s)

# Imprime o número de pontos encontrados
print("Número de pontos encontrados frontal:", len(points_f))
print("Número de pontos encontrados lateral:", len(points_l))
print("Número de pontos encontrados superior:", len(points_s))

# Converter a lista de pontos em um array NumPy e criar uma matriz de coordenadas em 3D
points_array_f = np.array(points_f)
coordinates_f = np.column_stack((points_array_f[:,0], points_array_f[:,1], points_array_f[:,1] * 0))

points_array_l = np.array(points_l)
coordinates_l = np.column_stack((points_array_l[:,0], points_array_l[:,1], points_array_l[:,1] * 0))

points_array_s = np.array(points_s)
coordinates_s = np.column_stack((points_array_s[:,0], points_array_s[:,1], points_array_s[:,1] * 0))
```

```
# Tratamento das vertices
##############################################################################
###

# Definir a distância máxima para considerar dois pontos como próximos
max_distance = 20

def find_average_coordinates(coordinates, max_distance):
    # Criar a árvore de busca k-dimensional
    tree = cKDTree(coordinates)

    # Encontrar os pontos próximos
    groups = tree.query_ball_tree(tree, r=max_distance)

    # Criar um novo array de coordenadas para armazenar os pontos médios
    new_coordinates = []

    # Iterar pelos grupos de pontos próximos
    for group in groups:
        if len(group) == 1:
            # Se o grupo tiver apenas um ponto, adicionar as coordenadas do ponto ao novo array
            new_coordinates.append(coordinates[group[0]])
        else:
            # Se o grupo tiver mais de um ponto, calcular a média das coordenadas
            mean_coord = np.mean(coordinates[group], axis=0)
            # Adicionar as coordenadas do ponto médio ao novo array
            new_coordinates.append(mean_coord)

    # Converter o novo array de coordenadas em um array NumPy
    new_coordinates_array = np.array(new_coordinates)

    return new_coordinates_array

# Chamar a função para obter as novas coordenadas para cada vista ortogonal
new_coordinates_array_f = find_average_coordinates(coordinates_f, max_distance)
new_coordinates_array_s = find_average_coordinates(coordinates_s, max_distance)
new_coordinates_array_l = find_average_coordinates(coordinates_l, max_distance)


# Deslocar as vistas ########################################################
```

```
new_coordinates_array_s[:,[1,2]] = new_coordinates_array_s[:,[2,1]]
new_coordinates_array_l[:,[0,2]] = new_coordinates_array_l[:,[2,0]]

# Encontre a coordenada z do ponto mais baixo em new_coordinates_array_s
min_z_s = np.min(new_coordinates_array_s[:,2])

# Encontre a coordenada z do ponto mais baixo em new_coordinates_array_l
min_z_l = np.min(new_coordinates_array_l[:,2])

# Calcule a diferença entre as coordenadas z
diff_z = min_z_l - min_z_s

# Adicione a diferença a todas as coordenadas z em new_coordinates_array_s
new_coordinates_array_s[:,2] += diff_z



# Salvar pontos e faces ########################################
def save_txt(filename, points):
    with open(filename, 'w') as f:
        unique_points = []
        for point in points:
            int_point = tuple(map(int, point))  # Converte os valores para inteiros
            if int_point not in unique_points:
                unique_points.append(int_point)
                f.write(f'{int_point[0]} {int_point[1]} {int_point[2]}\n')

# Chame a função save_txt para salvar o arquivo
save_txt('pontos_f.txt', new_coordinates_array_f)
save_txt('pontos_s.txt', new_coordinates_array_s)
save_txt('pontos_l.txt', new_coordinates_array_l)

# Plotagem do gráfico ##########################################################

# Cria um gráfico 3D
fig = plt.figure()
ax = plt.axes(projection="3d")

# Plota os pontos das três vistas ortográficas
```

```
ax.scatter(new_coordinates_array_f[:,0], new_coordinates_array_f[:,1], new_coordinates_array_f[:,2],
c='r', label="frontal")
ax.scatter(new_coordinates_array_l[:,0], new_coordinates_array_l[:,1], new_coordinates_array_l[:,2],
c='g', label="lateral")
ax.scatter(new_coordinates_array_s[:,0], new_coordinates_array_s[:,1],
new_coordinates_array_s[:,2], c='b', label="superior")

# Plotar as arestas do objeto 3D
ax.plot(new_coordinates_array_f[:,0], new_coordinates_array_f[:,1], new_coordinates_array_f[:,2],
c='r')
ax.plot(new_coordinates_array_l[:,0], new_coordinates_array_l[:,1], new_coordinates_array_l[:,2],
c='g')
ax.plot(new_coordinates_array_s[:,0], new_coordinates_array_s[:,1], new_coordinates_array_s[:,2],
c='b')

# Define o título e legenda do gráfico
ax.set_title("Objeto 3D")
ax.legend()

# Mostra o gráfico
plt.show()
```

APÊNDICE B - CÓDIGO 2

```
import bpy
import bmesh
import math

def criar_arestas_arquivo(nome_arquivo, nome_mesh):
    # Abrir o arquivo de texto
    with open(nome_arquivo, 'r') as arquivo:
        # Ler as linhas do arquivo
        linhas = arquivo.readlines()

    # Lista de pontos no formato (x, y, z)
    pontos = []
    for linha in linhas:
        # Separar os valores da linha e convertê-los para inteiros
        valores = [int(valor) for valor in linha.strip().split()]
        pontos.append(tuple(valores))

    # Criar um novo objeto do tipo "Mesh" para armazenar as arestas
    mesh = bpy.data.meshes.new(nome_mesh)
    obj = bpy.data.objects.new(nome_mesh, mesh)
    bpy.context.collection.objects.link(obj)

    # Criar o objeto BMesh
    bm = bmesh.new()

    # Criar os vértices
    vertices = []
    for ponto in pontos:
        x, y, z = ponto
        vert = bm.verts.new((x, y, z))
        vertices.append(vert)

    # Criar as arestas
    for i in range(len(vertices)):
        start_vert = vertices[i]
        end_vert = vertices[(i + 1) % len(vertices)]  # Índice circular para fechar o contorno
        bm.edges.new([start_vert, end_vert])

    # Atualizar a malha
```

```
    bm.to_mesh(mesh)
    bm.free()

# Chamada da função com o nome do arquivo
criar_arestas_arquivo("C:/Users/luizf/Downloads/pontos_f.txt", "frontal")
criar_arestas_arquivo("C:/Users/luizf/Downloads/pontos_l.txt", "lateral")
criar_arestas_arquivo("C:/Users/luizf/Downloads/pontos_s.txt", "superior")

# Preencher mesh
def preencher_mesh(mesh):
    bpy.context.view_layer.objects.active = mesh
    bpy.ops.object.mode_set(mode='EDIT')
    bpy.ops.mesh.select_all(action='SELECT')
    bpy.ops.mesh.edge_face_add()
    bpy.ops.object.mode_set(mode='OBJECT')

preencher_mesh(bpy.data.objects["frontal"])
preencher_mesh(bpy.data.objects["lateral"])
preencher_mesh(bpy.data.objects["superior"])

# Arrumar escala
def selecionar_e_diminuir_escala_meshs(escala):
    bpy.ops.object.select_all(action='DESELECT')  # Deseleciona todos os objetos

    # Seleciona apenas os objetos do tipo "MESH"
    bpy.ops.object.select_by_type(type='MESH')

    # Diminui a escala de cada objeto selecionado
    for obj in bpy.context.selected_objects:
        obj.scale = escala

escala = (0.3, 0.3, 0.3)  # Define a escala desejada
selecionar_e_diminuir_escala_meshs(escala)

# Extrudar meshs
def extrudar_mesh(mesh_nome, eixo, distancia):
    bpy.ops.object.select_all(action='DESELECT')  # Deseleciona todos os objetos
    mesh = bpy.data.objects.get(mesh_nome)  # Obtém a referência para o objeto mesh pelo nome

    if mesh is not None:
```

```
        bpy.context.view_layer.objects.active = mesh
        bpy.ops.object.mode_set(mode='EDIT')
        bpy.ops.mesh.select_all(action='SELECT')
        bpy.ops.mesh.extrude_region_move(
            TRANSFORM_OT_translate={
                "value": tuple(distancia if i == eixo else 0 for i in range(3))
            }
        )
        bpy.ops.object.mode_set(mode='OBJECT')

extrudar_mesh("frontal", 2, 300)
extrudar_mesh("lateral", 0, 300)
extrudar_mesh("superior", 1, 300)

# Criar a interseção das meshs
def intersect_meshes(mesh1_name, mesh2_name, output_name):
    # Selecionar as meshes
    bpy.context.view_layer.objects.active = bpy.data.objects[mesh1_name]
    bpy.data.objects[mesh1_name].select_set(True)
    bpy.data.objects[mesh2_name].select_set(True)

    # Modificador de interseção
    bpy.ops.object.modifier_add(type='BOOLEAN')
    bpy.context.object.modifiers["Boolean"].operation = 'INTERSECT'
    bpy.context.object.modifiers["Boolean"].object = bpy.data.objects[mesh2_name]

    # Aplicar o modificador
    bpy.ops.object.modifier_apply(modifier="Boolean")

    # Renomear o resultado da interseção
    bpy.data.objects[mesh1_name].name = output_name

intersect_meshes("frontal", "lateral", "intersex")
intersect_meshes("intersex", "superior", "final")

# Apagar meshs de apoio
def delete_mesh(mesh_name):
    # Verificar se a malha existe
    if mesh_name in bpy.data.objects:
        # Mudar para o modo de objeto se estiver no modo de edição
```

```
        if bpy.context.object.mode == 'EDIT':
            bpy.ops.object.mode_set(mode='OBJECT')

        # Deselecionar todos os objetos
        bpy.ops.object.select_all(action='DESELECT')

        # Selecionar o objeto da malha a ser removida
        bpy.data.objects[mesh_name].select_set(True)

        # Deletar todos os objetos selecionados
        bpy.ops.object.delete()

# Exemplo de uso:
delete_mesh("lateral")
delete_mesh("superior")



# Girar a mesh
def rotate_mesh_90_degrees(mesh_name):
    # Selecionar a malha
    bpy.context.view_layer.objects.active = bpy.data.objects[mesh_name]
    bpy.data.objects[mesh_name].select_set(True)

    # Converter 90 graus para radianos
    angle = math.radians(90)

    # Rotacionar ao longo do eixo X
    bpy.ops.transform.rotate(value=angle, orient_axis='X')

rotate_mesh_90_degrees("final")

# Mover a mesh pra (0,0,0)
def move_mesh_to_origin(mesh_name):
    # Selecionar a malha
    bpy.context.view_layer.objects.active = bpy.data.objects[mesh_name]
    bpy.data.objects[mesh_name].select_set(True)

    # Acessar a propriedade de localização da malha
    mesh_object = bpy.data.objects[mesh_name]
    mesh_object.location = (0, 0, 0)
```

```
move_mesh_to_origin("final")
```

REFERÊNCIAS

Andrade P. H. A., Vieira, J. W., Oliveira V. R. S., Veloso R. J. B., Limaz F. R. A. UM MÉTODO para voxelização de geometrias 3D de malhas. BRAZILIAN JOURNAL OF RADIATION SCIENCES, [*S. l.*], p. 1 - 10, 1 out. 2020.

ANTONELLO, Ricardo. Introdução a Visão Computacional com Python e OpenCV. Local: IFC, 2017.

AZEVEDO, E.; CONCI, A. Computação Gráfica, teoria e prática. v. 1. Rio de Janeiro: Elsevier, 2003.

BLENDER. [S. l.], 1 jun. 2023. Disponível em: https://www.blender.org/. Acesso em: 6 jun. 2023.

BUENO, Marcelo Lemes. Detecção de bordas através do algoritmo de Canny. Disponível em: https://www.inf.ufsc.br/~aldo.vw/visao/2000/Bordas/index.html

CANNY, J. A Computational Approach to Edge Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, V. 8, n. 6, pp. 679-698, 1986.

CARIAS, Beatriz. A evolução das novas tecnologias. 2020. Disponível em: https://digartdigmedia.wordpress.com/2020/11/25/nativo-e-imigrante-digital/

CATAPAN, Márcio Fontana. Apostila de desenho técnico. Curitiba. 2015

CIVIL SEEK. Orthographic projection, Drawing: A Comprehensive Guide. 2020. Disponível em: https://civilseek.com/orthographic-projection-drawing/

CONRADO, Daniel. Contextualização da Computação Gráfica 3D. 2019. Disponível em: https://www.linkedin.com/pulse/contextualização-da-computação-gráfica-3d-daniel-conrado/?originalSubdomain=pt

DANTAS, S, C; MATHIAS, C, V. Formas de revolução e cálculo de volume. 2016. Disponível em: https://www.redalyc.org/journal/4675/467549116016/html/

DHONDGE, Tanishka. Multidimensional arrays in python: a complete guide. 2023. Disponível em: https://www.askpython.com/python/array/multidimensional-arrays

Dissertação de Mestrado (Programa de Pós-Graduação em Modelagem Matemática e Computacional) - UFPB, [S. l.], 2021.

GOMES J., VELHO L. Computação Gráfica: Imagem. Série de Computação e Matemática. IMPA/SBM, Rio de Janeiro, 424p, 1994

LIMA, Rafael Pereira de. UM ESTUDO DO DIAGRAMA DE VORONOI PARA DOIS PONTOS GERADORES ESPECÍFICOS COM UM OBSTÁCULO CIRCULAR. 2021.

LOPES, Michele. O que é modelagem 3D?. [*S. l.*], 1 jun. 2023. Disponível em: https://ebaconline.com.br/blog/modelagem-3d-o-que-e-e-como-funciona#:~:text=Modelagem%203D%20%C3%A9%20considerado%20um,Maya%2C%20Blender%2C%20entre%20outros. Acesso em: 9 jun. 2023.

MATPLOTLIB: plotting. [S. l.], 30 dez. 2017. Disponível em: https://scipy-lectures.org/intro/matplotlib/index.html. Acesso em: 6 jun. 2023.

MIRANDA, A. L. Da natureza da tecnologia: uma análise filosófica sobre as dimensões ontológica, epistemológica e axiológica da tecnologia moderna. 2002 pp. 161 (Dissertação de mestrado). Programa de Pós-Graduação em Tecnologia do Centro Federal de Educação Tecnológica do Paraná (CEFET-PR)

MODELAGEM por Varredura. [*S. l.*], 13 fev. 2023. Disponível em: https://help.autodesk.com/view/INVNTOR/2023/PTB/?guid=GUID-FCC0126F-50ED-449B-A0EF-1DED0287170B. Acesso em: 23 jun. 2023.

MULINARI, Bruna . Introdução a Visão Computacional com Python e OpenCV. Numpy Python: O que é, vantagens e tutorial inicial, 2018. Disponível em: https://harve.com.br/blog/programacao-python-blog/numpy-python-o-que-e-vantagens-e-tutorial-inicial/. Acesso em: 06 jun. 2023.

NOVAIS, Jean Carlos. Geometria Espacial: Conceitos Básicos. Geometria Espacial, 2015. Disponível em: https://matematicabasica.net/geometria-espacial. Acesso em: 06 jun. 2023.

SCIPY. [S. l.], 12 jul. 2021. Disponível em: https://tmfilho.github.io/pyestbook/math/03_scip.html. Acesso em: 6 jun. 2023.

SILVA, M. E. Apostila de Automação Industrial. Piracicaba, 2007

SILVA, Walisson. Como instalar a biblioteca do OpenCV no Python?. OpenCV, 2023. Disponível em: https://www.walissonsilva.com/posts/como-instalar-a-biblioteca-do-opencv-no-python. Acesso em: 06 jun. 2023.

SNAKE-CONTOUR-EXAMPLE.JPG. [*S. l.*], 23 dez. 2005. Disponível em: https://commons.wikimedia.org/wiki/File:Snake-contour-example.jpg. Acesso em: 23 jun. 2023.

SPATIAL algorithms and data structures (scipy.spatial). [S. l.], 30 ago. 2018. Disponível em: https://docs.scipy.org/doc/scipy/reference/spatial.html. Acesso em: 6 jun. 2023.

VALE, Giocane Maia Do ; POZ, Aluir Pordírio Dal. O PROCESSO DE DETECÇÃO DE BORDAS DE CANNY: FUNDAMENTOS, ALGORITMOS E AVALIAÇÃO EXPERIMENTAL. Presidente Prudente: Unesp, 2002.