Sistemas de Informação
6º Período
12/02/2021

IFMG - Ouro Branco
Diego Santos Seabra
0040251

Trabalho 1

Projeto e Análise de Algoritmos

Hospedado no Github

Disponível para visualização e interação no Heroku

Contents

0040251

1 Introdução

O objetivo deste trabalho foi a aplicação de um algoritmo de correlação no intuito de remoção de ruídos de imagens. O algoritmo em questão usa uma matriz (aqui denominada de máscara) quadrada ímpar (3x3, 5x5, 7x7, etc) para a realização dos cálculos.

Como imagens são compostas de *pixels* em uma tela, é possível representar uma imagem através de uma matriz contendo os valores de cores dos *pixels*. Neste trabalho, por questões de simplicidade, foram escolhidas somente imagens em tons de cinza, que variam o valor de cor de um pixel de 0 (preto) a 255 (branco).

Para fins de simplicidade, o algoritmo de *correlação* também será referido neste artigo como *média*.

Para extensão do trabalho requisitado inicialmente, foram implementados mais dois algoritmos (*Salt and Pepper* e *Intervalo*) cuja pretensão foi a de otimizar tanto o processo de remoção dos ruídos das imagens quanto o tempo de execução e a ordem de complexidade dos mesmos.

0040251

2 Desenvolvimento

Para que o usuário final pudesse interagir com os algoritmos implementados de forma fácil e intuitiva, foi criado um aplicativo web para visualização das imagens e suas respectivas correções; desenvolvido em C++ usando o *framework* ImGui (para a criação da interface gráfica) e OpenGL (para o carregamento e visualização das texturas). Para o gerenciamento dos dados das imagens foi utilizada a biblioteca stb_image.

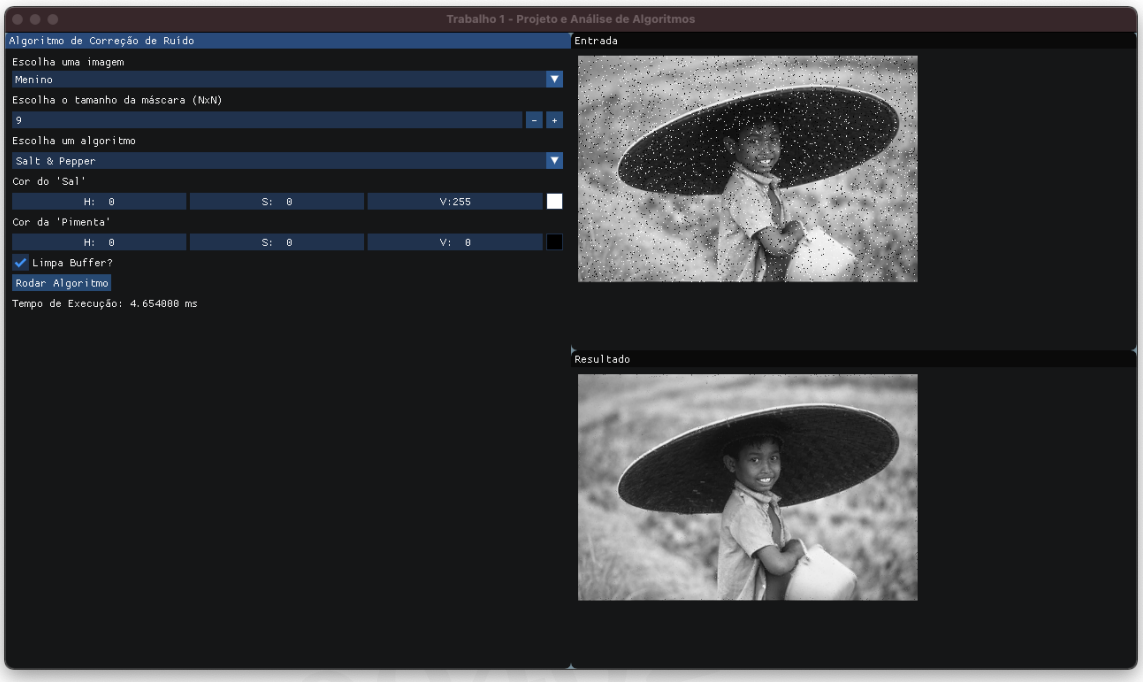

Fig. 1: Aplicativo em Funcionamento

No aplicativo web é possível selecionar uma imagem (pré carregada), o tamanho da máscara e o algoritmo de correção de ruído a ser aplicado. Caso queira, é possível acessar o referido aplicativo aqui.

2.1 Correlação (Média)

Como visto na introdução, o algoritmo de correlação usa uma matriz quadrada ímpar e passa *pixel* a *pixel* (até que a imagem acabe), realizando os seguintes passos.

1. Calcule a média de todos os *pixels* adjacentes ao atual, contidos dentro da máscara
2. Substitua o valor do *pixel* atual com o valor da média calculada
3. Vá ao próximo *pixel* (até que a imagem seja inteiramente lida e modificada)

Seu pseudocódigo pode ser descrito assim:

```
1: correlacao(Img, tamanhoMascara):
2: soma ← 0
3: media ← 0
4: limite ← (tamanhoMascara − 1)/2
5: for x ← 1 to tamanho[linha] do
6:    for y ← 1 to tamanho[coluna] do
7:       for i ← x − limite to x + limite do
8:          for j ← y − limite to y + limite do
9:             if i == tamanho[linha] then
10:               break
11:            end if
12:            if j == tamanho[coluna] then
13:               break
14:            end if
15:            soma ← soma + pixel(i, j)
16:         end for
17:      end for
18:      media ← soma/(tamanhoMascara × tamanhoMascara)
19:      substituiPixel(x, y, media)
20:      soma ← 0
21:   end for
22: end for
```

Deve-se ter em mente que este algoritmo, além de custoso (passa por **todos** os *pixels* de uma imagem), é também pouco eficiente pelo simples fato de aplicar uma correção para a imagem inteira sem qualquer tipo de regra em termos de identificação do ruído. Ele deveria ser usado então em casos onde o valor do ruído não é conhecido e se queira uma correção simplista.

0040251

Fig. 2: Imagem Original

Fig. 3: Imagem corrigida usando uma máscara de 3x3

Pelo exemplo demonstrado acima, a imagem corrigida fica apenas “borrada”.

2.2 Salt and Pepper

Já para as imagens onde os valores de ruído são conhecidos, como é o caso do *Salt and Pepper*, pode ser feita a implementação do mesmo algoritmo de correlação (média) porém alterando **somente** o *pixel* do ruído encontrado. Este exemplo é um dos mais famosos, caracterizado por ter ruídos brancos e pretos distribuídos na imagem (como se uma pessoa tivesse jogado sal e pimenta em cima da imagem).

Seu pseudocódigo pode ser descrito assim:

```
1: saltPepper(Img, tamanhoMascara, corSal, corPimenta):
2: soma ← 0
3: media ← 0
4: limite ← (tamanhoMascara − 1)/2
5: for x ← 1 to tamanho[linha] do
6:    for y ← 1 to tamanho[coluna] do
7:       if pixel(x, y) == corSal or pixel(x, y) == corPimenta then
8:          for i ← x − limite to x + limite do
9:             for j ← y − limite to y + limite do
10:               if i == tamanho[linha] then
11:                  break
12:               end if
13:               if j == tamanho[coluna] then
14:                  break
15:               end if
16:               soma ← soma + pixel(i, j)
17:            end for
18:         end for
19:         media ← soma/(tamanhoMascara × tamanhoMascara)
20:         substituiPixel(x, y, media)
21:         soma ← 0
22:      end if
23:   end for
24: end for
```

20402511

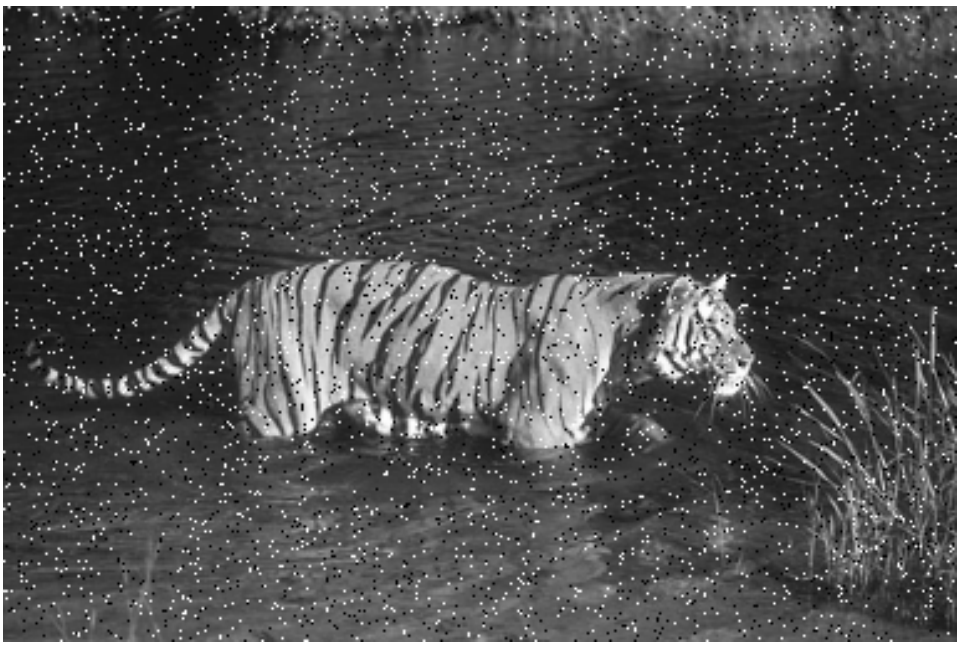

Fig. 4: Uma imagem com ruído

Fig. 5: Aplicação do Salt and Pepper com uma máscara de 9x9

Neste caso, os valores de ruído são, normalmente, 0 (preto) e 255 (branco), o que facilita na implementação e na aplicação da correção. Pelo demonstrado acima, é possível perceber que há uma diferença considerável quando se sabe os valores de ruído.

2.3 Intervalo

Há ainda os casos onde se quer corrigir uma imagem sabendo-se apenas que os ruídos se encontram em um intervalo fixo (ex.: de 60 a 90). Nestes casos foi implementado o algoritmo de correlação (média) porém alterando **somente** o *pixel* do ruído encontrado dentro do *intervalo*.

Seu pseudocódigo pode ser descrito assim:

1: **intervalo(Img, tamanhoMascara, corMin, corMax):**
2: $soma \leftarrow 0$
3: $media \leftarrow 0$
4: $limite \leftarrow (tamanhoMascara - 1)/2$
5: **for** $x \leftarrow 1$ to $tamanho[linha]$ **do**
6: **for** $y \leftarrow 1$ to $tamanho[coluna]$ **do**
7: **if** $pixel(x, y) \geq corMin$ and $pixel(x, y) \leq corMax$ **then**
8: **for** $i \leftarrow x - limite$ to $x + limite$ **do**
9: **for** $j \leftarrow y - limite$ to $y + limite$ **do**
10: **if** $i ==$ tamanho[linha] **then**
11: *break*
12: **end if**
13: **if** $j ==$ tamanho[coluna] **then**
14: *break*
15: **end if**
16: soma $\leftarrow soma + pixel(i, j)$
17: **end for**
18: **end for**
19: $media \leftarrow soma/(tamanhoMascara \times tamanhoMascara)$
20: $substituiPixel(x, y, media)$
21: $soma \leftarrow 0$
22: **end if**
23: **end for**
24: **end for**

0040251

Fig. 6: Uma imagem com ruído

Fig. 7: Aplicação do Intervalo com uma máscara de 9x9

3 Tempo de Execução

Em relação ao tempo de execução, para efeitos de conformidade com o pedido inicialmente no trabalho, os cálculos aqui demonstrados são referentes ao algoritmo de correlação (média). Para os cálculos dos tempos foram formadas 6 instâncias à partir de 3 máquinas, a saber:

Máquina 1 (Desktop)
Macbook Air (2017)
MacOS Big Sur (11.1)
1,8 GHz Dual-Core Intel Core i5
8 GB RAM
Intel HD Graphics 6000 1536 MB

Máquina 2 (Mobile)
Samsung Galaxy A70
Android 10.0
Qualcomm SDM675 Snapdragon 675
8 GB RAM
Adreno 612

Máquina 3 (Desktop)
Notebook CCE
Windows 10 Pro / Ubuntu LTS
2,30 GHz Intel Core i5
4 GB RAM
Intel HD Graphics 5000 1496 MB

0040251

As instâncias foram formadas da seguinte forma:

1. Máquina 1 / App rodando em modo de Debug
2. Máquina 1 / Safari
3. Máquina 1 / Google Chrome
4. Máquina 2 / Google Chrome
5. Máquina 3 / Windows 10 / Google Chrome
6. Máquina 3 / Ubuntu LTS / Firefox

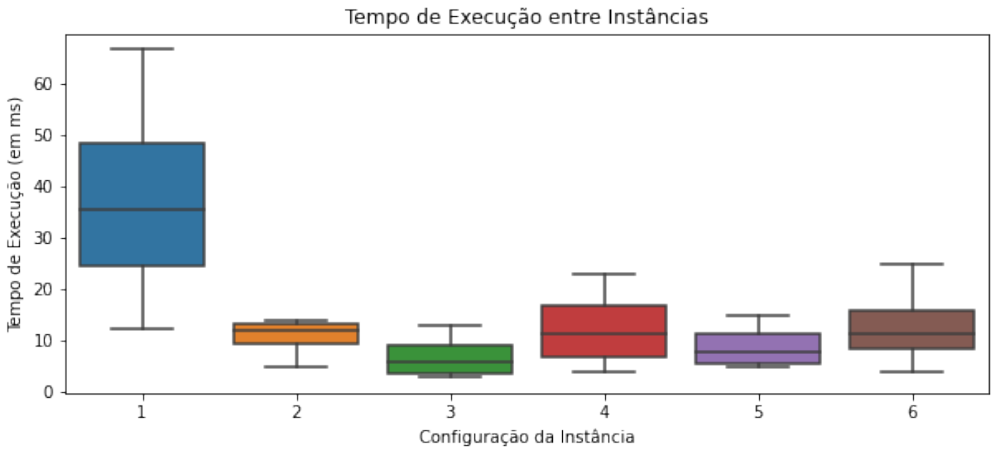

Fig. 8: Tempo de Execução entre Instâncias

Obs.: Acredita-se que o alto tempo de execução da máquina 1 se deve ao App estar rodando em modo de ***debug****.*

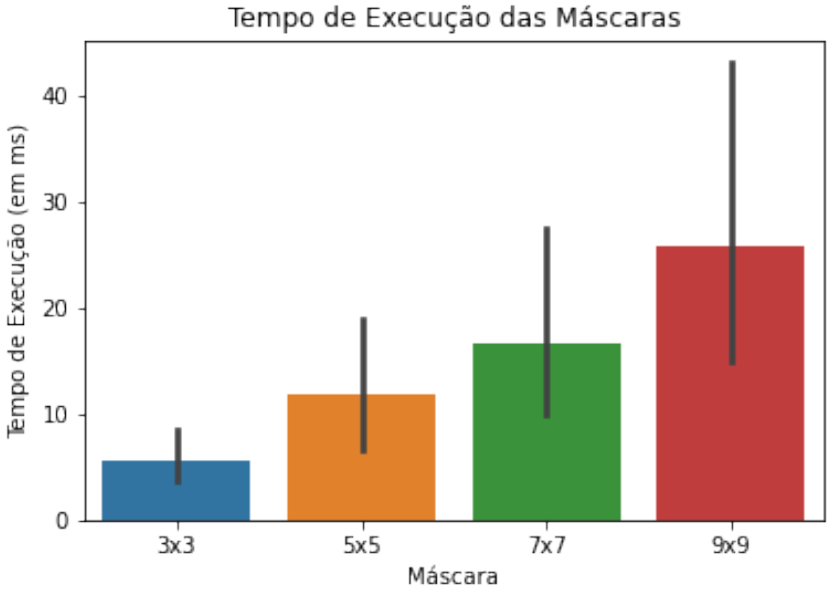

Fig. 9: Tempo de Execução entre Máscaras

Pode-se perceber pela figura acima que quanto maior a máscara aplicada, maior o tempo de execução; o que já era esperado pois com uma máscara maior, é necessário passar mais vezes por cada pixel, aumentando assim, a complexidade.

4 Análise de Complexidade

Em relação à análise de complexidade, para efeitos de conformidade com o pedido inicialmente no trabalho, os cálculos aqui demonstrados são referentes ao algoritmo de correlação (média) e é observado sempre o *pior caso*.

```
1:  correlacao(Img, tamanhoMascara):
2:  soma ← 0
3:  media ← 0
4:  limite ← (tamanhoMascara − 1)/2
5:  for x ← 1 to tamanho[linha] do
6:     for y ← 1 to tamanho[coluna] do
7:        for i ← x − limite to x + limite do
8:           for j ← y − limite to y + limite do
9:              if i == tamanho[linha] then
10:                break
11:             end if
12:             if j == tamanho[coluna] then
13:                break
14:             end if
15:             soma ← soma + pixel(i, j)
16:          end for
17:       end for
18:       media ← soma/(tamanhoMascara × tamanhoMascara)
19:       substituiPixel(x, y, media)
20:       soma ← 0
21:    end for
22: end for
```

Considerando que o pior caso possível é uma máscara do tamanho da própria imagem e levando em consideração que a máscara deve ser quadrada e ímpar, para esta análise também usaremos uma imagem com dimensões de linhas e colunas iguais e ímpares (ex.: 301x301).

Portanto, para o número de linhas ou colunas (tanto da imagem quanto da máscara), usaremos como representação a variável n.

0040251

Assim, temos os custos:

código	***custo***
inicialização de variavéis	1
for1 (para cada linha da imagem)	$n+1$
for2 (para cada coluna da imagem)	$n \times (n+1)$
for3 (para cada linha da máscara)	$n \times n \times (n+1)$
for4 (para cada coluna da máscara)	$n \times n \times n \times (n+1)$
calculo da média	$n \times n \times n \times n$

E o cálculo da complexidade como:

$$
\begin{aligned}
T(n) &= 1 + (n+1) + (n \times (n+1) + (n \times n \times (n+1)) + \\
&\quad (n \times n \times n \times (n+1)) + (n \times n \times n \times n)) \\
&= 1 + (n+1) + (n^2 + n) + (n^2 \times (n+1)) + (n^3 \times (n+1)) + n^4 \\
&= n + 2 + n^2 + n + n^3 + n^2 + n^4 + n^3 + n^4 \\
&= 2n + 2 + 2n^2 + 2n^3 + 2n^4 \\
&= \cancel{2n} + \cancel{2} + \cancel{2n^2} + \cancel{2n^3} + \cancel{2}n^4 \\
O(n) &= n^4
\end{aligned}
$$

0040251

5 Conclusão

Com certas imagens e a aplicação de determinados algoritmos, é possível obter resultados extremamente positivos:

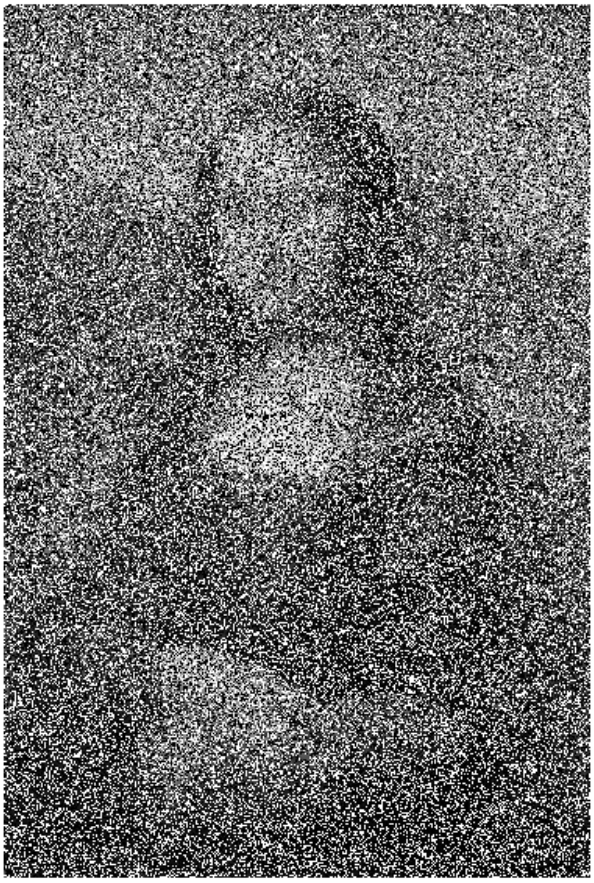

Fig. 10: Uma imagem com ruído

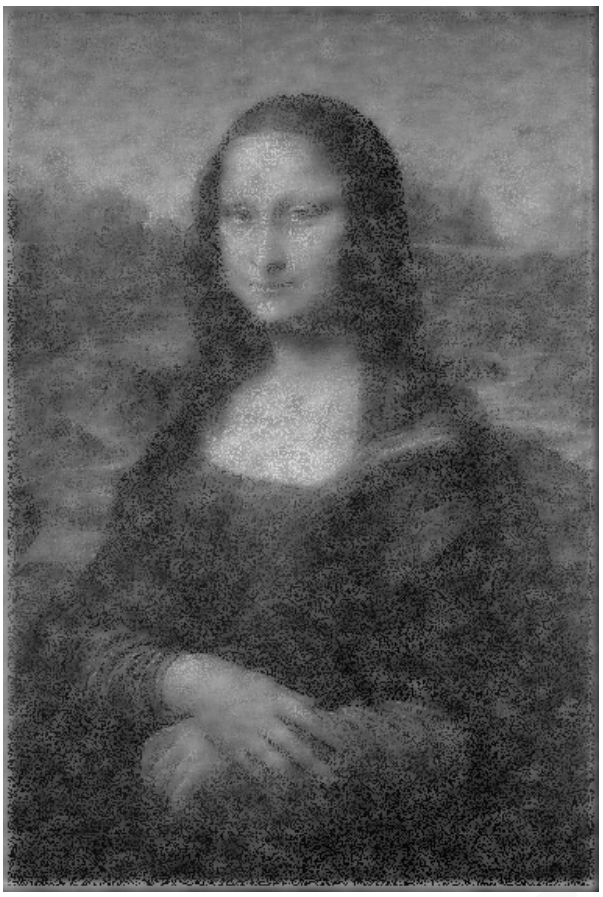

Fig. 11: Aplicação do Salt and Pepper com uma máscara de 11x11

Porém há outras imagens em que é simplesmente muito difícil de se realizar uma correção:

Fig. 12: Uma imagem com ruído

Fig. 13: Aplicação de Correlação (Média) com uma máscara de 7x7

Portanto, após testes com diferentes algoritmos e máquinas diferentes, é possível perceber que este algoritmo é geralmente muito custoso para a máquina que o executa ($O(n) = n^4$). Nos testes onde a máscara era acima de 13x13, houveram momentos em que a máquina ficou travada e não conseguiu se recuperar (sendo necessário o desligamento à força).

Outro ponto importante a ser ressaltado é que é bem mais fácil realizar a trativa de uma imagem cujo ruído já seja conhecido (o que não é muito comum no mundo real); então a escolha do algoritmo vai depender principalmente de qual imagem está sendo corrigida.

Após pesquisas sobre como o mercado implementa a correção de ruídos atualmente, foi observado que o uso dos algoritmos de machine learning e redes neurais, como pode ser visto aqui, aqui e aqui está "na moda" e demonstra resultados bem superiores aos conseguidos neste trabalho.

Como a implementação teve em seu foco a correção de imagens mais simples, pode-se afirmar que os resultados obtidos foram satisfatórios e dentro do esperado.

Referências Bibliográficas

[1] 8 ways to measure execution time in c/c++. https://levelup.gitconnected.com/8-ways-to-measure-execution-time-in-c-c-48634458d0f9.

[2] C programming - reading and writing images with the stb_image libraries. https://solarianprogrammer.com/2019/06/10/c-programming-reading-writing-images-stb_image-libraries/.

[3] Dear imgui. https://github.com/ocornut/imgui.

[4] docs.gl. https://docs.gl.

[5] Emscripten - docs. https://emscripten.org.

[6] Glm. https://github.com/g-truc/glm.

[7] How to load textures in opengl es efficiently. https://stackoverflow.com/questions/12244675/how-to-load-textures-in-opengl-es-efficiently.

[8] How to use opencv imshow() in a jupyter notebook. https://medium.com/@mrdatainsight/how-to-use-opencv-imshow-in-a-jupyter-notebook-quick-tip-ce83fa32b5ad.

[9] An introduction to the dear imgui library. https://blog.conan.io/2019/06/26/An-introduction-to-the-Dear-ImGui-library.html.

[10] Opencv - basic operations on images. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_core/py_basic_ops/py_basic_ops.html.

[11] Opencv - docs. https://docs.opencv.org/master/d4/da8/group__imgcodecs.html.

[12] Opengl. https://www.opengl.org/.

[13] Opengl 2.1 reference pages. https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/.

00/0251

[14] Opengl mathematics. https://glm.g-truc.net/0.9.9/index.html.

[15] Opengl wiki. https://www.khronos.org/opengl/wiki/Getting_Started.

[16] Python opencv - cv2.imread() method. https://www.geeksforgeeks.org/python-opencv-cv2-imread-method/.

[17] Simple directmedia layer wiki. https://wiki.libsdl.org.

[18] stb. https://github.com/nothings/stb.

[19] O. Campesato. *C Programming*. Mercury Learning and Information, 2019.

[20] V. Scott Gordon. *Computer Graphics Programming in OpenGL with C++*. Mercury Learning and Information, 2021.

[21] D. P. Kothari. *Mathematics for Computer Graphics and Game Programming*. Mercury Learning and Information, 2017.

0040251

Diego Santos Seabra
0040251