Projeto final

Visualização de Imagem Volumétrica
MC871

Isadora Sophia Garcia Rodopoulos
RA 158018

Professor
Alexandre Xavier Falcão

Unicamp, Universidade Estadual de Campinas

30 de Novembro de 2016

1 Introdução

A disciplina consistiu em uma distribuição de diversos projetos que dizem respeito à manipulação e a visualização de imagens volumétricas, com aplicação e exemplos ilustrados a partir de aplicações da área médica.

Os projetos realizados na disciplina consistem em:

1. Corte de imagem 2D a partir do modelo volumétrico;
2. Operações de brilho e contraste;
3. Coloração do modelo original a partir do modelo segmentado;
4. Operações de wireframe sobre as bordas do modelo volumétrico;
5. Corte planar do modelo, a partir de um ângulo e ponto de referência;
6. Reformatação planar;
7. Average e Maximum Intensity Projection;
8. Rendering de uma imagem, a partir do modelo de Phong;
9. Introdução de opacidade a partir do modelo segmentado.

2 Especificação dos projetos

2.1 Corte de imagem 2D

2.1.1 Algoritmo

Para a realização da extração do corte de imagem 2D, bastou-se extrair a componente $I(p)$ pertencente ao plano especificado pelo usuário dentro do modelo volumétrico. Dessa forma, o valor que será utilizado na imagem final estará dentro do intervalo de 0 a H, que corresponde a $I(p)$ normalizado no intervalo de H.

Assim, a realização do corte depende do plano (que pode ser definido pelo seu vetor normal ou pela especificação de seus eixos) e uma coordenada de origem. Para facilitar a interação do corte como API para fatias do radiologista, foi definida a seguinte assinatura de função:

```
int to2d_cut(MedicalImage* model, GrayImage** img, Cut type, int
cord)
```

- model: Imagem 3D a ser extraída o corte;
- img: Resultado do corte;
- type: Tipo de corte (axial, sagital ou coronal);
- cord: Coordenada de origem (opcional).

2.2 Brilho e contraste

2.2.1 Algoritmo

A aplicação de brilho e contraste se baseou na aproximação proposta em sala de aula. Basicamente, o usuário especifica dois valores **B** e **C**, de 0 a 100, que são tratados como:

$$B = (100 - B) * 0.01 * H; \tag{1}$$

$$C = (100 - C) * 0.01 * H; \tag{2}$$

Com base nesses valores, são estimados I_1 e I_2:

$$I_1 = (2 * B - C)/2; \tag{3}$$

$$I_2 = (2 * B + C)/2; \tag{4}$$

Finalmente, para cada pixel k, com intensidade $I \in \{0...H\}$, pertencente à imagem, é aplicada a seguinte relação:

$$k = \begin{cases} k_1, & \text{se } I < I_1, \\ \dfrac{(k_2 - k_1)}{(I_2 - I_1)}(I - I_1) + k_1, & \text{se } I_1 \leq I < I_2, \\ k_2, & \text{se } I \geq I_2. \end{cases}$$

Como proposto em sala de aula, define-se $k_1 = 0$ e $k_2 = H$.

2.3 Coloração

2.3.1 Algoritmo

A coloração da imagem é feita a partir do seguinte workflow:

1. O usuário passa o modelo original e o modelo segmentado como entrada;
2. O modelo segmentado é lido e transformado em um plano de corte 2D;
3. Como a imagem é tal que $I(p) \in \{0, 1, ..., C\}$, uma cor é associada a cada *label* da imagem através da tabela de cores RGB;
4. O modelo original é lido e uma função de normalização é aplicada de forma que cada pixel $I \in \{0...H\}$;
5. A imagem colorida é associada ao brilho de cada pixel do modelo original através da transformação de RGB para YCgCo para RGB;

Para a atribuição de cores do modelo segmentado, o modelo é lido e recebido como uma máscara de intensidade, em que $I(p) \in \{0, 1, ..., C\}$. Finalmente, para cada pixel k pertencente à imagem de extração do plano 2D do modelo segmentado, a seguinte cor é obtida:

$$
\begin{aligned}
V &\leftarrow I(p) \\
R &\leftarrow H * max\left\{0, \frac{(3 - |V - 4| - |V - 5|)}{2}\right\} \\
G &\leftarrow H * max\left\{0, \frac{(4 - |V - 2| - |V - 4|)}{2}\right\} \\
B &\leftarrow H * max\left\{0, \frac{(3 - |V - 1| - |V - 2|)}{2}\right\}
\end{aligned} \tag{5}
$$

O resultado após essas operações é apresentado abaixo:

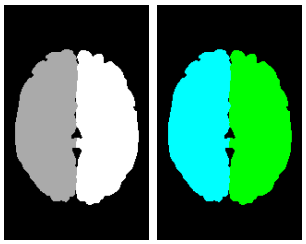


Figure 1: Imagem segmentada antes e depois da coloração

Assim, antes de aplicar as cores da imagem segmentada ao modelo original, é necessário realizar a extração do corte do modelo original com a escala e normalização das intensidades de cada pixel, de modo que ambas as imagens (segmentada e original) estejam no mesmo intervalo.

Logo, a seguinte função com seu respectivo algoritmo é utilizada para aplicar as cores da imagem segmentada ao modelo original:

```
void apply_mask(GrayImage* img, ColorImage* c_img)
```

1. Todos os valores de R, G, B (imagem segmentada) e $I(p)$ (imagem original) são normalizados de 0 a 1;
2. Os valores RGB são convertidos para $YCgCo$;
3. O Y é substituído por $I(p)$ e os valores de Cg e Co são multiplicados por $I(p)$;
4. Os novos valores de $YCgCo$ são convertidos novamente para RGB, ainda normalizados;
5. Multiplica-se os valores de RGB por H e obtemos a nova imagem.

O resultado final é apresentado abaixo:

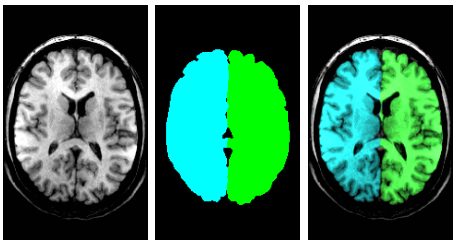


Figure 2: Aplicação da máscara colorida no modelo original

2.4 Demonstração: corte, coloração, brilho e contraste

Foi realizada uma aplicação em *python* que garantisse a interação mais fácil do usuário com os métodos de visualização da imagem volumétrica.

A partir dos códigos em C que realiza as manipulações gráficas, foi utilizada a interface em *python* **Tkinter** para oferecer botões e uma interação mais responsiva ao uso da API, bastando a comunicação com os programas já compilados em C.

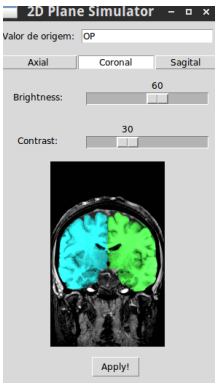


Figure 3: Exemplo da interface utilizada com o modelo **brain.scn**

2.5 Wireframe

2.5.1 Algoritmo

A implementação do wireframe foi feita a partir de um modelo e um vetor de visualização, ao qual é utilizado para extrair os ângulos θ_x e θ_y. O algoritmo se baseou em:

- Dada a transformação Φ em $P = (x, y, z)$:

$$\Phi(P) = T(q_c)R_y(\theta_y)R_x(\theta_x)T(-p_c)P^T \tag{6}$$

- Aplica a transformação Φ em cada um dos vetores das seis faces do modelo e descobre quais faces são visíveis;
- Aplica a transformação Φ nos vértices das faces visíveis;
- Realiza o algoritmo de **DDA** para todas as arestas visíveis, de $\Phi(P_1)$ a $\Phi(P_n)$.

2.5.2 Demonstração

O resultado foi ilustrado a partir da mesma library em *python* utilizada anteriormente, **Tkinter**. O usuário pode ajustar o vetor de visualização e produzir o wireframe correspondente, com uma cor gerada aleatoriamente.

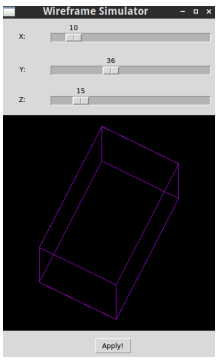


Figure 4: Exemplo da interface utilizada com o modelo **brain.scn**

2.6 Corte planar

2.6.1 Algoritmo

O corte planar é uma versão mais específica do corte em imagem 2D, apresentado anteriormente: ele permite a escolha de um ponto de referência p_1 e um vetor de visualização, garantindo maior liberdade de customização ao usuário. O algoritmo consiste em:

- Utiliza a transformação $\Phi^{-1}(P)$ em dado ponto $P = (x, y, z)$ e $q_c = (\frac{D}{2}, \frac{D}{2}, -\frac{D}{2})$, com D sendo a diagonal do modelo:

$$\Phi^{-1}(P) = T(p_1)R^{-1}T(-q_c)P^T \tag{7}$$

- Determina α_x, α_y e α_z para realizar as rotações adequadas;
- Aplica a transformação em todos os pontos P da imagem de tamanho $I = (D, D)$, de forma a capturar o pixel correspondente da cena;
- Utiliza algoritmo de interpolação para capturar intensidade adequada do pixel, a partir da cena.

2.6.2 Demonstração

Os testes para verificar corretude foram obtidos a partir do modelo de cubo, e posteriormente aplicados a modelos mais robustos a partir da interface interativa.

Figure 5: Cortes realizados na imagem cúbica

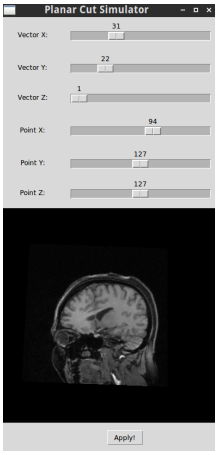


Figure 6: Exemplo de interface com o modelo **brain.scn**

2.7 Reformatação planar

2.7.1 Algoritmo

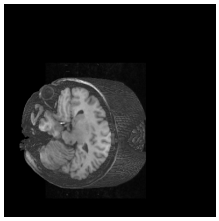
A reformatação planar proposta como projeto se baseia na obtenção de n cortes planares e **ortogonais** entre o segmento de p_1 e p_n. Dessa forma, a entrada fornecida pelo usuário se baseia na cena original, o número n de cortes, além dos pontos p_1 e p_n. O algoritmo se comporta em:

- Inicialmente, é obtido o vetor $\overrightarrow{v} = \dfrac{p_n - p_1}{||p_n - p_1||}$, utilizado como referência para realizar o corte planar.
- É estimado os valores (dx, dy, dz), com $dz < ||p_n - p_1||$. É aplicado um algoritmo parecido com o **DDA**, o qual decompõe λ em n valores, tais que $p_n = p_1 + n \cdot \lambda \overrightarrow{v}$.

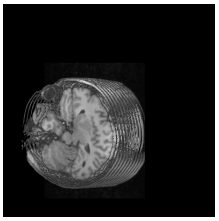
- Em seguida, são extraídos os n cortes da cena para cada valor de $i \cdot \lambda$, com $i \in \{1...n\}$. O corte planar resultante é acrescentado à imagem final, por meio de uma operação que verifica o valor de cada pixel entre o novo corte planar e a imagem resultante.
  - Para cada valor do pixel $p \in (D, I)$ em ambas as imagens (as quais possuem a mesma dimensão): caso o valor do pixel no novo corte seja maior que um determinado **threshold** (como, por exemplo, o valor 200), o valor é acrescentado à imagem final.

2.7.2 Demonstração

Como saída final, foram obtidos dois resultados distintos, uma vez que não estava muito claro qual seria o resultado esperado. O primeiro possui cortes de p_1 a p_n sucessivos, os quais possuem as dimensões interpoladas durante a extração dos cortes. O segundo possui cortes de p_1 a p_n em n intervalos, não necessariamente sucessivos.



(a) Cortes sucessivos



(b) n cortes

Figure 7: Resultados obtidos após a aplicação do algoritmo de reformatação planar

2.8 Average e Maximum Intensity Projection

2.8.1 Algoritmo

A realização de *Average* e *Maximum Intensity Projection* é feita a partir de um vetor de visualização, que representa o ponto de vista do usuário em relação ao modelo (utilizado para obter os ângulos θ_x e θ_y). O algoritmo consiste em:

- Utiliza a transformação $\Phi^{-1}(P)$ em dado ponto $P = (x, y, z)$, $q_c = (\frac{D}{2}, \frac{D}{2}, \frac{D}{2})$ e $p_c = (-\frac{x}{2}, -\frac{y}{2}, -\frac{z}{2})$, com D sendo a diagonal do modelo, x, y e z as dimensões que representam o tamanho do modelo:

$$\Phi^{-1}(P) = T(-p_c)R_x^{-1}(\theta_x)R_y^{-1}(\theta_y)T(-q_c)P^T \tag{8}$$

- Aplica a transformação em cada ponto P da imagem de tamanho $I = (D, D)$ e em $n = (0, 0, 1)$, de forma a obter o plano de visualização do usuário;

- Ainda em cada ponto P, aplica-se a seguinte relação de forma a descobrir o valores de λ para os pontos p_1 e p_n, para cada face $j \in \{1...6\}$:

$$\lambda = \frac{\overrightarrow{n_j} \cdot (c_j - \Phi^{-1}(P))}{\overrightarrow{n_j} \cdot \Phi^{-1}(n)} \tag{9}$$

  A partir de λ, é adquirido o ponto $Q = \Phi^{-1}(P) + \lambda \cdot \Phi^{-1}(n)$, se Q estiver dentro do domínio da cena, λ é adotado como um candidato válido para p_1 e p_n. O menor e maior λ encontrados correspondem a p_1 e p_n, respectivamente.

- Enfim, é aplicado o algoritmo de **DDA** para cada ponto P: para adquirir o *Maximum Intensity Projection*, basta retornar o maior valor encontrado, enquanto para adquirir o *Average Intensity Projection*, é retornado o valor médio entre todas as intensidades encontradas.

2.8.2 Demonstração

A verificação de corretude foi feita com o modelo **brain.scn**, a partir da mesma interface em *python* aplicada nos projetos anteriores.

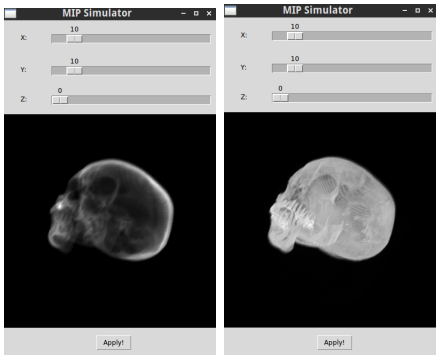


Figure 8: Exemplo de interface com o modelo **brain.scn** com *Average* e *Maximum Intensity Projection*, respectivamente

2.9 Rendering

2.9.1 Algoritmo

O processo de realização do *rendering* se originou de um algoritmo bastante semelhante ao apresentado anteriormente, utilizado para extrair o *Average* e *Maximum Intensity Projection*. As únicas diferenças estão no input (uma máscara de *labels* é fornecida como entrada, além da cena original) e no último passo, ao aplicar o algoritmo **DDA**, que se comporta da seguinte maneira:

- Após obter os pontos p_1 e p_n, o algoritmo **DDA** é aplicado. Para cada ponto p percorrido na função **DDA**, é verificado se os seus vizinhos esféricos de raio 1 possuem uma *label* de objeto. Caso positivo, esse ponto vizinho é tomado como novo ponto de referência p e é retornado $L(p)$ como valor final.

- O modelo de Phong aplicado em p, $L(p)$, corresponde a:

$$L(p) = k_a L_a + D(p)(k_d \cos(\theta) + k_s \cos^{n_s}(2\theta)) \tag{10}$$

  No modelo, foi considerado $k_a = 0.2$, $k_d = 0.5$, $k_s = 0.3$ e $n_s = 5$. L_a corresponde ao valor máximo de intensidade, isto é, H. Além disso, de forma a adquirir o valor correto de $L(p)$:

  - O valor de N, que seria a normal do ponto p na cena, foi calculado a partir dos vizinhos esféricos de p, utilizando raio 6. Dessa forma, para cada um dos pontos vizinhos v, é somado à normal a derivada entre p e v. Isto é:

$$N = \sum_{p_i=1}^{v} \frac{f(p_i) \cdot \Delta_i}{2 \cdot ||\Delta_i||} \tag{11}$$

    Em que $\Delta_i = p_i - p$.

  - O valor de θ, isto é, o ângulo entre o ponto de vista do usuário e a normal, foi obtido a partir de:

$$\theta = \cos^{-1}(-N \cdot \Phi^{-1}(n)) \tag{12}$$

    Em que N é a normal no ponto p e $\Phi^{-1}(n)$ se trata do vetor de visualização, já rotacionado. Além disso, $L(p)$ só é calculado se $0 \leq \theta \leq \frac{\theta}{2}$, enquanto a componente especular (k_s) é calculada se $0 \leq \theta \leq \frac{\theta}{4}$.

  - Finalmente, o valor de $D(p)$ é obtido como:

$$D(p) = H(1 - 0.8\frac{||p - \Phi^{-1}(q)||}{D} + 0.2) \tag{13}$$

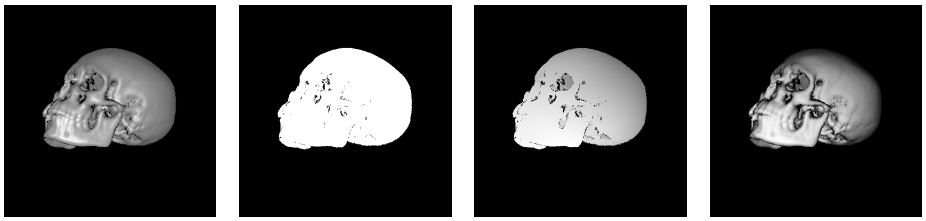
    Em que $\Phi^{-1}(q)$ é o ponto do plano de visualização (rotacionado e transladado), enquanto D se trata da diagonal da cena.

2.9.2 Demonstração

A seguir estão os resultados de cada uma das componentes utilizadas para o cálculo de $L(p)$, garantindo o resultado do *rendering*.

Além disso, também foi aplicada uma interface para a realização de testes e facilitar a manipulação dos resultados, também demonstrada a seguir.

É possível observar que o resultado final possui alguns pixels irregulares (com a mesma cor do fundo) entre as bordas do modelo. A ocorrência desses pixels se



(a) $L(p)$ (b) $k_a L_a$ (c) $D(p)$ (d) $D(p)k_d \cos(\theta)$

(e) $D(p)k_s \cos^{n_s}(2\theta)$

Figure 9: Demonstração de cada estágio do processo de renderização

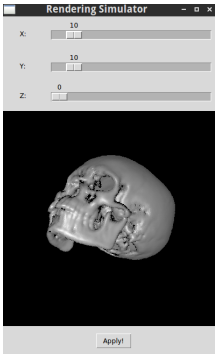


Figure 10: Interface com o modelo **skull.scn** para o *rendering*

deve, principalmente, à precisão da normal, derivada da aproximação do ponto p. De modo a corrigir este problema, deveria ter sido aplicado uma função de interpolação entre todas as normais vizinhas ao ponto p, ainda no **DDA**, além da identificação da máscara adequada, também utilizando a interpolação. Como o processamento iria tomar muito tempo com esse procedimento, o qual exigiria técnicas de paralelização para uma execução viável, o resultado final foi o aproximado.

2.10 Rendering com opacidade

2.10.1 Algoritmo

Novamente, o algoritmo para a aplicação do *rendering* com opacidade se deriva do próprio algoritmo do *rendering*, demonstrado anteriormente. As entradas são as mesmas (vetor de visualização, cena original e cena com máscaras de objeto) - entretanto, a diferença está na forma em que o **DDA** retorna o valor de intensidade:

- Para cada ponto p percorrido na função **DDA**, é verificado se os seus vizinhos esféricos de raio 1 possuem valor diferente da *label* atual, diferente da *label* do fundo da cena e que não tenha sido percorrido anteriormente.
  - Caso positivo, esse ponto vizinho é tomado como novo ponto de referência p.
    - Caso seja a primeira ocorrência de um novo objetivo, é adicionado $\alpha_1 L(p_1)$ ao valor final.
    - Caso contrário, é adicionado $\alpha_i L(p_i) + \prod_{j=1}^{i-1}(1 - \alpha_j)$ ao valor final. Cada valor $\alpha_j \in [0, 1]$ corresponde ao valor de opacidade do objeto j.
  - Caso contrário, apenas ignore e continue a função.
- O valor final é retornado após atingir o ponto p_n.

2.10.2 Demonstração

Os resultados e problemas encontrados são semelhantes aos do *rendering*. A seguir, é apresentada a interface com o modelo **brain.scn**, o qual possui diferentes objetos na cena e permite uma melhor visualização de opacidade.

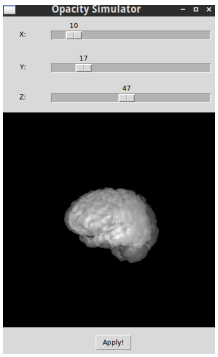


Figure 11: Interface com o modelo **brain.scn** para o *rendering* com opacidade (em que cada α_j possui valor 0.4)

3 Conclusão

Foram apresentados os diversos projetos apresentados na disciplina que dizem respeito à manipulação de imagens volumétricas. Finalmente, é importante ressaltar que todo código e material utilizado se encontra no repositório da disciplina [1].

References

[1] I. Sophia, "Projeto final de mc871," `https://github.com/isadorasophia/MC871`.