

# Análise de Métodos para Processamento de Imagens

Pedro Pillon Vanzella

18 de junho de 2015

## 1 Introdução

Dentre as muitas aplicações dos métodos numéricos, uma das mais populares é o processamento de imagens [1]. Estes métodos são empregados em um número de meios, como fotografia digital, tratamento de imagens, produção de vídeo e jogos.

Os problemas de redimensionamento e rotação de imagens são alguns dos mais simples e ao mesmo tempo mais comuns. Analisemos o problema de um jogo: imagens são utilizadas como texturas em praticamente todos os polígonos e um jogo. É possível que elas devam ser redimensionadas ou rotacionadas a cada quadro, e potencialmente milhares delas serão exibidas ao mesmo tempo. É interessante, então, que se faça uso de algoritmos de custo computacional mais baixo possível, de modo a tornar a experiência do jogo mais fluida e agradável.

Para este fim, utilizaremos o ambiente GNU Octave, que é um software livre para a computação numérica e científica [2].

Para os métodos de redimensionamento analisaremos as interpolações *Nearest Neighbor* (Seção 2.1), Bicúbica (Seção 2.3) e Bilinear (Seção 2.2). Para os métodos de rotação, analisaremos os métodos de Rotação por *Nearest Neighbor* (Seção 3.1), por Interpolação Bilinear (Seção 3.2) e por Interpolação Bicúbica (Seção 3.3).

Utilizaremos sempre a mesma imagem como base, de modo a podermos comparar os resultados. Podemos vê-la na Figura 1.

Será feita uma comparação do tempo de execução de cada algoritmo, bem como da qualidade da imagem gerada, de modo a tomar uma decisão quanto à melhor técnica para o caso hipotético de texturas em um jogo 3D.



Figura 1: Uma simpática vaquinha

## 2 Redimensionamento de Imagens

Redimensionamento de imagens é um processo não trivial, que envolve um balanço entre eficiência e qualidade de resultado [5].

Dentre os vários algoritmos existentes para esta tarefa, vamos abordar os três mais comuns [3]: Interpolação por *Nearest Neighbor*, Interpolação Bilinear e Interpolação Bicúbica.

É importante analisarmos exatamente este balanço de qualidade e performance. Uma solução que gere resultados perfeitos, mas demore muito não seria útil, pois deve ser executada milhares de vezes a cada quadro, e isto reduziria a performance do jogo. De mesmo modo, uma solução mais rápida não é útil caso gere resultados muito ruins, ou introduza artefatos que causem discontinuidades nas texturas, pois isso quebraria a ilusão de 3D no jogo.

Desta forma, a escolha de melhor método é, possivelmente, uma combinação de avaliações objetivas e subjetivas. Descarta-se as soluções que geram resultados pouco visualmente agradáveis<sup>1</sup>, e elege-se como melhor o que, dentre os que sobraram, traz resultados mais rápidos<sup>2</sup>.

### 2.1 Interpolação por *Nearest Neighbor*

Interpolação por *Nearest Neighbor* é o método mais simples de se implementar, bem como o mais rápido [3].

A idéia do algoritmo de *Nearest Neighbor* é simples - simplesmente copiar os pixels mais próximos para o lugar dos que estão sendo adicionados ao se ampliar a imagem. Isto gera resultados rápidos, porém de baixa qualidade, como podemos ver na Figura 2.c. Há um claro serrilhado que é introduzido ao se ampliar a imagem.

Reduzir uma imagem com este algoritmo também gera resultados pouco satisfatórios: novamente serrilhado é introduzido e detalhes são perdidos. Podemos ver isto na Figura 2.a.

---

<sup>1</sup>uma avaliação completamente subjetiva

<sup>2</sup>uma avaliação objetiva



Figura 2: Redução e Ampliação com algoritmo de *Nearest Neighbor*

A Tabela 1 mostra o tempo de execução para redução e ampliação em 50% da imagem da Figura 2.b. Estes dados não nos dizem muito por enquanto, mas serão úteis quando compararmos aos resultados das Seções 2.2 e 2.3. Podemos apenas ver que reduzir uma imagem demora menos que ampliar a mesma. Isto se dá pelo fato de haver menos pixels a serem calculados na imagem reduzida, e é esperado.

Tabela 1: Tempo de execução

Redução	Ampliação
0.17s	0.21s

O código em Octave que faz esta interpolação pode ser visto na Figura 3.

```

1 pkg load image;
2
3 I = imread("cow_very_small.png");
4
5 nI = imresize(I, 0.5, "nearest");
6 imwrite(nI, "cow_nearest_smallest.png");

```

Figura 3: Código-Fonte para interpolação *Nearest Neighbor* no Octave

O importante na listagem da Figura 3 é a linha 5, onde uma chamada para `imresize()` é feita. Esta função encapsula a chamada para a função `interp2()`, apenas adicionando algumas garantias de que o primeiro parâmetro é uma imagem válida e verificando se há múltiplos canais<sup>3</sup> e, se houver, executando a interpolação para cada um dos canais individualmente.

A função `interp2()`, por sua vez, recebe um parâmetro (passado pela `imresize()`) que diz qual tipo de interpolação deve ser feita. Neste caso, escolheu-se *nearest*.

O segundo parâmetro da função `imresize()` é o fator de escala. Um fator menor que 1 resultará em uma imagem reduzida. De mesmo modo, um valor superior a 1 resultará em uma ampliação da imagem original.

<sup>3</sup>i.e. a imagem está em RGB

## 2.2 Interpolação Bilinear

A idéia da Interpolação Bilinear é executar uma interpolação linear em uma direção e, em seguida, uma interpolação linear na outra direção.

Conforme [3], podemos calcular a interpolação da seguinte maneira:

Considere quatro pontos,  $A$ ,  $B$ ,  $C$  e  $D$ . Suas coordenadas são, respectivamente,  $(i, j)$ ,  $(i, j + 1)$ ,  $(i + 1, j)$ ,  $(i + 1, j + 1)$ . Para calcular o ponto  $P(u, v)$ , primeiro vamos calcular um ponto  $E$ , que é a interpolação linear de  $A$  e  $B$ .

$$f(i, j + v) = [f(i, j + 1) - f(i, j)]v + f(i, j)$$

Agora vamos calcular um ponto  $F$ , que é a interpolação linear de  $C$  e  $D$ .

$$f(i + 1, j + v) = [f(i + 1, j + 1) - f(i + 1, j)]v + f(i + 1, j)$$

Finalmente, basta calcular a interpolação linear de  $E$  e  $F$  para obter  $P$ .

$$f(i+u, j+v) = (1-u)(1-v)f(i, j) - (1-u)vf(i, j+1) + u(1-v)f(i+1, j) + uvf(i+1, j+1)$$

Isto deve ser feito, no caso de uma imagem colorida, para cada canal da imagem.

Podemos ver na Figura 4 os resultados de uma redução em 50% (Figura 4.a) e uma ampliação em 50% (Figura 4.c).

A imagem ampliada apresenta um serrilhado em todas as bordas, e a definição da grama é claramente perdida.

O maior problema se encontra na imagem reduzida: uma borda preta foi gerada na imagem, devido ao processo de interpolação. Isto torna este método inútil para a redução de texturas em um jogo.



Figura 4: Redução e Ampliação com algoritmo de Interpolação Bilinear

Na Tabela 2 vemos o tempo de execução para a redução e a ampliação utilizando interpolação bilinear. Novamente, a redução é mais rápida que a ampliação, por haver menos pixels a serem calculados.

O tempo de execução da redução consta na Tabela 2 somente para referência, já que o resultado da mesma não é útil para o problema proposto.

Tabela 2: Tempo de execução

Redução	Ampliação
0.15s	0.23s

Vemos na listagem da Figura 5 o código para fazer uma interpolação bilinear em uma imagem com o Octave.

```

1 pkg load image;
2
3 I = imread("cow_very_small.png");
4
5 nI = imresize(I, 0.5, "linear");
6 imwrite(nI, "cow_bilinear_smallest.png");

```

Figura 5: Código-Fonte para interpolação bilinear no Octave

O código é praticamente igual ao da Figura 3, mudando apenas o parâmetro do método de interpolação.

A função `imresize()` novamente chamará a função `interp2()`, passando desta vez o parâmetro *linear*.

## 2.3 Interpolação Bicúbica

A interpolação bicúbica é, das três testadas neste artigo, a mais complexa – tanto em implementação quanto computacionalmente. [3]

De acordo com [4], podemos calcular a interpolação bicúbica com uma convolução. Entretanto, o Octave implementa uma Interpolação de Lagrange de grau 3 [2] para este fim.

O Polinômio de Lagrange pode ser calculado da seguinte maneira [2]:

$$P(x) = \sum_{j=1}^n P_j(x)$$

Onde

$$P_j(x) = y_j \prod_{\substack{k=1 \\ k \neq j}}^n \frac{x - x_k}{x_j - x_k}$$

Podemos ver na Figura 6 os resultados da interpolação bicúbica.

A Figura 6.a é uma redução em 50%. Notamos nela uma borda preta introduzida pela interpolação. Este resultado não nos serve, pois geraria discontinuidades nas texturas<sup>4</sup>.

A Figura 6.c é uma ampliação em 50%. O resultado é bastante satisfatório, com pouco serrilhado. O nível de detalhe da grama e das montanhas ao fundo também é bom.

---

<sup>4</sup>Texturas são muito frequentemente replicadas lado-a-lado em superfícies de jogos, e qualquer discontinuidade estraga a ilusão da superfície



Figura 6: Redução e Ampliação com algoritmo de Interpolação Bicúbica

Na Tabela 3 vemos os tempos de execução deste algoritmo. Novamente o resultado da redução consta somente como referência, pois a imagem gerada não atende os requisitos do problema.

Tabela 3: Tempo de execução

Redução	Ampliação
0.19s	0.29s

A Figura 7 mostra o código em Octave para realizar a interpolação bicúbica em uma imagem.

```

1 pkg load image;
2
3 I = imread("cow_very_small.png");
4
5 nI = imresize(I, 0.5, "cubic");
6 imwrite(nI, "cow_cubic_smallest.png");

```

Figura 7: Código-Fonte para interpolação bicúbica no Octave

Este código é, novamente, muito similar ao da Figura 5 e da Figura 3. A única diferença é na chamada para `imresize()`, onde se passa o parâmetro `cubic`, que indica que é necessário chamar `interp2()` com o mesmo parâmetro.

## 2.4 Comparação de Resultados

Podemos ver na Tabela 4 uma comparação do tempo de execução dos três algoritmos.

Dada a avaliação subjetiva do resultado da imagem, já descartamos as reduções por Interpolação Bilinear (Seção 2.2) e por Interpolação Bicúbica (Seção 2.3), que podem ser vistas nas Figuras 4.a e 6.a, respectivamente.

Isto nos deixa a clara escolha do método de *Nearest Neighbor* para a redução de imagens, no nosso hipotético jogo.

	<i>Nearest Neighbor</i>	Bilinear	Bicúbica
Redução	0.17s	0.15s	0.19s
Ampliação	0.21s	0.23s	0.29s

Tabela 4: Resultados consolidados

A Tabela 4 nos mostra que este método não é necessariamente o mais rápido, mas esta diferença é negligenciável e pode muito bem se dar por alguma peculiaridade do momento de execução dos testes.

Para a ampliação, no entanto, a escolha não é tão trivial. Claramente o resultado da Interpolação Bilinear (Figura 4.c) é melhor que o obtido pelo método de *Nearest Neighbor* (Figura 2.c). Como a diferença de tempo de execução entre ambos é negligenciável, se a escolha devesse ser feita somente entre eles, a vitória seria da Interpolação Bilinear.

Entretanto, comparando o resultado da Interpolação Bilinear com a Interpolação Bicúbica (Figuras 4.c e 6, respectivamente), notamos que o resultado da segunda é claramente melhor que o da primeira – arestas são melhor definidas, principalmente. O problema se encontra no tempo de execução: o algoritmo de Interpolação Bicúbica é consideravelmente mais lento que o de Interpolação Linear.

Não há, então, uma escolha que atenda todos os casos. Num jogo onde menos texturas são utilizadas, ou há um *cache* das mesmas, resultando em poucos redimensionamentos, o custo maior do algoritmo de Interpolação Bicúbica pode compensar a qualidade inferior da Interpolação Bilinear. Caso muitas texturas diferentes sejam utilizadas, é possível que utilizar a Interpolação Bicúbica crie um gargalo na aplicação, causando lentidões e prejudicando a performance do jogo.

### 3 Rotação

Novamente temos um problema não trivial, com um balanço entre qualidade e eficiência [5].

Novamente, compararemos três métodos para realizar a rotação: O método de *Nearest Neighbor*, descrito na Seção 3.1; o método de Interpolação Bilinear, descrito na Seção 3.2 e o método de Interpolação Bicúbica, descrito na Seção 3.3.

Os algoritmos são os mesmos que os apresentados na Seção 2, apenas aplicados de maneira diferente.

#### 3.1 Rotação por *Nearest Neighbor*

Podemos ver o resultado na Figura 11.b.

Na Figura 8 vemos o código-fonte que executa a interpolação por *Nearest Neighbor* para realizar a rotação, no Octave.

```

1 pkg load image;
2
3 I = imread("cow_very_small.png");
4
5 nI = imrotate(I, 45, "nearest");
6 imwrite(nI, "cow_nearest_rotated.png");

```

Figura 8: Código-Fonte para rotação por *Nearest Neighbor* no Octave

A função `imrotate()`, bem como a `imresize()` (vista na Seção 2, chama a função `interp2()`). Ela recebe como parâmetros, em ordem, a imagem, o ângulo de rotação (em graus), e o método de interpolação, que é passado para a `interp2()`. [2]

### 3.2 Rotação por Interpolação Bilinear

O resultado da rotação pode ser visto na Figura 11.c.

A Figura 9 mostra o código em Octave para fazer esta rotação.

```

1 pkg load image;
2
3 I = imread("cow_very_small.png");
4
5 nI = imrotate(I, 45, "linear");
6 imwrite(nI, "cow_bilinear_rotated.png");

```

Figura 9: Código-Fonte para rotação por interpolação bilinear no Octave

A única diferença deste código para o mostrado na Figura 8 é o parâmetro da interpolação. Na Figura 8 é passado *nearest*, e neste é passado *linear*. Este parâmetro é propagado para a função `interp2()`, que é a responsável pela interpolação.

### 3.3 Rotação por Interpolação Bicúbica

A Figura 11.d mostra o resultado da rotação utilizando Interpolação Bicúbica.

Na Figura 10 podemos ver o código em Octave para executar esta rotação.

```

1 pkg load image;
2
3 I = imread("cow_very_small.png");
4
5 nI = imrotate(I, 45, "cubic");
6 imwrite(nI, "cow_cubic_rotated.png");

```

Figura 10: Código-Fonte para rotação por interpolação bicúbica no Octave

Mais uma vez, a única diferença deste código para o da Figura 8 é o parâmetro do tipo de interpolação, que aqui é *cubic*. A chamada é a mesma, e novamente a função `imrotate()` chamará a função `interp2()` para executar a interpolação.

O Octave utiliza uma interpolação de Lagrange de grau 3 para cada eixo da imagem, em cada um dos três canais [2].



### 3.4 Comparação de Resultados

Na Figura 11 podemos ver, lado a lado, o resultado das três rotações sobre a mesma imagem, com o mesmo ângulo.



(a) Original



(b) *Nearest Neighbor*



(c) Bilinear



(d) Bicúbica

Figura 11: Rotação em 45° por diferentes métodos

A primeira coisa que fica evidente é a borda preta introduzida pela rotação. Isto não é um problema em um jogo 3D – a imagem ganhou esta borda preta por artefato do formato de exibição [2].

Analisando o resultado das imagens, junto dos tempos de execução da Tabela 5, vemos que, entre a Interpolação Bilinear (Figura 11.c) e a Interpolação por *Nearest Neighbor* (Figura 11.b), vale mais a pena escolher a primeira. A imagem resultante é claramente melhor no caso da Interpolação Bilinear, apresentando menos serrilhados nas bordas, levando essencialmente o mesmo tempo para ser calculada.

<i>Nearest Neighbor</i>	Bilinear	Bicúbica
0.62	0.63	2.11

Tabela 5: Tempo de execução para rotações (em segundos)

Comparando a Interpolação Bicúbica (Figura 11.d) com a Interpolação Bilinear (Figura 11.c), no entanto, temos uma decisão mais difícil. Por um lado, a Interpolação Bicúbica gera resultados ainda melhores, com arestas mais claramente definidas e menor perda de detalhes complexos como a grama. Por outro lado, o tempo total de execução é mais de 3 vezes maior. Isto pode ser uma perda de performance inaceitável em um jogo, dependendo da quantidade de texturas diferentes e a capacidade de se fazer *cache* das mesmas.

## 4 Considerações Finais

A escolha do método de interpolação para realizar operações de ampliação, redução e rotação sobre imagens pode causar um impacto enorme na performance de um jogo, bem como no visual dele, e, portanto, deve ser feita com cuidado.

A partir da análise feita neste artigo, ficou claro que não há um método que seja superior aos outros em todos os casos. Por vezes, limitações de tempo de execução farão com que a escolha seja pelo método mais rápido, sacrificando qualidade de resposta. Em outros casos, a qualidade da imagem gerada é importante, e temos que lidar com um tempo de cálculo maior, tentando otimizar outros pontos do código.

Há, ainda, muitos outros métodos de interpolação que não foram analisados neste artigo, como as *splines*. Para rotações, há ainda a possibilidade de fazê-las através de multiplicação de matrizes, entre outros métodos. Uma análise mais aprofundada destes métodos, comparando os resultados tanto em qualidade quanto em tempo de execução é interessante.

Novamente, mesmo com a introdução de novos métodos, é possível que os métodos mais simples ainda sejam interessantes em alguns casos, principalmente aqueles onde a qualidade de resposta não é tão importante.

No geral, vemos a importância de uma análise cuidadosa de todos os métodos numéricos, de um ponto de vista de computação científica, ao aplicá-los mesmo a problemas aparentemente distantes da computação científica, clássica e acadêmica, como o desenvolvimento de jogos.

## Referências

- [1] Tinku Acharya e Ajoy K. Ray. *Image Processing - Principles and Application*. 1st. Wiley, 2005.
- [2] John W Eaton, David Bateman e Soren Hauberg. *GNU Octave Manual Version 3*. Network Theory Ltd., 2008. ISBN: 095461206X, 9780954612061.
- [3] Dianyuan Han. “Comparison of Commonly Used Image Interpolation Methods”. Em: *ICCSEE, Hangzhou, China* (2013).
- [4] R. Keys. “Cubic convolution interpolation for digital image processing”. Em: *Acoustics, Speech and Signal Processing, IEEE Transactions on* 29.6 (dez. de 1981), pp. 1153–1160. ISSN: 0096-3518. DOI: 10.1109/TASSP.1981.1163711.
- [5] Johannes Kopf e Dani Lischinski. “Depixelizing Pixel Art”. Em: *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011)* 30.4 (2011), 99:1–99:8.