UNIVERSIdade FEEVALE

MICHAEL DE REZENDE BECKER

Apresentando *pixel arts* em altas resoluções

Novo Hamburgo

2012

MICHAEL DE REZENDE BECker

Apresentando *pixel arts* em altas resoluções

Trabalho de Conclusão de Curso

apresentado como requisito parcial

à obtenção do grau de Bacharel em

Ciência da Computação pela

Universidade Feevale

Orientadora: Marta Rosecler Bez

Novo Hamburgo

2012

Agradecimentos

Gostaria de agradecer a todos os que, de alguma maneira, contribuíram para a realização desse trabalho de conclusão, em especial:

A Prof. Marta Rosecler Bez minha orientadora, exemplo de profissional e pessoa, a qual para este aluno tornou-se parâmetro de excelência na atividade acadêmica. De maneira muito segura e investigadora poliu as ideias aqui expostas e guiou de maneira extremamente determinante, me encorajando na escolha dos caminhos seguidos.

A toda minha família, por serem indispensáveis: em especial aos meus pais, Irineu Darcy Becker e Sandra Becker, por todo apoio que a mim foi dado, por acreditarem no meu potencial e me fazerem acreditar nos meus sonhos e principalmente por desde sempre manter e estimular todas as condições necessárias ao meu desenvolvimento acadêmico, pessoal e profissional.

A minha namorada Fernanda Carvalho, a minha irmã Daiane Becker e ao meu primo Diego Rezende Silva, por todo o apoio, ajuda e compreensão durante este período da minha graduação.

Resumo

É sabido que imagens em altas resoluções são o padrão na atualidade. Entretanto, existe uma vasta quantidade de material que foi produzido com limitações de resolução. Para reutilizar estas imagens desenvolvidas com um baixo número de *pixels*, sem a perda das intenções inicias do artista, são necessários algoritmos que melhorem a sua visualização em dimensões maiores. Para isto, é preciso aumentar o número de *pixels* destas imagens, ou convertê-las para uma representação vetorial, independente de resolução. Dessa forma, este trabalho se propõe a realizar uma pesquisa sobre os diversos métodos existentes para melhoria da qualidade gráfica de *pixel arts* e, por fim, apresentar o desenvolvimento de uma solução para vetorização destas imagens digitais. Os resultados comprovam que o processo de remodelagem dos *pixels* seguido da sua representação em formato vetorial e utilizando curvas de Cat-mull, apresenta uma qualidade superior às demais abordagens que ampliam o número de *pixels*.

Palavras Chave: *Pixel art*; Vetorização; *Upscaling*.

Abstract

It is known that images at high resolutions are the standard today. However, there is a vast amount of material that was produced with limitations in its resolution. To reuse these images developed with a low number of pixels, without losing the initial intentions of the artist, specialized algorithms are needed to improve their visualization in higher resolutions. To do so, is necessary increase the number of pixels in the image, or convert these images to a vector representation, resolution independent. In this way, this study aims to realize a research over the existent methods to improve the graphical quality of pixel arts and, finally, the development of a solution to vectorize these digital images. The results show that the process of remodeling the pixels cells followed by their representation in vector format and using curves of Cat-mull, features a superior quality than other approaches that expand the number of pixels.

Key words: Pixel art; Vectorization; Upscaling.

Lista de Figuras

[Figura 1.1 - *Super Mario Bros*. 14](#_Toc340040770)

[Figura 1.2 - *Pacman* e *Zelda*. 14](#_Toc340040771)

[Figura 1.3 – *Megaman* e *Final Fantasy*. 15](#_Toc340040772)

[Figura 1.4 - Comparação entre algoritmos de ampliação. 16](#_Toc340040773)

[Figura 1.5 - Representação do *pixel* a ser expandido 17](#_Toc340040774)

[Figura 1.6 - Algoritmo EPX. 17](#_Toc340040775)

[Figura 1.7 - Imagem original a esquerda, imagem tratada pelo 2xSaI a direita. 18](#_Toc340040776)

[Figura 1.8 - Implementação do Scale2x. 19](#_Toc340040777)

[Figura 1.9 - Resultados do Scale2x. 20](#_Toc340040778)

[Figura 1.10 - Algoritmo para conversão do formato RGB para YUV. 21](#_Toc340040779)

[Figura 1.11 - Condições que determinam se as cores são semelhantes ao olho humano. 21](#_Toc340040780)

[Figura 1.12 - Resultados do Hqx. 22](#_Toc340040781)

[Figura 1.13 - Resultado do Potrace. 23](#_Toc340040782)

[Figura 1.14 - Resultado do Potrace ao ser aplicado a uma imagem colorida. 23](#_Toc340040783)

[Figura 1.15 - Sistema Ardeco. 24](#_Toc340040784)

[Figura 1.16 - Aplicação do algoritmo de Orzan et al (2008). 24](#_Toc340040785)

[Figura 1.17 - Aplicação do *Adobe Live Trace* a um *pixel art.* 25](#_Toc340040786)

[Figura 1.18 - Aplicação do Vector Magic a um *pixel art*. 25](#_Toc340040787)

[Figura 2.1 - A esquerda, representação binária, a direita, representação visual da imagem. 27](#_Toc340040788)

[Figura 2.2 - Imagem vetorial. 30](#_Toc340040789)

[Figura 2.3 - Código fonte SVG da figura 2.2. 30](#_Toc340040790)

[Figura 2.4 - Imagem vetorial após aplicação de transformações. 31](#_Toc340040791)

[Figura 2.5 - Código fonte SVG da figura 2.4. 31](#_Toc340040792)

[Figura 2.6 - Imagem utilizando o elemento path. 32](#_Toc340040793)

[Figura 2.7 - Código fonte da figura 2.6. 32](#_Toc340040794)

[Figura 2.8 - Exemplo de curva quadrática de Bézier. 33](#_Toc340040795)

[Figura 2.9 - Exemplo de B-Spline quadrática. 34](#_Toc340040796)

[Figura 2.10 - Código fonte da figura 2.9. 34](#_Toc340040797)

[Figura 2.11 - Comparativo entre formatos de imagem. 35](#_Toc340040798)

[Figura 2.12 - Imagens após ampliação de quatro vezes. 36](#_Toc340040799)

[Figura 3.1 - Visão geral do processo de vetorização de *pixel arts*. 38](#_Toc340040800)

[Figura 3.2 - Representação do grafo de similaridade, em seu primeiro estágio, sobreposto ao *pixel art* original. 39](#_Toc340040801)

[Figura 3.3 – Exemplo da heurística da curva. Neste caso os dois pontos de conflito serão resolvidos por esta heurística, que manterá todos os nodos brancos conectados. 40](#_Toc340040802)

[Figura 3.4 - Exemplo da heurística dos *pixels* sobrepostos. Neste caso as conexões em roxo serão mantidas, por existir uma quantidade menor de *pixels* nesta cor. 40](#_Toc340040803)

[Figura 3.5 - Exemplo da heurística de ilhas. 41](#_Toc340040804)

[Figura 3.6 - Modelos de células remodeladas, antes e depois do processo. 42](#_Toc340040805)

[Figura 3.7 - Células remodeladas de acordo com o grafo de similaridade. 43](#_Toc340040806)

[Figura 4.1 - Esquema de desenvolvimento. 47](#_Toc340040807)

[Figura 4.2 - Grafo com ambiguidades resolvidas 49](#_Toc340040808)

[Figura 4.3 - Algoritmo para remodelagem dos *pixels* 50](#_Toc340040809)

[Figura 4.4 - Resultados após remodelagem dos *pixels.* 51](#_Toc340040810)

[Figura 4.5 - Mapeamento das curvas da imagem e suas terminações. 52](#_Toc340040811)

[Figura 4.6 - Representação da mão do personagem e seus respectivos pontos 54](#_Toc340040812)

[Figura 4.7 – Matriz de conversão da curva de Catmull-Rom para Bézier. 56](#_Toc340040813)

[Figura 4.8 - Antes e após a aplicação da curva de Catmull-Rom 57](#_Toc340040814)

[Figura 4.9 - Resultado final do algoritmo. 58](#_Toc340040815)

[Figura 4.10 - Comparação entre resultados de diversos algoritmos. 59](#_Toc340040816)

[Figura 4.11 – Comparação do algoritmo desenvolvido com Kopf e Linchinski 60](#_Toc340040817)

Lista de QUADROS

[Quadro 2.1 - Estrutura de arquivos Bitmap. 28](#_Toc340040818)

[Quadro 4.1 – Estrutura de dados de uma curva da imagem. 53](#_Toc340040819)

[Quadro 4.2 – Estrutura de dados de um objeto da imagem. 54](#_Toc340040820)

[Quadro 4.3 - Análise de desempenho do algoritmo. 61](#_Toc340040821)

Lista de Abreviaturas e Siglas

|  |  |
| --- | --- |
| CAD | *Computer-Aided Design* (Desenho auxiliado por computador) |
| CDR | Formato de arquivo do Corel Draw |
| CPU | *Central Processing Unit* (Unidade central de processamento) |
| EPX | *Eric's Pixel Expansion* (Expansão do pixel de Eric) |
| HQX | *High-quality magnification filter* (Filtro de ampliação de alta qualidade) |
| MAME | *Multiple Arcade Machine Emulator* (Emulador de multiplas máquina de arcade) |
| RGB | *Red*(vermelho), *green*(verde) e *blue*(azul) |
| SVG | *Scalable Vector Graphics* (gráficos vetoriais escaláveis) |
| WC3 | *World Wide Web Consortium* |
| XML | *eXtensible Markup Language* (Linguagem de marcação extensível) |
| YUV | Formato de cores formado por um elemento de brilho Y e dois de crominância U e V |
| 2xSaI | *The advanced 2x Scale and Interpolation engine* ( O Avançado motor de interpolação e escala 2x) |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Sumário

[Introdução 11](#_Toc340042166)

[1 O Pixel art 13](#_Toc340042167)

[1.1 Algoritmos de escalabilidade 15](#_Toc340042168)

[1.1.1 Eric's Pixel Expansion (EPX) 16](#_Toc340042169)

[1.1.2 2xSaI (*The advanced 2x Scale and Interpolation engine*) 18](#_Toc340042170)

[1.1.3 Scale2x 19](#_Toc340042171)

[1.1.4 Hqx (*High-quality magnification filter*) 20](#_Toc340042172)

[1.2 Algoritmos de vetorização 22](#_Toc340042173)

[2 Imagens vetorias e bitmaps 27](#_Toc340042174)

[2.1 Mapas de bits (*bitmaps*) 27](#_Toc340042175)

[2.2 Imagens vetoriais 28](#_Toc340042176)

[2.3 Comparativo entre imagens vetoriais e mapas de bits 35](#_Toc340042177)

[3 ALgoritmo para vetorização de *pixel arts* 37](#_Toc340042178)

[3.1 Grafo de similaridade 38](#_Toc340042179)

[3.1.1 Heurística da curva 39](#_Toc340042180)

[3.1.2 Heurística dos *pixels* sobrepostos 40](#_Toc340042181)

[3.1.3 Heurística das ilhas 41](#_Toc340042182)

[3.2 Remodelagem das células de *pixel* 41](#_Toc340042183)

[3.3 Aplicação de curvas 43](#_Toc340042184)

[4 Método APLICADO 45](#_Toc340042185)

[4.1 Aplicação 45](#_Toc340042186)

[4.2 Tecnologia 45](#_Toc340042187)

[4.3 Desenvolvimento 46](#_Toc340042188)

[4.3.1 Esquema de desenvolvimento proposto 46](#_Toc340042189)

[4.4 Algoritmos 47](#_Toc340042190)

[4.4.1 Mapeando a imagem em um grafo 48](#_Toc340042191)

[4.4.2 Resolução das ambiguidades 48](#_Toc340042192)

[4.4.3 Remodelagem do *pixel* 49](#_Toc340042193)

[4.4.4 Definição das curvas da nova imagem 51](#_Toc340042194)

[4.4.5 Definição dos objetos da imagem 53](#_Toc340042195)

[4.4.6 Suavização e geração da imagem final 55](#_Toc340042196)

[4.5 Análise dos resultados 58](#_Toc340042197)

[CONCLUSÃO 63](#_Toc340042198)

[Referências Bibliográficas 65](#_Toc340042199)

Introdução

O *pixel art* é uma forma de representar digitalmente uma figura com um pequeno número de *pixels*. Essa era muito difundida em computadores e vídeo games, nas décadas de oitenta e noventa, além de celulares e dispositivos que apresentam restrições de hardware com relação à resolução de tela.

Devido a estas restrições não era possível simplesmente diminuir a escala das imagens com alta definição, tendo em vista que apenas alguns *pixels* e cores estavam disponíveis para representar a figura, obrigando os gráficos desta época a serem desenhados manualmente, *pixel* por *pixel*. Contudo, a evolução dos computadores e a adoção de altas resoluções acabaram com a necessidade da utilização deste tipo de imagem.

A apresentação destes *pixel arts* por computadores e vídeo games modernos de altas resoluções se tornou pouco viável devido à baixa qualidade destas imagens quando apresentadas em altas resoluções. De encontro com esta necessidade, surgiram diversos algoritmos especializados em aumentar a escala de *pixel arts*.

Em sua maioria estes algoritmos foram desenvolvidos pelas comunidades de emuladores de vídeo games. Estas soluções funcionaram muito bem quando se tratavam de ampliação de menor escala, o dobro, o triplo e até o quádruplo da resolução original do *pixel art*, no entanto, para ampliações para os atuais padrões de resolução, estas soluções não conseguem satisfazer.

Uma abordagem mais definitiva para este problema da escalabilidade e tendo em vista atender as atuais resoluções e até futuras, leva em consideração a conversão deste tipo de imagem, que é baseada em um mapa de bits (*bitmap*), para um formato de representação independente de resolução, como o vetorial.

Em geral as ferramentas de vetorização de imagens existentes utilizam segmentação ou detecção de bordas para agrupar os *pixels* em grandes regiões, e por isto, não tendem a funcionar corretamente com este tipo específico de imagem.

Sendo assim, este trabalho se propõe a realizar uma pesquisa aplicada sobre as soluções existentes para este problema de ampliação de *pixel arts*. Apresentara um estudo das abordagens que consistem em aumentar o número de *pixels*, além de verificar os resultados obtidos pelas técnicas de vetorização de imagens existentes. E, pela necessidade de compreender o funcionamento dos formatos de armazenamento de imagens, uma pesquisa sobre estes formatos de arquivos de *bitmaps* e vetorial será realizada.

Como a melhor solução encontrada para o problema tratado foi a proposta de vetorização de *pixel arts* de Johannes Kopf e Dani Lischinski (2011), esta será o principal foco dos estudos deste trabalho, sendo analisada detalhadamente.

Por fim, neste projeto será realizado o desenvolvimento de um método para a solução do problema através de vetorização do *pixel art*, além de implementar algumas das técnicas existentes para o aumento da escala de *pixel arts*.

O presente trabalho será abordado de forma qualitativa, pois a avaliação dos resultados obtidos será realizada com base na percepção visual, obtida das imagens após a aplicação das técnicas pesquisadas e do método a ser desenvolvido. Os objetivos do mesmo, o caracterizam como pesquisa exploratória, pois visa proporcionar maior familiaridade com o problema, o tornando explícito (PRODANOV, 2009).

Baseando-se na pesquisa bibliográfica, composta em sua grande maioria de artigos científicos e projetos *open source*, devido à natureza do tema proposto, o trabalho pretende, através de experimentação das técnicas pesquisadas, aplicá-las e efetuar uma análise de seus resultados, além de propor e desenvolver uma abordagem própria ao problema.

Desta maneira, tendo em vista as grandes possibilidades na reutilização destas imagens, a utilização de algoritmos que tornem possível redimensionar estes *pixel arts* para altas resoluções, sem a perda de suas características artísticas originais, possui uma importância relevante. Sobretudo quando levamos em consideração o potencial econômico para as grandes empresas de jogos eletrônicos, com a possibilidade de relançar ao mercado jogos atualmente considerados clássicos.

Este trabalho irá, no primeiro capítulo, apresentar o pixel e exibir o histórico dos *pixel arts*. Em seguida, introduzirá o problema do aumento da escala dos *pixel arts* e mostrará os principais algoritmos de aumento da quantidade de *pixels* de uma imagem. São eles: *EPX, 2xSaI, Scale2x e HQX*. Posteriormente, serão apresentadas e analisadas algumas técnicas de vetorização e seus resultados quando aplicadas a *pixel arts.*

No segundo capítulo uma análise sobre os tipos de imagens será feita, exibindo os formatos *bimap* e vetorial e apresentando um comparativo entre os mesmos. No terceiro capítulo, uma análise profunda da abordagem realizada por Kopf e Lischinski (2011) será realizada. Pois se trata da melhor solução encontrada para o problema.

E por fim, no quarto capítulo, o método proposto para solução do problema será apresentado, mostrando as técnicas e algoritmos da abordagem desenvolvida e analisado seus resultados em comparação com as outras abordagens existentes.

# O Pixel art

*Pixel art* tem suaorigema partir da palavra, também da língua inglesa, *pixel,* que por sua vez, é uma popularização de *picture element*. O termo foi utilizado pela primeira vez na revista Wireless World, em 1927, na descrição sobre o funcionamento de aparelhos de televisão: “A imagem apresentada no aparelho é definida como um mosaico de pontos ou elementos de imagem, centenas de pequenos quadrados com variações na intensidade do brilho” (LYON, 2006). A tecnologia evoluiu, existem cores e resoluções superiores agora, contudo, o princípio de *pixel* continua o mesmo.

O termo *pixel art*, por sua vez, foi introduzido só em 1982, por Adele Goldberg e Robert Flegal da Xerox (GOLDBERG, 1982). Apesar de seus conceitos já serem utilizados desde a década de setenta, que consistiam em uma técnica para representar digitalmente uma figura com um pequeno número de *pixels*, essa foi amplamente difundida em computadores e vídeo games, nas décadas de oitenta e noventa, além de celulares e dispositivos que apresentam restrições de hardware com relação à resolução de tela.

Devido a estas restrições não era possível simplesmente diminuir a escala das imagens com alta definição, tendo em vista que apenas alguns *pixels* estavam disponíveis para representar a figura. Por exemplo, um olho de um personagem quase sempre era representado por um único *pixel* preto ou branco. Existia a necessidade dos artistas trabalharem com um número extremamente limitado de pontos, e uma paleta de cores restrita, obrigando os designers a organizarem manualmente os *pixels* na imagem, fazendo com que cada um detenha relevância para a imagem, não podendo ser ignorado.

Por essa razão, os *pixel arts* clássicos normalmente são marcados pela economia de recursos, minimalismo e uma certa modéstia que muitos dizem estar perdida nos gráficos computadorizados modernos. Os melhores *pixel arts* da chamada era de ouro dos vídeo games (1978 a 1994) são considerados obras-primas, como o clássico *Super Mario Bros* de 1985 (figura 1), formado de apenas 3 cores e resolução de 16x12 (Esposito, 2005).

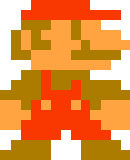


Figura .1 - *Super Mario Bros*.

Fonte: ©Nintendo, 1985.

Segundo o mesmo autor, estas obras trazem um sentimento de nostalgia e continuam a ser apreciadas até hoje, em versões relançadas para aparelhos modernos ou graças aos inúmeros emuladores desenvolvidos para substituir estes hardwares que já se encontram extintos.

A evolução da capacidade de processamento das máquinas e a adoção de altas resoluções fizeram com que a arte na escala do *pixel* perdesse sua necessidade. No entanto, ela se tornou um símbolo cultural para toda uma geração e atualmente suas características voltaram a ser exploradas por artistas, não em consequência das restrições de hardware, mas devido ao poder de representatividade e os sentimentos ligados a este tipo de imagens (Cottee, 2010).

Mais alguns exemplos de *pixel arts* são apresentados nas figuras a seguir.

|  |  |
| --- | --- |
| C:\Documents and Settings\mbecker\Meus documentos\Dropbox\TCC\Imagens\pacman_ghost_big.png | C:\Documents and Settings\mbecker\Meus documentos\Dropbox\TCC\Imagens\Zelda_big.png |

Figura .2 - *Pacman* e *Zelda*.

Fonte: ©Nanco, 1980 e ©Nintendo, 1991.

|  |  |
| --- | --- |
| C:\Documents and Settings\mbecker\Meus documentos\Dropbox\TCC\Imagens\megaman_big.png | C:\Documents and Settings\mbecker\Meus documentos\Dropbox\TCC\Imagens\finalfantasy3_big.png |

Figura .3 – *Megaman* e *Final Fantasy*.

Fonte: ©Capcom, 1988 e ©Square, 1987.

## Algoritmos de escalabilidade

Como a utilização de *pixel arts* em novos computadores e aparelhos de televisão de alta resolução se tornou pouco viável, devido à baixa qualidade das imagens quando apresentadas em altas resoluções, foram necessárias soluções para melhorar esta qualidade.

A abordagem clássica quando se trata de ampliar uma imagem é a utilização de filtros como o *nearest-neighbor* e *bicubic* (figura 1.4). No entanto, essas abordagens não foram projetadas para *pixel arts*, apresentando um aspecto borrado, no caso do *bicubic,* e mantendo o aspecto quadriculado, utilizando-se o *nearest-neighbor* (WOLBERG, 1996).

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **Original** | ***Nearest-neighbor*** | ***Bicubic*** |

Figura .4 - Comparação entre algoritmos de ampliação.

Fonte: Imagem processada de *Aladim,* ©Capcom, 1993.

Para a solução desta questão, em *pixel arts,* diversos algoritmos foram propostos. Em sua maioria foram criados pela comunidade desenvolvedora de emuladores de consoles e não foram publicados em artigos científicos. No entanto, como se tratam de soluções distribuídas em código aberto, é possível analisá-las. A localização destes projetos foi possível através de pesquisa nas páginas pessoais dos desenvolvedores, onde o código fonte e uma breve descrição dos projetos estavam disponíveis.

### Eric's Pixel Expansion (EPX)

O primeiro algoritmo desenvolvido visando resolver este problema foi apresentado por Eric Johnston, da LucasArts, por volta de 1992, para portar jogos da LucasArts (que executavam no IBM PC com resoluções de 320×200×256 cores)  para os novos computadores da Macintosh, que na época apresentavam o dobro da resolução (KAS, 1999).

Era necessário dobrar a resolução, ou seja, onde existia um *pixel*, agora existirão quatro. Para solucionar esta questão, o autor utilizou-se de uma abordagem simples, analisando apenas os quatro *pixels* vizinhos do *pixel* a ser expandido.

Caso o *pixel* vizinho superior (A) e o direito (B) forem de mesma cor, o novo *pixel* superior da direita (P1), na nova imagem, possuirá a mesma cor dos dois, em caso contrário, possuirá a cor do *pixel* que está sendo expandido na imagem de origem. Desta maneira o algoritmo consegue diminuir o efeito serrilhado quando são encontradas sequencias diagonais de *pixels* da mesma cor na imagem. Esta mesma lógica é aplicada a todos os novos *pixels*. O conceito pode ser mais bem compreendido ao analisar a figura e o algoritmo apresentados na sequencia.

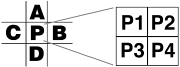


Figura .5 - Representação do *pixel* a ser expandido

Fonte: KAS, 1999.

|  |
| --- |
| SE C==A ENTAO P1=A  SE A==B ENTAO P2=B  SE B==D ENTAO P4=D  SE D==C ENTAO P3=C  SE em A, B, C, D, 3 ou mais são idênticos então P1=P,P2=P,P3=P,P4=P |

Figura .6 - Algoritmo EPX.

Fonte: Desenvolvido pelo autor.

### 2xSaI (*The advanced 2x Scale and Interpolation engine*)

Apresentado em 2001, por Derek Liauw Kie Fa, o *2xSaI* foi uma solução desenvolvida para resolver o efeito da ampliação de *pixel arts* nos emuladores de Super Nintendo, porém, com uma abordagem diferente dos demais (KIE FA, 2001)

Em sua abordagem o autor se propõe a duplicar a resolução da imagem original, entretanto, toma a liberdade de não utilizar apenas a paleta original de cores. Em seu processo transforma cada *pixel* em quatro, mantendo um deles na cor original e definindo novas cores para os demais, baseado em padrões detectados na imagem, como linhas e bordas (Kie Fa, 2001).

Desta maneira, o algoritmo define as novas cores através de uma mistura das cores visinhas. Na tentativa de recriar o efeito produzido por televisores de baixas resoluções, semelhantes aos existentes na década de oitenta, nos quais quando *pixels* de cores diferentes se encontram próximos, existe um efeito natural de suavização que não torna as imagens tão quadriculadas.

|  |  |
| --- | --- |
|  |  |

Figura .7 - Imagem original a esquerda, imagem tratada pelo 2xSaI a direita.

Fonte: Imagem processada de *Sunset Riders,* ©Konami, 1993.

### Scale2x

Foi desenvolvido pelo italiano Andrea Mazzoleni, também em 2001, porém para o projeto AdvanceMAME (MAME, 2006), cuja proposta era portar jogos de fliperama para o computador. Sua abordagem se destaca por ser a primeira a apresentar também a possibilidade de triplicar e quadruplicar o *pixel art*. Utilizando apenas as cores originais da imagem, este algoritmo se aproxima da implementação do EPX, porém é mais polido, rápido e obtém resultados superiores, visto que suporta ampliações maiores (figura 1.4) c

Este algoritmo possui resultados superiores, pois considera escalas maiores, e possui uma lógica específica no tratamento de ampliações de 3x. Além disso, sua implementação é otimizada, em relação ao EPX. Seu código, para ampliações de 2x, pode ser verificado no código em C a seguir, utilizando da mesma representação dos *pixels* da figura 1.5.

|  |
| --- |
| if (A != D && C != B) {  P1 = C == A ? C : P;  P2 = A == B ? B : P;  P3 = C == D ? C : P;  P4 = D == B ? B : P;  } else {  P1 = P;  P2 = P;  P3 = P;  P4 = P;  } |

Figura .8 - Implementação do Scale2x.

Fonte: Adaptado de Mazzoleni, 2001.

Segundo o código fonte da figura 1.8, se as cores dos *pixels* vizinhos superior (A) e inferior (B) do *pixel* analisado forem iguais, e se as cores dos vizinhos direito (C) e esquerdo (B) também forem iguais, o algoritmo aplicará uma lógica para definir as cores dos novos *pixels*, caso contrário, todos os novos *pixels* (P1, P2, P3 e P4) possuirão a cor do *pixel* que esta atualmente sendo analisado (P). Para a definição da cor dos novos *pixels* o algoritmo verifica se existe uma linha diagonal sendo formada entre os *pixels* correspondentes ao novo *pixel*. Por exemplo, se C e A possuírem cores iguais, a cor de P1 será igual à deles, caso as cores sejam diferentes, sua cor será igual ao do pixel original.

A imagem a seguir apresenta os resultados do Scale2x em ampliações de 2x e 4x.

|  |  |  |
| --- | --- | --- |
| http://scale2x.sourceforge.net/snap/mslug2-1s.png | http://scale2x.sourceforge.net/snap/mslug2-2s.png | http://scale2x.sourceforge.net/snap/mslug2-3.png |
| Original | Scale2x | Scale4x |

Figura .9 - Resultados do Scale2x.

Fonte: Imagem processada de *Metal Slug,* ©SNK, 1996.

### Hqx (*High-quality magnification filter*)

Em 2002, Maxim Stepin apresentou uma abordagem diferente em seu filtro para ampliação de *pixel arts*. Com uma implementação que possibilita a ampliação para o dobro, triplo ou quádruplo da resolução, ele não cria novas cores, e consegue os melhores resultados entre todas as técnicas (Stepin, 2002).

Seu algoritmo começa com uma análise da área de 3x3 ao redor do *pixel* de origem, classificando os oito vizinhos do *pixel* central como próximos ou distantes da cor do ponto a ser expandido. Essa classificação é feita a partir de uma análise das cores no formato YUV, por se aproximar mais da percepção humana das cores que o formato RGB. Esta conversão é realizada conforme código a seguir.

|  |
| --- |
| Y = (0.257 \* R) + (0.504 \* G) + (0.098 \* B) + 16  Cr = V = (0.439 \* R) - (0.368 \* G) - (0.071 \* B) + 128  Cb = U = -(0.148 \* R) - (0.291 \* G) + (0.439 \* B) + 128 |

Figura .10 - Algoritmo para conversão do formato RGB para YUV.

Fonte: JACK, 2001.

Sendo assim, as cores são convertidas para o padrão YUV (conforme apresentado na figura 1.10), onde são comparados os três canais de cores separadamente. Nesta comparação os canais do *pixel* atual são representados por Y, U e V, enquanto os do *pixel* vizinho são representados por Y’, U’ e V’. As cores serão consideradas semelhantes se as três condições a seguir forem verdadeiras:

|  |
| --- |
| Y – Y’ <= 48  U – U’ <= 7  V – V’ <= 6 |

Figura .11 - Condições que determinam se as cores são semelhantes ao olho humano.

Fonte: STEPIN, 2002.

Ou seja, segundo Stepin (2002), se valor do brilho (Y) do *pixel* analisado, menos o brilho (Y’) do *pixel* vizinho for menor ou igual a 48, e se o componente de crominância U, do *pixel* analisado, menos o componente U’, do *pixel* vizinho, for menor ou igual a 7, e se o componente de crominância, do *pixel* analisado, V menos V’, do *pixel* vizinho, for menor ou igual a 6; então estas duas cores são consideradas semelhantes, caso uma das condições não for verdadeira, as cores serão consideradas distintas.

Isso significa que duas cores podem variar mais em seu brilho (Y) que nos outros componentes da cor, e mesmo assim serem consideradas similares pelo algoritmo.

Segundo o mesmo autor, ao finalizar a classificação, uma tabela de pesquisa, já pré-definida, contendo as possíveis 256 combinações é consultada, o que retorna o padrão a ser aplicado de uma forma muito rápida, permitindo um resultado excelente, com um tempo de processamento extremamente baixo. O resultado da aplicação deste algoritmo pode ser verificado na figura a 1.12.

|  |  |
| --- | --- |
| http://www.hiend3d.com/img/randam_nn3x.png Original | http://www.hiend3d.com/img/randam_hq3x.png Hq3x |

Figura .12 - Resultados do Hqx.

Fonte: Imagem processada de ©SNK, 1996.

## Algoritmos de vetorização

Entre os algoritmos de vetorização existem algumas soluções, no entanto, quando aplicadas a *pixel arts,* estas soluções tendem a obter resultados insatisfatórios, pois elas não visam à vetorização deste tipo específico de imagem.

Uma das abordagens que realiza esta conversão de *pixel art* para vetores com sucesso, foi realizada por Peter Selinger em seu programa *Potrace*, que realiza a vetorização de imagem em baixas resoluções. No entanto, esse atende apenas a imagens binárias, ou seja, nas cores preto e branco (figura 1.13). Sua adaptação para imagens coloridas, criando-se canais separados para cada cor, acaba criando conflitos entre as diferentes camadas (figura 1.14) (SELINGER, 2003).

|  |  |
| --- | --- |
| [http://potrace.sourceforge.net/img/head-orig3.png](http://potrace.sourceforge.net/img/head.pbm) | [http://potrace.sourceforge.net/img/head-smooth3.png](http://potrace.sourceforge.net/img/head.eps) |
| Original | Potrace |

Figura .13 - Resultado do Potrace.

Fonte: SELINGER, 2003.

|  |  |
| --- | --- |
| http://research.microsoft.com/en-us/um/people/kopf/pixelart/supplementary/results_nearest/sbm1_04_nearest_16x.png | http://research.microsoft.com/en-us/um/people/kopf/pixelart/supplementary/results_potrace/sbm1_04_potrace_16x.png |
| Original | Potrace |

Figura .14 - Resultado do Potrace ao ser aplicado a uma imagem colorida.

Fonte: KOPF; LISCHINSKI, 2011.

Os autores Lecot e Lévy (2006) apresentaram o sistema Ardeco para vetorização de imagens matriciais. Essa abordagem trabalha segmentando o bitmap e posteriormente aplicando gradientes de cor. Entretanto, apesar de obter bons resultados em imagens de maiores resoluções, ele não consegue resultados aceitáveis em *pixel arts*, devido a natureza minimalista deste tipo de imagem (figura 1.14). De maneira semelhante, a abordagem realizada em Lai et al.(2009) também baseia-se em segmentação e, sendo assim, igualmente não consegue bons resultado.

|  |  |
| --- | --- |
| http://research.microsoft.com/en-us/um/people/kopf/pixelart/supplementary/results_nearest/sbm1_04_nearest_16x.png | http://research.microsoft.com/en-us/um/people/kopf/pixelart/supplementary/results_ardeco/sbm1_04_ardeco_deg1.png |
| Original | ARTDECO |

Figura .15 - Sistema Ardeco.

Fonte: KOPF; LISCHINSKI, 2011.

Os outros algoritmos para vetorização encontrados, como o de Orzan et al. (2008) e Xia et al. (2009) são ambos baseados no método de detecção de Canny (1986) e como este método não consegue ser aplicado a *pixel arts* com sucesso, os seus resultados também não são satisfatórios.

|  |  |
| --- | --- |
| http://research.microsoft.com/en-us/um/people/kopf/pixelart/supplementary/results_nearest/smw_yoshi_nearest_16x.png |  |
| Original | Orzan et al. |

Figura .16 - Aplicação do algoritmo de Orzan et al (2008).

Fonte: KOPF; LISCHINSKI, 2011.

Nas abordagens comerciais como *Adobe Live Trace* (ADOBE, 2010) e *Vector Magic* (2010)*,* das quais não se conhece o algoritmo, é possível observar que também falham quando aplicados a *pixel arts* (figura 1.17 e 1.18).

|  |  |
| --- | --- |
| http://research.microsoft.com/en-us/um/people/kopf/pixelart/supplementary/results_nearest/smw2_yoshi_02_nearest_16x.png | http://research.microsoft.com/en-us/um/people/kopf/pixelart/supplementary/results_livetrace/smw2_yoshi_02_livetrace_16x.png |
| Original | *Adobe Live Trace* |

Figura .17 - Aplicação do *Adobe Live Trace* a um *pixel art.*

Fonte: KOPF; LISCHINSKI, 2011.

|  |  |
| --- | --- |
| http://research.microsoft.com/en-us/um/people/kopf/pixelart/supplementary/results_nearest/smb_jump_nearest_16x.png | http://research.microsoft.com/en-us/um/people/kopf/pixelart/supplementary/results_vectormagic/smb_jump_vectormagic_16x.png |
| Original | *Vector Magic* |

Figura .18 - Aplicação do Vector Magic a um *pixel art*.

Fonte: KOPF; LISCHINSKI, 2011.

Algoritmos voltados para vetorização específica de *pixel arts*, o foco principal dos estudos deste trabalho, visam à solução definitiva ao problema de escala destas imagens, tendo em vista as propriedades de imagens vetorizadas. No entanto, estes algoritmos ainda são considerados novidade e foi localizada uma única publicação, realizada em 2011, por Kopf e Lischinski (Kopf; Lischinski, 2011). Sua funcionalidade será discutida detalhadamente no capítulo três. Todavia, para compreender seu funcionamento, é importante uma análise dos formatos de arquivo envolvidos, que será realizada no próximo capítulo.

# Imagens vetorias e bitmaps

Este capítulo tem por objetivo analisar a forma como as imagens são armazenadas, realisando uma apresentação do formato de mapas de bits (*bitmap*), no qual os *pixel arts* são contidos. Além de apresentar o formato vetorial (gerado no algoritmo proposto), serão demonstradas as principais diferenças existentes entre estas duas formas de registrar uma imagem.

## Mapas de bits (*bitmaps*)

O formato de armazenamento *bitmap,* também conhecido como *raster*, ou aindamatricial, consiste em registrar a imagem em um arranjo espacial bidimensional, ou seja, uma matriz retangular. Cada elemento representa um *pixel* e cada *pixel*, por sua vez, é definido pela sua cor, normalmente nos valores de 0 a 255, de vermelho, verde e azul (RGB). A figura 2.1 explica melhor a descrição de uma imagem matricial binária (imagem formada apenas por *pixels* de duas cores) (Azevedo; Conci, 2003).

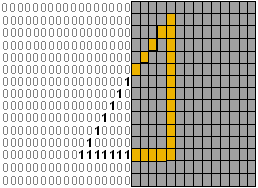


Figura .1 - A esquerda, representação binária, a direita, representação visual da imagem.

Fonte: COMPUTER DESKTOP ENCYCLOPEDIA, 2005.

Scanners, câmeras, e outros dispositivos de digitalização realizam a captura de imagens sempre através do formato *raster.* Do mesmo modo, dispositivos de saída como televisores, monitores, impressoras, também utilizam um número fixo de *pixels*, e por isto armazená-los nesta estrutura é a maneira natural de guardar os dados de uma imagem que pretende ser apresentada em tais dispositivos (Murray; vanryper, 1996).

Imagens formadas desta maneira são dependentes da resolução e não é possível escalar as mesmas para resoluções maiores sem perda aparente de sua qualidade. Para isto, normalmente algoritmos de escalabilidade como *bicubic* ou *nearest-neighbor* são aplicados. Este formato de armazenamento também é caracterizado por ser o formato de armazenamento utilizado em fotografias digitais.

Nesta mesma forma de armazenamento existem diversos formatos que se baseiam no formato de mapa de bits para armazenar imagens, muitos dos quais compactam os dados, com e sem perdas para a imagem. Entre eles é possível destacar JPG, GIF, PNG, BMP, etc. O formato bitmap normalmente possui a estrutura do armazenamento definida por um cabeçalho, seguido de uma paleta de cores e, por fim, os dados da imagem (tabela 2.1) (Murray; vanryper, 1996).

Quadro .1 - Estrutura de arquivos Bitmap.

|  |  |  |
| --- | --- | --- |
| Estrutura | Tamanho | Função |
| BITMAPFILEHEADER | 14 Bytes | Armazenar informações gerais sobre o tipo de arquivo bitmap |
| BITMAPINFOHEADER | Fixo (existem versões com diferentes tamanhos) | Informações sobre a imagem, e definição do formato do *pixel*. |
| RGBQUAD array | Variável | Define o valor dos *pixel*s. Cada linha da matriz é uma linha da imagem. |
| Color-index array | Variável | Paleta de cores. Definição das cores. |

Fonte: Adaptado de MSDN, 2012.

## Imagens vetoriais

Arquivos de imagens vetoriais contém, ao contrário do formato de mapa de bits, uma representação baseada em descrições matemáticas de um ou mais elementos, que são usados pelo aplicativo de renderização para construir a imagem final. Em outras palavras, é possível dizer que a imagem vetorial é constituída de um conjunto de instruções para desenhar alguma imagem.

É interessante ressaltar que o início da computação gráfica foi baseado quase inteiramente em gráficos vetoriais, contudo, o advento de maiores capacidades de armazenamento e processamento permitiram que grandes arquivos armazenados em formato bitmap pudessem ser utilizados, fazendo com que, atualmente, a maior parte das imagens utilizadas seja do padrão matricial. (Murray; vanryper, 1996).

O formato vetorial é estruturalmente mais simples que o de matricial, e tende a ser organizado apenas pelo cabeçalho, seguido dos dados, os quais podem estar registrados binariamente como arquivos bitmap, ou na forma arquivo de texto (conforme a figura 2.3).

A utilização deste formato inclui uma grande diversidade de aplicações como *Computer Aided Design* (CAD), design gráfico para impressoras de altas resoluções, linguagens para descrição de imagem, impressão e até animações. Por esta razão existem diversos formatos de arquivos vetoriais disponíveis. Para aplicação em imagens é possível destacar os formatos comerciais CDR, formato proprietário da Corel e o AI, formato do Adobe Illustrator. Entre os padrões abertos destaca-se o SVG (*Scalable Vector Graphics*), formato criado pela W3C (*World Wide Web Consortium*) em 1999 (W3C, 2011).

O padrão SVG possui uma grande vantagem em relação aos demais, pois se trata de uma aplicação do formato XML (W3C, 2008). Como arquivos XML se tratam de simples arquivos texto, eles permitem uma fácil abertura, edição e transparência dos arquivos, tanto para o usuário como para softwares, muito diferente dos modelos binários adotados comercialmente. (EISENBERG, 2002). Um exemplo de arquivo XML pode ser visto na figura 2.3.

A maioria dos formatos vetoriais suporta as seguintes primitivas.

* *Line*
* *Circle*
* *Rect*
* *Ellipse*
* *Polyline*
* *Polygon*
* *Text*
* *Image*

Todas estas primitivas podem ser dispostas na figura, como que em um plano cartesiano, e utilizadas na criação da imagem. Além do próprio desenho que cada uma proporciona, todas permitem que um grupo de atributos (*Attributes*) e estilos (*Styles*) seja descrito para cada elemento que compõem a imagem. Sua construção pode ser mais bem compreendida ao analisar uma figura e seu código fonte.



Figura .2 - Imagem vetorial.

Fonte: Desenvolvido pelo autor.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <svg width="140" height="170" xmlns="http://www.w3.org/2000/svg" >  <!-- Cabeça -->  <circle cx="70" cy="95" r="50" style="stroke: black; fill: yellow;" />  <!-- Olhos -->  <circle cx="55" cy="80" r="5" style="stroke: black; fill: blue;"/>  <circle cx="85" cy="80" r="5" style="stroke: black; fill: blue;"/>  <!-- Boca -->  <line x1="35" y1="110" x2="45" y2="120" style="stroke: black;" />  <line x1="45" y1="120" x2="95" y2="120" style="stroke: black;" />  <line x1="95" y1="120" x2="105" y2="110" style="stroke: black;" />  </svg> |

Figura .3 - Código fonte SVG da figura 2.2.

Fonte: Desenvolvido pelo autor.

Como pode ser verificado, a figura 2.2 foi criada a partir de apenas duas primitivas: *circle* e *line*. Na definição do código fonte é possível observar que no caso do círculo o posicionamento é relativo ao centro, enquanto na linha, são informados os pontos das duas extremidades da mesma.

Também se pode notar a utilização dos atributos e a definição do estilo. Onde encontrado *r=“5”* tem-se a definição de um dos atributos da primitiva, neste caso, o raio do elemento círculo. O estilo, por sua vez, está definido sempre dentro do atributo *style*, no entanto ele pode estar definido na linha, como no exemplo, ou definido de forma separada e aplicado a um ou mais elementos.

O formato vetorial ainda permite agrupar diversos elementos primitivos e criar objetos que podem ser usados diversas vezes dentro da imagem. Estes modelos podem ser copiados para a localização desejada na imagem, e ainda podem ser escalados, movidos, rotacionados ou modificados por outras operações matemáticas (conforme imagens a seguir).

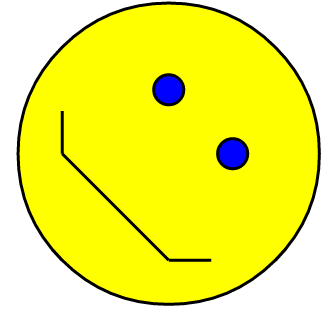


Figura .4 - Imagem vetorial após aplicação de transformações.

Fonte: Desenvolvido pelo autor.

|  |
| --- |
| <?xml version="1.0" standalone="no"?>  <!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">  <svg version="1.1" width="230" height="230">  **<g transform="translate(150,-130)">**  **<g transform="scale(2)">**  **<g transform="rotate(45)">**  <circle cx="70" cy="95" r="50" style="stroke:black; fill:yellow;" />  <!-- Olhos -->  <circle cx="55" cy="80" r="5" style="stroke: black; fill: blue;"/>  <circle cx="85" cy="80" r="5" style="stroke: black; fill: blue;"/>  <!-- Boca -->  <line x1="35" y1="110" x2="45" y2="120" style="stroke: black;" />  <line x1="45" y1="120" x2="95" y2="120" style="stroke: black;" />  <line x1="95" y1="120" x2="105" y2="110" style="stroke: black;" />  **</g>**  **</g>**  **</g>**  </svg> |

Figura .5 - Código fonte SVG da figura 2.4.

Fonte: Desenvolvido pelo autor.

Outra ferramenta disponibilizada pelo formato vetorial, mais especificadamente o SVG, é o elemento *path*. Trata-se de um elemento mais genérico, que permite a construção de linhas entre pontos, possibilitando a construção de diversas formas, com a utilização de retas, curvas ou elipses. O seu funcionamento pode ser mais bem compreendido ao observar a figura a seguir e seu código fonte.

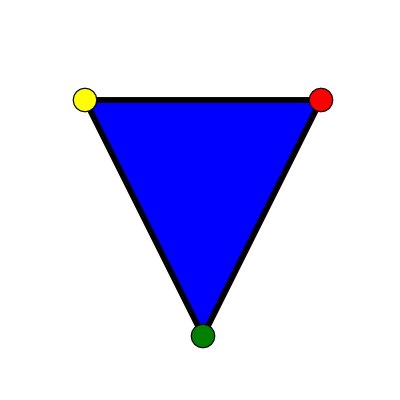


Figura .6 - Imagem utilizando o elemento path.

Fonte: Desenvolvido pelo autor.

|  |
| --- |
| <?xml version="1.0" standalone="no"?>  <!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">  <svg width="4cm" height="4cm" viewBox="0 0 400 400" xmlns="http://www.w3.org/2000/svg" version="1.1">  <path **d="M 100 100 L 300 100 L 200 300 z"** fill="blue" stroke="black" stroke-width="5" />  <circle cx="100" cy="100" r="10" fill="yellow" />  <circle cx="300" cy="100" r="10" fill="red" />  <circle cx="200" cy="300" r="10" fill="green" />  </svg> |

Figura .7 - Código fonte da figura 2.6.

Fonte: desenvolvido pelo autor.

Como se pode verificar, a definição do elemento (destacado em vermelho na figura 2.7) inicia pelo comando M, *move to,* responsável pela definição do ponto inicial do trajeto a ser desenhado, seguido das coordenadas do respectivo ponto, neste caso, o ponto amarelo (100,100). Depois, é utilizado o comando L, *line*, que define que o tipo a ser inserido é linha e terá seu ponto inicial na última coordenada recebida, e seu ponto final na coordenada informada após o comando L, no exemplo, o ponto vermelho (300,100). Em seguida, o mesmo ocorre para a próxima linha, que inicia no ponto vermelho (300,100) e termina no ponto verde (200,300). Para finalizar o triangulo, é utilizado o comando z, *closepath*, que simplifica o código e liga o ponto atual ao ponto inicial através de uma reta, finalizando o objeto, no exemplo ligando do ponto verde (200,300) ao amarelo (100,100).

Para a definição de curvas em imagens vetoriais é utilizada curva quadrática de Bézier. Para a descrição de uma curva destas são necessários os pontos de início (P1) e fim da curva (P2), e um terceiro ponto de controle (C12) do qual a curva irá apenas se aproximar. Pode-se imaginar que este terceiro ponto atua como um campo gravitacional para a linha original entre os pontos inicial e final, deformando-a elasticamente e formando uma curva suave (FROST ET AL., 2012). Sua definição formal pode ser localizada no livro de Boor (2001), *A practical guide to splines. Applied mathematical sciences.*

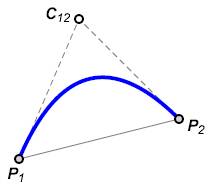


Figura .8 - Exemplo de curva quadrática de Bézier.

Fonte: FROST ET AL., 2012.

Em casos onde a curva possuiu mais pontos de controle, ela passa a ser chamada de curva cúbica de Bézier (SEDERBERG, 2003). Quando existe uma série de curvas de Bézier, unindo o fim de uma ao começo de outra, essa sequência é chamada de curva *Spline* de Bézier ou B-Spline. Em arquivos vetoriais SVG, estas curvas são descritas de acordo com o exemplo a seguir.

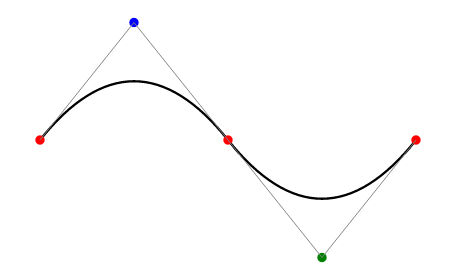


Figura .9 - Exemplo de B-Spline quadrática.

Fonte: Desenvolvido pelo autor.

|  |
| --- |
| <?xml version="1.0" standalone="no"?>  <!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">  <svg width="12cm" height="6cm" viewBox="0 0 120 60"  xmlns="http://www.w3.org/2000/svg" version="1.1">  <!-- Quadratica Spline Bezier -->  <path **d="M 20,30 Q 40,5 60,30 T 100,30"** fill="none" stroke="black" stroke-width="0.7"/>  <!-- Ponto Finais -->  <circle cx="20" cy="30" r="1" fill="red"/>  <circle cx="60" cy="30" r="1" fill="red"/>  <circle cx="100" cy="30" r="1" fill="red"/>  <!-- Ponto de Controle -->  <circle cx="40" cy="5" r="1" fill="blue"/>  <circle cx="80" cy="55" r="1" fill="green" />  <path d="M20,30 L40,5 L60,30 L80,55 L100,30" fill="none" stroke="grey" stroke-width="0.4" />  </svg> |

Figura .10 - Código fonte da figura 2.9.

Fonte: Desenvolvido pelo autor.

Como é possível observar na figura 2.10, a definição do elemento (destacado em vermelho) inicia com o comando M, que determina a posição inicial (20,30), seguido do comando Q, correspondente à quadrática de Bézier, que indica o ponto de controle da curva, o ponto azul, (40,5) seguido do ponto final desta curva (60,30). Para a segunda curva desta mesma linha foi utilizado o comando T, que exige apenas o ponto final da curva, e assume que o ponto de controle para a mesma é a reflexão do ponto de controle da curva anterior. Caso a última curva possuísse um ponto de controle diferente, bastaria se adicionar um novo comando Q, no lugar do T.

## Comparativo entre imagens vetoriais e mapas de bits

Nesta sessão será realizada uma comparação entre estes dois formatos de armazenamento de imagens, verificando seus pontos fortes e fracos. Pode-se iniciar esta análise comparando as imagens da figura a seguir, onde ambas são apresentadas na escala de 100x100 *pixels*.

|  |  |
| --- | --- |
|  | C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\rosto_capitulo2.png |
| **Vetorial** | **Matricial** |

Figura .11 - Comparativo entre formatos de imagem.

Fonte: Desenvolvido pelo autor.

As duas são muito semelhantes visualmente. Contudo, a representação vetorial é desenhada na tela a partir das especificações matemáticas do arquivo, enquanto a matricial define todos os 10000 *pixels* que compõem esta imagem. Em razão disto, a diferença visual entre essas imagens só ficará evidente quando forem ampliadas. A imagem formada pelo mapa de bits precisa criar novos *pixels*, ou seja, se a imagem for ampliada quatro vezes, cada pixel será substituído por quatro novos da mesma cor. No caso da representação vetorial, será necessário multiplicar todas as coordenadas por quatro e desenhar as primitivas nesta nova resolução (figura 2.5).

|  |  |
| --- | --- |
| C:\Users\Usuario\Desktop\Sem título.png |  |
| **Vetorial** | **Matricial** |

Figura .12 - Imagens após ampliação de quatro vezes.

Fonte: Desenvolvido pelo autor.

Como se pode verificar, a imagem vetorial permanece com contornos suaves após a ampliação e tem nesse seu principal ponto positivo. Entretanto, o registro de imagens naturais, como fotografias, só é possível através de imagens matriciais, visto que não são encontradas formas matemáticas perfeitas na natureza.

Em geral, a facilidade de uso dos arquivos de mapa de bits é superior, pois a quantidade de editores e visualizadores disponível é muito maior que os disponíveis para imagens vetoriais. Além disso, imagens vetoriais passaram a possuir padrões abertos, como o SVG, há pouco tempo, o que torna o tratamento de imagens com estes formatos mais restrito.

Outro aspecto a ser ressaltado é com relação ao tamanho dos arquivos. Enquanto arquivos vetoriais são formados por um documento com as descrições da figura, e pode ser utilizado independente da resolução, arquivos matriciais possuem o mapeamento de todos os *pixels* da imagem, e por isso, quanto maior a resolução, maior será o tamanho do arquivo. Em consequência disto, os arquivos vetoriais tendem a ser muito menores em relação ao seu tamanho de armazenamento.

# ALgoritmo para vetorização de *pixel arts*

O algoritmo estudado, que serve de base para esse trabalho de conclusão, foi publicado em 2011, por Johannes Kopf, pesquisador da Microsoft, e Dani Lischinski, professor da *Hebrew University* de Jerusalém. Esses apresentam uma abordagem totalmente nova para o problema da ampliação de *pixel arts*, a vetorização do *pixel art*. Ou seja, a conversão do *pixel art*, originalmente encontrado no formato matricial, para o formato vetorial, independente de resolução (Kopf; Lischinski, 2011).

Desta maneira, esta conversão possibilita a ampliação do *pixel art* para qualquer escala, eliminando o principal problema encontrado nos atuais algoritmos de aumento da escala, a limitação do grau de ampliação.

Segundo os autores, como o algoritmo desta abordagem não trata de um processo normal de vetorização, mas de um processo de vetorização de um tipo de imagem com características específicas, alguns desafios não triviais, pertinentes a imagens deste tipo, devem ser observados:

* **Todos os *pixels* são importantes.** Por exemplo, o olho de um personagem, é normalmente um *pixel* com a cor diferente dos seus vizinhos, e por isto não pode ser ignorado na imagem final.
* **As conexões com todos os oito *pixels* vizinhos devem ser observadas.** Quando existe uma conexão diagonal entre *pixels,* ela tende a parecer conectada quando a figura é observada em seu tamanho natural, entretanto, os *pixels* tendem a parecer desconexos quando ampliados. Sendo assim, é necessário que esse efeito seja respeitado durante o processo.
* **As ambiguidades locais devem ser resolvidas.** Por exemplo, em uma imagem de 2x2, em um padrão xadrez de duas cores, não fica claro quais *pixels* devem permanecer conectados. Estas ambiguidades devem ser resolvidas da melhor maneira possível, levando em consideração o contexto de cada *pixel*, se ele pertence ao fundo da imagem, ou a um detalhe ou traço.
* **A dificuldade em distinguir os efeitos criados devido as restrições de resolução da imagem dos efeitos pretendidos pelo autor.** As vezes é a intenção do autor que uma parte da figura seja representada de forma serrilhada, e nestes casos, ela não deve ser totalmente suavizada. Por exemplo, na figura 3.1(a), como definir se a boca deve permanecer ondulada, enquanto o contorno do fantasma deve ser suavizado.

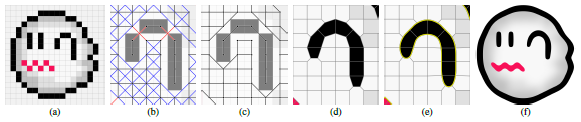


Figura .1 - Visão geral do processo de vetorização de *pixel arts*.

Fonte: KOPF; LISCHINSKI, 2011.

O principal elemento de representação vetorial utilizado nesta implementação é a primitiva *B-Spline*, que permite a aplicação de curvas dada uma sequencia de pontos de controle para a mesma. Para a definição destes pontos de controle, é necessário que os contornos da nova imagem vetorial estejam definidos, pois como está se tratando com *pixel arts,* algoritmos de detecção de bordas como o de Canny (CANNY, 1986) não podem ser utilizados.

## Grafo de similaridade

Desta forma, o primeiro passo para a execução deste algoritmo é a formação de um grafo de similaridade, que possui como principal objetivo a remodelagem do formato dos *pixels*, para que as conexões com seus vizinhos sejam consideradas, da maneira como apresentado na figura 3.1(d).

Inicialmente, neste processo, todo o *pixel art* é mapeado em um grafo. Cada *pixel* torna-se um nodo e cada nodo possui seus oito *pixels* vizinhos conectados através de arestas (figura 3.2). Neste momento, todas as conexões são analisadas, para que apenas as conexões pertencentes aos *pixels* de cores semelhantes sejam mantidas. Para isto, o mesmo critério utilizado pelo algoritmo HQX (conforme visto na sessão 1.1.4) é utilizado.

Sendo assim, as cores são convertidas para o padrão YUV (figura 1.8), onde são comparados os três canais de cor separadamente. Nesta comparação, caso seja detectado que as cores não são semelhantes, as arestas entre elas devem ser removidas. Esta análise de semelhança entre cores permitirá que efeitos de degradê possam ser adicionados mais tarde, tendo em vista que cores semelhantes permanecerão conectadas.

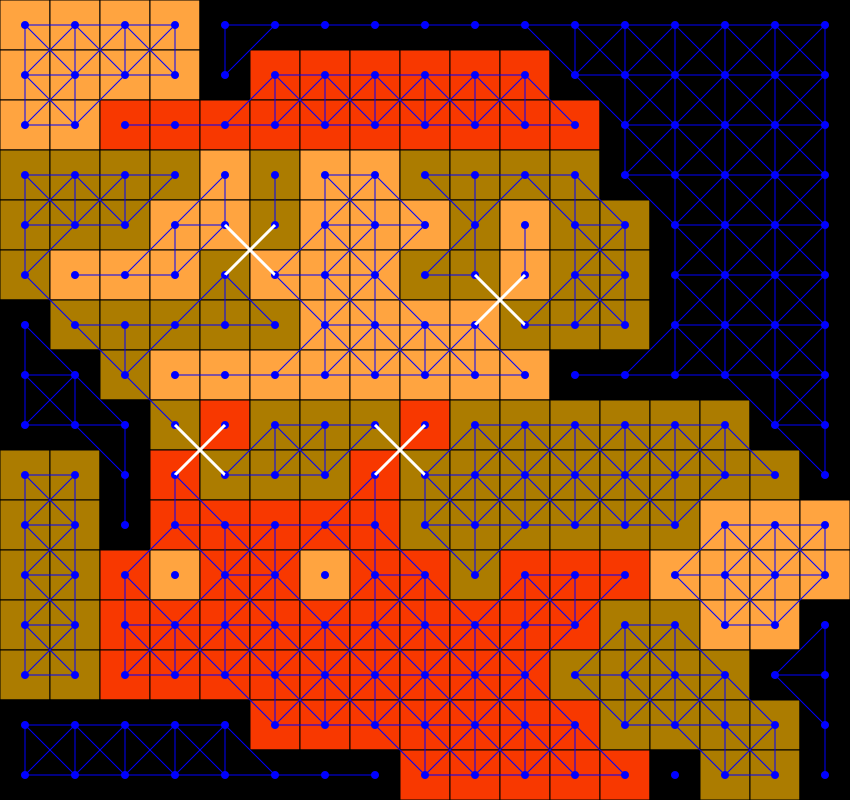


Figura .2 - Representação do grafo de similaridade, em seu primeiro estágio, sobreposto ao *pixel art* original.

Fonte: Imagem processada de Super Mario Bros, ©Nintendo, 1985.

Conforme apresentado na figura 3.2, no grafo de similaridade, os pontos azuis representam os nodos, e as linhas azuis, as arestas entre os nodos que se mantiveram conectados. Nesta fase mantêm-se conectado apenas os nodos com cores semelhantes, no entanto, alguns cruzamentos entre cores podem ocorrer (observe as linhas brancas na figura 3.2), e para que isto seja eliminado, é necessário que estas ambiguidades sejam resolvidas.

Como solução para estas ambiguidades os autores definiram três heurísticas, que devem ser consideradas para decidir qual das arestas deve ser eliminada.

### Heurística da curva

Este é o primeiro critério a ser verificado, se um dos *pixels* analisados pertence a uma curva, ou seja, está ligado a outros dois *pixels*. Caso isso ocorra, esta aresta deverá ser mantida, caso ambos pertençam a uma curva, a maior curva será mantida.

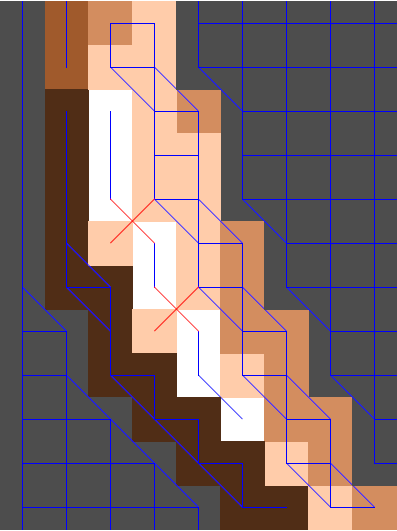


Figura .3 – Exemplo da heurística da curva. Neste caso os dois pontos de conflito serão resolvidos por esta heurística, que manterá todos os nodos brancos conectados.

Fonte: LOOS, 2011.

### Heurística dos *pixels* sobrepostos

O olho humano tende a ver um desenho contendo apenas duas cores e percebe a cor em maior quantidade como sendo a cor de fundo, e a cor em menor quantidade como estando a frente. Por isto, essa heurística propõem que se mantenham conectados os objetos que apareçam sobrepostos ao fundo, não deixando grupos de *pixels* que não pertencem a cor de fundo isolados. O algoritmo faz isso verificando um bloco de 8x8 ao redor da aresta que está sendo verificada. A aresta com menor número de *pixels* neste espaço será mantida conectada, pois não pertence a cor considerada como do fundo neste espaço.



Figura .4 - Exemplo da heurística dos *pixels* sobrepostos. Neste caso as conexões em roxo serão mantidas, por existir uma quantidade menor de *pixels* nesta cor.

Fonte: KOPF; LISCHINSKI, 2011.

### Heurística das ilhas

A última heurística é também a mais simples entre as três. Ela apenas se certifica que *pixels* não fiquem isolados, ou seja, caso exista uma cruzamento de arestas envolvendo um *pixel* que possa ficar isolado, este *pixel* deverá permanecer conectado, conforme apresentado na figura a seguir.



Figura .5 - Exemplo da heurística de ilhas.

Fonte: KOPF; LISCHINSKI, 2011.

Através do uso destas heurísticas, todas as ambiguidades podem ser resolvidas e o grafo de similaridade está finalizado. A partir daí é necessário realizar a remodelagem dos *pixels,* transformando-os em células poligonais que respeitem as conexões existentes no grafo de similaridade.

## Remodelagem das células de *pixel*

Uma das possibilidades para realizar isto é a utilização dos diagramas de Voronoi (AURENHAMMER, 1991), que consistem em dividir o plano em regiões determinadas pela distância de certos objetos. Neste caso, as coordenadas centrais do nodo e metade de suas arestas seriam os locais a se considerar, em relação as arestas e nodos vizinhos. No entanto, os resultados do Voronoi, não representam exatamente o necessário para este caso. Mais informações sobre Voronoi podem ser obtidas no artigo de Aurenhammer.

Como não é desejado o nível de precisão dos diagramas de Voronoi, é utilizada uma versão simplificada do mesmo, onde os novos pontos que formam o polígono de cada célula já estão pré-determinados para todos os casos possíveis.

Cada célula é analisada, em conjunto com seus oito vizinhos, observando todas as conexões existentes entre estes nodos para a aplicação do novo modelo de célula, que inclui todos os pontos necessários para a formação do novo polígono. Uma vez mapeados todos os padrões, para remodelar a célula, basta percorrer todo o grafo de similaridade, substituindo-se cada nodo por sua nova forma.

Na figura a seguir é possível verificar 18 exemplos de novos modelos para a célula de *pixel*. Na primeira imagem de cada modelo pode-se observar o *pixel* analisado e as respectivas conexões existentes entre ele e os *pixels* vizinhos. Na segunda imagem, pode-se verificar a nova forma do *pixel*.

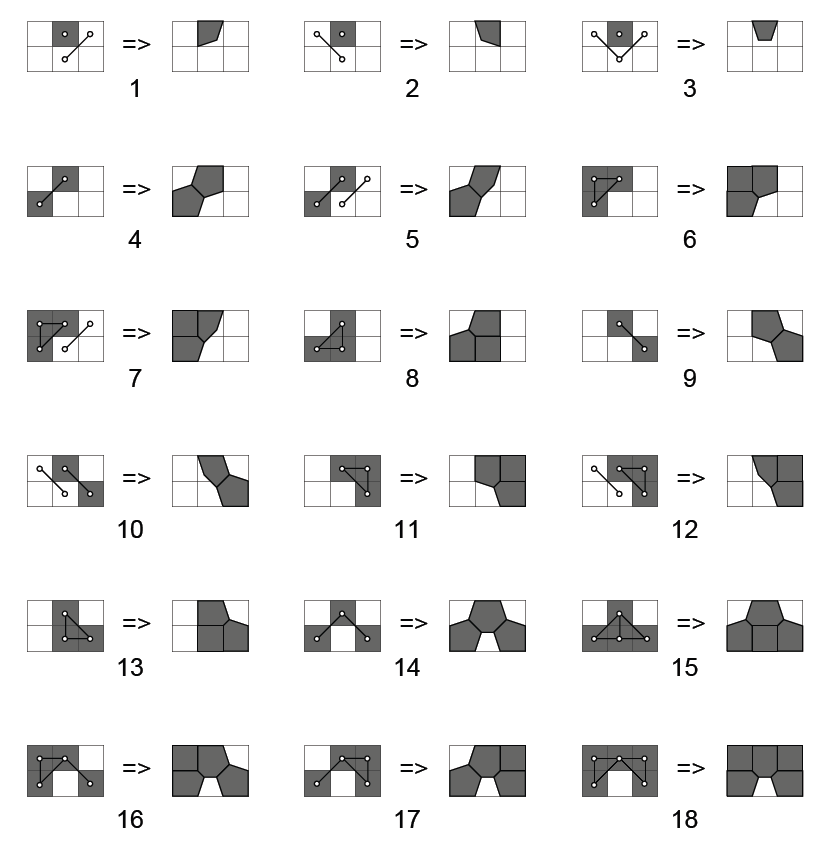


Figura .6 - Modelos de células remodeladas, antes e depois do processo.

Fonte: LOOS, 2012.

Estas novas células que substituíram o *pixel* darão a figura um aspecto menos quadriculado, sendo que agora ela é formada por polígonos e não mais simples quadrados. Essa mudança pode ser mais bem visualizada na figura a seguir.

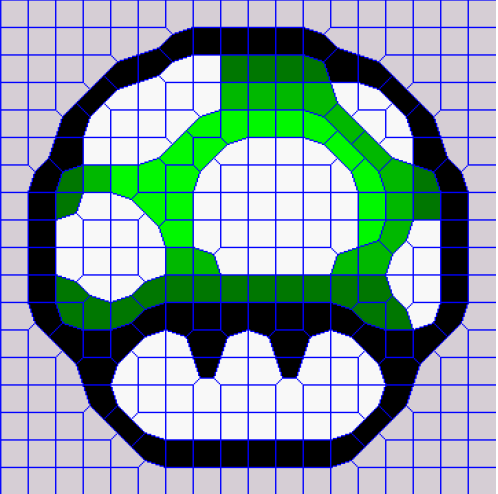


Figura .7 - Células remodeladas de acordo com o grafo de similaridade.

Fonte: LOOS, 2012.

Nesta remodelagem dos *pixels* é importante destacar quais bordas são visíveis, ou seja, possuem um vizinho de outra cor. Pois, é baseado nas arestas visíveis destes novos polígonos que o sistema irá aplicar as curvas de Bézier.

## Aplicação de curvas

Agora a informação encontrada pode ser utilizada na definição dos pontos de controle necessários para a aplicação de curvas na imagem (conforme sessão 2.2). Inicialmente, a partir de uma aresta visível, do novo grafo, se adicionam as coordenadas do nodo como início e fim desta borda, como pontos de controle. A seguir, é verificada a quantidade de bordas visíveis conectadas a este nodo, se a quantidade for dois, isso significa que existe outro nodo conectado a este. Sendo assim, não faz sentido mantê-los em curvas separadas, e ele é adicionado como um ponto de controle adicional a esta curva. Continua-se até chegar a um nodo que possua um número de arestas visíveis diferente de dois ou o nodo inicial for alcançado. Obviamente, se um nodo tem apenas uma aresta visível, não é possível continuar a curva. Se a quantidade de nodos é três ou mais, não existe como proceder, consequentemente, se inicia uma nova curva a partir deste ponto. Procede-se, desta maneira, até que todas as arestas visíveis estejam conectadas a uma curva.

A aplicação das curvas *B-Spline* melhora muito a suavidade do resultado final. No entanto, a imagem ainda possui alguns efeitos de ondulações que podem ser suavizados ainda mais, para isto, o algoritmo realiza uma otimização da localização dos pontos de controle das curvas. O resultado final pode ser verificado na figura 3.1(f).

Através da pesquisa realizada nestes três últimos capítulos foi possível o desenvolvimento do método próprio que será descrito no próximo capitulo

# Método APLICADO

Este capítulo realiza a apresentação da aplicação desenvolvida, que inclui os principais métodos de aumento da escala de *pixels* (capítulo 1), além de apresentar o método desenvolvido para a vetorização de *pixel arts,* obtendo, como resultado, uma imagem no formato aberto SVG (conforme sessão 2.2).

## Aplicação

Foi criado um ambiente para a implantação dos algoritmos. Neste ambiente, foram implementados os algoritmos de ampliação de *pixel arts*, apresentados no capítulo 1, para uma melhor compreensão de seu funcionamento e avaliação de seus resultados. Além disto, este mesmo ambiente serviu como base para a prototipação do algoritmo de vetorização proposto.

O ambiente para aplicação dos algoritmos contempla uma interface para a manipulação das imagens e aplicação das funções pretendidas. Sendo as seguintes:

* Abrir arquivos de imagens.
* Salvar arquivos de imagens.
* Aplicar ampliação na imagem (*zoom*) para melhor visualização dos resultados em tela (sem alterações nos dados da imagem, apenas apresentação ampliada).
* Opção para visualizar as imagens com a utilização dos filtros de suavização mais comuns como *bicubic*, *bilinear* e *Nearest-neighbor* (sessão 1.1).
* Opção para aplicação dos algoritmos de aumento de escala do *pixel art, EPX, Scale2x e HQX*, apresentados no capítulo 1.
* Visualizar as imagens em paralelo, possibilitando a visualização do antes e depois dos algoritmos.
* Geração do arquivo vetorizado, no formato aberto SVG, salvando-o diretamente em arquivo, para visualização em aplicativo externo específico.

## Tecnologia

Para o desenvolvimento do aplicativo foi selecionada a plataforma da Microsoft .NET, mais especificadamente C#, com desenvolvimento através da ferramenta Microsoft Visual C# 2010 Express (Microsoft, 2010), para execução da aplicação nos sistemas operacionais Windows XP, Windows Vista ou Windows 7, com o framework .NET.

Está decisão foi movida pela possibilidade de estudo da tecnologia pelo autor, além da praticidade em caso de futuras portabilidades da aplicação para o ambiente web, ou ainda a implementação de uma versão otimizada em C++, para execução junto a emuladores que disponibilizam interface para integração de filtros de imagem.

Além desta linguagem de programação, para a geração dos arquivos vetoriais, será utilizada a linguagem SVG (W3C, 2011), descrita na sessão 2.2.

## Desenvolvimento

Para o algoritmo proposto foi desenvolvida uma adaptação do método descrito por Kopf e Lischinski (2011), conforme o capítulo 3. No decorrer do desenvolvimento, ajustes serão realizados, buscando melhorias em termos de qualidade do resultado final e processamento.

### Esquema de desenvolvimento proposto

O desenvolvimento consiste em três estágios, iniciando pelo desenvolvimento do ambiente para a manipulação das imagens, seguido da implementação dos principais algoritmos de aumento de escala de *pixel arts*, e finalmente o desenvolvimento do algoritmo de vetorização de *pixel arts,* conforme o esquema a seguir.

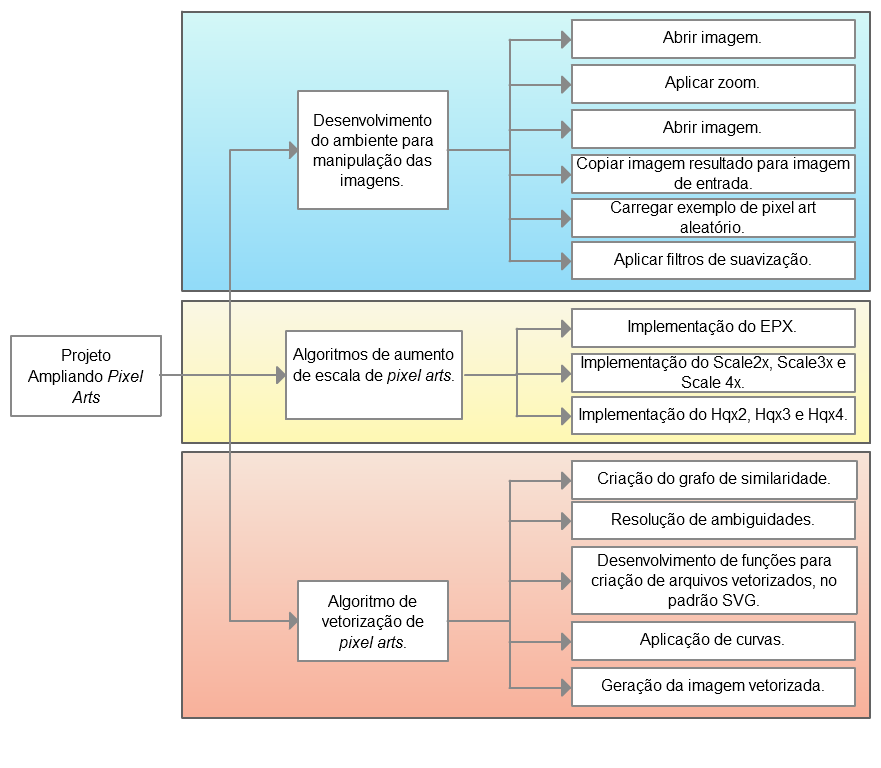


Figura .1 - Esquema de desenvolvimento.

Fonte: Desenvolvido pelo autor.

## Algoritmos

Nas sessões a seguir serão apresentados os algoritmos desenvolvidos na terceira fase do projeto. Sendo que os algoritmos referentes as duas primeiras fases são trivias, visto que, tratam-se da implementação da interface e dos algoritmos já existentes na bibliografia que serviram como base para estudo e compreensão do problema.[[1]](#footnote-1)

### Mapeando a imagem em um grafo

O algoritmo proposto inicia seu processamento específico percorrendo todos os *pixels* da imagem de origem e armazenando estes *pixels* em uma estrutura de grafo. Para a manipulação desta estrutura foi utilizada a biblioteca *quickgraph* (QUICKGRAPH, 2008). Neste grafo cada nodo é constituído de um objeto com atributos que definem a posição do pixel na imagem, sua cor e a valência do pixel, ou seja, a quantidade de arestas deste nodo.

Nesta estrutura, inicialmente armazena-se o *pixel* sendo processado e em seguida é realizada uma análise para localizar entre os *pixels* vizinhos se algum possui a mesma cor, caso exista, uma nova aresta entre eles é adicionada ao grafo.

Nesta análise são apenas verificadas as relações entre o atual *pixel* e os três vizinhos superiores e seu vizinho da esquerda, para que o algoritmo possua um melhor desempenho e para que não sejam criadas mais de uma aresta entre os mesmos *pixels* vizinhos. Como resultado nesta fase tem-se um grafo mapeado de acordo com a figura 3.2.

### Resolução das ambiguidades

De maneira semelhando ao algoritmo de Kopf e Lischinski (2011), nesta fase são localizadas e resolvidas as ambiguidades resultantes do processo anterior, utilizando-se as mesmas heurísticas propostas por eles, conforme sessão 3.1. Na figura 3.2, estas mesmas ambiguidades podem ser vistas destacas na cor branca. O resultado obtido é representado pela figura a seguir, onde se pode observar que essas foram integralmente resolvidas.

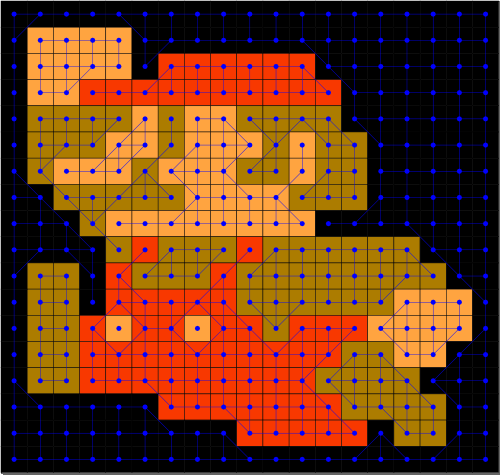


Figura .2 - Grafo com ambiguidades resolvidas

Fonte: Desenvolvido pelo autor.

### Remodelagem do *pixel*

Com a estrutura mapeando todos os *pixels* que devem permanecer conectados é possível remodelar o formato das células de pixel para um novo formato que respeite estas conexões.

Para esta fase do algoritmo uma abordagem diferente da apresentada por Kopf e Lischinski (2011) foi implementada. Cada *pixel* é analisado e um polígono equivalente é criado. O procedimento inicia por um dos quatro cantos do *pixel* e, baseado nas conexões existentes entre os vizinhos, o circula, determinando os pontos do novo polígono.

A definição destes pontos foi obtida através de desenhos em papel e diversos testes para localizar as posições ideais, onde não existam conflitos entre células. Este processo pode ser mais bem compreendido com a análise do algoritmo simplificado a seguir.

|  |
| --- |
| Para cada pixel do grafo  //**Canto Superior Esquerdo**  Se existe uma aresta entre este pixel e o pixel superior esquerdo  adicionaPonto(x - 2, y + 2)  adicionaPonto(x + 2, y - 2)  Se existe uma aresta entre o pixel superior e o vizinho da esquerda  adicionaPonto(x + 2, y + 2)  Senão  adicionaPonto(x + 0, y + 0) // Posição atual do pixel  //**Canto Superior Direito**  Se existe uma aresta entre este pixel e o pixel superior direito  adicionaPonto(x + 5, y - 2)  adicionaPonto(x + 9, y + 2)  Se existe uma aresta entre o pixel superior e o vizinho da direita  adicionaPonto(x + 5, y + 2)  Senão  adicionaPonto(x + 7, y + 0)  //**Canto Inferior Direito**  Se existe uma aresta entre este pixel e o pixel inferior direito  adicionaPonto(x + 9, y + 5)  adicionaPonto(x + 5, y + 9)  Se existe uma aresta entre o pixel inferior e o vizinho da direita  adicionaPonto(x + 5, y + 5)  Senão  adicionaPonto(x + 7, y + 7)  //**Canto Inferior Esquerdo**  Se existe uma aresta entre este pixel e o pixel inferior esquerdo  adicionaPonto(x + 2, y + 9)  adicionaPonto(x - 2, y + 5)  Se existe uma aresta entre o pixel inferior e o vizinho da esquerda  adicionaPonto(x + 2, y + 5)  Senão  adicionaPonto(x + 0, y + 7)  Fim Para |

Figura .3 - Algoritmo para remodelagem dos *pixels*[[2]](#footnote-2)

Fonte: Desenvolvido pelo autor

No mesmo momento que estes pontos são definidos, eles já são adicionados a um novo grafo, onde cada nodo será representado pela localização do ponto e cada aresta possui a cor dos *pixels* envolvidos na ligação. Este recurso será importante para as próximas fases. O resultado da imagem neste estágio pode ser verificado na figura a seguir.

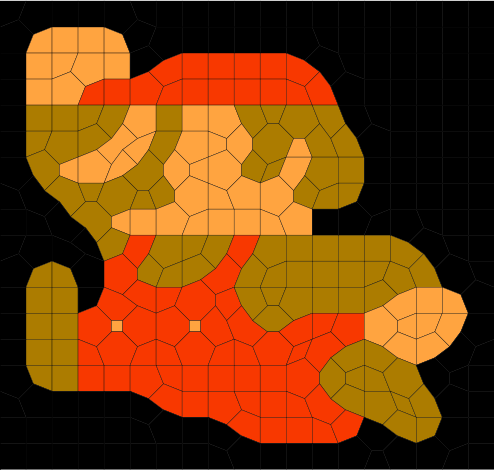


Figura .4 - Resultados após remodelagem dos *pixels.*

Fonte: Desenvolvido pelo autor.

### Definição das curvas da nova imagem

Este estágio começa com um grafo que fornece todas as arestas visíveis das novas células, ou seja, as que se encontram entre duas cores. Para o desenho das novas curvas é necessário ligar estas arestas, formando assim, as sequencias de pontos que formarão as curvas da imagem final.

Este procedimento percorre as arestas do grafo, inicialmente a procura de arestas que possuam nodos de valência diferente de dois, pois estes nodos indicam que existe a intersecção com mais cores e, portanto, trata do início ou final de uma curva. Uma vez localizado o início, são percorridos todos os pontos subsequentes desta curva, até terminar em um nodo com valência diferente de dois novamente.

Ainda é necessário o tratamento especial para as arestas que sobraram, estas arestas formam um *loop* de nodos com valência dois, como, por exemplo, os botões no macacão do Mario (figura 4.4). Para estes casos, basta selecionar uma aresta e percorrer as seguintes até localizar a primeira novamente.

Estas sequências de pontos são agrupadas em ordem dentro de *arrays,* formando assim curvas. Cada curva possui uma cor associada, em sua grande maioria, com exceção das curvas que formam as bordas da imagem, que são armazenadas duas vezes, uma para cada cor que cerca a aresta. Por exemplo, a curva que forma o contorno da orelha do Mario (figura 4.4), é adicionada uma vez com a cor amarela e outra com a cor marrom, pois fará parte dos dois objetos na próxima fase.

A imagem a seguir representa curvas com suas respectivas terminações destacadas por pontos vermelhos. Também é possível verificar a curva destacada em verde, que apresenta as terminações dos pontos que a constituem.

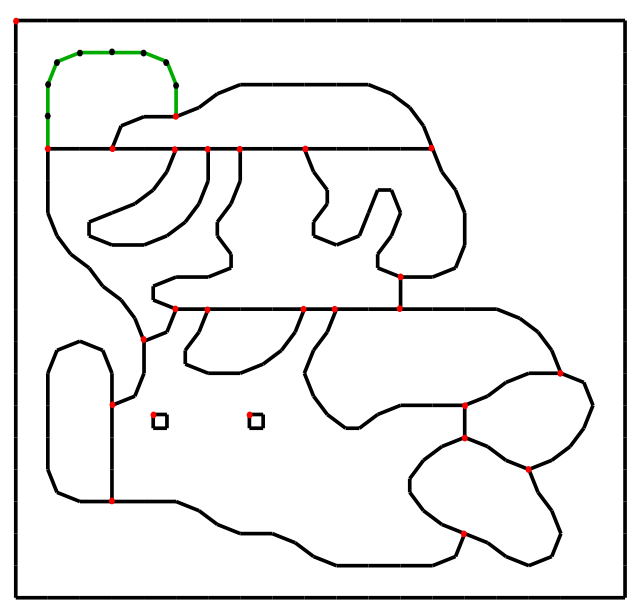


Figura .5 - Mapeamento das curvas da imagem e suas terminações. [[3]](#footnote-3)

Fonte: Desenvolvido pelo autor.

Esta curva destacada em verde, na figura 4.5, tem sua organização de dados apresentada no quadro 4.1. Através dele é possível verificar que as cores amarela e preta são armazenadas, pois o preto corresponde a cor do fundo e amarelo a cor da mão do personagem, conforme pode ser visto na figura 4.4. Esta informação será necessária para unir as curvas mais tarde.

Quadro 4.1 – Estrutura de dados de uma curva da imagem.

|  |  |  |  |
| --- | --- | --- | --- |
| x | y | Cor A | Cor B |
| 35 | 21 | RGB = (255, 164, 64) = Amarelo | RGB = (0, 0, 0) = Preto |
| 35 | 14 |
| 33 | 9 |
| 28 | 7 |
| 21 | 7 |
| 14 | 7 |
| 9 | 9 |
| 7 | 14 |
| 7 | 21 |
| 7 | 28 |

Fonte: Desenvolvido pelo autor.

### Definição dos objetos da imagem

Para a geração de uma imagem vetorial no formato SVG a partir dos pontos de curvas obtidos até este momento, é necessário que estes pontos de curvas estejam agrupados dentro de um objeto *path* (sessão 2.2) que percorra todo o perímetro do objeto até que o último ponto encontre o primeiro, formando assim, um caminho fechado. Desta maneira é possível associar a respectiva cor a este objeto, pois funções que dissipam a cor a partir de um ponto não estão disponíveis em SVG.

Para realizar a união das curvas da imagem obtidas, na última sessão, o algoritmo procede conforme segue. Todas as curvas que o último ponto desta curva é igual ao primeiro, logo esta curva já é um objeto e pode ser adicionada a lista de objetos da imagem. Caso contrário, verifica se entre todas as curvas não processadas qual de mesma cor que a analisada possui o primeiro ou último ponto na mesma posição do último ponto desta curva. Quando esta for localizada, ela é adicionada no final da curva anterior e continua a pesquisa até encontrar uma curva cujo último ponto é igual ao primeiro ponto da primeira curva. Ao chegar neste ponto, esta curva pode ser adicionada aos objetos da imagem.

Na figura abaixo, é possível observar o objeto que forma a mão do personagem, seguido do quadro 4.2, que apresenta estrutura de dados na qual foi armazenado este objeto. Os pontos de ligação entre as diversas curvas estão destacados com cores iguais.

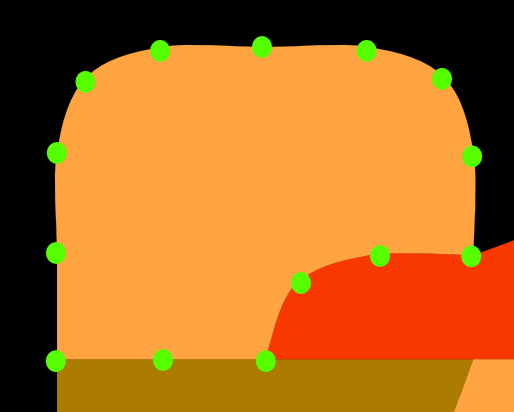


Figura .6 - Representação da mão do personagem e seus respectivos pontos

Quadro 4.2 – Estrutura de dados de um objeto da imagem.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Cor do objeto | | RGB = (255, 164, 64) = Amarelo | | | |
| Linha1 | | Linha2 | | Linha3 | |
| x | y | x | y | x | y |
| **35** | **21** | **7** | **28** | **21** | **28** |
| 35 | 14 | 14 | 28 | 23 | 23 |
| 33 | 9 | **21** | **28** | 28 | 21 |
| 28 | 7 |  |  | **35** | **21** |
| 21 | 7 |  |  |  |  |
| 14 | 7 |  |  |  |  |
| 9 | 9 |  |  |  |  |
| 7 | 14 |  |  |  |  |
| 7 | 21 |  |  |  |  |
| **7** | **28** |  |  |  |  |

Fonte: Desenvolvido pelo autor.

Após todas as curvas serem processadas e os objetos criados, é necessário que os objetos ambíguos sejam eliminados, como, por exemplo, o botão do macacão do personagem (figura 4.4). Devido ao processo de formação das curvas, existe uma curva amarela e outra vermelha e, em razão disto, dois objetos, ambos com os mesmos pontos, foram criados, um com cada cor.

Para que a solução destas ambiguidades seja encontrada, inicialmente são identificados os objetos com dimensões e posição iguais, mas com cores distintas. A seguir, através de um algoritmo que permite localizar um pixel dentro de um polígono (FINLEY, 2007), é possível encontrar algum *pixel* da imagem original localizado dentro do objeto, e a partir da cor deste pixel original, realizar a eliminação do objeto incorreto[[4]](#footnote-4).

O ajuste final desta fase do processo visa resolver o problema que será gerado na renderização do arquivo final. Quando os objetos são inseridos na imagem SVG eles são dispostos sobre os objetos anteriores e, caso possuam áreas conflitantes, irão sobrepor os objetos que vieram antes. Por exemplo, é necessário que o arquivo possua o objeto referente ao macacão vermelho do personagem disposto antes dos objetos que formam os botões amarelos do macacão, pois caso contrário, o macacão vermelho irá sobrepor os botões amarelos (figura 4.4),

Para corrigir esta questão, todos os objetos são ordenados utilizando o tamanho de suas áreas (FINLEY, 2006), para que os objetos maiores, como o macacão vermelho apresentado no exemplo anterior, sejam sempre inseridos antes no arquivo SVG, evitando que existam sobreposições indesejadas.

### Suavização e geração da imagem final

Com os objetos que formam a imagem já definidos, é necessário gerar o arquivo de saída do algoritmo. Para isso foi desenvolvida uma classe para geração de arquivos SVG, que utiliza das funções existentes em C# para manipulação de arquivos XML, sendo que arquivos SVG tratam-se apenas de uma aplicação do formato XML (sessão 2.2).

Para a codificação do resultado final foi utilizada unicamente a primitiva *path,* que permite o desenho de objetos dada uma sequência de coordenadas. No entanto, as coordenadas obtidas não são suficientes para formar de maneira suave o contorno do objeto utilizando os comandos padrões da linguagem SVG. Utilizando as curvas de Bézier, presente na codificação SVG (figura 2.9), são exigidos pontos de controle adicionais, que não foram obtidos.

Desta maneira, como não se possui os pontos de controle adicionais para a utilização de curvas de Bézier e não foi encontrada uma maneira de obtê-los, foi realizada uma pesquisa em formas alternativas para a representação das curvas em nosso projeto. Como resultado, se optou por utilizar a curva de Catmull-Rom (CATMULL; ROM, 1978).

Esta curva permite uma interpolação suave entre coordenadas chave sem a necessidade de pontos de controle adicionais, como na de Bézier, e estas coordenadas chave são exatamente os pontos que se obtém. Contudo, esta curva não é suportada pelo formato SVG. Mas sua aplicação no algoritmo foi possível com a utilização de uma matriz de conversão (figura 4.7), que permite sua representação como curva de Bézier (MILLINGTON, 2009), obtendo assim os pontos adicionais necessários e utilizando uma curva que é amplamente suportada pelo formato.

|  |
| --- |
|  |

Figura .7 – Matriz de conversão da curva de Catmull-Rom para Bézier.[[5]](#footnote-5)

Fonte: MILLINGTON, 2009

A figura a seguir demonstra o objeto que representa a mão do personagem antes e após a aplicação das curvas de Catmull-Rom.

|  |  |
| --- | --- |
|  |  |
| **Antes da aplicação das curvas** | **Após aplicação das curvas** |

Figura .8 - Antes e após a aplicação da curva de Catmull-Rom

Fonte: Desenvolvido pelo autor.

Após a conversão dos pontos, como todos os objetos que formarão a imagem final já se encontram na ordem correta (sessão 4.4.5) para a geração do arquivo vetorial. Neste momento, basta a criação do arquivo vetorial final e a inserção de uma primitiva *path* para cada objeto da imagem final associando a respectiva cor. O resultado final pode ser verificado na figura 4.9.



Figura .9 - Resultado final do algoritmo.[[6]](#footnote-6)

Fonte: Desenvolvido pelo autor.

## Análise dos resultados

O algoritmo desenvolvido foi aplicado a uma grande quantidade de *pixel arts*, alguns dos seus resultados podem ser verificados na imagem a seguir (figura 4.10), sendo apresentados ao lado de outras soluções utilizadas para a ampliação da escala de *pixel arts.*

|  |  |  |
| --- | --- | --- |
| C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\comparações\pacman_original.png | C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\comparações\pacman_epx.png | C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\comparações\pacman_michael.png |
| **Imagem original** | **EPX** | **Algoritmo desenvolvido** |
| C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\comparações\kart_original.png | C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\comparações\kart_scale3x.png | C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\comparações\kart_michael.png |
| **Imagem original** | **Scale3x** | **Algoritmo desenvolvido** |
| C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\comparações\mage_original.png | C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\comparações\mage_hq3x.png | C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\comparações\mage_michael.png |
| **Imagem original** | **Hq3x** | **Algoritmo desenvolvido** |
| C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\comparações\help_original.png | C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\comparações\help_kopf_linschinski.png | C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\comparações\help_michael.png |
| **Imagem original** | **Kopf e Linchinski** | **Algoritmo desenvolvido** |

Figura .10 - Comparação entre resultados de diversos algoritmos.

Fonte: Desenvolvido pelo autor.

Conforme demonstrado nas imagens anteriores, os resultados do algoritmo desenvolvido possuem uma melhor definição e qualidade que os algoritmos de aumento de escala EPX, Scale3x e HQX, principalmente pelo fato que estas técnicas são limitadas quanto à resolução, enquanto a solução vetorial não possui esta limitação, obtendo assim, um resultado mais suave e claro.

Entretanto, também é possível observar que o algoritmo desenvolvido possui uma qualidade de resultados inferior ao algoritmo proposto por Kopf e Linchinski (2011). Para uma análise das diferenças é interessante observar os resultados de ambos os algoritmos na imagem a seguir.

|  |  |
| --- | --- |
| C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\comparações\dolphin_kopf_linschinski.png | C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\comparações\dolphin_michael.png |
| **Kopf e Linchinski** | **Algoritmo desenvolvido** |

Figura .11 – Comparação do algoritmo desenvolvido com Kopf e Linchinski

Fonte: Desenvolvido pelo autor.

Conforme é possível verificar nesta imagem (figura 4.11), as principais diferenças estão relacionadas a dois aspectos não implementados no algoritmo deste projeto. São eles: a suavização dos pontos de controle das curvas e a aplicação de efeito de degradê entre cores semelhantes. É possível perceber que no algoritmo desenvolvido as curvas não tem um efeito tão suave quanto no resultado de Kopf e Linchinski e o efeito de degradê que é verificado entre os diferentes tons de azul do corpo do golfinho. No algoritmo desenvolvido as cores semelhantes não se mesclam da maneira como na outra abordagem.

Apesar de não ser o foco principal deste trabalho, dados referentes a performance do algoritmo foram obtidos. Os dados foram extraídos de um computador *single core* de 2,4 Ghz de CPU. Os testes são apresentados no quadro 4.1.

Quadro 4.3 - Análise de desempenho do algoritmo.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Imagem | Tamanho | Geração do grafo | Solução das ambiguidades | Remodelagem das células | Criação das curvas | Criação dos objetos | Total |
| C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\source\mslug2-1.png  Metal Slug | 40x52 | 582 ms | 401 ms | 2.164 ms | 24 ms | 351 ms | 3522 ms |
| C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\source\mario_8bit_transparent.png  Mario | 19x18 | 59 ms | 21 ms | 61 ms | 5 ms | 3 ms | 149 ms |
| C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\source\mage.gif  Mage | 20x28 | 79 ms | 38 ms | 165 ms | 3 ms | 8 ms | 293 ms |
| C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\source\SMB3.gif  Mario 3 | 25x29 | 121 ms | 54 ms | 281 ms | 15 ms | 8 ms | 479 ms |
| C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\source\supermariokart.png  Mario Kart | 34x35 | 224 ms | 148 ms | 743 ms | 11 ms | 32 ms | 1158 ms |
| Pacman | 18x19 | 78 ms | 8 ms | 64 ms | 11 ms | 16 ms | 177 ms |
| Ghost | 18x18 | 57 ms | 21 ms | 56 ms | 7 ms | 14 ms | 155 ms |
| C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\source\mushroom.png  Mushroom | 18x18 | 46 ms | 13 ms | 61 ms | 14 ms | 5 ms | 139 ms |
| C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\source\Bomberman.png  Bomberman | 17x26 | 55 ms | 20 ms | 105 ms | 14 ms | 11 ms | 205 ms |
| C:\Users\Usuario\Documents\My Dropbox\TCC\Imagens\source\dolphin_kopf.png  Dolphin | 42x18 | 128 ms | 58 ms | 297 ms | 15 ms | 14 ms | 512 ms |
| Média |  | 142,9 ms | 78,2 ms | 400 ms | 11,9 ms | 46,2 ms | 678,9 ms |

Fonte: Desenvolvido pelo autor.

Através destes dados, foi possível observar que o desempenho do algoritmo está diretamente ligado ao tamanho da imagem de entrada e, por consequência, ao tamanho do grafo gerado e o tempo de pesquisa dentro dele. Mas também existe uma grande influência da quantidade de cores e texturas da imagem. Quanto maior a fragmentação das cores, maior a quantidade de células remodeladas e, em consequência, mais pesquisas ao grafo e maior o tempo de processamento. É possível analisar esse aspecto observando a primeira figura do quadro 4.3.

CONCLUSÃO

Com base no estudo bibliográfico realizado, foi observado que o problema da apresentação de *pixel arts* em altas resoluções não possui uma solução trivial. Entre as abordagens existentes a que obtêm melhores resultados é a vetorização de *pixel arts,* proposta por Kopf e Lischinski (2011). Foi verificado que as técnicas que propõem o aumento da escala (sessão 1.1), apesar de conseguirem bons resultados, falham quando aplicadas para grandes resoluções. Assim como, entre as soluções para a vetorização de imagens em geral (sessão 1.2) foi possível observar que estas não atingem bons resultados no processamento de *pixel arts*. Isso porque, as características específicas dos *pixel arts* não se enquadram nas premissas destas abordagens.

A análise realizada sobre as formas de armazenamento de imagens possibilitou que se confirmasse a teoria, que para uma solução com melhores resultados seria necessária uma conversão para o formato vetorial. Isso pode ser observado profundamente, no capítulo 3, na análise realizada do algoritmo de Kopf e Lischinski (2011).

A pesquisa bibliográfica realizada permitiu que se formassem os conceitos utilizados na definição do algoritmo desenvolvido, apresentado no capítulo 4. Durante o desenvolvimento é interessante resaltar que as fases iniciais do mapeamento da imagem em grafo (sessão 4.4.1) e resolução das ambiguidades (sessão 4.4.2), não ofereceram muita dificuldade, tendo em vista que as técnicas de resolução de ambiguidades propostas por Kopf e Lischinski (2011) foram utilizadas.

Na fase seguinte, de remodelagem do pixel (sessão 4.4.3), extensivos testes, para definição dos pontos das novas células de pixel, foram necessários até ser encontrado um resultado satisfatório e sem a existência de conflitos.

Durante as etapas de definição das curvas (sessão 4.4.4) e definição dos objetos (sessão 4.4.5), devido as particularidades das imagens vetoriais no formato SVG, foram realizadas tentativas, através de diversas abordagens, até se obter a solução apresentada. A qual utiliza curvas, que contêm as cores referentes a ambos os lados da curva, para possibilitar no processo de definição dos objetos a ligação entre as curvas e a formação dos objetos.

De modo semelhante, a solução dos conflitos e sobreposições, apresentadas na sessão 4.4.6, só foram possíveis após a utilização dos algoritmos auxiliares para localização do pixel dentro de um polígono e a realização da ordenação dos objetos de acordo com suas áreas.

Durante a parte final do algoritmo (sessão 4.4.6), outra dificuldade foi encontrada. Não foi possível localizar os pontos adicionais para a utilização direta de uma curva de Bézier. Por isto, como os pontos existentes formavam os de uma curva de Catmull-Rom, optou-se pela representação através deste tipo de curva, convertendo-a para Bézier através de uma matriz de conversão, possibilitando assim, a sua apresentação no arquivo final SVG.

Após o a etapa de desenvolvimento, através de uma análise dos resultados obtidos, foi possível observar que o algoritmo é superior a outras abordagens, no entanto ainda não consegue superar a qualidade dos resultados apresentados por Kopf e Lishinski (2011).

Para trabalhos futuros é importante destacar a implementação de um processo de otimização dos pontos de controle das curvas e a aplicação de efeitos de degradê no resultado final. Ainda é uma possibilidade a conversão do algoritmo para linguagem C, com o objetivo melhorar a performance do mesmo.

É importante ressaltar que o atual algoritmo foi disponibilizado como projeto open source[[7]](#footnote-7) e espera-se com o auxílio da comunidade desenvolvedora de emuladores integrá-lo a um emulador de jogos, oferecendo assim, uma solução livre para o problema.

Outro fato a ser destacado é que as técnicas desenvolvidas na aplicação deste trabalho podem potencialmente beneficiar diversas aplicações da área da vetorização de imagens. Assim como da criação e manipulação de arquivos vetoriais, do formato aberto SVG.

Referências Bibliográficas

ADOBE, INC. ***Adobe Illustrator CS5.*** 2010. Disponível em: <http://adobe.com/products/illustrator/> Acesso em: 02/2012

Adobe. ***PostScript Language Reference***. 1999. Disponível em: http://www.adobe.com/products/postscript/pdfs/PLRM.pdf. Acesso em: 04/2012.

AURENHAMMER, F. Voronoi Diagrams - A Survey of a Fundamental Geometric Data Structure. ACM Computing Surverys, Vol 23 No 3, 1991.

Azevedo, E; Conci, A. **Computação Gráfica Teoria e Prática**. Editora Elsevier Ltda. 2003.

Boor, C. ***A practical guide to splines.*** Applied mathematical sciences. Springer-Verlag New York, 2001.

Canny, J. ***A Computational Approach To Edge Detection***. IEEE Trans. Pattern Analysis and Machine Intelligence, Vol. PAMI-8, No. 6, 1986.

CATMULL, E; ROM, M. ***Recursively Generated B-spline Surfaces on Arbitrary Topological Meshes.*** Computer-aided design, Elsevier, 1978.

Computer Desktop Encyclopedia. 2005. Disponível em: http://encyclopedia2.thefreedictionary.com/bitmap. Acesso em: 04/2012.

Cottee, S. ***PIXEL - A pixel art documentary.*** Disponível em: [http://www.simoncottee.com/2010/05/*pixel*-*pixel*-art-documentary.html](http://www.simoncottee.com/2010/05/pixel-pixel-art-documentary.html). Acesso em: 02/2012.

Eisenberg, D. ***SVG Essentials***. Editora O’Reilly. 2002.

Esposito, N. ***How Video Game History Shows Us Why Video Game Nostalgia Is So Important Now*.** 2005. Disponível em: <http://www.utc.fr/~nesposit/publications/esposito2005history.pdf>. Acesso em: 02/2012.

FINLEY, D. ***Point-In-Polygon Algorithm — Determining Whether A Point Is Inside A Complex Polygon.*** 2007. Disponível em: <http://alienryderflex.com/polygon/>. Acesso em: 07/2012

FINLEY, D. ***Ultra-Easy Polygon Area Algorithm With C Code Sample.*** 2006. Disponível em: <http://alienryderflex.com/polygon_area/>. Acesso em: 07/2012

Frost, J; Goessner, S; Hirtzler, M. ***Learn SVG – The Web Graphics Standard*.** 2012. Disponível em: http://www.learnsvg.com. Acesso em: 05/2012

Goldberg, A; Robert, F. ***ACM president's letter: Pixel art.*** ACM, Vol. 25. 1982.

Jack, K. ***Video Demystified***. 3ª ed. LLH Technology Publishing, Eagle Rock. 2001.

KAS, T. ***Fast Blit Strategies: A Mac Programmer's Guide.***1999. Disponível em: [http://www.mactech.com/articles/mactech/Vol.15/15.06/FastBlitStrategies/index.html](http://www.mactech.com/articles/mactech/Vol.15/15.06/FastBlitStrategies/index.html%20). Acesso em: 02/2012.

Kie Fa, D*.* ***2xSaI - The advanced 2x Scale and Interpolation engine*.** Projeto open source. 2001. Disponível em: <http://vdnoort.home.xs4all.nl/emulation/2xsai/>. Acesso em: 02/2012.

Kopf, J; Lischinski, D. ***Depixelizing pixel art.*** SIGGRAPH 2011.

Lai, Y; Hu, S; Martin, R. ***Automatic and topology-preserving gradient mesh generation for image vectorization.*** ACM Graph. v.28. 2009.

Lecot, G; Lévy, B. ***ARDECO: Automatic Region Detection and Conversion.*** *17th Eurographics Symposium on Rendering.* 2006.

Loos, C. ***Vectorization of Pixel art.*** Diploma Thesis, Universität Augsburg, 2012

Lyon, R**. *A Brief History of ‘Pixel’*.** Electronic Imaging: Digital Photography II, SPIE-proceedings. 2006.

MAME. **Projeto MAME (*Multiple Arcade Machine Emulator*).** Projeto open source. 2006**.** Disponível em: http://mamedev.org/. Acesso em: 03/2012.

Mazzoleni, A. ***Scale2x*.** Projeto open source. 2001. Disponível em: <http://scale2x.sourceforge.net/>. Acesso em: 02/2012.

MSDN. ***Microsoft Developer Network – Bitmap Storage***. 2012. Disponível em: http://msdn.microsoft.com/en-us/library/dd183391(v=vs.85). Acesso em: 05/2012.

MICROSOFT. ***Microsoft Visual C# 2010 Express***. 2010. Disponível em: http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-csharp-express. Acesso em: 05/2012.

MILLINGTON, I. ***Matrices and Conversions for Uniform Parametric Curves.*** 2009. Disponível em: http://idm.me.uk/math/curves.pdf. Acesso em: 08/2012

Murray, j; vanryper, W. ***Encyclopedia of Graphics File Formats.*** Second Edition. O'Reilly & Associates. 1996.

Orzan, A. et al. ***Diffusion curves: a vector representation for smooth-shaded images.*** ACM Trans. Graph. 27. 2008.

PRODANOV, C. C.; FREITAS, E. C. **Metodologia do Trabalho Científico – Métodos e Técnicas da Pesquisa e do Trabalho Acadêmico. Editora Feevale**, Novo Hamburgo, 2009.

QUICKGRAPH. ***QuickGraph 3.6: Generic Graph Data Structures and Algorithms for .NET***. Projeto open source. 2008. Disponível em: http://quickgraph.codeplex.com/. Acesso em: 09/2012

SEDERBERG, T. ***An Introduction to Bézier Curves.*** 2003. Disponível em: <http://tom.cs.byu.edu/~455/>. Acesso em: 05/2012.

Selinger, P. ***Potrace: a polygon-based tracing algorithm.*** Projeto open source. 2003. Disponível em: <http://potrace.sourceforge.net/>. Acesso em: 02/2012.

Stepin, M. ***HQ3x - High-quality 3x magnification filter.*** Projeto open source. 2002. Disponível em: <http://www.hiend3d.com/>. Acesso em: 02/2012.

Vector Magic, Inc. ***Vector Magic*.** 2010.Disponível em: <http://vectormagic.com> Acesso em: 02/2012.

W3C. ***Extensible Markup Language (XML) 1.0. Fifth Edition.*** 2008. Disponivel em: http://www.w3.org/TR/2008/REC-xml-20081126/. Acesso em: 04/2012

W3C. ***Scalable Vector Graphics (SVG) 1.1. Second Edition***. 2011. Disponível em: http://www.w3.org/TR/SVG11/. Acesso em: 04/2012.

WOLBERG, G. ***Digital Image Warping***. IEEE Computer Society Press Los Alamitos, CA, EUA. 1996.

XIA, T. et al. ***Patch-based image vectorization with automatic curvilinear feature alignment.*** ACM, Vol 28, No. 5. 2009.

1. Código fonte completo destas implementações está disponível em: https://github.com/michaelrbk/pixelart-vectorizer/blob/master/Image/Filters.cs [↑](#footnote-ref-1)
2. Código fonte desenvolvido disponível em: https://github.com/michaelrbk/pixelart-vectorizer/blob/master/Vectorization/Vectorize.cs#L225 [↑](#footnote-ref-2)
3. Código fonte do algoritmo disponível em: https://github.com/michaelrbk/pixelart-vectorizer/blob/master/Vectorization/Vectorize.cs#L420 [↑](#footnote-ref-3)
4. Código fonte disponível em: https://github.com/michaelrbk/pixelart-vectorizer/blob/master/Vectorization/Vectorize.cs#L577 [↑](#footnote-ref-4)
5. Código fonte da aplicação da matriz no algoritmo está disponível em: https://github.com/michaelrbk/pixelart-vectorizer/blob/master/Vectorization/SvgVector.cs#L194 [↑](#footnote-ref-5)
6. Código fonte da imagem SVG disponível em: https://github.com/michaelrbk/pixelart-vectorizer/blob/master/Resources/Mario.svg [↑](#footnote-ref-6)
7. Projeto na íntegra disponível em https://github.com/michaelrbk/pixelart-vectorizer [↑](#footnote-ref-7)