Universidade Federal Fluminense

Marco Aurélio Gonçalves da Silva

Real Time Pixel Art Remasterization on GPUs

NITERÓI

2013

Universidade Federal Fluminense

Marco Aurélio Gonçalves da Silva

Real Time Pixel Art Remasterization on GPUs

Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Topic Area: Visual Computing.

Advisor:
Prof. D.Sc. Anselmo Antunes Montenegro

NITERÓI

2013

Real Time Pixel Art Remasterization on GPUs

Marco Aurélio Gonçalves da Silva

Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Topic Area: Visual Computing.

Niterói, September 11th, 2013.

Acknowledgments

I would like to thank my parents Alcenir Carlos Dutra da Silva and Silvana Maia Gonçalves da Silva for all the support through my Master's degree studies and, of course, during my whole life.

To Anselmo Antunes Montenegro for all the support as advisor, standing for my problems, giving great ideas and freedom of choice.

To Esteban Walter Gonzales Clua, MediaLab coordinator, a place vitally important to the process of development and writing of this work.

To my friend Thales Luis Sabino which would always help me with technical issues. Also to my friends and great roommates Tássio Knop, João Paulo Peçanha, Daniel Madeira and Lucas Lattari for the company and all the enjoyable moments.

To Christian Ruff who was able to get a GPU for my workstation. To Giancarlo Taveira for the english latex template, to André Luiz Brandão and others colleagues at MediaLab.

To all my friends of my Bachelor's degree studies who are still there for me, specially Marcelo Caniato for all the tips during this period.

I also would like to thank the Foundation for Research Support of Rio de Janeiro State (FAPERJ) for the support (Process E-26/102.334/2012).

Resumo

Vários métodos foram propostos para resolver o problema de redimensionar *pixel art* através dos anos. Neste trabalho descrevemos uma nova abordagem para ser aplicada em uma arquitetura massivamente paralela, que é capaz de solucionar o problema em tempo real. O resultado final é uma imagem com regiões delimitadas por curvas suaves e livre de desconexão visual.

A maioria dos jogos antigos foram baseados em *pixel art* até a metade dos anos 1990's. Embora esses jogos possam parecer graficamente ultrapassados, ainda existe uma grande demanda por eles. Nosso objetivo principal é aplicar o método em quadros inteiros desses jogos antigos e manter a taxa de atualização do jogo original na saída do algoritmo.

Para alcançar este resultado desenvolvemos um algoritmo local e independente de contexto, que é eficientemente implementado na GPU, entregando quadros inteiros para o usuário processados em tempo de interação. O método pode ser visto como uma pipeline de cinco estágios, que são a construção do grafo de similaridade, construção do diagrama de células, subdivisão das células, triangularização das células e renderização. Para a formulação da abordagem algumas questões comuns quando se usa algoritmos paralelos surgem. Essas questões são descritas, assim como suas soluções.

Como etapa final, a renderização baseada em triangulos é mencionada. Finalmente os resultados são mostrados, incluindo comparações visuais e a taxa de atualização alcançada por diferentes resoluções de consoles.

Palavras-chave: arte em pixel, ampliação, vetorização, retro.

Abstract

Several methods have been proposed to overcome the pixel art scaling problem through the years. In this work we describe a novel approach to be applied through a massively parallel architecture that can address this issue in real time. The final result is a smoother image free of visual disconnections.

Most of old games were based on pixel art graphics until the half of the 1990's. Although they look graphically outdated, there is still a large demand for playing them. Our main goal is to apply the method on full frames of old games and be able to keep the frame rate of the original game as the output frame rate.

To achieve this result we designed a local and context independent algorithm that enables an efficient parallel implementation on GPU, delivering full frames output at response time for the user interaction. The method can be seen as a pipeline of five stages, which are the similarity graph building, cell diagram building, cell subdivision, cell triangulation and rendering. In order to formulate the approach some issues that naturally arise when using parallel computing had to be solved. These issues are described, as well as its solutions.

As the final step, the rendering technique is mentioned. Finally we show the results, including visual comparisons and the frame rate output achieved for different console resolutions.

Keywords: pixel art, upscaling, vectorization, retro.

"90% of what is considered impossible is, in fact, possible. The other 10%
will become possible with the passage of time and technology"

Hideo Kojima

Contents

Glossary

CPU	:	Computer Processing Unity
CUDA	:	Compute Unified Device Architecture
EPX	:	Eric's Pixel Expansion
FPS	:	Frame Per Second
GPU	:	Graphics Processing Unity
OPENGL	:	Open Graphics Library
PIXEL	:	Pixel Element
PSN	:	Playstation Network
SNES	:	Super Nintendo Entertainment System
VBO	:	Vertex Buffer Object

List of Figures

List of Tables

Chapter 1

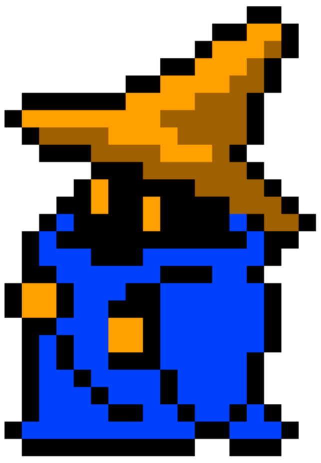
Introduction

For years games were designed in a pixel art graphical style due to the hardware constraints of the time. Back in the late 70's through the first half of the 90's, the video output of video games and PCs in general were limited to a few lines of pixels that were projected on the analogical video outputs, like CRT monitors and TVs. Until the early 90's, consoles and arcade machines hardly had more than 224 lines of resolution. Another restriction was that artists had to deal with a limited number of colors simultaneously available on the color palette. At the same time Operational Systems also used pixel art graphics style to represent features like icons and cursors.

Pixel art is a form of digital art, created through the use of raster graphics software, where images are edited on the pixel level [Wikipedia 2013]. When the artist is creating pixel art he follows the idea that every single pixel matters in the final result. Figure 1.1 shows an example of a classic pixel art sprite and a modern game with pixel art based design.

Although games based on pixel art can look outdated when compared to modern consoles titles, mainly in terms of graphics, there is still a large demand for them, thanks to the nostalgia of older gamers and the discovery of young gamers. This is clearly evidenced by the digital media delivery service of today's consoles, like the PSN (Playstation Network), Xbox Live and Nintendo Network, which offers a lot of these old titles in their library. This movement towards the past is called "Retro" and is a cyclical and natural movement that comes as opposition to changes in different societies. In [Mendes 2013] Bernardo Mendes discusses the retro phenomenon for video games and analyses the reasons that led to this phenomenon.

While remixed versions of old classics have been released in the last years, just a


(a)

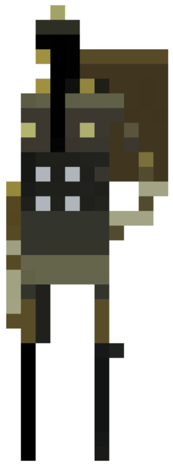

(b)

Figure 1.1: Pixel art examples (16x scale) (a) Final Fantasy (1989) (Original image ©Square Enix Holdings Co., Ltd) (b) Superbrothers: Sword & Sworcery EP (2011) (Original image ©Capybara Games Inc, Superbrothers Inc).

limited number of games receives this remasterization process and, in those cases, too much time and resources are spent to accomplish a high resolution result. For a example, DuckTales Remastered has been announced to be released on this USA summer and the active development of the title started late 2011 [Svensson 2013].

Directly scaling pixel art images by nearest neighbor filtering with the aim to use it in modern video devices results in blocky images with visual disconnections, an issue that was not supposed to appear in the original representation. Figure 1.2 shows the visual disconnection problem. The pixels diagonally adjacent on the TV antenna in Figure 1.2 share only one vertex, causing the disconnection problem. Most algorithms existing for upscaling pixel art are originated in the emulation community. The majority of such algorithms are open source and were not published by scientific paper.

The existing vectorization algorithms are supposed to be applied on natural images, but their use for pixel art input is not appropriate. A straightforward use of such algorithms causes the loss of small features of the original input. Thus, important details end up being lost, since in the design of pixel art every pixel matters as they were placed pixel by pixel by the artist.

In this work we present a novel approach for pixel art upscaling capable of achieving real time results. Our algorithm can deliver full frame output vectorized from pixel art input. The final result is smoother and free of disconnections as we can see in Figure 1.3. The algorithm response time enables the player to play pixel art designed games at any

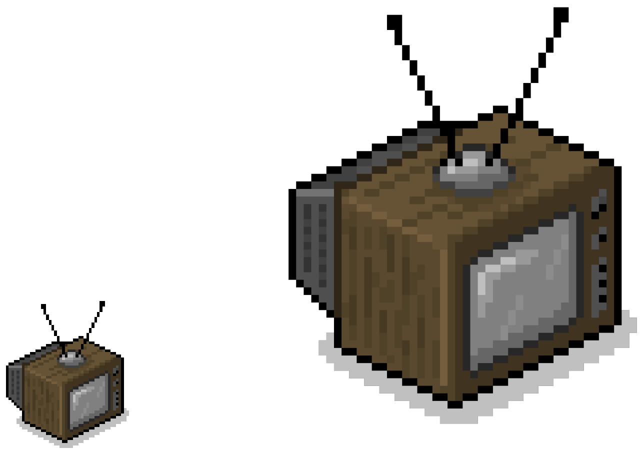

(a) (b)

Figure 1.2: Pixel art upscale visual disconnection example (a) Input (b) 16x nearest neighbor scale. The TV antenna became jagged, as the TV contour, showing sharp edges instead of a continuous line.

scale factor without the inherent problems of scaling this type of input. To achieve this result we propose a new algorithm that is able to use the power of a massively parallel architecture, more specifically, the GPU.

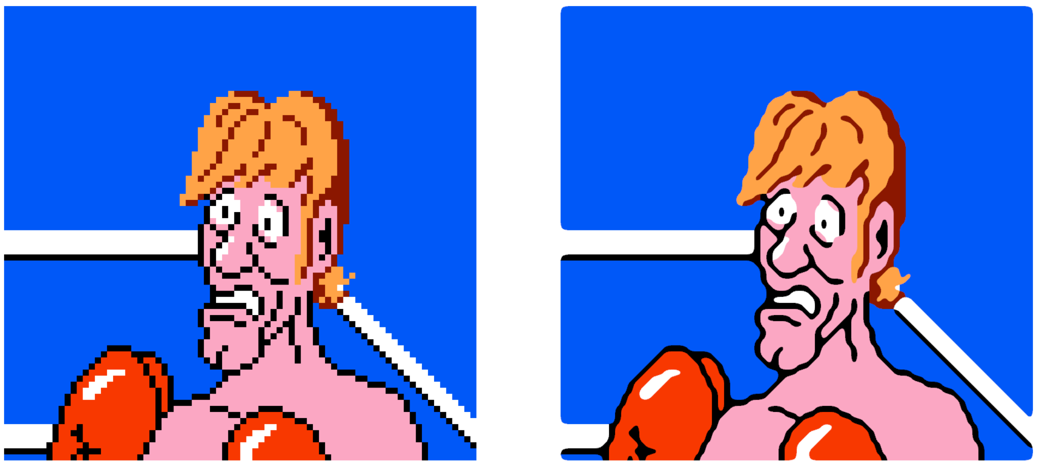

(a) (b)

Figure 1.3: Comparison between input and final result for a 10x scale (a) Nearest Neighbor (b) Our result (Original image ©Nintendo Co., Ltd).

Chapter 2

Related Work

Many methods have been proposed by the emulation community in order to enhance the display quality of old games' graphics on modern video devices. These methods were created by emulator programmers and were not published in scientific papers. Some emulators started implementing naive approaches like interlaced black lines (usually called *scanlines*) and linear interpolation which can be seen in Figure 2.1. These algorithms are far from yielding good results because they do not take into account substantial information of pixel art data, like the pixel neighborhood.

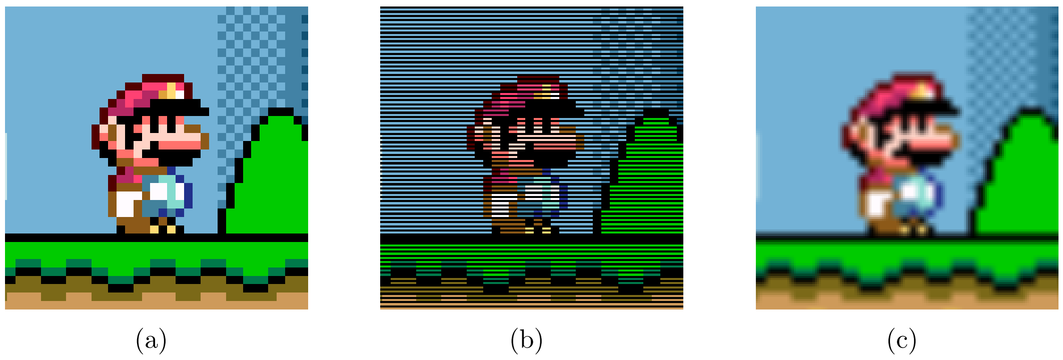


Figure 2.1: 4x Upscale (a) Nearest Neighbor (b) Interlaced black lines (c) Bilinear interpolation (Original image ©Nintendo Co., Ltd).

2.1 Emulation Community Algorithms

Pixel Art images are represented as bitmaps. A bitmap represents an image as a grid of RGB pixels. The algorithms created by the emulation community are all based in the idea of processing the input bitmap and producing as output another bitmap. This states their resolution dependence nature.

In 1992, LucasArts created the algorithm called EPX to port its games to a higher resolution platform [Wikipedia 2013]. Later, some versions of this algorithm, like Scale2x, appeared on emulators. The Scale2x produces an output with double of the input resolution. Scale4x version is basicaly Scale2x applied 2 times [Mazzoleni 2001]. This approach introduces some undesirable features like cross-shaped patterns at some points of the image. Algorithm 1 describes Scale2x rules using Figure 2.2 patterns. Figure 2.3 shows example of results of EPX and Scale2x applied 2 times to provide 4x scale.

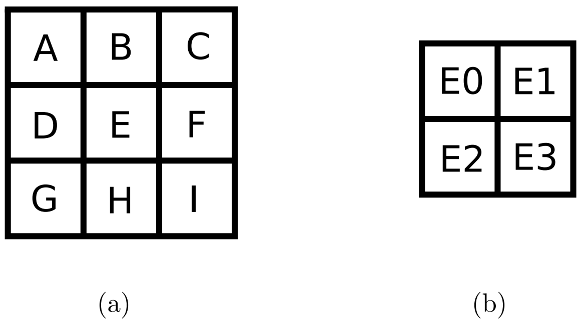
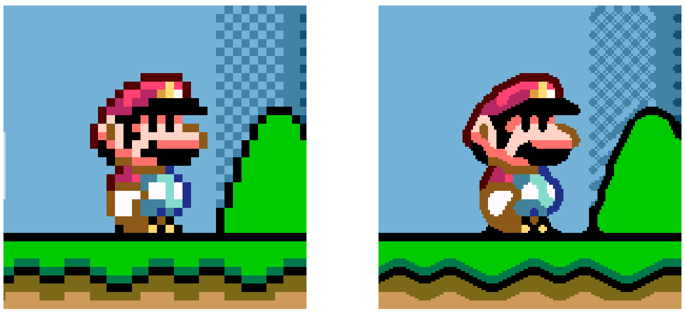


Figure 2.2: Patterns used on Scale2x algorithm (a) 9 pixels square to evaluate the 8 neighbors pixels (b) 4 pixels generated from input pixel E.

Algorithm 1 Scale2x Algorithm for the patterns of Figure 2.2

```
if B $\neq$ H & D $\neq$ F then
    E0 $\Leftarrow$ D = B ? D : E
    E1 $\Leftarrow$ B = F ? F : E
    E2 $\Leftarrow$ D = H ? D : E
    E3 $\Leftarrow$ H = F ? F : E
else
    E0 $\Leftarrow$ E
    E1 $\Leftarrow$ E
    E2 $\Leftarrow$ E
    E3 $\Leftarrow$ E
end if
```

There is also the Eagle algorithm, which can be seen in Algorithm 2, that creates blocks of 2x2 pixels by also comparing the neighbors of a pixel in a 3x3 block. In the sequel, more complex approaches were created like the 2xSaI algorithm that generates a scale of two times where the additional pixels are generated by detecting patterns such as lines and edges and interpolating additional pixels on that basis using techniques such as anti-aliasing. Super2xSaI (a combination of the Eagle and 2xSaI algorithm) and SuperEagle versions intend to do more blending [Wikipedia 2013].



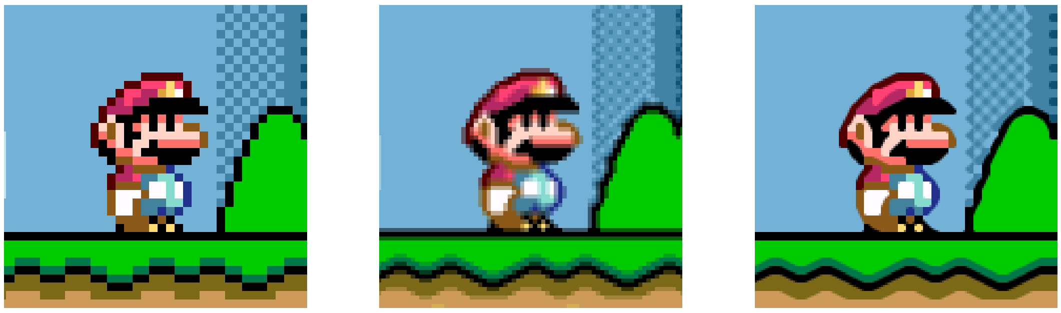
(a) (b)

Figure 2.3: 4x scale (a) Nearest Neighbor (b) Scale4x/EPX (Original image ©Nintendo Co., Ltd).

Algorithm 2 Eagle Algorithm for the patterns of Figure 2.2

```
$E0 \Leftarrow (D = A \ \& \ A = B) \ ? \ A : E$
$E1 \Leftarrow (B = C \ \& \ C = F) \ ? \ F : E$
$E2 \Leftarrow (D = G \ \& \ G = H) \ ? \ G : E$
$E3 \Leftarrow (H = I \ \& \ I = F) \ ? \ I : E$
```

The hqx family of algorithms [Stepin 2007] is known to have the best results of such methods used in emulation. Developed by Maxim Stepin the hqx exists in three versions: hq2x, hq3x and hq4x for two, three and four times scale, respectively. The algorithm compares each pixel with the eight neighbors using a threshold in the YUV color space and queries a pattern in a table to replace that pixel. The lookup table of hq4x version has 256 entries and returns a pixel block pattern that aims to smooth the final image. Figure 2.7 shows examples of results of 2xSaI and hqx.



(a) (b) (c)

Figure 2.4: 4x scale (a) Nearest Neighbor (b) 2xSaI (c) hq4x (Original image ©Nintendo Co., Ltd).

These approaches can produce reasonable results, but they are resolution dependent given the limited scale factor of 2x, 3x and 4x and, so, present poor results when displayed on larger resolution such as Full HD (1080p) devices. When the 4x scale result is not sufficient to fit the display size we still need a interpolation upscale.

2.2 Classic Vectorization and Novel Approach

Vector graphics is the use of geometrical primitives such as points, lines, curves, and shapes or polygon(s). In [Selinger 2003] a vector image is described as

> “A vector outline describes an image via an algebraic description of its contours, typically in the form of Bezier curves. The advantage of representing an image as a vector outline is that it can be resized as any size without loss of quality. Outline images are independent of the resolution of any particular output device.”

Vectorization algorithms were created to deal with general image input. Thus, they rely on techniques like segmentation and edge detection that do not perform well on pixel art input. This happens because in a pixel art image a single pixel makes an important role in the final art design and these techniques do not take these details into account.

The Potrace algorithm [Selinger 2003] produces good results, but is designed for black and white high resolution images. It decomposes the bitmap in a small number of paths, approximates each path by an optimal polygon, which is transformed into a smooth outline, finally the curve is optimized by joining consecutive Bezier curves segments together. When applied to pixel art it produces results like in Figure 2.5.

Vector Magic and Adobe Live Trace are commercial tools and details of these algorithms is not known, but they also do not perform well on pixel art input, as we can see on Figure 2.6.

A method to extract polygonal surfaces from volumetric models based on voxels is described in [Muniz 2012]. This is a similar problem, although applied to 3D models.

The most recent and solid addition to this field was the Depixelizing Pixel Art algorithm [Kopf and Lischinski 2011]. This method produces a vectorized image formed by b-spline curves and color diffusion. To achieve this result the approach relies on the building of a similarity graph that gives information about connectivity of similar color pixels. This similarity graph is used in the next step to reshape the pixels resulting in a cell diagram. Finally curves are extracted from this cell diagram and optimized to look smoother.

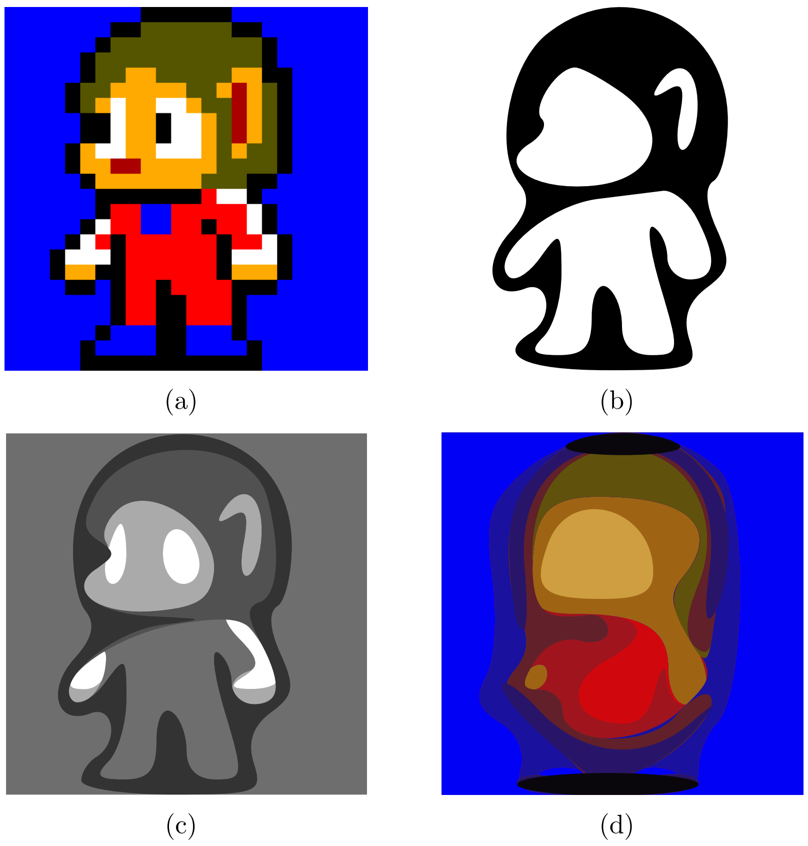


Figure 2.5: Potrace results for the Alex Kidd input (a) Original input (b) Potrace algorithm (c) Inkscape's trace bitmap based on Potrace (d) Inkscape's trace bitmap based on Potrace with colors (Original image ©Sega Corporation).

Although it produces great results, the global nature of the algorithm makes it very time consuming. In [Loos 2011], Loos reproduces Kopf and Lischinski work and gives a detailed explanation of the implementation, which includes some steps that are not described in details in the original work. Our method takes the main concepts of Depixelizing Pixel Art algorithm and proposes a novel parallel method for a full GPU solution.

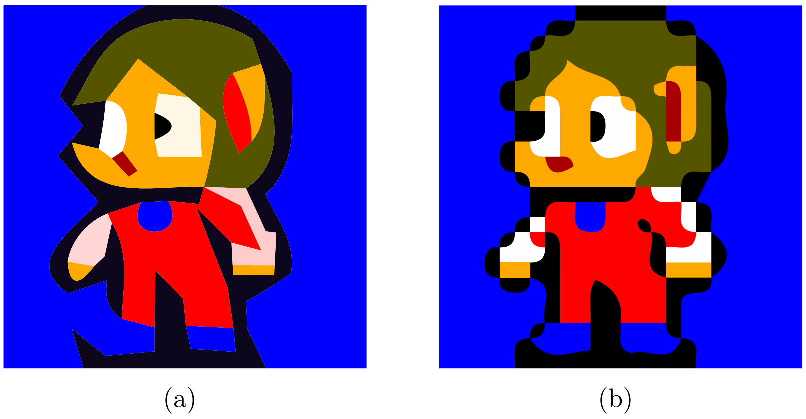


Figure 2.6: Results for the Alex Kidd input (a) Vector Magic (b) Adobe Live Trace.

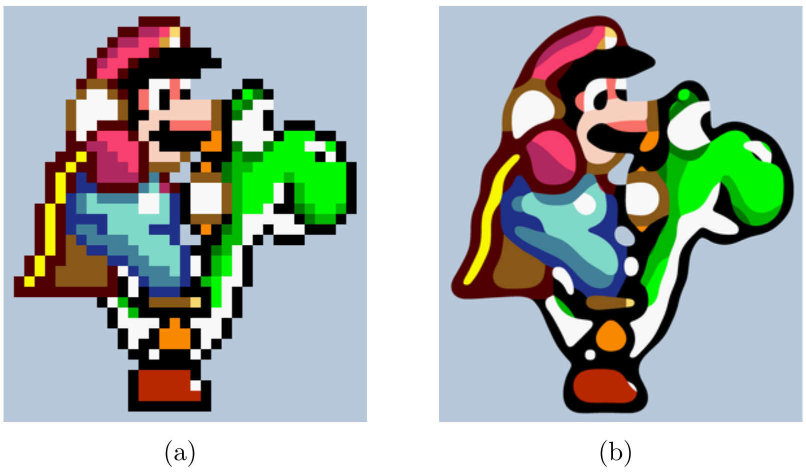


Figure 2.7: 8x scale (a) Nearest Neighbor (b) Depixelizing Pixel Art [Kopf and Lischinski 2011] (Original image ©Nintendo Co., Ltd).

Chapter 3

Proposed Method

3.1 Method Overview

The method proposed in this work is inspired by [Kopf and Lischinski 2011], named Depixelizing Pixel Art. Kopf and Lischinski's approach proposed to extract features at the pixel level, making every pixel relevant. While the proposed algorithm can achieve good results, because of its global nature, the extraction and, mainly, the optimization of the spline curves are very expensive and thus, because of the high processing time, it is impossible to use it for interactive frame rates.

With the purpose of processing a pixel art game input in real time, Kopf and Lischinski's method was adapted in this work and modified to fit a massively parallel architecture, more specifically, the GPU. This work was initially developed in [Silva et al. 2013]. Most of the steps can be solved locally or adapted to a local approach and they can be interpreted as a pipeline. The similarity graph building and cell diagram building steps are based on the same concept introduced by [Kopf and Lischinski 2011]. All the stages will be explained in the next subsections.

Figure 3.1 shows the pipeline as a diagram. The parallel computing platform used in this work was CUDA, created by NVIDIA. Each stage was implemented separately in a specific GPU CUDA kernel. In the first stage (Section 3.3) we extract the similarity graph which gives us information about pixel neighborhood. Each node of the graph has a specific pattern that will be used to shape a cell in the next stage (Section 3.4). These cells represent the reshaped pixels and already present a solution to the problem of diagonal discontinuity of the original pixel upscaling. The following stage (Section 3.5) takes these cells and subdivides their borders using a curve subdivision scheme returning a smoother polygon. On the final stage (Section 3.6), the cells are split into triangles and finally rendered.

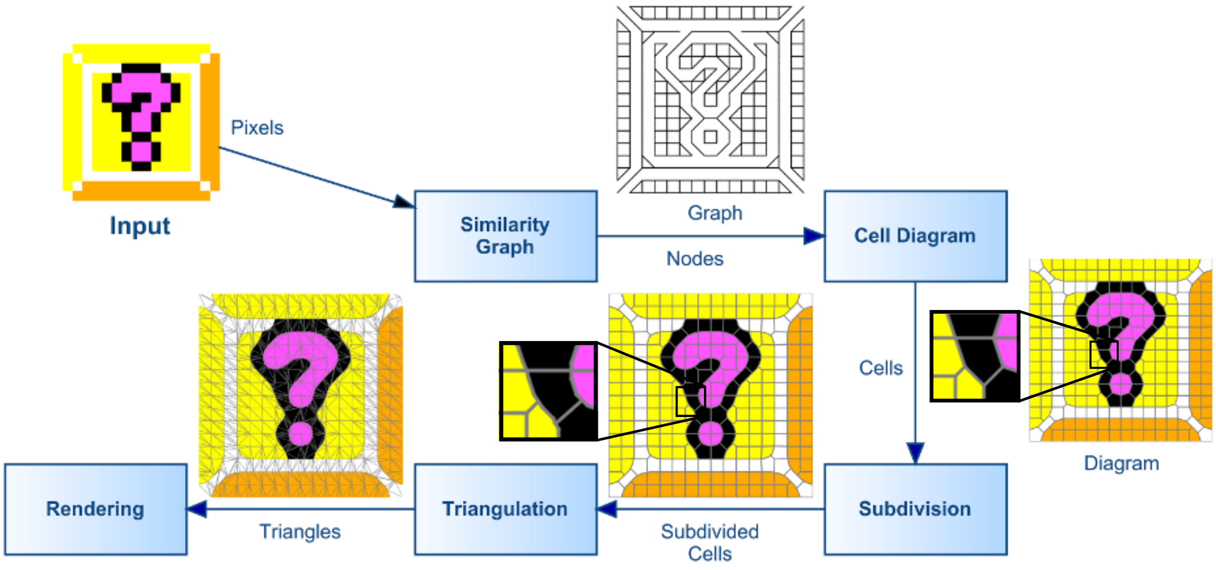


Figure 3.1: Pipeline Diagram. The Similarity Graph stage extracts a node for each input pixel. The Cell Diagram stage computes the cell shape for each graph node. In the following the cell border edges are subdivided to give the polygon a smoother aspect. Afterwards, cells are split into triangles that feed the rendering process.

Before we start describing the pipeline stages it is important to take an overview of the memory arrangement used. As we can see in Figure 3.1, the three main components of the method are the input image, the similarity graph and the cell diagram. Thus we have memory space arranged as an array to hold the data for each of these three objects. A pixel of the input image will have the same index of its relative node and cell of the graph and cell diagram respectively. The last two stages, the subdivision stage and the triagulation stage use the same memory space reserved to the diagram cell. Figure 3.2 shows the memory arrangement.

As much as cells are defined as a varying number of points, each unity of the diagram cell array must have a fixed size that can hold the cell defined by the maximum number of points, data that can be taken empirically. This implies that part of the array space is unused, because not every cell will reach this value defined as the maximum number of points of a cell, but the use of a dynamically sized array to hold the cells would lead to a very inefficient algorithm due to memory reallocation. Later it will be shown that, in fact, to determine the size of the diagram cell array we take into account the number of points generated to the cell triangulation to take advantage of a memory space already allocated. A fourth auxiliary array is used to store the size of each cell and it is indexed the same way the others are.

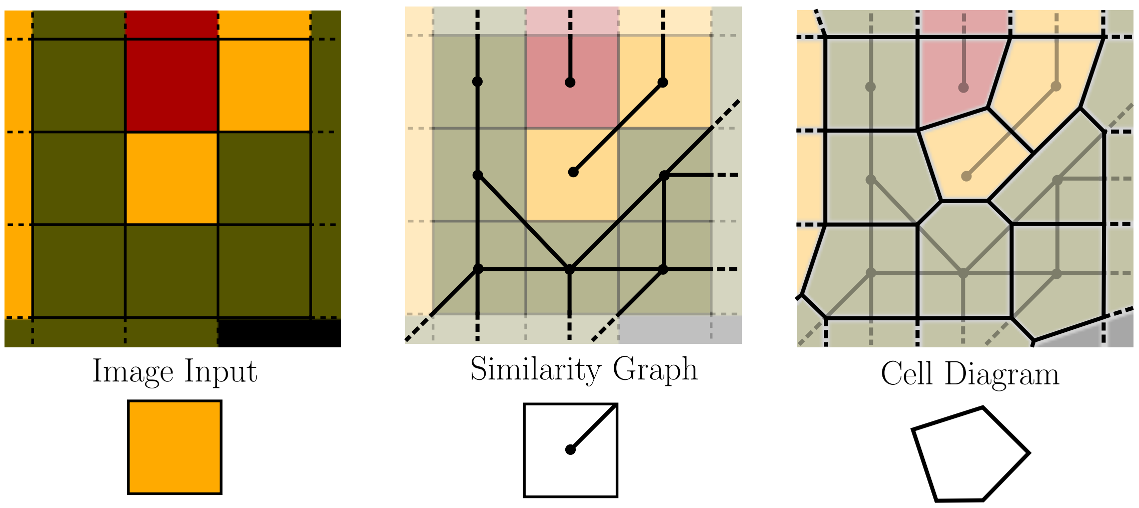


Image $[i]$ = Pixel$_i$
Pixel$_i = \{255, 170, 0\}$

Graph $[i]$ = Node$_i$
Node$_i = \{0, 0, 1, 0, 0, 0, 0, 0\}$

Diagram $[i]$ = Cell$_i$
Cell$_i = \{P_0, P_1, P_2, P_3, P_4\}$

Figure 3.2: Memory arrangement. A fourth array is used to store the size of each cell.

3.2 Nonparallel Approach Versus Parallel Approach

Before describing the method, it is important to understand the differences between a nonparallel approach and a parallel approach. [Kopf and Lischinski 2011] and [Loos 2011] implement a nonparallel approach for the method. While producing satisfactory visual results, the time taken to process a single input makes the algorithm unfeasible to produce the input game frame rate as output. Even if we ignore the curve optimization step these approaches could not handle real time playing.

The main reasons that make a typical nonparallel approach incapable of producing efficient results is the nonlocal nature of curves extraction. We need to walk on the similarity graph to find closed region borders and therefore extract its border that is defined by the external points of the border cells. Figure 3.3 shows an example of extraction of such points that will be used as the control points of a b-spline. This process of finding closed regions is a trial and error approach, since there will exist nodes that will be processed but do not make part of a closed region border, like the internal nodes. It is necessary to walk through the graph checking and classifying cells (corresponding to the current node) as already processed or discarded and always checking if we have achieved the first node to close the current closed curve being processed. A curve can be generated using b-splines after the extraction of the border cells points.

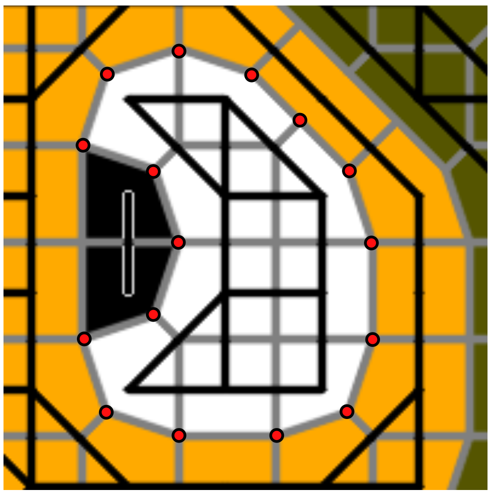


Figure 3.3: Curve extraction for closed regions. The black lines are the graph edges and the gray lines are the cells edges. The red points are the external points for the border cells of the closed region that forms the white area of the Alex Kidd eye.

In our approach the curves extraction is replaced by the subdivision of cell's border. Since each cell's edge can be divided independently, this step can be solved in parallel. As will be discussed later, the subdivision will be applied to the cell edges that will be determined as a border edges. Furthermore, the subdivision method used tend to produce a quadratic b-spline curve from the cell edges.

Not only the curve extraction is replaced by a parallel solution, but the other steps can be adapted as well. As the input is an image, which consist of a fixed number of rows and columns of pixels, we can massively parallelize this type of data by processing one pixel per thread. The input of the subsequent stages, the similarity graph and the cell diagram, will be two dimension data with the same number of elements as the number of the input image pixels. Hence, we can also process these data with one thread per element. Algorithms 3, 4 and 5 shows the kernel structure for these stages.

Algorithm 3 Graph kernel structure

```
{Thread Index}
$i \leftarrow$ (blockIdx.x $*$ blockDim.x $+$ threadIdx.x)
$j \leftarrow$ (blockIdx.y $*$ blockDim.y $+$ threadIdx.y)

**if** $(i < width)$ & $(j < height)$ **then**
    graph$[(j * width) + i] \leftarrow$ ProcessPixel(image$[(j * width) + i]$)
**end if**
```

Algorithm 4 Cell Diagram kernel structure

```
{Thread Index}
$i \leftarrow$ (blockIdx.x $*$ blockDim.x $+$ threadIdx.x)
$j \leftarrow$ (blockIdx.y $*$ blockDim.y $+$ threadIdx.y)

{edge_count stores cell edge number}
if $(i < width)$ & $(j < height)$ then
    diagram[$(j * width) + i$] $\leftarrow$ ProcessNode(graph[$(j * width) + i$],
                                          edge_count[$(j * width) + i$])
end if
```

Algorithm 5 Subdivision kernel structure

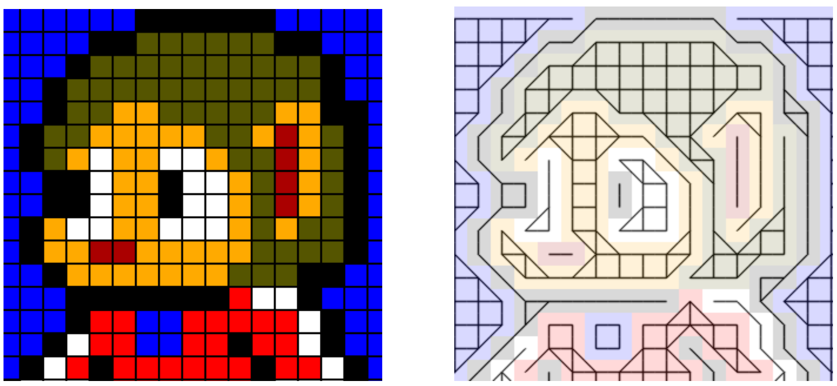
```
{Thread Index}
$i \leftarrow$ (blockIdx.x $*$ blockDim.x $+$ threadIdx.x)
$j \leftarrow$ (blockIdx.y $*$ blockDim.y $+$ threadIdx.y)

{edge_count stores cell edge number}
if $(i < width)$ & $(j < height)$ then
    diagram[$(j * width) + i$] $\leftarrow$ ProcessCell(diagram[$(j * width) + i$],
                                          edge_count[$(j * width) + i$])
end if
```

3.3 Similarity Graph

As in Depixelizing Pixel Art, our method relies on the notion of a *Similarity Graph*. The *Similarity Graph* is a graph induced by pixels structured according to a similarity measure. Let the similarity graph $SG = (V, E)$ be a graph where V is a set of nodes $n_0, n_1, \ldots, n_{n-1}$ induced by each pixel from the input image and E a set of edges. An edge e connects two nodes n_i and n_j associated to the pixels u and v if those pixels are neighbors and are considered similar according to a similarity threshold. The threshold used is a variation of the one used in the hqx algorithm to get a more sensible distinction of colors. In fact, the chosen threshold was $\frac{5}{255}$, $\frac{7}{255}$, $\frac{6}{255}$ for the YUV channels respectively. If the difference between two pixels is greater than the threshold, then they are considered to be dissimilar and will appear disconnected on the graph. A example of a similarity graph can be seen on Figure 3.4.

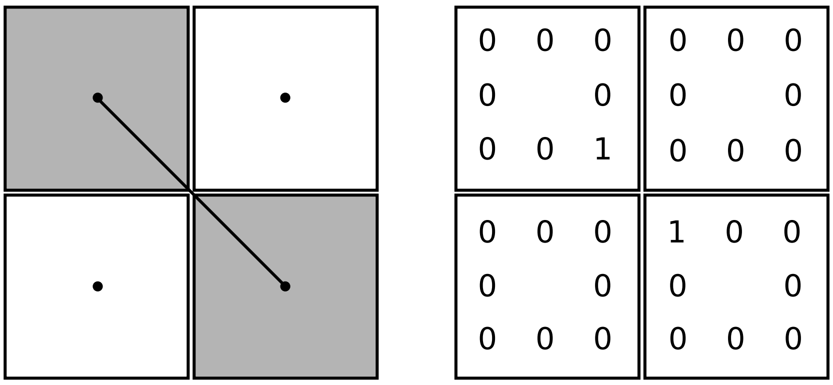
In our implementation, each edge e associated to two nodes n_i and n_j is represented by two *links*, one for n_i and another for n_j. Each node can store at most 8 links corresponding to similarity relations to its eight neighbors. The set of links is codified by a binary number of eight bits, one for each possible link in the eight directions. Figure 3.5 shows how links



(a) (b)

Figure 3.4: Similarity Graph extraction. (a) Input Image. (b) Similarity Graph (the black line represents the node edges).

are stored for each node.



(a) (b)

Figure 3.5: Edge representation. (a) 2x2 node similar graph for a 2x2 pixel block. (b) Links codification.

3.3.1 Parallel Design

Our method creates one node per pixel of the input image. The connectivity relation based on similarity can be computed in a parallel way using several threads running independently. The independence of threads can be achieved because their actions are limited to comparisons between the eight neighbors of the pixel and to the storage of the links related to the current node. The link data is made to be redundant, so the information about the connectivity of two nodes is stored in both nodes. Each node saves its link to a memory space which can be accessed by the index of the pixel corresponding to the node. This guarantees the independence of each thread.

So, when the kernel that checks for similar pixels launches a thread for each pixel, each thread will read the color information of each of the eight neighbors and write information about the connectivity status for the node that have the same index of the current processed pixel. This information is written in the memory array reserved for the similarity graph as can be seen on Figure 3.2.

3.3.2 Heuristics

After the first step we have a graph with crossing edges that must be removed in order to solve ambiguities that can arise where we cannot decide which valence-2 path should be maintained. Nodes belonging to 2x2 fully connected block are the trivial case and its crossing edges can be directly removed. A 2x2 fully connected block is the consequence of a 2x2 pixel block of the same color. Figure 3.6 shows the trivial case.

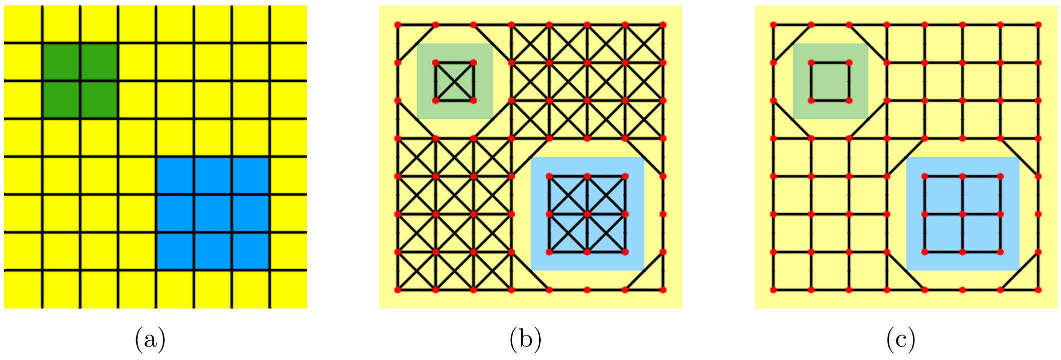


Figure 3.6: 2x2 fully connected block (trivial case) example. (a) Input image with 8x8 pixels. There is a 2x2 pixel block of similar color and a 3x3 pixel block of similar color. (b) Similarity graph for the input image. We can see the crossing edges for the 2x2 pixel block of similar color that forms the trivial case. The 3x3 pixel block of similar color has 4 blocks of these blocks. (c) Similarity graph with all crossing edges removed for the trivial case.)

As crossing edges can only be yielded by four nodes, a natural solution is to treat groups of nodes in blocks of 2x2 in a integrated way. However, care must be taken if one considers an approach based on many threads. A single thread cannot be in charge of modifying the connectivity of these four nodes because the neighbor nodes of each link will be processed at the same time. To guarantee the independence of the node modification process we need to evaluate the heuristics using a node by node processing. The thread will process the current node by reading its neighborhood data and change only the current node links status. Figure 3.7 shows how this happens in the trivial case.

There are others cases of crossing edges that need to be treated by heuristics to make sure that the final results represents the right connectivity. In this work we use the curve

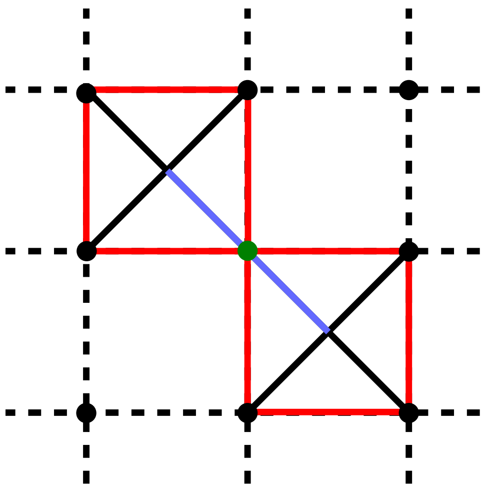


Figure 3.7: Removing trivial crossing edges from the graph. In this case we are processing the center node (green dot). We look for each of the four blocks of 2x2 nodes block in its neighborhood, including the center node. If neighbor nodes are connected by the red edges this means we can unassign the links represented by the blue half edge.

and island heuristics pointed by Kopf and Lischinski in [Kopf and Lischinski 2011].

The curve heuristic execution is based on traversing the nodes of the graph. We need to measure the length of the paths composed of valence-2 nodes to solve possible ambiguities and finally decide which edge to remove in an ambiguous crossing edges configuration. However, this is a process that is not straightforwardly translated into a code for the GPU's parallel architecture. Our solution is to restrain the area around in which the heuristic will be evaluated, that is, considering a node local neighborhood. In other words, the execution of the traversal of the valence-2 nodes is distance limited. Although this seems to be limiting, in practical cases the heuristic evaluation rarely needs to traverse long distances and the results produced are quite satisfactory.

The parallel algorithm for the removal of crossing edges in the ambiguous case, when we have to decide which curve to preserve, is made in a way that is similar to the trivial case of fully connected 2x2 blocks of nodes. We analyse blocks of 2x2 neighbor nodes in a 3x3 nodes block window and modify the connectivity of only one node per thread. But in this case we need an entire copy of the similarity graph after the previous step (the one that solves the trivial case). This is necessary because in this heuristic we need to walk through nodes to find the longest curve and if we find a node already processed during the traversal, this may be the result of topological changes previously done. In such cases the topological combinatorial graph information would be inconsistent justifying the necessity of maintaining a copy of the previous configuration.

When solving the heuristics with a parallel algorithm another issue arises from the fact that some of the four 2x2 node blocks in the 3x3 node block may contain crossing edges not solved yet. If none of the nodes forming the crossing edge is of valence-2 we cannot evaluate the size of valence-2 paths. This happens because none of these 2x2 node blocks with crossing edges were not evaluated yet in this step. In a non-parallel approach, using a specific order for the heuristic evaluation, for instance, the scanline order (each node is relative to a pixel), this issue is less troublesome because the nodes below the current processing line were already evaluated, thus a sequence of valence-2 path would be already chosen. Figure 3.8 shows an example of this issue.

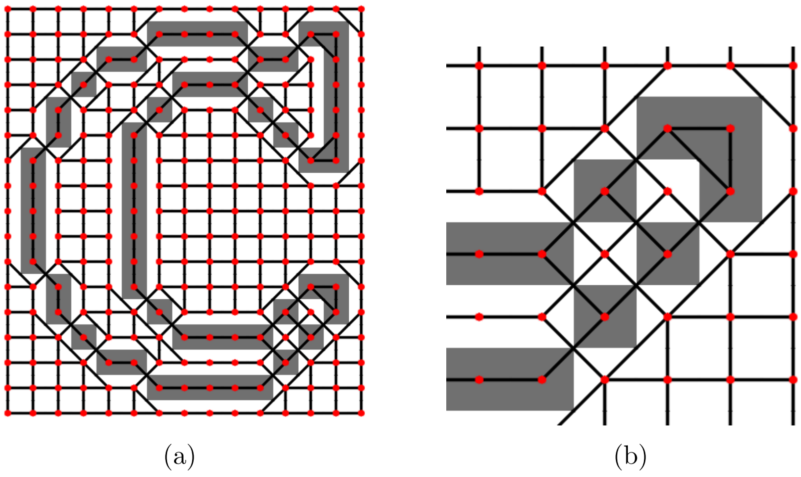


Figure 3.8: Curve heuristic issue example. (a) Input image with similarity graph superimposed. The similarity graph still have ambiguous crossing edges to solve. (b) Zoomed area showing nodes surrounded by 2x2 blocks of crossing edges. Figure 3.9 shows two specifics examples.

To solve this issue we can recursively evaluate the eight neighbors nodes of the central node in the 3x3 nodes block until one of the 2x2 nodes block satisfies the the heuristic evaluation proceed. If we have an image input that looks like an chess board, there will be a big area of ambiguous crossing edges. In such case, we would need many recursion steps to evaluate internal nodes and, therefore, lose efficiency. However, as big areas containing such patterns have a small chance to appear in usual input images, we can restrain the area to be evaluated in order to guarantee efficiency of the algorithm, that is intended to run in real time. The actual approach used to deal with this issue was the use of small windows running a sequential solution as is described in Chapter 4.

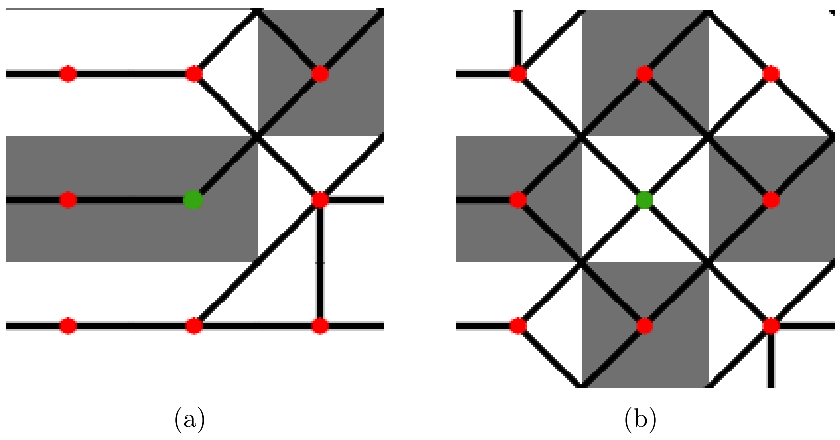


Figure 3.9: Two cases of curve heuristic resolution for the Figure 3.8. (a) The green node being processed has a crossing edge linked to it in the upper-right 2x2 node block. This case can be immediately solved because the crossing edge block has one node that is valence-2 (the green node). (b) The green node being processed has four crossing edge blocks linked to it. None of these crossing edges node blocks can be solved in this step because none of them have a valence-2 node to find which direction have the longest valence-2 path. This happens thanks to the fact that the neighbor nodes were node evaluated yet by the same heuristic in this step.

3.4 Cell Diagram

In order to reshape the pixels, we use the graph nodes to build cells corresponding to the deformation of the pixels according to the node pattern in a way inspired by the Voronoi Diagram [Aurenhammer 1991]. In fact, if we chose to apply the Voronoi approach we would need the Voronoi of line segments, due the fact that the input of this stage is the similarity graph. The exact Voronoi Diagram of line segments would be very costly to compute, affecting the method performance. An approximated solution, which we propose here produces quite good results and is general except for few number of special cases. Moreover, it can be implemented in a full parallel way due to its locality and independence among nodes.

Let the *Cell Diagram* be defined as

$$CD = \{C_0, C_1, C_2, ..., C_{n-1}\} \tag{3.1}$$

where n is the number of nodes in the similarity graph, which is the same number of pixels in the input image. Now let each cell be defined as a set of points

$$C_i = \{p_0, p_1, p_2, ..., p_{m_i-1}\} \tag{3.2}$$

where $i = 0, 1, ..., n-1$ and m_i is the total number of points that vary for each C_i. The cell points' coordinates and number are induced by the graph nodes connections. The process of defining the cell shape will be show in Subsection 3.4.2.

3.4.1 Parallel Design

To fit a parallel solution, each cell is calculated individually in a independent form. One cell is defined by its own set of points, which means these points are not shared between two or more cells. The reason for this redundancy is to maintain the threads independence.

The points are assigned as if each cell has its own coordinate system. The entire diagram is the final result itself, as the cells are to be rendered after being triangulated. The is no curve extraction as a subsequent step. Hence, we do not need to handle the points redundancy problem.

3.4.2 Building

The cell shape is defined by rules that assign vertex positions for each of the eight directions relative to the graph links. The vertex positions are quantized to a quarter of the pixel dimension, such as proposed in [Kopf and Lischinski 2011]. These points will define the shape of the cell as a polygon. This is a simple way to build the cells that can easily be implemented in a parallel solution and greatly enhances the performance of the overall method.

At the end of the new vertex assignment we calculate the convex hull of these points and the result is the cell that reshapes the original pixel. Figure 3.10 shows how the cells building works. An exception exists when there are two adjacent links. In this case we add just one point to the control polygon. This can be seen in Figure 3.11, where the upper-left connected link adds just one point to the final control polygon. Others examples of final cell shapes can be seen in Figure 3.12.

However, the steps shown in Figure 3.10 will fail in cases where the neighboring nodes are not connected together (except the upper-left node). The neighboring nodes need to be evaluated in order to make sure that no gaps will be left between the cells. To deal with this exception we propose to check the up-right and down-right links for the left node and the up-left and down-left links for the right node. If these links exist we know that there is an edge connecting these nodes with the top and bottom node. Figure 3.13 shows how the final control polygon sticks to the square borders when the neighboring nodes are not

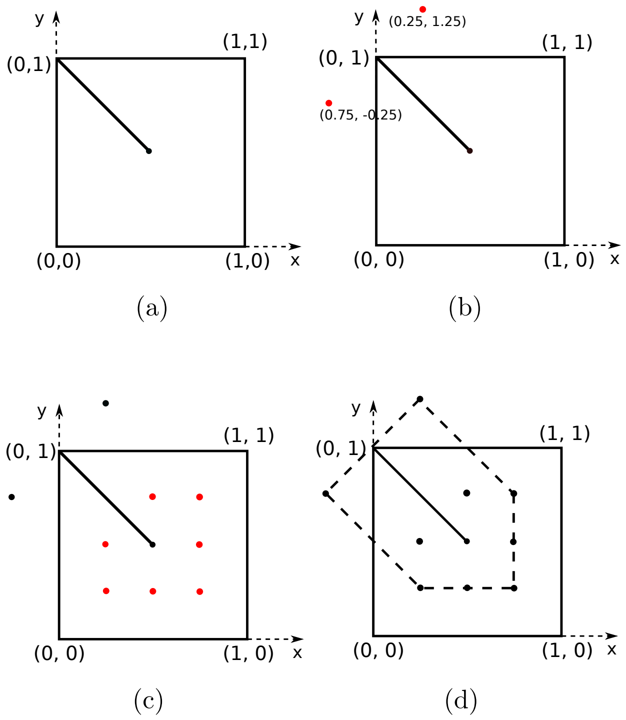


Figure 3.10: Cell building. (a) Node pattern. In this case we have only one link to the upper-left node. (b) For each one of the eight possible directions that have a link we add two points for the polygon that will form the cell. (c) For each direction that does not have a link we add one point half way distant from the center. Except for the upper-left direction, all the others do not have a link. (d) Given all this points we now take the convex hull that will give us the the cell shape.

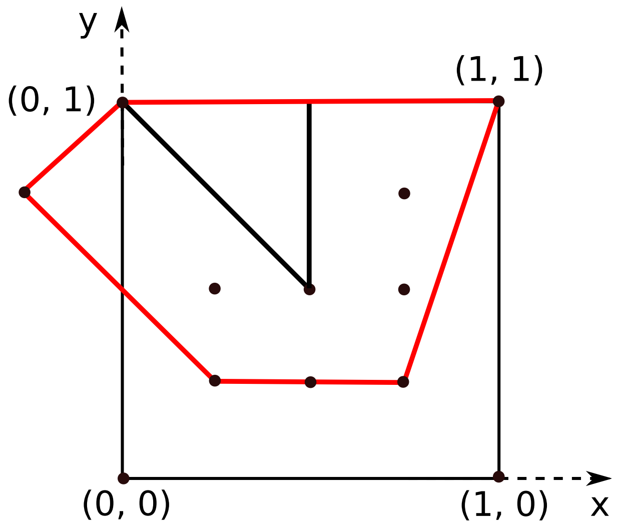


Figure 3.11: Node with two adjacent links. The link to the upper-left node adds just one point to the control polygon that shapes the cell and not two points, like in Figure 3.10.

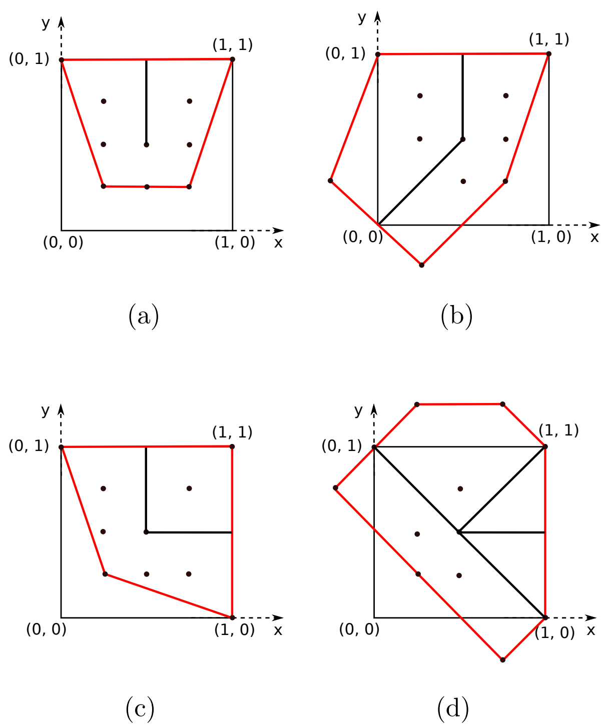


Figure 3.12: Other examples of cell shapes

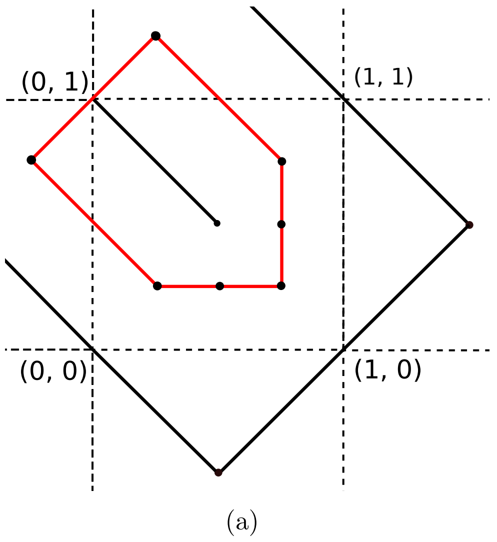
connected together. Figure 3.14 shows how the same node pattern form a different shape of cell for different configuration of neighbor nodes. The entire cell diagram for the part of Alex Kidd input can be seen on Figure 3.15.

The cell contour points assignment steps for the upper-left link of index 0 and the top link of index 1 can be seen as pseudocode on Algorithm 6. The diagonal links of indexes 0, 2, 5 and 7 have similar rules, as well as the right angle links of indexes 1, 3, 4 and 6.

3.5 Smoothing

After the execution of the cells construction stage it is built a much better representation of the original image that solves the discontinuity problem related to diagonal neighbor nodes. At this stage the appearance of the image is not blocky like the initial input obtained by nearest neighbor scale, but the shape of the cells still appears not so smooth, and for greater scales the result will look worse.

The Cell Diagram can still be represented as $CD = \{C_0, C_1, C_2, ..., Cn - 1\}$, but the total number of points for each cell may be modified on this stage. This means that for each cell $C_i = \{p_0, p_1, p_2, ..., p_{m_i - 1}\}$, m_1 will be the same or will increase.



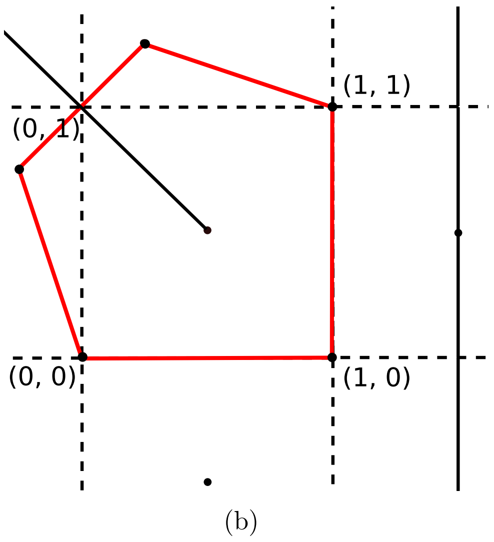


Figure 3.13: Cell building with neighborhood check. (a) Neighbors connected. (b) Neighbors not connected. Note the cell expansion to the square border. The red edges form the polygon shape. The black solid edges are graph edges.

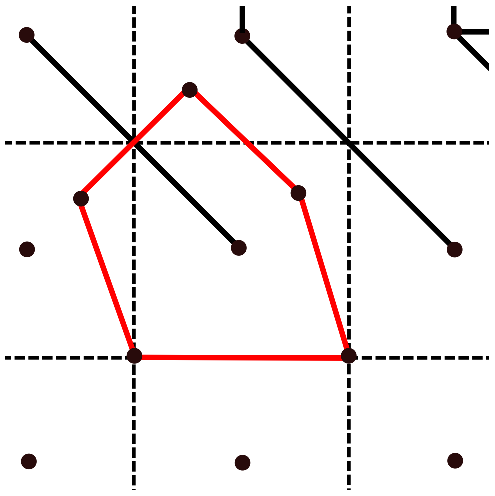


Figure 3.14: Same node pattern as the Figure 3.13, but in this case the right neighbor node is connected to the top node. This make the up-right point to be displaced half-way to the center. The red edges form the polygon shape.

3.5.1 Chaikin's Method

Kopf and Lischinski proposes in [Kopf and Lischinski 2011] to make these cells look smoother by building splines using the cells points as control points. Again, this solution is not local, thus do not have a well suitable implementation in a parallel architecture. To solve this problem in a parallel friendly solution we use a local solution based on the Chaikin's Method of curve subdivision. The Chaikin's Method has been shown to be equivalent to a quadratic b-spline curve [Riesenfeld 1975]. This algorithm is applied on most cells by a GPU thread and we just need data from the neighbors' cells and to know

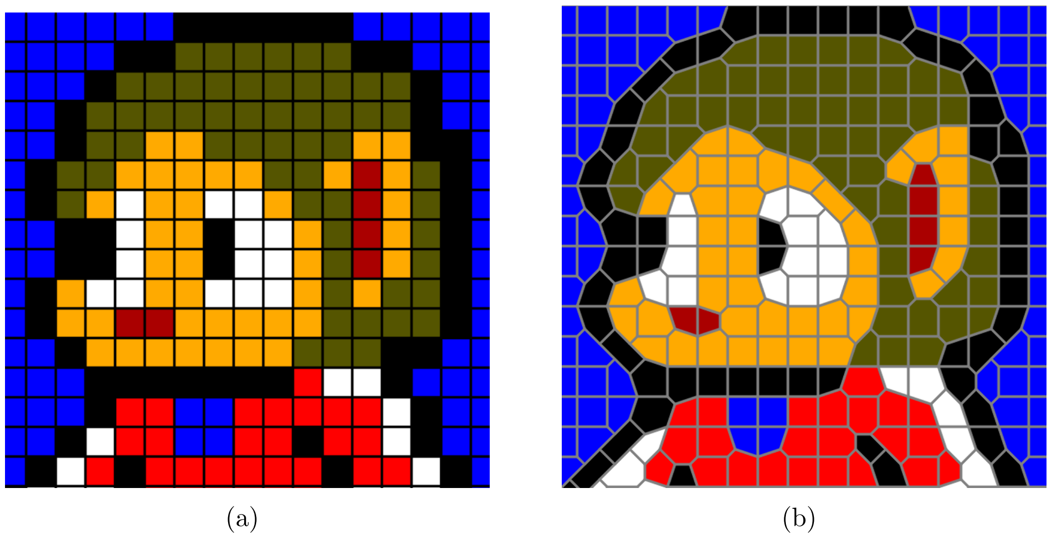


Figure 3.15: Cell diagram result. (a) Input image. (b) Cell diagram. Each cell represented by its relative pixel color.

the cell edges that are not borders, which are the ones that are not crossed by a graph edge. The only cells that do not need to be treated are the ones formed by internal nodes, because they do not represent borders.

The Chaikin's algorithm will be applied on some border edges of the cells by inserting new points that will guide the cutting of the corners. Consider one cell to be defined as a set of points $\{P_0, P_1, ..., P_n\}$. We refine this polygon by generating a new sequence of points $\{Q_0, R_0, Q_1, R_1, ..., Q_{n-1}, R_{n-1}\}$ where the positions of Q_i, R_i are defined by the subdivision rule in Equation 3.3.

$$\begin{aligned} Q_i &= \tfrac{3}{4}P_i + \tfrac{1}{4}P_{i+1} \\ R_i &= \tfrac{1}{4}P_i + \tfrac{3}{4}P_{i+1} \end{aligned} \tag{3.3}$$

Such subdivision results in a new polygon that has two times the number of points of the original, as can be seen in Figure 3.16. However, for this application, we do not want to refine the edges that are adjacent to a neighbor cell of a similar color. Hence, this exception makes the number of points in the new polygon to vary. If the edge P_i, P_{i+1} is adjacent to a similar color cell, the P_{i-1}, P_i and P_{i+1}, P_{i+2} edges will have to connect its new point with an edge of this neighbor cell, more specifically the one that would be considered adjacent by walking on the border edges. Figure 3.17 shows the new edges as a red line.

To proceed to the next stage we need an entire copy of the cell diagram as the neighborhood information is needed to subdivide each cell. This happens because one

Algorithm 6 Cell contour points assignment

```
{Check link of index 0}
if Node.link[0] is assigned then
  if Node.link[3] is not assigned & Node.link[1] is assigned then
    cell.push $(-0.25, 0.75)$
  else if Node.link[3] is assigned & Node.link[1] is not assigned then
    cell.push $(0.25, 1.25)$
  else
    cell.push $(-0.25, 0.75)$
    cell.push $(0.25, 1.25)$
  end if
else
  if Node.link[2] is assigned then
    cell.push $(0.25, 0.75)$
  else
    cell.push $(0.0, 1.0)$
  end if
end if

{Check link of index 1}
if Node.link[1] is assigned then
  cell.push $(0.0, 1.0)$
  cell.push $(1.0, 1.0)$
else
  cell.push $(0.5, 0.75)$
end if

{Check others six links}
(...)

ConvexHull(cell)
```

thread could take a neighbor cell already processed by other thread and the result would be inconsistent.

3.5.2 Especial Subdivision Case

A small number of types of cells, those that have long diagonal edges, need to be treated as a special case because such edges are adjacent to two neighbor cell edges. In such cases the weights used in Equation 3.3 are changed to 1/8 and 7/8. This is equivalent to dividing these edges in two halves and applying the same original subdivision rule. If such change do not be applied it would lead to an inconsistent result caused by the overlapping of pieces of the neighbor cells. This case can be seen in Figure 3.18.

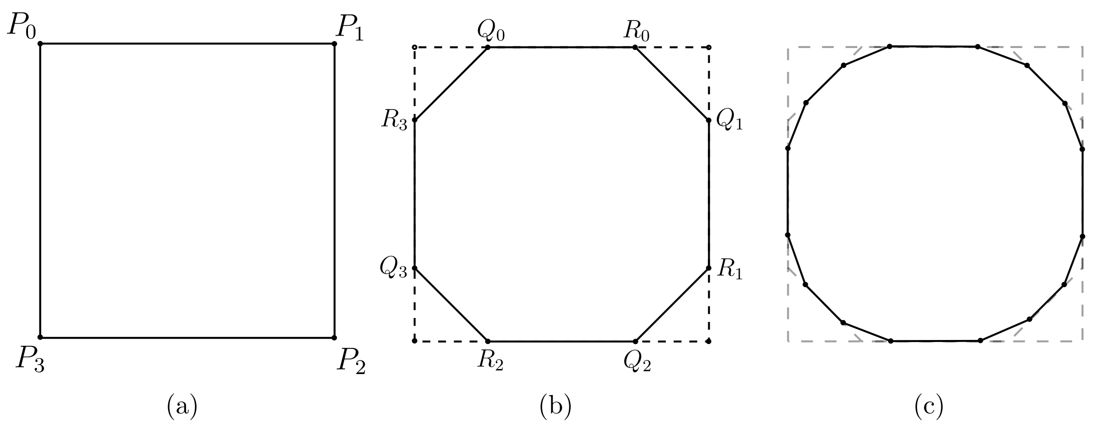


Figure 3.16: Example of Chaikin's method application. a) iteration 0 (4 points) b) iteration 1 (8 points) c) iteration 2 (16 points).

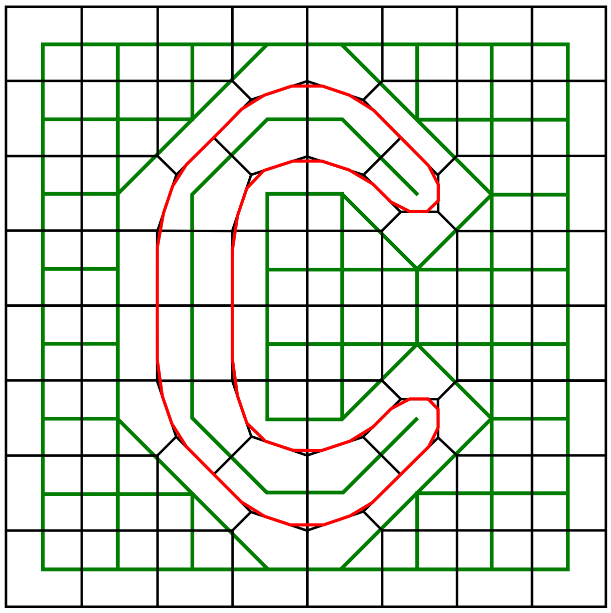


Figure 3.17: Cells subdivision. The red edges are the new ones generated by Chaikin method in one iteration. Note that the cell edges that are adjacent to a similar color cell are not processed.

3.5.3 Cross Junction Case

To prevent holes among the cells, another special case has to be considered when we have cross junction, which consists in three or four dissimilar colors in a 2x2 pixel block. Internal points of the cross junction must not be displaced. In such cases we hinder the subdivision of edges incident to the internal point in the cross junction. The cross junction detection is shown in Figure 3.19.

A few iterations of the Chaikin's Method is enough to yield a better result and the number of subdivisions can be set according to the scale given to the image. For the

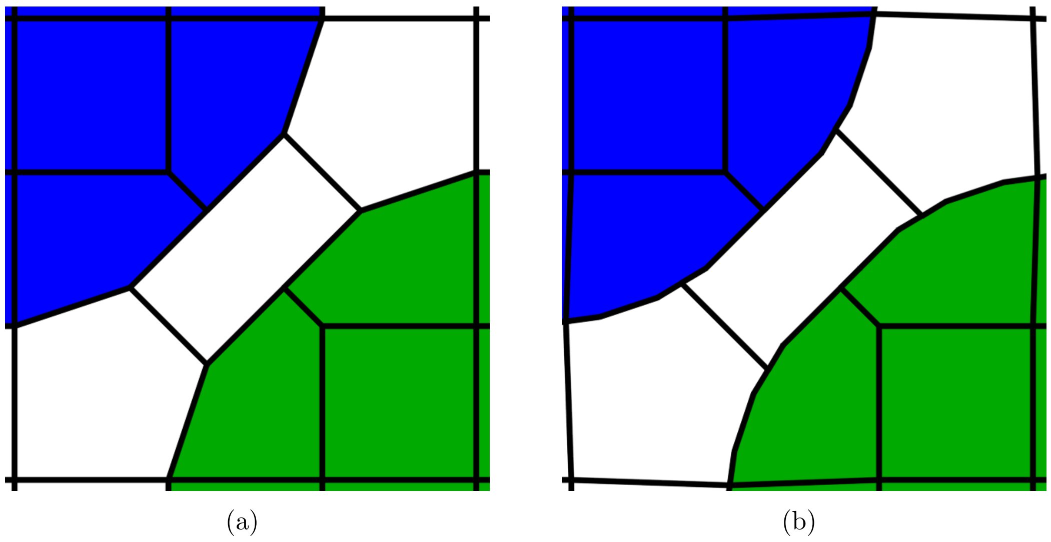


Figure 3.18: Diagonal edge exception (a) The diagonal cell has two of its edges adjacent to other two edges of neighboring cells. (b) The subdivision of those diagonal edges will be done using a similar rule to Equation 3.3 changing the original weights to 1/8 instead of 1/4 and 7/8 instead of 3/4.

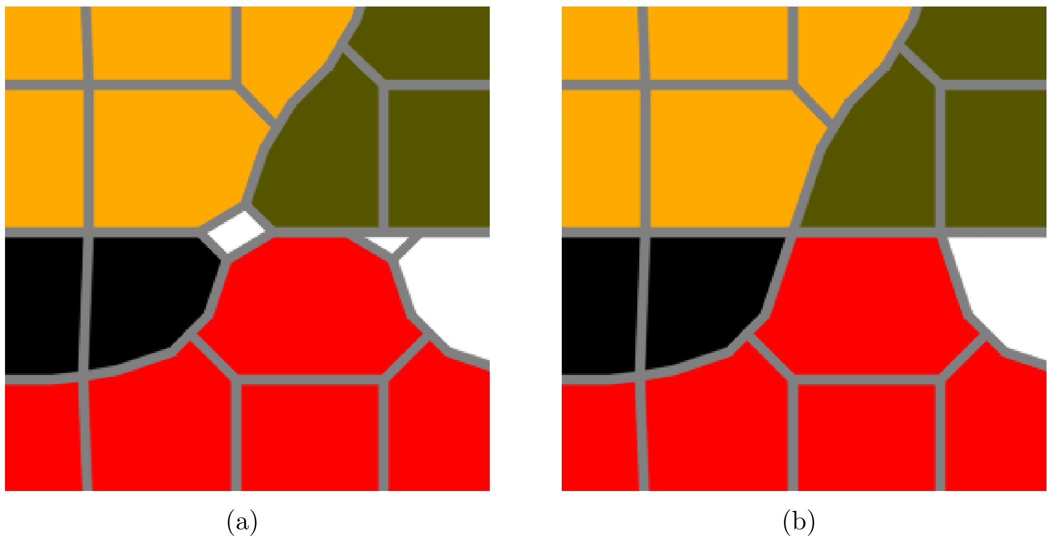


Figure 3.19: Cross junction detection in part of the Alex Kidd sprite. (a) Subdivision without cross junction detection. The blocks of 2x2 cells with three or four dissimilar colors create a blank hole. (b) Subdivision with cross junction detection.

results of this work we used just one iteration in order to maintain performance.

3.6 Triangulation and Rendering

As the entire method is based on the building of the cells, the rendering of the output can be entirely based on the triangulation of these polygons. This fact leads to a high performance rendering, since triangles are the basic primitive of rendering graphics systems. Again, each cell will be processed by a GPU thread and the triangulation process guarantees that the polygons will be rendered with the proper shape. The triangulation of the cells needs to handle concave polygons that can be generated before the smoothing stage. The algorithm used in this work can be seen in [Ratcliff 2000].

As a consequence of the subdivision stage, the smoother cells are represented by more triangles. More triangles result in a greater amount of memory usage and less rendering performance. Figure 3.20 shows a cell diagram already processed by the subdivision stage. The triangle points are written back to the same memory space used by the diagram cell and subdivision stages. In this stage the triangle points are finally assigned to its relative cell diagram position. The triangle array is directly rendered as a OpenGL VBO with multisample anti-aliasing of 4x. CUDA interoperability with OpenGL allows the display without additional data transfer overhead.

To calculate the array size that will be need to hold the triangles vertices for the entire Cell Diagram we need to know the total number of points that defines the cell with higher number of points. Let $|C_{max}|$ be the cell defined by the highest number of points, the array size needed is

$$ArraySize = 3 \times |C_{max}| \times n \tag{3.4}$$

where n is the number of cells on the diagram.

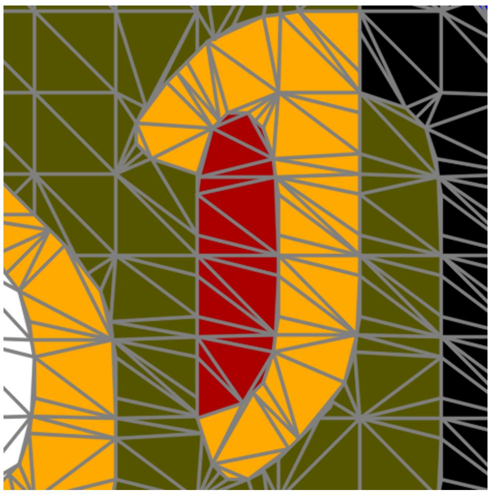


Figure 3.20: Triangulated Cells

Chapter 4

Results

Our method can be applied to full video game frames based on pixel art and keeps a frame rate good enough to achieve a real time response. Using a computer with a 3.0GHz CPU and a GPU GTX 580 Geforce with 512 CUDA cores a full frame of a SNES game (256x224 pixels) can achieve an average of 49 fps with every frame being totally recomputed from the start to the end of the method's pipeline. A table with an average result for 10 inputs of each console is shown in Table 4.1

The difference between each input instance does not affect significantly the computation time of each frame, but the image size does affect. A single sprite of 24x24 pixels such as the one used for Alex Kidd character can be computed in 0.58ms, but the entire frame, which has 256x192 pixels, takes approximately 9.7ms for the machine cited previously. In comparison, the average results of [Kopf and Lischinski 2011] for small sprites, not full frames, take 0,08s without the spline optimization. A comparison with real time methods can be seen in Figure 4.1 and another comparison with a result example from the Depixelizing Pixel Art article can be seen in Figure 4.2. When tested to produce a video output using consecutive frames dumped from a emulator the method presented temporal consistency, allowing coherence on the game animation. Figures 4.5, 4.7, 4.6 and 4.8 shows more full frame results.

The contour of a single sprite is better processed as a single input, in other words, with no background, except a single dissimilar color. A complex background pattern with dissimilar colors from the sprite border can cause a great number of cross junction cases, which lead to a jaggy sprite contour. This can be seen on 4.3.

To solve the curve heuristic problem cited on 3.3.2 we divided the resolution of that nodes by small sequential graph evaluations (windows) using scanline order. Although

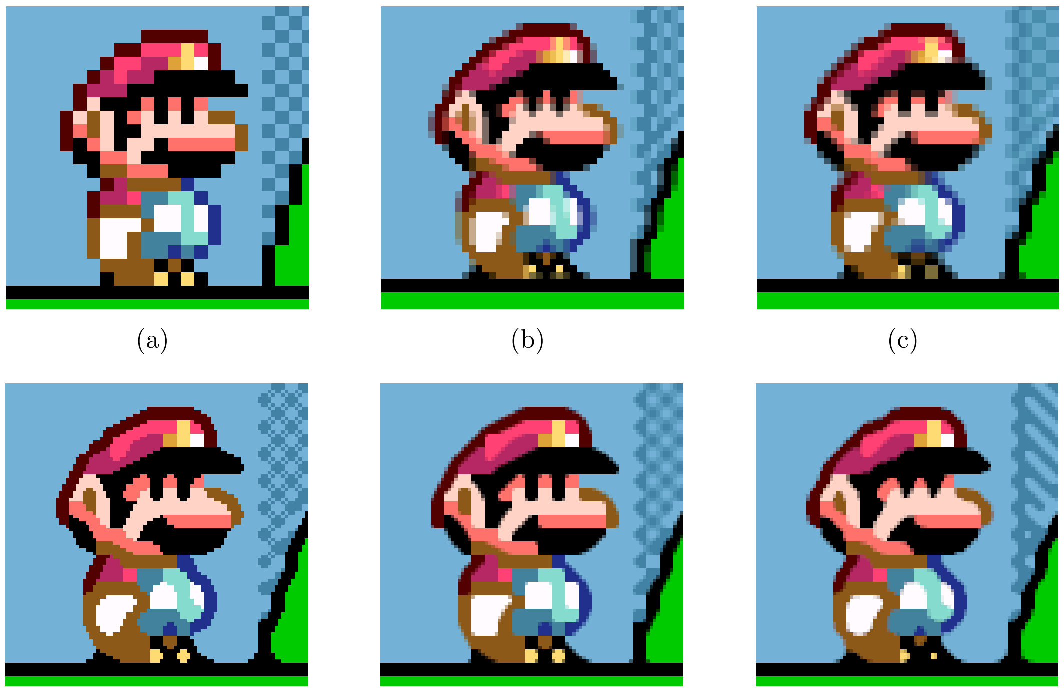


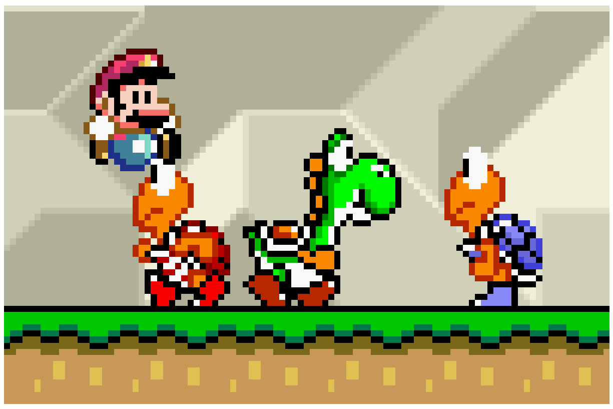
Figure 4.1: Super Mario World partial frame scaled 4 times (a) Nearest Neighbor (b) Super2xSaI (c) SuperEagle (d) Scale4x (e) hq4x (f) our method ©Nintendo Co., Ltd)

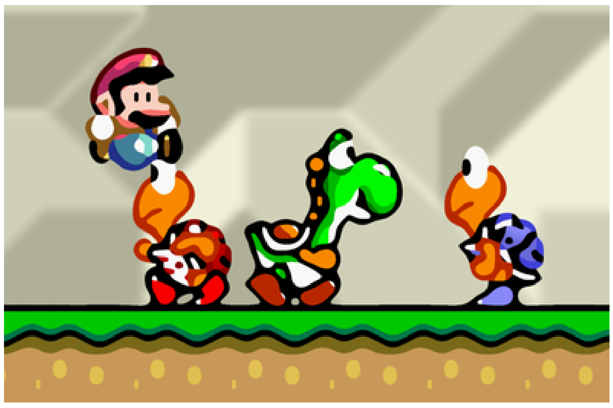
Table 4.1: Average frame per second for each console input. Resolution are expressed in pixels. The output resolution is 1280x720

Console	**Resolution**	**FPS**
Game Boy	160 x 144	104.3
NES	256 x 224	48.6
Master System	256 x 192	56.9
SNES	256 x 224	47.1
Genesis	320 x 224	39.1

improving the results of the final graph, it still leads to a limitation because the window size used to evaluate the graph sequentially may cause miscalculation of the longest path. Figure 4.4 shows the results of this approach. The use of such algorithm causes loss of efficiency and was not used in the final real time results.

A important detail is that all the stages of the implementation benefits from memory coalescence because neighboring threads access neighboring cells in memory. This occurs because of the way image data is stored in memory and the consequent use of the same arrangement for the rest of the data results.


(a)


(b)

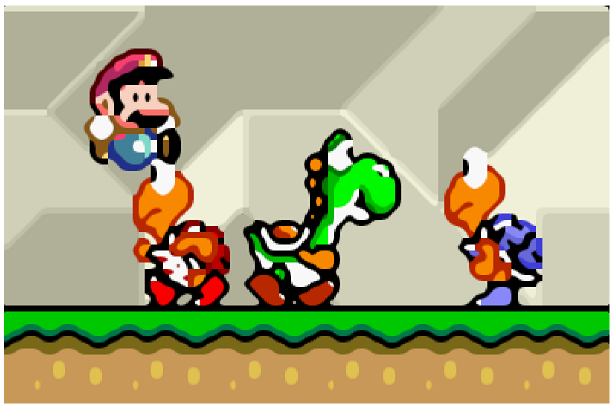

(c)

Figure 4.2: Super Mario World partial frame scaled 4 times (a) nearest neighbor (b) Depixelizing Pixel Art [Kopf and Lischinski 2011] (c) our method (Original image ©Nintendo Co., Ltd)

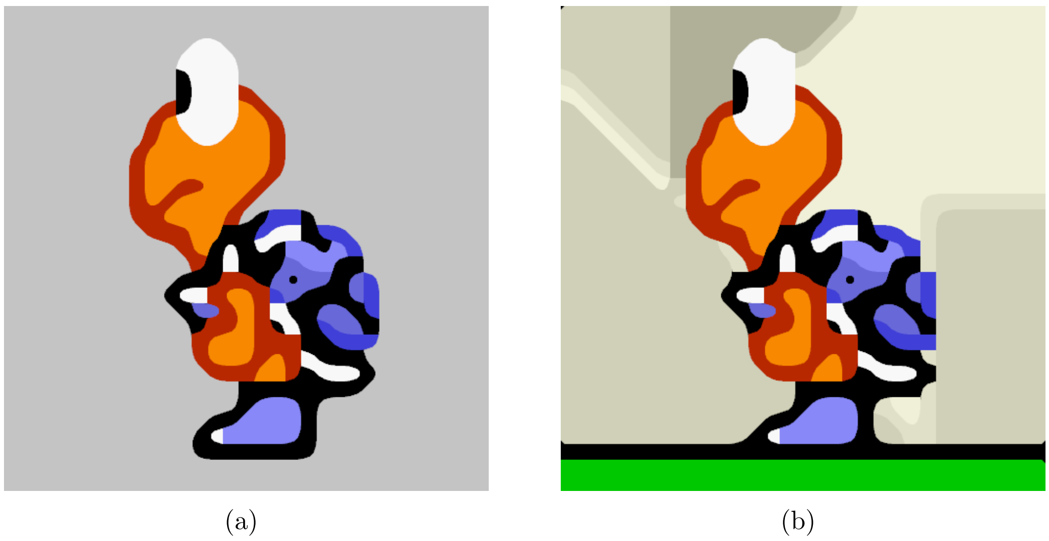


Figure 4.3: Jaggy sprite contour issue (16x scale) (a) Sprite with flat background (b) Sprite with actual game background causing cross junctions on the sprite contour.

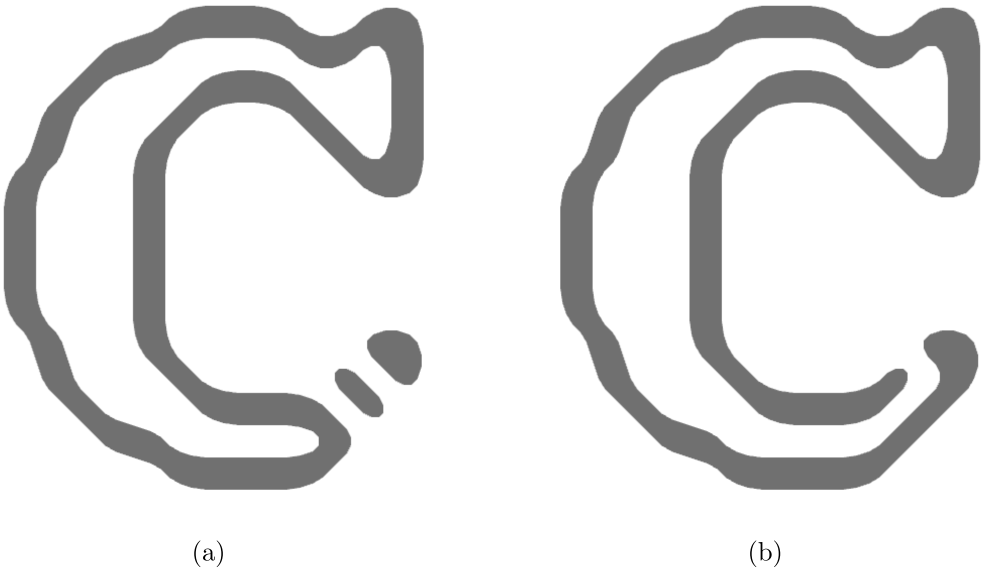


Figure 4.4: Results for the input image of Figure 3.8 (a) Without the window approach (b) Using the window approach.

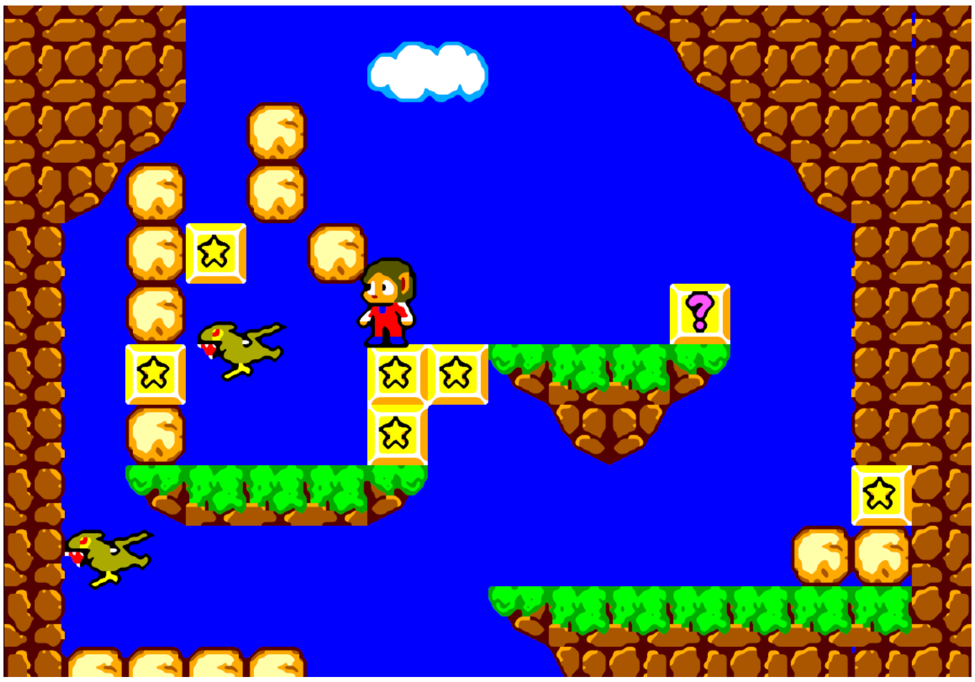


Figure 4.5: Alex Kidd in Miracle World (Master System) full processed frame with average 55 fps (Original image ©Sega Corporation).

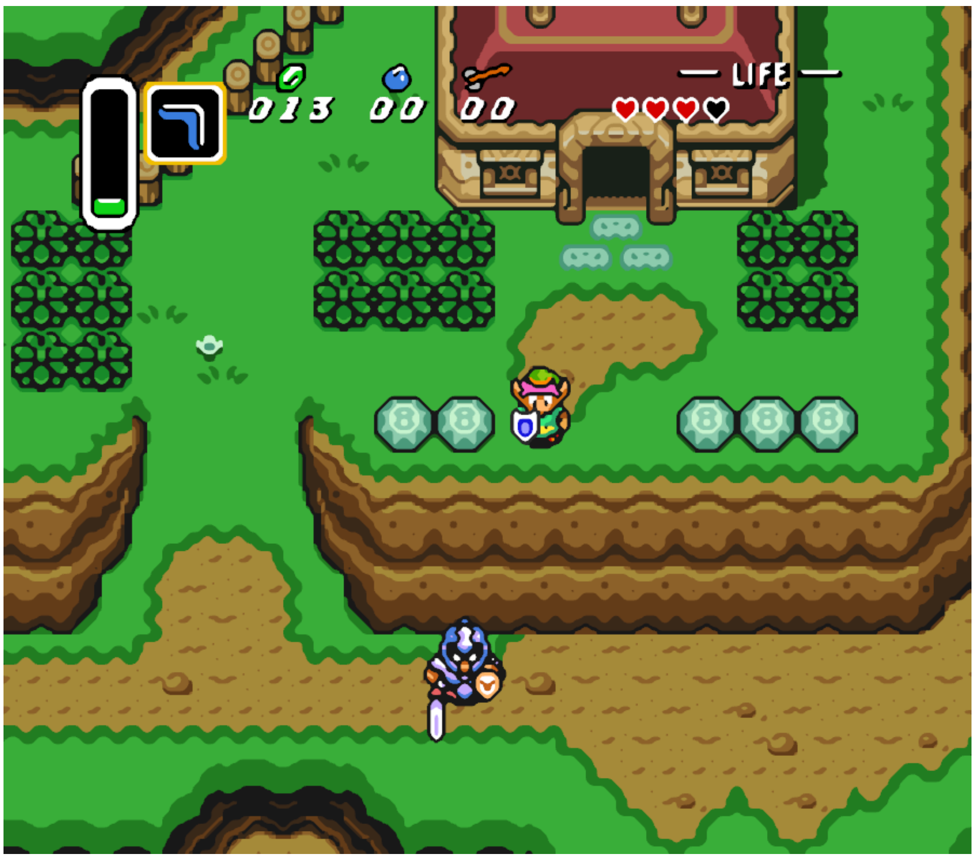
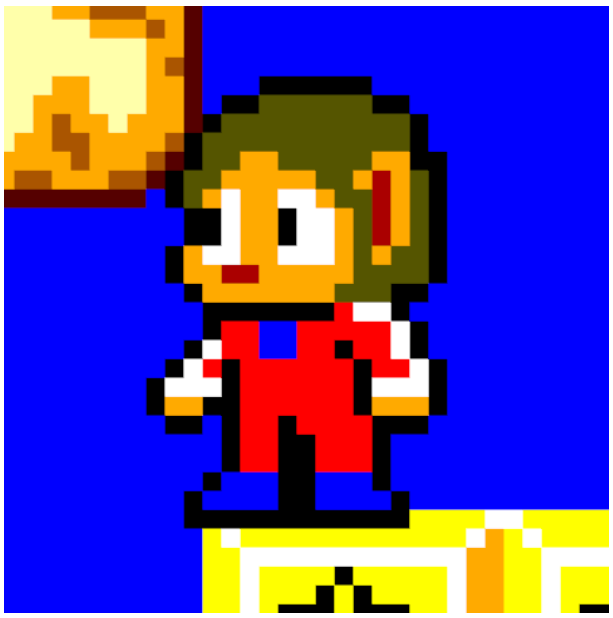
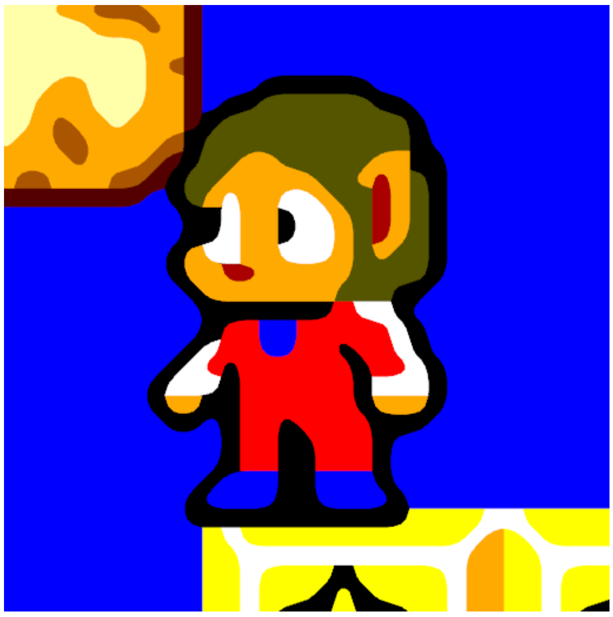


Figure 4.6: The Legend of Zelda: A Link to the Past (SNES) processed frame with average 49 fps (Original image ©Nintendo Co., Ltd).

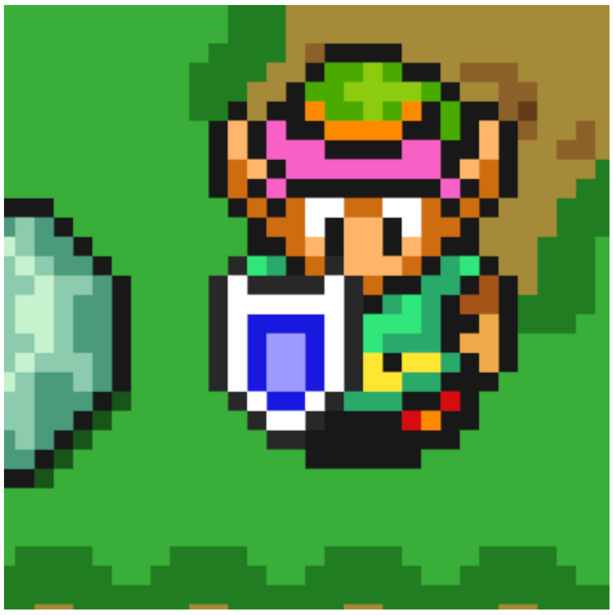


(a)

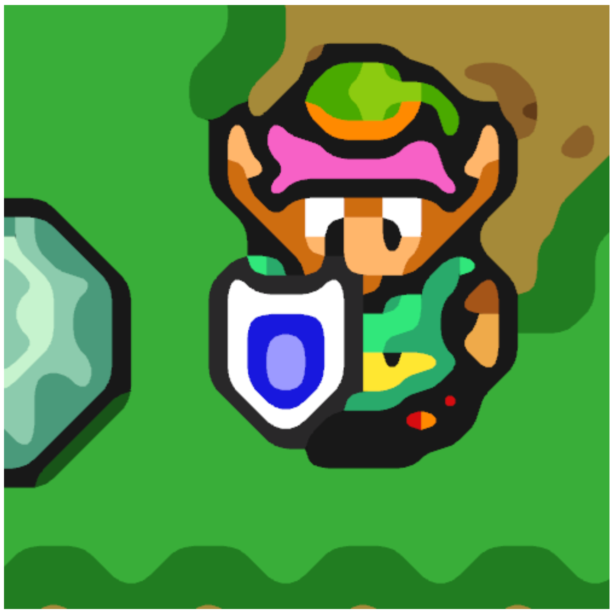


(b)

Figure 4.7: Alex Kidd in Miracle World frame (Master System) scaled 16 times (a) nearest neighbor (b) our method (Original image ©Sega Corporation).



(a)



(b)

Figure 4.8: The Legend of Zelda: A Link to the Past (SNES) frame scaled 16 times (a) nearest neighbor (b) our method (Original image ©Nintendo Co., Ltd).

Chapter 5

Conclusion

As demonstrated, with the power of a massively parallel architecture, such as the GPU, and a local and independent well modelled approach we can create a vector representation of a pixel art image in a efficient way that can led to a real time response for the player, in case of old games input. The visual output will look much smoother than the original when scaled several times.

Our approach based on treating every element of each stage in a parallel solution results in a better performance when compared to [Kopf and Lischinski 2011]. One of the biggest performance boost is obtained thanks to the subdivision method used instead of a b-spline extraction.

Given the parallel nature of the approach, our method do not extract contour borders. The cells are just rendered by the conventional graphical pipeline. This raises a limitation such as the impossibility of using colorization algorithms which depend on contour knowledge. A example of this type of algorithm is given in [Lessa 2011].

The future additions to the presented approach could be an approach to optimize border curves generated by the curve subdivision method. To do this it would be necessary to reduce the problem to a local scope in hope to maintain the efficient flow of the parallel process. Another addition would be the use of a different threshold to detect smooth variance of colors and apply a different method of rendering the cells with a gaussian blur or diffusion colors [Jeschke et al. 2009].

Bibliography

[Aurenhammer 1991] Aurenhammer, F. (1991). Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405.

[Jeschke et al. 2009] Jeschke, S., Cline, D., and Wonka, P. (2009). A gpu laplacian solver for diffusion curves and poisson image editing. *ACM Trans. Graph.*, 28(5):116:1–116:8.

[Kopf and Lischinski 2011] Kopf, J. and Lischinski, D. (2011). Depixelizing pixel art. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011)*, 30(4):99:1 – 99:8.

[Lessa 2011] Lessa, M. K. (2011). Construção e modificação de imagens 2d iluminadas por mapas de normais reconstruídos em tempo de interação. Master's thesis, UFF, Niterói, Brazil.

[Loos 2011] Loos, C. (2011). Vectorization of pixel art. Master's thesis, Universität Augsburg Fakult at fur Angewandte Informatik, Augsburg, Germany.

[Mazzoleni 2001] Mazzoleni, A. (2001). Scale2x/about. Available on `http://scale2x.sourceforge.net/index.html`.

[Mendes 2013] Mendes, B. (2013). Estilo retro em video games - a relação com o jogador. Master's thesis, PUC-Rio, Rio de Janeiro, Brazil.

[Muniz 2012] Muniz, C. E. V. (2012). Extração de malhas poligonais a partir de modelos volumétricos criado por artistas. Master's thesis, UFF, Niterói, Brazil.

[Ratcliff 2000] Ratcliff, J. W. (2000). Efficient polygon triangulation. Available on `http://www.flipcode.com/archives/Efficient_Polygon_Triangulation.shtml`.

[Riesenfeld 1975] Riesenfeld, R. (1975). On chaikins algorithm. *IEEE Computer Graphics and Applications*, 4:304–310.

[Selinger 2003] Selinger, P. (2003). Potrace: a polygon-based tracing algorithm. Available on `http://potrace.sourceforge.net/potrace.pdf`.

[Silva et al. 2013] Silva, M. A. G., Montenegro, A., Clua, E., Vasconcelos, C., and Lage, M. (2013). Real time pixel art remasterizarion on gpus. In Nina Hirata, Luciana Nedel, C. S. K. B., editor, *SIBGRAPI 2013 (XXV Conference on Graphics, Patterns and Images)*, Arequipa, Peru.

[Stepin 2007] Stepin, M. (2007). Hiend3d/demos & docs - hq4x. Available on `http://web.archive.org/web/20070717064839/http://www.hiend3d.com/hq4x.html`.

[Svensson 2013] Svensson, C. (2013). Ask capcom - thread - how long has ducktales been in development? i ask because. Available on `http://www.capcom-unity.com/ask_capcom/go/thread/view/7371/29931937/how-long-has-ducktales-been-in-development-i-ask-because?post_num=17#532330059`.

[Wikipedia 2013] Wikipedia (2013). Pixel art. Available on `http://en.wikipedia.org/wiki/Pixel_art_scaling_algorithms`.