Real Time Pixel Art Remasterization on GPUs

Marco A. G. Silva, Anselmo Montenegro, Esteban Clua, Cristina Vasconcelos, Marcos Lage
Instituto de Computação, Universidade Federal Fluminense (UFF)
CEP 24210-240 Niterói, RJ, Brazil
Email: {marcogs, anselmo, esteban, crisnv, mlage}@ic.uff.br

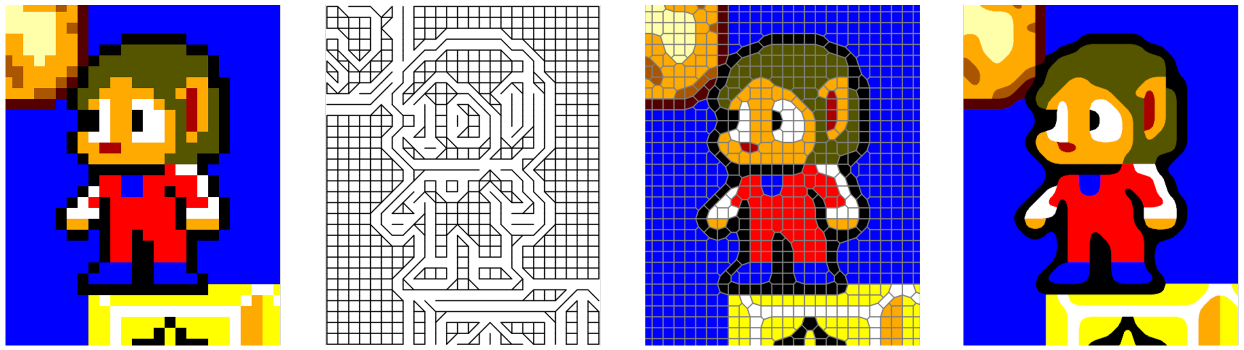


Fig. 1: Example of the stages of our method applied to a pixel art based game. The first image shows the input image scaled 16x with nearest neighbor. The second image represents the similarity graph for the image. The third image shows how pixels are reshaped into cells and the last image is the final result with the smoother triangulated cells (Original image ©Sega Corporation).

***Abstract*—Several methods have been proposed to overcome the pixel art scaling problem through the years. In this article we describe a novel approach to be applied through a massively parallel architecture that can address this issue in real time. To achieve this we design a local and context independent algorithm that enables an efficient parallel implementation on the GPU, delivering full frames output at response time for the user interaction. Our main goal is to apply the method on full frames of old games, which were based on pixel art graphics until the half of the 1990's, and keep the output frame rate good enough for playing.**

***Keywords*-pixel art; upscaling; vectorization; retro;**

I. 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 in 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 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.

Although these games can look outdated if compared to modern consoles titles, mainly in terms of graphics, there is still a large demand for them, thanks to the nostalgia of older gamers and the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 [1] Bernardo Mendes discusses the retro phenomenon for video games and analyse the reasons that led to this phenomenon.

While remixed versions of old classics have been released in the last years, just a limited number of games are receiving this remaster process and, in those cases, too much time and resources are spent to accomplish a high resolution result.

Directly scaling these pixel art with modern video devices results in blocky images, with visual disconnections, an issue that was not supposed to appear in the original representation. The existing vectorization algorithms are supposed to be applied on natural images but its usage 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 hand by hand by the artist. The existing methods designed specially for pixel art upscaling, briefly described in the next section, produce reasonable results, but are fixed to the magnification factor of two, three or four times.

Contributions: In this article we present a novel approach 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

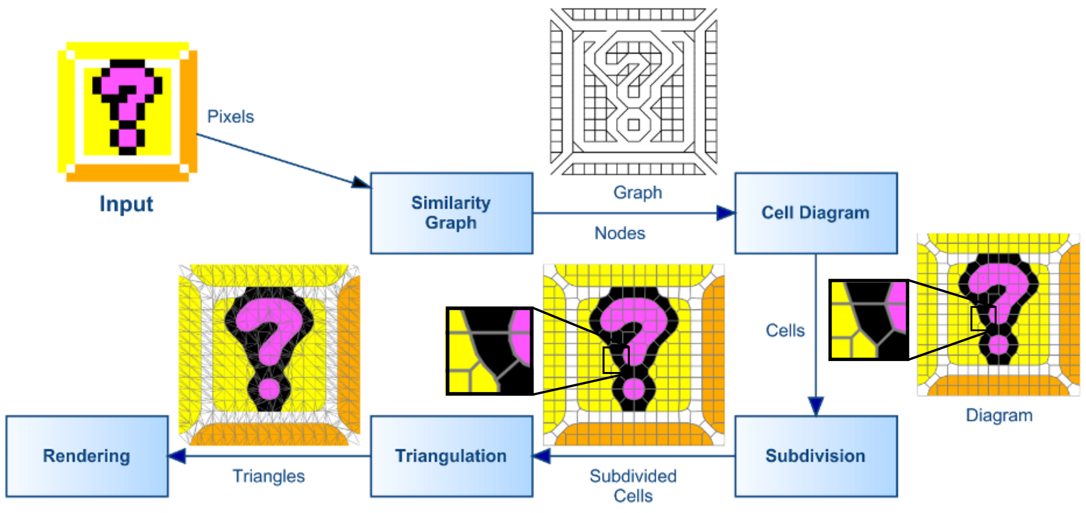


Fig. 2: Pipeline Diagram. The Similarity Graph stage extracts a node for each input pixel. The Cell Diagram stage computes the cell shape for each graph node. In the following the cell border edges are subdivided to give the polygon a smoother aspect. Afterwards, cells are split into triangles that feed the rendering process.

see in Figure 1. The algorithm response time enables the player to play pixel art designed games at any scale factor without the inherent problems of scaling this type of input. To achieve this we propose a new algorithm that is able to use the power of a massively parallel architecture, more specifically, the GPU.

II. Related work

Many methods have been proposed by the emulation community in order to make the old games graphics be better displayed on modern video devices. Some emulators started implementing naive approaches like interlaced black lines (usually called *scanlines*) and linear interpolation. 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.

In 1992, LucasArts created the algorithm called EPX to port its games to a higher resolution platform [2]. Later, some versions of this algorithm appeared on emulators like Scale2x, which produces an output with double of the input resolution. Scale4x is just Scale2x applied twice [3].

In another approach, the 2xSaI algorithm 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. There is also the Eagle algorithm that creates blocks of 2x2 pixels by comparing the neighbors of a pixel in a 3x3 block. Super2xSaI and SuperEagle do more blending [2].

The hqx family of algorithms [4] is known to yield the best results among 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 the hq4x version has 256 entries and returns a pixel block pattern that aims to smooth the final image.

These approaches can produce reasonable results, but they are resolution dependent, and present poor results when displayed on larger resolution such as Full HD (1080p) devices. On the other hand, the 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. A method to extract polygonal surfaces from volumetric models based on voxels is described by Muniz on [5]. 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 [6]. This method produces a vectorized image formed by B-spline curves and color diffusion. Although it produces great results, the global nature of the algorithm makes it very time consuming. On [7] Loos reproduces Kopf and Lischinski's work and gives a detailed explanation of the implementation, which includes some steps that are not described in details in the based work. Our method takes the best of Depixilizing Pixel Art algorithm and proposes a novel parallel method for a full GPU solution.

III. Method

The method proposed in this work is inspired by the Kopf and Lischinski work [6], named Depixelizing Pixel Art. Their approach was proposed to extract features at the pixel level, making every pixel relevant. While the authors 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 to achieve interactive frame rates as it is now.

With the purpose of processing a pixel art game input in real time, this method was adapted and modified to fit a massively parallel architecture, more specifically, the GPU. Much of the steps can be solved locally or adapted to a local approach and they can be interpreted as a pipeline, whose stages will be explained in the next subsections.

Figure 2 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 (subsection III-A) 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 (subsection III-B). 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 (subsection III-C) takes these cells and subdivides their borders using a curve subdivision scheme returning a smoother polygon. On the final stage (subsection III-D) the cells are split into triangles and finally rendered.

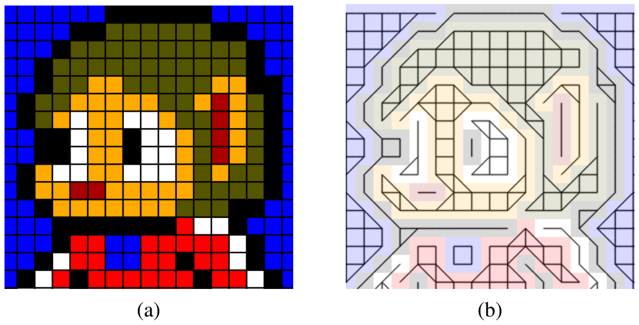


Fig. 3: Similarity Graph extraction. (a) Input Image. (b) Similarity Graph (the black line represents the node's edges).

A. 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 $n_0, n_1, \ldots, n_n$ 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 considered similar according to a similarity threshold. The threshold used is a variation 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. An example of a similarity graph can be seen on Figure 3.

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.

Our method creates one node per pixel of the input image. The connectivity relation based on similarity can be computed in parallel using several threads run independently. The independence of threads happens because their actions are limited to comparisons between the eight neighbors of the pixel and to the storage of the links related with 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.

After the first step we have a graph with crossing edges that must be removed in order to solve ambiguities. 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.

As crossing edges can only be yielded by four nodes, a natural solution is to treat groups of nodes in blocks of 2x2 in an 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 5 shows how this happens in the trivial case.

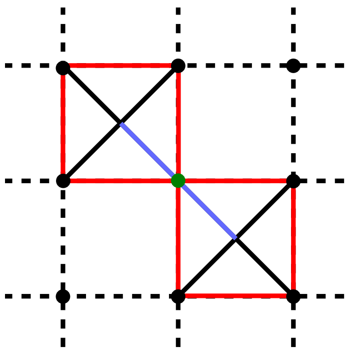


Fig. 5: Removing trivial crossing edges from the graph. In this case we are processing the center node (green dot). We look for each of the four blocks of 2x2 nodes in its neighborhood, including it. If neighbor nodes are connected by the red edges this means we can unassign the links represented by the blue half edge.

There are others cases of crossing edges that need to be treated by heuristics to make sure that the final results represent the right connectivity. Here, we use the curve and island heuristics pointed by Kopf and Lischinski at [6].

The 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 a ambiguous

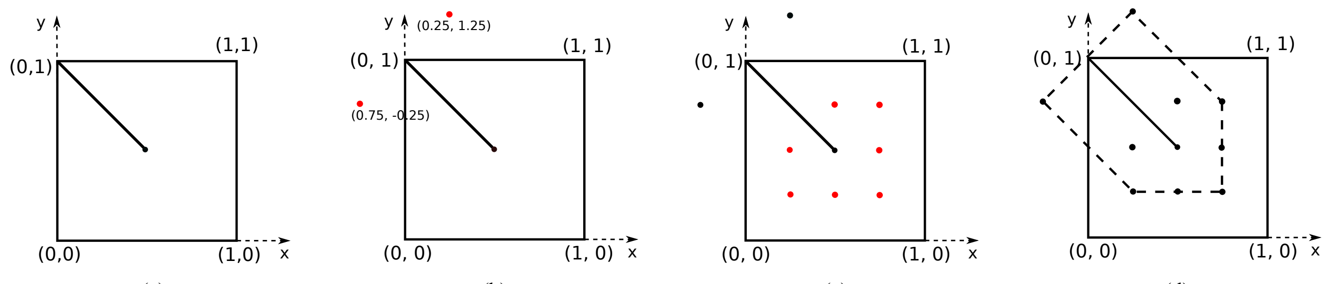


Fig. 4: Cell building. (a) Node pattern. In this case we have only one link to the upper-left node. (b) For each one of the eight possible directions that have a link we add two points for the polygon that will form the cell. (c) For each direction that does not have a link we add one point half way 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.

crossing edges configuration. However, this is a process that is not straightforward translated into a code for the GPU's parallel architecture. Our solution is to restrain the area around which the heuristic will be evaluated, that is, considering a node's 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 similarly to the trivial case of fully connected 2x2 blocks of nodes. We analyze blocks of 2x2 neighbor nodes 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.

B. Cell Building

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 [8]. The exact Voronoi Diagram 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.

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 [6]. These points will define the shape of the cell as a polygon. This is a simple

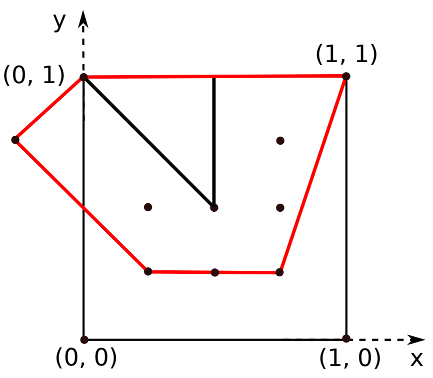
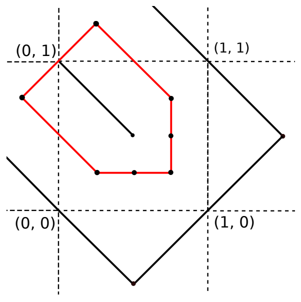


Fig. 6: Node with two adjacent links. The link to the upper-left node adds just one point to the control polygon that shapes the cell and not two points, like in Figure 4.

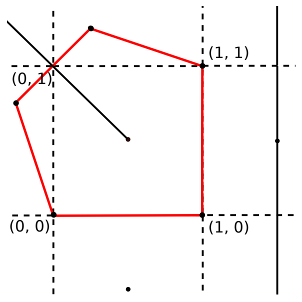
way to build the cells that can easily be implemented in a parallel solution and greatly enhances the performance of the overall method.

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 4 shows how the cells are built. 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 6, where the upper-left connected link adds just one point to the final control polygon.

However, the steps shown in Figure 4 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 7 shows how the final control polygon sticks to the square borders when the neighboring nodes are not connected together. Figure 8 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 9



(a)



(b)

Fig. 7: 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.

C. Smoothing

After the execution of the cells construction stage there is 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 keeps getting worse.

Kopf and Lischinski proposes on [6] to make these cells look smoother 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 [9]. This algorithm is applied on most cells by a GPU thread and we just need data from the neighbors cells and to know 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 represents borders.

The Chaikin's algorithm will be applied on some border edges of the cells by inserting new control 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 control polygon by generating a new sequence of control 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 1.

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

Such subdivision results in a new control polygon that has two times the number of points of the original. However, for

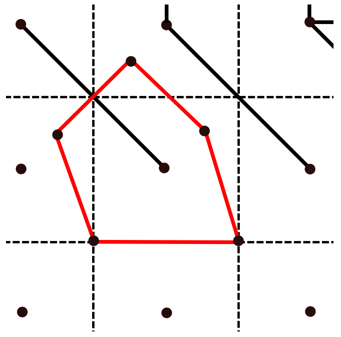


Fig. 8: Same node pattern as the Figure 7, but in this case the right neighbor node is connected to the top node. This make the up-right point to be displaced half-way to the center. The red edges form the polygon shape.

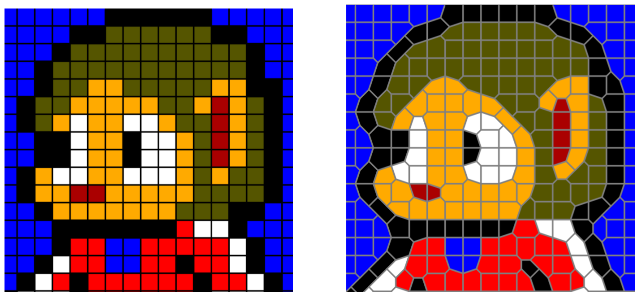

(a) (b)

Fig. 9: Cell diagram result. (a) Input image. (b) Cell diagram. Each cell represented by its relative pixel color.

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 control polygon 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 control point with an edge of this neighbor cell, more specifically the one that would be considered adjacent by walking on the border edges. Figure 10 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 thread could take a neighbor cell already processed by other thread and the result would be inconsistent.

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 1 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. Otherwise it would lead to an inconsistent result caused by the overlapping of pieces of the neighbor cells. This case can be seen in Figure 11

To prevent holes among the cells, another special case has to be considered when we have T-junctions, which consists in

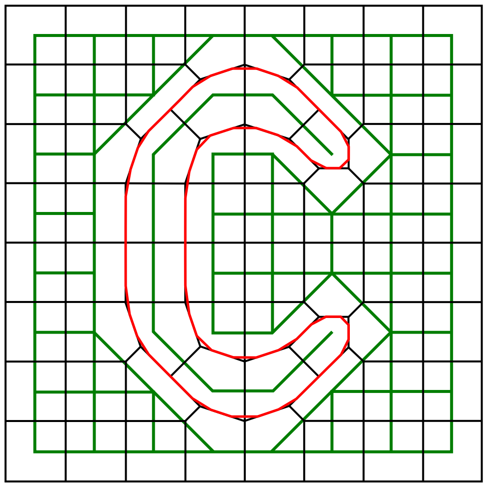


Fig. 10: Cells subdivision. The red edges are the new ones generated by Chaikin method in one iteration. Note that the cell edges that are adjacent to a similar color cell are not processed.

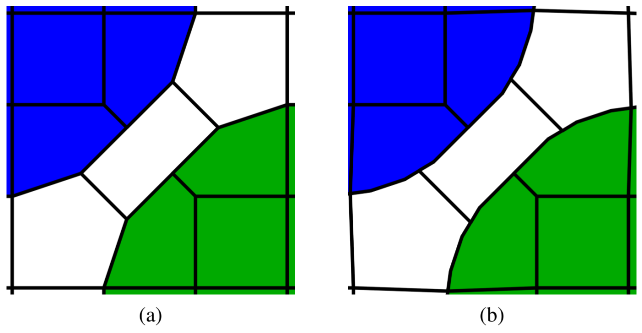


Fig. 11: Diagonal edge exception (a) The diagonal cell has two of its edges adjacent to other two edges of neighboring cells. (b) The subdivision of those diagonal edges will be done using a similar rule to Equation 1 changing the original weights to $1/8$ instead of $1/4$ and $7/8$ instead of $3/4$.

three or four dissimilar colors in a 2x2 pixel block. Internal points of the T-junctions must not be displaced. In such cases we hinder the subdivision of edges incident to the internal point in the T-junction. The T-junction detection is shown in Figure 12.

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

D. 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 graphic systems. Again, each cell will be processed

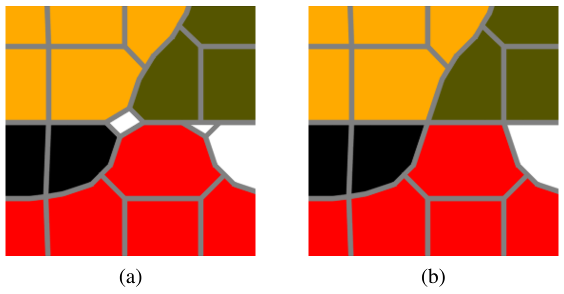


Fig. 12: T-junction detection in part of the Alex Kidd sprite. (a) Subdivision without T-junction detection. The blocks of 2x2 cells with three or four dissimilar colors create a blank hole. (b) Subdivision with T-junction detection.

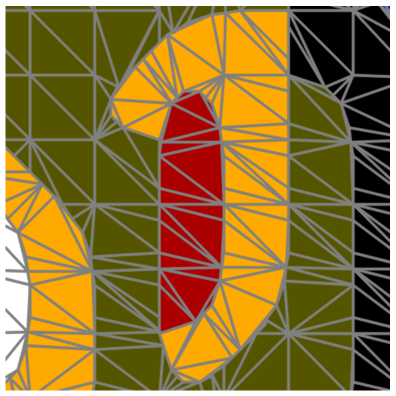


Fig. 13: Triangulated Cells

by a GPU thread and the triangulation process guarantees that the polygons will be rendered with the proper shape.

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 13 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. The triangle array is directly rendered as a OpenGL VBO with multisample anti-aliasing of 4x.

IV. Results

Our method can be applied in 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 I

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,

TABLE I: Average frame per second for each console input. Resolution are expressed in pixels.

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

the average results of [6] for small sprites, not full frames, take 0.08s without the spline optimization. A comparison with real time methods can be seen on 14 and another comparison with a result example from the Depixelizing Pixel Art article can be seen on Figure 15. 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 16, 17, 18 and 19 shows more full frame 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.

V. 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 lead 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 [6]. 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 regions' 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. An example of this type of algorithm is given in [10].

The future additions to the presented proposal could be an approach to optimize border curves generated by the curve subdivision method. To do this it would be necessary a reduction of the problem to a local scope in hope to maintain the efficient flow of the parallel process. Another addition would be the use 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 [11].

Acknowledgment

The authors would like to thank the Foundation for Research Support of Rio de Janeiro State (FAPERJ) for the support.

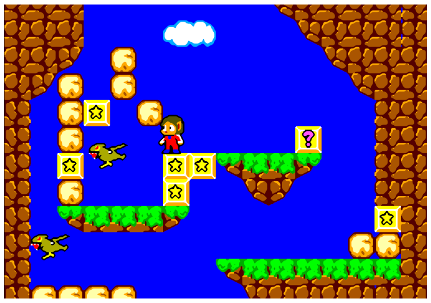


Fig. 16: Alex Kidd in Miracle World (Master System) full processed frame with average 55 fps (Original image ©Sega Corporation)

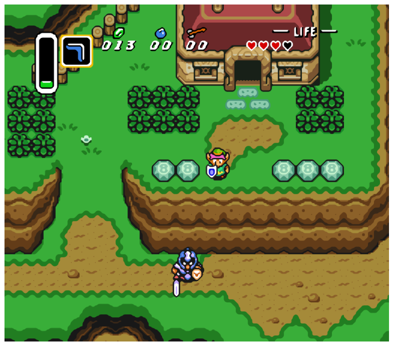


Fig. 18: The Legend of Zelda: A Link to the Past (SNES) processed frame with average 49 fps (Original image ©Nintendo Co., Ltd)

References

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

[2] Wikipedia, "Pixel art," april 2013. [Online]. Available: http://en.wikipedia.org/wiki/Pixel_art_scaling_algorithms

[3] A. Mazzoleni, "Scale2x/about," 2001. [Online]. Available: http://scale2x.sourceforge.net/index.html

[4] M. Stepin, "Hiend3d/demos & docs - hq4x," 2007. [Online]. Available: http://web.archive.org/web/20070717064839/http://www.hiend3d.com/hq4x.html

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

[6] J. Kopf and D. Lischinski, "Depixelizing pixel art," *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011)*, vol. 30, no. 4, pp. 99:1 – 99:8, 2011.

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

[8] F. Aurenhammer, "Voronoi diagrams - a survey of a fundamental geometric data structure," *ACM Comput. Surv.*, vol. 23, no. 3, pp. 345–405, september 1991.

[9] R. Riesenfeld, "On chaikins algorithm," *IEEE Computer Graphics and Applications*, vol. 4, pp. 304–310, september 1975.

[10] M. K. Lessa, "Construção e modificação de imagens 2d iluminadas por mapas de nor mais reconstruídos em tempo de interação," Master's thesis, UFF, Niterói, Brazil, 2011.

[11] S. Jeschke, D. Cline, and P. Wonka, "A gpu laplacian solver for diffusion curves and poisson image editing," *ACM Trans. Graph.*, vol. 28, no. 5, pp. 116:1–116:8, Dec. 2009.

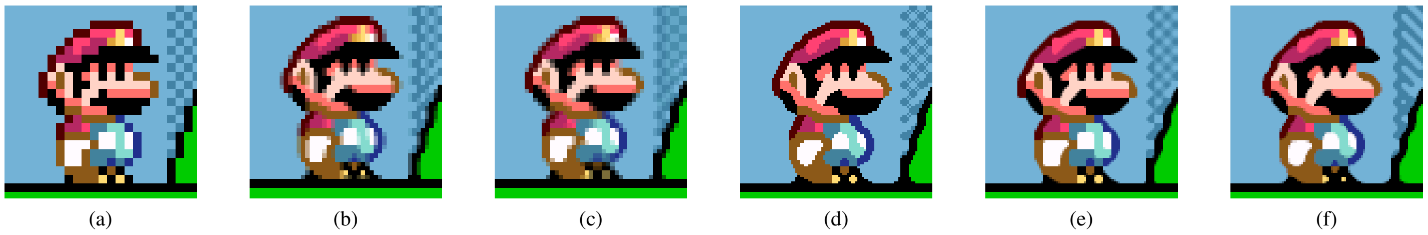


Fig. 14: Super Mario World partial frame scaled 4 times (a) Nearest Neighbor (b) Super2xSaI (c) SuperEagle (d) Scale4x (e) hq4x (f) our method ©Nintendo Co., Ltd)

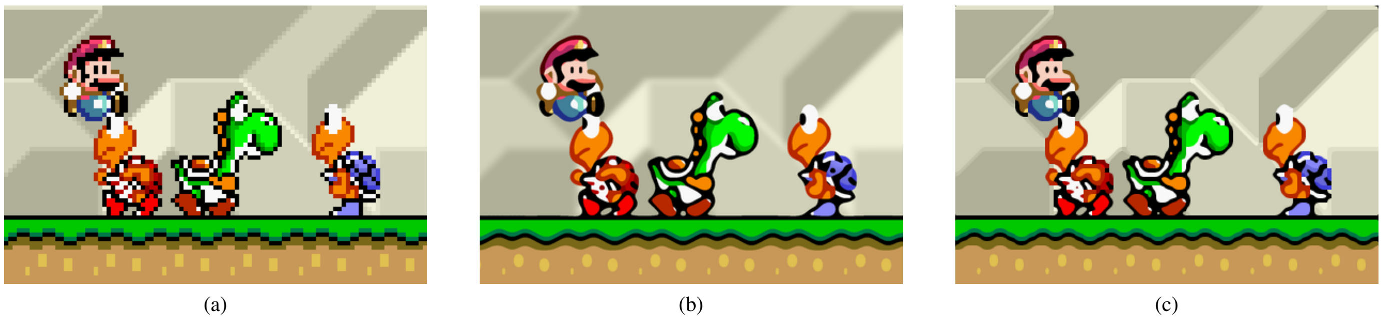


Fig. 15: Super Mario World partial frame scaled 4 times (a) nearest neighbor (b) Depixelizing Pixel Art [6] (c) our method (Original image ©Nintendo Co., Ltd)

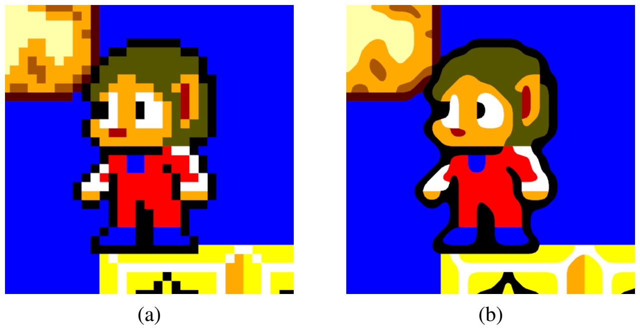


Fig. 17: Alex Kidd in Miracle World frame (Master System) scaled 16 times (a) nearest neighbor (b) our method (Original image ©Sega Corporation)

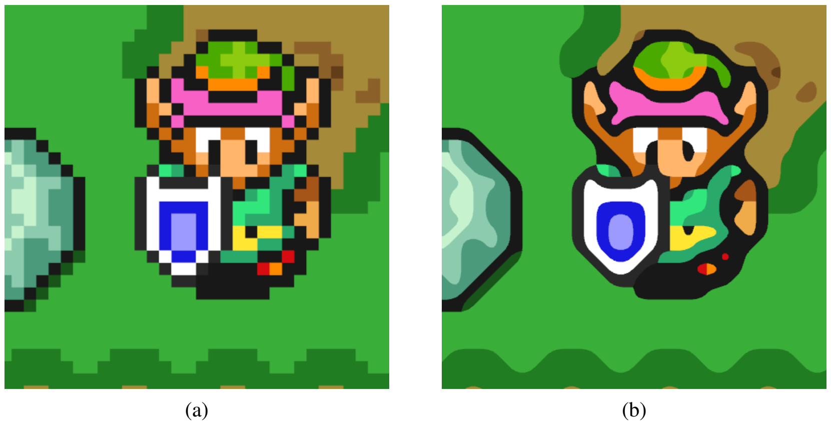


Fig. 19: The Legend of Zelda: A Link to the Past (SNES) frame scaled 16 times (a) nearest neighbor (b) our method (Original image ©Nintendo Co., Ltd)