Referências Bibliográficas

[Ade84] ADELSON, E.; ANDERSON, C.; BERGEN, J.; BURT, P. ; OGDEN, J.. **Pyramid methods in image processing**. RCA Engineer, 29(6):33–41, 1984. 4.1

[Aga04] AGARWALA, A.; DONTCHEVA, M.; AGRAWALA, M.; DRUCKER, S.; COLBURN, A.; CURLESS, B.; SALESIN, D. ; COHEN, M.. **Interactive digital photomontage**. ACM Trans. Graph., 23(3):294–302, 2004. 4.6

[Ble06] BLEYER, M.; RHEMANN, C. ; GELAUTZ, M.. **Segmentation-based motion with occlusions using graph-cut optimization**. p. 465–474. 2006. 4.6

[Ble07] BLEYER, M.; GELAUTZ, M.. **Graph-cut-based stereo matching using image segmentation with symmetrical treatment of occlusions**. Image Commun., 22(2):127–143, 2007. 4.6, 5

[Boy98] BOYKOV, Y.; VEKSLER, O. ; ZABIH, R.. **Efficient restoration of multicolor image with independent noise**. Technical report, Ithaca, NY, USA, 1998. 4.6, 4.7.1

[Boy04] BOYKOV, Y.; KOLMOGOROV, V.. **An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision**. IEEE Transactions on Pattern Analysis and Machine Intelligence, 26:359–374, 2004. 4.6.1, 4.6.1

[Bru07] BRUNNER, R.; DOEPKE, F. ; LADEN, B.. **Object detection by color: Using the gpu for real-time video image processing**. In: Nguyen, H., editor, GPU GEMS 3, chapter 26, p. 563–574. Addison Wesley, July 2007. (document), 3.3.2, 3.3.2, 3.8

[CTM08] **Ati ctm guide. technical reference manual.** Technical report, AMD, 2006. 2, 2.3

[CUDA08] **Cuda programming guide 1.1**. Website. , 2007. (document), 2, 2.1, 2.3, 3.1.2

[Cas08] C., R.; LEWINER, T.; LOPES, H.; TAVARES, G. ; BORDIGNON, A. L.. **Statistical optimization of octree searches.** Computer Graphics Forum, 27(6):1557–1566, 2008. 1

[Cat08] CATANZARO, B.; SUNDARAM, N. ; KEUTZER, K.. **A map reduce framework for programming graphics processors.** In: THIRD WORKSHOP ON SOFTWARE TOOLS FOR MULTICORE SYSTEMS (STMCS 2008) IN CONJUNCTION WITH THE IEEE/ACM CGO., 2008. 3.1

[Che07] CHE, S.; MENG, J. ; SHEAFFER, J. W.. **A performance study of general purpose applications on graphics processors.** In: FIRST WORKSHOP ON GENERAL PURPOSE PROCESSING ON GRAPHICS PROCESSING UNITS, Oct. 2007. 2.3

[Che08] CHE, S.; BOYER, M.; MENG, J.; TARJAN, D.; SHEAFFER, J. W. ; SKADRON, K.. **A performance study of general-purpose applications on graphics processors using cuda.** J. Parallel Distrib. Comput., 68(10):1370–1380, 2008. 2.3

[Fat08] FATAHALIAN, K.; HOUSTON, M.. **A closer look at gpus.** Commun. ACM, 51(10):50–57, 2008. 2.1

[Fer03] FERNANDO, R.; KILGARD, M. J.. **The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.** Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. 2

[Fleck] FLECK, S.; BUSCH, F.; BIBER, P. ; STRAÍER, W.. **Graph cut based panoramic 3d modeling and ground truth comparison with a mobile platform - the wägele.** Image Vision Comput., 27(1-2):141–152, 2009. 4.6

[Flu06] FLUCK, O.; AHARON, S.; CREMERS, D. ; ROUSSON, M.. **Gpu histogram computation.** In: SIGGRAPH '06: ACM SIGGRAPH 2006 RESEARCH POSTERS, p. 53, New York, NY, USA, 2006. ACM. 3.1.2, 5

[For02] FORSYTH, D. A.; PONCE, J.. **Computer Vision: A Modern Approach.** Prentice Hall, 2002. 3.2, 4, 4.1

[Fri02] FRISKEN, S. F.; PERRY, R. N.. **Simple and efficient traversal methods for quadtrees and octrees.** Journal of Graphics Tools, 7(7):2002, 2002. 4.3.2

[GPGPU community] **Gpgpu : General-purpose computation using graphics hardware**. Website. . 2.2

[GPU Gems 2] FERNANDO, R.. **GPU Gems 2 - Programming techniques for High-Performance Graphics and General Purpose Computation**. Addison-Wesley Professional, USA, 2005. 1, 2.2, 2.2, 2.2, 2.4

[GeForce 8800 Technical Brief] **Technical brief. nvidia geforce 8800 gpu architecture overview**. Website. , Nov. 2006. 2.1.1, 2.2

[Gon01] GONZALEZ, R. C.; WOODS, R. E.. **Digital Image Processing**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. 3.2, 4.2, 4.6

[He08] HE, B.; FANG, W.; LUO, Q.; GOVINDARAJU, N. K. ; WANG, T.. **Mars: a mapreduce framework on graphics processors**. In: PACT '08: PROCEEDINGS OF THE 17TH INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, p. 260–269, New York, NY, USA, 2008. ACM. 3.1

[Hen07] HENSLEY, J.. **Close to the metal**. In: SIGGRAPH '07: ACM SIGGRAPH 2007 COURSES, p. –. ACM, 2007. 2.3

[Hil86] HILLIS, W. D.; GUY L. STEELE, J.. **Data parallel algorithms**. Commun. ACM, 29(12):1170–1183, 1986. 3.1

[Hon04] HONG, L.; CHEN, G.. **Segment-based stereo matching using graph cuts**. Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on, 1:I–74–I–81 Vol.1, June-2 July 2004. 5

[Hor05] HORN, D.. **GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation**, chapter Stream reduction operations for GPGPU applications, p. 573–589. Addison Wesley, 2005. 3.1

[Hus07] HUSSEIN, M.; VARSHNEY, A. ; DAVIS, L.. **On implementing graph cuts on cuda**. In: FIRST WORKSHOP ON GENERAL PURPOSE PROCESSING ON GRAPHICS PROCESSING UNITS, Oct. 2007. 4.6

[Knu73] KNUTH, D. E.. **Fundamental Algorithms**, volumen 1 de **The Art of Computer Programming**, section 1.2, p. 10–119. Addison-Wesley, Reading, Massachusetts, second edition, jan 1973. 4.3

[Kol01] KOLMOGOROV, V.; ZABIH, R.. **Computing visual correspondence with occlusions using graph cuts.** In: IN INTERNATIONAL CONFERENCE ON COMPUTER VISION, p. 508–515, 2001. 4.6

[Kot02] KÖTHE, U.. **Deriving topological representations from edge images.** In: THEORETICAL FOUNDATIONS OF COMPUTER VISION, p. 320–334, 2002. 4.2

[Kru03] KRÜGER, J.; WESTERMANN, R.. **Linear algebra operators for gpu implementation of numerical algorithms.** In: SIGGRAPH '03: ACM SIGGRAPH 2003 PAPERS, p. 908–916, New York, NY, USA, 2003. ACM. 3.1

[Kwa03] KWATRA, V.; SCHÖDL, A.; ESSA, I.; TURK, G. ; BOBICK, A.. **Graphcut textures: Image and video synthesis using graph cuts.** ACM Transactions on Graphics, SIGGRAPH 2003, 22(3):277–286, July 2003. 4.6

[Lez03] LEZORAY, O.; CARDOT, H.. **Hybrid color image segmentation using 2d histogram clustering and region merging.** ICISP, 3:289–396, 2003. 4.2

[Mar80] MARR, D.. **Visual information processing: the structure and creation of visual representations.** Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences, 290(1038):199–218, 1980. 1, 1.1, 3

[Mar06] MARFIL, R.; MOLINA-TANCO, L.; BANDERA, A.; RODRÍGUEZ, J. A. ; SANDOVAL, F.. **Pyramid segmentation algorithms revisited.** Pattern Recogn., 39(8):1430–1451, 2006. 4.1

[Moo65] MOORE, G. E.. **Cramming more components onto integrated circuits.** Electronics, 38(8), Apr. 1965. 2.4

[Owe07] OWENS, J.. **Data-parallel algorithms and data structures.** In: SIGGRAPH '07: ACM SIGGRAPH 2007 COURSES, p. 3. ACM, 2007. 2.1.1, 3.1, 3.1.2

[Owe07b] OWENS, J.; LUEBKE, D.; GOVINDARAJU, N.; HARRIS, M.; KRÜGER, J.; LEFOHN, A. E. ; PURCELL, T.. **A survey of general-purpose computation on graphics hardware.** Computer Graphics Forum, 26(1):80–113, Mar. 2007. 2.4.3, 3.1, 3.1.2

[Owe08] OWENS, J. D.; HOUSTON, M.; LUEBKE, D.; GREEN, S.; STONE, J. E. ; PHILLIPS, J. C.. **Gpu computing.** Proceedings of the IEEE, 96(5):879–899, 2008. 2

[Par99] PARHAMI, B.. **Introduction to Parallel Processing: Algorithms and Architectures**. Kluwer Academic Publishers, Norwell, MA, USA, 1999. 3.1, 3.1.2

[Raj05] RAJ, A.; ZABIH, R.. **A graph cut algorithm for generalized image deconvolution**. In: ICCV '05: PROCEEDINGS OF THE TENTH IEEE INTERNATIONAL CONFERENCE ON COMPUTER VISION, p. 1048–1054, Washington, DC, USA, 2005. IEEE Computer Society. 4.6, 4.7.1

[Rod02] RODRÍGUEZ, J. A.; URDIALES, C.; BANDERA, A. ; SANDOVAL, F.. **A multiresolution spatiotemporal motion segmentation technique for video sequences based on pyramidal structures**. Pattern Recogn. Lett., 23(14):1761–1769, 2002. 4.1

[Rog07] ROGER, D.; ASSARSSON, U. ; HOLZSCHUCH, N.. **Efficient stream reduction on the gpu**. In: Kaeli, D.; Leeser, M., editors, WORKSHOP ON GENERAL PURPOSE PROCESSING ON GRAPHICS PROCESSING UNITS, oct 2007. 3.1

[Roh97] ROHALY, A. M.; AHUMADA, A. J. ; WATSON, A. B.. **Object detection in natural backgrounds predicted by discrimination performance and models**. Vision Research, 37(23):3225–3235, 1997. 4.6

[Ros06] ROST, R. J.. **OpenGL(R) Shading Language, Second Edition**. Addison Wesley Professional, USA, 2006. 2.1

[Rot04] ROTHER, C.; KOLMOGOROV, V. ; BLAKE, A.. **"grabcut": interactive foreground extraction using iterated graph cuts**. ACM Trans. Graph., 23(3):309–314, August 2004. 4.6, 4.6.2, 4.7.1

[Sa06] SA, A.; BERNARDES, M.; MONTENEGRO, A.; CARVALHO, P. C. ; VELHO, L.. **Actively illuminated objects using graph-cuts**. In: PROCEEDINGS OF SIBGRAPI 2006 - XIX BRAZILIAN SYMPOSIUM ON COMPUTER GRAPHICS AND IMAGE PROCESSING, p. –, Manaus, October 2006. SBC - Sociedade Brasileira de Computacao, IEEE Press. 4.6, 4.6.2

[Saa94] SAARINEN, K.. **Color image segmentation by a watershed algorithm and region adjacency graph processing**. In: INTERNATIONAL CONFERENCE ON IMAGE PROCESSING (ICIP-94), volumen 3, p. 1021–1025, 1994. 4.2

[Sam90] SAMET, H.. **The design and analysis of spatial data structures.** Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. 4.3, 4.4.3, 4.4.4

[Sam08] SAMET, H.. **Sorting in space: multidimensional, spatial, and metric data structures for computer graphics applications.** In: SIGGRAPH '08: ACM SIGGRAPH 2008 CLASSES, p. 1–106, New York, NY, USA, 2008. ACM. 4.3

[Sch07] SCHEUERMANN, T.; HENSLEY, J.. **Efficient histogram generation using scattering on gpus.** In: I3D '07: PROCEEDINGS OF THE 2007 SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES, p. 33–37, New York, NY, USA, 2007. ACM. 3.1.2, 4.6.4

[Str06] STRENGERT, M.; KRAUS, M. ; ERTL, T.. **Pyramid methods in gpu-based image processing.** In: WORKSHOP ON VISION, MODELLING, AND VISUALIZATION VMV '06, p. 169–176, 2006. 4.1

[Szy95] SZYMANSKI, B. K.; MANIATTY, W. ; SINHAROY, B.. **Simultaneous parallel reduction on simd machines.** Parallel Processing Letters, 5:437–449, 1995. 3.1

[Tan75] TANIMOTO, S. L.; PAVLIDAS, T.. **A hierarchical data structure for picture processing.** Computer Graphics and Image Processing, 4:104–119, 1975. 4.3

[Tan77] TANIMOTO, S. L.; PAVLIDIS, T.. **The editing of picture segmentations using local analysis of graphs.** Commun. ACM, 20(4):223–229, 1977. 4.2

[Tan83] TANIMOTO, S. L.. **A pyramidal approach to parallel processing.** In: ISCA '83: PROCEEDINGS OF THE 10TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, p. 372–378, Los Alamitos, CA, USA, 1983. IEEE Computer Society Press. 3.1.2

[Vas07] VASCONCELOS, C. N.; SÁ, A.; CARVALHO, P. C. ; GATTASS, M.. **Using quadtrees for energy minimization via graph cuts.** In: PROCEEDINGS OF THE 12TH VISION, MODELING, AND VISUALIZATION WORKSHOP (VMV 2007), p. 71 – 80, 2007. 4.5, 4.5.1, 4.6, 4.6.4, 4.6.4, 4.7, 4.7.1

[Vas08b] VASCONCELOS, C. N.; SÁ, A.; TEIXEIRA, L.; CARVALHO, P. C. ; GATTASS, M.. **Real-time video processing for multi-object**

chromatic tracking. In: PROCEEDINGS OF THE 12TH (BMVC 2008), p. 113 – 122, 2008. 3.1.2, 3.1.2, 3.3, 3.4.2

[Vas08c] VASCONCELOS, C. N.; SÁ, A.; CARVALHO, P. C. ; GATTASS, M.. **Lloyd's algorithm on gpu**. In: ISVC '08: PROCEEDINGS OF THE 4TH INTERNATIONAL SYMPOSIUM ON ADVANCES IN VISUAL COMPUTING, p. 953–964, Berlin, Heidelberg, 2008. Springer-Verlag. 3.1.2, 3.4

[Vas08a] VASCONCELOS, C. N.; SÁ, A.; CARVALHO, P. C. ; GATTASS, M.. **A gpu-friendly quadtree leaves neighborhood structure**. In: PROCEEDINGS OF THE 26TH COMPUTER GRAPHICS INTERNATIONAL CONFERENCE (CGI 2008), p. 0–0, 2008. 4.4, 4.5

[Vin08] VINEET, V.; NARAYANAN, P.. **Cuda cuts: Fast graph cuts on the gpu**. In: CVGPU08, p. 1–8, 2008. 4.6

[Wan05] WANG, J.; BHAT, P.; COLBURN, R. A.; AGRAWALA, M. ; COHEN, M. F.. **Interactive video cutout**. In: SIGGRAPH '05: ACM SIGGRAPH 2005 PAPERS, p. 585–594, New York, NY, USA, 2005. ACM. 4.6, 4.7.1

[Wan07] WANG, J.; XU, W.; ZHU, S. ; GONG, Y.. **Efficient video object segmentation by graph-cut**. In: MULTIMEDIA AND EXPO, 2007 IEEE INTERNATIONAL CONFERENCE ON, p. 496–499, 2007. 4.6

[Wei05] WEI, Y.; QUAN, L.. **Asymmetrical occlusion handling using graph cut for multi-view stereo**. In: CVPR '05: PROCEEDINGS OF THE 2005 IEEE COMPUTER SOCIETY CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION (CVPR'05) - VOLUME 2, p. 902–909, Washington, DC, USA, 2005. IEEE Computer Society. 4.6

[Wil05] WILCZKOWIAK, M.; BROSTOW, G. J.; TORDOFF, B. ; CIPOLLA, R.. **Hole filling through photomontage**. In: 16th British Machine Vision Conference 2005 - BMVC'2005, Oxford, United Kingdom, p. 492–501, July 2005. 4.6

[Yin04] LI, Y.; SUN, J.; TANG, C.-K. ; SHUM, H.-Y.. **Lazy snapping**. ACM Trans. Graph., 23(3):303–308, 2004. 4.6, 4.7.1

[Yun06] YUN-TAO, J.; SHI-MIN., H.. **Interactive graph cut colorization**. The Chinese Journal of Computers, 29(3):508–513, 2006. 4.6, 4.7.1

[Zac03] ZACHMANN, G.; LANGETEPE, E.. **Geometric data structures for computer graphics**. In: PROC. OF ACM SIGGRAPH. ACM Transactions of Graphics, jul 2003. 4.3, 4.3.2

[Zie07] ZIEGLER, G.; DIMITROV, R.; THEOBALT, C. ; SEIDEL, H.-P.. **Real-time quadtree analysis using HistoPyramids**. In: Kehtarnavaz, N.; Carlsohn, M. F., editors, REAL-TIME IMAGE PROCESSING 2007, volumen 6496 de **Proceedings of SPIE-IS&T Electronic Imaging**, p. 1–11, San Jose, USA, January 2007. International Society for Optical Engineering (SPIE), SPIE and IS&T. 4.3.1, 4.3.2, 4.5, 4.5.3, 4.5.4

A
МОСТ

Real-Time Video Processing for Multi-Object Chromatic Tracking

Cristina N. Vasconcelos[*,+], Asla Sá[+], Lucas Teixeira[*,+],
Paulo Cezar Carvalho[±] and Marcelo Gattass[*,+]
[*]PUC-Rio, [+]Tecgraf, [±]IMPA
{crisnv,lucas,aslasa,mgattass}@tecgraf.puc-rio.br, pcezar@impa.br

Abstract

This paper presents **MOCT**, a **m**ulti-**o**bject **c**hromatic **t**racking technique for real-time natural video processing. Its main step is the MOCT localization algorithm, that performs local data evaluations in order to apply a multiple output parallel reduction operator to the image. The reduction operator is used to localize the positions of the object centroids, to compute the number of pixels occupied by an object and its bounding boxes, and to update object trajectories in image space. The operator is analyzed using three different computation layouts and tested over several reduction factors.

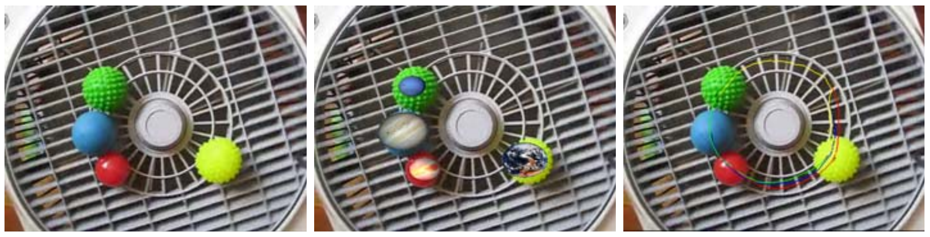


Figure 1: Multi-Object Real-Time Chromatic Tracking of Natural Images: (left) Original Video Frame, (center) Composite and (right) Trajectories

1 Introduction

Object localization is a well studied topic in early vision research. Countless possibilities are opened when such information is extracted in real-time, especially for the development of augmented reality applications.

Motivated by current graphics hardware processing power, Brunner et al. [1] presented a GPU-based technique for finding the location (centroid position and mass) of a single, uniquely colored object in a scene. This technique is attractive for its simplicity, but has serious limitations: besides dealing with a single object, in the composition step it assumes that the object has a square bounding-box (for instance, it does not treat adequately long, thin objects).

In this paper we present **MOCT**, a GPU-based technique for **m**ulti-**o**bject **c**hromatic **t**racking. Our main contribution is a reduction operator that works over the video frames for localizing a set of n distinct objects, each with a unique color (in parallel programming, a *reduction* is a computation that produces a smaller stream from a larger one [7, 5, 6]).

The proposed reduction operator is analyzed using three different computation layouts: row, column and tile oriented. We also consider several reduction factors: traditionally, reductions are computed halving each dimension at each step but we conclude that computing time can be reduced by adjusting the reduction factor. The results of this analysis can be used to make more efficient use of GPU resources, by optimizing its memory access pattern.

Additional contributions are: the extension of the reduction operator for computing bounding boxes, thus extending its applicability to objects having non-square bounding boxes; and the storage of the obtained results into a *trajectory history texture*, allowing the tracking in time of the centroid position and bounding box coordinates for each object. The *trajectory history texture* can be useful for computing movement predictions or producing temporal video effects in GPU.

The paper is structured as follows: In Section 2 we briefly describe the foundations of the parallel programming pattern used and its *General-Purpose Computation on GPU*(GPGPU) version, in order to clarify the structure adopted in MOCT. The processing steps of the MOCT technique are described in Section 3. In Section 4, the efficiency of the MOCT localization algorithm is compared to the technique presented in [1] and also compared against a CPU implementation. The comparison shows that our algorithm is faster than both of them, obtaining real-time rates for a set of several markers. We also show that for the MOCT the ratio between the computing time and the number of objects gets smaller as the number of tracked objects increases. The operator layout analysis is presented in Section 5. Finally conclusions are drawn and future work is discussed in Section 6.

2 Background and Related Work

The MOCT localization algorithm takes advantage of a parallel programing pattern called *reduction operator*. Such parallel programing pattern, also known as semigroup or fan-in operator, is defined by Parhami in [7] as follows: given an associative binary operator $\otimes$, a reduction is simply a pair $(S, \otimes)$, where S is a set of elements on which $\otimes$ is defined. Reduction computation is defined as: given a list of n values x_0, x_1, ..., x_{n-1}, compute $x_0 \otimes x_1 \otimes x_2 ... \otimes x_{n-1}$. Common examples for the operator $\otimes$ include +, $\times$, $\vee$, $\wedge$, $\otimes$, $\cup$, $\cap$, max and min.

Parhami [7] shows that a binary-tree architecture is ideally suited for this computation. Each inner node of the binary-tree receives two values from its children, applies the operator to them, and passes the result upward to its parent, and after $O(\lg_2 p)$ steps, the root processor will have the computation result.

The reduction operator pattern characteristics are extremely well suited for graphics hardware architecture as they offer task balancing design across the processors into independent kernels, and are widely used in GPGPU applications in cases where it is required to generate a smaller stream (usually a single element stream) from a larger input stream [3, 6, 5, 8].

Its design attends to GPGPU challenges as each one of its nodes is responsible for computing partial computations, in a manner that can be seen as an independent processing kernel with gather operations on previously computed values, i.e, by reading the corresponding values from an texture where the previous results have been saved. Thus, while a reduction is computed over a set of n data elements in O($\frac{n}{p}\log n$) time steps using the parallel GPU hardware (with p elements processed in one time step), it would cost $O(n)$ time steps for a sequential reduction on the CPU [6].

Traditionally, the reduction operator is used over a 2D texture by reducing by one-half in both vertical and horizontal directions. In this case, instead of a binary-tree, its structure is a pyramid representing a tree whose nodes have four children each. The input image corresponds to the pyramid base, which is reduced in multiple passes creating higher levels that correspond to intermediary computations until reaching its root, with 1×1 pixel dimension, corresponding to the final desired result.

The proposed reduction operator used by MOCT is generally classified as a multiple parallel reduction, as it can run many reductions in parallel (O($log_2 N$) steps of O(MN) work [4, 2, 6, 5]). This class of reduction operators, despite of its applicability, has not been widely explored yet.

An interesting example of a multiple parallel reduction operator is presented by Fluck et al. [2]. In that work, it is used for computing a histogram over a entire input image or over selected regions of it. The input image is initially subdivided into square tiles. During the processing, each texel within a tile represents a counter for the occurrence of the bin that it represents. Such subdivision prepares the input image for the reduction by computing partial histograms within the area covered by each tile. Finally, the partial histograms are then reduced by multiple halving steps into a single square tile, as proposed in [4].

In the field of chromatic tracking, Brunner et al. [1] use a pyramidal reduction operator on GPU to find the centroid position and pixel mass of a uniquely colored object in a scene. The obtained information is used for overlaying a textured square over the tracked object, creating video composition effects. Their technique starts by creating a binary image mask that indicates whether or not the pixel falls within the object, by comparing each pixel with the target object color in a thresholding procedure. In a second pass, the mask is reduced to a 1×1 image that stores the object mass and its centroid x and y coordinates (scaled by the mass), by means of a multi pass reduction operator that computes the sums of the positions of the pixels in the mask that are considered as belonging to the object. Finally, the data obtained is used for image composition.

Bruner et al.'s algorithm can be used for tracking multiple objects, each with a unique color, but this requires multiple executions of the algorithm, one for each object. Also, the overlaid object does not adjust itself to the shape of the object being tracked. Our algorithm, described in the next section, addresses these shortcomings.

3 MOCT: Multi-Object Chromatic Tracking

This section details MOCT - multi-object chromatic tracking - in GPU. The core of our tracking proposal is a n-object localization procedure via a multiple parallel reduction operator. The routine consists of two steps: a local evaluation (described in section 3.1) and a multiple parallel reduction (in section 3.2). The operator can be extended to objects having non-square bounding boxes, by means of bounding box extraction (in section 3.3). The thresholding sensitiveness can be reduced by transforming the input video frames color space as shown in section 3.4. Finally, the MOCT routine cycle is completed by storing a trajectory history of object movement as detailed in 3.5.

3.1 Local Evaluation

As in Fluck et al. [2] proposal for histogram computation, before applying the reduction operator we prepare each video frame using local evaluations, producing what we call a *base texture*. The basic idea is to build a texture that contains localization information regarding the objects in the scene. However, since we are tracking multiple objects, a masking texture where texels are in 1-1 correspondence with the pixels of the original image, as the one used in [1], does not work. Instead, following the ideas in [2], our base texture is subdivided into cells of size n, where n is the number of objects being tracked. Each cell contains tracking information concerning the corresponding region in the imput image. More precisely, the i-th texel of a given cell stores information regarding the count and localization of the pixels in the corresponding image region that are identified with object i. We investigate three different layouts for the cells in the base texture (Figure 2): a *vertical layout*, where cells are sets of n consecutive texels on the same column; a *horizontal layout*, where the n texels are on the same row; and a *square layout*, where the cells are squares with sides equal to $\lceil\sqrt{n}\rceil$ (this last one is the arrangement used in [2]). Note that in the last layout some of the texels of the cell may be unused for storing object information. Observe, also, that the size of the base texture may be slightly larger than that of the input image, since each of its dimensions must be an integer multiple of the corresponding cell dimension.

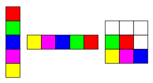


Figure 2: Operator Layouts: Vertical, Horizontal and Square

Whatever the layout chosen, local evaluation is done in a fragment shader that, for each fragment being produced, stores information regarding the occurrence, in the image region associated with the cell to which the fragment belongs, of pixels identified with the object corresponding to the fragment position within the cell. More precisely, the shader counts how many of the input image pixels are considered to belong to the corresponding object and also stores information regarding the position of their centroid. In order to do that, the fragment shader sweeps the region in the input image associated with the current cell, keeping track of the number of pixels classified as belonging to the corresponding object and of the sums of their x and y coordinates in image space. At the end of the local evaluation, the data is saved in the R, G and B channels of the *base texture* (the alpha channel can optionally be used to save the object ID).

Figure 3 illustrates a local evaluation for the vertical layout, showing that the counter indicates an evaluation over the area in the input image associated to the cell and not only over the corresponding pixel (observe the zoomed area). In the figure, positions that are not numbered have zero-valued counters.

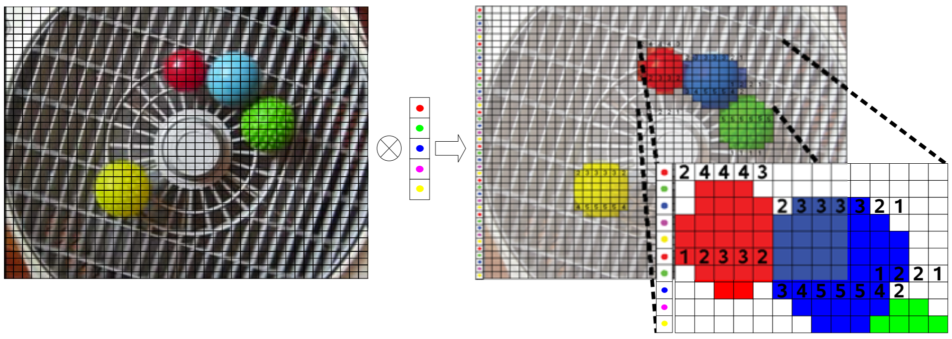


Figure 3: Base Preprocessing Using the Vertical Layout

3.2 Multiple Parallel Reduction

Now that local evaluators have set apart each object data into well defined cells within the *base texture*, the goal of the step presented in this section is to assemble such data from the texture generated into a single storage space for each object. Thus, the goal of this procedure is to reduce the *base texture* into a new texture, by gathering the data corresponding to each object. As in any reduction, each new fragment computed within a level must read the appropriate samples from the previous level and gather their representative data into the newly generated fragment. Usually, reductions are designed in such a way as to group information regarding a set of $2 \times 2 = 4$ texels into a same texel, but in section 5 we discuss different reduction factors.

In our case we produce, at each level, a texture subdivided in cells, according to the layout chosen for the base texture. Each texel in the newly generated texture stores information regarding a specific object, obtained by simply adding the values in each R, G, B channel of the texels in the same position in the cells being aggregated.

When the reduction process is completed, a texture composed of a single cell is produced. Each texel of this cell corresponds to one of the objects and stores the number of pixels belonging to that object and the sums of their x and y coordinates from the input image. Thus, centroid position for each object may be obtained by simply dividing those sums by the number of pixels.

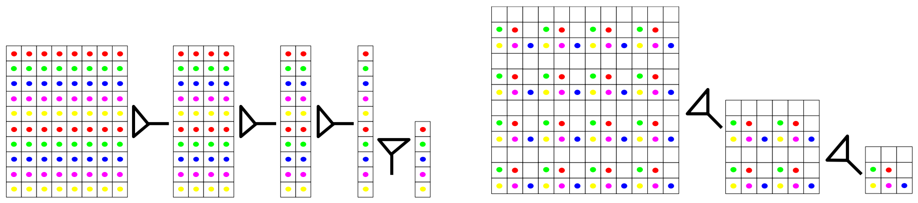


Figure 4: Reduction Processing for (left) Vertical and (right) Square Layouts

Figure 4 illustrates the reduction for the vertical and square layouts. A reduction for the horizontal layout is a trivial variation of the vertical case presented. It is important to notice here that, while in vertical and horizontal layouts the data gathered will be read from neighboring positions, in the square layout case they will be separated by a distance corresponding to the square side. A more thorough analysis of the layouts efficiency is presented in Section 5.

3.3 Bounding box

The use of the object centroid and mass for video composition suggested by Brunner et al. [1] yields good results for objects having a square bounding-box, such as spheres, but not for objects having other shapes. It is known that minimal bounding boxes can be used to give the approximate location of an object, offering a very simple descriptor of its shape, ideally suited for the composition effects desired.

Our operator can also be used to compute the objects bounding boxes. For that goal, during the *base texture* creation, the pixels classified as belonging to a given object should have their coordinates compared to local minimum and maximum coordinates. Thus, after this computation the *base texture* will contain local bounding boxes over the cells. During the reduction, the gathering is done again by choosing the minimum and maximum coordinates. Then, after gathering all the data contained in the *base texture* into a single cell, each object data will represent the minimum and maximum x and y coordinates, thus, its axis-aligned minimum bounding box.

3.4 Color Space

When colors similar to the objects' colors occur in other parts of the scene, it is difficult to calibrate the colors representative of each object. In some cases, better results can be obtained by working in a different color space. For instance, we may apply a preprocessing step to convert the input image from RGB to HSV, using a simple fragment shader, and then put more weight in the hue information. Figure 5 illustrates a situation, featuring a yellow background with a similar yellow object, where classification using a RGB color space fails, but classification in the HSV space succeeds.

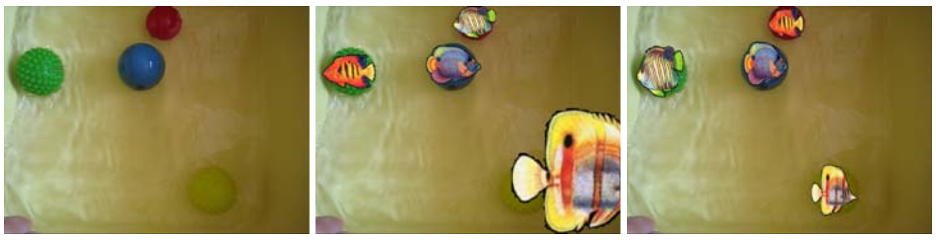


Figure 5: Original Frame (left); When Normalized RGB Goes wrong (center); HSV processing (right)

3.5 Trajectory History

For each video frame processed, after the localization procedure computed the objects centroids localization and bounding boxes, the MOCT processing organizes such data in order to maintain a sequential history of the trajectory of the objects. For that purpose, we create what we call the *trajectory history texture* by using a new fragment shader that receives the reduced texture and a time counter indicating the frame from which the data was extracted from. Then, the shader updates the trajectory by adding the object data to the next appropriate position in a sequential order, thus fulfilling the *trajectory history texture*. Figure 6 shows an illustration of the configuration for saving the data collected by the vertical layout. When the frame number is larger then the *trajectory history texture* width, the data is saved in a new row of texels as shown in figure 6 (center)).

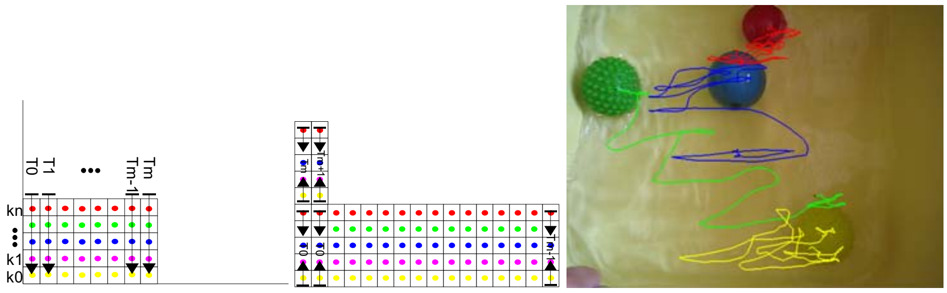


Figure 6: Trajectory History Textures (left and center); Trajectory Drawing (right)

4 Results

This sections presents the timing results comparing the MOCT localization procedure with an CPU implementation and also with the multi-pass Brunner's et al. algorithm ([1]), executed once for each object. The tests were performed using a Intel Core 2 Duo processor E6550 2.33Ghz with 2GB of RAM memory processor and a NVidia GeForce 8800 GTX (768MB) graphics card. The tests with more than 32 objects were done using synthetic images instead of natural ones.

As expected, the CPU implementation is slower than both of the others (left in Figure 7). It is interesting to note that while multi-pass Bruner's et al. algorithm for a set of objects presents constant ratio between the computing time and the number of objects, in our solution such ratio goes down, as the number of objects increases. The only configuration for which the extension of Brunner's et al. presents better performance than MOCT is the case of a single object. In that case the small difference observed in performance is associated with overhead incurred with layout positioning computations. In all cases where more than one object is localized, MOCT achieved significantly better performance. The results are quite expected, as the MOCT localization procedure was developed for an optimized texture access over the graphics card cache policy as detailed in next section. Those results demonstrate the MOCT applicability to real time tracking applications, as it achieves from 500 fps to 40 fps on tests using 1 to 128 objects, respectively.

5 Operator Layout Analysis

The present analysis is inspired by two observations: the number of texture access decreases when the proportion of the reduction increases, and, graphics card texture accesses are faster if the texture was already cached in the corresponding processor.

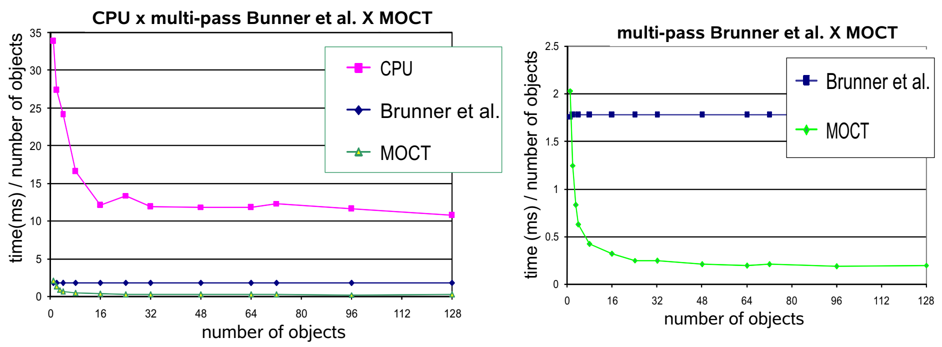


Figure 7: Comparison between CPU, multi-pass Brunner et al. and MOCT

Observe that the number of texture accesses of a reduction operator from a p-pixel input, until reaching the n-sized desired output with a reduction factor of r, can be described by the sum:

$$nr^i + nr^{i-1} + nr^{i-2} + \ldots + nr^3 + nr^2 + nr \tag{1}$$

where

$$i = min\left\{k \,|\, p \leq nr^k\right\} \tag{2}$$

This means that the higher the reduction factor used, the smaller the total number of texture samples read. In the limiting case, a single step is used during the reduction, making the total number of texture accesses equal to $n * r$, i. e., each sample is read only once. That configuration may not be the better performance result for the parallel computation as it can leave some processors without any work (as long as the number of processors is larger than the number of objects, n) but it shows that increasing the reduction rates induces a smaller number of multiprocessing steps and fewer texture sample readings.

As a second performance analysis factor, we test the three different proposed layouts. Even though graphics card cache memory policy is not open, it is known that memory access pattern does interfere in algorithm efficiency.

The graphs in figure 8 show how processing time changes as a function of the reduction rate, for each type of layout. We observe that better performance is obtained using the vertical and horizontal layouts instead of the square layout, what can be explained by the memory access localization principle. During the reducing, the texture samples read on vertical or horizontal layouts are neighboring texture samples that have a higher chance of representing a cache hit according to a cache policy that corresponds to caching a texture region, while for the square layout, corresponding objects are spaced by the square side, thus increasing the chance of a cache miss.

The different reduction scales and computation layouts presented in figure 8 provide a basis for an attempt of reducing texture already cached waste while maintaining the reduction computation distribution over the multi-processors. The graphs show that the best configuration for the MOCT efficiency was obtained using the vertical layout with a reduction rate of 40, i. e., 40 samples are read per reduction processing kernel.

We observe that the results presented in this section can be extended to other reduction operators such as the histogram computation presented by Fluck et al. [2]. Even

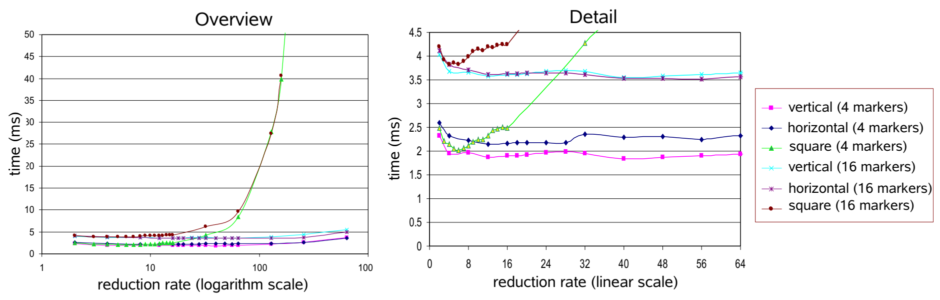


Figure 8: Layout and Reduction Factor Analysis: (right) overview, (left) detail

though the timing numbers shown here reflect the specific graphics card used they show that reductions other than the usual (simply halving the sizes) should be considered and confirm our assumption that understanding texture samples access patterns is essential to the development of an efficient reduction operator.

6 Conclusion

We presented MOCT, a technique under the *General-Purpose Computation on GPU* paradigm that defines procedures for tracking a set of objects identified by their colors from natural videos. It is composed by a localization procedure and a gathering step for collecting the objects' trajectory data.

The MOCT localization algorithm can be generally classified as a Multiple Parallel Reduction, whose goal is to find object centroids, mass and bounding boxes (allowing its applicability to objects having non-square bounding boxes).

As shown by the timing results, the MOCT localization algorithm is faster than a CPU localization procedure and also faster than applying several times, one for each target object, the technique proposed by Brunner et al. [1], originally devised to track a single object.

As an additional contribution, we compare three different layouts for the reduction operator and several reduction factors. We have shown how those choices can affect the overall efficiency of the reduction operators as they can be used for optimizing the number of texture samples readings, multi-processors occupancy and texture sample access patterns.

In summary, we conclude that the MOCT technique is well suitable for applications requiring the identification and localization of a set of uniquely colored objects at real-time rates.

References

[1] Ralph Brunner, Frank Doepke, and Bunny Laden. Object detection by color: Using the gpu for real-time video image processing. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 26, pages 563–574. Addison Wesley, July 2007.

[2] Oliver Fluck, Shmuel Aharon, Daniel Cremers, and Mikael Rousson. Gpu histogram computation. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Research posters*, page 53, New York, NY, USA, 2006. ACM.

[3] Daniel Horn. *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter Stream reduction operations for GPGPU applications, pages 573–589. Addison Wesley, 2005.

[4] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 908–916, New York, NY, USA, 2003. ACM.

[5] John Owens. Data-parallel algorithms and data structures. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 3. ACM, 2007.

[6] John Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[7] Behrooz Parhami. *Introduction to Parallel Processing: Algorithms and Architectures*. Kluwer Academic Publishers, 1999.

[8] David Roger, Ulf Assarsson, and Nicolas Holzschuch. Efficient stream reduction on the gpu. In David Kaeli and Miriam Leeser, editors, *Workshop on General Purpose Processing on Graphics Processing Units*, oct 2007.

B
CVD

Lloyd's Algorithm on GPU

Cristina N. Vasconcelos[1], Asla Sá[2], Paulo Cezar Carvalho[3], Marcelo Gattass[1,2]

[1] Depto. de Informática - Pontifícia Universidade Católica (PUC-Rio).
[2] Tecgraf (PUC-Rio).
[3] Instituto de Matemática Pura e Aplicada (IMPA).
Emails:{crisnv@inf.puc-rio.br, asla@tecgraf.puc-rio.br, pcezar@impa.br, mgattass@tecgraf.puc-rio.br}

Abstract. The Centroidal Voronoi Diagram (CVD) is a very versatile structure, well studied in Computational Geometry. It is used as the basis for a number of applications. This paper presents a deterministic algorithm, entirely computed using graphics hardware resources, based on Lloyd's Method for computing CVDs. While the computation of the ordinary Voronoi diagram on GPU is a well explored topic, its extension to CVDs presents some challenges that the present study intends to overcome.

1 Introduction

The Voronoi Diagram is a well known partition of space determined by distances to a specified discrete set of points in space. Formally it is defined as follows:

Given an open set Ω of $\Re^d$, a set of n different sites (or seeds) $z_i, i = 1...n$, and a distance function d, the *Voronoi Diagram* (or Tessellation) is defined as n distinct cells (or regions) C_i such that:

$$C_i = \{w \in \Omega | d(w, z_i) < d(w, z_j), \quad for\ i, j = 1...n, j \neq i\} \tag{1}$$

Voronoi Diagram computation is a topic of great interest not only in Computational Geometry but also in several scientific fields. One of its important variants is the *Centroidal Voronoi Diagram* (CVD), a special kind of Voronoi Diagram for which the points comprising the set that generates the tessellation are also the centers of mass of the Voronoi cells.

Generally speaking, CVD application is motivated by its capacity to cluster data, to select the optimal location for point placement, and its characterization as minimizer of an energy functional. Relevant theoretical and applied papers involving the computation of CVDs, whose properties have been well studied, are available in the literature [1–3].

A traditional sequential algorithm for CVD computation is Lloyd's algorithm [1], which iterates the computation of Voronoi tessellations and their regions' centroids until a convergence criterion is satisfied (similarly to optimal k-means cluster computation). Formally it is described as follows [1]: Given a set $\Omega \in \Re^n$, a positive integer k, and a probability density function ρ defined over the considered domain:

1. **Initialization**: select an initial set of k points $\{z_i\}_{i=1}^k$;
2. **Voronoi Tessellation**: compute $\{C_i\}_{i=1}^k$ of Ω associated with $\{z_i\}_{i=1}^k$;
3. **Centroid Computation**: compute the mass centroids of the Voronoi regions $\{C_i\}_{i=1}^k$ found in step 2. These centroids are the new set of points $\{z_i\}_{i=1}^k$;
4. **Convergence Test**: if this new set of points meets a convergence criterion, terminate; otherwise, return to step 2;

The CVD has been used in several different contexts, such as data and image compression, image segmentation and restoration, decorative mosaics, quantization, clustering analysis, optimal distribution of resources, cellular biology, statistics, studies on the territorial behavior of animals, optimal allocation of resources, grid generation, meshless computing and many others [1, 4, 5].

As a consequence of its large applicability, algorithms for an efficient and accurate construction of CVDs are of substantial interest. Our goal is to redesign Lloyd's algorithm in order to propose an efficient parallel implementation on GPU, taking advantage of the decreasing cost of programmable graphics processing units (GPUs).

The computation of Discrete Voronoi Tessellation using graphics hardware has been explored using different approaches [6–10], but its extension to CVD computation entirely on GPU presents some interesting challenges. Usually, in the literature, the Voronoi diagram is computed on GPU, while centroid computation and update, and the convergence of Lloyd's algorithm are computed on CPU, demanding a data read-back time related to passing the GPU-computed Voronoi diagram to the CPU as well as passing the new site positions computed on the CPU back to the GPU.

Modern GPU architectures are designed as multiple pipelines with massive floating-point computational power dedicated to data-parallel processing. The algorithm proposed here fulfills its architectural requirements by presenting a solution with independent data-parallel processing kernels, with no communication between data elements in each step of Lloyd's algorithm's computation.

This paper is structured as follows: the next section describes the existing methods for sequentially computing the CVD and the proposals found in the literature for computing the Voronoi diagram on GPU (Section 2). Then, an overview of the parallel algorithm suitable for current Graphics Hardware resources is presented (Section 3), and centroid computation is detailed (Section 4).In Section 5 we present efficiency results for different scenarios that illustrate the speed and quality of our solution compared to common CPU-GPU solutions.

2 Related Work

The computation of 2D and 3D Discrete Voronoi Diagrams using graphics hardware was initially proposed by Hoff et al. [6]. In their proposal, a mesh is created representing the distance function for each Voronoi site with bounded error. The distance mesh is orthogonally projected in a way that, for each sample in image space, the closest site and the distance to that site is solved by means of hardware-implemented polygon scan-conversion and Z-buffer depth comparison. After projection, each pixel in the frame buffer stores a color-coded identification of the site to which it is closest, while the depth buffer stores the distance to that site.

The evolution of programmable graphics hardware spurred the development of new methods for computing the Discrete Voronoi Diagram and its dual, the Delaunay triangulation, as can be seen in recent publications [7–10].

Recently, Rong and Tan [8] proposed a novel algorithm called Jump Flooding Algorithm (JFA) based on the idea of information propagation. This parallel algorithm solves the 2D Voronoi Diagram with almost constant time throughput regardless of the number of Voronoi sites used, but only in the final resolution adopted. The approach was later extended by the authors ([9] and [10]). We have adopted the solution proposed in [8] to implement the discrete Voronoi computation step of Lloyd's Method, as will be discussed in Section 3.

CVD computation based on Lloyd's method with a mixed CPU-GPU approach was initially proposed by Hausner [4], and formulated in the k-means context by Hall and Hart [11]. In both studies, the GPU is used to perform distance computations (computing the clusters, composed by Voronoi regions) while the CPU is responsible for computing and updating the centroids and for checking convergence at each iteration.

During the cluster/Voronoi region construction step, the graphics hardware evaluates the covered space and writes the minimum metric value for each sampled point of the space in the depth buffer. It also registers the IDs of the cluster/Voronoi regions that generated those values in the color buffer, producing a texture that represents the processing space within the cluster/Voronoi regions. As the texture that stores the cluster IDs must be read back to the CPU for further processing, these methods face a huge efficiency bottleneck related to communication from the GPU to the CPU.

The centroid computation step has not been solved in GPU to date. In the literature, the closest proposal to our method consists in finding the centroids using a variant of the parallel programming pattern called *parallel reduction operator*, adapted to generate multiple outputs as described in Subsection 4.1.

The reduction operator pattern is widely used in GPGPU applications in cases that require generating a smaller stream (usually a single-element stream) from a larger input stream [12–14]. Common examples of reduction operators include $+$, $\times$, $\vee$, $\wedge$, $\otimes$, $\cup$, $\cap$, max and min. Its design responds to GPGPU challenges, as each one of its nodes is responsible for performing partial computations, which can be seen as independent processing kernels with gather operations over previously computed values, i.e. that operate by reading the corresponding values from a texture where the previous results have been stored. Thus, while a reduction is computed over a set of n data elements in $\mathrm{O}(\frac{n}{p}\log n)$ time steps using parallel GPU hardware (with p elements processed in one time step), it would cost $O(n)$ time steps for a sequential reduction on the CPU [12].

A variant of the reduction operator, called multiple parallel reduction, can run many reductions in parallel with $(\mathrm{O}(log_2 N)$ steps of $\mathrm{O}(MN)$ work [12–16]).

It is useful for reducing an input dataset to multiple outputs, such as in the proposal presented by Fluck et al. for computing image histograms [16], and the uniquely colored object localization from natural images by Vasconcelos et al. [17].

As described in Subsection 4.2, the algorithm proposed in this paper for centroid computation can be seen as a *multiple parallel regional reduction operator*, and can be extended to other applications beyond CVD centroid computation.

3 Parallel Pipeline for Lloyd's Algorithm on GPU

This section presents an overview of our proposal designed for data-parallel processing considering currently available GPU resources. The main questions to be solved are how to formulate the processing steps for parallel computing and how to define the data flow between processing steps, eliminating CPU-GPU transfers.

The overview of the proposed data flow is illustrated in Figure 1. It represents the interaction between consecutive Lloyd's Algorithm steps by producing intermediate results within GPU memory to be read at the next step of the pipeline. The following subsections describe how each step interacts with such flow.

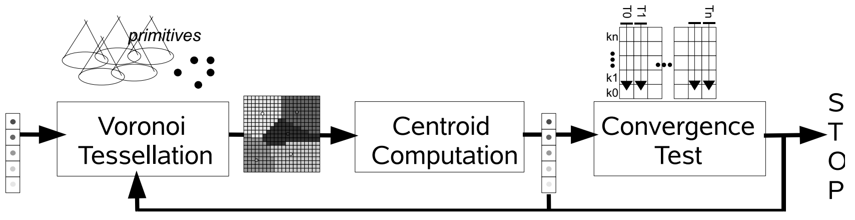


Fig. 1. Algorithm Pipeline and Data Flow

3.1 Voronoi Tessellation

Motivated by the near constant output rate for a varying number of sites, our study has adopted the solution proposed by [NOMES DOS AUTORES] [8] to implement the discrete Voronoi Tessellation step of Lloyd's Method. Other GPU solutions could be used, as long as they generate a texture with the space partition.

Traditionally, each Voronoi site is represented with a unique random color. In our method, we initially create the colors (IDs) of the sites using a sequential enumeration. By creating such sequential IDs, we are able to use them in the mapping algorithm created for centroid computation. This enumeration is done only once in a preprocessing step, during the creation of the sites, so that for n sites, the IDs vary between 0 and n-1.

Another adaptation implemented in our method is that the ID of each Voronoi site is saved using a single channel of the output texture. Observe that when Voronoi computation using graphics hardware [4] was proposed by Hausner, representing the IDs using a single channel would limit the number of sites to 256, as in older graphics hardware each color channel was limited to 8 bits. Thus, the use of the three color channels in previous proposals was a requirement for the construction of Voronoi diagrams with more than 256 sites. However, modern GPUs offer the resource of using float 32-bit textures, providing enough precision to identify uniquely a huge set of sites using a single color channel.

In our pipeline, the *Voronoi Tessellation* processing step is responsible for reading site positions from a texture and computing the corresponding space tessellation. Site positions are read from a texture directly from GPU memory space rather than being passed from CPU to GPU. This processing step only reads from the texture, leaving the *Centroid Computation* step responsible for writing the position updates to such texture. By arranging the site data into a texture, our iteration cycle can pass its contents along the algorithm pipeline without requiring CPU intervention.

In previous proposals, the CPU would calculate the new centroid positions and then create primitives that set the sites over the centroid positions found. In our algorithm, all primitives are created over the origin ((0,0) in 2D or (0,0,0) in 3D) but are translated to their positions on GPU by the *Voronoi Tessellation* procedure after the corresponding position of each site has been read from the texture.

After a new Voronoi Tessellation is computed, the output generated is a single-channel texture with enough resolution to cover the represented space, where each sample of the space is represented with a texel with an identification of the Voronoi site that is closer to that sample.

3.2 Centroid Computation and Convergence Test

The second step of our pipeline receives the texture representing the Voronoi Tessellation and is responsible for generating a *Centroid Matrix* containing the new (x, y) or (x, y, z) coordinates of the centroids. In a textural representation, each channel of the texture can be used to save one of the centroid coordinates. Centroid computation will be detailed in Section 4.

Each iteration of the proposed cycle generates a centroid matrix. Instead of overwriting the previously calculated centroid matrix, we fit them sequentially into a new texture, storing convergence history so that convergence analysis can be done by processing this texture over time.

The criterion used for convergence is a threshold on the total sum of the distances between current and previous centroid positions. The total sum of distances is calculated by initially creating a 1D texture (the size of one texel per Voronoi site) which, for each texel coordinate n, stores the distance between the current and the previous positions of the Voronoi site identified with the n-th ID. Both current and previous values are read from the *Convergence History Texture* using as texture coordinates a time counter (current iteration number) and the ID of the site. The total sum is found by repeatedly applying a reduction operator over this 1D texture that produces partial sums of the distance values, until producing a single value representing the total.

4 Centroid Computation

The *Centroid Computation* step is responsible for generating the *Centroid Matrix* by collecting data from a texture representing the space mapped into the Voronoi Tessellation.

4.1 Centroid Computation by Multi-dimensional Reduction

Centroid Computation can be implemented by applying a multi-dimensional reduction operator as proposed by Fluck et al. and Vasconcelos et al. [16, 17]. Both methods consist of two steps: several local evaluations analyzing the Voronoi Tessellation texture against the site set, and a multiple parallel reduction to add up these partial results. Here, we follow [17] rather than [16], as it also considers texture cache patterns.

Initially a base is constructed containing partial sums of the location information regarding the Voronoi regions, i.e. partial sums of their pixel coordinates. The base texture has an implicit subdivision into tiles that defines the local evaluation domains. Each tile is a grouping of size n, where n is the number of Voronoi sites. The size of the base texture may be larger than that of the Voronoi Tessellation texture as its resolution must be large enough to cover it with the tiles; thus, each of its dimensions must be an integer multiple of the corresponding tile dimension.

The parallel algorithm to create the base is defined in such a way that each processing unit is associated with a single site and is responsible for producing an evaluation of the Voronoi Tessellation texture restricted to the pixels covered by a tile. More precisely, each processing unit counts how many pixels of the Voronoi Tessellation texture within the tile domain actually belong to the corresponding Voronoi region and stores the sum of their pixel positions. Thus, the ith texel of a given tile stores information regarding the count and location of the pixels that are identified with Voronoi site i. In order to do that, each processing unit sweeps the region in the Voronoi Tessellation texture associated with the current tile, keeping track of the number of pixels classified as belonging to the corresponding Voronoi region (see figure 2) and of the sums of their x and y coordinates in image space.

Once the base is created, a multi-dimensional parallel reduction is used to assemble the local evaluators of each Voronoi site's data from the different tiles into the base texture and generate a global result in a single storage space for each site, i.e. to produce a single tile output.

Each site data is gathered by recursively adding the values read from the base positions corresponding to that site, and storing the number of pixels belonging to the site and the sums of their x and y coordinates from the input image. Thus, centroid position for each object is obtained by simply dividing those sums by the number of pixels.

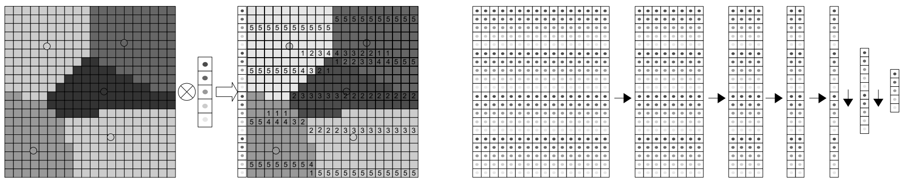


Fig. 2. Multi-dimension Reduction Operator: (from left to right) Voronoi Tessellation; a Single Tile; Base Texture; and a Set of Reductions

It is important to note that the method described by Vasconcelos et al. [17] creates a cell representing an object frequency in the base level by testing its represented color against each pixel covered by the corresponding tile. That means that for base creation, such method performs enough operations to compare each pixel in the input image against each object color, applying a total of $(nP \times nO)$ texture reads, where nP is the number of pixels in the original image and nO is the number of objects. This number of texture reads is prohibitive in our context. While for many natural video processing applications dealing with the object localization problem the number of objects is usually limited to a few dozen, in Voronoi Tessellation applications there are usually hundreds of sites. Moreover, since Lloyd's Method is a cyclic procedure, the number of reads increases even more as it has to be multiplied by nI, the number of iterations computed by the algorithm before convergence, thus yielding a total of $(nP \times nO \times nI)$ texture reads.

4.2 Centroid Computation by Multi-Dimensional Regional Reduction

We have shown that the multi-dimensional reduction operator suffers from scalability as the number of Voronoi sites increases. To overcome this we propose a new kind of parallel operator that we call *Multi-Dimensional Regional Reduction.*

The multi-dimensional reduction operator is designed as a data gathering operator. It makes no assumption about where within the input data the relevant regions are. When used for object localization from natural videos [17], it works like a global search covering the whole frame once for each object search without any region-of-interest clue. For the CVD we are interested in processing rendered data (the previously generated Voronoi Tessellation), therefore our idea is to use the sites' primitive data as an initial guess about where the objects we are looking for are, and then create a distributed local search limited to an area around such primitives.

Our method retrieves the local frequencies of the Voronoi sites by applying a total of $(nR \times nO \times nI)$ texture reads, where nR is the number of pixels in a region around each primitive used as an adjustable input parameter for the algorithm which is expected to be much smaller than the total number of pixels nP. The local optimization proposed is based on the assumption that for any fixed resolution of the Voronoi Tessellation texture (nP), as the number of sites grows (nO), fewer pixels are covered by each site and such pixels are arranged around the site.

To compute such local evaluation, we have created a space subdivision hierarchy defined around each Voronoi site (see Figure 3) to be used by our algorithm. The higher level of such hierarchy is the *Quadrant level*. It is composed by a set of four quadrants Q_0, Q_1, Q_2, Q_3 surrounding a Voronoi site, which are placed in a left-right, bottom-up order. The area covered by the four quadrants of a site defines the region of interest within the Voronoi Tessellation texture to be analyzed by the *Multi-Dimensional Regional Reduction* operator when looking for the centroid of the Voronoi region corresponding to that site. By definition, the dimensions of the quadrants should be chosen to cover the maximum area expected for a single Voronoi region.

The next level of the hierarchy subdivides each quadrant into regular units named *patches*. The set of patches inside a quadrant is placed in a left-right, bottom-up order. Each patch defines an area within a quadrant (thus, within the Voronoi Tessellation texture), to be evaluated by a single processing unit. This level provides a mechanism to distribute centroid computation into as many processing units and processors as desired.

Each patch receives an unique number, Id_{patch}, that represents its position within the ordered set of patches related to the same Voronoi site. Such enumeration starts at the left-bottom patch from quadrant Q_0 and is sequentially incremented one by one in a left-right, bottom-up order. After all the patches within a quadrant have been numbered, the enumeration continues in the next quadrant, also in a left-right, bottom-up order. The identification number of a patch (Id_{patch}) is determined using Equation 2:

$$Id_{patch} = Q * \alpha + y' * \beta + x' \tag{2}$$

where Q represents the number of the quadrant where the patch is located, varying between 0 and 3; x' and y' are the horizontal and vertical coordinates of the patch within its quadrant, measured in number of patches; and α and β are constants that represent respectively the number of patches within each quadrant and the number of patches per line of the quadrant. An illustrative example using α as 9 and β as 3 (thus, a 3x3 patches-per-quadrant subdivision) is shown in Figure 3.

Now that the space subdivision is defined, we will describe the *Multi-Dimensional Regional Reduction* (MDRR) operator and how it is used to compute the CVD.

The general similarity between MDRR and the algorithm presented in Subsection 4.1 is that both are composed by a two-step procedure where the first step is responsible for computing local evaluations and the second step is responsible for collecting such data into a well-defined storage space with global results. In both algorithms, the first processing step generates a 2D texture with each texel saving a local evaluation of the Voronoi Tessellation texture against a single Voronoi site. The significant differences between the algorithms are related to how the local domains are defined (tiles versus patches) and the overall area covered by the set of such local domains to process each site (the whole Voronoi Tessellation texture versus the Quadrants defined around each site).

During the first step of the *MDRR* operator each processing unit is responsible for outputting a texel. The texels placed in the same column represent the results of the evaluations of the Voronoi Tessellation texture against a single Voronoi site. More precisely, the horizontal coordinate of the output texel defines the ID of the site currently being evaluated within the processing unit. Different processing units producing texels to be placed in the same column are responsible for testing the same Voronoi site but against different areas of the Voronoi Tessellation texture. Such areas are defined using the vertical coordinate of the texel, which therefore defines the space within the Voronoi Tessellation texture to be swept by the processing unit. The texture storing local evaluations is shown in the right side of Figure 3.

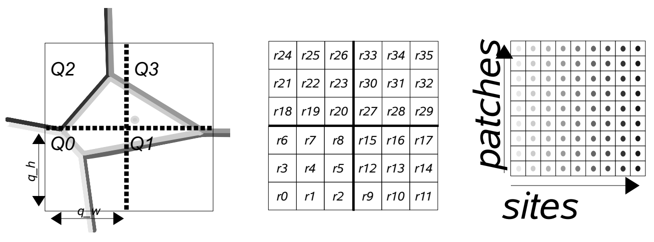


Fig. 3. Quadrants (left); Patches (center); Local Evaluations Texture (right)

The area covered by each processing unit is determined by reversing the patch enumeration procedure. The output texel's vertical coordinate is used as a patch number and an image space area within the Voronoi Tessellation texture is generated.

Reversing Equation 2, as α and β are constants, can be accomplished with the following procedure: Initially the patch quadrant is obtained through an integer division of the patch number by α (the number of patches within each quadrant). The remainder of this division represents the patch's number within its quadrant. This number is then divided (integer division) by β (the number of patches per line of the quadrant) so that the result represents the vertical position (y'), and the remainder is the horizontal position (x') of the patch within the quadrant.

$$Q = Id_{patch}/\alpha; \qquad y' = Q/\beta; \qquad x' = Id_{patch} - Q * \alpha - y' * \beta; \tag{3}$$

Each patch location must be obtained in image space coordinates (pixels) in order to access the Voronoi Tessellation texture. It is possible to use the Voronoi site's pixel coordinates and the input parameter defining quadrant size to determine each quadrant's origin in pixel coordinates (Q_{x0}, Q_{y0}). For simplicity, we consider that the patches are square regions of pixels and that the number of pixels on each side of such square is δ. As (x') and (y') are measured in number of patches within a quadrant, the image space coordinate (x_0, y_0) of the origin (left-bottom pixel) of a patch is retrieved through the following component-wise sum:

$$(x0, y0) = (Q_{x_0}, Q_{y_0}) + (x' * \delta, y' * \delta); \tag{4}$$

By reversing the patch enumeration procedure, each processing unit will know which area from the input image it should cover, and then it can sweep the pixels within a patch comparing the texels read from the Voronoi Tessellation texture against the represented site's ID. During the evaluation, it counts the frequency of pixels identified with the represented Voronoi region and sums their coordinates, saving such values in the generated texel.

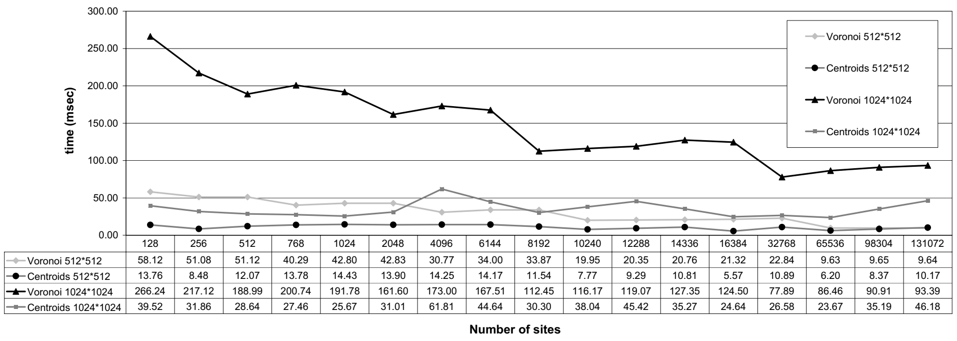
Finally, the Multi-dimensional Regional Reduction operator performs a reduction procedure in which the local results are added into a single line, where each position represents a single site's data. The centroids can be retrieved from this line by dividing each coordinate sum by the total number of pixels of the represented site.

5 Results

The quadrant sizes were chosen in order to cover a large area around each site thus safely including the related Voronoi region. The quadrant area used was four times larger than the area obtained by dividing the number of pixels of the Voronoi Tessellation by the number of sites. From a parallel programming point of view, it is important to stress that the total number of patches times the number of Voronoi sites defines the number of individual processing units to be distributed among the multiprocessors. Besides, patch dimensions define how many pixels are read by each one of the processing units (the texture area). Thus, the patch level was designed to provide a balance mechanism among the several processors, as well as texture cache patterns that can be adjusted in order to improve performance according to the architecture of the graphics card used.

To test the algorithm presented, an implementation using CUDA running over a GeForce 8600 GT was created. The timings were obtained for sets of different numbers of sites (from 128 to 128k) and two different resolutions of the Voronoi Tessellation texture (512×512 and 1024×1024). They are shown in Figure 4.

For testing the sets composed of 128 and 256 sites, quadrants subdivided into 9 and 4 patches, respectively, were used. For the other cases, 1:1 quadrants were used in the patch subdivision. The results have shown that by properly using the spatial subdivision hierarchy, centroid computation time is kept close to constant even if the size of the site sets varies significantly.



	128	256	512	768	1024	2048	4096	6144	8192	10240	12288	14336	16384	32768	65536	98304	131072
Voronoi 512*512	58.12	51.08	51.12	40.29	42.80	42.83	30.77	34.00	33.87	19.95	20.35	20.76	21.32	22.84	9.63	9.65	9.64
Centroids 512*512	13.76	8.48	12.07	13.78	14.43	13.90	14.25	14.17	11.54	7.77	9.29	10.81	5.57	10.89	6.20	8.37	10.17
Voronoi 1024*1024	266.24	217.12	188.99	200.74	191.78	161.60	173.00	167.51	112.45	116.17	119.07	127.35	124.50	77.89	86.46	90.91	93.39
Centroids 1024*1024	39.52	31.86	28.64	27.46	25.67	31.01	61.81	44.64	30.30	38.04	45.42	35.27	24.64	26.58	23.67	35.19	46.18

Fig. 4. Timing for Computing Voronoi Tessellations and Centroids over 512*512 and 1024*1024 Images

There is still room for optimization of the implementation tested, especially exploring CUDA memory hierarchy, but the objective of the tests presented was to compare the CPU-GPU model and the proposed algorithm. The tested implementation was constructed using only GPU programming resources that could be translated into shader languages.

We do not present the number of iterations before convergence because such number is intrinsic to Lloyd's Method's formulation. Therefore, it is expected to be the same for our GPU parallel formulation as for other CPU or CPU-GPU formulations, as long as the same initial conditions (set of sites and distance metric) are used.

6 Conclusions

This paper presented a computation of the Centroidal Voronoi Diagram through Lloyd's Method fully adapted to GPU resources. We showed how a data flow can be constructed so that it passes data through Lloyd's Method's iteration steps, eliminating the CPU-GPU texture reading presented in previous solutions. In particular, we described an efficient parallel computation algorithm to compute region centroids and to test convergence. By computing these steps on GPU we eliminate the read-back time related to passing the Voronoi diagram to the CPU, as is the case of previous proposals.

As future work, we plan to extend the proposed method to be used with varying distance metrics and with varying density functions. This can be obtained directly by changing the Voronoi Tessellation and by including a weight in centroid computation, respectively.

As a general contribution, the proposed Multi-dimensional Regional Reduction operator combined with the space subdivision hierarchy presented ensure an almost constant time processing throughput for a varied number of sites, thus motivating its use instead of the traditional reduction operator in cases where an initial localization clue, or a region of interest, is available.

References

1. Du, Q., Faber, V., Gunzburger, M.: Centroidal voronoi tessellations: Applications and algorithms. SIAM Rev. **41** (1999) 637–676
2. Har-Peled, S., Sadri, B.: How fast is the k-means method? In: SODA '05: Proceedings of the 16th ACM-SIAM Symp. on Discrete algorithms. (2005) 877–885
3. Du, Q., Emelianenko, M.: Acceleration schemes for computing centroidal voronoi tessellations. Numerical Linear Algebra with Applications **13** (2006) 173–192
4. Hausner, A.: Simulating decorative mosaics. In: SIGGRAPH '01: Papers, ACM (2001) 573–580
5. Du, Q., Gunzburger, M., Ju, L., Wang, X.: Centroidal voronoi tessellation algorithms for image compression, segmentation, and multichannel restoration. J. Math. Imaging Vis. **24** (2006) 177–194
6. Kenneth E., H.I., Culver, T., Keyser, J., Lin, M., Manocha, D.: Fast computation of generalized voronoi diagrams using graphics hardware. In: SCG '00: Proceedings of the 16th Annual Symp. on Computational geometry, ACM (2000) 375–376
7. Denny, M.: Solving geometric optimization problems using graphics hardware. In: EUROGRAPHICS 2003. Volume 22 of Computer Graphics Forum. (2003) 441–451
8. Rong, G., Tan, T.S.: Jump flooding in gpu with applications to voronoi diagram and distance transform. In: I3D '06: Proceedings of the Symp. on Interactive 3D graphics and games, ACM (2006) 109–116
9. Rong, G., Tan, T.S.: Variants of jump flooding algorithm for computing discrete voronoi diagrams. In: ISVD '07: Proceedings of the 4th Int. Symp. on Voronoi Diagrams in Science and Engineering, IEEE Computer Society (2007) 176–181
10. Rong, G., Tan, T.S., Cao, T.T., Stephanus: Computing two-dimensional delaunay triangulation using graphics hardware. In: SI3D '08: Proceedings of the 2008 Symp. on Interactive 3D graphics and games, ACM (2008) 89–97
11. Jesse D. Hall, J.C.H.: Gpu acceleration of iterative clustering. In: Manuscript accompanying poster at GP2: The ACM Workshop on General Purpose Computing on Graphics Processors, and SIGGRAPH 2004 poster, ACM (2004)
12. Owens, J., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum **26** (2007) 80–113
13. Owens, J.: Data-parallel algorithms and data structures. In: SIGGRAPH '07: Courses, ACM (2007) 3
14. Roger, D., Assarsson, U., Holzschuch, N.: Efficient stream reduction on the gpu. In: Workshop on General Purpose Processing on Graphics Processing Units. (2007)
15. Krüger, J., Westermann, R.: Linear algebra operators for gpu implementation of numerical algorithms. In: SIGGRAPH '03: Papers, ACM (2003) 908–916
16. Fluck, O., Aharon, S., Cremers, D., Rousson, M.: Gpu histogram computation. In: SIGGRAPH '06: Research Posters, ACM (2006) 53
17. Vasconcelos, C., Sá, A., Teixeira, L., Carvalho, P.C., Gattass, M.: Real-time video processing for multi-object chromatic tracking. In: BMVC 2008. (2008) 113–123

C
QuadN4tree

CGI 2008 Conference Proceedings

Cristina Vasconcelos · Asla Sá · Paulo Cezar Carvalho · Marcelo Gattass

QuadN4tree:

A GPU-Friendly Quadtree Leaves Neighborhood Structure

Abstract We propose a new model to represent the neighborhood relationship of quadtree leaves, called *QuadN4tree*. Quadtrees commonly store the represented data in their leaf nodes. Thus, several applications require a method to navigate across leaves. In the two traditional quadtree representations, namely hierarchi- al and linear representation, neighboring queries include spectively traversing internal nodes or arithmetically sting pairs of leaves. *QuadN4tree* allows the navigation rough such neighborhood systems from leaf to leaf, ithout passing through internal nodes, thus achieving n optimal cost for the retrieval of sets of neighbors.

A remarkable feature of *QuadN4tree* is that it is GPU- iendly, due to the fact that it stores a predefined num- er of references per leaf. In this work we present how our af neighborhood model can be used with both the hi- archical and the linear representations, commonly im- lemented in CPU, as well as how it can be constructed r a quadtree representation on GPU.

eywords Data Structures · Quadtrees · Quadtree eaves Neighborhood System

1 Introduction

QuadN4tree is a new model for the representation of quadtree leaf neighborhood system that allows the construction of query and navigation algorithms through neighboring leaves without traversing through any internal node of a quadtree. We achieve such properties economically by storing four references to specially chosen neighbors that allow spanning the 2D space covered by the quadtree.

This idea was motivated by the widespread use of quadtree data structures in applications such as image processing, spatial indexing, 2-D collision detection, view frustum culling of terrain data, sparse data storage and many others.

A key advantage of using quadtree structures is the reduction of the analysis space of a given input spatial data. For instance, it is useful for solving the lack of scalability of known computer vision methods, such as when dealing with large-resolution images or video streaming processing.

Using quadtree leaves as the basic processing unit may require querying their leaf neighborhood system. QuadN4tree allows retrieving the complete neighborhood relationship at a cost of one reference reading per neighbor.

Storing topology relations directly with a structure is not a novelty and can be found in solutions like the Winged Edge Structure [4]. Winged edge topology is a method that mathematically defines all elements in a model by their relationships to other elements of the model. This is done through a set of tables, namely face, edge and node tables. Maintaining and storing such structure can be impractical for applications dealing with a large volume of data. As discussed in section 3.1, the QuadN4tree achieves a balance between storing references to all neighbors within each leaf (which can be costly for updating space and time) and not storing any reference (like in traditional hierarchical and linear quadtree representations). QuadN4tree represents neighborhood relations in a compact way that allows retrieving neighboring sets of each leaf at the optimum cost of one reference reading per neighbor.

Several applications can benefit from the proposed QuadN4tree model. In Section 5 we exemplify how the recent computer vision applications proposed by Agarwala [1], Vasconcelos et. al [8] and Loaiza et. al [5] can take advantage of the QuadN4tree in different ways.

C. Vasconcelos and M. Gattass
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio).
E-mail: crisnv@inf.puc-rio.br
E-mail: mgattass@tecgraf.puc-rio.br

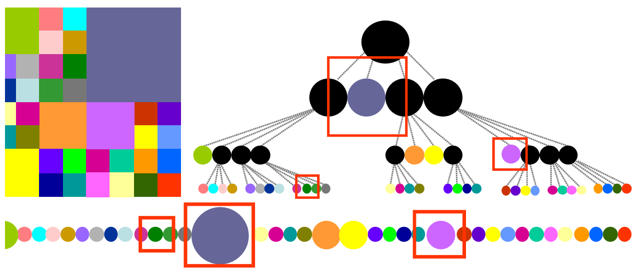
A. Sá and M. Gattass
Tecgraf (PUC-Rio)
E-mail: asla@tecgraf.puc-rio.br

P. C. Carvalho
Instituto de Matemática Pura e Aplicada (IMPA).
E-mail: pcezar@impa.br

The remainder of this paper is structured as follows. In Section 2, we briefly describe previous works on quadtree representation and how neighborhood searches are queried in each of them, as well as on representing such structures using graphics hardware. In Section 3, the QuadN4tree model is presented and it is also described how to use the QuadN4 references in navigation queries and how to update it. In Section 4 we discuss how to aggregate the proposed reference model to traditional quadtree representations and how to construct such reference model using GPU resources for representing the proposed neighborhood system on GPUs. In Section 5 we discuss why existing applications would benefit from the proposed model. Finally, conclusions are addressed in Section 6.

2 Quadtree Representation

In this section we briefly describe the two main forms used for representing quadtrees: hierarchical and linear (pointerless) representations (Figure 1). Detailed infor-ıation can be found in the extensive literature on the ıbject, as in [7,6,9,2]. We also describe the GPU im-lementation proposed by Ziegler et al. [3].


ig. 1 Quadtree spatial subdivision(upper-left); Hierar-ıical Representation (upper-right); Linear Representation ›ottom)

The hierarchical representation represents a quadtree by a tree structure composed by a root node, internal nodes, and leaf nodes [9]. For each tree node, pointers for its parent and children are stored. The basic operations, like point localization (search for the leaf that contains a point of the covered space) and leaf neighbor query (search for the leaves with common edges), are done following paths through the tree using the pointers.

The linear representation uses a pointerless representation [2]. In this case, only the quadtree leaves are stored, organized into a list of leaves. A *localization code* is associated to each leaf, working as a key for leaf positioning inside the list of leaves. Different rules are used for defining the localization codes, according to the querying method proposed. They encode a directional code sequence that allows leaf localization by using arithmetic expressions related to identifying different paths through the quadtree.

The construction of a quadtree structure for general purposes in GPU is proposed by Ziegler et al. in [3]. In that work, the result of the quadtree construction is a list of the found leaves. A reduction operator is described that creates an image pyramid called QuadPyramid. The operator writes in each fragment of the pyramid texture whether it represents a grouping of similar pixels or if it should be threaded as a quadtree internal node, in this case saving the number of leaves covered by the region represented by the fragment. A second shader is used to identify the quadtree leaves reading the pyramid texture repeatedly, simulating tree traversals from root to leaves. Relative counters, read from the pyramid texture, are used to control such traversals. The origin and size of the found leaves are saved in a output texture, organized as a point list.

Ziegler et al. [3] does not explore any leaf neighborhood relation and, as a consequence, the leaf list generated by the algorithm is not properly sorted for localization or leaf neighborhood queries. The QuadN4tree model offers characteristics that are well suited for a GPU implementation, and can be used to aggregate neighborhood relations to the output generated by [3].

3 QuadN4tree Model

The first attempt to create a structure that answers to neighborhood queries within the set of quadtree leaves would be to store references in each leaf to all of its neighboring leaves. The problem is that the number of neighbors of a leaf can vary up to, in the worst case, approximately the number of leaves of the entire tree. This happens even if a topological data structure (such as the winged edge) is used. Therefore, such reference structure requirements for storage space and updating time could be impractical for many applications. In order to solve this problem, we propose a structure for storing the leaf neighborhood allowing efficient neighborhood queries.

Our purpose is to save a minimum number of neighbors for each leaf in a manner that it should span the quadtree neighborhood system. Observing the quadtree subdivision rules, such minimum should cover both the vertical and horizontal neighboring queries, each of them in its both opposites directions. We conjecture that this minimum is achieved by using four references for each leaf and we show how to choose a reference quartet that spans the quadtree neighborhood system allowing to circulate along the neighboring leaves.

Our model, named QuadN4tree (where N4 refers to the number of references to neighbors) considers as origin for the references a pair of opposite corners of each leaf. We standardize the left bottom corner (*lbc*) and right top corner (*rtc*) as reference origins. From the *lbc* we take references to the north and right neighbors and from the

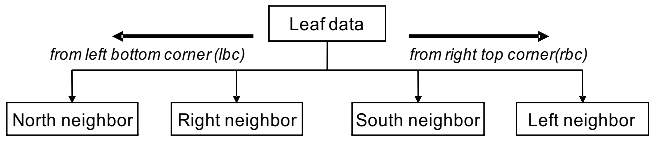


Fig. 2 QuadN4: The Neighborhood Reference Model

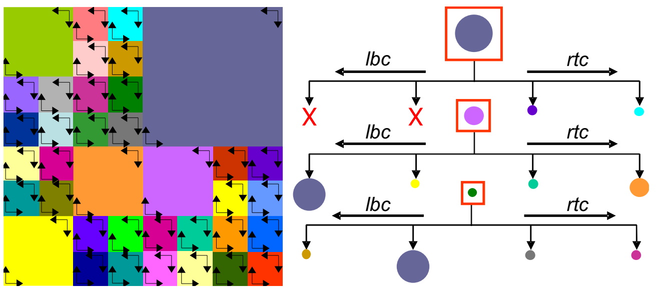


Fig. 3 QuadN4tree model for Figure 1

rtc we take references to the south and left neighbors, as illustrated in Figure 2. We applied the QuadN4tree ıodel to the example of Figure 1 to get the neighborhood lustrated in Figure 3.

To see why such references span the neighborhood ystem we initially observe the trivial case of a pair of eighboring leaves with the same size. In this case each af of the pair necessarily has a reference to the other ; the quadtree space subdivision rules impose a border ınge alignment. Thus, neighbors in horizontal direction ill reference each other with right and left references espectively. Analogously, neighbors in vertical direction ill reference to each other using north and south referıces.

The non trivial case happens with a pair of differıtly sized neighboring leaves. In this case, the smaller $_s$ has a reference pointing to the larger L_l, since the uadtree space subdivision guarantees that the border of the smaller is completely contained within the range of the border of the larger one. However, we can not be sure about if the larger has a reference pointing directly to the smaller. Thus, the question about if it is possible to reach the smaller leaf looking for the neighbors of the larger one remains. If the smaller leaf is aligned to one of the four references the answer is trivial: it can be reached by following such reference. Otherwise, there is another leaf L_{s1} aligned to the border of L_l where the leaf L_s is situated. Such new leaf L_{s1} should also be smaller than L_l or the first assumption, that L_l and L_s are neighbors, would be broken. The QuadN4tree allows us to walk along the border of L_l where both L_s and L_{s1} are located, initially using a reference from L_{s1}, and then navigating to same direction, from leaf to leaf, until reaching the desired leaf, L_s.

The ability of walking along each one of the four borders of any leaf, no matter its size, guaranties that the QuadN4tree spans the neighborhood system over the 2D space. Navigation rules will be presented in subsection 3.1.

The QuadN4tree property that a leaf always points directly to any neighboring leaf of the same size or larger than it can be used in queries for testing directly the neighborhood relation between a pair of leaves. For instance, it is possible to conclude whether two leaves are neighbors or not by checking if at least one of them has a reference to the other.

Finally, we observe that such model can be extended to include more references if that makes sense in some specific context. In particular, the natural extension to 3D would be to carefully choose six references from each leaf (taken from opposite corners), aiming to cover the three dimensions, each one in both opposite directions. Analogously to the QuadN4tree nomenclature, its 3D extension would be named as *OctN6tree*. Its formalization is left as future work.

In the next subsections we show navigation queries using the QuadN4tree (Sect. 3.1) and how such references can be updated in response to union or splitting of leaves operations (Sect. 3.2).

3.1 QuadN4tree Neighborhood Navigation

This section describes simple rules for querying the neighbors of a leaf using the proposed structure. Note that the internal nodes of the quadtree hierarchy are not used. After such rules, the cost of the proposed structure is compared to the traditional quadtree representation approaches.

Rules for finding the neighbors . Without loss of generality, consider a given a leaf A, To retrieve the neighbors along the north border of A, follow A's reference to the north and for each found neighbor B, if B does not cross the horizontal limit of A (defined by A's horizontal position and size), continue the search for new neighbors of A, by following B's reference to the east. Figure 4 illustrates the neighborhood navigation. The same reasoning should be used to retrieve the neighbors along the east border, that is, follow A's reference to the east then for each found neighbor follow it's references to the north while checking for the vertical limit of A (defined by A's vertical position and size). Analogously, to retrieve neighbors along the west border follow A's reference to the west then to the south. Finally, to retrieve neighbors along the south border follow A's reference to the south and then to the west. Observe that for south and west borders cases, the limits used are A's left-bottom corner positions.

Observe that the tests for the limits of a leaf used during navigation can be adapted for including the leaf neighbors in diagonal direction, by allowing the inclusion of the leaf covering the pixel situated just after such limits.

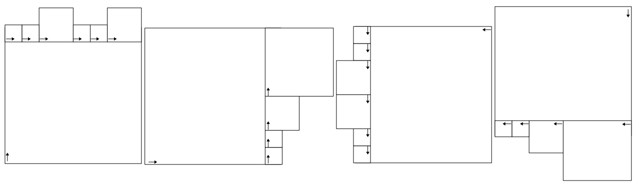


Fig. 4 Navigation Rules Along Borders (from left to right): North, East, West, South

The navigation rules can be used for converting from the QuadN4 neighborhood representation to a representation that stores all the neighbors of a leaf. For each leaf, such conversion would cost exactly one reference reading per neighbor, showing that the cost of manipulating the set of neighbors of a leaf in our structure is associated with following one reference per consecutive neighbor. Thus, finding the n consecutive neighbors of a leaf is obtained with the minimum cost O(n), which is the same cost of reading all the neighbors in a structure that stores all the references to them, but requiring much less storage space for the references.

Using the hierarchical representations, as it do not fer a circulation along neighbors method, the search r a set of neighbors of size n is related to the cost of times a neighboring query. Thus, finding the set of n eighboring leaves using the hierarchical representation ould be proportional to n searches for a neighbor in a uadtree of depth d (resulting in $O(n * d)$). In a linear epresentation, the search for neighbors is made through pairwise comparison testing between the input leaf veris the set of existing leaves. Considering each test with ost O(1), the search for neighboring sets of a leaf is ssociated with O(number of leaves).

However, the cost of looking for a specific neighbor sing the QuadN4 neighborhood has some interesting ases. The worst case happens when looking for the last af along a border, and the search is coming from a af with the biggest size that can be represented in the quadtree, and all of its neighboring leaves in the border to be followed have the smallest size represented in the quadtree. In a scenario where a quadtree represents a $2^n \times 2^n$ space, that the minimum sized leaves represent 1×1 areas, and that the maximum sized leaves represent $2^{(n-1)} \times 2^{(n-1)}$ areas, the worst case would consider following $2^{(n-1)} - 1$ leaves before reaching the last leaf of such huge border.

In that case, the use of a hierarchical representation for looking for one specific node would be preferable. Any neighbor is found by transversing the tree looking for an common ancestral and then going down until finding the desired neighbor [7]. In the hierarchical representation the worst case happens when both neighbors have the smallest size represented in the tree (sited in the highest depth) and only have the root node as common ancestral. In this case, the search for the neighbor would require following a number of references equal to two times the depth of the tree. However, using the QuadN4tree the cost of reaching such neighbor would be associated with reading only one reference, as they have the same size. Therefore, when looking for a specific neighbor, the performance of the proposed structure improves as the borders size ratio of neighboring leaves gets closer to 1, requiring less readings than the hierarchical representation if such ratio do not get higher that the depth of the tree.

In so far as, Section 3.1 showed that the QuadN4tree is optimum for queries involving the search of a leaf set of neighbors; that pairwise neighboring testing can be done just by checking if at least one of them has a reference to the other; and that the performance of the search for a specific neighbor is improved diminishing the difference between the size of neighboring borders.

3.2 Neighborhood Updating

The two basic operations that require the updating of a quadtree structure are: grouping four leaves into a new one and splitting a leaf into four new leaves [7]. This section examines the updating process of the references proposed in the QuadN4tree model.

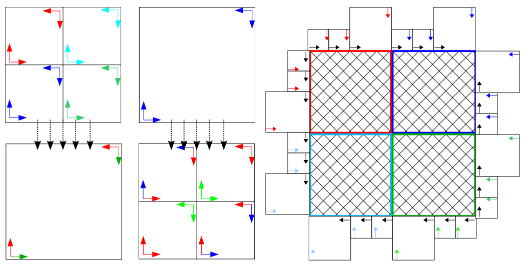


Fig. 5 (Left) References after grouping; (center) references after splitting and (right) external references affected

When creating a new leaf from the union of four leaves the references of the new one can be read directly from the original by reading the new north and left references from the input leaf situated in the north left position, while the south and right references can be read directly from the input leaf situated in the south right position (the chosen references are illustrated in the left side of figure 5)

When splitting a leaf into four new leaves, eight of the new references created inside those leaves are pointing to themselves, four of it are read from the original leaf and another four have to be searched within the neighborhood of the affected area by testing the new leaves limits versus those neighboring leaves for obtaining the missing references (see right side of figure 5).

Updating is a local operator in the sense that it affects only the neighborhood of the updated leaves, referred here as *operation area* (cross hatched area in figure 5). The alignment imposed to these operations guarantees that such set can be found by circulating along the *operation area*, as demonstrated in subsection 3.1, by using the *operation area* limits as the navigation limits criterion. Updating steps are detailed below:

Updating process: 1) create a new leaf (in case of union), or four new leaves (in case of splitting); 2) update the new leaf(ves) internal references that can be read from the input leaves, and also the references that point to themselves (only during split); 3) find the affected neighbors external to the *operation area*; 4) for each affected neighbor, if it has a reference that previously was pointing to the *operation area*, it should be updated to the new leaf created. In case of split operation, also test if this neighbor range is located over the new border created and, in such case, update the proper missing reference in the new leaf.

Observe that the cost of the updating operations are related to the cost of creating the new leaf(ves) references, and finding the affected neighbors to update at most one of their references. As we previously showed that we optimally retrieve the set of neighbors of any af by using the proposed structure, and that the af- cted neighbors can be found by circulating the *opera- on area*, then, clearly, the number of references to be pdated and created here is much smaller than in struc- ıres that store references for all the leaf references.

Representing the QuadN4tree

his section demonstrates how natural is to aggregate ıe QuadN4tree to traditional representations, and then, ı subsection 4.2, that the QuadN4tree can be used for ɛpresenting the quadtree neighborhood system using ıe graphics hardware.

4.1 The QuadN4tree and the Quadtree Traditional Representations

The QuadN4 reference model can be easily aggregated to both quadtree traditional representations. For the hierarchical representation, where relations are usually stored as pointers, four pointers should be included for each leaf, referencing the QuadN4tree neighbors. Then, such pointers can be used for navigating between leaves without traversing through internal nodes.

For the linear representation, supposing that each leaf is represented as a data in a linear list, associated with a identifier related to its list position, the QuadN4 reference model can be incorporated by adding four fields per leaf, representing the linear list position of the referred neighbors. Such fields can be used for non-linear navigation within the linear list, reproducing the navigation through our reference model.

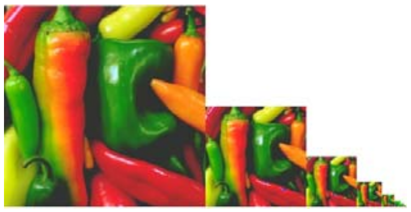


Fig. 6 Image Pyramid: grouping similar pixels into leaves

4.2 The QuadN4tree and the GPU Representation

This subsection shows how the QuadN4tree can be constructed for a GPU quadtree representation. It is important to notice that, since it is not know a priori how many neighbors a leaf will have, and the existing graphics hardware lack of variable-sized data structures, a parallel construction for saving all of its neighbors using the graphics hardware resources is not trivial. It is beyond our knowledge any previous proposal for representing the quadtree neighborhood system on GPU. The QuadN4tree model, being a compact neighborhood structure of predefined size (four references for each leaf), also contributes enabling a quadtree leaves neighborhood representation on GPU. The steps for aggregating the QuadN4 model into a GPU representation are presented bellow.

4.2.1 Quadtree Construction

As in Ziegler et al. [3], the quadtree construction on GPU starts by a reduction operator that creates an image pyramid, called by them as QuadPyramid. In order to do that, a texture is created over the 2D space area to be covered by the quadtree in such a way as that each texel represents the smallest grid element of the quadtree.

From the input image, one creates a image pyramid by using a recursive fine-to-coarse approach (Figure 6). Starting at the most refined level, all pixels are initially considered as candidates to be leaves of the quadtree. Then, in each level, for each fragment in the current pyramid level, the algorithm reads four texture samples the previous pyramid level, representative of its four children in the quadtree. Depending on the data represented by the quadtree, different criteria are used to decide if the leaves in the previous level should be grouped.

When following Ziegler et al.'s processing steps, the QuadPyramid constructed is processed for creating a list of leaves representing the quadtree. We are going to use the resulting list of leaves of [3] later, as the quadtree data representation. However, in this section we are focused in reviewing the QuadPyramid construction in a manner that allows the extraction of the neighborhood relations from this texture. The reduction operator proposed here prepares the QuadPyramid for being used as input for the extraction of the *leaf mask texture* as described in next subsection.

Our reduction operator analyses if the four samples read represent nodes that should not be grouped using the adopted criterion, and, in this case, classifies the fragment as a tree internal node. Otherwise, the fragment is provisionally classified as a new leaf, and receives a value representative of its four children. Such grouping are not yet the final leaves as they can be regrouped later, in the subsequent processing levels, into larger leaves. This evaluation is saved in the alpha channel of the pyramid to be analyzed in the next step. The reduction operator is performed level by level until reaching the pyramid root level (a single pixel representing the entire image).

Aiming the neighborhood system extraction, such tests describes a reduction operator that is simpler than the one presented in [3]. While grouping leaves, [3] also computes relative counters in fragments representing internal nodes. Those counters indicate how many leaves are covered by the internal node being processed and are used in [3] method for creating the leaves list. We do not use these partial counters during the neighborhood processing steps because we are interested in producing a mask with the found leaves, and not a list of aves as in [3] since the neighborhood disposition was .ready lost in such list. For grouping both operators in ıe QuadPyramid, our classification of leaves candidates : internal nodes can be read respectively from zeroed ıd non-zeroed counters of [3] pyramid classification.

The next subsection explains how this pyramid texıre is processed in other to identify, among the nodes rovisionally classified as leaves, the ones that are actu.ly leaves.

.2.2 *Identifying Final Leaves*

ı order to identify the quadtree leaves in the pyramid xture, we propose a leaf isolation method that does not quire computing several texture transversal, as used in]. Our leaf isolation algorithm reads the image pyramid previously constructed and discards texels representing nodes that are not leaves. For that, we use a new shader that reads our pyramid texture and discards all fragments that should not be leaf nodes in the final tree. This shader produces the output texture in a single rendering pass that makes at most two texture accesses per fragment, creating a mask with the identified the leaves.

Initially the fragment classification (candidate leaf / non-leaf) is read from the alpha channel of the pyramid texture. If the sample is already classified as non-leaf (internal node), the fragment is immediately discarded. Otherwise, the pyramid texture is queried again, now on its corresponding parent texture coordinate. When the parent was classified as a leaf it means that this fragment was grouped with its neighbors into a higher level leaf, so it can also be discarded. However, in the case of a non-leaf parent, this means that the previous shader could not group this node with its neighbors and that the fragment represents a leaf in the final tree.

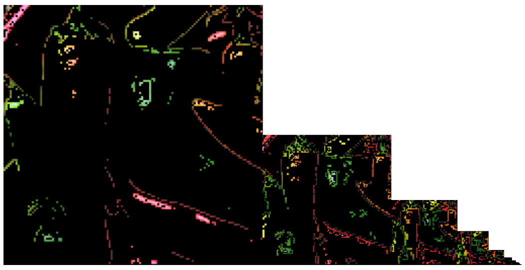


Fig. 7 Leaves Mask Texture

The fragments that pass through those tests are considered as being the final quadtree leaves and are written in the output texture that we call the *leaf mask texture*, saving in its channels the data to be associated with the leaves that the fragments represent (see figure 7). The *leaf mask texture* can be compared to a hierarchical representation as the identified leaves are still stored in different levels while Ziegler's leaves list [3] can be compared to the quadtree traditional linear representation.

Observe that the *leaf mask texture* can also be constructed with an operator that recreates a pyramid by reading the output generated by [3] and by repositioning each leaf from their list to its position and corresponding level within the pyramidal structure.

To construct the list of leaves for a quadtree of m leaves over a square image of N pixels, [3] algorithm may need $(m * \log(\sqrt(N)))$ texture accesses in the worst case. The algorithm presented here does at most two texture accesses for each leaf candidate, identifying real leaves from the previously created pyramid with less than $O(2 * m)$ texture accesses.

4.2.3 Querying with the leaf mask texture

This section shows how the *leaf mask texture* can be used for mapping each point of the space covered by the quadtree to the unique leaf that contains that point – that is, the leaf bottom-left corner x and y coordinates and level. With this information, we can retrieve all elements required for the QuadN4tree navigation.

In order to build such a mapping function, we use the *leaf mask texture* as a binary mask. Recall that each point in space is covered by a single leaf. Since the *leaf mask texture* preserves the leaves level and spatial arrangement, it is used as a mask for activating within the mapping function the unique level in which the considered point of the 2D space was represented into a leaf. Thus, the mapping between the (x, y) displacement of the 2D space and its representative leaf in the quadtree is given by the following sum:

$$LeafData(x,y) = \sum_{i=1}^{N} \alpha_i * (level_i, dx_i(x), dy_i(y)) \qquad (1)$$

where N is the maximum level of the quadtree, α_i indicates whether i is the level of the leaf covering (x, y)

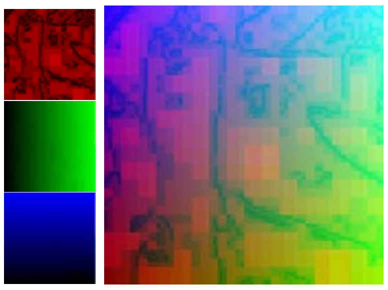


Fig. 8 2D Space Mapping Channels: leaves levels (in red); corner horizontal position (in green); corner vertical position (in blue); Composed Map

(a binary variable read from the *leaf mask texture*), and $dx_i(x)$, $dy_i(x)$ denote the coordinates of the bottom-left corner of a leaf of level i that covers point (x, y) (obtained by quantizing the x and y coordinates to the appropriate leaf size). Figure 8 applies the space mapping to the *leaf mask texture* presented in Figure 7.

4.2.4 Representing QuadN4tree in a texture

omputing the map described in the previous subsec- on is extremely simple and efficient, and queries re- ırding the neighbors of a given leaf can be answered y a few texture accesses in such map. However, one : the main goals for using a quadtree structure is to re- uce the space required for data representation, and such ıapping would require a storage space of the same size : the space covered by the quadtree. The representation resented next uses the mapping proposed, but only for nding the four QuadN4 neighbors of each leaf.

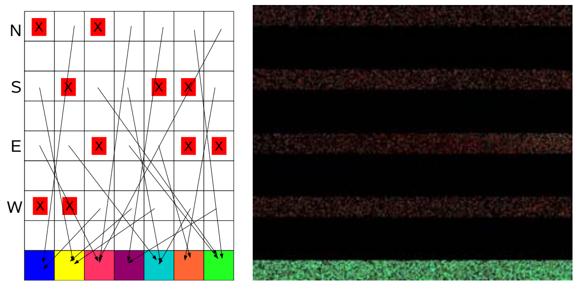


Fig. 9 QuadN4tree on GPU: Diagram and actual Texture

After creating a leaves list (as in Ziegler's [3]), and a *leaf mask texture* (as described above), four extra lists are created, each one storing one of the QuadN4 references (Figure 9). Such references are found by applying the mapping to a coordinate inside of a QuadN4 neighbor to obtain its leaves list position. Thus, the mapping between the (x, y) coordinate referred by a quadN4 reference and its leaves list position is given by the following sum:

$$Reference(x, y) = \sum_{i=1}^{N} \alpha_i * (LL_x(x, y, i), LL_y(x, y, i)) \quad (2)$$

where (x, y) are the coordinates within a neighbor (found by following the QuadN4 model), N, and α_i represent the same as in equation 1, and $LL_x(x, y, i)$, $LL_y(x, y, i)$ denote the texture coordinates within the leaves list of the neighboring leaf that covers point (x, y) in level i.

The presented representation limits the memory used to at most five times the number of leaves of a quadtree, at the same time allowing a navigation through neighbors by following the QuadN4tree navigation model. Storing a list of leaves and four lists of references into a texture can be further optimized. As the four references are 2D data, they can be grouped two by two for using the four texture channels. Thus, the representation would require a texture space of only tree times the number of leaves. We do not consider the storage space of textures generated by the intermediary processing steps (fot the QuadPyramid and Leaves Mask Texture) in this required storage space analysis as they can be discarded along the algorithm pipeline.

Table 1 presents the processing time (in milliseconds) for constructing the QuadN4tree representation on the GPU by grouping similar pixels into leaves for images of different resolutions. A NVidia GeForce 8800 GTX graphic card was used. It shows that the construction of the structure can be done in high frame rates, what makes the QuadN4tree representation suitable for real-time applications.

Table 1 Processing time (in ms)

Space Resolution:	$2^9 \times 2^9$	$2^{10} \times 2^{10}$	$2^{11} \times 2^{11}$
Image Pyramid:	0.14	0.29	0.37
Mask:	0.28	1.00	1.29
Leaves List:	0.47	1.02	1.37
Neighbors List:	0.22	0.72	1.17

5 Application Scenarios

In order to illustrate the applicability of the QuadN4tree structure in Graphics Problems this section discusses how it can be used in three recently proposed algorithms.

5.1 Panoramic Image Processing

In [1], Agarwala uses a quadtree structure to improve the efficiency of gradient-domain compositing by solving the problem in a reduced space. The use of a quadtree is motivated by the lack of scalability of gradient-domain compositing that was not previously practical to use for large resolution images. To obtain a smooth solution, a restriction is imposed to the quadtree: no two nodes that share an edge may differ in tree depth by more than one level. Agarwala's approach uses a pointer-based representation, where neighboring relations are found by tree traversals.

With the depth variation restriction over the neighboring leaves, the cost to find an specific neighbor in the QuadN4tree is at most the cost of two references reading, or at most eight references reading for finding all the eight neighbors of any leaf. The small number of reference reading must yields an improvement on the traversal efficiency.

5.2 Quadtree and Graph-Cuts

In [8], we proposed to accelerate the computation of energy minimization using graph-cuts over images by applying a preprocessing step that reduces the number of graph nodes and edges. During the preprocessing step, pixels are grouped into quadtree leaves using a similarity criteria. A general formulation for the energy function using the leaves as its variables and a general graph-cut ›rmulation over the quadtree leaves is presented.

The construction of a graph using the leaves of a ıuadtree as its nodes requires retrieving all the neighors from each leaf, as each neighborhood relation should e represented with an edge in the graph [8]. Applicaons with the same requirement (of retrieving the whole eighborhood system), can be greatly benefited from the ʒuadN4tree as our structure presents the optimum cost ›r retrieving all the neighbors of a leaf, with the cost of ıe reading per neighbor.

.3 Tracking Markers Detection

ıı [5], Loaiza et al. present an algorithm to group, label, identify and perform optical tracking of marker sets. Such markers are grouped into two specific configurations, defining collinear and coplanar patterns. The quadtree structure is initially used in a divide and conquer strategy to segment an image for finding the coordinates of markers that are spread over unknown positions. Then, quadtree traversals are used to group tracking markers into the specified patterns. Loaiza et al.'s grouping step suffers from scalability as tests are made with all possible combinations of markers positions.

With the QuadN4tree structure it is possible to retrieve the markers neighborhood within a chosen area, yielding to a grouping algorithm that does not have to consider all the marks presented in the image to test for pattern candidates. Thus, neighborhood relations would diminish the combinatorial grown of the algorithm proposed by [5], which can improve their scalability to a larger number of tracking markers.

6 Conclusions

In this work we have presented the QuadN4tree model. QuadN4tree's neighborhood scheme is based on a set of four neighbors and allows navigating throughout leaves that span the space covered by the quadtree. The proposed navigation method provides an optimum retrieval of the set of neighbors of any leaf. We also observe that the model can be extended to octrees by choosing the correct set of neighbors.

In Section 4 the QuadN4tree structure representation on GPU was modeled to offer independent computation of separate stream elements within a single kernel. The transcription of algorithms from CPU-based to GPU-based is not straightforward. The main restriction related to the representation of data structures is the lack of dynamic variable-sized data structures. This limitation precludes the direct use on GPUs of traditional quadtree representations designed for CPU use, but, as showed, does not preclude the use of the QuadN4tree model, making it GPU-friendly.

Although the proposed neighborhood scheme is quite natural, to the best of our knowledge it has not yet been proposed in the literature on data structures.

References

1. Agarwala, A.: Efficient gradient-domain compositing using quadtrees. ACM Trans. Graph. **26**(3), 94 (2007)
2. Frisken, S., Perry, R.: Simple and efficient traversal methods for quadtrees and octrees. Journal of Graphics Tools **7**(3), 1–11 (2002)
3. G. Ziegler R. Dimitrov, C.T., Seidel., H.P.: Real-time quadtree analysis using histopyramids. In: Proceedings of the IS&T and SPIE Conference on Electronic Imaging (2007)
4. Guibas, L.J., Stolfi, J.: Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. In: STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing, pp. 221–234. ACM, New York, NY, USA (1983)
5. Loaiza, M., Raposo, A.B., Gattass, M.: A novel optical tracking algorithm for point-based projective invariant marker patterns. 3rd International Symposium on Visual Computing ISVC 2007.Advances in Visual Computing - Lecture Notes in Computer Science **4841**, 160–169 (2007)
6. Samet, H.: The quadtree and related hierarchical data structures. ACM Comput. Surv. **16**(2), 187–260 (1984)
7. Samet, H.: The design and analysis of spatial data structures. Addison-Wesley Longman Publishing Co., Inc. (1990)
8. Vasconcelos, C., Sá, A., Carvalho, P.C., Gattass, M.: Using quadtrees for energy minimization via graph cuts. Proccedings of VMV - 12th Vision, Modeling, and Visualization Workshop. pp. 71–80 (2007)
9. Zachmann, G., Langetepe, E.: Geometric data structures for computer graphics. ACM SIGGRAPH 2003 Course #16 Notes (2003)

D
QuadCut

Using Quadtrees for Energy Minimization Via Graph Cuts

Cristina N. Vasconcelos[1], Asla Sá[2], Paulo Cezar Carvalho[3], Marcelo Gattass[1],[2]

[1] Depto. de Informática - Pontifícia Universidade Católica (PUC-Rio).
Rua Marquês de São Vicente, 225. 22453-900 - Gávea, Rio de Janeiro, RJ, Brasil
[2] Tecgraf (PUC-Rio). Rua Marquês de São Vicente, 225. 22453-900 - Gávea, Rio de Janeiro, RJ, Brasil
[3] Instituto de Matemática Pura e Aplicada (IMPA).
Estrada Dona Castorina, 110. 22460 - Jardim Botânico, Rio de Janeiro, RJ, Brasil
Emails:{crisnv@inf.puc-rio.br, asla@tecgraf.puc-rio.br, pcezar@impa.br, mgattass@tecgraf.puc-rio.br}

Abstract

Energy minimization via graph cut is widely used to solve several computer vision problems. In the standard formulation, the optimization procedure is applied to a very large graph, since a graph node is created for each pixel of the image. This makes it difficult to achieve interactive running times. We propose modifying this set-up by introducing a pre-processing step that groups similar pixels, aiming to reduce the number of nodes and edges present in the graph for which a minimum cut is to be found. We use a quadtree structure to cluster similar pixels, motivated by fact that it induces an easily retrievable neighborhood system between its leaves. The resulting quadtree leaves replace the image pixels in the construction of the graph, substantially reducing its size. We also take advantage of some of the new GPGPU concepts and algorithms to efficiently compute the energy function terms, its penalties and the quadtree structure, allowing us to take a step toward a real time solution for energy minimization via graph cuts. We illustrate the proposed method in an application that addresses the problem of image segmentation of natural images by active illumination.

1 Introduction

Many important problems in image analysis can be posed as optimization problems involving the minimization of some kind of energy function. For some of those problems, methods based on computing the minimum cut on graphs offer the possibility of finding global minimum for some classes of energy functions [3].

These methods explore the fact that algorithms for computing minimum cuts in polynomial time have been known for some time [1].

Much research has been done in setting the mathematical requirements for the energy functions that justify the use of Graph Cut minimization for both exact and approximate cases [1],[2],[3]. The applicability of the technique has also been shown by several papers in themes like image segmentation [7], foreground/background extraction [11], clustering [4], texture synthesis[10], photo composition[9] and so on.

However, the use of graph-cut methods for real-time applications has been limited by the size of the graph in which optimization must take place. In this paper we propose a pre-processing of the input images, in order to produce a new set of nodes and edges, instead of the image pixels and its neighborhood commonly used for the graph construction. The proposed sets are considerably smaller, inducing a significant reduction on the running time of the graph-cut procedure. We call Quad Cut the use

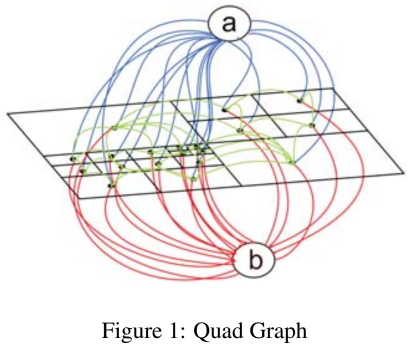


Figure 1: Quad Graph

of graph cut minimization in this modified way, the concept is illustrated in Figure 1.

The idea of the preprocessing is to group similar pixels, but in a way that creates a well known neighborhood system. For that reason, we choose to group them into Quadtree nodes. The metric used for grouping should be a similarity criteria appropriate to the context being analyzed by the energy function.

After constructing the quadtree, its leaves are used, instead of image pixels, as the basis for the construction of the graph. An appropriate energy function and neighborhood relationships are created to be used in this new procedure.

As we are interested in offering a fast approximation for the computer vision problems that rely on computing the minimum cut on an appropriately constructed graph, in addition to reducing the graph size, we also explore graphics hardware to efficiently compute energy function terms, its penalties and the Quadtree structure. Inspired by [15], we can take advantage of the Graphics Processing Unit (GPU) parallelism to compute all the preprocessing steps, including an efficient construction of a Quadtree with all the information needed for the optimization algorithm, leaving the CPU free to minimize the Graph constructed with the quad leaves.

As an application, we address the problem of foreground/background image segmentation aided by active illumination, in which graph cuts are used to compute an optimal binary classification, starting with an initial background/foreground separation, provided by the difference in intensity levels for two different illumination levels [11]. Figure 2 illustrates the application. Observe that the quality of the binary segmentation produced can be used for matting.

The paper is organized as follows: some applications that use energy minimization via graph cuts in vision are reviewed in the next Section; Section 3 briefly describes the basic concepts for energy minimization via Graph Cuts; then, in Section 4 we argue that grouping pixels into the Quadtree structure is useful to substantially reduce the nodes of the final graph to be cut. An GPU implementation to construct the quad tree structure is discussed in section 5. In Section 6 we present an illustrative implementation to accelerate the active illumination segmentation problem. Results are discussed in Section 6.4 followed by conclusions and future work.

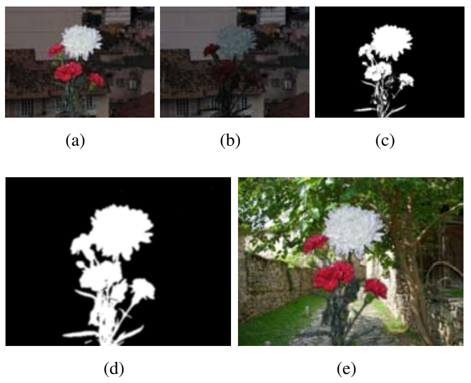


Figure 2: Example of minimization via Graph Cuts to the image segmentation of natural images aided by active illumination. In (a) and (b) the input images are shown. In (c) the initial segmentation provided by active illumination is compared to the final optimized segmentation shown in (d). The composition result (using parameters $\sigma_L = 0.25$, $\sigma_C = 0.05$) is shown in (e).

2 Related Work

In the Computer Vision and Graphics context, the graph cut method, can be interpreted as a clustering algorithm that works in a image feature space to produce spatially coherent clusters as result. Several recent works creatively models different applications as a labeling problem, then uses graph cuts to optimize the proposed labeling. This is the case in [9], where a framework for composing digital photos into a single picture, called digital photomontage, is described. Having n source images $S_1, ..., S_n$ to form a photo composition, the problem is posed as choosing a label for each pixel p, where each label represents a source image S_i. The proposed method extends the applicability of graph cuts to compute selective composites, photo extended depth of field, relighting, stroboscopic visualization of movement, time-lapse photo mosaics and panoramic stitching.

In [4], the spatial clustering problem is modeled as a labeling problem. The spatial coherence is guaranteed by the penalty imposed for neighboring pixels to have different labels, that are used as weights for the edges between neighbor pixels in the graph.

In [10], texture synthesis is modeled as labeling. The method generates textures by copying input texture patches into a new location, the graph-cut technique is used to find the optimal region inside the patch to be transferred to the output image. Such patch fitting step is a minimum cost graph cut problem using a matching quality measure for pixels from the old and new patch.

The problem of monochrome image colorization is modeled as a segmentation problem in [5]. The input image is partitioned interactively while the user specifies input colors, maintaining smoothness almost everywhere except for the sharp discontinuity at the boundaries in the image.

Image segmentation problem can also be solved by minimization via graph cuts. The main work lies in defining the energy function that models the specified application. In particular, background/foreground segmentation can be solved by means of Graph Cuts. In [6], [7] and [8] the user has to indicate coarsely the foreground and the background pixels, as initial restrictions for a minimization process. Then, graph cuts are used to find automatically the globally optimal segmentation for the rest of the image.

Similarly to our algorithm, [8] proposed the use of the image uniform regions as the nodes used in the graph construction in stead of the image pixels. They group similar pixels into such regions segmenting the original image using the watershed method. We believe that such segmentation do not provide a neighborhood system neither a boundary perimeter and area as easy to compute as the one presented in our proposal provided by the quadtree structure.

In this paper we will concentrate on applying graph cuts for image foreground-background segmentation aided by *active illumination*, as in [11]. Active illumination consists of using an additional light source in the scene that illuminates the foreground objects more strongly than the background. This gives *a priori* clues of the foreground. The information derived from this difference in illumination replaces the indication of object and background pixels by the user. These initial clues are then used as seeds for an optimization procedure in order to obtain a high quality segmentation. Potentially, the approach could be used for video capture, since a projector can be controlled to produce alternating illumination conditions at 60 Hz.

3 Basic concepts in Energy Minimization via Graph Cuts

In Computer Vision and Graphics, energy functions minimization is commonly computed using the min-cut/max-flow algorithms. The general goal for using the min-cut/max-flow algorithms is to find a labeling L, that assign each variable $p \in P$ (usually associated with the pixels of the input image) to a labeling $L_p \in L$, which minimizes the corresponding energy function.

The number of possible values assumed by the variables of the energy function is assumed finite, and modeled as a set of labels L, each label representing a possible output value.

The energy function to be optimized can be generally represented as [2]:

$$E(L) = \sum_{p \in P} D_p(L_p) + \sum_{p,q \in N} V_{p,q}(L_p, L_q), \quad (1)$$

Traditionally, $N \subset P \times P$ is a neighborhood system on pixels, $D_p(L_p)$ is a function that measures the cost of assigning the label L_p to the pixel p, while $V_{p,q}$ measures the cost of assigning the labels $\{L_p, L_q\}$ to the adjacent pixels p and q and is used to impose spatial smoothness.

The method of Graph Cuts to minimize (1) is applied by the creation of a graph normally containing nodes corresponding to each of the image pixels and some additional special nodes, called terminals, corresponding to each of the possible labels. There are two types of edges in the graph: n-links and t-links. N-links are the edges connecting pairs of neighboring pixels, representing the neighborhood system in the image, while t-links are edges connecting pixels with terminals nodes. All edges in the graph are assigned some weight or cost related to the energy function terms. The cost of a t-link corresponds to a penalty for assigning the corresponding label to the pixel, derived from the data term D_p in (1). The cost of a n-links corresponds to a penalty for discontinuity between the pixels. These costs are usually derived from the pixel interaction term $V_{p,q}$ in (1).

The Graph Cut finds a minimum of the energy function (1), providing an optimal labeling for the graph nodes [2].

4 Grouping Pixels into Quadtrees Leaves

When modeling computer vision problems as a energy-minimization problem, one can use different kinds of image features (e.g., luminance, color, gradient, frequency) and different metrics (e.g., statistical functions, differences between images, min/max relations). However, whatever the image feature or the metric used in the energy function, most natural images have areas of pixels presenting similar values according to them. Those pixels are expected to receive the same label in the energy minimization output. Our approach takes advantage of this fact, grouping pixels of such uniform areas, thus decreasing the graph size on which the min-cut algorithm is to be applied.

One more question arises here. If, on one hand, grouping pixels reduces the size of the graph, on the other hand, it may cause its adjacency topology to be more complex than the usual 4- or 8-connected pixel neighborhood systems. This may lead to spending considerable time both to find suitable clusters of pixels and to compute their adjacency relationships, overcoming the benefits by the smaller graph size.

Driven by these observations, we propose the use of a quadtree structure for grouping pixels into regions using a similarity criteria, while, at the same, creating a manageable neighborhood system between the quadtree leaves, in which adjacency relationships are easily retrievable.

In the next subsections we show how a graph for energy minimization can be constructed using quadtree leaves. The construction of the quadtree itself is discussed in section 5.

4.1 Graph Cuts using Quadtree Leaves

Using the quadtree leaves as the input data for the energy minimization via graph cuts, our goal is to find a labeling L, that assigns a label $L_t \in L$ to each leaf $t \in T$ of the quadtree, that minimizes the energy function adopted. The same set of the labels L may be used here. The modified energy function can be generally represented as:

$$E(L) = \sum_{t \in T} \alpha * D_t(L_t) + \sum_{t,u \in N} \beta * V_{t,u}(L_t, L_u), \quad (2)$$

Where $N \subset T \times T$ is a neighborhood system on the quadtree leaves, $D_t(L_t)$ is a function that measures the cost of assigning label L_t to leaf t, and $V_{t,u}$ measures the cost of assigning labels $\{L_t, L_u\}$ to the adjacent leaves t and u. The α and β terms are weights for balancing the energy function, explained below.

In such energy function model, the energy variables represent the quadtree leaves. Thus, graph cut minimization is applied to a graph containing nodes corresponding to each leaf of the quadtree and terminal nodes corresponding to each of the possible labels. Now, the n-links connect pairs of neighboring leaves, while t-links connect leaves with terminals nodes.

4.1.1 Weighting the Quadtree Nodes

The α and β factors were added to equation (2) in order to balance the energy metric according to leaves topology. The number of pixels inside a leaf t is $(2^{level(t)})^2$, while the number of pixels in the border between two neighboring leaves t and u is $2^{\min(level(t),level(u))}$. Therefore, we can rewrite (2) by taking α, that represents the weight for the regional term, as the leaf area, and β, that represents the weight for the boundary term, as the number of neighboring pixels between the two leaves.

With the suggested weights, we ensure that larger leaves have greater impact than smaller ones, while also enhancing the neighborhood influence of larger borders.

$$E(L) = \sum_{t \in T} (2^{level(t)})^2 * D_t(L_t) + \sum_{t,u \in N} 2^{\min(level(t),level(u))} * V_{t,u}(L_t, L_u), \quad (3)$$

5 Efficiently computing the Quadtrees

In this section we describe how the quadtree can be constructed efficiently using graphics hardware.

5.1 Quadtrees in GPGPU

The increasing use of the Graphics Processing Unit (GPU) for general-purpose computation (GPGPU) is motivated by its newest capability of performing more than the specific graphics computations which they were designed for.

In the context of our proposal, the GPU can be used for efficiently computing the energy function terms and also for constructing the quadtree whose leaves will be used as nodes in the graph cut minimization. For saving the partial results, we apply the useful concept of ”Playing Ping-Pong with Render-To-Texture” [17], rendering to Frame Buffer Objects (FBO) [19] when 32-bit floating-point precision is necessary.

A solution for constructing a quadtree structure for general purposes in GPU is presented in [15]. A reduction operator is described that creates an image pyramid called QuadPyramid. The operator writes in each fragment of the pyramid texture whether it represents a grouping of similar pixels or if it should be threaded as a quadtree internal node, in this case saving the number of leaves covered by the region represented by the fragment.

In a second shader, they identify the quadtree leaves reading the pyramid texture repeatedly, simulating tree traversals from root to leaves. Relative counters, read from the pyramid texture, are used to control such traversals. The origin and size of the found leaves are saved in a output texture, organized as a point list. To construct such list for a quadtree of m leaves over a square image of N pixels, their algorithm may need $(m * \log(\sqrt(N)))$ texture accesses in the worst case.

For our purposes, the resulting quadtree leaves will be used in CPU for graph construction. In addition to the origin and size of the leaves, we will also need leaf values that are used as the graph weights. We propose a simpler image pyramid operator for quadtree construction than the used in [15] and a new algorithm for identifying leaves from the pyramid texture. Next sections explain our methods for quadtree construction and leaves identification.

5.2 Quadtree Construction

Once a similarity criteria has been selected, the input image should be transformed to the adopted metric space, previously to the quadtree construction. For example, when grouping pixels by luminance, the original image should be transformed to the luminance space.

Here, as in [15], the quadtree construction starts by a reduction operator, creating an image pyramid. For each fragment in the pyramid level being constructed, the operator reads four texture samples from the previous pyramid level, representative of its four children in the quadtree. If the samples represent similar nodes, then, the fragment is classified as a leaf, grouping them into a single node that receives its children mean value. Otherwise, the fragment is classified as a tree internal node. The reduction operator is performed until the pyramid top level (1×1 pixel dimension) is reached.

Our algorithm is simpler than the one presented in [15]. While grouping leaves, [15] also computes relative counters in fragments representing internal nodes. Those counters indicate how many leaves are covered by the internal node being processed. In our case, we do not count the existing leaves inside a internal node region because this information is not needed in our leaves isolation solution.

For our purposes, the pyramid texture is used for saving the grouping decision (alpha channel) and the leaves values (RGB channels). Figure 3 shows an image pyramid found using the example application of section 6.

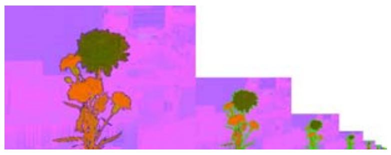


Figure 3: image pyramid found using the example application

5.3 Identifying Final Leaves

In order to identify the quadtree leaves in the pyramid texture, we propose a leaf isolation method that does not require computing several texture transversals, as used in [15], and, as a consequence, does not impose the use of a GPU supporting several nested branches.

Using the pyramid image as input, this processing step produces a texture whose pixels contain the data corresponding to a quadtree leaf (its size, position and representative value), or a color associated with empty data. This texture saves all the data needed for building the graph *a posteriori*.

Our algorithm erases texels representing other than leaf nodes in the pyramid texture. For that, we use a new fragment shader that reads our pyramid texture and discards all fragments that should not be leaf nodes in the final tree. This shader produces the output texture in a single rendering pass that makes at most two texture accesses per fragment.

The cleanup shader initially reads the fragment classification (leaf/non-leaf) from the alpha channel of the pyramid texture. If the sample is already classified as non-leaf, the fragment is immediately discarded. Otherwise, the pyramid texture is queried again, now on its corresponding parent texture coordinate. When the parent was classified as a leaf, this means that this fragment was grouped with its neighbors into a higher level leaf, so it can also be discarded. However, in the case of a non-leaf parent, this means that the previous shader could not group this node with its neighbors and that the fragment represents a leaf in the final tree.

The fragments that pass through those tests are considered as final quadtree leaves and are written in the output texture, saving in its channels all the data to be associated with the leaf that the fragment represents (see figure 4). By doing this, we guarantee that subsequent steps of the graph construction do not have to query any other texture.

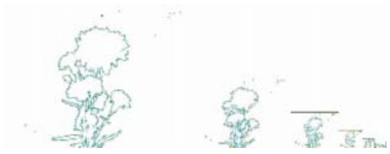


Figure 4: Quadtree Leaf Texture

All the information necessary for graph-cut computing is contained in this texture. For illustration, in figure 5 we reconstruct the entire quadtree using only the leaf texture shown in figure 4). Each leaf is painted according to its level.

6 Application to Active Segmentation

In this section we describe in detail an application of the proposed method to the problem of image segmentation by active illumination using graph cuts.

Segmentation using active illumination employs a single, intensity-modulated light source that stays in a fixed position between shots, as proposed in [11]. The two shots, differently illuminated, are used to obtain an initial segmentation used as a seed, referred as *segmentation seed*, and to attribute

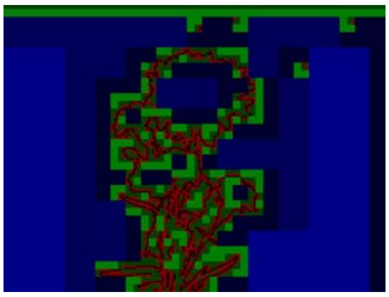


Figure 5: Found Quadtree (leaf color according with its level).

weights to the pixels that are used in graph cut optimization step to produce a improved final segmentation.

6.1 Energy Function Definition

The objective function adopted is the same proposed in [11]. The regional term considers the luminance difference between the two input images and the object color histogram as information that characterize the segmentation. The luminance difference for background pixels is considered to have Gaussian distribution, with density

$$\mathrm{p}_B(p) = \frac{1}{\sqrt{2\pi}\sigma_L}\exp\left(\frac{-|L_{I_2}(p) - L_{I_1}(p)|^2}{2\sigma_L^2}\right), \tag{4}$$

where σ_L is the standard deviation of the luminance differences, illustrated in figure 6 b.

The segmentation seed is defined as $O = \{p \mid \mathrm{p}_B(p) < t\}$, where t is a small threshold.

The color histogram of these initial foreground pixels are used to characterize the object as in [6]. In this work, only the components a and b of the Lab color systems are considered to characterize the object color distribution. For simplicity, the histogram is defined over a uniform partition.

The object distribution function is modeled as

$$\mathrm{p}_O(p) = \frac{n_k}{n_O} \tag{5}$$

where n_k is the number of pixels assigned to the bin k and n_O is the number of pixels in the object region O.

Observe that only one of the input images is used to construct the histogram information, since mixing different images may distort color information.

In most cases, we use the image corresponding to the lowest projected intensity.

The regional term of the energy function is:

$$R(x_p) = \begin{cases} -\log(\mathrm{p}_O(p)), & \text{if } x_p \text{ is } 1 \\ -\log(\mathrm{p}_B(p)), & \text{if } x_p \text{ is } 0 \end{cases} \quad (6)$$

where 1 is foreground and 0 is background.

The likelihood function for neighboring boundary pixels given by

$$\mathrm{B}(p,q) = 1 - \exp\left(\frac{-(||Lab(p) - Lab(q)||)^2}{2\sigma_C^2}\right), \quad (7)$$

where $Lab(p)$ denotes the color at point p and σ_C is the standard deviation of the L^2-norm of the color difference.

The boundary term for neighboring pixels p, q is given by $-|x_p - x_q| \log \mathrm{B}(p,q)$, where points q are the neighbors of p.

The final objective function combines both the regional and the boundary term and is given by:

$$E(\mathbf{X}) = \sum_{p \in I_1} R(x_p) - \sum_{p,q \in I_1} |x_p - x_q| \cdot \log \mathrm{B}(p,q), \quad (8)$$

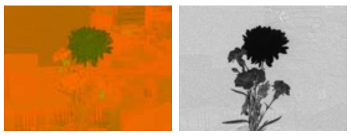
As shown in [11], the proposed energy function is regular, which means that it can be minimized by graph-cuts. This remains valid for the modified energy function defined on quadtrees leaves. As a consequence, Quad-Cuts can be applied to minimize the modified energy function.

6.2 Energy Function in GPU

The next sections describe how shaders can be used to compute efficiently the regional and boundary terms of the active illumination energy function applying GPGPU.

To pass the computed data efficiently across the algorithm we create what we call a *Stratified Texture*, illustrated in Figure 6.

The Stratified Texture is generated by saving, in its different channels, red, green, blue and alpha, all the data needed for the following steps of our algorithm. In this example application, the red and green channels are used for storing the a and b channels of the input image converted to Lab color space, the blue channel for storing the initial seed segmentation obtained by thresholding the luminance difference, and the alpha channel for storing the background distribution.


(a) a and b channels from *Lab* color space (b) background probability

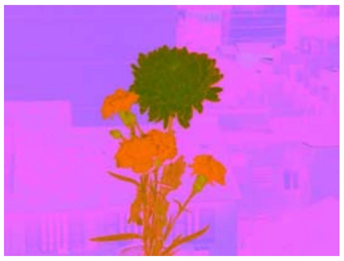

(c) RGBA are respectively *a, b*, segmentation seed and background probability

Figure 6: Stratified Texture.

6.2.1 Color Space Conversion

The input images are converted from RGB to CIE Lab color space, to exploit metrics in a perception-based color space presenting orthogonality properties between luminance and chrominance information.

Shaders for color space conversion have been used intensively by GPGPU programs. However, in order to efficiently compute the RGB to Lab conversion with high precision we also take advantage of the concept of rendering to texture with 32 bit floating point internal format using frame buffer objects (FBO) [19]. We save the Lab a and b computed channels in the resulting texture r and g channels, as illustrated in figure 6(a).

6.2.2 Background Probability

The background probability is computed in GPU according to equation (4), measuring the distribution of the luminance difference of the lit and unlit images. The result is illustrated in Figure 6(b).

For efficiently using the GPU parallelism, we pre-compute the constants $1/\sqrt{2\pi}\sigma_L$ and $1/(2\sigma_L^2)$ of equation (4) for a fixed σ_L. Those values are passed to the shader, avoiding repeatedly calculating it for every fragment.

6.2.3 Computing the Color Distribution

In order to compute the object distribution function using equation (5), we construct the histogram of the a and b channels from Lab color space (saved in stratified texture red and green channels), distinguishing object pixels using the object seed (from the stratified texture blue channel).

Motivated by its performance in computing histograms with a large set of bins, we choose to adapt [12] to our application context. Originally, that approach was proposed for monochromatic histograms, computing the histogram bin selection in a vertex shader, by loading the texture using either vertex texture fetches or by rendering the input image pixels into a vertex buffer, according to the graphics hardware capability.

We propose to adapt [12] to a vertex shader that computes bin selection in a 2D mapping, modifying it to compute a histogram representing the frequencies of occurrence in both input channels. Our vertex shader computes the vertex position by reading the a and b channels, multiplying their normalized values by the number of bins in the corresponding dimension, and then transforming the resulting values to frame coordinates.

Observe that a histogram of a trichromatic image could also be computed in GPU using techniques for representing 3D arrays such as those proposed in [18].

6.3 Application pipeline

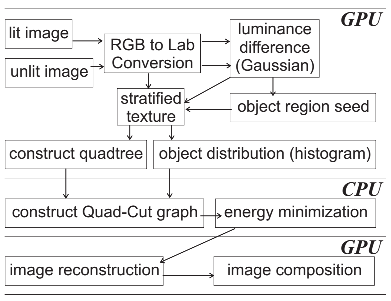


Figure 7: The proposed Quad-Cut method.

The main steps of the example application are illustrated in Figure 7.

The lit and unlit input images are converted to Lab color space. Then, another shader computes the background distribution texture. The result of those shaders are grouped in the stratified texture as described in section 6.2 and illustrated in figure 6.

The object distribution function is obtained by computing the object histogram of the a and b channels read from the stratified texture, using only pixels that failed the background threshold test (read from its blue channel). This histogram is saved in a texture to be used later in the energy function construction.

Then, the quadtree is created using our reduction operator through the stratified texture. Following the method in section 5.3, the resulting pyramid texture in cleaned, generating a texture that contains only the leaf nodes. that contains all information needed about each leaf: its level, from its relative position in the texture; its a and b from LAB conversion saved in the red and green channels; and the luminance distribution, saved in the blue channel.

All the above steps are computed in GPU. After them, the graph is constructed in CPU by reading the data from the leaf texture (fig. 4) and from the histogram textures.

In CPU we store the quadtree leaves in a pointer less representation, as a linear quadtree. The leaves are associated with location codes for fast neighbor search as in [16].

The graph is constructed using the leaf data, which stores the previously computed terms of the objective function, according to the method explained in section 4, which is minimized by the Graph-Cut minimization as in [1].

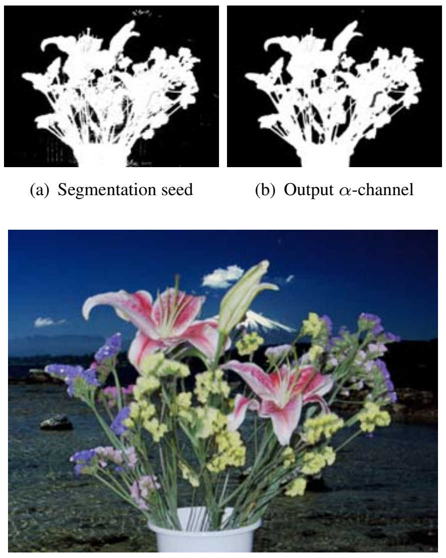
The solution of the minimization provides the classification of the quadtree leaves as background or foreground. So, using the position and size of each leaf, we reconstruct the resulting image that represents the alpha mask solution.

Back to the GPU, for the final composition, a smooth shader is applied to the computed alpha mask. Finally, a blending operator $\alpha F + (1 - \alpha)B$ is applied to the segmented foreground and the new background.

6.4 Results

Segmentation results using Quad-Cuts and the final compositions are shown in figures 2 and 8. To illustrate the considerable reduction in the number of variables in the minimization problem, both figures 2 and 8 are originally 800×600 (480,000) pixels, while the computed quadtrees have 9,556 (2%)

leaves and 30,036 (6%) leaves, respectively. Notice that the special characteristics of figure 8 (that presents many holes and thin structures) are automatically preserved through 15,992 leaves in the lowest level (1×1 pixel) and 8,718 in the next level (4×4 pixels).


(a) Segmentation seed (b) Output α-channel

(c) Composite

Figure 8: Composition Result 2 (using $\sigma_L = 0.25$, $\sigma_C = 0.05$).

We also measured the execution time of an background/foreground segmentation using graph-cut and active illumination with a Quad-Cut implementation with its preprocessing steps computed in GPU. A NVIDIA GeForce 7900 graphic card was used for the timings shown in Table 1.

Table 1: Processing time

step	*in seconds*
Energy function on GPU:	
RGB to Lab	< 0.001
Background prob	< 0.001
Histogram	< 0.015
Quad on GPU:	
Pyramid Construction	0.047
Quad Leaves Isolation	< 0.001
Quad on CPU:	
Reading Texture to CPU	0.015
Leaf List	0.016
Neighborhood	0.014
Graph-cut Minimization	0.001
Answer Reconstruction	0.016

7 Conclusions

We propose to accelerate the computation of energy minimization using graph cuts by applying a preprocessing step for reducing the number of graph nodes and edges. In this pre-processing, pixels are grouped by a similarity criteria according to the problem context.

We argue in favor of using a quadtree structure for managing such clustering regions, motivated by the easily retrievable neighborhood system between its leaves. In order to support our claim, we present a general formulation of the energy function using the leaves as its variables, and we also presented a general graph-cut construction over the quadtree leaves.

We also show how the quadtree structure can be constructed using graphics hardware. Initially, we use a reduction operator for constructing an image pyramid that writes in each texel whether a similarity clustering was applied or not. Such shader is simpler than the one proposed in [15]. Then we propose a leaf isolation method that discards from the pyramid texture all the texels that do not represent a quadtree leaf, efficiently removing unneeded information of non-leaf nodes. The proposed method requires fewer texture readings than the method proposed by [15], due to fact that the algorithm that it employs for finding leaves does not compute tree traversals for discovering each leaf in the tree.

Our graph construction method does not compute a point list on GPU of the quadtree leaves, as [15] does. Instead, as explained before, we use the leaf texture data to save the weights of the computed energy function, and the leaf texture coordinates are used to set the leaf level, size and corner position. Saving all the data needed for the posterior steps into such leaf texture allows an efficient interplay between the result generated in GPU and the energy minimization on CPU.

We also presented an application of our method to the foreground/background segmentation problem. It can be observed from the presented results (figures 2 and 8) that the proposed method for grouping pixels into quad leaves conserved image fine grain details of the original image (by creating leaves as small as 1×1) while also featuring a good grouping rate, by creating large leaves in regions of similar pixels .

We also show that the efficient implementation of all preprocessing steps on GPU leads to reasonably fast processing rates. As a consequence, we believe that our method constitutes an important step towards real time segmentation and matting using active segmentation.

References

[1] Y. Boykov, O. Veksler and R. Zabih. Fast Approximate Energy Minimization via Graph Cuts. *IEEE Transactions on PAMI,* 23(11):1222-1239, 2001.

[2] Y. Boykov and V. Kolmogorov. An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Computer Vision. In *Proc. of Int'l Workshop Energy Minimization Methods in Computer Vision and Pattern Recognition,* 2001.

[3] V. Kolmogorov and R. Zabih. What energy functions can be minimized via graph cuts? *Proc. IEEE Transactions on Pattern Analysis and Machine Intelligence,* 26(2):147-159, 2004.

[4] R. Zabih and V. Kolmogorov. Spatially Coherent Clustering Using Graph Cuts. in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'04),* 2:437-444, 2004.

[5] J. Yun-Tao and H. Shi-min. Interactive Graph Cut Colorization. *The Chinese Journal of Computers,* 29(3):508-513, 2006.

[6] C. Rother, V. Kolmogorov and A. Blake. GrabCut - Interactive Foreground Extraction using Iterated Graph Cuts. *ACM Trans. Graph.,* 23(3):309–314, 2004.

[7] J. Wang, P. Bhat, R. A. Colburn, M. Agrawala and M. F. Cohen. Interactive Video Cutout. In *Computer Graphics Proceedings ACM SIGGRAPH,* 2005.

[8] Y. Li, J. Sun, C. Tang, H. Shum. Lazy Snapping. *ACM Trans. Graph.,* 23(3):303–308, 2004.

[9] A. Agrawala, M. Doncheva, M. Agrawala, S. Drucker, A. Colburn, B. Curless, D. Salesin and M. F.Cohen. Interactive Digital Photomontage. In *Computer Graphics Proceedings ACM SIGGRAPH,* 294-302, 2004.

[10] V. Kwatra, A. Schdl, I. Essa, G. Turk, A. Bobick. Graphcut Textures: Image and Video Syntesis using Graph Cuts. In *ACM Transactions Graphics, Proc. SIGGRAPH,* 22(3):277-286, 2003.

[11] A. Sá, M. B. Vieira, A. Montenegro, P. C. Carvalho and L. Velho. Actively Illuminated Objects using Graph-Cuts. In *Proceedings of SIBGRAPI,* 2006.

[12] T. Scheuermann and J. Hensley. Efficient histogram generation using scattering on GPUs. In *SI3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games,* 33–37, 2007.

[13] O. Fluck, S. Aharon, D. Cremers and M. Rousson. GPU histogram computation. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Research posters,* 53, 2006.

[14] S. Green. Image Processing Tricks in OpenGL. In *Game Developers Conference (GDC05), 2005.*

[15] G. Ziegler, R. Dimitrov, C. Theobalt and H.-P. Seidel. Real-time Quadtree Analysis using HistoPyramids. In *IS&T and SPIE Conference on Electronic Imaging,* 2007.

[16] S. F. Frisken and R. Perry. Simple and Efficient Traversal Methods for Quadtrees and Octrees. *Journal of Graphics Tools,* 7(3):1-11, 2002.

[17] D. Goddeke. Playing Ping Pong with Render-To-Texture. *http://www.mathematik.uni-dortmund.de/g̃oeddeke,* 2005.

[18] M. Harris, D. Luebke, I. Buck, N. Govindaraju, J. Kruger, A. Lefohn and T. Purcell. GPGPU: General-Purpose Computation on Graphics Hardware. In *Tutorial at ACM SIGGRAPH 2005,* 2005.

[19] S. Green. The OpenGL Framebuffer Object Extension. In *Games Developers Conference (GDC),* 2005.