GPU Point List Generation through HistogramPyramids

Gernot Ziegler, Art Tevs, Christian Theobalt, Hans-Peter Seidel

MPI–I–2006–4–002 June 2006

Author's Address

Gernot Ziegler, Art Tevs
Christian Theobalt, Hans-Peter Seidel
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
Germany

Abstract

Image Pyramids are frequently used in porting non-local algorithms to graphics hardware. A Histogram pyramid (short: HistoPyramid), a special version of image pyramid, sums up the number of active entries in a 2D image hierarchically. We show how a HistoPyramid can be utilized as an implicit indexing data structure, allowing us to convert a sparse matrix into a coordinate list of active cell entries (a point list) on graphics hardware. The algorithm reduces a highly sparse matrix with N elements to a list of its M active entries in O(N) + M (log N) steps, despite the restricted graphics hardware architecture. Applications are numerous, including feature detection, pixel classification and binning, conversion of 3D volumes to particle clouds and sparse matrix compression.

Keywords

image pyramid, mipmap, graphics hardware, GPU, point cloud, feature detection, quadtree, voxelization

GPU Point List Generation through Histogram Pyramids

Gernot Ziegler, Art Tevs, Christian Theobalt, Hans-Peter Seidel

Abstract

Data Pyramids, as created during a reduction process of 2D image maps, are frequently used in porting non-local algorithms to graphics hardware. A Histogram pyramid (short: HistoPyramid), one incarnation of a data pyramid, collects the number of active entries in a 2D image. We show how a HistoPyramid can be utilized as an implicit indexing data structure, allowing us to convert a sparse matrix into a coordinate list of active cell entries (a point list) on graphics hardware . The algorithm reduces a highly sparse matrix with N elements to a list of its M active entries in O(N) + M (log N) steps, despite the restricted graphics hardware architecture. Applications are numerous, including feature detection, pixel classification and binning, conversion of 3D volumes to particle clouds and sparse matrix compression.

Categories and Subject Descriptors (according to ACM CCS): I.3.1. [Computer Graphics]: Graphics processors, I.3.5[Computer Graphics]: Point Representation, I.4.1. [Image Processing and Computer Vision]: Segmentation, I.4.10 [Image Processing and Computer Vision]: Hierarchical Image Representation

Introduction

As graphics hardware has become more programmable, new applications like general matrix calculation, sorting applications or physics processing have become feasible (e.g. [HAR04], [KW03], [FM05], [BFG*04]). But ever since the first of these applications had been implemented, it had been clear that the stream processing nature of graphics hardware, which gives it tremendous processing power, also requires considerable rethinking of data structures and algorithms. Many non-local calculations, virtually trivial on single-thread systems, like counting active cells in a 2D image, become hard to solve on the GPU, since its inherently parallel nature can only be utilized if the output of several parallel units is combined.

The thought approach of data pyramids solved this problem: A so-called *reduction operator* ([BP04]) summarizes the content of four cells at once, and writes out the result into a 2D image of half the input size. This is repeated until only one cell prevails. This way, all parallel units in the GPU can contribute equally to the final calculation of the result. We build on this concept and introduce the term *Histogram Pyramid* (short: HistoPyramid) for a special form of data pyramid that uses a reduction operator to sum up the active cells in a 2D image. “Active cells” are determined by applying a user-provided *discrimination function* to all cells in the image.

Therefore, we approach a related algorithmic problem, the generation of a *list of active cells* in a 2D image (a *point list*) on stream processing hardware. It is common knowledge that on such architectures, it is *not allowed to forward data* from one output element to the next one. Therefore, the trivial CPU solution, which traverses a 2D image sequentially in order to count all occupied pixels, and writes down cell coordinates as they are encountered, is not applicable.

We therefore present a completely GPU-based algorithm which uses the aforementioned histogram pyramid to generate a point list of a 2D image. For each point list entry to be generated, it traverses the histogram pyramid from the top level downwards until the corresponding point has been found. The histogram pyramid thus serves as an *implicit in-*

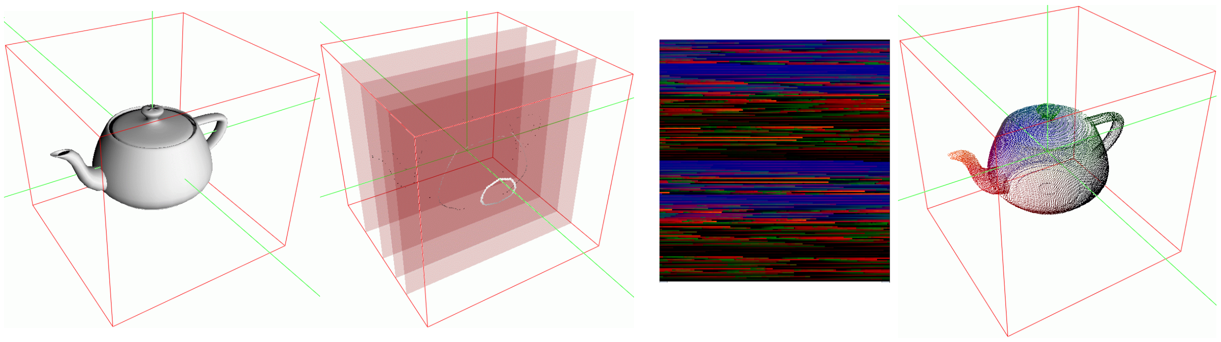


Figure 1: Dicer, our demo application, decomposing a teapot into a point cloud on the GPU, by rendering it repeatedly into slices of a 256x256x256 volume (marked in red). The volume is filled with approximately 34000 surface points.
Left to right: 3D model of teapot; volume slices at z=70/130/190/201; resulting 2D point list; particle cloud based on point list, the false colors stand for the 3D texture coordinates of the voxels.

dexing data structure. The GPU then needs $4\log_2(max(\text{size}_x, \text{size}_y))$ texture accesses to generate one point in the list (and $\log_2(max(\text{size}_x, \text{size}_y))$ for the vec4-HistoPyramid variant, see section 6).

We show that this approach runs considerably faster on the GPU than comparable hybrid CPU/GPU- or CPU-based solutions, provided that the input 2D image already resides in video memory or the point list is needed in further GPU processing and must otherwise be uploaded from the CPU. The algorithm scales with future GPU architectures and performs in a very cache-friendly way. Finally, we exemplify the algorithm's practical use in *Dicer*, a demo application which converts arbitrary 3D models into particle clouds in real-time, running completely on the GPU.

1. Related Work

Data pyramids have been used in Binary Tree Predictive Coding ([HAN85], [ROB97]). For example, a quad tree leaf can signal if all of its descendants are identical, and therefore skip the transmission of its descendants. Our algorithm uses similar ideas to skip empty regions during the HistoPyramid traversal which builds up the point list.

The data building process for the mentioned HistoPyramid is adopting the well-known parallel "reduction operation". It is applied in custom mipmapping (see also [BP04]), and processes n^2 elements in $\log_2(n)$ passes. Our summing operator builds a Laplacian pyramid of partial histograms. One variant of our algorithm uses bilinear texture interpolation to accelerate the summing operation, as already described in [NVa04], see also section 6.4.

Occlusion queries, as proposed in [NVb04], are admittedly faster for a total histogram count, as they only need one rendering pass to sum all active cells. However, we cannot take advantage of this method, as our traversal algorithm requires the partial sums produced on the way to reconstruct cell coordinates.

Bitonic merge sort, as exemplified in [GHL*04], could also be used for point isolation in sparse images by giving seed points a different sorting key than invalid points. However, since this sorting algorithm is optimized for a plentiful of key values, it runs suboptimally ($O(n (\log n)^2)$ steps) for a 2D image where only a binary partitioning is required.

Finally, [HOR05] introduces the concept of data compaction, i e filtering of unwanted data elements from a given data stream. It does this by successively producing a running sum, describing where to skip unwanted elements to obtain a packed result. The algorithm needs log(n) iterations to produce this running sum, and keeps the number of output elements constant. The implementation was based on the stream processing language Brook.

Our algorithm takes a similar approach as [HOR05], but utilizes a quad tree to represent its intermediate data, a considerable reduction in intermediate data output ([HOR05] has $\log_2(n).n$, while we produce at maximum 2n due to the pyramid). However, we have to use *all* intermediate data levels to generate the final, compacted output, not only the last level - a feature where graphics hardware architectures differ from the classical stream architecture. We further utilize both the GPU's vector processing and bilinear texture interpolation if they yield an advantage (see algorithmic variants of the tree traversal in section 6).

2. Overview

Figure 3 illustrates the workflow between the different computation steps. All data is being processed on the GPU – the CPU is only handling data if the input data originates there or if the point list shall be downloaded for further processing in a non-GPU application.

The input is a 2D image or a stack of 2D images (up to the GPU's number of texture units). The image cells may be of arbitrary type (single/RGBA, byte/float), as long as the Discriminator is able to handle cells of such type.

The **Discriminator** determines if a cell's content is regarded as active or not. It generates a binary representation of the same size as the 2D image, and sets the base level for the histogram pyramid. We will exemplify some useful discrimination operators in section 3.

The **HistoPyramid Builder** creates the Laplacian pyramid levels of histogram information. Its reduction operator repeatedly processes four input cells into one throughout the whole input image, starting at the resolution level of the original input image. It finishes when only one output cell remains. We describe its GPU implementation in section 4.

The **PointList Builder** takes the HistoPyramid, and creates an initial, empty 2D list in the form of a 2D image

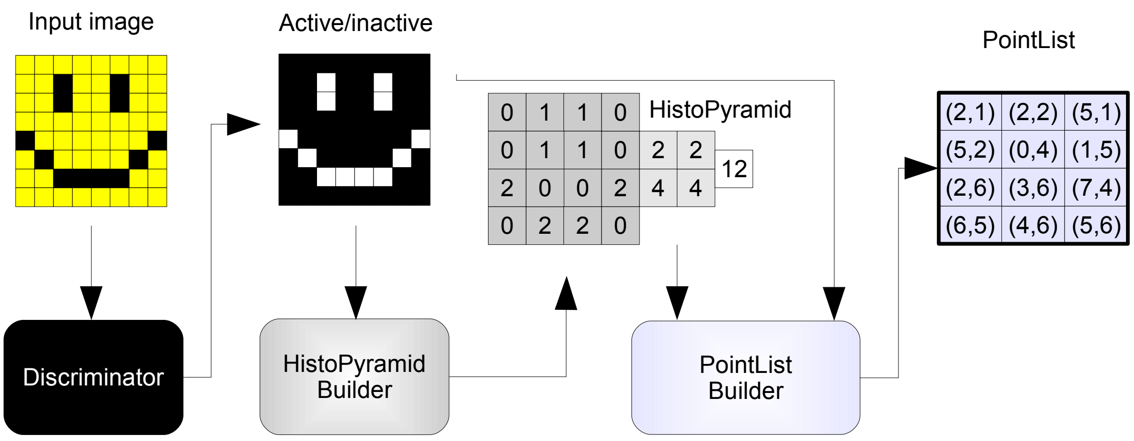


Figure 3: Overview over the internal workflow.

(based on the number of found active cells), henceforth called *point list*. Afterwards, it fills the list using a hierarchical traversal of the histogram pyramid for each list entry. Section 5 describes details of the traversal in diagrams, and presents a log of the traversal decisions. It also points out which GPU restrictions hamper performance, and how their removal might improve future implementations.

We have also devised **algorithmic variants**, including a more intricate, but faster implementation which uses the GPU's native vector capabilities, and a version which utilizes bilinear texture interpolation. A discussion of these variants can be found in section 6.

The basic, or, for CPU programmers, fairly straightforward concepts underlying our algorithm can make it hard to understand the full range of new applications that a GPU implementation opens. Therefore, section 7 details several real-time **applications** that become feasible with a GPU implementation of this algorithm.

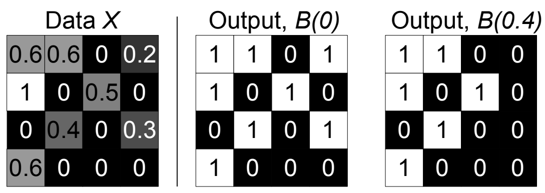
Section 8 summarizes the current performance results that we obtained by running the algorithm's variants on state-of-the-art graphics hardware. It describes the surrounding test setup, and analyzes the runtime behaviour.

3. Discriminator

As discussed, the subsequent stages operate on *binary* images, that is, each cell has to be either active (1) or not (0). Therefore, we must first preprocess our input data into binary images. Any operator that can map an image's cells into such a binary decision can be utilized here.

The most trivial operation is the *threshold operator* (Figure 5). It determines if the cell data is equal to or above a certain threshold.

Combined application of different thresholds, equal to a *range operator*, can yield point lists of „watershed cliffs“ (items that are below a certain treshold ([-inf, range$_{max}$])or within a certain range ([range$_{nub}$, range$_{max}$])). These are very useful for dominant component detection in signal analysis and data compression, as proposed in section 7. The same range discriminator lends itself to "binning" operations (and can thus be extended to a sorting algorithm, if the bins become fine-grained enough).

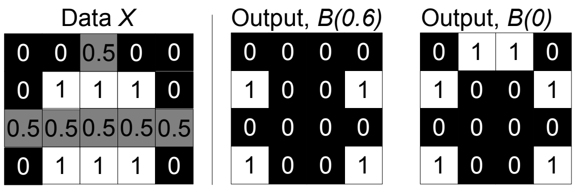


$$B_{i,j}(a)=\begin{cases}1: X_{i,j}\geq a\\ 0: X_{i,j}<a\end{cases}$$

Figure 5: Example input data and two possible outputs (top, left to right) for the thresholding function (bottom).

Many *edge detection operators* use thresholds on the spatial image gradients, which are approximated as differences between adjacent data points. One example can be seen in Figure 6. Again, it is possible to apply several thresholds to detect edges of varying intensity.

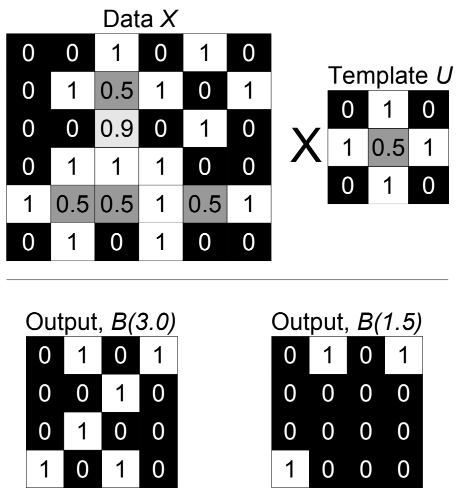
Most general is the *folding and thresholding operator*, see Figure 4. It folds a data cell region with a user-provided data template and applies thresholding to signal the match between template and input data in this particular position. This is useful for detecting a certain image template in a larger image without knowing the number of occurances in advance (otherwise a common restriction in GPU-based image processing).



$$B_{i,j}(a)=\begin{cases}1: |(X_{i+1,j}-X_{i,j})|>a\\ 0: otherwise\end{cases}$$

Figure 6: Example input data and two possible outputs (top, left to right) for the edge detection function (bottom).

Our *Dicer* demo utilizes thresholding to detect 3D model pixels that have been earmarked with `alpha=1.0`. Since thresholding is a local and inexpensive operation, it was integrated into the first stage of the HistoPyramid Builder. Note, though, that in order to detect valid pixels on the lowest level, PointList Builder will have to *redo* the thresholding operation in that case,. More complex discrimination operations should be kept separate from the next processing step to avoid redundant calculations.



$$B_{i,j}(a)=\begin{cases}1: \sum_{s=-1}^{1}\sum_{t=-1}^{1}|(X_{i+s,j+t}-U_{i+s,j+t})|\leq a\\ 0: otherwise\end{cases}$$

Figure 4: Example input data and two possible outputs (top, left to right) for the folding function (bottom).

4. HistoPyramid Builder

The HistoPyramid, short for histogram pyramid, is a Laplacian pyramid with the Discriminator's binary output as its base. On this base level, each active cell is treated as a 1, while inactive or empty cells are interpreted as 0. Our reduction operator simply sums up four underlying cells and writes the result into the prepared output level image until the final level consists of only one cell. The algorithm annotates the level of this cell (the top level) for the subsequent stages, and terminates. The output is a Laplacian stack of output images with integer content (32bit floating point in the GPU implementation), see Figure 2.

Note that only square, power-of-two image dimensions can provide the algorithm with a constant number of input and output cells. The HistoPyramid algorithm itself could easily be adapted to rectangular and non-power-of-two textures, but current GPU programmability restrictions (namely, the inability to provide explicit texture sizes for each pyramid level) would severely limit the performance of the PointList Builder. In the meanwhile, we propose to pad the input image's dimensions in order to be quadratic and a power of two.

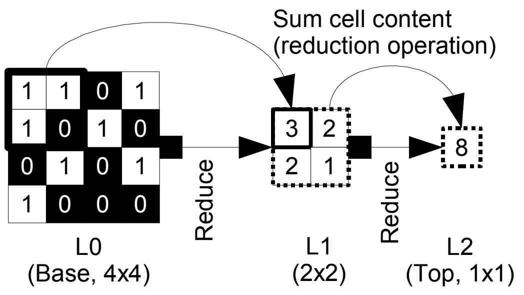


Figure 7: Basic HistoPyramid building process. L0, L1 and L2 are the pyramid levels. While generating the next pyramid level, the GPU sums four adjacent cells into one, thereby halving resolution, until only one cell remains.

5. PointList Builder

Given the HistoPyramid as input, it is now possible to determine the *number of entries* in the final list output. PointList Builder accesses the top level of the pyramid to retrieve this value, and allocates the *point list*, a 2D image whose sidelength is equal to the square root of the number of entries (see the 2D point list in Figure 8 for an example). The reason for choosing a 2D layout is that the GPU currently can handle only 4096 entries at maximum in a 1D image.

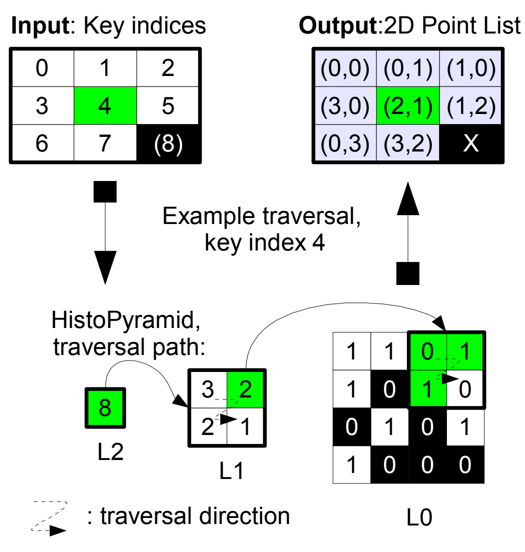
Now, actual point list reconstruction commences. In our example in Figure 8, the PointList Builder shader will now be called for all nine possible list entries in the 2D image.

The shader first determines its own index (the *key index*) from its 2D coordinate in the point list. Since it also has been given the total number of entries (the list count, here: 8), it immediately terminates if the key index exceeds the list count (such an entry is empty, it is only an artifact from the 2D image allocation - our example marks it with an **X**).

The algorithm descends one level if the key index lies within the *index range* of a HistoPyramid cell. Intuitively, the index range of a HistoPyramid cell describes the possible or *covered* range of key indices that active cells in this covered part of the 2D input image can receive. The top level's single cell poses a good example, its range covers all active cells' indices. A different way to see it is that all active cells in a given index range will be HistoPyramid or quadtree descendants of this cell.

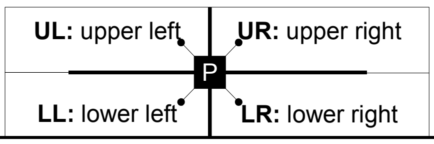
During traversal of the HistoPyramid, the current index range [**start, end**] is updated as follows:

- **start** is initialized to zero.
- **end** is assigned the sum of the cell's content (looked up from the HistoPyramid) and **start**.
- Before a new cell is examined on the same level, **start** becomes the former index range **end**.
- If we descend one level, we retain the **start** value that was active during the parent cell's range check.



0	1	2
3	4	5
6	7	(8)

(0,0)	(0,1)	(1,0)
(3,0)	(2,1)	(1,2)
(0,3)	(3,2)	X



(2D list size: (plw, pwh) = 3 x 3)

PointList Builder, run at (px, py)=(1,1):
- Key index: ki = py * plw + px = 4
- L2: Index Range [0,8[: **fits key index !**
- (descend to L1, retain start=0)
- L1, cell UL: Index Range [0,3[
- L1, cell UR: Index Range [3,5[:**fits key index !**
- (descend to L0, retain start=3)
- L0, cell UL: empty
- L0, cell UR: Index Range [3,4[
- L0, cell LL: Index Range [4,5[: **fits key index !**
- **Result: (2,3)**

Figure 8: PointList Builder's internal data traversal for an example key index. Left: graphical illustration. Right, top: naming convention for lookup directions, as seen from a parent cell O. Right bottom: Algorithm's log on made decisions.

Note that the traversal order is irrelevant as no sorting is enforced; it only needs to be the *same* order for all point list entries to avoid doublettes, but that is fulfilled as the PointList Builder shader is the same for all pixels.

This repeats until the base level has been reached. There, the final target cell can be chosen after the same index range criterion, if we interpret an active cell as a value of 1. The found target cell's coordinates are written into the point list output image.

The final result is thus a 2D image containing point list entries of all active cells in the image, the *point list*. PointList Builder assigns a *unique* active cell to each index, but the indexing order is somewhat unintuitive (based on a fractal traversal pattern). Optionally, the algorithm can provide line-wise indexing if a line-wise CPU traversal is desired. In that case, the reduction operator takes four horizontal cells instead of a square of 2x2. However, this would probably hamper texture caching performance during the HistoPyramid construction.

6. Algorithmic variants

6.1. Merged Discriminator and HistoPyramid Builder

For simple thresholding, the Discrimination Operator and the HistogramPyramid Builder are usually simple and can thus be *fusioned*. In that case, we use a shader that behaves differently on the base level of HistoPyramid, and apply the discrimination operator to detect active cells during the build process. This saves storage space and calculation time, since the discrimination results never have to be written to video memory. It should be noted, though, that PointList Builder has to *redo* these operations on the base level to determine if it has found the correct target cell. Therefore, it is only advisable to use this variant if the discrimination operator's calculation costs are negligible in comparison to writing and re-reading the binary image.

6.2. Point list entry cloning

If more output data shall be added in the point list (e.g.to create multiple vertices from one point), we propose *„stretching“ the key index* (applying a modulo in the key index calculation) so that the same key indexing result will be written to *several* point list entries. This way, a point list can e.g. serve as vertex list for quads centered around each discovered cell coordinate, by cloning each list entry four times while the point list is generated. We thus generate a certain number of vertices for each active cell in the input.

6.3. Faster traversal with partial sums in vec4

This variant makes use of the GPU's vector capabilities. As it is capable of manipulating four float values in each target cell, we store the *partial sums of the leaf cells* in the parent cell, instead of only the overall sum (see Figure 9).

This way, it is not necessary to do four texture lookups (in the leaf cells) to decide in which quadtree branch to descend. Instead, this decision can already be made based on the partial sums in the level above. The algorithm can thus save up to three texture lookups for every traversed HistoPyramid level. We call this the *vec4-HistoPyramid*.

6.4. Bilinear interpolation for faster summation

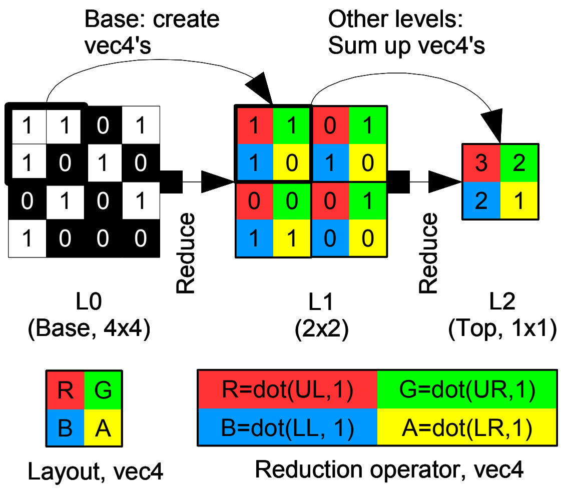


Figure 9: dicer_vec4's extended RGBA storage and the accordingly modified reduction operator.

A third variant uses the GPU's bilinear texture interpolation. Our *interpolation-based HistoPyramid Builder* places a texture lookup exactly in the middle between the four input cells of the level below, which makes texture interpolation return the *average* of the four input cell values. A multiply with four yields the *sum*. This is faster than calculating the sum from four explicit texture lookups in the shader, since graphics hardware contains special data paths for such interpolated lookups. Unfortunately, current hardware restricts such interpolation to 16 bit float values. As 16 bit floats are not enough to represent more than 32768 points, we have devised a way to split a 20 bit integer into two 16 bit floats. But such a splitting requires constant rebalancing of the interpolated values in the HistoPyramid Builder, and an extra dot product to reconstruct cell values in PointList Builder. Also note that interpolation cannot easily be combined with the *vec4-HistoPyramid* from section 6.3.

7. Applications

7.1. Image analysis

The most promising application for this algorithm is GPU-accelerated image processing. We are confident that this algorithm paves the trail for efficiently solving computer vision problems solely on graphics hardware. As an example, GPUs can now *analyze the folding result* after conducting the actual folding, and thus augment the algorithms proposed in [FM05]. The expensive download of half-processed image data can thus be avoided, and only the discovered feature point set needs to be transferred.

7.2. Volume analysis

We have currently only described how to analyze 2D images, but the algorithm can straightforwardly be extended to the 3D case if the HistoPyramid became a *hierarchy of 3D volumes*. Unfortunately, current render-to-texture functionality can only write to one 2D slice at a time (if at all: NVidia drivers currently do not support that), which slows down performance due to framebuffer setup times. In the meanwhile, we suggest to lay out 3D volumes in a 2D texture, in the same way as *Dicer* demonstrates, and to reconstruct 3D texture coordinates from the found 2D image coordinates. As a proof of concept, we have recently used this algorithm for detecting seed points in 3D flow data on the GPU. We are very confident that other applications, such as level-set identification, are possible. Even a GPU based marching cubes algorithm is within reach, provided that the algorithm can generate geometry after the relevant voxel/mesh mappings have been identified.

7.3. Sparse matrix creation

[KW03] has demonstrated how to process large sparse matrices by packing them into a special representation. Up to now, it was not possible to *create* such sparse matrix representation *on the graphics hardware*,. Our algorithm could be used to convert matrices into such sparse matrix representations, and thus save memory and computation time.

7.4. Quadtree Builder

Many simulation problems deal with processing data of varying sample density (e.g. fluid simulations, [HAR04], or [KW03]). Also, compression and encoding often require the clustering of similar regions. If HistoPyramid Builder is modified to count the largest-size regions of common cell values, and to mark at which level they are found in the hierarchy, then PointList Builder can output a quadtree whose leafs terminate at the level where only identical values remain. This way, computation and storage could adapt to sample density, in much the same spirit as sparse matrix computations do not waste resources on empty regions.

8. Results

In order to test the real-time behaviour of our algorithm we implemented *Dicer*, a small Linux application able to convert 3D models to point clouds. The 3D model, a teapot generated with `glutTeapot(0.6)`, is stored in a display list to maximize geometry throughput. The software slices the mesh by rendering it into 256 2D slices of 256x256 each, spanning a volume of [-1,-1,-1] to [1,1,1] in world space (see also Figure 1). The output is put into 16 x 16 tiles of an 8-bit RGBA texture at 4096x4096 resolution. Valid pixels belonging to the 3D model are marked with `alpha=1.0`. Additionally, we experiment with smaller texture sizes to measure the performance scaling, effectively producing volumes of 256x128x128 (2048x2048) and 256x64x64 (1024x1024).

After slicing, the algorithm analyzes the resulting 2D texture and retrieves the list of actually occupied voxels. Typically, it finds around 33000 points, and renders them as a particle cloud.

The tests were conducted on a Dell Precision M70 laptop with Nvidia Quadro FX Go 1400 and 256 MB video memory, connected over PCI Express. It contained an Intel Pentium M (2.13 Ghz) and 2 GB of main memory. The AGP download timings came from an Athlon XP2400 system with an Nvidia GForce 6600, AGP 8x. We compare four variants of the algorithm:

dicer_single is the most classic implementation, and follows the basic algorithm as described in Figure 8. It uses the OpenGL texture format GL_TEXTURE_2D, which provides real mipmap levels and render-to-texture, but current restrictions force it to build the HistoPyramid in a 32 bit-float RGBA texture, even though only one data channel is used.

dicer_vec4 is similar, but makes better use of the four 32-bit components by storing partial sums in the RGBA vec4, effectively delaying the cell sum-up by one level (see Figure 9 and section 6.3). This accelerates PointList Builder, as the tree traversal has do to less texture lookups to make its branching decisions.

dicer_rect utilizes GL_TEXTURE_RECTANGLE, a texture format with no mipmap levels - but render-to-texture allows 32-bit single float textures here, which saves considerable amounts of memory. Since PointList Builder needs to access all levels in one pass, we were forced to create a pseudo-mipmap layout in a single texture (see Figure 10).

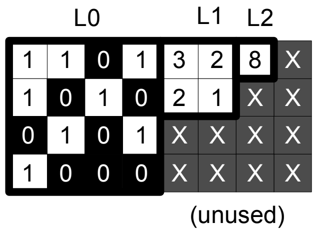


Figure 10: dicer_rect's pseudo-mipmap layout for a single, rectangular texture without mipmap capability.

dicer_bil is similar to dicer_single, but uses *bilinear texture interpolation* to accelerate the HistoPyramid construction, as proposed in section 6.4. Unfortunately, the algorithm proved to be numerically unstable. It could not faithfully reproduce a complete list of active cells in a test image, and was thus skipped in evaluation. Instead, we wait for the introduction of ShaderModel 4.0 based graphics cards to verify this algorithmic variant with the forthcoming bilinear texture interpolation for 32-bit float values.

Table 1 lists the timings common to all implementations. Volume slicing (the conversion of the 3D model to the 2D volume texture) dominates all timings, presumably due to the repeated geometry processing.

Instead of taking the time consumed by the OpenGL call delays on the CPU side, we measured actual GPU timings with the GL_EXT_timer_query extension ([NVc04]).

For Table 2, we implemented a classic CPU loop to compare our GPU algorithm with a standard single-thread implementation. Here, the CPU downloads the 2D volume texture as RGBA8 (over AGP or PCI express, depending on the system), generates an output list after line-wise traversal of the texture, and uploads it to the GPU again. Aggressive compiler optimization accelerates the CPU based analysis, but we did not employ SIMD techniques. CPU timings were taken as virtual process time measurement by the `getitimer()` function of Linux systems. It is clear that for large textures, the texture download greatly outweighs the actual analysis.

Table 3 shows the time spent on the GPU for creating the HistoPyramid. Both dicer_single and dicer_vec4 suffer heavily from the restriction to RGBA, 32-bit float textures: The texture data is obviously being swapped to main memory, causing large performance penalties for 4096x4096. dicer_rect can use single-component 32-bit float textures, and therefore scales as expected. But even without memory

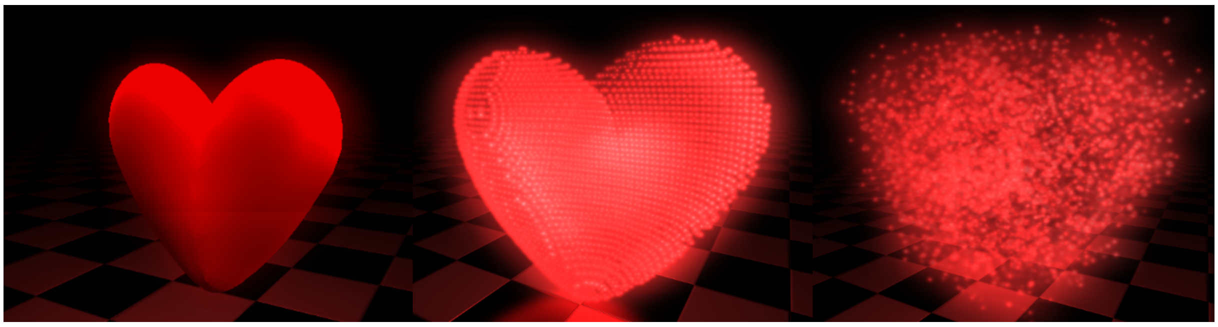


Figure 11: Screenshots from our FX demo *HeartBreaker*. From left to right: Solid model (5000 triangles); Point cloud representation (1872 points, in a 256x64x64 grid, first iteration: 90 ms, subsequently: 25 ms); Particle explosion effect.

restrictions, it shows that single-component render-to-texture is considerably faster than RGBA render-to-texture.

Constants	4096x4096	2048x2048	1024x1024
# of active cells	33989	8595	2130
Volume slicing	470 ms	470 ms	470 ms

Table 1: Common timings for all implementations.

Constants	4096x4096	2048x2048	1024x1024
AGP download	560 ms	142 ms	36 ms
PCIe download	172 ms	40 ms	12 ms
CPU traversal	25 ms	25 ms	24 ms

Table 2: CPU algorithm timings.

HistoPyramid	4096x4096	2048x2048	1024x1024
dicer_single	~2000 ms	20 ms	6 ms
dicer_vec4	~2000 ms	20 ms	6 ms
dicer_rect	30 ms	10 ms	2 ms

Table 3: HistoPyramid creation timings.

PointList	4096x4096	2048x2048	1024x1024
dicer_single	16 ms	12 ms	~6 ms
dicer_vec4	14 ms	7 ms	~6 ms
dicer_rect	9 ms	6 ms	~2 ms

Table 4: Point list creation timings.

Finally, Table 4 documents the timings of point list creation. Here, results are more comparable, and dicer_vec4 can outperform dicer_single due to its improved traversal algorithm. dicer_rect *outperforms both*, however, and as soon as dicer_single is able to render to single-component textures, it will probably also be in the same speed ranking. Therefore, additional tests are required to verify the gain of dicer_vec4's ncreased storage and bandwidth consumption for volume analysis. The situation can be different for binning and sorting operations, where the whole volume of data needs to be rearranged and no data will be thrown away - but we lacked such a test situation.

After summing up the timings from Table 3 and Table 4 and comparing it with Table 2, we are now confident that GPU-based image/volume analysis has become competitive with the help of HistoPyramids. The speed advantages are only small for medium-sized textures, but for large textures, the impact for CPU texture download is so profound that it pays off to keep the data on the GPU. Further, it saves both memory and CPU time to let the GPU process data that already resides there (like volume slicing results).

Our second demo *HeartBreaker* uses a similar approach, but downloads the final particle cloud to the CPU. The presented animations, shown in Figure 11, demonstrate how our method can deliver new and unusual visual effects, such as particle explosions of arbitrary geometry models.

9. Conclusions and Outlook

We have presented a novel, fast and easy-to-use GPU algorithm for rapidly generating point lists. The possible applications for this technology are numerous, ranging from GPU-based computer vision applications, such as 2D image analysis or feature detection, to 3D volume processing, such as occupancy testing or seed point selection. Also, impressive visual effects for the development of computer games, such as the rapid conversion of arbitrary geometry into particle clouds have now become feasible.

Through experiments we have shown that our purely GPU-based implementation is significantly faster than a hybrid GPU/CPU implementation.

Our algorithm and its variants are currently restricted by limitations of the graphics hardware and its driver. We are therefore looking forward to single-component render-to-texture for OpenGL GL_TEXTURE_2D and await eagerly the advent of Shader Model 4.0-capable graphics hardware in order to test how 32-bit float interpolation and integer handling can improve performance. This way, we hope to make dicer_rect obsolete, as its pseudo-mipmap handling is rather complicated and would hamper wide-spread use of this algorithm on the whole. We would also like to test render-to-texture for 3D textures in the future, as 3D HistoPyramid traversal would cache more efficiently, and trilinear texture lookups accelerate the HistoPyramid building process. Finally, we are curious on how geometry shaders under Shader Model 4.0 compare to the presented algorithm.

Despite current limitations, we have presented a versatile algorithm with a multitude of applications. It will be highly interesting to see how it maps into image analysis, data compression and general purpose computation. In general, there should now only be few computational tasks left that can *not* be done on GPUs.

References

[BP04]: Buck, I., and T. Purcell: A Toolkit for Computation on GPUs. *GPU Gems*, pp.621-636,2004.

[BFG*04]: Bolz, J., Farmer, I., Grinspun, E., and Schröder, P.: Sparse matrix solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Transactions on Graphics 22*, pp. 917-924, 2003.

[GHL*04] Govindaraju N., Henson M., Lin M. and Manocha D.: Computations among Geometric Primitives in Complex Environments. *Proc. ACM Symposium on Interactive 3D Graphics and Games,* 2005.

[HAN97]: Hanan, S.: Data structures for quadtree approximation and compression. *Communications of the ACM*, Volume 28, Issue 9, pp. 973-993, 1985.

[HAR04]: Harris, M.: Fast Fluid Dynamics Simulation on the GPU. *GPU Gems*, pp.637-665,2004.

[HOR05] Horn, D.: Stream Reduction Operations for GPGPU applications. *GPU Gems 2*, pp. 621-636, Addison-Wesley.

[KW03] Krüger, J. and Westermann, R.: Linear algebra operators for GPU implementation of numerical algorithms. *Proc. ACM SIGGRAPH 2003,* pp. 908-916, 2003.

[NVa04]: Simon Green, NVidia Corp.: OpenGL Image Processing Tricks. *GDC 2005 Presentations*, 2005. **http://tinyurl.com/f59r4**

[NVb04]: NVidia Corp.: Image Histogram. *SDK Code Samples - Video and Image Processing*, 2004. **http://tinyurl.com/kmlr2**

[NVc04]: Simon Green, NVidia Corp.: NVidia OpenGL Update. *GDC 2005 Presentations*, 2005, pp. 40-42. **http://tinyurl.com/pngld**

[ROB97]: J A Robinson, Efficient General-Purpose Image Compression with Binary Tree Predictive Coding. *IEEE Transactions on Image Processing*, Vol 6, No 4, April 1997, pp 601-607.

[FM05]: Fung J., and Mann S.: OpenVIDIA: parallel GPU computer vision. *Proc. 13th annual ACM international conference on Multimedia*, 2005, pp. 849 - 852. **http://openvidia.sf.net**

MPI-I-2006-RG1-001	C. Weidenbach, T. Hirth, C. Karl	Automatic Infrastructure for ..... Analysis
MPI-I-2006-5-004	F. Suchanek, G. Ifrim, G. Weikum	Combining Linguistic and Statistical Analysis to Extract Relations from Web Documents
MPI-I-2006-5-003	V. Scholz, M. Magnor	Garment Texture Editing in Monocular Video Sequences based on Color-Coded Printing Patterns
MPI-I-2006-5-002	H. Bast, D. Majumdar, R. Schenkel, M. Theobald, G. Weikum	IO-Top-k: Index-access Optimized Top-k Query Processing
MPI-I-2006-5-001	M. Bender, S. Michel, G. Weikum, P. Triantafilou	Overlap-Aware Global df Estimation in Distributed Information Retrieval Systems
MPI-I-2006-4-007	O. Schall, A. Belyaev, H. Seidel	Feature-preserving Non-local Denoising of Static and Time-varying Range Data
MPI-I-2006-4-006	C. Theobalt, N. Ahmed, H. Lensch, M. Magnor, H. Seidel	Enhanced Dynamic Reflectometry for Relightable Free-Viewpoint Video
MPI-I-2006-4-005	S. Yoshizawa	?
MPI-I-2006-4-004	V. Havran, R. Herzog, H. Seidel	On Fast Construction of Spatial Hierarchies for Ray Tracing
MPI-I-2006-4-003	E. de Aguiar, R. Zayer, C. Theobalt, M. Magnor, H. Seidel	A Framework for Natural Animation of Digitized Models
MPI-I-2006-4-002	G. Ziegler, A. Tevs, C. Theobalt, H. Seidel	GPU Point List Generation through Histogram Pyramids
MPI-I-2006-4-001	R. Mantiuk	?
MPI-I-2006-2-001	T. Wies, V. Kuncak, K. Zee, A. Podelski, M. Rinard	On Verifying Complex Properties using Symbolic Shape Analysis
MPI-I-2006-1-007	I. Weber	?
MPI-I-2006-1-006	M. Kerber	Division-Free Computation of Subresultants Using Bezout Matrices
MPI-I-2006-1-005	I. Albrecht	?
MPI-I-2006-1-004	E. de Aguiar	?
MPI-I-2006-1-001	M. Dimitrios	?
MPI-I-2005-5-002	S. Siersdorfer, G. Weikum	Automated Retraining Methods for Document Classification and their Parameter Tuning
MPI-I-2005-4-006	C. Fuchs, M. Goesele, T. Chen, H. Seidel	An Emperical Model for Heterogeneous Translucent Objects
MPI-I-2005-4-005	G. Krawczyk, M. Goesele, H. Seidel	Photometric Calibration of High Dynamic Range Cameras
MPI-I-2005-4-004	C. Theobalt, N. Ahmed, E. De Aguiar, G. Ziegler, H. Lensch, M.A.,. Magnor, H. Seidel	Joint Motion and Reflectance Capture for Creating Relightable 3D Videos
MPI-I-2005-4-003	T. Langer, A.G. Belyaev, H. Seidel	Analysis and Design of Discrete Normals and Curvatures
MPI-I-2005-4-002	O. Schall, A. Belyaev, H. Seidel	Sparse Meshing of Uncertain and Noisy Surface Scattered Data