A Genetic Texture Packing Algorithm on a Graphical Processing Unit

Karsten Kaul[1] **Christian-A. Bohn**[2]

Wedel University of Applied Sciences
Feldstr. 143, 22880 Wedel, FR Germany
E-Mail: [1]k.arts@gmx.net, [2]bo@fh-wedel.de

Abstract

Interactively modeled virtual scenes usually contain hundreds of single separate textures. A common task to prepare these scenes for real-time rendering is to compile the set of many textures into one large texture which is better suited to be handled by modern graphics hardware.

We present an approach for accomplishing the above task automatically. First, a patchwork of near-optimal compactness is calculated through a *genetic algorithm (GA)*, second — since "GA are slow" — we implement the genetic algorithm on a GPU and show that it easily outperforms a standard CPU implementation.

Keyword: Computer Graphics, Genetic Algorithms, GPU programming, Textures, Real-Time Rendering, Interactive Modeling

1 Introduction

In todays graphics applications *textures* — images glued on polygons for imitating surface structure which then does not have to be modeled explicitely — are heavily used to enhance the look of virtual objects.

To enable real-time applications and concurrently using vast amounts of textured surfaces on modern *graphical processing units (GPU)* it is a good idea to avoid feeding the GPU with *many separate* textures but instead using a patchwork of them packed into *one single* large texture. Switching between separate textures is then accomplished by changing texture coordinates such that vertices "point" on the according locations in the patchwork.

Creating an *efficient* patchwork means positioning and rotating the set of original textures in way that they occupy the smallest possible rectangular area. This task is strongly related to the commonly known *knapsack problem* from the field of combinatorial optimization. Since it is *NP-complete* — hardly to solve by deterministic approaches — we decide to use an iterative *genetic optimization* algorithm. Genetic algorithms are very convenient, easy to implement and quite efficient concerning the quality of the solution — they mostly end up close to a global optimum. Nevertheless, they need huge computing resources. This together with the fact that they are typically inherently parallel algorithms lead us to the development of an algorithm which utilizes modern computer graphics accelerator hardware (GPUs).

The description of this work is split as follows.

First, by adapting a genetic algorithm in section 2 we propose a new idea for compiling a compact texture patchwork. Genetic operations are described in detail.

Second, the genetic algorithm is imple-

mented on a GPU's parallel hardware which is described in section 3. Here, we propose the separation of the genetic algorithm into several GPU programs and the triggering mechanism of the main routine running partly in the CPU.

Finally, we evaluate this approach through a results section and conclude in a summary.

2 Genetic Packing Algorithm

2.1 Overview of Genetic Algorithms

Genetic algorithms are a useful tool for iteratively finding a local, near-optimal solution within an arbitrary search space [1]. They are usually applied to search spaces of vast dimensions where it is hardly possible to find the global optimum due to the huge complexity of deterministic search algorithms.

Consider a problem and its optimal solution like an n-dimensional search space containing all solutions and an n-dimensional point in this search space. A genetic algorithm randomly selects initial points from the search space at the start. Then, these points are reused — parts of the points' components are randomly combined to new coordinates in the search space. Seeing points like strings of coordinate components, parts of the strings are split off and recombined to new coordinates. By terming a set of points as "population", each point as "individual", coordinate strings as "genome" and the algorithm which chops strings up composing pieces into new ones as "*Mendel's Genetic Laws*" we receive a mind model for the process of biological evolution. Since biological evolution mostly generates near-optimal creatures, genetic algorithms (which emulate this evolutionary process) are expected to find similar good solutions to similar complex problems.

To find a suitable coding of the problem under consideration as a string-like data structure similar to a genome string is the vital challenge in genetic algorithms.

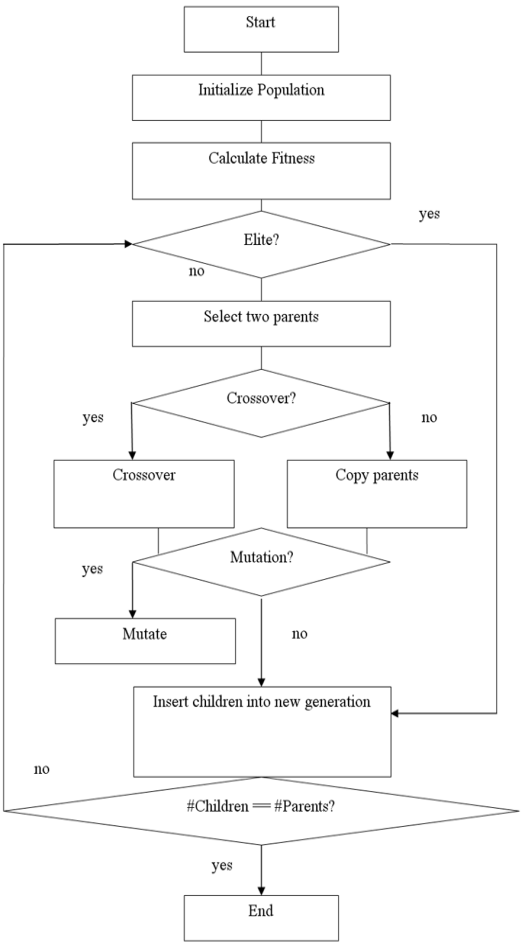

Figure 1: Typical work flow chart for processing one generation of a genetic algorithm.

Figure 1 shows the typical work flow of a genetic algorithm. The scheme illustrates the processing of one generation. First, the population is initialized with random values, then the *fitness* of each individual is computed. The fitness measures the quality of the solution. If the fitness of an individual matches some criteria it may directly be put into a new (*elite-*) generation. Two individuals are selected depending on their fitness values to create two children. During this *crossover* operation the children retrieve parts from their parents' genome string to form a new genome. To realize genetic *mutation* in the copy phase some *gene* values are randomly changed. As soon as the number of children equals the number of parents, the process for one generation

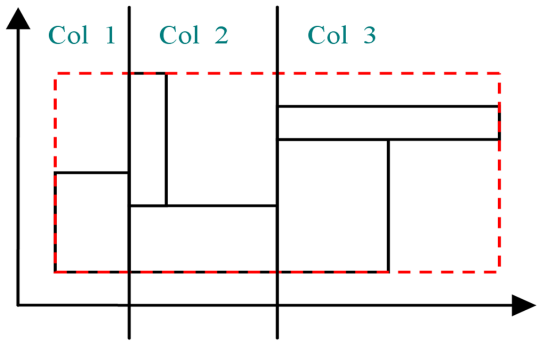

Figure 2: Textures are ordered in columns. The width of one column is determined by the width of the widest texture in this column. The goal texture dimension is represented by the dotted rectangle.

is finished. The whole process of creating subsequent generations stops when either an adequate optimum is reached or a defined number of generations has been calculated. For a detailed description of genetic algorithms we recommend [1].

2.2 Genetic Search for a Good Patchwork

Texture objects are taken as simple rectangular images. Our problem now is to find the smallest possible *goal texture (GT)* which completely holds the set of our smaller original textures. The textures on the GT must not overlap each other or overhang the GT.

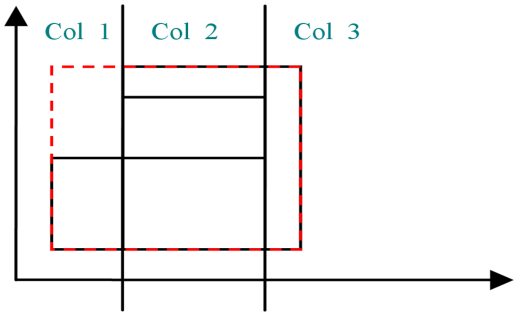

Figure 3: Textures from Figure 2 are ordered and rotated differently in a way that the area they occupy is minimized.

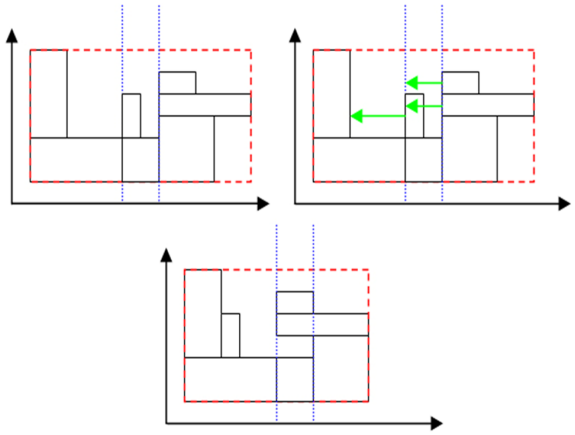

Figure 4: Textures are shifted to the left to further decrease the area of the goal texture.

Search space

We order our n textures, T_i, $i = 1...n$ through virtual columns like Figure 2 shows. The height H_{GT} of the GT equals the height of the highest column

$$H_{GT} = max(HC_k), \text{with } k < m \quad \text{and } HC_k = \sum_{l=1}^{n_k} H_l.$$

HC_k denotes the height of column k out of m columns. n_k is the number of textures in a column k and H_l is the height of the base texture l. The width W_{GT} is the sum of the maximum width of the textures in each column,

$$W_{GT} = \sum_{k=1}^{m} WC_k, \text{with } WC_k = \sum_{l=1}^{n_k} W_l$$

where W_l denotes the width of a texture T_l.

To minimize the space which the texture-objects occupy, they may be rotated and positioned in arbitrary columns (see Figure 3). To further minimize the GT's area the textures may be shifted one column to the left as Figure 4 shows.

Despite this limiting of the degree of freedom when moving through the search space, in almost all cases the GA finds acceptable

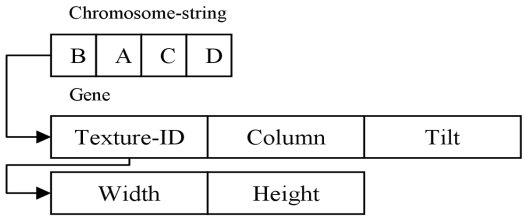

Figure 5: Structure of the whole genome and one example gene. Width and height of a texture are not stored within each gene.

results. We choose these operations intuitively to significantly reduce the dimension of the search space.

Encoding search space

The individual from the preceeding paragraph has now to be encoded into a string-like data structure — a genome (see Figure 5). Every genome contains several *genes* — all of which are representatives of a single base texture. Each gene stores a reference number of a certain texture, the column in which this texture resides and a flag marking that the texture is tilted compared to a fixed initial orientation. Each genome's length is identical to the number of base textures.

Traveling through search space

Searching is done by randomizing the parameters of the genes and letting the genomes split, mutate, and crossover according to certain evolutionary rules from general genetic learning. Due to the number of parameters the search space is untypically large compared to common problems from genetic algorithm literature like the *Traveling Salesman* problem [1].

3 Genetic Algorithm on a GPU

3.1 Overview of GPUs

The graphical capabilities of modern computer hardware are crucial due to the high demand in todays graphics applications (i.e., *computer games*) resulting from a strong consumer market demand. This demand led to a stronger focus on the development of hardware for specific graphical operations (*graphical processing units (GPUs)*) instead of hardware for general purpose computational operations.

On the one hand, it is quite obvious that nowadays GPUs outperform general purpose hardware if used for pure graphical operations, on the other hand, it is amazing that under certain conditions GPUs are also faster in executing non-graphics applications. One of these conditions is the inherently parallel nature of the algorithm to be implemented.

The above has been proven by several researchers. [2] presented a solution for the raytracing algorithm. Physical simulations using GPUs have been realized by [3]. Moving further apart from graphics leads to pure linear algebra [4], robot motion planning [5], cryptography [6], and also neural networks [7, 8].

3.2 Porting GA to GPU

Overview

Like many numerical problems put on GPUs also genetic algorithms are not suitable to be compiled as a whole on the GPU. Instead we break the problem down to different subproblems to be solved separately on the GPU. In our approach crossover, mutation, and the fitness calculation are these separate parts. The CPU concatenates these steps by transferring results from one to the next and then triggering the execution of these single steps.

The GPU implements *one point crossover* and standard *random mutation*. Greedy variants for crossover and mutation do not

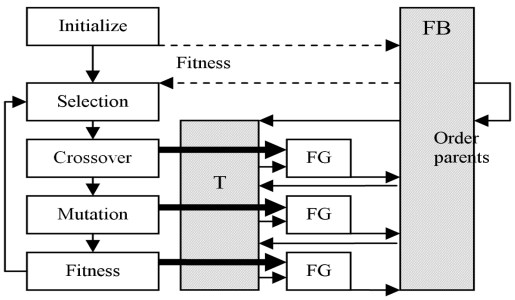

Figure 6: General work flow chart of the GPU implementation. Abbreviations are framebuffer (FB), texture (T), fragment shader (FG).

suit the parallel architecture of a GPU. The GPU approach does not left-shift texture-objects since it is not possible to get better results than when using the CPU.

During execution, all needed data resides in the *framebuffer (FB)*, i.e., it is not transferred to the CPU. Instead it is read back into texture memory from where it can be reused for the operations in following steps.

General work flow

The general work flow is shown in Figure 6. In an initializing step all individuals, their initial fitness values, and the widths and heights of the texture-objects are written into the FB. Then, the calculation of the next generation is realized by reading the fitness values from the FB, computing the choice of parents, reordering them, calculating mutation and crossover, and finally determining new fitness values. In order to accomplish these steps, several times the FB-data must be written into textures for a later reuse by the fragment shaders (FG).

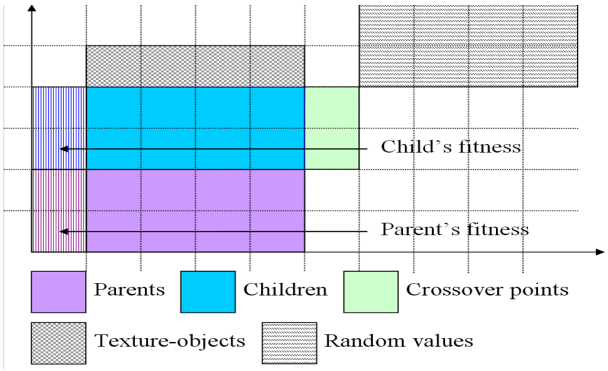

Figure 7: Framebuffer data structure, shown as a grid of single pixels. Individuals' chromosome strings are ordered in horizontal manner. Texture data for one texture and genes are stored within one pixel.

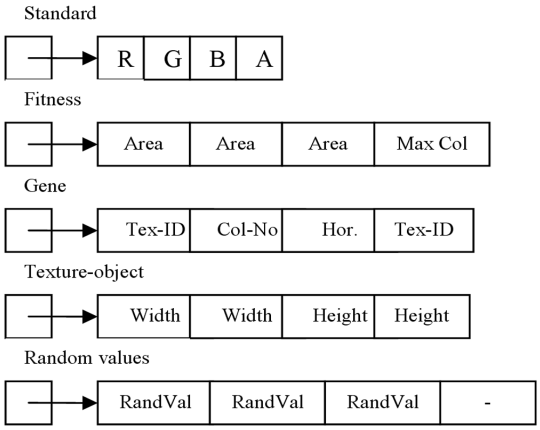

Figure 8: Different pixel data formats using RGBA texture format. "Tex-ID" is a reference to a texture. "RandVal" stores a random value (between 0-255). "Col-No" refers to a column number. "Hor" is set when textures should be rotated. "Max. Col" refers to the number of columns a solution possesses.

Framebuffer as temporary data storage

The Framebuffer is the global memory for the GPU parts of the algorithm. It is partitioned in order to hold all the different data types needed. In Figure 7 there are four individuals, two parents and children respectively. The crossover points are stored next to the children's chromosome strings.

Depending on the location on the FB the pixel data represents different information (see Figure 8). The FB data format is RGBA, whereas each color component holds 8 bit. The fitness pixels store the area that the GT occupies within three bytes of memory. The widths and heights of the texture-objects are each stored within 2 bytes to

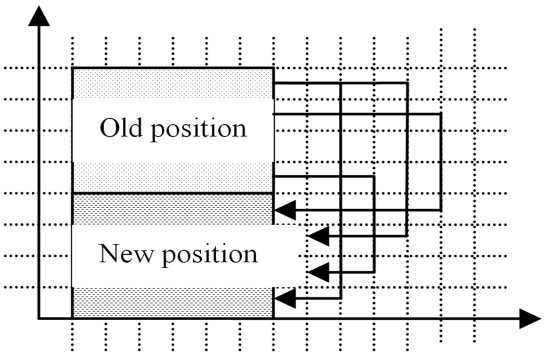

Figure 9: Copying parents to their new positions, e.g. parents pairs 1+4, 1+2. Parent 3 will be deleted.

allow for bigger texture sizes than just 255 × 255.

For mutation three random values are needed, which are stored in the RGB part of a pixel. A gene stores the texture-id by one byte supporting a maximum of 255 texture-objects.

Shader main program

The GPU operates through a *fragment shader* program on the FB. Practically spoken, pixels are drawn to the FB and in advance to blending the source and the destination, a *fragment program* is executed for each fragment to be drawn. The main program has to lock a certain region of the framebuffer for these purposes. All operations are generally realized by reading the FB into a texture, and drawing with that texture again into the FB. Thus, a shader program is executed once by each pixel-

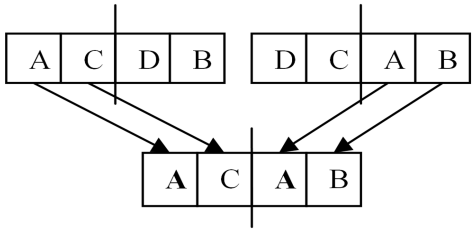

Figure 10: One Point Crossover without protection. Gene A is double occupied.

blend operation. Operands mostly are textures and the FB receives intermediate results which can be reused by reading it back into a texture.

3.3 Realization of the Genetic Operations

Best fit

Parent selection is computed on the CPU using *tournament selection* [1] where any of the possible parents compete against each other and the one with the best fitness is selected. Tournament selection is fast and its focus on the fitness value is not too strong. This avoids yielding local minima in search space. Once chosen parents are reordered in the FB by the CPU (see Figure 9). Superfluous parents get overwritten. Since the parents of a new generation are the children of the last generation, they initially reside in the child-section.

Crossover

Crossover is the crucial part of the process. Many of our first attempts in developing this algorithm delivered poor results concerning execution speed. Thus, we introduce a new approach capable of outperforming the CPU as follows.

Consider the texture-ids in each individual's chromosome string being unique and a solution provides each texture-id once. Now, simple crossover — like being used for binary chromosome strings — is not reasonable (see Figure 10). To address this problem, first all genes to be set are tested if they already exist in the child's solution. This can hardly be accomplished efficiently with the GPU. Here, a CPU attempt is faster.

For no left-shifting in the GPU approach, an easier and more efficient solution can be implemented. While in case of left-shifting the sequence of texture-ids is important in a chromosome-string, this is irrelevant for not left-shifting (see Figure 11) — thus they can simply be ordered by the texture-id. In this case, a simple crossover approach can be used without information loss (see Fig-

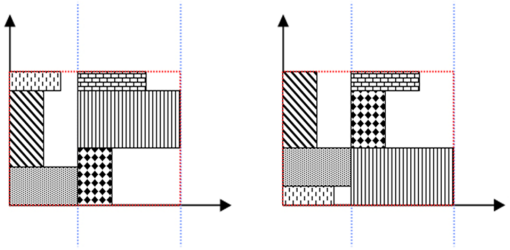

Figure 11: Different order of textures within a column, providing the same sized large texture.

ure 12) allowing for a better overall performance of the GPU approach. A shader program just needs to read the crossover points from the FB (using a texture) and to determine the parent from which a gene has to be taken for writing it into the child's chromosome string.

Mutation

During mutation, the horizontal flag might be set and the column in which the texture resides might be changed.

To determine if mutation of each of these two pieces of data should occur two random values are needed. To choose a new column in the case of mutation, a third random value is needed. These random values are stored in every pixel in the appropriate section in the FB (see Figure 7).

A shader reads the random values and resets the child's gene in case of mutation.

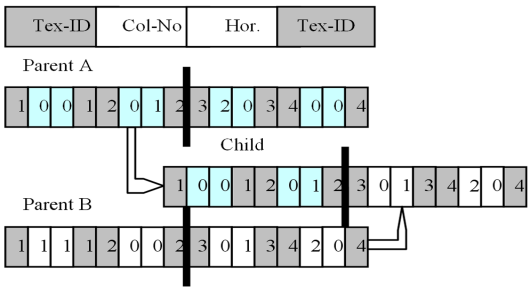

Figure 12: Texture-id's are set in ascending order.

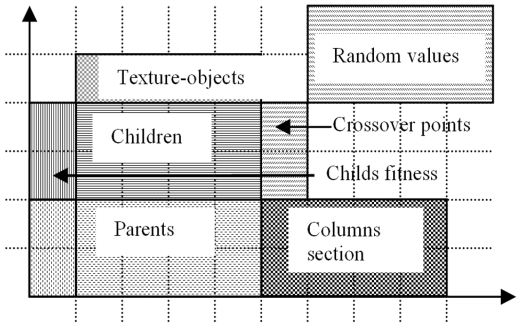

Figure 13: The column calculation section is set next to the parents section in the framebuffer.

Setting the column has to be restricted to ensure that there are only few empty columns. Although empty columns do not affect the size of the GT (they are ignored) they may yield to flat GTs which might indicate a poor solution. Thus the randomly chosen column number is restricted to being not greater than the actual maximum column obtained from the associated fitness pixel (see Figure 8).

The random values in the FB have to be generated in every generation cycle.

Fitness

While a naive shader approach is only capable of calculating the fitness for very few texture-objects, this approach allows for computing the fitness of more than 255 textures.

Due to shader restrictions, parts of the calculation have to be evacuated into the FB (see Figure 13), two shader programs are needed for the computation. The first shader program calculates the widths and heights of all columns of a solution and stores these values in the columns section. The second shader program then calculates the size of the GT using the columns' widths and heights computed by the first shader. The shader also stores the maximum column number in the "fitness-pixel" needed by the mutation.

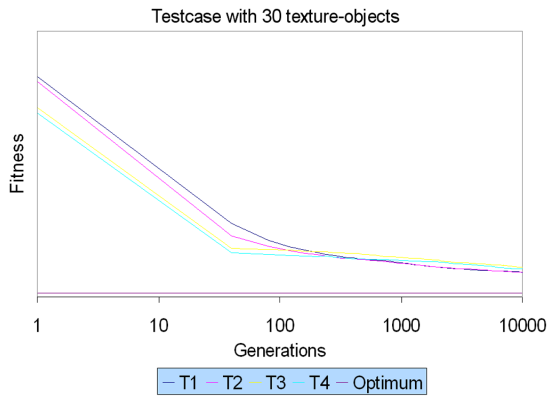

Figure 14: Results for 30 texture-objects without left-shifting. Curves from top to button T1: NC+NM, T2: GC+NM, T3: NC+GM, T4: GC+G

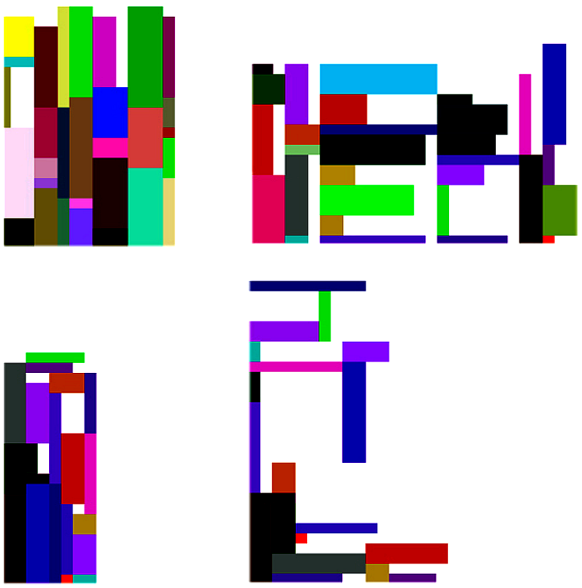

Figure 16: Results of the space optimization: lower pictures show the left-shifted approach. On the lower right, a bad solution for 20 textures can be seen. On the lower left, a good solution for 20 textures. On the upper right, a bad solution for 30 textures. On the upper left, a good solution for 30 textures. Good solutions are calculated after few generations.

4 Results

4.1 Quality of GA Solutions

Even in the case of vast search spaces the SO approach yields very good solutions in short time — either with or without left-shifting the textures.

Four different combinations were tested using the CPU approach: greedy (G) or normal (N) versions for crossover (C) and mutation (M). Greedy algorithms calculate possible solutions during the process of mutation or crossover by choosing the genes which actually lead to the best fitness. As Figure 14 exposes, the SO is capable of finding a good solution after less than 50 generations with an arbitrary configuration of 30 texture-objects (search space of 30 objects). The left-shifting approach generally achieved the same results.

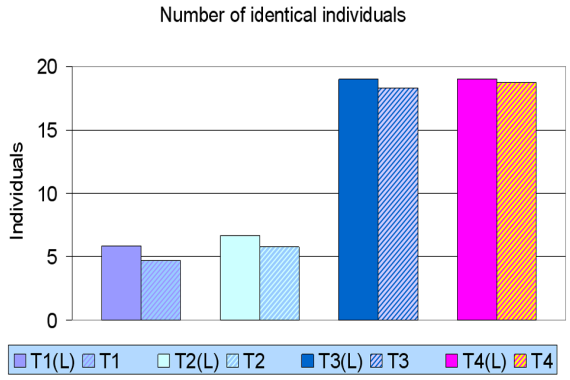

Figure 15: Number of identical individuals after 50 generations for test cases with and without left-shifting (L). Population size of 20. T1: NC+NM, T2: GC+NM, T3: NC+GM, T4: GC+GM

The greedy approaches, especially when left-shifting of textures is allowed, were not capable to escape local optima easily (see Figure 15). To address this problem, twins are deleted during the process and then replaced by new individuals.

Figure 16 exposes some typical texture packages generated by the presented approach.

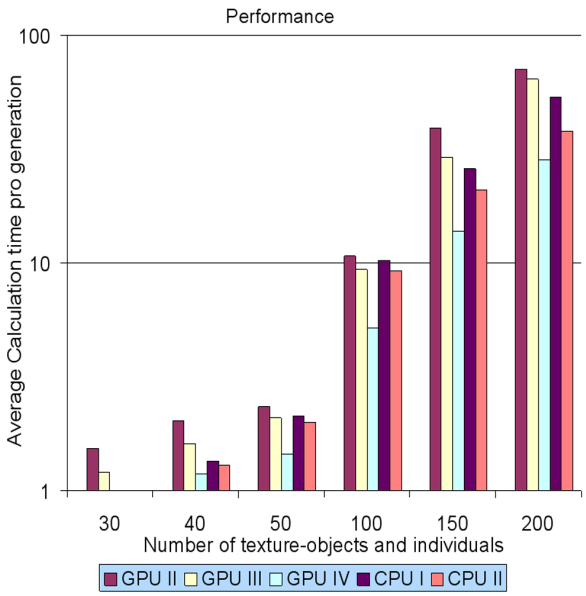

Figure 17: Performance comparison between GPU and CPU approaches. GPU II: Improved approach, can compute more than 200 texture-objects. GPU III: Improvement of GPU II approach. GPU IV: like GPU III but using optimal crossover approach. CPU I: normal CPU approach. CPU II: CPU approach using optimized crossover.

4.2 Performance on the GPU

Test hardware was a Pentium IV processor with 2GHz clock rate and 768 MB RAM under *Windows XP* using a *GeForce 6800 GT* graphics board with 128 MB RAM, 16 pixel pipelines at 400 MHz clock rate.

The GPU approach clearly outperforms the CPU implementation. Different SO approaches using the GPU are compared with two CPU versions in Figure 17. CPU I denotes the normal CPU approach, CPU II the CPU approach using optimal crossover. The GPU approaches differ in the way the shaders were programmed for mutation, crossover, and fitness calculation. Early approaches were only capable to compute 8 texture-objects (GPU I). GPU II and III approaches can compute 200 and more texture-objects by out-housing computation to other shaders. The GPU IV approach is capable to outperform the CPU approaches by using the optimal crossover attempt. It becomes faster than the CPU versions when more than 40 texture-objects are under consideration, although the difference is marginal (about 0.2 ms). When computing 100 and more texture-objects the GPU IV clearly leaves the CPU versions behind. Using 200 texture-objects, the GPU IV version is almost twice as fast as the CPU I approach and about three thirds faster than the CPU II approach.

To sum up, although GPU versions I-III are not capable of running faster than the CPU versions, GPU IV version outperforms both CPU versions in cases with more than 40 objects under consideration.

5 Summary

Our approach exposes the following: First, we showed that it is easily possible to build a tool capable of *automatically* and efficiently packing many granular textures into one large. It helps modelers in freely building virtual scenes without caring for efficient usage of texture memory. The results of the space optimization are like one knows from common CPU-based implementations. Also on the GPU, the algorithms run fast, robust, and yield very good local minima.

Second, we proved that general graphics processing units' inherent parallelism can be exploited to accelerate genetic algorithms. Only for cases where the number of textures is very small — the search space is very compact — the usual CPU implementation still remain faster than those on a GPU.

Like in almost all approaches which try to adapt general numerical problems to general graphics processing hardware, we separated the algorithm into several stand-alone parts (shader programs) executed independently on the GPU, whereas the transfer between them is done by the CPU.

References

[1] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1999.

[2] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. In *SIGGRAPH '02: Proc. 29. Conf. on Computer Graphics and Interactive Techniques*, pages 703–712. ACM, 2002.

[3] M. J. Harris, W. V. Baxter, T. Scheuermann, and A. Lastra. Simulation of Cloud Dynamics on Graphics Hardware. In *HWWS '03: Proc. ACM/EUROGRAPHICS Conf. on Graphics Hardware*, pages 92–101. Eurographics Association, 2003.

[4] J. Krüger and R. Westermann. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. *ACM Trans. Graph.*, 22(3):908–916, 2003.

[5] J. Lengyel, M. Reichert, B. R. Donald, and D. P. Greenberg. Real-Time Robot Motion Planning using Rasterizing Computer Graphics Hardware. In *SIGGRAPH '90: Proc. 17. Conf. on Computer Graphics and Interactive Techniques*, pages 327–335. ACM, 1990.

[6] J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, and L. Westover. Pixelflow: the realization. In *HWWS '97: Proc. ACM/EUROGRAPHICS Workshop on Graphics Hardware*, pages 57–68. ACM, 1997.

[7] C.-A. Bohn. Kohonen Feature Mapping through Graphics Hardware. In Paul P. Wang, editor, *Proc. JCIS'98*, volume II, pages 64–67. ACM, 1998.

[8] F. Haar and C.-A. Bohn. Compiling the Kohonen Feature Map into Computer Graphics Hardware. In *Proc. 8. Int. Conf. on Computer Graphics Applications and Artificial Intelligence*. ACM, 2005.