

# Optimized Pattern-Based Adaptive Mesh Refinement Using GPU

Ricardo Lenz, Joaquim Bento Cavalcante-Neto, Creto Augusto Vidal  
*Federal University of Ceará, Fortaleza, CE, Brazil*  
*riclc@hotmail.com, {joaquimb, cvidal}@lia.ufc.br*

## Abstract

*The high performance of GPUs and the increasing use of their programming mechanisms have fostered the development of graphics applications that better exploit the raw power of these devices to achieve higher levels of realism. Silhouette refinement, as one of the techniques that help to improve realism, has profited from GPUs' advances in recent years. In this paper, we present a method for triangular mesh refinement which alleviates the problem of rugged silhouettes. We demonstrate that, through a clever indexing scheme, our method is able to use adaptive patterns in an optimized way, taking full advantage of the GPU's parallelism. The ideas we used were adapted from distinct previous works, but our method presents astounding performance gains. Also, our method works very well with existing meshes, therefore, it can improve the visual appearance of existing models without mesh redesign.*

## 1. Introduction

One of the traditional ways to achieve higher levels of realism in real time graphics applications has been the use of textures, which are mapped onto the surface of three-dimensional objects, together with light effects calculated for each pixel [8, 10]. However, applying texture does not modify the object's mesh, and, consequently, the silhouette exposes the flaws of the underlying model, partially destroying the realism one was trying to achieve with textures and per-pixel effects. In order to alleviate the problem of rugged silhouette and maintain good level of realism, a complementary set of techniques have been proposed in the literature [1, 5, 6, 8, 16, 19]. They exploit the object's geometry for generating a new silhouette that helps to enhance the object's expressiveness. Those techniques perform a geometric synthesis over an underlying existing model, i.e., a mesh is passed as input parameter to a function that drives the mesh refinement in order to generate an output mesh. For example, the mesh refinement may be an interpolation of a cubic Bézier surface based on data from the original mesh. In this case, it is commonly used the so called Curved PN Triangle [19], which produces

a mesh with softer silhouette curves than the original version.

The purpose of this paper is to present a method for refinement of triangular meshes that can take advantage of the high computational power of a GPU, following the recent trends of high performance computer graphics [15]. Some of the main features of the latest generation of GPUs are used, as reported in [2, 7, 11]. We demonstrate that, through a clever indexing scheme, our method is able to use patterns in an optimized way, taking full advantage of the GPU's parallelism. The ideas we used were adapted from distinct previous works, but our method presents astounding performance gains. Also, our method works very well with existing meshes, therefore, it can improve the visual appearance of existing models without mesh redesign.

The remainder of this paper is organized as follows: Section 2 deals with related work and makes a comparative discussion between them; Section 3 explains some of the key concepts involved in the method; Section 4 addresses the details of the proposed technique, and Section 5 shows some examples, making a comparison and discussing the overall performance of the method; finally, Section 6 presents the conclusions and some recommendations for future work.

## 2. Related Work

Some ideas on how to work with mesh refinement on GPU have emerged from the theoretical framework proposed by Shiue and others [18]. The initial works, such as the one by Bolz and Schröder [3], used GPGPU (General-Purpose computing on the GPU) [15] techniques, i.e., they exploited the GPU's computational power by using more general techniques simulated in this device, without, however, making use of the specific graphical and geometric features of the GPU. In these GPGPU algorithms, sometimes a mesh of vertices is represented in an adapted form as a table of pixels, for example. The work of Boubekeur and Schlick [5] contains a list of these initial historical works.

However, the task of making a mesh refinement, using the specialized geometric features of GPUs, would be easier to setup if the type of refinement to be applied were a simple one. For example, it would be easy to

make a simple mesh smoothing through interpolation of a local surface (Figure 1), if the only information required were its vertices and normals, which are elements present already in the graphical architecture of the GPU platform.

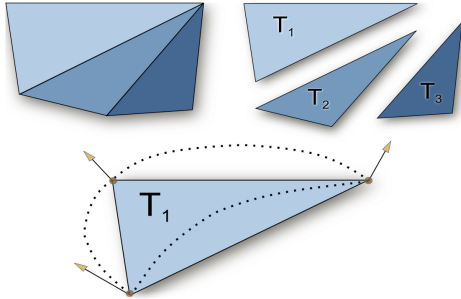


Figure 1: Local refinement method.

The cubic Bézier triangular surface of the Curved PN Triangle, presented in [19], fulfilled those conditions. Basically, the Curved PN Triangle acts as a specific heuristic for determining a particular triangular surface based on the more general triangular cubic Bézier surface. This heuristic uses only the original data of the coarse mesh, in a scenario where there are only scarce geometric descriptions of the objects being represented. Its ease of implementation in hardware was soon evident, when it was implemented on some GPUs [1]. Subsequently, Boubekeur and others [4] presented a modified version of the Curved PN Triangle, where they altered some of its coefficients to allow further manipulations of the final surface being generated. This was done by introducing three new scalar parameters for each vertex, the scalar tags: sharpness ( $\sigma$ ), tension ( $\theta$ ) and bias ( $\beta$ ). Because of this, Boubekeur's model was called ST-Mesh.

The pioneering work of Boubekeur and Schlick [5] on local refinement of triangular meshes taking advantage of the GPU's graphical and geometrical features was made using the smoothing technique of Curved PN Triangle with Scalar Tags (ST-Mesh). Their method used topological patterns that were uniform (a brief explanation of those concepts is given in Section 3). Later, the same authors presented an enhanced version that worked with adaptive patterns [6]. They called the the new method Adaptive Refinement Kernel (ARK).

Recently, Lorenz and Döllner [12] introduced another method for triangular mesh refinement using GPU, the so-called Dynamic Mesh Refinement (DMR). That method also uses adaptive topological patterns, and complements the ARK method, in the sense that it outperforms ARK only in the specific scenarios where ARK is incapable of having a good performance. However, as the authors point out, DMR was conceived specifically to complement ARK, not to replace it, so

DMR's performances in certain situations are worse than those of ARK. A fundamental difference of those two methods is that ARK performs a long list of calls from the CPU to the GPU, while DMR does not. In ARK, each call processes one element of the input mesh at a time, by using the Vertex Shader and uniform variables (for further information on basic concepts of GPU programming, see [17]). As there is some cost involved within each call to process data on the GPU, it is recommended to minimize the number of tasks passed to the GPU and to maximize the size of each task data, so that the processing done surpasses this associated cost. The ARK method is affected by this in a negative way, because it delivers a great deal of small tasks to the GPU. The DMR method, on the other hand, adopts an opposite approach, where the number of GPU calls is minimized, and the GPU must perform almost everything on its own. To accomplish this, DMR uses the Geometry Shader [7] in a multi-pass algorithm. The problem with this strategy is that Geometry Shader is still very limited to do geometrical amplification, that is, it is very limited on the number of new vertices it can emit at a time, and so, several passes must be done, which greatly slows down the algorithm execution for higher refinement levels. In synthesis, DMR is better for lower refinement levels and ARK is better for higher refinement levels.

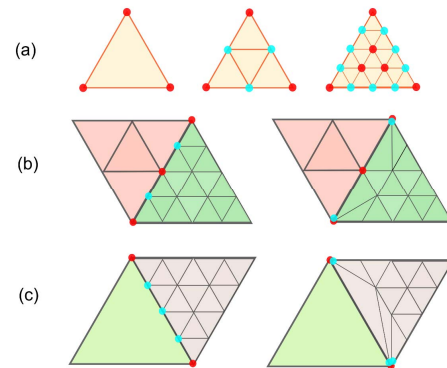


Figure 2: SUAPT method uses uniform patterns (a) and adapt them at the boundary (b, c). Blue points are moveable, red ones are not.

Another triangular mesh refinement method using the GPU, Semi-Uniform Adaptive Patch Tessellation, to be further referenced here as SUAPT, was very recently proposed by Dyken and others in [9]. This method basically uses uniform topological patterns, as the primitive ARK did, but exploiting a modern GPU feature called Instancing in order to maximize the parallelism. The SUAPT method, however, does not use the uniform patterns in the same old way: it "merges" one patch with another by a simple algorithm, making the final mesh crack-free (conforming mesh). But the "merging" process is very limited and simple: it often

generates very abrupt regions [14] on the final mesh (Figure 2), which is a problem inherent to the nature of this method. This SUAPT problem occurs because the region that is adapted is a very small region (in fact, just the patch boundary). In contrast, by using real adaptive topological patterns, all the red points in Figure 2 could also be moved, which makes adaptive patterns more powerful.

### 3. Basic Concepts

All the triangular mesh refinement methods with GPU that were mentioned in Section 2 deal with topological patterns (Figure 3a), and are, therefore, called pattern-refinement methods. These patterns are just subdivided triangles according to some topology, with all their vertices' positions described in barycentric coordinates  $(u, v, w)$ . The triangle subdivision complies with the discretization level given in each edge. For example, an edge discretization level  $d = 3$  indicates that it should be recursively divided in half three times. The notation  $[d_1, d_2, d_3]$  is then used to refer to the pattern being described by the edge discretization levels of  $d_1, d_2$  and  $d_3$ . For example, the second pattern in Figure 3a would be  $[0, 1, 0]$ . A pattern is said to be uniform when the restriction  $d_1 = d_2 = d_3$  is satisfied. When there is no such a restriction, the pattern is said to be adaptive.

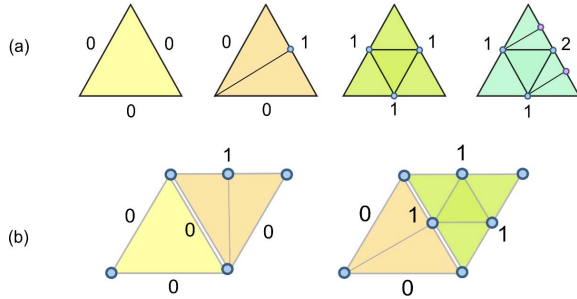


Figure 3: Some topological patterns (a) and their careful use on patches (b).

Initially, for each triangle of the original mesh, the refinement method must choose a pattern that fits in the topological discretization desired for that particular triangle. This process is called pattern selection (Figure 4). In the adaptive selection of patterns, the resulting mesh is kept crack-free because the choice of patterns occurs using edge information. For example, a triangle that uses pattern  $[0, 1, 0]$  perfectly matches with a neighbor that uses pattern  $[1, 1, 1]$  on the shared edge, because the same level of discretization  $d = 1$  occurs on both boundaries (Figure 5b). This means that each edge of the mesh will be assigned a  $d$  value which dictates the desired discretization there, and it will be widely used in the pattern selection process.

To comply with current graphics systems, it is easier to specify an attribute for each vertex of the mesh rather than for each edge. This means that it is easier to have a tuple  $(d_{v_1}, d_{v_2}, d_{v_3})$  defining the level of discretization in each of the vertices  $v_1, v_2$  and  $v_3$  of a triangle, and then use this information to find out which pattern  $[d_1, d_2, d_3]$  to define, as if  $d_1, d_2$  and  $d_3$  were directly specified on the triangle's edges. This can be accomplished by computing the mean of the two vertices of an edge. For example,  $d_1$  could be computed as the mean of  $d_{v_1}$  and  $d_{v_2}$ . The choice of which level of discretization  $d_v$  to set for each vertex can be made according to different criteria: distance to camera, curvature operator such as a simple discrete Gaussian curvature [13] or even the method that Dyken and others [8] proposed for the visual silhouette refinement.

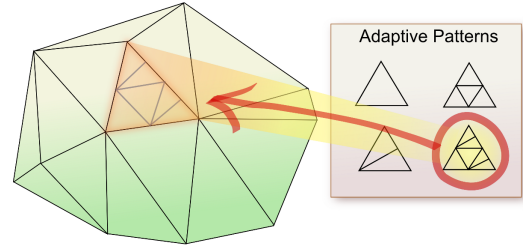


Figure 4: Pattern selection.

After the selection of patterns, the refinement method can finally generate the output mesh by mapping all the chosen pattern barycentric coordinates to the desired coordinates. This can be done in a straightforward way, and several applications arise in here. For example, in order to generate just a simple discretization of each triangle of the original mesh, the following equation can be used, where  $(u, v, w)$  are the barycentric coordinates and  $(P_1, P_2, P_3)$  are the triangle coordinates:

$$V_{final} = P_1 u + P_2 v + P_3 w \quad (1)$$

### 4. Adaptive Mesh Refinement Method with Fewer Patterns and Instancing

The method proposed in this paper is based on the ARK method, but modified for better parallelism in a performance-and-memory optimization done by using a new adaptive pattern indexing scheme. Because fewer adaptive patterns are needed and the GPU Instancing feature is being used, the proposed method is named ARKFPI (ARK with Fewer Patterns and Instancing).

The method follows the basic outline provided in Section 3. The general algorithm can be seen as follows: first, the method uploads all the stored adaptive patterns (which are generated off-line by an external piece of software) to the GPU memory. This is done once and for all. Second, the algorithm does the pattern selection process, assigning to each mesh element some  $[i, j, k]$

pattern, using the new indexing scheme that is proposed in this paper. Then, all elements which use the same base pattern are arranged in one group, this is done for every base pattern being used. Finally, all those groups are processed using the GPU Instancing feature, which will render each element's pattern accordingly. Notice that, since the rendering process is run on the GPU, using barycentric coordinates, after rendering, the results must be mapped to the desired coordinates with the help of the input mesh's data and of a custom parametric function. On ARKFPI, an additional step is required during the rendering process inside the GPU. This is necessary due to the additional level of indirection used by ARKFPI, which stores only a set of base patterns that span all the patterns used in the mesh refinement.

#### 4.1. New Pattern Indexing Scheme

A certain  $[i, j, k]$  pattern can easily be used as if it were the  $[j, i, k]$  pattern, for example. This occurs because equation (1) of Section 3 can be easily modified to work with the barycentric coordinate values  $(u, v, w)$  in a different order. Instead of associating  $u$  with  $P_1$ ,  $v$  with  $P_2$  and  $w$  with  $P_3$ , one can associate  $w$  with  $P_1$ ,  $v$  with  $P_2$  and  $u$  with  $P_3$ . The effect of this is as if the  $[j, i, k]$  pattern were used instead of the original  $[i, j, k]$  pattern. Therefore, instead of having all the six different permutations of  $[i, j, k]$  stored as distinct adaptive patterns in memory (Figure 5a), one needs to store only one such pattern (Figure 5b) and recover the other five through permutation of the  $(u, v, w)$  variables.

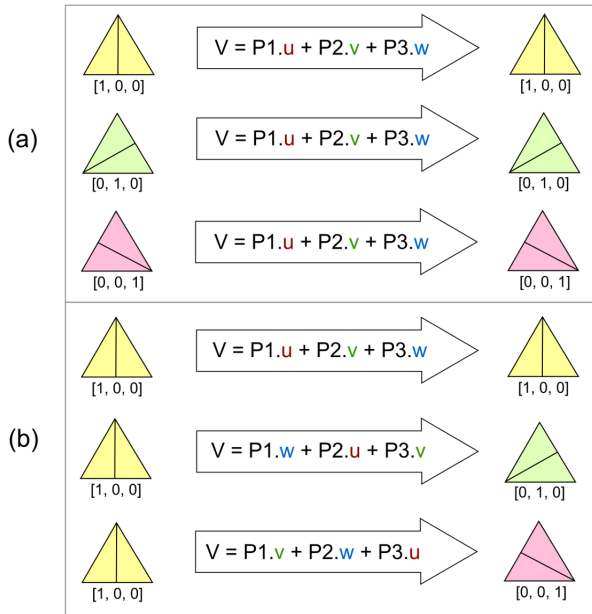


Figure 5: Permutation of the  $(u, v, w)$  barycentric coordinate values to obtain new patterns from existing adaptive patterns.

The question of what pattern should be regarded as the base pattern is a matter of choice and any criterion can be established. In this paper, the following criterion was adopted: the base pattern chosen is the one whose  $i, j, k$  stored values satisfy the restriction  $i \geq j \geq k$ . Thus, in the example shown in Table 1 only the  $[2, 1, 0]$  pattern is stored.

Table 1: A sample set of adaptive patterns in the new indexing scheme.

$[2, 1, 0]$	Derived from $[2, 1, 0]$ as $(u, v, w)$
$[2, 0, 1]$	Derived from $[2, 1, 0]$ as $(v, u, w)$
$[1, 2, 0]$	Derived from $[2, 1, 0]$ as $(w, v, u)$
$[1, 0, 2]$	Derived from $[2, 1, 0]$ as $(v, w, u)$
$[0, 2, 1]$	Derived from $[2, 1, 0]$ as $(w, u, v)$
$[0, 1, 2]$	Derived from $[2, 1, 0]$ as $(u, w, v)$

There are cases, however, where the six permutations of  $(u, v, w)$  do not correspond to six different adaptive patterns. This are the cases of base patterns  $[1, 1, 1]$  and  $[1, 2, 1]$ , for example. Considering this, the following equation computes the total number of  $[i, j, k]$  adaptive patterns, with  $i, j, k \leq N$ , that actually need to be stored (i.e., the number of base patterns):

$$\frac{N^3 + 3N^2 + 2N}{6} \quad (2)$$

As can be seen in Figure 6, the reduced number of stored patterns on ARKFPI becomes significant for a higher value  $N$  of maximum discretization. For example, for  $N = 10$ , 1000 adaptive patterns must be stored when standard adaptive pattern-based methods like ARK or DMR are used; however, only 220 adaptive patterns are stored when ARKFPI is used, a reduction of 78%.

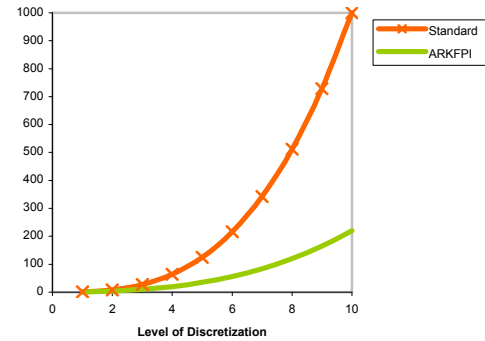


Figure 6: Number of stored adaptive patterns: ARKFPI vs. Standard adaptive methods.

#### 4.2. Which Permutation to Choose

During the pattern selection process, each element of the original mesh is assigned a specific  $[i, j, k]$  pattern, but, now, only the patterns that meet the criterion  $i \geq j \geq k$  are stored. Thus, the  $(u, v, w)$  barycentric coordinate values pass through a permutation to obtain the  $[i, j, k]$

desired pattern from a base pattern that is stored. Yet, the following question arises: What permutation of  $(u, v, w)$  must be chosen in order to achieve the desired pattern from the base patterns available?

This question can be easily solved by introducing a new  $\lambda$  variable for each element of the mesh (Figure 7). This simple variable defines what permutation must be done to obtain that element's desired pattern from the base patterns stored. The question to be addressed now is how to compute and use  $\lambda$  itself.

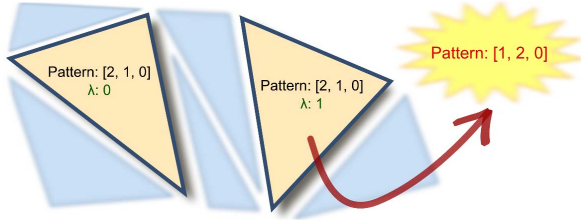


Figure 7: Storage of  $\lambda$  for retrieval of desired pattern.

In order to compute  $\lambda$ , recall that the base pattern has always  $i \geq j \geq k$ , so that a simple sorting of any  $[i, j, k]$  pattern can lead to its associated base pattern. Because a sorting algorithm can be thought of as a sequence of swap operations, if the sorting algorithm keeps track of all swap operations it does, it is possible to know precisely which permutation of its numbers lead to the sorted sequence. Note that only three swap operations may be done when sorting a sequence with three numbers, so it is possible to design a very simple algorithm to compute  $\lambda$  (Figure 8).

```

1.  $\lambda \leftarrow 0$ 
2. if  $i < j$ , swap  $i \leftrightarrow j$ ,  $\lambda \leftarrow \lambda + 1$ 
3. if  $i < k$ , swap  $i \leftrightarrow k$ ,  $\lambda \leftarrow \lambda + 2$ 
4. if  $j < k$ , swap  $j \leftrightarrow k$ ,  $\lambda \leftarrow \lambda + 4$ 

```

Figure 8: Sorting algorithm that keeps track of the swap operations on  $[i, j, k]$ .

By using the  $[i, j, k]$  sample pattern values from Table 1 as the input to the algorithm of Figure 8, it is possible to compute  $\lambda$  for each of the sample patterns, and by looking at the associated  $(u, v, w)$  permutations specified on Table 1, one can finally know which  $\lambda$  values lead to which permutations (Table 2).

Table 2: Association of each possible  $\lambda$  value with a permutation of  $(u, v, w)$ .

$\lambda$	Permutation
0	$(u, v, w)$
1	$(w, v, u)$
2	$(u, w, v)$
4	$(v, u, w)$
5	$(w, u, v)$
6	$(v, w, u)$
7	$(u, w, v)$

As a matter of fact, the case  $\lambda = 3$  will never happen, because if the conditions of steps 2 and 3 of the algorithm in Figure 8 are true, then the condition in step 4 will also be necessarily true (that is, if  $a < b$  and  $b < c$ , then  $a < c$ ).

### 4.3. Group Arrangement and Dispatching

Thanks to the new pattern indexing scheme proposed, the list of patterns is smaller, and a larger number of triangles will use the same base patterns, although with different permutations. Thus, the GPU Instancing feature can be used effectively, since there is data to be used many times. Instancing allows all parts of the mesh associated with the same base pattern to be processed in a single call (Figure 9). The only thing required is to arrange all mesh elements that use the same base pattern into one large group, which will be processed with Instancing. That must be done for each base pattern being used.

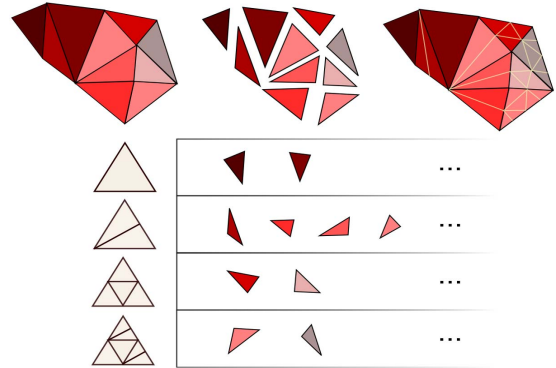


Figure 9: All mesh elements that use the same base patterns are processed in groups.

Although one could suppose that there would be many groups of patterns, such as the groups  $[2, 1, 0]$  and  $[1, 2, 0]$ , recall that some of those patterns are associated with the same base pattern, for example, patterns  $[2, 1, 0]$  and  $[1, 2, 0]$  are associated with base pattern  $[2, 1, 0]$ . Furthermore, each mesh element already has its own  $\lambda$  (Figure 7). Thus, as Figure 10 shows, it is straightforward to form groups of patterns, which have a common associated base pattern. Also, because the listing of groups is compacted by using base patterns instead of patterns, the parallelism is increased.

In the final step, all those groups are dispatched for rendering and the coordinate mapping is done, as explained in Section 3. Many different effects can be accomplished here. It may be used per-vertex displacement mapping to increase the realism of the surface with a texture height map [16], for example, or to perform the fitting of a smooth parametric surface on the mesh data, similar to what is done with the Curved PN Triangle; even procedural methods can be applied.

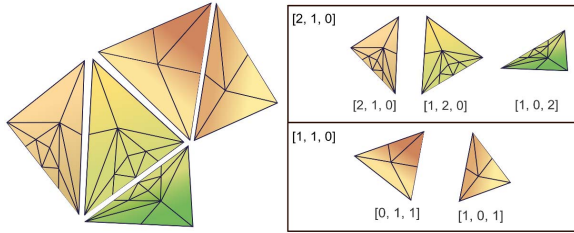


Figure 10: Groups are assembled by the same base pattern, not just the same pattern.

## 5. Examples and Results

This section provides some examples for the ARKFPI method, showing the results obtained and also a discussion that examines the overall performance of the method.

### 5.1. Example: Rugged vs. Smooth Silhouette

In Figure 11, we used ARKFPI in order to apply Curved PN Triangles to the Stanford bunny's mesh for smoothing the rugged silhouette.

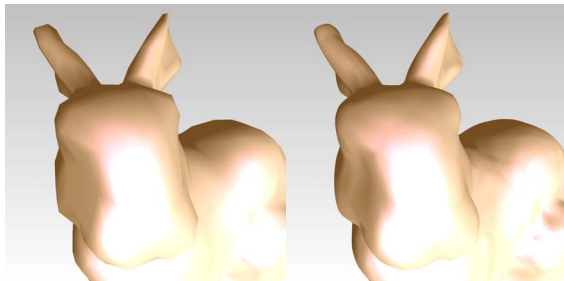


Figure 11: The Stanford bunny (rugged vs. smooth silhouette).

### 5.2. Example: Customized Smooth Silhouette

In Figure 12, we applied the ST-Mesh technique to the clown model. Some of the mesh's vertices have customized scalar tags. This example clearly shows that the original mesh and original appearance of the model can be significantly improved.

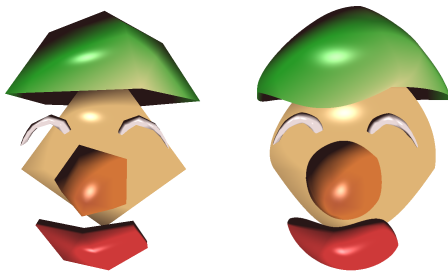


Figure 12: Clown (rugged vs. customized smooth silhouette).

### 5.3. Example: Procedural Silhouette - Fur

The refinement criteria is sufficiently flexible for many applications. With a single mechanism, it is possible to accomplish many different effects. For example, in Figure 13, the Stanford bunny is shown with procedural fur, which is achieved with ARKFPI by using a custom function. Parameters like fur size and frequency can be set easily.

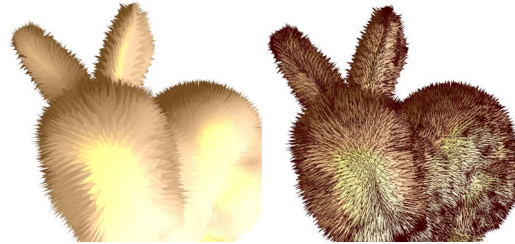


Figure 13: Stanford bunny with procedural fur.

### 5.4. Example: Procedural Silhouette – Terrain

In Figure 14, we applied Per-Vertex Displacement Mapping [16] to the coarse input mesh, thus adding relief information to the surface. The basic mesh was adaptively augmented by a procedural method that generated the surface's own texture and also its own height map, which was used in order to create the relief. Parameters such as the height of the terrain can be adjusted easily.

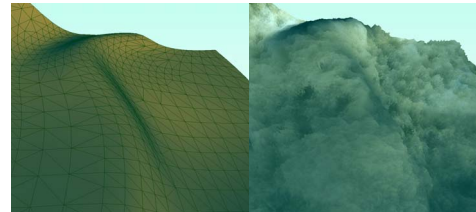


Figure 14: Procedural Terrain.

### 5.5. Example: Displacement Mapping with Smoothing Silhouette

The example in Figure 15 presents a Curved PN Triangle technique combined with Displacement Mapping. A simple triangle has its normals manipulated by the user and the generated mesh moves along its directions, with all its relief information together. It is a very interactive technique.

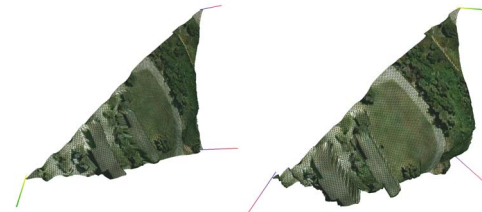


Figure 15: Curved triangular piece of terrain with displacement mapping applied.






## 5.6. Performance

We present here a set of examples to clearly expose a performance comparison of ARK with ARKFPI. The examples shown in Table 3 are, respectively: Sphere 1, Sphere 2, Pacman 1, Jet and Pacman 2.

Note that the time spent processing the Curved PN Triangle interpolation or the Displacement Mapping technique is the same for both methods. This is important for comparison purposes and reveals the care we took for doing the same implementation of those techniques in ARK and ARKFPI. Thus, the difference on the final performance lies not on the technique being applied, but on the method's own nature. We also were careful to apply the same discretization to the same mesh using both methods in order to have a fair comparison.

In every test, we indicate the number of elements of the input mesh as well as the number of elements of the output mesh. The I/O factor is the number of output elements divided by the number of input elements. The rendering performance is measured in frames per second (FPS). The various discretization levels being used are chosen at random, and are the same for both methods. All tests were performed with a GeForce 9600 GT, using OpenGL.

**Table 3: Comparison of ARK with ARKFPI.**

Example	Elements		I/O factor	FPS		Increasing Performance
	In	Out		ARK	ARKFPI	
	16	25312	1582	159	285	79.2%
	16	11006	688	168	404	140.5%
	16	8848	553	219	639	191.8%
	48	4758	99	130	499	283.8%
	48	8856	185	123	396	222.0%
	48	16476	343	69	269	289.9%
	110	73094	664	46	118	156.5%
	110	25154	229	52	188	261.5%
	110	14154	129	59	452	666.1%
	303	1174	4	23	728	3065.2%
	303	4363	14	23	498	2065.2%
	303	10157	34	23	338	1369.6%
	303	25226	83	23	199	765.2%
	490	1974	4	14	501	3478.6%
	490	3846	8	14	437	3021.4%
	490	8084	16	14	365	2507.1%
	490	15170	31	14	239	1607.1%
	490	98146	200	14	80	471.4%
	490	521536	1064	12	20	66.7%

## 5.7. Discussion

In ARKFPI, the entire input mesh is stored on the GPU as a buffer, and so each time a big Instancing-group of patterns is processed inside the GPU, every needed vertex of the input mesh, which is already there, is fetched and processed. Thus, no CPU-GPU extra communication is needed. On the other hand, in ARK the entire input mesh is in the CPU memory, and, so, the mesh vertices must be sent to the GPU on every rendering.

Furthermore, ARK does have a bottleneck: because it does a large number of calls from the CPU to the GPU, proportional to the number of elements in the input mesh, it can be very slow for a large input mesh, even if the I/O factor is low. Therefore, ARK is very sensitive to the mesh input size. On the other hand, ARKFPI better saturates the GPU, and so it is, naturally, more sensitive to the number of elements in the output mesh than ARK. Thus, ARK's main bottleneck does not exist in ARKFPI. Instead, when the input mesh's size is very high but the I/O factor is low, that is precisely the best-case scenario where ARKFPI clearly wins by far (on the tests made, up to 3479% increase of performance was achieved, as can be seen in Table 3). In the worst-case scenario, however, ARKFPI still outperformed ARK by 66.7%.

## 6. Conclusions and Future Work

This paper has proposed the ARKFPI method, which is based on the original ARK method. Our method exploits very well the parallelism by using GPU Instancing and a reduced adaptive pattern set with a compact listing of groups. This, together with the fact that there is no "per face sending of vertex data" in the rendering process, clearly makes our method much more GPU-intensive than ARK, both on processing and storing data.

In ARKFPI, the number of adaptive patterns that have to be stored is much lower than in ARK because of the new pattern indexing scheme that we proposed. This reduction on the number of stored adaptive patterns not only improves the memory footprint, but also allows for better use of the Instancing feature of modern GPUs, which will greatly increase the overall performance. The method works basically on existing triangular meshes, and outputs a new generated mesh based on the coarse input mesh guided by some user function that manipulates geometric data in a customized way. The method works well enough for the widely available coarse triangular meshes that exist today in various graphics applications and games, and can be used for many different effects.

For future work, one can investigate other ways to deal with the topological patterns in order to reduce their numbers even more. A possible incorporation of the PNG1 surface [10] is also being considered. It is interesting to point out that new advances in GPU devices are being made which will appear very sound to mesh refinement methods, like the *Hull Shader* and *Domain Shader*. Another future work may be a comparison of ARKFPI with DMR.

## 7. References

- [1] ATI Inc. TRUFORM white paper, 2001.
- [2] David Blythe. The Direct3D 10 system. Proceedings of ACM SIGGRAPH, p. 724-734, 2006.
- [3] Bolz J. and Schroder P. Evaluation of subdivision surfaces on programmable graphics hardware, 2003.  
<http://www.multires.caltech.edu/pubs/GPUSubD.pdf>.
- [4] Tamy Boubekeur, Patrick Reuter, Christophe Schlick. Scalar Tagged PN Triangles. Eurographics 2005 Short Presentations, p. 17-20, 2005.
- [5] Tamy Boubekeur, Christophe Schlick. Generic Mesh Refinement on GPU. Proceedings of the ACM Siggraph/Eurographics Conference on Graphics Hardware, p. 99-104, 2005.
- [6] Tamy Boubekeur, Christophe Schlick. A Flexible Kernel for Adaptive Mesh Refinement on GPU. Computer Graphics Forum, Volume 27, Issue 1, p. 102-113, 2007.
- [7] Pat Brown and Barthold Lichtenbelt. EXT\_geometry\_shader4, 2006.  
[http://www.opengl.org/registry/specs/EXT/geometry\\_shader4.txt](http://www.opengl.org/registry/specs/EXT/geometry_shader4.txt).
- [8] C. Dyken, M. Reimers, J. Seland. Real-Time GPU Silhouette Refinement using Adaptively Blended Bézier Patches. Computer Graphics Forum, Volume 27, Issue 1, p. 1-12, 2007.
- [9] C. Dyken, M. Reimers, J. Seland. Semi-uniform Adaptive Patch Tessellation. Computer Graphics Forum, 2009 (to be published).
- [10] Christoph Fünzig, Kerstin Müller, Dianne Hansford, Gerald Farin. PNG1 Triangles for Tangent Plane Continuous Surfaces on the GPU. ACM International Conference Proceeding Series, Volume 322, Proceedings of graphics interface, pages 219-226, 2008.
- [11] Barthold Lichtenbelt and Pat Brown. EXT\_gpu\_shader4, 2006.  
[http://www.opengl.org/registry/specs/EXT/gpu\\_shader4.txt](http://www.opengl.org/registry/specs/EXT/gpu_shader4.txt).
- [12] Haik Lorenz, Jürgen Döllner. Dynamic Mesh Refinement on GPU using Geometry Shaders. Proceedings of the 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, p. 97-104, 2008.
- [13] Mark Meyer, Mathieu Desbrun, Peter Schroeder and Al Barr. Discrete Differential Geometry Operators for Triangulated 2-Manifolds. International Workshop on Visualization and Mathematics, 2002.  
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.3427>
- [14] Henry Moreton. Watertight tessellation using forward differencing. Graphics Hardware, p. 25-32, 2001.
- [15] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohn and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. Computer Graphics Forum, Volume 26, Issue 1, p. 80-113, 2007.
- [16] László Szirmay-Kalos, Tamás Umenhoffer. Displacement Mapping on the GPU - State of the Art. Computer Graphics Forum, Volume 27, Issue 6, p. 1567-1592, 2008.
- [17] Randi J. Rost. OpenGL Shading Language, Second Edition. Addison Wesley Professional, 2006.
- [18] Shiue, L.-J., Goel, V., and Peters, J. Mesh mutation in programmable graphics hardware. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, p. 15-24, 2003.
- [19] Alex Vlachos, Jörg Peters, Chas Boyd, Jason Mitchell. Curved PN Triangles. ACM Symposium on Interactive 3D Graphics, p. 159-166, 2001.