A Flexible Kernel for Adaptive Mesh Refinement on GPU
Tamy Boubekeur, Christophe Schlick

To cite this version:

HAL Id: inria-00260825
https://hal.inria.fr/inria-00260825
Submitted on 5 Mar 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
A Flexible Kernel for Adaptive Mesh Refinement on GPU
Tamy Boubekeur  Christophe Schlick
LaBRI - INRIA - University of Bordeaux

Abstract
We present a flexible GPU kernel for adaptive on-the-fly refinement of meshes with arbitrary topology. By simply reserving a small amount of GPU memory to store a set of adaptive refinement patterns, on-the-fly refinement is performed by the GPU, without any preprocessing nor additional topology data structure. The level of adaptive refinement can be controlled by specifying a per-vertex depth-tag, in addition to usual position, normal, color and texture coordinates. This depth-tag is used by the kernel to instantiate the correct refinement pattern, which will map a refined connectivity on the input coarse polygon. Finally, the refined patch produced for each triangle can be displaced by the vertex shader, using any kind of geometric refinement, such as Bezier patch smoothing, scalar valued displacement, procedural geometry synthesis or subdivision surfaces. This refinement engine does neither require multi-pass rendering nor any use of fragment processing nor special preprocess of the input mesh structure. It can be implemented on any GPU with vertex shading capabilities.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Picture/Image Generation]: Display Algorithms - I.3.5 [Computational Geometry and Object Modeling]: Object Hierarchies - I.3.1 [Hardware Architecture]: Graphics Processors

1. Introduction
For a wide range of applications, image synthesis techniques leverage the amount of information required for creating realistic animated pictures. In particular, for real-time rendering, the application has just to provide a set of polygons describing the geometry of a scene, and the graphics hardware will automatically produce a coherent grid of pixels through the usual rasterization pipeline. However, the bandwidth bottleneck between the application and the graphics hardware limits the size of geometric description that can be transmitted for real-time rendering, and thus also limits the realism of the rendered pictures.

On-the-fly geometry synthesis addresses this issue by allowing an additional level of abstraction in the graphics pipeline. For interactive applications, geometry synthesis is usually cast as a mesh refinement process. Rather than enumerating the huge number of polygons that would be required to get an accurate discrete approximation of a complex shape, mesh refinement techniques split the surface representation into a coarse polygonal mesh combined with a continuous displacement function. Then, at rendering time, mesh refinement basically performs two successive operations on the coarse mesh: a tessellation step followed by displacement one.

During the first step, a refined mesh topology is generated at a target level-of-detail, simply by splitting each coarse polygon into a set of finer ones, without any actual geometric modification. Then, during the second step, each newly inserted vertex is translated to its final position, obtained by sampling the continuous displacement function. Many existing computer graphics techniques can be expressed under this paradigm, such as spline-based or wavelet-based surface representation, subdivision surfaces, hierarchical height fields, etc. The key feature that makes this process work well, is that the continuous displacement function can usually be defined by providing a very small amount of data compared to the size of the huge refined mesh. Examples of such additional data include subdivision masks for smooth surface generation, bitmap textures for displaced meshes, or a bunch of numerical data for procedural geometry synthesis.

However, performing a full GPU implementation of this two-step process remains a problem with current devices.
While graphics hardware offers a flexible vertex shader stage that allows an efficient implementation of the displacement step, the lack of geometry creation on GPU makes the implementation of the tessellation step really tricky. Last generation devices, launched at the end of 2006, embed a geometry shader stage [Bly06] which has been specifically designed for geometry upscale. Unfortunately, even if the geometry shader clearly represents a step in the right direction, it does not provide the ultimate high-level flexible solution demanded by many applications. One of its main limitations is that the geometry shader cannot output (i.e. generate) more than a fixed amount of floating point numbers (1024 in the original specification), which means that only about 2 or 3 levels of refinement can be applied on each coarse triangle. If deeper refinement is required, multi-pass geometry shading has to be employed, which obviously reduces overall performances.

The lack of flexible geometry synthesis on GPU, has led some researchers to cast the mesh refinement problem as a general purpose computation problem, using a GP-GPU approach [GPGo06]: by converting the coarse mesh as a standard rectangular image, the tessellation step becomes a simple image upscaling operator, and the displacement step can be implemented in the fragment shader stage. However, such an approach induces several strong restrictions. First, it requires an additional preprocessing step to convert the mesh into an adapted image format. Second, it involves intensive use of multi-pass rendering and fragment shading, while the vertex shading stage is greatly under-exploited, as it only has to process a few full-screen quads. Third, the whole process has potentially to be restarted for each frame in the case of dynamic meshes. Fourth, additional hardware pipelines (e.g. physics simulation hardware) are not directly compatible with such an approach, since no object space geometry is really produced. And last, multiresolution and adaptivity cannot be easily handled by such a process.

In this paper, we propose an alternative approach, that we call adaptive refinement kernel (ARK), based on three key features. First, a flexible control of the adaptive level-of-detail is obtained by a simple and generic depth-tagging process. Second, a set of adaptive refinement patterns (ARP) is employed to allow crack-free adaptive multiresolution refinement. And third, a specific single-pass vertex program, called adaptive refinement shader (ARS), performs both tessellation and displacement steps involved in mesh refinement. By combining all three ingredients, we obtain a flexible kernel for adaptive on-the-fly mesh refinement on GPU.

This kernel does not involve any preprocessing of input coarse meshes, as it directly processes the basic mesh representation used in low-level APIs, such as polygon soups or indexed triangle sets, without requiring additional high-level data structures (e.g. half-edge representation). With our kernel, the final mesh is never generated on the CPU, never transmitted on the graphics bus, and even never explicitly stored on the GPU. All the refinement is performed by our single-pass generic vertex program, which totally frees the fragment shaders for including additional visual enrichments. This kernel is particularly well-suited for dynamic meshes which are deformed on a frame-to-frame basis (animation of characters, physics simulation, etc.) and for procedural shapes that usually include high frequency features and require fine tessellation at rendering time.

The remainder of this paper is organized as follows: Section 2 reviews related work, Section 3 describes our new adaptive refinement kernel, Section 4 shows some applications of the ARK, Section 5 gives results and performances of our system and Section 6 concludes and proposes some directions for future work.

2. Previous Work

Existing mesh refinement methods can basically be divided in two main categories: either direct or indirect refinement.

Direct Refinement: This first category includes pure geometry synthesis approaches, where the input coarse mesh is directly refined in object-space, without additional conversion steps. Multi-scale rendering of numerical models of terrains are maybe the most classical examples of on-the-fly direct refinement [AH05], but the involved algorithms are usually limited to height-field configurations. Another well-studied topic includes all the techniques that target an efficient GPU implementation of subdivision surfaces [ZS00], as pioneered by Pulli and Segal [PS96]. These techniques use a memory-efficient depth-first algorithm in order to refine an arbitrary triangle mesh. They precompute a set of tabulated basis functions for a prefixed depth refinement, one for each possible configuration of the one-ring neighborhood. At rendering time, this table is used for each coarse triangle according to its one-ring neighborhood. A uniform triangle refinement at a prefixed depth is then performed, and the generated vertices are projected on the limit surface. Such a refinement is specific to each subdivision scheme, and can benefit from low level implementations, using either SIMD instructions of modern CPUs [BS02] or programmable GPUs [BS03]. Specific hardware has also been proposed in order to reduce the bandwidth between CPU and GPU [BKs00, dRBA02].

In our previous work, we have proposed the idea of barycentric interpolation to perform uniform mesh refinement [BS05]. A vertex program has been designed to replace each coarse triangle by a precomputed tessellated triangle, which is then displaced according to a procedural function. This system allows efficient implementation of Curved PN Triangles and their extensions [VPBM01, CK03b, CK03a, BRS05] (which use triangular Bézier patches to reproduce a “visually” smooth surface) as well as other alternative procedural mesh refinement techniques. However, as all the other techniques using a direct refinement approach, this solution cannot easily perform adaptive refinement.

Indirect Refinement: This second approach casts mesh refinement as a kind of image processing algorithm [BW06].
Before the introduction of recent unified architectures, fragment processing was much more flexible than vertex processing. Thus, several algorithms have been proposed that consider meshes as textures rather than geometry. Basically, these methods work in three steps: first the input mesh is converted on CPU to an image-based representation. For instance, Shiue et al. \cite{SJP05} start with a two-step subdivision of the initial mesh on the CPU (basically to sufficiently separate vertices with extraordinary valence), and then, unfold each original vertex with its two-ring neighborhood in a 1D texture. Similarly, Bunnell \cite{Bun05} breaks the original surface into small pieces, projecting them on 2D textures. With such an approach, the “geometric” texture can then be upsampled, by using a render-to-texture function and replacing the usual image filtering kernel by the mesh subdivision one. This is done recursively until reaching a given depth or an error bound. Finally, upsampled images are converted back to geometry, rasterized and rendered on screen. These algorithms work well for small refinement depth, but inherit the intrinsic limitations of GP-GPU approaches: they require a conversion of the input model to a specific format and employ intensive multi-pass rendering. When the input is not a mesh but an object with a global parameterization, such as NURBS or T-Spline surfaces, the indirect method proposed by Guthe et al. \cite{GBK05,GBK06} is more efficient, as the parameterization already acts as image coordinates.

Adaptivity and Local Control: Including adaptivity within mesh refinement can strongly improve the overall performance, by reducing the number of polygons in areas classified as less important (e.g. flat areas, far areas, partially hidden areas). Multiresolution mesh representation \cite{Hop96} is based on this notion. Kähler et al. \cite{KHS03} have proposed an interesting curvature-based approach for CPU adaptive mesh tessellation. Nevertheless, adaptive refinement methods are not easily amenable to GPU implementation, due to their highly dynamic adjacency information.

Local control of a given mesh refinement process has been frequently solved by including additional per-vertex/edge/face boolean or scalar tags, which can be used to edit the shape (e.g. crease, tension, bias, etc) of the refined surface around the tagged simplex \cite{BS95,BMZB01,BRS05}. Here, we introduce a similar tagging scheme, but this one is not intended to control the geometry but rather the topology of the refined mesh. This per-vertex tagging scheme is then used to generate adaptive tessellation of the coarse polygons, by employing a similar principle of barycentric interpolation as our previous work \cite{BS05}.

3. Adaptive Refinement Kernel

3.1. Overview

The adaptive refinement kernel (ARK) presented in this paper offers the following properties:

- Standard geometry structures used by rendering APIs (polygon soups or indexed triangle sets) can be employed as-is, without any preprocessing (e.g. global or local parameterization) nor any additional data structures often required by refinement techniques (e.g. half-edge structure).
- Only the coarse mesh is transmitted from the CPU to the GPU. The only required additional data is a simple per-vertex scalar attribute, called depth-tag, that indicates the level-of-detail desired in the vicinity of each vertex. Note that this depth-tagging may be generated either automatically or under user supervision.
- As mesh refinement is performed on-the-fly, on a frame-by-frame and triangle-by-triangle basis, arbitrary level-of-detail can be obtained, even for animated meshes.
- The whole two-stage adaptive mesh refinement (tessellation and displacement) is performed on the GPU, by a single-pass vertex program, which totally frees the fragment shaders for additional visual enrichments.

The workflow architecture used by our ARK is described in Figure 2. The key idea is to precompute all the possible refinement configurations of one single triangle, for various per-vertex depth-tags, and encode them using barycentric coordinates. Each possible configuration is called an adaptive refinement pattern (ARP) and is stored, once for all on the GPU, as a vertex buffer object. Then, at rendering time, the attributes of each polygon of the coarse mesh, as well as the attributes of the displacement function are uploaded to
the GPU and the adequate ARP is chosen according to the depth-tags. Finally, the vertex program simultaneously interpolates the vertices of the current coarse polygon, and the displacement function, by using the barycentric coordinates stored at each node of the ARP. The first interpolation generates the position of the node on the polygon (i.e. tessellation step) and the second one translates it to its final position (i.e. displacement step).

3.2. Depth-tagging

On the CPU-side, the application specifies the usual per-vertex attributes of the mesh (position, normal, color, etc) as well as a specific one; the vertex depth-tag that indicates the level-of-detail desired in the vicinity of each vertex. The depth-tagging process can either be performed once for all for static meshes, or dynamically recomputed at each frame for animated meshes.

Once this vertex depth-tagging has been set, it is employed at rendering time to adaptively refine each coarse polygon, according to a set of precomputed configurations. More precisely, the depth-tags will be used for selecting a per-edge tessellation rate. To ensure crack-free refinement, the tessellation must be consistent on the two side of a given edge. Thus a consistent edge depth-tag is computed simply by taking the arithmetic mean of the two adjacent vertex depth-tags. Moreover, to easily manage general non-triangulated meshes, a centroid split is performed for each coarse polygon with \( n \) vertices to get a set of \( n \) triangles. The depth-tag of the centroid, called face depth-tag is computed as the mean of the \( n \) surrounding edge depth-tags.

Such a tagging approach is very generic, as the tag values can be set according to any metric. In this article, we do not propose new metrics, but rather show how to set the depth-tag according to any existing one. For instance, Figure 3 shows a static tagging generated by using a modified version of the curvature estimator proposed by Rusinkiewicz [Rus04], as well as a dynamic tagging generated by using a simple camera-to-vertex distance metric.

3.3. Adaptive Refinement Patterns

According to the classification of Shiue et al. [SJP05], our technique can be considered as a patch-based refinement. It is derived from our previous \textit{Refinement Patterns} technique [BS05]. To emphasize the difference between both techniques, we propose to rename our previous work as \textit{Uniform Refinement Patterns} (URP). Figure 4 presents the principle of the URP technique. The tessellation and displacement steps are computed on a single-pass vertex program, by a simple barycentric interpolation of a URP, which is a fixed tessellated triangle encoded as barycentric coordinates.

Basically, at rendering time, the attributes of each coarse triangle are uploaded to the GPU and the URP is drawn instead of the coarse polygon. The barycentric coordinates stored at each node of the URP are used to interpolate the per-vertex attributes (e.g. positions, normals) of the coarse triangle, and to output each refined vertex in the graphic con-
text of the currently processed coarse triangle. This virtually generates vertices on GPU and can be seen as a procedural instantiation method for refinement purpose, or as a refinement mapping.

The URP technique strongly reduces the CPU-GPU bottleneck, as only the coarse mesh is transmitted, while the GPU synthesizes the high-resolution mesh on a per-triangle basis. This approach is particularly well-suited for dynamic objects that cannot be refined and stored on the GPU once for all, as well as for procedural displacement textures, that usually require highly tessellated meshes. In this case, the URP technique enables to stream more geometry toward the screen than could even be stored on the CPU or the GPU.

Unfortunately, providing only uniform refinement is a major drawback for most applications, as it is almost impossible to avoid either over-tessellated or under-tessellated meshes, even in the easy case of a moving camera in a static scene. So, we propose here to extend our previous approach by storing on the GPU a set of adaptive refinement patterns (ARP). Basically, the idea is to precompute all the different topological configurations of a refined polygon both for regular and irregular situations, and to encode their nodes in the barycentric space. Then, at rendering time, the low-level API can select the correct ARP, according to the depth-tag configuration of the coarse polygon.

### 3.3.1. ARP for Triangular Meshes

For triangular meshes, it is possible to encode all configurations up to an upper bound of the refinement depth. Since different tessellation rates may appear for different edges of a triangle, the set of ARPs is implemented as a matrix of \( l^3 \) patterns, with \( l \) being the deepest refinement level allowed (left part of Figure 5). This matrix is precomputed and uploaded to the GPU once for all. The quality of adaptivity for a given refinement scheme is usually rated with its support size [Kob00]. The larger is the support, the “smoother” will be the transition between two different tessellation rates, but additional vertices are required (see right part Figure 5).

Our system thus allows any kind of adaptive transition, as soon as its support fits in the area of the coarse triangle. Possible adaptive refined topologies range from border-split patterns to variational angle-maximizing one. In most cases, simple border-split topologies as the one presented at the upper-right corner of Figure 5 offer good results. Note that the ARPs might be harder to convert into triangles strips (lossless topology compression) than regular ones. Thus, an automatic stripping is performed using the STRIPE algorithm [ESV96, RBA05]. The pseudo-code of the algorithm used on the CPU-side for triangular meshes is presented below:

```c
GLuint ARPPool[MaxDepth][MaxDepth][MaxDepth];
void precomputeARPs () {
    generateAndStripARPs (ARPPool);
    sendARPVertexBufferToGPU ();
    bindVertexBuffer ();
    sendToGPU (ARPPool);
}
void render (Mesh M) {
    if (dynamic)
        for each Vertex V of M do
            V.tag = computeRefinementDepth [V];
        for each CoarseTriangle T of M do
            bindIndex (ARPPool[T.v0.tag][T.v1.tag][T.v2.tag]);
            drawElement ();
        }
}
```

### 3.3.2. ARP for General Polygonal Meshes

While the memory footprint remains low when storing refinement patterns for triangles, it becomes a problem for more general polygons. If no care is taken, the number of different configurations to store may quickly become impractical when the tessellation level \( l \) increases. Indeed, as each edge of the polygon includes its own tessellation rate defined by its depth-tag, the number of different tessellation patterns is \( l^3 \) for a triangle, \( l^4 \) for a quadrangle and more generally \( l^n \) for a polygon with \( n \) vertices or edges. One possibility which strongly reduces the total number of configurations is to use constrained depth-tagging, for which the variation among

![Figure 6: ARP factorization for non-triangular patterns.](image-url)
Let us take the general case where the

displacement gl_Vertex, y;

p1 (n1) = void uxyz); *

Moreover, in addition to vertex displacement, p0

are an efficient alterna-

main 5) {

= + w); Moreover, in addition to vertex displacement,

displace 0.

The depth-tag of each inner edge of these triangles is

ond, the polygon is split into a set of

ing process of the initial vertex depth-tags, which will nat-

arily soften abrupt variations of the tessellation rate. Sec-

ond, the polygon is split into a set of \( n \) triangles by linking

each pair of adjacent vertices to the centroid of the poly-

gon. The depth-tag of each inner edge of these triangles is

set to \( \delta \). This guarantees that each triangle only contains two

not-so-different depth-tags \( \delta \) and \( \bar{\delta} \) because they have been

smoothed by double averaging. The inner part of the trian-

gle (green area on Figure 6) will be uniformly tessellated at

the rate provided by the face depth-tag \( \delta \), while the outer

part strip will generate a crack-free junction between level

\( \delta \) and level \( \bar{\delta} \). As will be deeper explained in Section 5,

all the (very reduced) number of possible configurations

for this adaptive triangle strip are concatenated at the end of the

uniform tessellation of the inner part of the triangle, and the

whole data is stored on the GPU as a single index buffer.

Each specific configuration can thus be simply retrieved by

providing an offset in that buffer.

Basically, with our solution, one single strip of tessellated

triangles at the outline of the initial coarse polygon is used

to manage the crack-free junction between different adaptive

levels, while most area of the polygon is tessellated accord-

ing to the face depth-tag. In other words, we solve the adap-

tivity problem on a per-polygon basis, which can thus be

done without complex high-level data structures to encode

the neighboring topology for each polygon. For pathological

cases where \( \delta \) and \( \bar{\delta} \) differ too much, two border strips in-

stead of one may be employed to create smoother transition

between coarse and fine tessellation, and thus better avoid

elongated triangles. Finally, note that since all ARPs are pre-

computed and uploaded once for all on the GPU, rendering

one polygon with uniform tessellation, and one with adap-

tive tessellation, takes exactly the same time, for a equivalent

tessellation rate. This is far from being true with existing

adaptive mesh refinement techniques.

3.4. Adaptive Refinement Shaders

Our kernel uses a specific single-pass vertex program called

Adaptive Refinement Shaders (ARS), that successively per-

forms the tessellation and the displacement steps. During the
tessellation step, the coordinates of the current ARP are used
to generate a barycentric interpolation of the standard per-
vertex attributes (position, normal, etc). Then, during the dis-
placement step, the resulting vertices are displaced using ad-
ditional attributes (e.g. textures for displacement mapping).

Note that since all ARPs are encoded in the barycen-
tric space, refinement shaders are totally independent of the

 topology of the patterns. So, the same shader is used, what-
ever the given ARP. Here is an example in GLSL [KBR04] of

a refinement (vertex) shader which performs a simple pro-
cedural refinement with linear tessellation:

```
const uniform vec3 p0, p1, p2, n0, n1, n2;
float displace (vec3 p) {...}

void main (void) {

    // Tessellation by barycentric interpolation
    float u = gl_Vertex.x;
    float v = gl_Vertex.z;
    float w = gl_Vertex.w; // 1-u-v
    gl_Vertex = vec4 (p0.u + p1.u + p2.u, gl_Vertex.w);  
    gl_Normal = n0.u + n1.u + n2.u;  

    // User Defined Displacement
    float d = displace (gl_Vertex.xyz);  
    gl_Vertex += d * gl_Normal;  

    // Shading and Output
    ...
}
```

Note that the barycentric coordinates may be used for non-
linear interpolation (e.g. quadratic interpolation for normals [VPBM01]). Moreover, in addition to vertex displacement, the same process can further be used to interpolate any other per-vertex attribute during the refinement process. Finally, as the refinement is totally performed on a per-polygon basis, meshes with arbitrary genius and even non-manifold can be directly processed (see Figure 7).

4. Refinement Zoo

In this section, we present various examples of on-the-fly mesh refinement algorithms which have been implemented with our kernel.

4.1. Bézier Smoothing

Curved PN Triangles [VPBM01] are an efficient alternative to usual subdivision surfaces. This method generates

![Figure 7: Adaptive refinement of a deformable genius-4 shape. The refinement provided by the ARK is not restricted to a particular topology, nor manifold conditions.](image-url)
4.2. Displacement Mapping

Recent graphics hardware allows vertex-texture fetches \cite{Fer05}. This means that displaced subdivision surfaces \cite{LMH00} can be easily implemented by storing the displacement in a floating point texture, and accessing it in the second stage of the refinement shader. However, GPU evaluation of subdivision surfaces can be expensive on the vertex shader because it requires complex computation for vertices with high valency (see Section 6). Fortunately, in the work of Lee et al. \cite{LMH00}, the subdivision process is only used for smoothly sampling a base domain for vertex displacement, while the final geometric continuity is expressed by the displacement and not the subdivision. In most cases, Curved PN Triangles \cite{VPBM01} provide a smooth enough base domain compared to genuine subdivision surfaces, with the additional benefit that no local neighborhood has to be transmitted to the vertex shader to achieve the refinement of a given coarse triangle. Figure 9 gives an example of the rendering of such Displaced PN Triangles.

4.3. Procedural Refinement

Geometry synthesis by procedural refinement is clearly one of the best examples that enlightens the strength of our ARK. These techniques often define a very coarse mesh, with complex displacement functions, potentially requiring a very high tessellation rate to correctly sample all high frequency features. Figure 10 shows several examples of such refinement, which only require to transmit a small set of user-defined parameters to define the corresponding procedural displacement function.

4.4. Adaptive Terrain Rendering

While dedicated systems exists for efficiently adaptive rendering of terrains \cite{AH05,LC03}, the ARK allows very simple adaptive refinement of height-field models. We use a basic ground made of few hundreds polygons as a coarse mesh, and upload an high resolution height-field as a floating point texture to the GPU memory. Then, at rendering time, we tag the vertices of the coarse ground using a view-dependent distance metric. Finally, the coarse ground is adaptively tessellated on-the-fly by the ARK and displaced using vertex texture fetch from the height-field texture (see Figure 11).

4.5. Animated Mesh Refinement

Animated meshes are another important application that could significantly benefit from our ARK. Indeed, as mesh refinement is performed on-the-fly, without storage and
This terrain has been rendered at an average framerate of 44 FPS (6M tri.), by using a single height-map texture to displace the refined tessellation. The refinement is driven by a view-dependent depth-tagging. **Top:** Topology for input ground, uniform and adaptive on-the-fly refinement with the ARK. **Bottom:** Final adaptive real-time rendering. Without specific per-object precomputation, an animated mesh just requires a frame-by-frame update of its depth-tag configuration, in addition to usual vertex position update by the application. An adequate adaptive refinement will then be generated at each frame. Figure 12 presents two frames of a face animation sequence with dynamic adaptive refinement. The depth-tagging is based on a local curvature estimation performed frame-by-frame, while the refinement process uses smoothing by Curved PN Triangles over the coarse mesh.

### 5. Implementation and Performance

Our implementation runs under Linux, using OpenGL and GLSL. All tests have been performed on an nVidia GeForce 8800 GTX with 768MB of memory, on an Intel P4 2.4GHz with 1GB of memory.

**GPU Implementation of ARP:** The ARP is the central structure of our system. In order to tightly reach the maximum performance at rasterization time, the ARP is encoded as an indexed vertex buffer of degenerated triangle strips [SWND05], directly on the GPU memory. Moreover, because we use dyadic refinement, each refinement level is actually a superset of the previous one, so we can further reduce the global memory footprint by separating the geometry from the topology. A vertex buffer is used to encode all the geometry by storing the set of barycentric coordinates for the nodes that belong to the deepest uniform ARP. Then the topology of any given ARP is encoded by using an index buffer, as an indexed strip over this maximum configuration. So, at rendering time, when the application selects a given ARP for refining a coarse triangle, the only action performed by the API is to bind the corresponding index buffer and set the correct offset, while always keeping the same vertex buffer, which guarantees cache-friendly access.

Regarding memory usage, on the CPU side, the only memory overhead comes from the storage of the set of ARP identifiers. This overhead is extremely small and totally independent of the current 3D scene. For instance, if the maximum refinement level is set to 10 (which offers a maximum refinement of $1024 \times 1024$ sub-triangles for each coarse triangle), the precomputation (all ARP generation) time is less than half a second, the main memory overhead is less than 4kB. On GPU side, the memory overhead required to store the set of ARP at this resolution is about 26MB.

For uniform refinement patterns, we already mentioned that in the case of deep refinements, rendering performances were very close to the one obtained with static meshes (i.e. refined during a preprocessing step and stored on the GPU) [BS05]. We have observed a similar behavior for our new adaptive refinement kernel.

Note that in restricted conditions, with 16-bit precision (e.g. PDAs), our ARP encoding allows a maximum refine-
ment level of $256 \times 256$ for each coarse triangle. At the other extremum, with a modern GPU and very high resolution displays, we have experimented real-time performance when using up to $2048 \times 2048$ tessellation for each coarse triangle. Even higher resolutions can easily be reached, since the ARK fully runs in object space.

Figure 13 presents the rendering frame rate obtained for various models. The measure integrates the tessellation step and a simple procedural displacement step for an animated mesh. A dynamic adaptive refinement has been performed frame-by-frame, based on an approximated local curvature estimation, combined with a view-dependent refinement bound. Compared to our previous URP technique, our new ARP scheme offers a gain ranging from 250 to 460% depending on the model, while providing the same final image quality. This can be explained by a finer gradation of the tessellation, avoiding rendering of unnecessary small triangles (e.g. flat areas, far areas). In many cases, the quality is even better, since the aliasing of over-tessellated meshes (more than one triangle for a pixel) is strongly reduced. In extreme cases, the gain can even reach one order of magnitude, when the depth-tagging is static and the input mesh is very coarse (see Figure 10, for instance). Compared to our optimized CPU implementation of adaptive refinement, our GPU refinement kernel improves the frame rate between one and three orders of magnitude, depending on the overall complexity of the refinement.

Figure 14 shows the frame rate obtained for a target refined mesh made of 1M triangles, under various input size vs refinement depth ratios. It clearly appears that coarse meshes with high refinement depth totally outperforms medium meshes with low refinement depth, for the same total number of triangles. This comes from the fact that for the latter, transmission of input polygons attributes becomes a bottleneck on the graphics bus. At the other extreme of the spectrum, when the target shape can be described by a very coarse mesh with deep refinement, the ARK runs in an optimal context and completely saturates the GPU processing horsepower.

**Limitations:** The technique presented here has essentially two main limitations. First, the refinement depth must be sufficient to avoid the bottleneck involved in the transfer of the per-vertex attributes. Second, on elder graphics hardware, vertex texture fetch is slow, which limits applications such as terrain rendering and displacement mapping. Fortunately, this restriction has recently disappeared with the introduction of unified shader architectures on graphics hardware. For instance, the terrain render application at Figure 11, which uses intensively texture access from vertex shader, runs at about 2 FPS on an nVidia Geforce 6800 and 44 FPS on an nVidia Geforce 8800 (unified shader architecture), for an average refinement depth of 8, which produces about 6M polygons.

Another concern may be the question whether the depth tagging should be better performed on the GPU instead of the CPU. This could be done by using a preliminary rendering pass that would store vertex depth-tags in a texture. However, this would involve a strong limitation on the kind of depth-tagging that can be implemented, as many useful information may only be available for the application running on the CPU. Moreover, as the depth-tagging is performed on the coarse mesh, the computation overhead remains negligible, particularly in the case of deep refinements.

**Refinement Kernel vs Geometry Shader:** DirectX 10 technology [Bly06] has introduced a new geometry shader (GS) stage in the hardware rendering pipeline. The first graphics devices including these functionalities have been launched at the end of 2006. Even if the GS can obviously be used to perform mesh refinement, its features are quite different from the way we have structured our ARK. The main limitation when using the GS to perform mesh refinement is that the level of geometry upsampling is hardware limited and fixed. For instance, only 1024 floating point numbers can be output with current specifications [Bly06]. This is far from being able to tessellate up to $2048 \times 2048$ triangles per coarse polygon for instance, as with our ARK. Multi-pass GS rendering may be employed to reach deeper refinement, but it would obviously strongly reduce overall performance. In practice, as mentioned by hardware manufacturer [Gre06], it is not even possible to reach the single pass upper bound, without observing a huge performance degradation.

Even without the limit of geometry upsampling, implementing adaptive mesh refinement with the GS would also require to correctly manage crack-free junctions between different tessellation rates. With our precomputed ARPs, this problem is solved once for all and stored, while the GS would have to generate consistent topologies on-the-fly and thus require complex shader code. Notice that, as the GS implements a superset of the vertex shader functionalities, the solution provided by the ARPs can straightforwardly be implemented on the GS.
Actually, we consider that the GS stage represents a complement to the ARK, rather than an alternative. By combining both approaches, one may generate more complex refinement in a two stage process. First, at the VS stage, the ARK tessellates and displaces a base domain (e.g. apply Bézier smoothing on very coarse meshes) and then additional vertices are inserted at the GS stage (e.g. local extrusion to create hairy objects). We can also imagine using the GS for low refinement depth where the ARK is less efficient, and then switch to ARK to get high refinement depth when needed.

6. Conclusion

We have presented a simple and efficient GPU kernel for adaptive geometry synthesis by mesh refinement based on a generic depth-tagging process, that makes it suitable for any refinement control that can be expressed on a per-vertex basis (e.g. curvature, view-dependent LOD, area of interest penalty, local estimation of displacement variation, etc). We have introduced Adaptive Refinement Patterns and Adaptive Refinement Shaders, and have shown that their combination allows the implementation of various kind of dynamic refinement, with almost no modification of the rendering loop at application level. The CPU processing is reduced to the transmission of (dynamic) coarse meshes to the GPU, eventually combined with additional dynamic data for driving the refinement.

The kernel allows very deep adaptive refinement, using a single-pass vertex program. It does not impose any conversion of input mesh (such as local or global parameterization) and allows further on-GPU geometric processing, since it consistently performs geometry synthesis in object space. The solution is more efficient and even more flexible than prior software-based methods. In practice, the benefit of the ARK is proportional to the depth of the adaptive refinement. The kernel permits to “saturate” the GPU with geometry to draw, and with its intrinsic CPU-to-GPU streaming principle, it is possible to draw a refined surface almost independently of the amount of available memory, either on CPU or GPU (the ultimate limitation is represented by the storage on GPU of the set of ARP required for the chosen depth).

Our refinement kernel exhibits an interesting collaboration between the CPU and GPU (the global analysis at coarse resolution is let to the CPU for depth-tagging, while local fast refinement is performed on the GPU, driven by these tags). This corresponds to the idea of shared rendering workload between powerful multi-core CPUs and GPUs, as stated by Pharr [Pha06]. Future graphics hardware and API developers seek for refinement methods based on generic barycentric interpolation, as mentioned by Sloan [Slo06]. Thus this paper can also be considered as a first step, performing a software emulation of such future on-board refinement shader stage.

Future work: As future work, we would like to explore single-pass refinement of subdivision surfaces using the ARK. As already mentioned above, genuine subdivision surfaces are very different to mesh smoothing techniques based on Bézier patches, as the refinement of each coarse polygon is usually implemented in a recursive procedure depending on its one-ring (or even two-ring neighborhood). We are currently working on two single-pass approaches to this problem:

- **exact subdivision surface rendering**: as stated by Stam [Sta99], an exact evaluation of limit surfaces at arbitrary parameter is possible by tiling the parameter values domain in a set of triangular patches and performing an eigen analysis of the so-defined parameterisation. Unfortunately, in the case of a triangle indexing an extraordinary vertex, the implementation requires a huge amount of additional data for each triangle, which is no more compatible with efficient rendering. So, we are developing a hybrid CPU-GPU implementation which delays a large part of the computation on the ARK, keeping the horsepower of modern multi-core CPUs for non regular cases. Figure 15 gives a preliminary example of our current work on a Loop subdivision with our kernel.

- **approximate subdivision surface rendering**: we are also developing approximations of subdivision surfaces for interactive applications, which can be implemented efficiently with the ARK, reaching real-time performances for millions of polygons output while being visually very similar to exact subdivision surfaces. See [BS07] for the first results.

References


Appendix A: Low level API extensions
The presented kernel can be integrated at the driver level, in any standard graphics API such as OpenGL, without any additional hardware capabilities. In this case, the control interface is a reduced set of functions:

- glARKinit(GLuint maxLevel): builds the set of ARP (special indexed vertex buffers) on GPU, and stores the corresponding identifiers, indexed by depth-tags.
- glEnable(GL_ARK): when activated, the ARK refinement will replace any triangle drawing call by the corresponding ARP.
- glDisable(GL_ARK): restore usual OpenGL behavior;
- glDepthTag1i(GLuint d): set the current vertex depth-tag state (for upcoming vertices).

The fixed pipeline would provide a simple linear refinement, which can then be tuned by setting user specific adaptive refinement shaders. With this set of functions plus some additional commodity callbacks, the use of the ARK is totally transparent to programmers (direct port of existing source code). Alternatively, finer control of refinement can be provided through specific functions (drewARP () for instance) in order to mix refined and regular drawing calls without switching the mode.