

# Dynamic Mesh Refinement on GPU using Geometry Shaders

Haik Lorenz  
Hasso-Plattner-Institute  
University of Potsdam  
August-Bebel-Str. 88  
14482 Potsdam, Germany

haik.lorenz@hpi.uni-potsdam.de

Jürgen Döllner  
Hasso-Plattner-Institute  
University of Potsdam  
August-Bebel-Str. 88  
14482 Potsdam, Germany

doellner@hpi.uni-potsdam.de

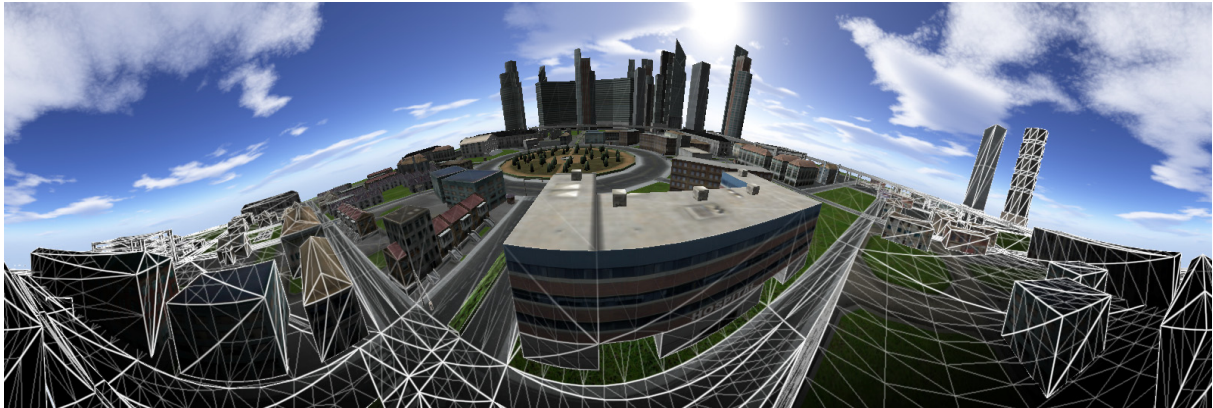


Figure 1. A 360° panorama view generated with dynamic mesh refinement. The frame's input mesh is hinted with thick lines. Actually rendered primitives have thin lines.

## ABSTRACT

This paper presents a real-time rendering technique for dynamic, incremental 3D mesh refinement. The technique can be applied to any triangulated 3D mesh with arbitrary topology and connectivity. The functionality relies on geometry shaders that are used to amplify or remove geometry based on precalculated refinement patterns. For each triangle, the instantiated refinement pattern is selected dynamically. Due to limitations of current hardware, on-the-fly pattern instantiation cannot be implemented on the GPU. Instead, the complete refined mesh must be generated through pattern copying. We propose an incremental approach where the refined mesh is generated by using the previous refined mesh as primitive source. This algorithm runs exclusively on the GPU and requires no continuous data exchange between CPU and GPU. Due to the necessary mesh generation, the approach is particularly suitable for applications with small refinement levels. It complements traditional pattern-based refinement approaches that deliver high throughput for large refinement levels, but incur a substantial CPU-GPU communication overhead otherwise. Interesting applications include view-dependent mesh smoothing and interactive non-planar projections. In these areas, our algorithm enables efficient vertex-based implementations due to adaptive refinement.

## Keywords

Geometry shaders, GPU, mesh refinement, refinement patterns, barycentric coordinates

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Copyright UNION Agency – Science Press, Plzen, Czech Republic.

## 1. INTRODUCTION

Geometry specification and its efficient rendering is a crucial aspect of computer graphics. For high-performance rendering, most parts of the graphics pipeline have become implemented in graphics hardware and are fed with primitives through standardized APIs. Real-time applications are forced to limit the number of primitives passing through the graphics pipeline. For fine meshes, this translates to *mesh simplification*, where a detailed mesh is

degraded into a coarse mesh while limiting visual artifacts. The simplification is either offline (statically) or online (dynamically). In general, level-of-detail algorithms (LoD) follow this paradigm.

If geometry is available in form of some compact functional description, e.g., subdivision surfaces, *geometry synthesis* is required for generating a mesh representation for processing by graphics hardware. Online synthesis allows for adapting the mesh to view parameters and thus limiting its size. In real-time applications, geometry synthesis is often posed as *mesh refinement* problem where the intended geometry is defined as function over a coarse base mesh [Bou07]. The base mesh is then refined so that the geometry function can be evaluated per vertex with sufficient frequency in screen space.

The same idea of using mesh refinement to ensure sufficient vertex frequency can help to map computer graphics algorithms to graphics hardware. In a *vertex-based approximation* scheme, complex, non-linear computations are evaluated per vertex and linearly interpolated in between. Examples include global illumination or non-planar projections.

Our motivation is the use of mesh refinement for view-dependent geometry synthesis and vertex-based approximation techniques. A number of hardware-accelerated techniques have been proposed for generic [Bou07] or special-purpose [Bun05, Shi05] mesh refinement. However, all of these techniques require continuous data transfer between CPU and GPU proportional to the input mesh size. While exhibiting great performance for high refinement levels, this makes them inefficient for view-dependent techniques, whose refinement level is dynamic and rather low on average.

We propose a dynamic approach fully operating on the GPU to make mesh refinement efficient for view-dependent problems. Our technique uses the idea of *barycentric refinement patterns* as proposed by Boubekeur et al. [Bou05b, Bou07] and follows their coarse algorithm. However, we relieve the CPU from selecting and instancing the refinement patterns by exploiting recently introduced *geometry shaders* [Bly06]. This enables us to dynamically amplify or remove geometry on the GPU, but at the same time shader limitations require us to create and store a representation of the complete refined mesh. Even though being able to provide arbitrary refinement levels, the mesh creation makes our approach most efficient for low average refinement levels and thus complements Boubekeur’s method, as suggested in [Bou07].

Our algorithm exhibits four major properties that make it particularly suitable to add view-dependency to both, geometry synthesis and vertex-based approximation techniques. It is

- *generic*, i.e., it operates on a triangular mesh of arbitrary topology and connectivity. It does not impose any restrictions on the refinement strategy or rendering technique and thus is adaptable to a wide range of applications.
- *pattern-based on triangle level*, i.e., it uses precalculated refinement configurations to replace each original input triangle. The patterns can have any suitable structure. Patterns are not limited in their vertex count.
- *dynamic*, i.e., it selects each triangle’s refinement pattern for each frame. Then, a refined mesh is build and rendered accordingly.
- *constant and minimal in communication overhead*, i.e., it fully saturates the GPU regardless of the input or rendered mesh size.

Our first contribution is the removal of communication overhead. While the first three properties are inherited from Boubekeur’s approach, we remove the need for per-triangle draw calls despite a non-uniform refinement. Our second contribution is an incremental refinement scheme that enables fast arbitrary geometry amplification with geometry shaders despite their output limit.

The paper is organized as follows: Section 2 gives an overview over related work, Section 3 briefly introduces geometry shaders, Section 4 gives an in-depth description of our generic refinement algorithm along with its limitations, Section 5 demonstrates the refinement for two examples, Section 6 provides performance results and comparisons, and Section 7 concludes.

## 2. RELATED WORK

This section concentrates on hardware-accelerated mesh refinement algorithms. We distinguish two groups based on application dependency.

In application-independent refinement, Boubekeur’s work [Bou05b, Bou07] is of most importance to our method, as described earlier. His methods are applicable to any triangulated mesh. The refinement presented by Bokeloh and Wand [Bok06] needs to restrict the input to rectangular patches that become hierarchically subdivided. They cast the refinement problem as image upscaling and use fragment shaders for implementation on the GPU. The new image data is then interpreted as vertex data for rendering.

Application-dependent refinement is used for geometry synthesis. For example, various approaches to subdivision surface rendering exist. Hardware-accelerated methods use patch-based approaches as introduced by Pulli and Segal [Pul96]. Bunnell [Bun05] describes a GPU implementation for adaptive tessellation including displacement mapping. While this approach uses a 2D texture to

represent surface patches, Shiue et al. [Shi05] uses a spiraling scheme to unfold each patch into a 1D texture. Both methods then use image operations to upscale the texture for actual geometry synthesis and recast the final texture as vertex buffer for rendering.

Guthe et al. [Gut05] present a synthesis approach to trimmed NURBS. They use predefined sampling grids similar to our barycentric refinement patterns to render a hierarchy of bi-cubic approximation surfaces. They focus on artifact-free and efficient trimming of the resulting rendering.

Hardware-accelerated mesh smoothing has been described by Losasso et al. [Los03]. They use geometry images [Gu02] to represent the control points for a B-Spline-based surface approximation. The control points are then transformed to a refined surface by performing image convolution operations on GPU. Finally, the image is interpreted as vertex data and rendered.

A new generic approach to rendering has been proposed by Whitted and Kajiya [Whi05]. They outline changes to the current graphics hardware to enable processing of fully procedural geometry. As they propose point sampling in screen space, such a rendering processor eliminates the need for mesh refinement.

Geometry shaders are a recent extension of the hardware graphics pipeline [Bly06]. So far, their use has rarely been reported in literature. Tariq [Tar06] presents some examples provided by nVidia, including fur rendering, cloth animation, and isosurface extraction. DeCoro and Tatarchuk [DeC07] exploit geometry shaders for mesh simplification through vertex clustering. An interesting future development is the proposal of a dedicated hardware tessellation unit [Tat07].

### 3. GEOMETRY SHADERS

Geometry shaders [Bly06] introduce an additional processing unit in the graphics pipeline. This unit is located after vertex processing and before clipping and rasterization. It operates on whole primitives (points, lines, or triangles with or without newly introduced adjacency information) having random access to all a primitive's transformed vertices and their attributes. The geometry shader program is invoked once for each assembled input primitive. The program then emits zero or more primitives, whose type is fixed but can differ from the input primitive type. The output completely replaces the input, allowing for geometry amplification and deletion.

The amount of output data cannot exceed a certain hardware limit, typically 4096 bytes. The programmer can further restrict the maximum output size, since the shader's performance and efficiency

improves significantly with smaller maximum output sizes, regardless of the actual data output per invocation.

An additional feature introduced together with geometry shaders is *transformation feedback*, a.k.a. *stream out*. This allows for capturing transformed vertices into a buffer in sequential order before clipping but after the geometry shader stage. The exact number of vertices captured in a buffer can be queried after command completion. As the capture buffer resides in GPU local memory, it can be used as input to subsequent rendering commands without copying. Optionally, the vertices can be discarded after capturing without rasterization taking place.

## 4. DYNAMIC MESH REFINEMENT

### 4.1 Overview

Our refinement approach is pattern-based, i.e., each triangle of the original mesh is replaced by a generic precomputed refinement pattern consisting of *subtriangles* before rendering. Each selected pattern is then adapted to and rendered instead of its originating triangle.

Except for pattern selection, our approach works on single triangles only. Thus, we impose no restrictions on the original triangular mesh's topology or connectivity. In conjunction with a local pattern selection scheme, e.g., based on the screen space edge length, the dynamic mesh refinement is instantly applicable to any (indexed or non-indexed) triangle soup accepted by standard graphics APIs. No additional attributes or data structures (e.g. half-edge structures) are required.

The pattern selection is computed on GPU for each frame and can follow any scheme suitable for the target application. Moreover, the selection can easily take backface and view volume culling into account to avoid unnecessary refinement.

The core of our method is an incrementally updated intermediate mesh of all subtriangles. It is motivated by three observations. First, current GPUs are not capable of dynamically instantiating geometry without the control of the CPU. Second, efficient use of geometry shaders requires to minimizing each invocation's output by all means (cf. Figure 8). Third, in an incremental approach the geometry shader's output limit only limits the growth of a triangle's refinement instead of its size.

The intermediate mesh is a concatenation of one plain pattern copy per input triangle. During update, each triangle's existing copy is replaced by the newly selected one. This update is computed on subtriangle level using geometry shaders on the GPU without the need for CPU control and thus without continuous communication between CPU and GPU. This removes the bottleneck of Boubekeur's

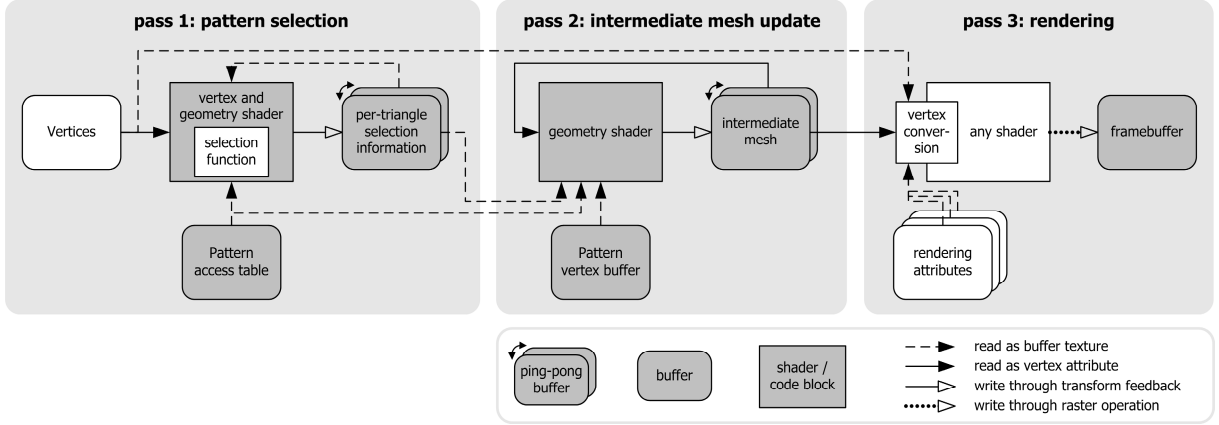


Figure 2. Algorithm outline. White boxes mark application dependent parts.

method. However, the need for creating and storing the complete intermediate mesh favors applications that need only low average refinement levels.

Finally, the whole intermediate mesh is rendered. Only then, the generic pattern subtriangles are adapted to their originating triangles. Vertex positions and other rendering attributes are computed and immediately used for further processing. This includes per-vertex approximations or the evaluation of surface descriptions in geometry synthesis. To a subsequent rendering technique, refined vertices become indistinguishable from original vertices after conversion from their barycentric form. Thus, our method can be used for any rendering effect.

The result is a three-pass-algorithm as outlined in Figure 2. Application independent parts are shown in gray. Each pass requires exactly one vertex array draw call with all information residing in GPU local memory. Hence, the CPU utilization and communication costs between CPU and GPU are negligible and constant regardless of the input or intermediate mesh’s size. In the following, we introduce our refinement patterns and explain each pass in detail.

## 4.2 Refinement Patterns

Similar to [Bou07], each refinement pattern represents a triangle tessellation encoded in barycentric coordinates. A barycentric vertex  $(u, v, w)^T$  represents a point inside a triangle as weighted sum of the triangle’s vertices with  $u + v + w = 1$  and  $u, v, w \geq 0$ . For efficient storage, the third coordinate can be omitted as  $w = 1 - u - v$  holds.

The exact pattern structure is application-dependent. As all patterns are precomputed, any suitable structure can be used. After generating all patterns, the resulting barycentric subtriangles are stored in a combined vertex buffer. A separate table buffer stores for each possible refinement pattern

its triangle count and starting location in the combined vertex buffer.

In contrast to [Bou07], our method does not allow for optimizing refinement patterns through the use of indexing. If we used indexing, the same pattern vertex index in different pattern copies referred to different original triangles. As we render the whole refined mesh with a single draw call, the graphics hardware then might wrongly reuse cached vertex shader results belonging to another original triangle. Non-indexed stripping can be used provided that only a single strip for the whole refined mesh is generated [Eva96, Reu05, Dia06], and that artifact-free rendering is achieved.

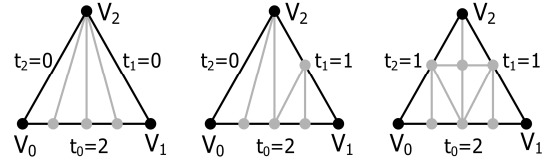


Figure 3. Some sample refinement patterns.

Our examples in Section 5 use patterns controlled by their edges’ tessellation degrees  $t_0$ ,  $t_1$ , and  $t_2$  with a regular and homogeneous interior triangulation. The borders are subdivided in a recursive fashion into  $2^{t_i}$  sections with  $t_i \in [0; t_{\max}]$  denoting the edge’s refinement depth. This results in  $t_{\max}^3$  possible refinement patterns. Some examples are shown in Figure 3.

## 4.3 Pattern Selection

Pattern selection is the first pass of our algorithm. For each input triangle, it selects one pattern from the available precomputed refinement patterns. The selection function is fully application-dependent and maps each triangle to a pattern index. However, usually two goals are desired: a crack-free and minimal refinement.

Since our approach uses a local refinement, cracks can only be prevented by ensuring identical tessellation of the shared edge of two adjacent

triangles. This can be achieved by selecting a pattern based on each edge's tessellation degree  $t_i$  as implemented in the examples.

Ensuring a minimal refinement obviously improves total performance. Most importantly, hidden triangles can be left unrefined to save computation time. It is not possible to omit such triangles altogether, which is explained in Section 4.4. GPU-friendly hidden triangle detection methods include backface and view volume culling. Additionally, any screen space based selection function inherently limits the applied refinement to the current view's demands. Examples include screen space error estimators for subdivision surfaces [Bun05], height fields [Lin96], or texture deviation [Coh98].

Pattern selection is implemented in a geometry shader operating on triangles, whose input depends on the requirements of the selection function. Usually, this includes only a subset of the available rendering attributes, such as vertex position and normal. Additionally, the previous frame's selection is fed into the shader. Its output only contains 3 values per triangle: the selected pattern's index  $p$ , the number of required subtriangles  $s_r$ , and the number of available subtriangles  $s_p$  from the previous frame. For proper operation, the pattern growth, i.e., the ratio  $s_r$  to  $s_p$ , must not exceed the output limit of the update pass, even though this might introduce temporary cracks. Instead of being rendered, this output in form of point primitives is captured in a buffer (using stream out or transformation feedback) and then discarded before rasterization.

The previous frame's selection is accessible through the use of *buffer textures*. This enables a shader to randomly access an arbitrary buffer residing on the GPU via the 1D texture interface. With the primitive ID as index, a triangle's number of subtriangles can be read from the previous frame's selection buffer and copied to the number of available subtriangles for the current frame.

#### 4.4 Intermediate Mesh Update

The update pass uses only the previous frame's intermediate mesh consisting of subtriangles as primitive input. Each subtriangle is identified by the original mesh's triangle id  $i$ , and the subtriangle id  $j$  within the pattern. Together with the barycentric coordinates  $(u,v)$ , this information is stored in every vertex. Additionally, the geometry shader has access to the selection pass's output buffer, the precomputed pattern vertex buffer, and the accompanying table buffer via buffer textures.

For each original triangle, the update pass completely replaces an existing refinement by a

copy of the newly selected refinement pattern. The copying task is evenly distributed across all respective subtriangles, such that the output of a single geometry shader invocation is minimized.

For each subtriangle  $(i, j)$ , the geometry shader fetches the original triangle's pattern selection  $(p, s_r, s_p)$ . It can then determine the corresponding range of subtriangles from pattern  $p$ . It emits each new subtriangle by reading its barycentric vertices from the pattern vertex buffer and emitting them augmented by the triangle id  $i$  and a new subtriangle id  $j'$ . Similar to pass 1, the shader output is captured in a buffer and discarded before rasterization.

For proper refinement, this algorithm requires at least one subtriangle per original triangle. Once an original triangle is lost from the intermediate mesh it can never reappear. Consequently, hidden triangles must not be omitted but kept in the intermediate mesh as a single subtriangle. Additionally, the intermediate mesh must be initialized once to one subtriangle per original triangle.

The buffers for storing the intermediate mesh need to be sufficiently sized in advance. Transformation feedback allows for buffer overrun detection, but for best performance, buffer reallocation should be avoided.

#### 4.5 Rendering

The final pass comprises the conversion of the intermediate mesh's barycentric vertices to the intended geometry and the actual image generation. It is fully application dependent.

The conversion part is of most interest to geometry synthesis, as it implements the surface description evaluation. E.g., it can use the barycentric coordinates for procedural generation, as texture coordinates for displacement mapping, or as parameters for spline surface evaluation. For vertex-based approximations, usually a simple weighted sum for interpolation in object space is sufficient. If an interpolation in screen space is desired, the barycentric coordinates require a preceding transformation, e.g., by hyperbolic interpolation [Bli92].

Image generation is not affected by our mesh refinement. Since any rendering attribute can be computed during vertex conversion, any rendering technique is applicable. This includes multi-pass techniques, even though each additional pass might require its own vertex conversion. Alternatively, conversion results can be stored using transformation feedback and reused in each subsequent pass.

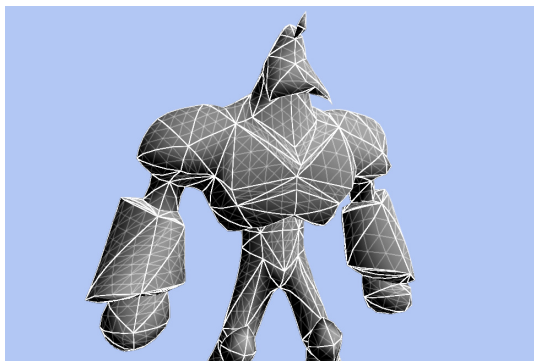
## 5. APPLICATIONS

For usage, pass 1 and 3 from our algorithm need to be adapted (white boxes in Figure 2). In pass 1, a suitable pattern selection function needs to be implemented in a shader. Pass 3 implements the application-specific rendering. This pass can use the original shaders as if no mesh refinement was in place. Only the vertex shader portion needs to be extended by a vertex conversion function converting the generic barycentric pattern coordinates to the actual vertex attributes as needed for further processing.

In the following, we demonstrate our refinement for one geometry synthesis and one vertex-based approximation technique.

### 5.1 Curved PN-Triangles

Curved PN-Triangles have been proposed by Vlachos et al. [Vla01] as a simple heuristic to smoothing a triangular mesh. It has been designed with a dedicated hardware implementation and readily available input in mind [ATI01]. It only uses vertex positions and normals to construct a triangular Bézier surface over the triangle. This surface is then used for geometry synthesis. For many coarse meshes, this method instantly generates visually pleasing results without any change to the original mesh. A respective tessellation unit can be found on some graphics hardware. The original approach has been further investigated, e.g., by Boubekeur et al. [Bou05a] or Choi et al. [Cho04].



**Figure 4. PN-triangles. Thick lines show the original mesh, thin lines the refined mesh.**

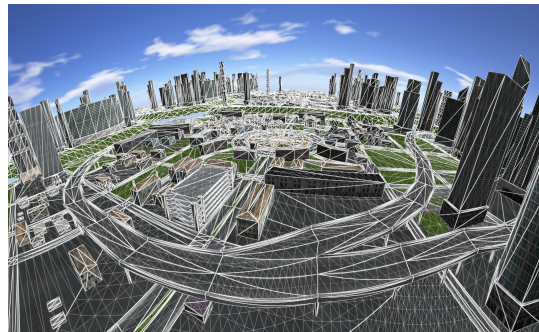
We demonstrate an implementation of the original approach based on our dynamic mesh refinement. Since the curved PN-triangle surface closely follows the original mesh, the original triangle's screen space edge length is a reasonable basis for a view-dependent metric. At the silhouette, this is not sufficient as some minimal refinement should be retained to prevent popping artifacts. Consequently, the normal orientation is incorporated into the pattern selection. In pass 3, we use the formulas given in [Bou05b] to calculate the refined vertex's

position and normal from the barycentric coordinates. The result is shown in Figure 4.

### 5.2 Cylindrical Projection

Current graphics hardware relies on linear interpolation (in homogeneous space) during rasterization. This prevents a straight-forward vertex-based implementation of non-planar projections or view deformations. The actual result can only be approximated due to the missing correct non-linear interpolation. Traditionally, in real-time applications an image space solution involving multiple render-to-texture passes and subsequent image warping [Yan05] has been used. This solution trades approximation errors for warping related interpolation errors.

As alternative, Spindler et al. [Spi06] proposed “camera textures”, a generic object space approach to per-vertex view deformation. If the input mesh is sufficiently dense, approximation errors become negligible. For coarse meshes, they suggest the use of a dynamic mesh refinement.



**Figure 5. A cylindrical projection with 160° field of view. Our refinement is usable with any rendering technique. Geometry is hinted as in Figure 4.**

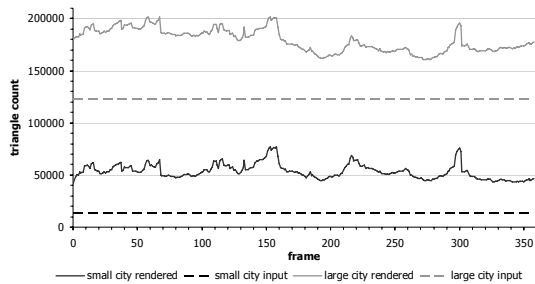
We implement a per-vertex cylindrical projection to demonstrate a vertex-based approximation relying on mesh refinement. The pattern selection function is based on an edge's horizontal screen length. Since the vertical direction does not introduce approximation errors (it uses a perspective projection), this reduces the overall refinement level. Vertex conversion in pass 3 is a simple weighted sum. Figure 5 shows the result and hints both, the unrefined and refined mesh. This example also exhibits the use of additional rendering attributes (texture coordinates etc.).

## 6. RESULTS

For performance evaluation, we compare our method (DMR) to [Bou07] (ARP) for the cylindrical projection described in Section 5.2. Both implementations use identical refinement patterns, represented as non-indexed triangles for DMR and as indexed degenerated trisrips for ARP. Our pattern selection pass (pass 1) with an

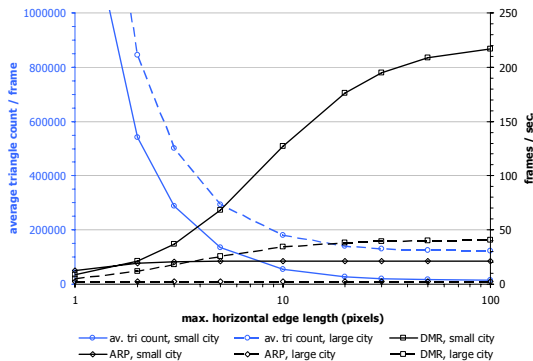
additional buffer readback has been added to ARP to enable identical dynamic refinement. Both methods are implemented with one frame latency per pass to hide dependencies between passes.

The measurements have been taken on a PC with an AMD Athlon 64 X2 4400+ with 2GB RAM and an nVidia GeForce 8800GTS with 640MB RAM. The viewport size was 1600x1200. For optimal performance, the programs have been forced to run on a single core during measurements. The measurements use the same flight path through a textured small (13639 triangles) and large (122751 triangles) city data set.



**Figure 6. Triangle count for the test animation.**

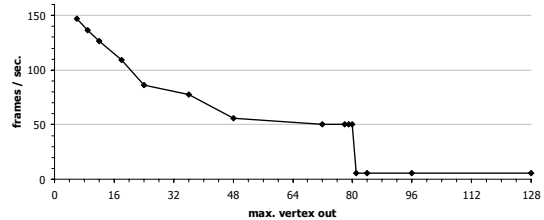
Figure 6 shows the input and rendered triangle count for each frame of the flight path at a maximum horizontal edge length of 10 pixels. It shows the dynamic view-dependent refinement. As expected, the refinement ratio is rather low and almost constant. Since the small data set is only a portion of the large one, the curves look very similar.



**Figure 7. Cylindrical projection performance for varying refinement settings. The x-axis uses a logarithmic scale.**

Figure 7 shows the resulting frame rates and average triangle counts for varying maximum edge length. For the tested use cases, Boubekeur’s method is almost independent of the rendered triangle count and only depends on the input triangle count, i.e. CPU-GPU communication is the method’s bottleneck as described in [Bou07]. Our method scales with both, input and rendered triangle count. By removing the communication

overhead, our method can always fully saturate the GPU while ARP requires a sufficient refinement ratio for that. Consequently, our method outperforms ARP for reasonable error bounds despite the more complex algorithm. Note, that we were not able to reproduce the frame rates for ARP reported by [Bou07]. This has no impact on its general performance behavior.



**Figure 8. Performance results for varying geometry shader output limit.**

Figure 8 shows the geometry shader’s performance for different output limits, measured for the small city data set at a maximum horizontal edge length of 10 pixels. The actual refinements and thus geometry shader outputs for pass 2 are identical for all test runs (cf. Figure 6), only the allowed maximum vertex output varies. Up to a maximum output of 80 vertices, the curve drops as expected. Starting from 81 up to the maximum of 1024 vertices (not shown), the frame rate suddenly drops to a constant 5.75 fps. This curve shows the importance of choosing a sensible maximum vertex output. We use 12 vertices, i.e., 4 triangles, as trade-off to allow for fast pattern growth (e.g., for an invisible triangle entering the viewport) and high frame rates.

## 7. CONCLUSIONS

We presented a novel approach to generic mesh refinement that first leverages the power of geometry shaders to run exclusively on the GPU in an incremental multi-pass scheme. Without continuous communication between CPU and GPU, our method is most efficient for problems with large input triangle meshes and/or low refinement ratios. In particular, view-dependent approaches to geometry synthesis or vertex-based approximations profit from our approach. Nevertheless, it does not replace previous methods, but rather complements them as its efficiency drops with higher refinement ratios.

Our current implementation largely depends on the design of DirectX 10 class GPUs. It could be improved by using indexed trisrips for the refinement patterns. For that, a new hardware feature is required. Similar to the existing “primitive restart” feature, a prospective “next instance” feature could enable a geometry shader to emit blocks of indices separated by a special index

to a single buffer. During vertex fetch, this special index increments the instance ID instead of provoking a vertex. In a subsequent pass, the blocks could then be identified as separate instances and processed with varying shader parameters accordingly.

Future developments, such as a separate programmable hardware tessellation unit [Tat07], will hopefully ease generic mesh refinement and improve its performance. Our future work concentrates on applications for mesh refinement. In particular, we explore the use of vertex-based approximations as alternative approach to existing solutions. The cylindrical projection described in Section 5.2 represents an example and initial result that competes with traditional image warping techniques.

Finally, we explore a unified approach to mesh refinement and mesh simplification. We hope to find a continuous level-of-detail algorithm that not only simplifies distant meshes to gain performance, but also refines close meshes to gain visual detail. For example, outdoor scenes could benefit from such an algorithm as only a single medium sized mesh per object can be used for all rendering.

## 8. REFERENCES

- [ATI01] ATI Inc. TRUFORM white paper. 2001.
- [Bli92] Blinn, J. Hyperbolic Interpolation. *IEEE Computer Graphics and Applications* 12, No. 4, pp. 89-94, 1992.
- [Bly06] Blythe, D. The Direct3D 10 system. In *Proc. of ACM SIGGRAPH*, ACM Press, pp. 724-734, 2006.
- [Bok06] Bokeloh, M. and Wand, M. Hardware accelerated multi-resolution geometry synthesis. In *Proc. of the 2006 symp. on Interactive 3D graphics and games*, ACM Press, pp. 191-198, 2006.
- [Bou05a] Boubekur, T., Reuter, P. and Schlick, C. Scalar Tagged PN Triangles. In *Eurographics (Short Paper)*, pp. 17-20, 2005.
- [Bou05b] Boubekur, T. and Schlick, C. Generic mesh refinement on GPU. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS conf. on Graphics hardware*, ACM Press, pp. 99-104, 2005.
- [Bou07] Boubekur, T. and Schlick, C. A Flexible Kernel for Adaptive Mesh Refinement on GPU. *Computer Graphics Forum Online Early Articles*, 2007.
- [Bun05] Bunnell, M. Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping. In M. Pharr (eds.). *GPU Gems 2*, Addison-Wesley, pp. 109-122, 2005.
- [Cho04] Choi, Y.-S., Chung, K.-S. and Kim, L.-S. Adaptive Tessellation of PN Triangles Using Minimum-Artifact Edge Linking. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences E87-A*, No. 10, pp. 2821-2828, 2004.
- [Coh98] Cohen, J., Olano, M. and Manocha, D. Appearance-preserving simplification. In *Proc. of the 25th annual conf. on Computer graphics and interactive techniques*, ACM Press, pp. 115-122, 1998.
- [DeC07] DeCoro, C. and Tatarchuk, N. Real-time mesh simplification using the GPU. In *Proc. of the 2007 symp. on Interactive 3D graphics and games*, ACM Press, pp. 161-166, 2007.
- [Dia06] Diaz-Gutierrez, P., Bhushan, A., Gopi, M. and Pajarola, R. Single-strips for fast interactive rendering. *Vis. Comput.* 22, No. 6, pp. 372-386, 2006.
- [Eva96] Evans, F., Skiena, S. and Varshney, A. Optimizing triangle strips for fast rendering. In *Proc. of the 7th conf. on Visualization '96*, IEEE Computer Society Press, pp. 319-326, 1996.
- [Gu02] Gu, X., Gortler, S. J. and Hoppe, H. Geometry images. In *Proc. of the 29th annual conf. on Computer graphics and interactive techniques*, ACM Press, pp. 355-361, 2002.
- [Gut05] Guthe, M., Balázs, Á. and Klein, R. GPU-based trimming and tessellation of NURBS and T-Spline surfaces. *ACM Trans. Graph.* 24, No. 3, pp. 1016-1023, 2005.
- [Lin96] Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L. F., Faust, N. and Turner, G. A. Real-time, continuous level of detail rendering of height fields. In *Proc. of the conf. on Comp. graphics and interactive techniques*, ACM Press, pp. 109-118, 1996.
- [Los03] Losasso, F., Hoppe, H., Schaefer, S. and Warren, J. Smooth geometry images. In *Proc. of the 2003 Eurographics/ACM SIGGRAPH symp. on Geometry processing*, Eurographics Association, pp. 138-145, 2003.
- [Pul96] Pulli, K. and Segal, M. Fast rendering of subdivision surfaces. In *Proc. of the eurographics workshop on Rendering techniques '96*, Springer-Verlag, pp. 61-70, 1996.
- [Reu05] Reuter, P., Behr, J. and Alexa, M. An Improved Adjacency Data Structure for Fast Triangle Stripping. *Journal of Graphics Tools* 10, No. 2, pp. 41-50, 2005.
- [Shi05] Shiue, L.-J., Jones, I. and Peters, J. A realtime GPU subdivision kernel. *ACM Transactions on Graphics (TOG)* 24, No. 3, pp. 1010-1015, 2005.
- [Spi06] Spindler, M., Bubke, M., Germer, T. and Strothotte, T. Camera textures. In *Proc. of the 4th international conf. on Computer graphics and interactive techniques in Australasia and Southeast Asia*, ACM Press, pp. 295-302, 2006.
- [Tar06] Tariq, S. Next Generation Effects in DirectX10. In *Nvidia tutorials session at SIGGRAPH*, 2006.
- [Tat07] Tatarchuk, N. Real-Time Tessellation on GPU. In *Course 28: Advanced Real-Time Rendering in 3D Graphics and Games. ACM SIGGRAPH 2007*, 2007.
- [Vla01] Vlachos, A., Peters, J., Boyd, C. and Mitchell, J. L. Curved PN triangles. In *Proc. of the symp. on Interact. 3D graphics*, ACM Press, pp. 159-166, 2001.
- [Whi05] Whitted, T. and Kajiya, J. Fully procedural graphics. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS conf. on Graphics hardware*, ACM Press, pp. 81-90, 2005.
- [Yan05] Yang, Y., Chen, J. X. and Beheshti, M. Nonlinear Perspective Projections and Magic Lenses: 3D View Deformation. *IEEE Comput. Graph. Appl.* 25, No. 1, pp. 76-84, 2005.