Hardware Accelerated Multi-Resolution Geometry Synthesis

Martin Bokeloh* Michael Wand°

* WSI/GRIS, University of Tübingen ° Computer Graphics Laboratory, Stanford University

Abstract

In this paper, we propose a new technique for hardware accelerated multi-resolution geometry synthesis. The level of detail for a given viewpoint is created on-the-fly, allowing for an almost unlimited model resolution in rendering without excessive memory usage. The models consist of regularly sampled rectangular patches that are subdivided hierarchically by a programmable shader in order to create different levels of resolution. The approach is inherently parallel and lends itself to an implementation on vector processor-like parallel architectures. We demonstrate this property by an implementation on programmable graphics hardware. This implementation shows a substantial performance benefit over a CPU-based implementation by up to more than an order of magnitude. We apply the framework to rendering of smooth surfaces and to rendering of complexly structured fractal landscapes using a novel multi-channel fractal subdivision technique. Due to the hardware acceleration, it is possible to perform interactive editing and walkthroughs of such scenes in real-time.

Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture/Image Generation – Display Algorithms; I.3.6 [Computer Graphics]: Methodology and Techniques – Graphics data structures and data types.

Keywords: multi-resolution modeling, games and GPUs, graphics hardware, large data sets

1 INTRODUCTION

In the last 10 years, computer graphics has experienced a dramatic increase in performance of rendering hardware. Contemporary graphics coprocessors (GPUs) are capable of processing several hundred million primitives per second, allowing for highly complex geometry to be displayed in real-time. Given the capabilities for sophisticated rendering of complex content, additional attention has to be paid to the problem of modeling complex scenes and the coupling of the modeling and the rendering process.

In terms of effort for a human modeler, it is virtually impossible to create complex geometries by editing on a per-primitive basis. Consequently, *procedural modeling* techniques are frequently used to create detailed 3d models. Such techniques allow (generally speaking) the control of a more complex geometry by only a few parameters to a modeling algorithm. This property is usually called *data amplification* in computer graphics literature. Procedural techniques range from spline surfaces to complex fractal models, which provide a varying degree of data amplification. An important advantage of procedural modeling techniques is memory efficiency: By storing only the parameters for the procedural model instead of the generated set of geometric primitives, memory requirements can be drastically reduced. Rendering primitives (triangles, micro-polygons, ray sample points) are generated on-the-fly, during rendering. A further benefit is level of detail control: For many procedural rendering techniques, the number of primitives being generated for rendering can be easily adapted to the current requirements (such as the viewpoint), resulting in a significant reduction of rendering time.

Although being commonly used in offline rendering (see e.g. [Cook et al. 87]), procedural generation of geometry is only rarely used in interactive graphics. Most often, triangle meshes are precomputed and transferred to the graphics board for rendering. Only surface shading is commonly performed by procedural techniques (and in hardware), as this is directly supported by the architecture of current GPUs. In cases where a more compact procedural description of geometry is available, this causes avoidable storage and bandwidth problems. Ideally, the evaluation of the procedural model should be performed on-the-fly, at rendering time. For current PC hardware, this means that geometry synthesis should be performed by the GPU, avoiding the bandwidth and processing bottlenecks of the main CPU. For other architectures (such as the upcoming multi-core game console architectures), a similar processing model, enabling the usage of several computational hardware units in parallel, is also desirable.

In this paper, we propose a new approach for hardware accelerated geometry synthesis. It employs a restricted quadtree subdivision of rectangular, regularly sampled patches, corresponding to different levels-of-detail of an object. Higher resolution patches are created by subdividing lower resolution patches into four; new points are functions of fixed neighborhoods of the corresponding lower resolution points. Multiple attribute channels are employed to represent additional information to guide the subdivision process. The subdivision routine, which accounts for most of the computational demands of the algorithm, can be implemented using a single instruction stream on large amounts of data in parallel so that it can be executed very efficiently by a vector-processor style parallel architecture. We demonstrate the performance benefits of this approach by implementing the algorithm using the pixel shaders of current GPUs, resulting in a speedup of up to more than an order of magnitude in comparison to a CPU implementation. The proposed framework is very flexible, allowing for applications ranging from simple smooth surfaces to complex landscape models.

The proposed technique combines several well-known algorithmic building blocks. Our main contribution is a composite modeling architecture that can be implemented efficiently on parallel graphics hardware and is still flexible enough to create complexly structured models. Efficient execution on parallel hardware is achieved by the usage of a fixed subdivision kernel for all data points. However, a problem with this approach is the stationarity of the subdivision rule, leading to models where different parts have similar geometric characteristics. The main idea

* martinbokeloh@gmx.net

° mwand@stanford.edu

to overcome these limitations is the usage of multiple attributes per data point. The additional attribute channels store meta information (such as surface roughness or vegetation density) to control the subdivision process, which themselves are altered by higher level subdivision steps. This results in more flexibility and variability in the synthesized model. We apply this modeling approach to the synthesis of complexly structured fractal landscapes. Due to the hardware acceleration and the multi-resolution approach, modeling and interactive editing of such scenes can be performed in real-time while maintaining a high model and image quality.

2 RELATED WORK

Our system is based on several techniques from literature, such as deterministic and stochastic subdivision for geometry synthesis and restricted quadtree triangulations for level of detail control. In this section, we discuss the relation to literature in these areas as well as to recent GPU-based geometry synthesis techniques.

Modeling by subdivision: Many procedural modeling techniques can be expressed as subdivision algorithms. Spline surfaces can for example be rendered by a repeated application of the de Casteljau algorithm [Bartels et al. 1987]. Subdivision surfaces [Catmull and Clark 1978, Doo 1978] generalize modeling of smooth surfaces to meshes of general topology (see e.g. [Zorin et al. 2000] for a survey).

Stochastic subdivision / fractal modeling: Subdivision techniques can also be used to create irregular, non-smooth surfaces. Such surfaces can be characterized as random noise with a certain frequency spectrum (often proportional to $1/f^h$ for some fixed h) [Musgrave 1993]. A subdivision algorithm takes a regularly sampled noise signal, upsamples it to a higher sampling rate and adds additional high-frequency noise that has not been represented by the lower resolution version. This scheme has been first introduced by [Fournier et al. 1982] and extended by several authors: [Miller et al. 86] propose a smooth interpolation scheme to avoid discontinuity artifacts. A general analysis of stochastic subdivision of scalar data arrays is given by [Lewis 1986]. Noise properties are modeled by 2nd order statistics (mean, variance, autocorrelation). It is shown how different noise characteristics (such as different roughness or anisotropy, e.g. to create ocean waves) can be translated into subdivision rules of fixed neighborhoods. Our approach can handle subdivision rules that create high resolution points as a function of a fixed neighborhood of the original data (with performance depending on the neighborhood size). This demand is met by all aforementioned subdivision schemes.

A non subdivision-based technique is described by [Perlin 1985]: Noise functions of several input attributes are used to create complexly structured textures, allowing a pixel-parallel evaluation. We apply a similar idea to describe the subdivision function. Rendering of fractal landscapes with dynamic level of detail is also provided by commercial software packages such as MojoWorld [Pandromeda 2005] or Terragen [Planetside 2005]. These packages offer a high image quality; however, generating such images takes at least several minutes.

Multi-resolution modeling: The classic approach for level of detail control is the construction of a triangle hierarchy that allows a refinement or coarsening of the model by local triangle insertions and deletions (see e.g. [Lubke et al. 2003] for a survey). This hierarchy can allow general triangle meshes [Hoppe 1996] or restricted classes of meshes, such as subdivision connectivity meshes [Lindstrom et al. 1996]. Many techniques have been described that target especially at the case of terrain visualization (see e.g. [Duchaineau et al. 1997, Pajarola 1998, Röttger et al. 1998, Lindstrom and Pascucci 2001]), mostly being based on restricted triangle hierarchies. Recent level of detail techniques mostly operate batch oriented, employing hierarchies with several thousand triangles per hierarchy node to optimize the throughput to the GPU [Cignoni et al. 2003, Larsen and Christensen 2003, Balázs et al. 2004]. The technique of [Losasso and Hoppe 2004] is especially optimized for streaming data to the GPU. Arguing that geometry setup and transfer is typically more often a limiting factor than vertex processing by the GPU, their technique does not perform feature dependent mesh optimization but uploads regular grids of different mip-map levels to the GPU.

Our technique uses a subdivision connectivity hierarchy (restricted quadtree) of regularly sampled patches, similar to [Larsen and Christensen 2003]. The regular sampling is needed to facilitate the subdivision modeling process. The geometry is rendered batchwise, directly from graphics memory (where it has been created). Following the arguments of [Losasso and Hoppe 2004], we think that the benefits of the regular structure, which guarantees a good utilization of the rendering pipeline, outweight the losses due to reduced adaptivity of the locally uniform mesh. Currently, we use a single rectangular patch to parameterize and sample the data, which currently excludes general base meshes as topology (which is subject of future work).

In addition to mesh simplification-based level of detail techniques, there are also point-based level of detail techniques that are favorable for objects of complex mesh topology. Applications to landscape rendering have been demonstrated for example by [Stamminger and Dretakis 2001] or [Wand et al. 2001].

Parallel / GPU-based geometry synthesis: The desire for hardware accelerated geometry synthesis and rendering is not new: For example [Max 1981] describes an implementation of a raytracer for procedural terrain models implemented on a Cray-1 vector computer. [Perlin and Hoffert 1989] employ a massively parallel raytracer for efficient rendering of procedurally defined noise volumes, coined “Hypertextures”. Recently, several papers have been published that deal with geometry synthesis on contemporary GPUs. [Dachsbacher and Stamminger 2004] propose a multi-resolution rendering technique based on image warping: The geometry of a terrain is encoded in a regularly sampled patch. This patch is then upsampled non-uniformly to a higher resolution, spending more space in “important” regions (according to camera distance, orientation, view frustum). Additionally, fractal noise is added to the geometry to increase the level of detail. This technique is conceptually elegant but aims at a different application than our technique. For use as general modeling primitive, the application of fractal noise in distorted space is probably difficult to control in contrast to regular hierarchical subdivision. [Shiue et al. 2003] propose an extension to current GPU shader APIs to support general mutation and subdivision operations. [Guthe et al. 2005] describe an approach for rendering trimmed NURBs and T-spline surfaces on graphics hardware using a bit-counting scheme for efficient, hardware-based evaluation of trimming curves. They report a drastic performance boost due to the GPU implementation. [Bolz and Schröder 2003] describe a GPU-based algorithm to evaluate subdivision surfaces using precomputed tables reflecting the mesh topology. A refined technique is presented by [Shiue et al. 2005] using spiral enumeration of vertices. In contrast to our proposal, these technique support general topologies of base meshes but do not provide an intra-patch multi-resolution scheme, thus not being applicable to rendering of extended objects such as landscapes. The same argument also applies to the method of [Boubekeur and Schlick 2005], who propose mesh refinement in the vertex shader. Recently, a fully procedural rendering hardware has been proposed by [Whitted and Kajiya 2005] that executes procedures in hardware to create point rendering primitives.

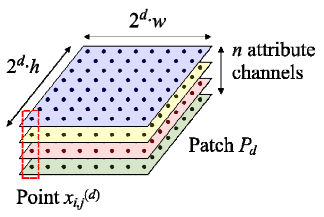


Figure 1: Layout of a single patch. The border points (2*k* to each side) are not shown.

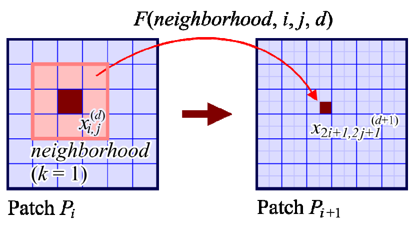


Figure 2: Patches are subdivided by applying a subdivision function *F* to a *k*-neighborhood

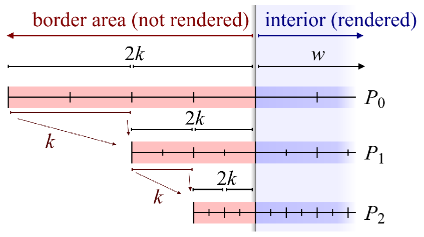


Figure 3: Handling of borders: *k* neighbors are needed to create a new point. Thus, a border area of 2*k* points is needed.

3 Multi-Resolution Geometry Synthesis

In this section, we describe our proposal for hardware accelerated multi-resolution geometry synthesis. First, we define geometry through subdivision, then we describe the hierarchical multi-resolution scheme and the GPU-based prototype implementation.

3.1 Modeling by Subdivision

In order to facilitate a hardware implementation, we describe geometry as rectangular, regularly sampled patches. Each patch describes a surface with disc topology; for more complex topologies, several patches have to be combined. Each sample point $x_{i,j}$ in a patch is a *n*-dimensional vector of *attributes* (Figure 1). The initial patch is given by a $w \times h$ array that is enlarged by a border of 2*k* sample points to each side:

$$\text{Initial patch } P_0\text{:}$$
$$x_{i,j}^{(0)} \in \mathbb{R}^n, \quad -2k < i < w + 2k, \quad -2k < j < h + 2k$$

The values for the initial patch are specified by the human modeler. *k* is the support of the subdivision function (see below). The border is necessary to define a consistent subdivision function (border issues are discussed in the following subsection). Subsequently, higher resolution versions of the initial patch are created with the number of sample points doubling at each iteration:

$$\text{Higher resolution patch } P_d,\ d > 0\text{:}$$
$$x_{i,j}^{(d)} \in \mathbb{R}^n, \quad -2k < i < w \cdot 2^d + 2k, \quad -2k < j < h \cdot 2^d + 2k$$

Sometimes, it is useful to identify points in a patch by a unique *parameter coordinate* $(i/2^d, j/2^d) \in [-2k, w + 2k] \times [-2k, h + 2k]$ rather than by the indices (i, j). The higher resolution patches are created procedurally by applying a *subdivision function F* to data from the previous level (Figure 2). The subdivision function obtains a fixed $(2k + 1)^2$ neighborhood of values from the previous level as well as the current point index and subdivision level as input and creates a new point:

$$x_{i,j}^{(d)} = F\left(\begin{bmatrix} x_{i/2-k,j/2-k}^{(d-1)} & \cdots & x_{i/2+k,j/2-k}^{(d-1)} \\ \vdots & \ddots & \vdots \\ x_{i/2-k,j/2+k}^{(d-1)} & \cdots & x_{i/2+k,j/2+k}^{(d-1)} \end{bmatrix}, d, i, j\right)$$

The function *F* is specified by the human modeler as a procedure. There are no general restrictions to *F* other than being computed in finite time. However, for an efficient implementation on vector processors, the *instruction stream* for computing *F* must not depend on the values $x_{i,j}^{(d-1)}$, *i* or *j* but only on *d*. This does *not* mean that the computed value is independent of these quantities, only the sequence of instructions doing the computation is restricted. For more general architectures (such as DirectX 9 pixel shader 3.0 hardware [Ati 2005, nVidia 2005]) this restriction can be relaxed, requiring only a spatially coherent rather than identical instruction stream.

The attribute vectors $x_{i,j}^{(d)}$ do not need to represent geometric quantities (such as a position in three space) but may describe arbitrary attributes. To create the actual geometry, a mapping function $R: \mathbb{R}^n \rightarrow \mathbb{R}^m$, $m \geq 3$ is applied. This function computes a geometric position in three space for each attribute vector, probably along with other rendering parameters such as normals, colors or texture coordinates. Rendering buffers will be cached in memory; therefore, employing this extra mapping step avoids overhead during rendering as the mapping is only performed once.

Handling Borders

Please note that the subdivision procedure outlined above leads to *shrinking* patches: With each subdivision step, a border region of *k* sample points to each side is removed. However, their size in the original parameter domain shrinks by $1/2^d$. This means, for *d* subdivision steps, a region of at most

$$\sum_{i=0}^{d} k/2^i \leq 2k$$

points, measured in parameter coordinates (i.e. sample spacing of the original patch), is removed. This is the reason for choosing a border size of $\pm 2k$ for the initial patch. It is guaranteed that the "lost" area after an arbitrary number of subdivisions does not exceed this boundary area (see Figure 3). Consequently, only geometry at coordinates within $[0, w] \times [0, h]$ (parameter coordinates) is rendered. The border region is never shown, it only affects the shape of the inner region indirectly, similar to boundary points of uniform B-splines [Bartels et al. 1987].

This effect does only occur at boundaries. If we consider more general topologies, where several patches are stitched together along their boundaries to form a quad mesh of arbitrary topology, we only need to provide boundary values at topological borders. In other areas, the boundary values are taken from the adjacent patch. An special case is a patch with cyclic boundary conditions, referring to a topological torus. In this case, no boundary values are needed. In general, the same is true for arbitrary manifold meshes without (topological) boundaries. Currently, our implementation supports cyclic and border boundary conditions for a single patch only, more general topologies are still subject to future work.

3.2 Multi-Resolution Hierarchy of Patches

Employing the subdivision process outlined above, the amount of data to be processed is quadrupled at each subdivision step. If the viewer is very close to the surface, demanding for a high resolution for adequate rendering, the processing costs can easily become prohibitive. This problem can be alleviated by a multi-resolution approach: Instead of increasing the resolution for the whole patch at once, we divide each patch in four equally sized subpatches and apply the refinement step separately to each subpatch if it is necessary. This leads to a quadtree subdivision scheme (Figure 4). Each node in the quadtree corresponds to a

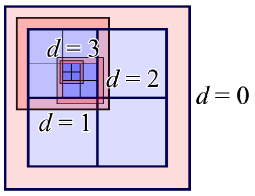


Figure 4: Subdividing patches. A border of $2k$ points is attached to each patch to allow the computation of near-border values.

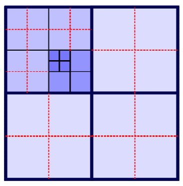


Figure 5: A restricted quadtree (8 neighborhood) is used to make neighboring values available. Red: additional hierarchy levels, enforcing at most one level difference.

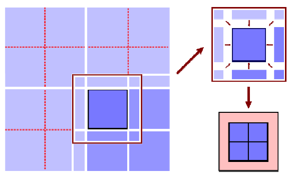


Figure 6: Patch subdivision in hardware – first neighboring area is assembled to an enlarged patch, then the subdivision shader is employed.

$w \times h$ array of sample points. This approach causes some subtle issues that have to be addressed by the subdivision algorithm:

Handling Inner borders

The first problem is handling inner borders. In order to refine a sample point with index (i, j) in a patch, all its neighbors with indices $[i-k \ldots i+k] \times [j-k \ldots j+k]$ have to be known. This means that we must have computed the 8 direct neighbors of a patch to be able to compute the next level of refinement for this patch. In other words, the hierarchy must be a (well-known) restricted quadtree [de Berg et al. 1997]. Adjacent levels of resolution must not deviate by more than one level of resolution. In our case, adjacency is defined by the 8 neighborhood of a patch (Figure 5). Then, we can access the values of neighboring patches to obtain values at the borders.

This constraint can easily be enforced by on-demand computation: Whenever a patch has to be refined, all eight direct neighbors are retrieved by a sibling search algorithm. If the demanded patch (or one of its parents) does not yet exist, we call the creation procedure recursively. After some patch subdivisions, all necessary neighboring nodes have been build and the patch of the next higher resolution level can be finally created. This “balancing” step adds additional overhead to the multi-resolution scheme. However, this overhead is only O(1), which is easy to see by assigning “overhead” subdivisions to the neighbor that demanded for them [de Berg et al. 97]. In practice, the overhead factor is rather small: Overhead nodes only occur at the border of the view frustum (which is typically only a one-dimensional border in the parameter domain, affecting $O(n^{1/2})$ of the n nodes). The varying resolution due to the distance to the camera is a smooth function which usually already demands small spacings in resolution by itself.

A second problem, also caused by the variation of resolution, is the triangulation of the surface: We would like to display a continuous, triangulated surface when rendering the patches. As we are already forced to build a restricted quadtree hierarchy for modeling, the solution is straight-forward: Considering one node, the neighboring nodes can only differ by at most one level of resolution. Correspondingly, only a small number of triangulations can occur which can easily be precomputed (similar to [Larsen and Christensen 2003]) and instantiated during rendering.

Multi-Resolution Rendering

During rendering, we traverse the quadtree top-down and stop the descent if a node meets the precision requirements (if a node does not exist, it is created, as outlined above). Different metrics can be employed at this step. We currently use the following, rather canonical rule: The decision whether to render a node is solely based the bounding box of its geometry (which has to be determined during or after geometry synthesis, see below). Nodes with bounding boxes completely outside the view frustum are never rendered. Nodes inside the view frustum are rendered (and the descent is stopped) if their projected, on-screen resolution exceeds a user defined threshold. The on-screen resolution is estimated by dividing the side length of the largest side of the bounding box by the number of points along one edge of the patch (we employ square patches only, with $w = h$). This value is then projected onto the screen by dividing by the minimum z-value of the bounding box and scaling by a constant according to resolution and viewing angle. A near-clipping plane is included in the view frustum to avoid demanding infinite resolution (additionally, a fixed upper limit can also be specified, if desired).

When all patches have been selected from the hierarchy, each patch is rendered as a triangle mesh. The mesh is chosen from a list of precomputed vertex indexing buffers by considering the resolution of the neighboring patches.

3.3 Hardware Implementation

The algorithm involves two major tasks: Management of the restricted hierarchy and processing of the points. Hierarchy management involves the traversal of irregular data structures which is difficult to accelerate by special purpose hardware. Thus, this task is done by the CPU. If the resulting CPU load is too high, we have the option to increase the number of sample points per patch, trading-off the adaptivity of the multi-resolution representation for less hierarchy management workload for the CPU. Larger patch sizes lead to a less accurate view frustum culling and some oversampling at parts of the patch farther away from the viewer. However, for typical patch sizes of about 32^2 - 64^2 triangles, such adverse effects are small while already placing the main computational burden to the hardware accelerated patch processing.

Hardware Subdivision

The first step for creating higher resolution patches is the assembly of the $2k$-neighborhood: The original patch is copied into a buffer enlarged by $2k$ sample values at each side. Then, the 8 neighboring patches are fetched from the hierarchy and the values at the border to the current patch are copied to the border regions of the larger buffer (Figure 6) using a BitBlit operation on the graphics hardware.

The second step is the computation of the high resolution data. First, four $w \times h$ sized destination buffers for the 4 children are allocated. Then, the subdivision function F has to be evaluated. This step is usually the most expensive of the algorithm and the main goal of our architecture was to allow for an efficient hardware implementation at this point. This evaluation can be implemented very efficiently on a vector processing architecture (SIMD): The patch consists of several sample points that can all be processed in parallel, using the same instruction stream. In our implementation, we use typically 32^2 patches corresponding to 1024 potentially parallelizable function evaluations.

The third step is the creation of rendering data by applying the mapping function R. For this step, a new buffer (probably with a different number of attributes per point) of the same size as the source patch (but omitting the border region) has to be allocated first. Then, R is applied to each point of a patch independently and the result is written into the output buffer. This process can be executed on the same hardware as the subdivision process, the only difference is that no upsampling takes place.

The last step is the rendering step: A precomputed index buffer of triangles is chosen and the data in the rendering buffer provides the vertices of the mesh. Each vertex provides a position in 3 space and probably further shading attributes such as normals and color. This data can be processed directly and very efficiently by a contemporary programmable GPU.

The created patches, both subdivided and rendering data, are not deleted after rendering but kept in memory for future use. A LRU scheme is applied to track the reusage of these buffers. If memory is filled-up, patches that have not been used for the longest time are deleted first to free memory.

GPU-based Implementation

We have implemented a prototype of our algorithm on a programmable GPU, using OpenGL and CG as API (see [ATI 2005, nVidia 2005] for details on the programming capabilities mentioned below). We map the computationally intense steps of subdivision (F) and mapping (R) to the pixel shader of the GPU. These units provide several parallel ALUs that can be used in a SIMD programming model: Each pixel is being computed independently, using the same instruction stream. Additionally, the number of output pixels has to be specified in advance while the amount of input data may vary, according to the shader program. These conditions are met by our geometry synthesis technique.

Mapping of the algorithm to a programmable GPU is straightforward: Patches are represented as textures (if being used as source) or render targets (if being used as destination). In order to avoid switching of render targets, only one render target is created and used as temporary buffer. The data is copied to a texture associated with a patch directly after each computation via onboard memory transfer.

The attribute channels of the patches are implemented using multiple render targets: On the latest hardware, each pixel shader can read from up to 16 textures and output to up to 4 render targets, both providing up to 4 32-bit floating point channels each. In this way, up to 16 floating point attribute channels can be handled in one rendering pass. For more attribute channels, multiple rendering passes are necessary. The example scenes in this paper use 12 (landscapes) and 8 (subdivision surfaces) 32 bit floating point channels, respectively.

In our implementation, initial data for patch P_0 can be specified by importing data from data sources such as landscape elevation data or by interactive painting on the 3d-geometry. We allow arbitrary amounts of initial data, main memory permitting. If the initial data is larger than a patch (i.e. typically 32^2 plus border), a multi-resolution pyramid is build in main memory by subsampling (currently nearest-neighbor subsampling) the original data in a quadtree of patches. This initial pyramid is handled in software and patches are transferred to the graphics board on demand. If the demanded rendering resolution exceeds that of the initial data, the hardware accelerated geometry synthesis is invoked.

The subdivision function F and the mapping function R of the geometry synthesis are represented as pixel shader programs. The latest shader standard (DirectX 9, shader model 3.0, [Ati 2005, nVidia 2005]) even allows data dependent branching in the pixel shader, extending the strict SIMD model. The achieved performance depends on the coherency of the instruction streams for neighboring data. In our example scenes, we do not use data dependent branching but only conditional writes that do not alter the instruction stream, which has turned out to be sufficient for our models.

Lastly, a further vertex/pixel shader pair is used for final rendering of the resulting triangle meshes. The render buffers are created by copying the content of the render target directly to a vertex buffer, which is supported by current OpenGL vendor extensions. Copying to a vertex buffer is very efficient on current hardware. An alternative would be the usage of texture fetches in the rendering vertex shader. This method has the advantage of easily allowing for interpolation between adjacent subdivision levels to avoid popping artifacts, which is not included in our current implementation based on copying buffers.

Our GPU-based implementation processes all geometry data on the GPU only, with one exception: In order to control the multi-resolution rendering, the bounding boxes of the synthesized geometry have to be known to the CPU. Thus, the position channel of the rendering data has to examined and the minimum and maximum x, y and z coordinates must be determined. This is done in two steps: First, we reduce the amount of data to be transferred by scaling down the patches [Buck and Purcell 2004]: We use a pixel shader that computes the minimum and maximum values of 4×4 neighborhoods and outputs them to an eightfold reduced patch of data. This process can be repeated iteratively. In our experiments, one such reduction pass was sufficient; a second pass did not lead to a further reduction of the overall computation time. After reduction, the resulting data is read back to main memory and the bounding box is computed by the host CPU, now requiring only little transfer bandwidth. Up to 16 read back operations are performed in one batch from the same reduction buffer to reduce synchronization overhead.

4 Modeling

We have implemented two different modeling techniques to examine the practical applicability of our proposal:

Smooth surfaces: To model smooth surfaces, we first need a parameterization of the surface as a planar patch. Then, well-known techniques such as subdivision surfaces or spline subdivision can be employed. As an example, we have implemented the bicubic B-spline subdivision model of the Stanford bunny described in [Lossaso et al. 2003]: The authors create a geometry image of the bunny geometry and compute vertices for a least square B-spline subdivision surface approximation. We have used the data from this paper (which is available on the web) and reimplemented the subdivision process. In addition to the original paper, our implementation provides adaptive multi-resolution modeling and rendering, allowing for close-ups of objects without loss of detail or serious penalties to the rendering performance.

Multi-channel fractals: The multi-resolution approach of our modeling technique allows handling of large, extended models such as an entire landscape. To define such models, we employ a fractal modeling technique which we call *multi-channel fractals*. The object is described by a set of attribute channels corresponding to different surface properties. In our example, we use a height channel describing the landscape as a height field. Additionally, we have channels for surface roughness, vegetation density for different layers of vegetation (shown as different colors during rendering), and a snow layer. Each channel contains fractal $1/f^h$ noise (with non-stationary h). To create believable landscapes, interdependences between these channels are introduced in the subdivision step:

The height field is created by first interpolating the local neighborhood using a smoothing filter. Then, random noise is added with an amplitude of 2^{-dh} with h being a smoothness parameter which is stored in a separate channel. The h-channel is a fractal itself: It is also created by smooth filtering of neighboring h values and random additions. However, we prefer larger values of h (leading to smoother terrains due to smaller noise increments) if the value in the height channel is small (i.e. we are in the area of a vally). Conversely, h is decreased (leading to more roughness) if the slope of the height field at the current level of resolution is large, leading to more roughness at steep mountainsides. Both is implemented by blending between the *h* channel and a height/slope depended *h* according to the subdivision level. At low levels, a strict correlation of roughness to height and slope is enforced while more randomness is allowed at smaller scales.

Similarly, vegetation textures and a snow density are created by employing fractal channels, which are influenced but not determined entirely by height and slope. For snow, we expect a smooth surface appearance at thick layers of snow. The thicker the layer of snow, the more high frequency details are attenuated. Consequently, the values in the roughness channel *h* are strongly enlarged in regions with a large value in the snow channel. The result conveys an quite realistic look of snow-covered areas in a rough mountain range. This interplay of fractal randomness and parameter interdependence yields landscapes with irregular attributes but believable mutual influence and can probably be employed to approximate a variety of other natural phenomena, too. Of course there are limitations. For example, we cannot directly simulate global physically-based effects such as erosion [Musgrave et al. 89].

5 RESULTS

The results reported in this section have been measured on a system equipped with an nVidia GeForce 6800GT AGP graphics board (256MB video ram) and a 2.6GHz Pentium 4 CPU. The software has been implemented in C++ and all shaders have been implemented in CG [nVidia 2005]. The shader code is canonical C code, no assembly code or hardware specific optimizations have been employed. Figure 8 shows renderings of example models created with the techniques described in Section 4. The images are annotated with the rendering time (from cache), the rebuild time (rendering with emptied caches) and a typical rendering time for a walkthrough (as shown in the accompanying video).

Smooth surface: The bunny model in Figure 8(a) has been constructed using the technique of [Lasasso et al. 2003], as described in Section 4. The subdivision shader performs smoothing and normal vector computation, rendering is done by a simple environment mapping shader (to show the surface smoothness). For a typical viewpoint, we obtain 33 frames per second and only moderate reduction for a moving observer (see video).

Fractal landscapes: The landscape models in Figure 8(b) - (e) have been created using the multi-channel fractal technique. For rendering, antialiased shadow maps (12 samples) and an approximate atmospheric scattering model have been employed [Hoffman and Preetham 2002]. The vegetation texture (different shades of green) and the snow have been modeled as fractal attribute channels (as described in Section 4); the grass has been additionally modulated by a periodic 2d texture.

A basic landscape scene is shown in Figure 8(b). The shown view consists of 604 patches of 32^2 vertices, accounting for about 1.2 million triangles. At the shown quality level, it can be rendered at about 6 frames per second. The throughput of the rendering stage is currently limited by the complexity of the rendering shaders which have to compute the quite involved lighting model. Additionally, some of the mapping steps (such as coloring of vegetation layers) are still computed during rendering to facilitate interactive landscape design. A rebuild of all geometry from scratch takes 2.4 seconds; however, due to temporal coherence, the average frame rate during a walkthrough does not drop significantly (see video). Figure 8(d) shows a similar scene, but with more roughness and more snow. Please note how the snow channel automatically damps out high frequency noise, leading to the impression of rough terrain covered by a layer of snow of different thickness. Figures (e) and (f) show a variant of the model from Figure 8(b). Here, a second fractal layer has been introduced to model water. The second layer is computed for each patch after the landscape layer so that its attributes can be accessed for defining the second layer. It is rendered with a water shader (using an additional rendering pass to create a mirrored and a refracted image of the landscape). The foam at the coastline is created by a fractal channel similar to the vegetation channel. The overall shape depends on water depth but also shows random variations. Due to the double layer modeling and the multiple rendering passes, the framerates are lower. Figure (e) and (f) have been created with different level of detail settings, varying the projected vertex spacing parameter as described in Section 3.2. A last example is shown in Figure 8(c). For this scene, we have used height field data of the grand canyon [US Geological Survey 2005] and added different fractal channels. The original data is 400^2, a 32^2 patch sized multi-resolution pyramid of the original data is created by the CPU, geometry synthesis is applied for deeper levels of subdivision (see the video for an interactive walkthrough). Figure 7 shows the variation of the rendering time and the number of overall and rebuild patches per frame during the walkthrough of this scene. Due to caching, only a few patches have to be rebuild for each frame so that interactive walkthroughs are possible.

Evaluation: We have measured how much time is consumed by the different parts of the algorithm: Comparing the costs for geometry synthesis and rendering, we observe a factor of about 6-14. It is interesting to further split up the synthesis costs into actual hardware processing costs and time needed do the bounding box calculation (which involves reading back data from GPU memory). Due to the min/max reduction step (aggregation of 4×4 neighborhoods in a pixel shader), only a moderate overhead is observed: 10% of the rebuild time (landscape scene Figure 8(b)) and 25% (bunny scene), respectively, are spent for bounding box calculation. Without prior reduction, the overhead is significantly larger (41% and 57% respectively). The overhead is larger for the bunny scene because the subdivision shader is less complex. During animations, the average percentage of rendering time spent for bounding box calculations is about 1% for all scenes (due to caching) so that this overhead is not really an issue in practice.

A last, important point is to examine the benefit of a hardware implementation. We have compared the execution speed of the GPU implementation with a CPU implementation. As all shaders have been written in CG, we were able to compile almost the original code with a C++ compiler (Intel C++ 7.1, all optimizations enabled). Only a few CG specific commands and data types had to be translated into macros and classes with inline functions. Textures have been modeled as conventional, two dimensional C arrays. This approach (plain C++ code) reflects the typical programming approach in practice. However, it is still biased a bit in favor of the GPU, as vector data types are not intrinsic in standard C++ (although the employed compiler automatically tries to employ SSE SIMD instructions) and we do not use an optimized texture memory layout. Hence, the results should be considered with some care. Using this setup we have measured the computation time of the subdivision shader on both the CPU and GPU of the test system. We have obtained a computation time of 0.5 ms per patch for the GPU and 7.25 ms for the Pentium 4 2.6 GHz (factor 14.5) for the subdivision shader of the landscape model of Figure 8(b). For the bunny model subdivision shader, the result is 0.4 ms for the GPU and 1.3 ms for the CPU (factor 3.25)[1]. The

[1] We have also repeated the CPU benchmark on a Pentium-4 3.4 GHz, the fastest machine we had access to. This yielded performance factors of 11.8 and 2.5, respectively, to the GeForce 6800 GT GPU. Unfortunately, we did not have a system available with an ATI Radeon X1800 or nNvidia GeForce 7800 GTX graphics board for comparison with a high end GPU.

speedup for the bunny scene is appreciable but significantly smaller than for the landscape scene. Again, this is due to the much shorter shader which puts more emphasize on additional CPU-GPU communication overhead. For the landscape scene, the speedup is more than an order of magnitude. Hence, it is probably save to assume that the hardware-based implementation will still provide a substantial performance benefit for synthesizing complex geometry even if more aggressive low-level CPU optimizations are applied.

6 CONCLUSIONS AND FUTURE WORK

We have proposed a new hardware accelerated modeling and rendering technique that can be implemented on data parallel architectures such as current GPUs. The algorithm employs a hierarchy of regularly sampled patches to facilitate an efficient implementation on SIMD processing arrays. This structure maps well to pixel shaders of current GPUs, allowing for executing modeling and rendering almost entirely on the GPU, yielding a substantial performance improvement.

There are several directions for future work. First, some technical implementation issues (such as blending between resolution levels to avoid popping) could be improved. More importantly, the implementation should be generalized to support general base meshes. Currently, each patch is treated separately (this can be seen by a small hole in the bunny surface; the triangulation scheme does not connect the outer borders of the geometry image to a closed surface). This extension is mostly straightforward. The main issue is handling of neighborhoods at extraordinary vertices (valance $\neq 4$). Here, the technique of [Bolz and Schröder 2003] could be a starting point to be generalized for more general, stochastic subdivision techniques. Lastly, the subdivision topology could be made more flexible: Due to the limitations of current GPUs, we can only handle regularly sampled, rectangular patches. It would be interesting to examine subdivision rules that allow a change of topology during subdivision. In combination with a point-based rendering approach, more general shapes could be created. This would involve a generalized concept of neighborhoods and a subdivision unit with a variable number of output data points.

Acknowledgements

This work has been supported by the state of Baden Württemberg (Germany) and the Max Planck Center for Visual Computing and Communication. The authors wish to thank Leonidas Guibas, Martin Frisch, Timo Schairer, Andreas Schilling, and Wolfgang Straßer and the anonymous reviewers for their valuable comments. We especially wish to thank Alexander Berner, Arno Fleck, Mark Hoffmann, Philipp Jenke, Benjamin Maier for help with the implementation and Robert Kuchar for providing the skylight maps. The environment map for the bunny scene has been taken from P. Debevec's web site (www.debevec.org).

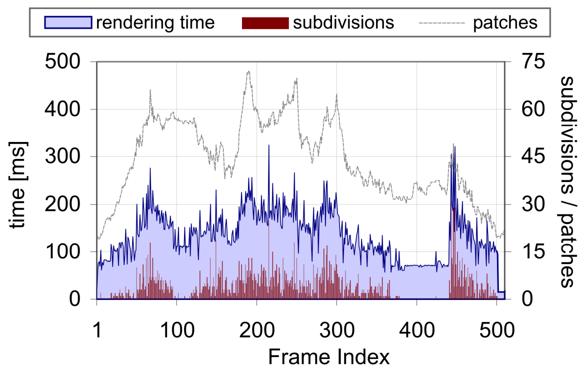


Figure 7: Overall rendering time, number of rebuild patches and number of overall patches for the frames of the Grand Canyon flyover (see accompanying video)

References

ATI, 2005. *ATI developer relations*. http://www.ati.com

BALÁZS, Á., GUTHE, M., AND KLEIN, R., 2004. Fat borders: Gap filling for efficient view-dependent lod rendering. In: *Computers & Graphics*, 28(1), 79–86.

BARTELS, R. H., BEATTY, J. C., and BARSKY, B. A. 1987. *An Introduction to Splines for use in Computer Graphics and Geometric Modeling*, Morgan Kaufmann Publishers.

BOLZ, J. and SCHRÖDER, P., 2003: *Evaluation of Subdivision Surfaces on Programmable Graphics Hardware*. http://www.multires. caltech.edu/pubs/GPUSubD.pdf

BOUBEKEUR, T., and SCHLICK, C., 2005: Generic Mesh Refinement on GPU. In: *Graphics Hardware 2005*.

BUCK, I., PURCELL, T., 2004: A Toolkit for Computation on GPUs. In: *GPUGems*, Addison-Wesley.

CATMULL, E., and CLARK, J. 1978. Recursively generated B-spline surfaces on arbitrary topological meshes. In: *Computer Aided Design*, 10(6), 350–355.

CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., PONCHIO, F., and SCOPIGNO, R. 2003: Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM). In: *Visualization 2003 Proceedings*.

COOK, R.L., CARPENTER, L., and CATMULL, E., 1987. The Reyes image rendering architecture. In: *Comptuer Graphics*, 21(3), 95–102.

DACHSBACHER, C., and STAMMINGER, M., 2004: Rendering Procedural Terrain by Geometry Image Warping. In: *Proc. of Eurographics Symposium on Rendering 2004*.

DE BERG, M., VAN KREVELD, M., OVERMARS, M., and SCHWARZKOPF, O., 1997: Computational Geometry – Algorithms and Applications, Springer Verlag.

DOO, D. 1978. A subdivision algorithm for smoothing down irregularly shaped polyhedrons. In: *Proc. on Interactive Techniques in Comuter Aided Design*, 157–165.

DUCHAINEAU, M., WOLINSKY, M., SIGETI, D. E., MILLER, M. C., ALDRICH, C. and MINEEV-WEINSTEIN, M. B. 1997: ROAMing Terrain: Real-time Optimally Adapting Meshes. Visualization 97 Proceedings, 81–88.

FOURNIER, A., FUSSEL, D., and CARPENTER, L., 1982. Computer Rendering of Stochastic Models. In: *Communications of the ACM* (25)6, 371–384.

GUTHE, M. BALÁZS, Á, and KLEIN, R., 2005: GPU-based trimming and tessellation of NURBS and T-Spline surfaces. In: *ACM Transactions on Graphics, 24(3).*

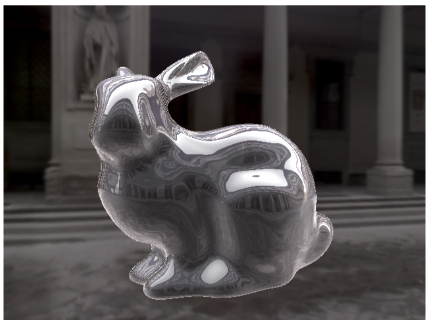
HOFFMAN N., and PREETHAM, A.J. 2002: Rendering Outdoor Light Scattering in Real Time. *http://www.ati.com/developer/techpapers.html*

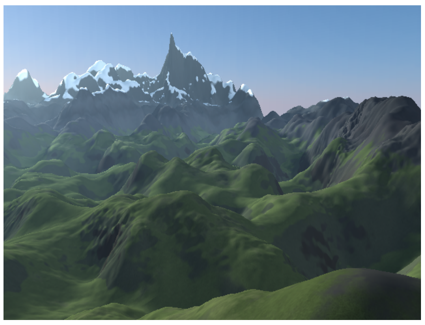
HOPPE, H.: Progressive meshes. In: *SIGGRAPH 96 Proceedings, Annual Conference Series*, 99–108.

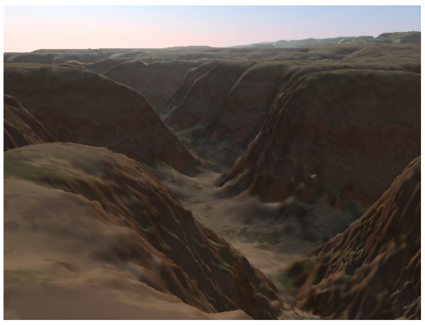
LARSEN, B.D., and CHRISTENSEN, N.J., 2003: Real-time Terrain Rendering using Smooth Hardware Optimized Level of Detail. In: *Journal of WSCG*, Vol.11, No.1.

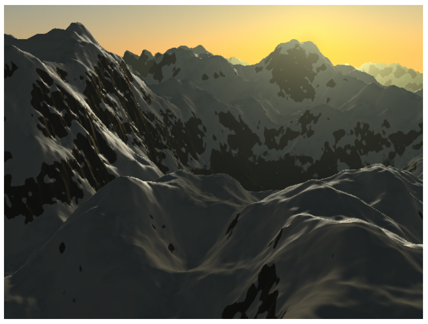
LOSASSO, F., and HOPPE, H., 2004: Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids. In: *ACM Transactions on Graphics, 23(3).*

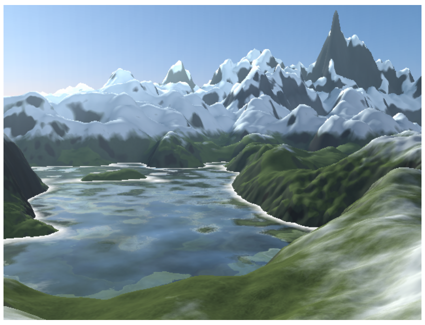
LOSASSO, F., HOPPE, H. , SCHAEFER, S., and WARREN., J., 2003: Smooth geometry images. In: *Eurographics Symposium on Geometry Processing 2003*, 138 – 145. Data taken from *http://research.microsoft.com/~hoppe*.


(a) Stanford Bunny – subdivision surface rendering (c.f. [Losasso et al. 2003]), 1089 control points, 184 patches (188416 vertices), rendering time 30 ms, rebuild from scratch 251ms


(b) Landscape scene – 604 patches, rendering time 169 ms, rebuild from scratch 2438 ms, walkthrough (see video) 152 ms per frame (av.)


(c) Grand Canyon – initial 400^2 height field from [US Geological Survey 2005], 433 patches, rendering time: 130 ms, rebuild from scratch 955 ms, walkthrough (see video) 136 ms per frame (av.)


(d) Mountain range at sunset – rendering time 228 ms, rebuild from scratch 2487 ms, 985 Patches, walkthrough (as shown in the video) 231 ms per frame (av.).


(e) Mountain Lake – a variant of landscape (b), medium resolution (2.2 pixel per triangle, 2×586 patches), rendering time: 196 ms. Rebuild from scratch: 1079 ms.

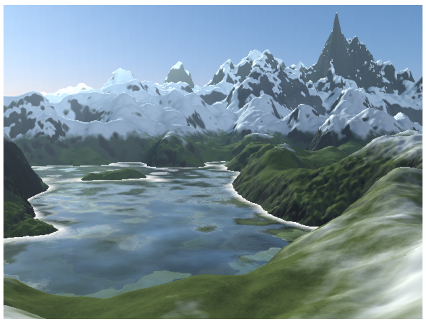

(f) Mountain Lake – a variant of landscape (b), high resolution (1 pixel per triangle, 2×1039 patches), rendering time: 370 ms. Rebuild from scratch: 3241 ms.

Figure 8: Application examples. In all examples, a multi-resolution patch contains 32^2 vertices (and k = 3 vertices border).

LEWIS, J.P., 1987. Generalized Stochastic Subdivision. In: *ACM Transactions on Graphics*, (6)3.

LINDSTROM, P., KOLLER, D., RIBARSKY, W., HODGES, L.F., FAUST, N., and TURNER, G.A., 1996. Real-time, continuous level of detail rendering of height fields. In: *SIGGRAPH 96 Proceedings, Annual Conference Series*, 109–118.

LINDSTROM, P., and PASCUCCI, V., 2001. Visualization of Large Terrains Made Easy. In: Visualization 2001 Proceedings.

LUEBKE, D., REDDY, M., COHEN, J.D., VARSHNEY, A., WATSON, B., and HUEBNER, R.: *Level of Detail for 3D Graphics*, Morgan Kaufmann Publishers, 2003.

MAX, N.L. 1981. Vectorized Procedural Models for Natural Terrain: Waves and Islands in the Sunset. In: *Computer Graphics*, (15)3.

Miller, G.S.P., 1986. The Definition and Rendering of Terrain Maps. In: *Computer Graphics*, (20)4.

MUSGRAVE, K.F., KOLB, C.E., and MACE, R.S., 1989. The synthesis and rendering of erroded fractal terrains. In: *Computer Graphics*, (23)3.

MUSGRAVE, K.F., 1993. *Methods for Realistic Landscape Imaging*. PhD thesis, Yale University.

NVIDIA, 2005. *nVidia developer relations*. http://www.nvidia.com

PANDROMEDA 2005: *MojoWorld software package. http://www.pandromeda.com*

PERLIN, K., and HOFFERT, E.M., 1989. Hypertexture. In: *Comptuer graphics*, 23(3).

PERLIN, K., 1985. An Image Synthesizer. In: *Comptuer Graphics*, 19(3).

PLANETSIDE SOFTWARE 2005: *Terragen software package. http://www.planetside.co.uk*

PAJAROLA, R., 1998. Large Scale Terrain Visualization Using The Restricted Quadtree Triangulation. In: *Visualization 98 Proceedings*.

RÖTTGER, S., HEIDRICH, W., SLUSSALLEK, P., and SEIDEL, H.P. 1998: Real-Time Generation of Continuous Levels of Detail for Height Fields. In: *Proceedings of the 6th International Conference in Central Europe on Computer Graphics and Visualization*, 315–322.

SHIUE, L.J., GOEL, V., and PETERS, J. 2003. Mesh Mutation in Programmable Graphics Hardware. In: *Graphics Hardware 2003*.

SHIUE, L.J., JONES, I., and PETERS, J.: A Realtime GPU Subdivision Kernel. In: *ACM Transactions on Graphics, 24(3)*.

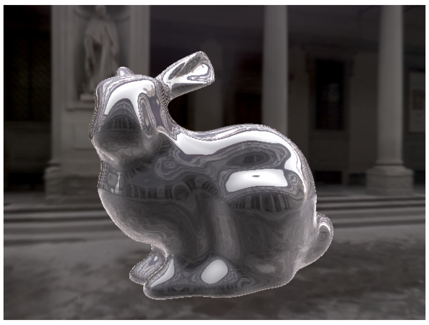
STAMMINGER, M., and DRETTAKIS, G., 2001: Interactive Sampling and Rendering for Complex and Procedural Geometry. In: *Rendering Techniques 2001*.

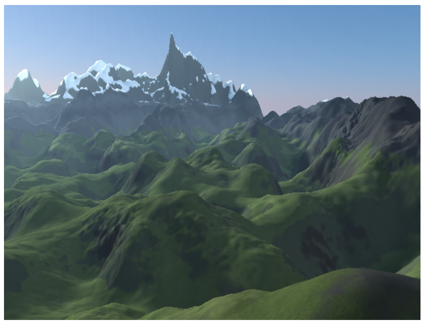
US GEOLOGICAL SURVEY, 2005: *http://edc.usgs.gov/geodata/*

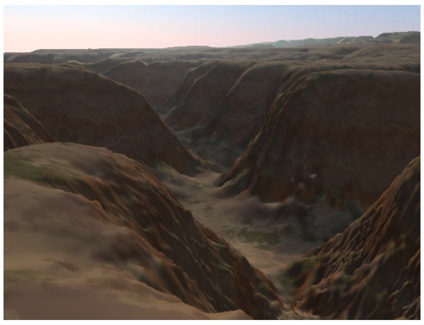
WAND, M., FISCHER, M., PETER, I., MEYER AUF DER HEIDE, F., and STRAßER, W., 2001: The Randomized z-Buffer Algorithm: Interactive Rendering of Highly Complex Scenes. In: *SIGGRAPH 2001 Proceedings, Annual Conference Series*, 361–370.

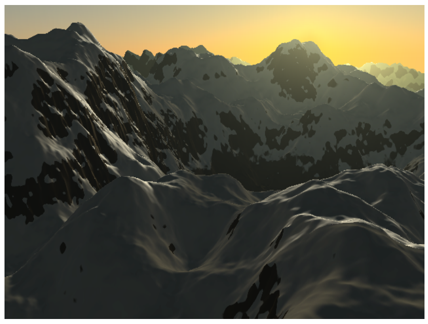
WHITTED, T., and KAJIYA, J., 2005: Fully Procedural Graphics. In: *Graphics Hardware 2005*.

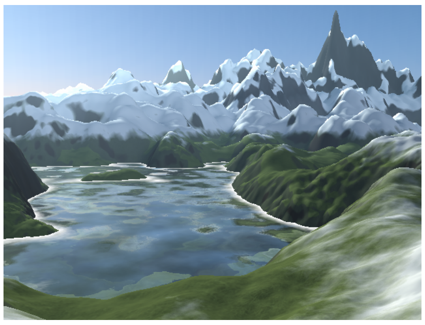
Zorin, D., Schröder, P., DeRose, T., Kobbelt, L., Levin, A., and Sweldens, W., 2000. Subdivision for Modeling and Animation. In: *Siggraph 2000 Course Notes*.

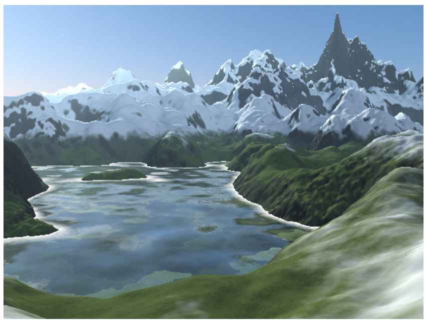

(a) Stanford Bunny (30 ms / 251 ms)


(b) Landscape scene (169 ms / 2438 ms)


(c) Grand Canyon (130 ms / 955 ms)


(d) Mountains at sunset (228 ms / 2487 ms)


(e) Mountain Lake (low resolution, 169 ms / 1079 ms)


(f) Mountain Lake (high resolution, 370 ms / 3241 ms)

Color Plate: *Hardware Accelerated Multi-Resolution Geometry Synthesis*: Figure 8, Application examples.
Timings: rendering from cache / rendering with full rebuild.