Silhouette Partitioning for Height Field Ray Tracing

Tomas Sakalauskas
Vilnius University
Naugarduko 24,
Lithuania,03225,Vilnius
tomas.sakalauskas@prewise.lt

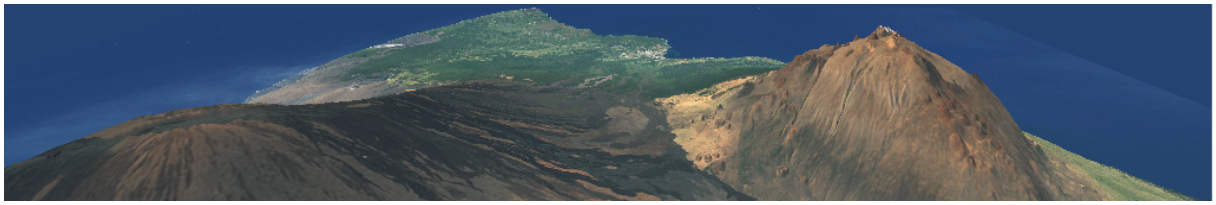
ABSTRACT

This paper presents parallel algorithm to ray trace height fields that is suitable for recent GPUs. No data preprocessing is needed, therefore this algorithm can render dynamic height fields. Partitioning binary tree for screen space is calculated each frame. It takes silhouettes of height field regions as partitioning curves. Ray tracing step uses binary search in this silhouette tree to find height field coordinates of pixel visible at given screen coordinates.

Presented algorithm takes fixed amount of samples into consideration to produce value in each pass, therefore worst-case scenario is deterministic. This enables the implementation for GPUs having limited dependent texture lookups.

Keywords

View dependent partitioning, dynamic height field visualization, terrain rendering, ray tracing, parallel rendering, GPU, silhouette detection.



1. INTRODUCTION

Recent developments in consumer level GPUs enable bringing algorithms and strategies previously used solely in offline rendering to real-time implementations. Despite available fragment processing power, GPU development focuses on providing best triangle rendering performance because this is de-facto standard for representing geometry in most of today's graphical applications. Still number of triangles that modern GPUs can handle is usually order of magnitude lower than number of pixels same GPU can process. If we need to render extremely detailed meshes, where most of the triangles cover just one pixel or less, triangle processing power is the bottleneck limiting complexity of model we can render real time. Height field visualization is one of the areas facing this problem. One way to solve it is finding triangulation that approximates original geometry with minimal amount of triangles. Another option is avoiding triangle based representation of height field by implementing ray-tracing - launching a ray for each screen pixel trying to find closest intersection with height field.

This paper describes ray tracing method well suited for GPU implementation. The main design goal was to create algorithm that needs no preprocessing and uses fixed number of reads to produce pixel as opposed to "scanning" algorithms. Some GPUs are very limited on dependent texture lookups some cannot abort calculations once intersection is found. Therefore GPU friendly algorithm should have predictable behavior in worst-case scenario and be fast enough even in cases when this worst-case scenario has to be executed for every pixel.

The idea for the algorithm presented was born trying to choose visualization method for simulating terrain erosion. It requires detailed representation of constantly changing height field.

2. RELATED WORK

Triangulation Algorithms

One of the first methods generating triangulated-irregular networks (TIN) from digital elevation maps (DEM) was introduced in [Fow79a]. First, they classify the points by automatically choosing some "important" features of the terrain, such as ridges and peaks. Then, they incrementally compute a triangulation of the points; in their case, they chose to use the Delaunay triangulation. At each step, a new point is added to the triangulation until no points are farther from the original surface than a certain predefined threshold. Substantial research has been conducted on creating hierarchical structures on top of TINs [Flo89a, Sca92a, Ber95a]. These methods calculate complete starting triangulation which is either refined by adding or decimated by removing redundant points. These algorithms require heavy preprocessing and keeping complete triangulation representation in memory. [Cla95a] considers input DEM to be an instance of TIN with very high resolution and simplifies this input TIN surface to create new TIN that has a fewer triangles, but is still within a specified error bound of the original surface.

Such algorithms calculate triangulation based on original model minimizing number of triangles and trying to keep error in object space as small as possible. When active view is covering a small region of terrain, few triangles are rendered not using triangle processing potential of GPU.

Level of Detail (LOD) is another family of algorithms. [Lin96a] performs coarse level of simplification to select discrete levels of detail for blocks of the surface mesh, followed by further simplification through repolygonalization in which individual mesh vertices are considered for removal. [Sch06a] presents algorithm that tiles the domain in preprocess, and computes for each tile a discrete set of LODs using a nested mesh hierarchy. Any triangulation recalculation in runtime is avoided. [Sch06a] implementation is more GPU friendly.

LOD algorithms try to minimize error in screen space by selecting correct LOD for region. In general LOD algorithms perform quite well, but do not have consistent triangulation resolution where two segments are joined. Near the seam one segment is over sampled and the other under sampled.

Adaptive algorithms are a mix of static and LOD algorithms –usually trying to create complete optimal triangulation (like static) using error in screen space (like LOD). Theoretically this approach should give smallest screen errors with same triangle number as previous two types of algorithms. [Duc97a] presents real time optimally adapting meshes (ROAM) algorithm that uses regular adaptive triangulation in real time.

Current GPU architecture does not allow changing topology of geometry inside GPU; therefore adaptive algorithms need not only calculate but also transfer geometry in each frame. It is possible to use frame coherence to transfer only the changes in geometry. Frame coherence can also be used to minimize time spent in recalculating triangulations. But such algorithms face performance or quality problems when there is minimal or no frame coherence.

Ray-tracing Algorithms

In its most basic form, height field ray tracing involves traversing rays in steps across height field cells. This procedure is called incremental ray tracing [Mus88a].

Naive generation of one column of the image has a time complexity of O(ml), where l is length of column footprint and m is the number of pixels in image column. [Coh96a] uses ray coherence to achieve O(l) time to render single column of the image. As elevation map structure cannot describe cavities, simple fact is noted that pixel traces to same or further position on elevation map than pixel located directly below it. Therefore ray tracing the next pixel can continue from grid position where intersection was detected in pixel below, vertical coordinate of ray can also be calculated using ray coordinates of previous intersection and view plane. Run-based ray traversal algorithm proposed in [Hen04a] utilizes the fact that mapping ray to discrete grid creates footprint that consists of clusters of connected cells or runs. Analysis of this footprint proves that run length at given ray distance can be determined without a need to iterate runs from ray start. Run-based algorithm performs ray intersection tests on runs instead of individual cells, gaining average 125% performance improvement.

Unfortunately these algorithms do not transfer well to GPU implementation, because calculation results in one pixel cannot be easily transferred to next pixel unless many passes are used. It is also hard to adapt these algorithms to GPUs that do not support loops – worst case scenario for rendering one pixel can require O(l) data reads.

Number of height field ray-tracing steps can be dramatically reduced by traversing rays in steps across inverted cones of empty space [Pag94a]. This method, known as linear parametric ray tracing, requires the empty space above the height field surface to be represented with a set of inverted cones of empty space. There is one inverted cone centered above each height field cell, defined by values of the apex height and opening angle parameters. Such empty space representation is called the linear

parameter plane transform (PPT) and is generated off-line prior to ray tracing. However, steps across inverted cones of empty space along rays close to the base of a steep ridge will be short, even if there are no obstructions along the line of sight, because the cones will be narrow. [Pag98a] describes how this weakness can be virtually eliminated by directionalizing the PPT, i.e., by allowing the opening angles of the inverted cones of empty space to vary between contiguous sectors in the xy plane such that the inverted cones are wider within sectors that are less obstructed. This requires even heavier pre-processing steps thus making algorithm inapplicable for dynamic height fields. It relies on scanning height field so execution time is not stable.

3. HEIGHT FIELDS

Height field is defined for rectangular regularly spaced grid UxV and associates an elevation h to each position (u,v) in the grid.

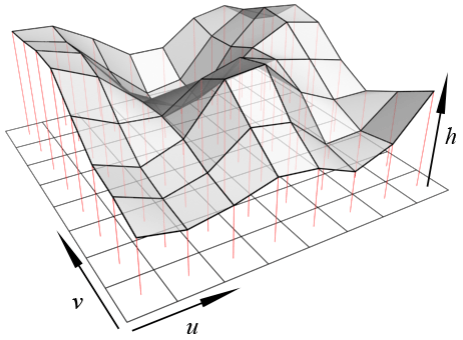


Figure 1. Height field representation.

Height field is a convenient structure to define functions of two parameters, represent real or artificial terrain. Such representation is used to describe geographical data and is called digital elevation maps (DEMs).

Dynamic Height Fields

Often height fields are used to represent static terrain models. In such cases algorithms can move significant amount of calculations to pre-processing stage, where intermediate structures are created optimizing run-time performance. If we use height field for dynamic scenarios like fluid visualization, or some other task requiring real-time animation of the surface, algorithms relying on pre-processing are difficult or impossible to apply.

4. SILHOUETTE PARTITIONING

Top silhouette of rendered height field is the boundary line dividing the screen to area covered by height field and area above the height field. Silhouette line intersects all vertical screen lines exactly once, therefore silhouette curve can be seen as function $S(x) \equiv y$ where y is y coordinate of silhouette intersection with vertical line at x.

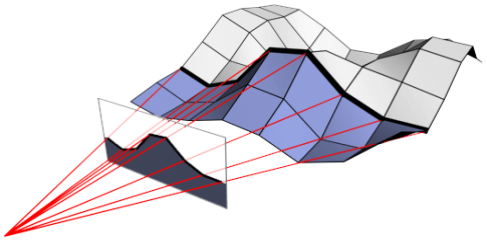


Figure 2. Silhouette.

We can define silhouette for segment of the height field. Fig. 2 shows silhouette $S_{v\le a}(x)$ for segment having v coordinate less or equal to constant a. $S_{v\le a}(x)$ has useful properties:

$$S_{v\le a}(x) > y \Rightarrow trace\_v(x,y) > a \qquad (1)$$

$$\forall a,b,v \in V : a < b \Rightarrow S_{v<a}(x) \le S_{v<b}(x) \qquad (2)$$

Eq.1 comes from silhouette definition. Eq.2 states that silhouette of segment is greater or equal to that of sub-segment.

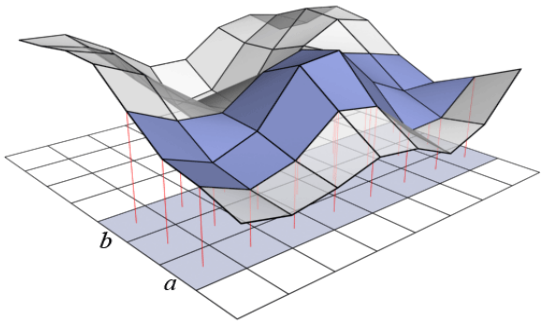


Figure 3. Height field segment.

Another interesting segmentation of height field is range of v values $a \le v \le b$ (Fig.3). We shall mark silhouette for this segment $S_{a<v\le b}(x)$. The following is true:

$$\forall a,b,c \in V, a \le b \le c : \quad S_{a\le v\le c}(x) = \max(S_{a\le v\le b}(x), S_{b\le v\le c}(x)) \qquad (3)$$

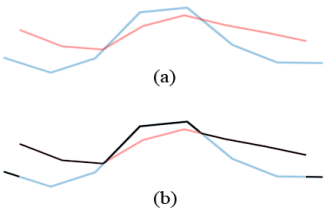


Figure 4. Combining two silhouettes. (a) Source silhouettes (b) Result shown in black.

Binary Segment Tree

We can organize the height field into a binary tree slicing it by v coordinate. Root (*l=0)* node contains the whole $0 \le v \le 1$ height field, marked as $H^0_{[0,1]}$. Tree is defined recursively by splitting height field $H^l_{[a,c]}$ at midpoint $b = (a+c)/2$:

- Left node $H^{l+1}_{[a,b]}$,
- Right node $H^{l+1}_{[b,c]}$.

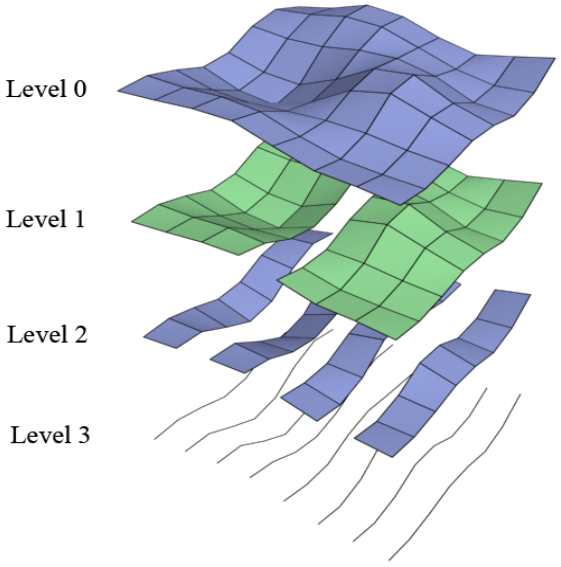


Figure 5. Binary tree.

There are V leaves in this binary tree and they represent discrete v values covering exactly one row in elevation map. Height of the tree therefore is $\log_2(V)$ as can be seen in Fig. 5.

4.1.1 Silhouette Calculation

Each node $H^{l+1}_{[a,b]}$ in the tree described above has silhouette $S^l_{[a,b]}(x)$. We can use bottom-up approach to build silhouettes for the segments in the tree by using Eq.3. Having children nodes $S^{l+1}_{[a,b]}$ and $S^{l+1}_{[b,c]}$, we get parent node silhouette

$$S^l_{[a,c]}(x) = \max(S^{l+1}_{[a,b]}(x), S^{l+1}_{[b,c]}(x)) \qquad (4)$$

Fig. 6. illustrates construction of silhouettes. Left column contains bottom level of the tree – projections of discrete v rows. Pairs of bottom level silhouettes are used to calculate parent silhouette by taking maximum at every x coordinate (Eq.4) until root silhouette is built representing the silhouette for whole height field.

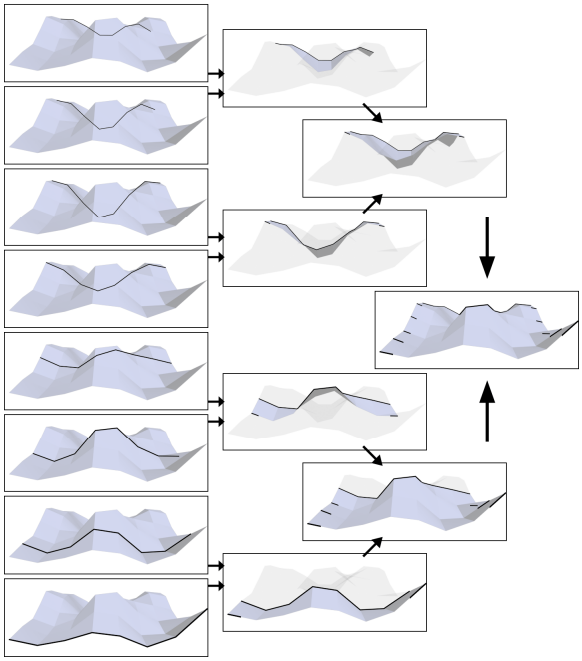


Figure 6. Constructing silhouettes for segment tree.

4.1.2 Calculating Silhouettes for Bottom Nodes of Segment Tree

Bottom level of segment tree contains discrete v values. Silhouette for a line on a grid is equal to projection of that line to screen space.

Rendering surface has discrete x values, therefore sufficient approximation of silhouette is finding $S(x)$ for every x on the screen. $S^n_v(x) \equiv S^n_{[v,v]}(x), n = \log_2 V$ is calculated as intersection of x vertical line with projection of height field at fixed v. Deterministic way to find this intersection is using binary search over row v of elevation map. Intersect function accepts current search position u and range that is being checked - du. For given x and v, calling *Intersect* with $u = 1/2$ and $du = 1$ returns u and y coordinates of an intersection.

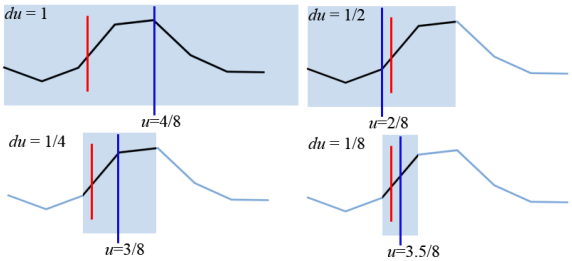


Figure 7. Binary search state is expressed as center of interval *u* and width of interval *du*. Red line represents *x* coordinate being searched.

Pseudo-code for *Intersect* function implementing such binary search is presented below.

```
// x,v – query coordinates
// u,du – recursion search range
Intersect(x,v,u,du)
  if(du==1/U)
    return InterpolateU(x,v,u,du/2);
 if(projected_x(map[u][v])<x)
    return Intersect(x,v,u-du/4,du/2);
  else
    return Intersect(x,v,u+du/4,du/2);
```

Intersect function terminates recursion when $du = 1/U$ – discrete cell in map is traced and u is pointing to middle of that cell. Values of u and y at specific x are calculated performing linear interpolation.

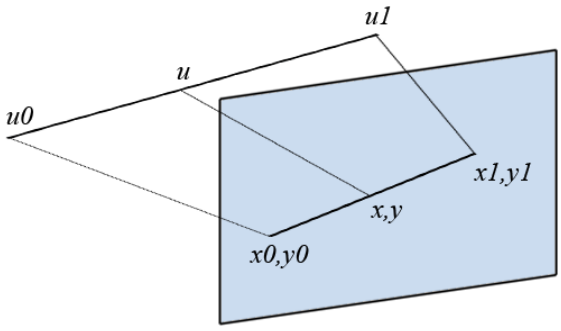


Figure 8. Linear interpolation of *u* and *y* values, based on *x* location in [*x0*, *x1*] interval.

```
InterpolateU(x,v,u,du)
  u0 = u – du/2; //segment start
  u1 = u + du/2; //segment end
  x0 = projected_x(map[u0][v]);
  x1 = projected_x(map[u1][v]);
  y0 = projected_y(map[u0][v]);
  y1 = projected_y(map[u1][v]);
  r = (x-x0)/(x1-x0); // mix ratio
  u_intersect = u0 + (u1-u0) * r;
  y_intersect = y0 + (y1-y0) * r;
```

4.1.3 Data Representation – Silhouette Map

We use algorithm described above to calculate the bottom level for silhouette tree and store u and y values in X x V texture: **R** channel holding y value, **G** channel - u value.

We shall refer to this data structure as silhouette map. Fig.9. illustrates silhouette map (c) for given elevation map (b) when viewed from camera position (f).

Silhouette Partitioning Tree

Silhouette partitioning tree is derived from segment tree and is used as structure that allows binary search in segment tree. Each node in silhouette partitioning tree corresponds to segment in segment tree $H^l_{[a,c]}$ and holds silhouette of closer child $S^{l+1}_{[a,b]}, b = (a+c)/2$ – it is needed when performing binary search. Therefore it has smaller depth than trees described above as the bottom level would represent discrete v values – no child silhouette is available to assign to it.

We mark the silhouette of node at level l S^l_v, where $v = (a+c)/2$ is the middle of range of segment assigned to this node $H^l_{[a,c]}$.

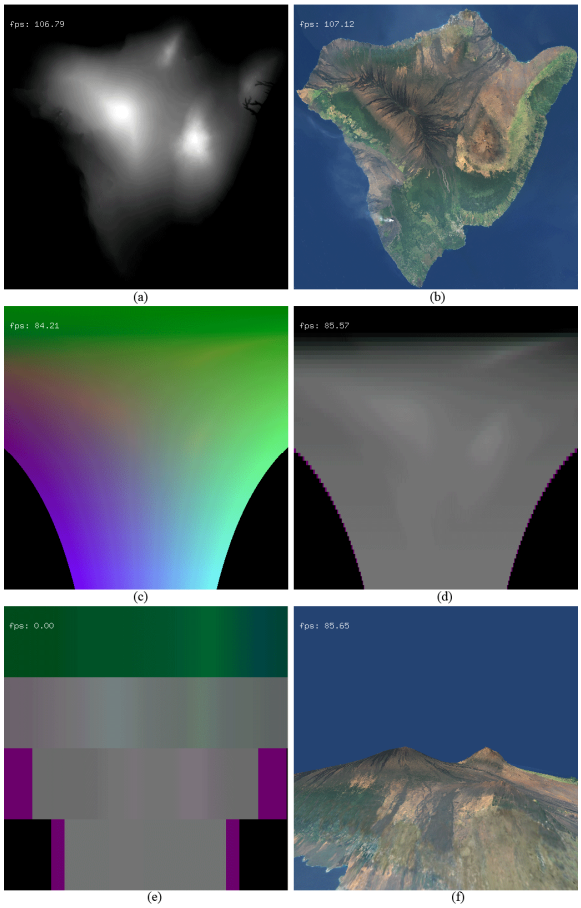


Figure 9. (a) height map (b) color map (c) silhouette map (d) silhouette levels 6-7 (e) silhouette levels 2-3 (f) final render.

4.1.4 Silhouette Partitioning Tree Representation

Bottom level of silhouette partitioning tree contains segments covering two adjacent v rows of elevation map. Silhouettes contained in this level represent discrete v rows – these are calculated and stored in silhouette map as described above.

Each pair of higher levels is packed to single texture:

- **R** – silhouette contained in closer child,
- **G** – silhouette contained in farther child,
- **B** – silhouette contained in parent node.

Calculating higher level silhouettes using Eq.4 requires having silhouettes of segments from level below. Thus we add this information to available texture channel:

- **A** – silhouette for segment in parent node.

4.1.5 *Building Silhouette Partitioning Tree*

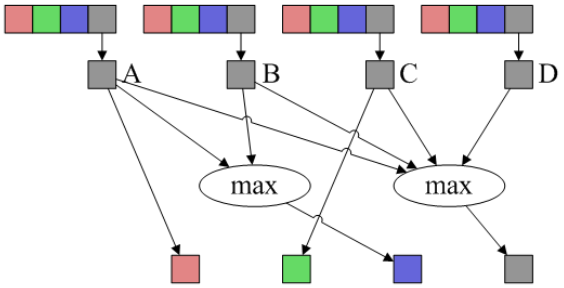


Figure 10. Calculating two levels of silhouette partitioning tree.

Algorithm for building next two levels of silhouette partitioning three is as follows:

- Take 4 silhouettes from previous level as A,B,C and D (A being closest).
- Assign A to closer child node -> **R**,
- Assign C to farther child node -> **G**,
- Assign max(A,B) to parent node -> **B**,
- Assign *max*(A,B,C,D) as silhouette for parent segment used to calculate higher levels -> **A**.

Fig. 9. (d)-(e) shows some levels of silhouette partitioning tree.

5. RAY-TRACING

Ray tracing is finding (u,v) coordinates of height field given screen position (x,y). Knowing how silhouette tree is built we can do binary search to determine v at given (x,y). The approach is similar to one described in *Calculating Silhouettes for Bottom Nodes of Segment Tree (Sec. 4.1.2).* We start at $v=1/2$ and $dv=1$ range covering whole height field; it represents the segment located at the root of the silhouette partitioning tree and go down the tree halving dv until it covers single row of elevation map and v points to middle of that row.

To decide which side to choose when going down the binary tree y is compared to value of silhouette of current node $S_v^l(x)$. Algorithm looks as follows:

```
// x,y – query coordinates
// l – recursion level
// v,dv – recursion search range
Trace(x,y,l,v,dv)
  if(dv==1/V)
    return InterpolateV(x,y,l,v,dv/2);
  if(sil_y[l][v][x]<x)
    return Trace(x,y,l+1,v-dv/4,dv/2);
  else
    return Trace(x,y,l+1,v+dv/4,dv/2);
```

Final values of u and v are calculated performing linear interpolation by y coordinate.

```
InterpolateV(x,y,l,v,dv)
  v0 = v – dv/2; //segment start
  v1 = v + dv/2; //segment end
  y0 = sil_y[l][v0][x];
  y1 = sil_y[l][v1][x];
  u0 = sil_u[l][v0][x];
  u1 = sil_u[l][v1][x];
  r = (y-y0)/(y1-y0); //mix ratio
  u_intersect = u0 + (u1-u0) * r;
  v_intersect = v0 + (v1-v0) * r;
```

Having u and v for screen coordinate (x,y) texture lookup is done to determine color of pixel.

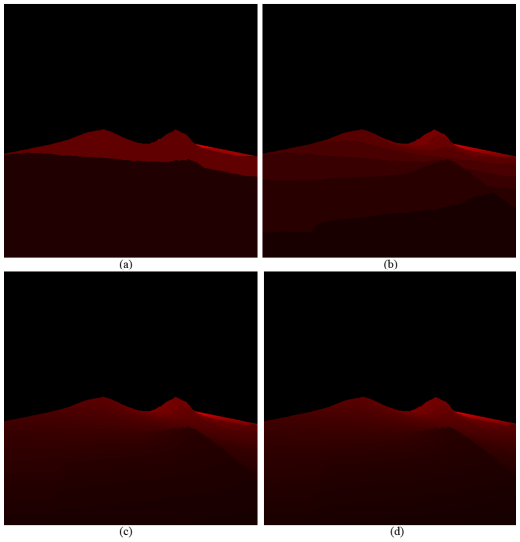


Figure 11. Trace function results (a) *l*=1 (b) *l*=3 (c) *l*=5 (d) l=9.

Ray Tracing Implementation

On GPUs not limiting number of dependent texture lookups ray tracing can be performed in a single pass. If instruction count or texture lookups are limited a separate pass can be used for every two levels in silhouette partitioning tree rendering v value, which is the read in next pass and refined further. Fig. 11. shows refinement v after various passes of tracing.

6. BALANCING GPU

Described algorithms fully rely on performance of fragment shader, leaving vertex shader idle. Calculating silhouettes for bottom level of segment tree can be moved to vertex shader or balanced between these two shaders.

6.1.1 *Silhouettes in Vertex Shader*

Every row in silhouette map corresponds to single row in elevation map. Silhouette map width is equal to screen width and line segments in original geometry map to the same x range as in final rendering. We can define view dependent projecting transformation between height field coordinate space and silhouette map, then render each row of elevation map as list of line segments connecting discrete u values contained in that row.

Vertex shader receives vertices with (u,v) coordinates projects them to (x,y) using camera projection. It ouputs (x,v) as transformed coordinates. This maps segment to correct location in silhouette map. Vertex shader outputs u and y values that are interpolated and passed to fragment shader. Fragment shader simply writes these values to target texture. This approach involves primitive counts comparable to brute force rendering of full height-field triangulation.

6.1.2 Splitting the Work

Similar idea can be used to render segments of row covering more than one cell. U is split evenly into $N = 2^n$ segments representing n^{th} level of binary search and these segments are rendered using same vertex shader as described above.

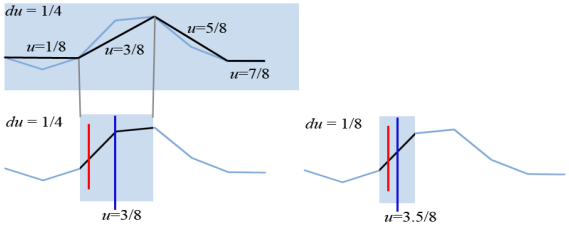


Figure 12. Vertex shader replacing two levels of binary search of *Intersect* function.

Fragment shader is given parameter du specifying the length of such segments. Based on interpolated u value it determines which segment is being rendered at given pixel and calculates center u of that segment. It can proceed as if *Intersect*(x,v,u,du) was invoked. Number of segments to use per row is determined by trial and error, optimizing load balancing between vertex and fragment shaders.

7. RESULTS

The algorithm was tested with two data sets. Artificial 2048x2048 terrain was generated using Perlin noise functions and texture was modeled using ecosystem approach. Another data is based on DEM and LandSat5 data of Hawaii Island 2048x2048.

The tests were conducted on AMD Athlon 64 3500+, 1GB RAM, NVidia GeForce 7900 GTX.

Data	Resolution	Min Fps	Max Fps
Art2K	512x512	84	135
Hawaii2K	512x512	84	135
Hawaii2K	1024x1024	37	55

8. CONCLUSIONS

Algorithm presented in this paper has well controlled worst case scenario and very stable performance – slowest frames are rendered only 33-38% slower than frames where whole terrain is completely off-screen. Variation comes from vertex shader – when segment is not projected to visible area it is culled by hardware. Algorithm is stable regarding input data as well– it runs at constant speed for given camera position independent of elevation map content.

Future research will focus on visualization of bigger data sets, 6 degrees of freedom camera movement and possible optimizations of the current method.

9. REFERENCES

[Ber95a] M. de Berg and K. Dobrindt. On levels of detail in terrains. In *Proc. 11th Annu. ACM Sympos.Comput. Geom.*:C26-C27, 1995.

[Cla95a] Claudio T. Silva, Joseph S. B.Mitchell, and Arie E. Kaufman. Automatic generation of triangular irregular networks using greedy cuts. In *Proc. Visualization '95*, 1995.

[Coh96a] Daniel Cohen-Or, Eran Rich, Uri Lerner, Victor Shenkar. A real-time photo-realistic visual flythrough. *IEEE Transactions on Visualization and Computer Graphics*. 2(3):255–264, 1996.

[Duc97a] Mark Duchaineauy, Murray Wolinsky, ROAMing Terrain: Real-time Optimally Adapting Meshes, *IEEE Visualization '97 Proceedings*, 1997.

[Flo89a] L. De Floriani. A pyramidal data structure for triangle-based surface representation. *IEEE Comput. Graph. Appl*. 9:67-78, 1989.

[Fol91a] J. Folby, M.Zyda, D.Pratt, R. Mackey. Npsnet: Hierarchical data structures for real-time three dimensional visual simulation. *Computers and Graphics*, 17(1): 437-446, 1991

[Fow79a] R. J. Fowler and J. J. Little. Automatic extraction of irregular network digital terrain models. *Computer Graphics*, 13(2):199-207, 1979.

[Hen04a] C.Henning, P.Stephenson. Accelerating the Ray Tracing of Height Fields. *Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*: 254-258, 2004.

[Lin96a] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust. Real-Time, Continuous Level of Detail Rendering of Height Fields. *Proceedings of ACM SIGGRAPH 96*: 109-118, 1996.

[Mus88a] F. Kenton Musgrave. Grid Tracing: Fast Ray Tracing For Height Fields. *Research Report YALEU/DCS/RR-639*, 1988.

[Pag94a] Paglierioni, D., Petersen, S. Terrain visualization by ray tracing a conical height field transformation, U.S. Patent 5,355,442, issued Oct. 11, assignee: Loral Western Development Laboratories, 1994.

[Pag98a] Paglieroni, D. The directional parameter plane transform of a height field. In *ACM Transactions on Graphics*. 17(1): 50-70, 1998.

[Sca92a] L. Scarlatos and T. Pavlidis. Hierarchical triangulation using cartographics coherence. *CVGIP: Graph. Models Image Process*. 54(2):147-161, 1992.

[Sch06a] J. Schneider, R. Westermann. GPU-Friendly High-Quality Terrain Rendering. *Journal of WSCG*, 2006