Hybrid Terrain Shadow Ray Casting

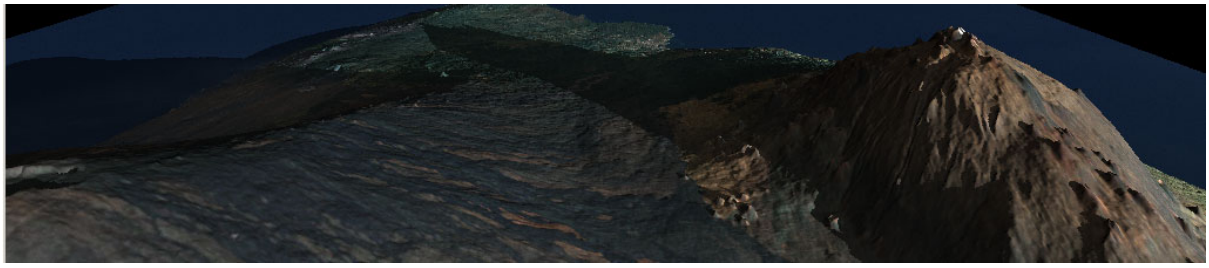
Tomas Sakalauskas
Vilnius University
Naugarduko 24,
Lithuania,03225,Vilnius
tomas.sakalauskas@prewise.lt

ABSTRACT

This paper describes an efficient algorithm for real-time terrain shadowing by directional light. The main concept for this algorithm is fast preprocessing stage in model space that dramatically speeds up shadow calculation for pixels in rasterization phase. Preprocessing is very fast and can be performed every frame still achieving interactive frame rates. Therefore it is suitable for dynamic height field and moving light source visualization.

Keywords

Landscape shadows, dynamic height field visualization, terrain rendering, ray casting, GPU, silhouette detection.



1. INTRODUCTION

Shadows play important role in perception of three-dimensional objects. It is difficult to determine landscape feature dimensions and placement on the terrain without proper shadows and lighting.

Displaying large terrain is a difficult task by itself, rendering shadows makes this problem even harder. Ever growing processing power of modern GPUs makes some algorithms formerly used for offline rendering and image processing available on enthusiasts desktop in real-time. Still, amount of the data that has to be processed in a landscape shadow calculation can choke even the fastest GPUs available today.

This article presents data structures and algorithms that enable precise real-time shadow calculations for large terrains. Our algorithm is fully suited for dynamic height-field visualization as it relies on light-weight preprocessing which can be performed each frame. Even though it is done every frame, we refer to this stage as preprocessing because it prepares the structures used in the actual rendering pass.

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.

Implications of rendering algorithm selection

When a landscape is rendered using LOD (level of detail) or other adaptive algorithms, rendered geometry is not always the best choice for shadow calculation. For example some feature may be simplified as a result of visibility check or because such simplification would produce small visual error in screen space. But the simplification or elimination of shadow casted by such feature may have a dramatic impact on overall rendering quality. Our algorithm uses original height-map for shadow calculation so it can be integrated into any landscape rendering algorithm as long as we can determine height-map coordinates of pixel being rendered.

2. RELATED WORK

We focus on hard shadow algorithms, analyzing their suitability for landscape rendering. Analysis of

broader range of shadow algorithms is given in [Ber95a].

Hard shadow algorithms can be classified to object space algorithms, dealing with vector representation of objects, and image space algorithms that project objects to shadow maps and use the maps in rendering phase. Hybrid algorithms, like one described in [Cha04a], use shadow maps for points that are fully lit or shadowed and fall back to object space algorithms in boundary conditions to avoid aliasing problems.

Shadow maps

A shadow map [Wil78a] is z-buffer of scene rendered from light position. To determine if a point lies in a shadow, its depth in light-space is compared to the value in the shadow map.

Shadow maps suffer from aliasing artifacts – when points projected to distant locations in screen space are mapped to same or neighbor pixels in the shadow map.

In [Sta02a] shadow maps are generated in normalized device coordinate space, i.e., after perspective transformation. This gives better shadow map resolution in areas close to the camera. Still the resulting precision is view-dependent and does not completely eliminate aliasing.

Due to floating point precision errors shadow maps do not work very well for large self-shadowing objects such as landscapes.

Shadows in object space

Atheron et. al. [Atr78a] proposes subdivision of polygons in the scene, clipping projected shadow source polygons to receiver polygons. Every polygon after such subdivision is fully lit or shadowed. Correct material for every polygon has to be selected and the scene can be rendered as usual.

Blinn [Bli88a] projects the vertices of shadow source object to a plane of receiver polygon and uses resulting polygons to modulate the surface color.

Due to the large amounts of polygons involved in landscape rendering, full scene triangulation is usually not performed – some adaptive algorithms may be used instead, or landscape remains in original height-map form that is used for ray-casting.

Another object space technique is shadow volume algorithm [Cro97a]. Shadow volume for an object is semi-infinite volume behind the shadow source object extending from the silhouette lines as seen from the light source. Shadow volume divides object space in two: areas that are in shadow and areas that are not. To detect whether a particular point is in the shadow, crossings between shadow volume polygons and any ray extending from that point are counted.

Shadow volumes can be hardware accelerated using stencil buffers to count crossings [Hei91a]. Bounding polygons of shadow volume are rendered using the camera transformation, incrementing or decrementing stencil buffer value for every pixel based on the face orientation. Only the pixels having stencil value zero are lit. This algorithm easily becomes fill-bound if light direction approaches horizontal. Also the amount of vertex processing power needed to shadow large terrain is huge.

Yet another approach is ray-casting [App68a]. The idea behind ray casting is to shoot rays from eye, one per pixel and find the closest object blocking the path of that ray. Hard shadow determination using ray casting is a simplified problem in the sense that only the existence of objects between the point being rendered and the light source needs to be determined without a need to identify the object blocking the ray.

We use a hybrid approach that shows some properties of shadow maps, shadow volumes and ray-casting.

3. BRUTE FORCE RAY CASTING

Given the directional light

$$L = (L_u, L_v, L_w), L_v = 1,$$

calculating shadow for a model point $P = (P_u, P_v, P_w)$ involves detecting whether a ray, starting at point P and going direction opposite to L, intersects any geometry.

Brute force approach for terrain shadow detection is straightforward. We trace height map from the point P along $-L$ direction checking the height map points T^i at integral v positions:

$$T^i = P - L * (v_i - P_v), 0 \le v_i < P_v.$$

$$T_h^i \equiv height(T_u^i, T_v^i)$$

If at any trace point we go below the surface - $\exists T^i, T_w^i < T_h^i$,

P is in shadow, otherwise it is lit.

This is the algorithm for brute force shadow calculation:

```
IsShadow(P)
  d0 = fract(P.v); // distance to integral V
  T = P-L*d0; // trace point
  while(Tu,Tv inside heightmap)
  {
    // get heightmap value at Tu,Tv
    h = height[Tu,Tv]
    if(h>Tw)// tracepoint below surface
      return true;
    T = T – L;
  }
  return false; // no hit detected
```

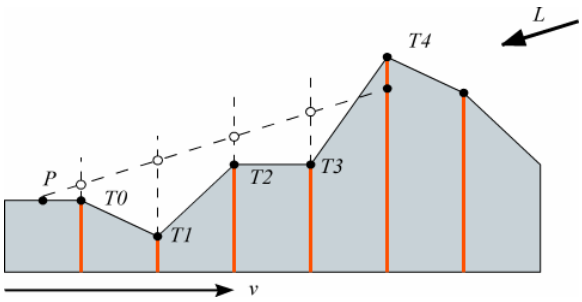


Figure 1. *Scanning the height map.*

Fig.1. illustrates the process. T0 is a point with integral v, where the scan starts. T0, T1, T2, T3 are scanned, but the tracing position passes above the landscape. T4 generates a hit and the search is stopped.

Silhouette points

It may be noted that it is sufficient to check only those height map positions T^i that form up a silhouette when looking from light position. Point T^i is a silhouette point if the segment before T^i is facing the light and the segment after T^i is back-facing or

$$T_h^i > T_h^{i-1} + L_w \text{ and } T_h^i > T_h^{i+1} - L_w \qquad (1)$$

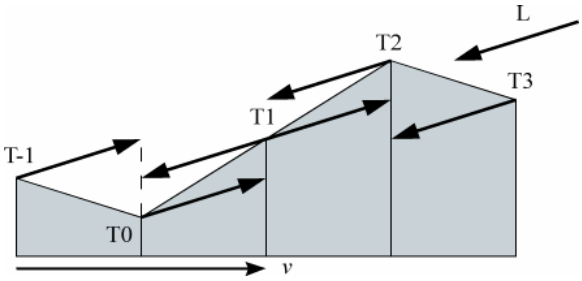


Figure 2. *Silouette detection.*

Fig. 2. illustrates the silhouette detection – only T2 is a silhouette point as projections of its neighbours T1 and T3 are below T3 itself. Points T0 and T1 may be skipped when performing ray-casting.

```
IsSilhouette(h0,h1,h2)
  if h1>h0+Lw and h1>h2-Lw
    return true;
  else
    return false;
```

This observation suggests that we could greatly minimize the number of positions visited if we knew how to skip through non-silhouette points.

4. APPROACH

To understand our approach, consider a beam of rays that all cross a single v line within a segment ab of unit length.

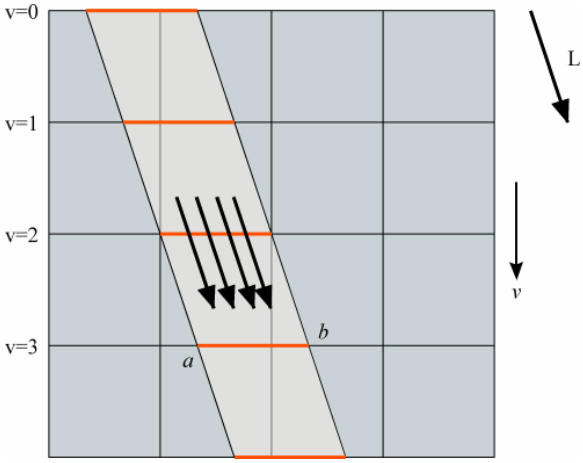


Figure 3. *Beam having unit cross section ab.*

Shadow calculation for a landscape section covered by such beam of light can be optimized if we perform two preprocessing steps:

- For every v along the beam check if there are any silhouette points in cross-section ab, that satisfty (1). If such points exist we mark v position **Black**, otherwise it is marked **White**. Section 5 describes this step.
- For every v determine the closest v that is marked **Black** going the light direction. See Section 6.

SMap

For a beam we make a structure - SMap. For any v SMap[v] defines Color and NextV. SMap allows us to rewrite shadow detection algorithm, to skip over non-silhouette (white) points:

```
IsShadow(P)
  // advance to integral V
  d0 = fract(P.v);
  T = P-L*d0; // trace point

  if SMap[Tv].Color = White
    // advance to next black V
    T = T -L*(SMap [Tv].NextV-Tv)

  while(Tu,Tv inside heightmap)
  {
    h = height [Tu,Tv]
    if(h>Tw)
      return true;

    // advance to next black V
    T = T -L*(SMap[Tv].NextV-Tv)
  }
  return false;
```

Fig.4. shows this algorithm in action.

 ISBN 978-80-86943-16-9

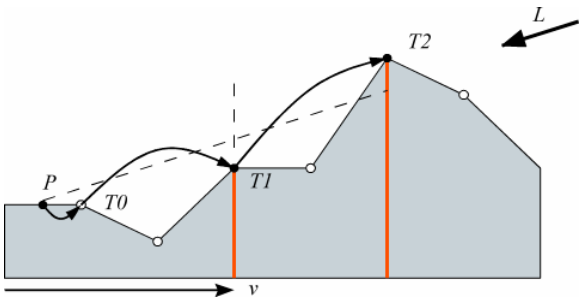


Figure 4. *Scanning for shadow. P – target point, T0 – trace at integral v, T1 – first black vertex (below trace line), T2 – second black vertex (hit).*

The algorithm scans only Black V's. According to the statistics on our sample data for different view positions, black points occupy 5.7% of the total data in the height map in average. Therefore efficiency of per-pixel ray casting process is greatly improved.

We may cover all landscape with an array of beams each having unit cross-section, then calculate SMap for each beam, and finally make IsShadow function select the correct beam before performing the scan. Section 7 describes a way to store SMap in GPU texture.

5. DETECTING SILHOUETE LINES

To calculate coordinates of a point P in cross-section ab (see Fig.3.) we have to read 3 points from the height map:

A = (floor(a), v, height[floor(a), v])

B = (Au+2, v, height[Au+2, v])

C = (Au+1, v, height[Au+1, v])

Corrdinates of the point P are then found by interpolating A, B and C:

```
GetPoint(A,B,C,u)
  if(u<Cu)
    return mix(A,C,u-Au)
  else
    return mix(C,B,u-Bu)
```

Fig.5. shows calculation of point P when u is less than C_u.

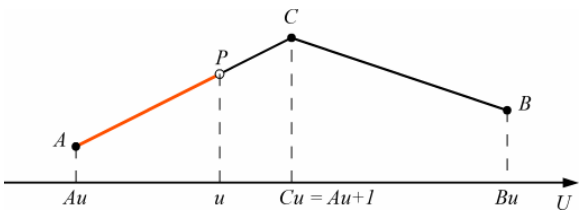


Figure 5. *Interpolation of P at u.*

Detecting the color of a particular element SMap[v] is done by analyzing cross-section segment ab, searching for silhouette points. As integral coordinates in height map define extreme points, examining 5 positions for each SMap element is sufficient.

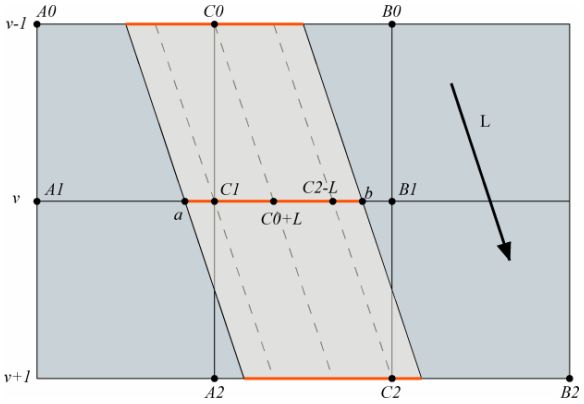


Figure 6. *Detecting the color for SMap[v].*

In case any of the points with the following u values $a, b, C1_u, C0_u + L_u, C2_u - L_u$ (Fig.6.) are silhouette points, we mark element **Black**.

We get all non-grid points needed for such calculation using GetPoint function, so we need to read 9 height-map values as seen from Fig. 6.

The following code implements this logic:

```
// u,v coordinates of a
HasSilhouette (u,v)

  // line v
  a1 = u
  b1 = a1 + 1
  v1 = v
  u1 = floor(a1)
  A1 = (u1,v1,height[u1,v1])
  B1 = (u1+2,v1,height[u1+2,v1])
  C1 = (u1+1,v1,height[u1+1,v1])

  // line v-1
  a0 = u - Lu
  b0 = a0 + 1
  v0 = v - 1
  u0 = floor(a0)
  A0 = (u0,v0,height[u0,v0])
  B0 = (u0+2,v0,height[u0+2,v0])
  C0 = (u0+1,v0,height[u0+1,v0])

  // line v+1
  a2 = u + Lu
  b2 = a2 + 1
  v2 = v + 1
  u2 = floor(a2)
  A2 = (u2,v2,height[u2,v2])
  B2 = (u2+2,v2,height[u2+2,v2])
  C2 = (u2+1,v2,height[u2+1,v2])

  // check if a is silhouette
  U = a1
  p0 = GetPoint(A0,B0,C0,U-Lu)
  p1 = GetPoint(A1,B1,C1,U)
  p2 = GetPoint(A2,B2,C2,U+Lu)
  if IsSilhouette(p0,p1,p2)
    return true;

  // check if b is silhouette
  U = b1
  ...
  // check if C1 is silhouette
```

 ISBN 978-80-86943-16-9

```
U = C1u
...
// check if C0+L is silhouette
U = C0u+Lu
...
// check if C2-L is silhouette
U = C2u-Lu
...

return false;
```

Fig. 7. shows silhouette points (black) of our sample data calculated from 2 light directions. Black area is relatively small (5.7% of total map).

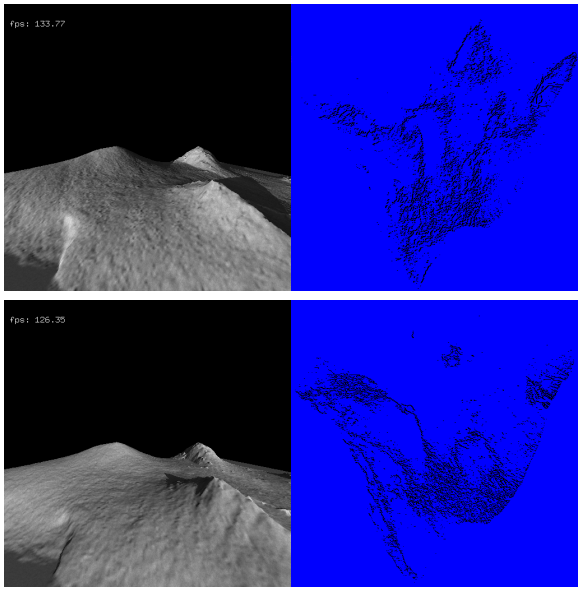


Figure 7. *Examples of silhouette detection pass.*

6. FINDING BLACK V

Finding closest black element in SMap with linear scan in the worst case is performed in $N^3/2$ lookups, which is no better than linear scan for per-pixel shadow detection. Unlike the screen pixels, SMap elements make a regular structure and logarithmic algorithms can be used. We do this by running the search $\log_2 N$ times, each time doubling the search block size DIST.

Fig.8. shows SMap being built. Different shades of blue above SMap mark the range of DIST, arrows on the top show lookupV - position that is checked for NextV, which is at the first element in the next range. If it contains back point lookupV is used, otherwise NextV stored at that position is taken. Square arrows below SMap show NextV pointers after the step.

Each step of algorithm ensures that blocks of size 2*DIST have properly connected elements. Such blocks are separated by dashed line. Each step merges 2 blocks from a previous step.

Step-0. DIST=1

For every element SMap[v]

```
FindNextV(DIST,v)
  If v mod 2 = 1 // even V coordinate
  {
    lookupV = v - 1; // check odd V
    if SMap[v-1].Color = Black
       SMap[v].NextV = v-1;
    else
      SMap[v].NextV = unknown;
  }
```

Step-n. DIST= 2^n

For every element SMap[v]

```
FindNextV(DIST,v)
  If v/DIST mod 2 = 1
  {
    If SMap[v].NextV = unknown
    {
      lookupV = v – (v mod DIST) - 1;
      if SMap[lookupV].Color = Black
        SMap[v].NextV = lookupV;
      else
        SMap[v].NextV = SMap[lookupV].NextV;
    }
  }
```

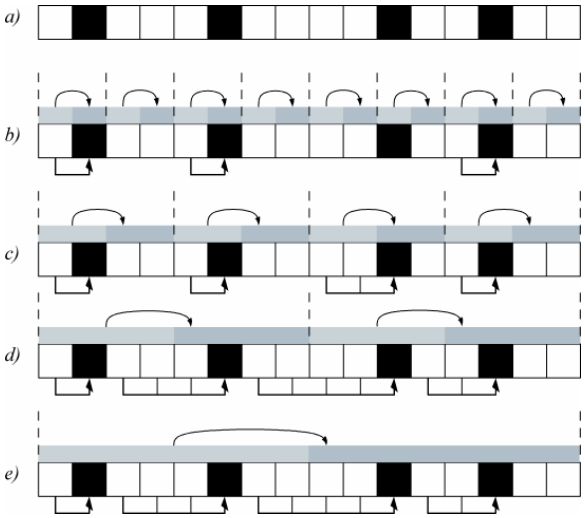


Figure 8. *Finding black V: a) source SMap, b) DIST=1, c) DIST=2, d) DIST=4, e) DIST=8.*

Step-n is generic - it can be also used for step-0 if we set SMap[v].NextV to *unknown* in color detection phase.

Optimization

Steps needed to compute SMap can be halved if we merge 4 blocks from a previous step instead of 2. This is the rewritten algoritm:

```
FindNextV(DIST,v)
{
  lookupV = v – (v mod DIST) - 1;
  minLookupV = v - v mod DIST*4;

  while lookupV >= minLookupV
  {
    If SMap[v].NextV = unknown
    {
      if SMap[lookupV].Color = Black
```

```
      SMap[v].NextV = lookupV;
    else
      SMap[v].NextV = SMap[lookupV].NextV;
  }
  lookupV = lookupV – DIST;
 }
}
```

Fig. 9. Illustrates optimized calculation.

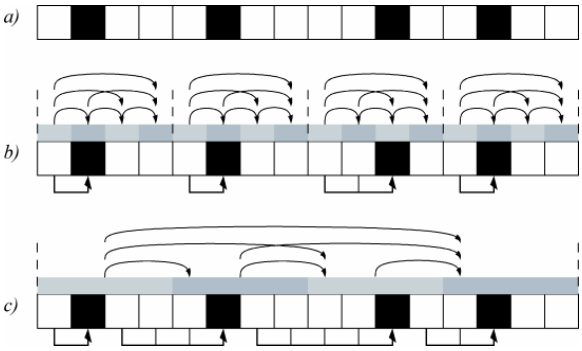


Figure 9. *Optimized black V search: a) source SMap, b) DIST=1, c) DIST=4.*

7. SMap TEXTURE FORMAT

Each beam and grid line cross-section has unit length, but it takes more than N beams to cover whole landscape as Fig. 10. illustrates.

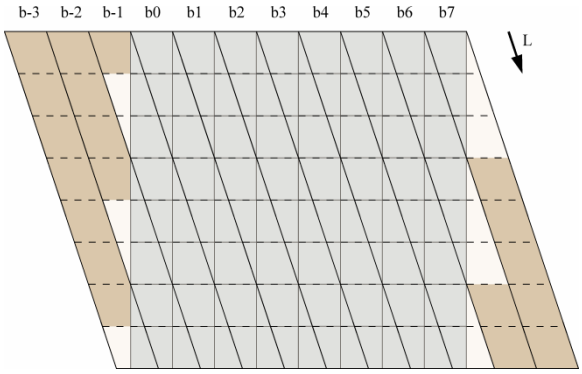


Figure 10. *Beams.*

SMap can be stored in texture with dimensions (N+1) x N as brown elements contain no relevant information. Pixel (u, v) in SMap texture contains information about cross-section ab of line v, where:

$$a = u + fract(L_u * v),$$
$$b = u + 1 + fract(L_u * v) \quad (2)$$

8. IMPLEMENTATION

We define two helper functions for mapping between landscape coordinates and SMap texture coordinates:

```
// returns beginning of cross-section
// represented at u,v position of SMap
GetA(u,v)
  return u + fract(Lu*v)

// returns SMap u-coordinate that holds
// information about given point
GetU(a,v)
  return a – fract(Lu*v);

Pass0(u,v) // detect color
  a = GetA(u,v)
  if(HasSilhouette(a,v))
    SMap[u,v].Color = Black
  else
    SMap[u,v].Color = White
  SMap[u,v].NextV = unknown

Pass1stepN(u,v,DIST) // find V
  a = GetA(u,v)
  lookupV = v – (v mod DIST) - 1;
  minLookupV = v - v mod DIST*4;

  while lookupV >= minLookupV
  {
    If SMap[u][v].NextV = unknown
    {
      lookupA = a + (lookupV-v)*Lu;
      lookupU = GetU(lookupA,lookupV);
      lookup = SMap[lookupU][lookupV];
      if lookup.Color = Black
        SMap[u][v].NextV = lookupV;
      else
        SMap[u][v].NextV = lookup.NextV;
    }
    lookupV = lookupV – DIST;
  }
```

Integrating shadow calculation

Landscape rendering is out of scope of this article, but rewritten **IsShadow** function can be integrated to any rendering algorithm. It detects whether a point $P = (u, v, w)$ in landscape coordinate system is shadowed. It may be used for any points, not just landscape surface.

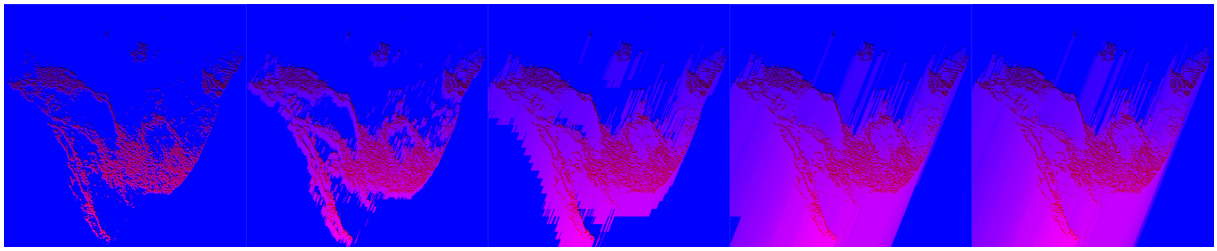


Figure 11. *Successive steps searching for black V (512x512). DIST=1,4,16,64,256. Red - NextV, Blue – Color.*

```
IsShadow(P)
  // advance to integral V
  d0 = fract(P.v);
  T = P-L*d0;

  lookupU=GetU(Tu,Tv)
  lookup = SMap[lookupU,Tv]
  if lookup.Color = White
    // advance to next black V
    T = T -L*(lookup.NextV-Tv)

  while(Tu,Tv inside heightmap)
  {
    h = height [Tu,Tv]
    if(h>Tw)
      return true;
    // advance to next black V
    lookupU=GetU(Tu,Tv)
    lookup = SMap[lookupU,Tv]
    T = T -L*(lookup.NextV-Tv)
  }
  return false;
```

9. OPTIMIZATIONS

The following additional optimizations were done to improve the performance of the algorithm.

Choosing better NextV values

Instead of choosing closest black V actual implementation performs more complex analysis when setting NextV value of SMap. If silhouette point is found, it is not stored to NextV if the silhouette cannot shadow the segment ab being analyzed. Additionally, NextV value is overwritten when another silhouette that completely shadows currently stored NextV is found. This greatly reduces the number of points to check when performing scan in IsShadowed function.

Parallel computation of NextV

FindNextV algorithm was rewritten to calculate 4 neighboring points by encoding the information into RGBA channels of the texture and using GLSL functions that handle vec4 data.

10. RESULTS

The algorithm was tested with DEM and LandSat5 data of Hawaii Island 2048x2048. Two resolutions of data were used – original 2048x2048 and resized 512x512. The tests were conducted on AMD Athlon 64 3500+, 1GB RAM, NVidia GeForce 7900 GTX.

To get consistent results, all measurements were done using fly-by camera path consisting of 1000 frames. Table 1 presents the actual results of our algorithm.

First row shows the performance of underlying terrain renderer without any shadow-related work being performed. Full scan is brute-force ray-casting performance. Binary and quad search modes illustrate performance gains of optimization described in section "*Finding Black V*". Performance boost of quad search with bigger data sets can be explained by minimized memory footprint of intermediate structures. Parallel computation of NextV reduces memory and calculations requirements even further.

A test with precomputed SMap shows performance of the rendering stage only with SMap for static light source generated before the rendering.

11. CONCLUSIONS AND FUTURE WORK

We have presented an algorithm that achieves the precision of brute-force ray-casting in a fraction of time. This enables rendering big terrain models at real-time and achieves interactive frame rates even for large data sets.

It is a hybrid approach that scales much better than pure shadow-map solutions with increasing screen resolution. Proposed object space parameterization for a shadow-map does not depend on camera and light relation, giving consistent results from all viewing angles.

Current algorithm deals with directional light as it assumes constant unit length cross-section between beams of light and height-map grid lines. Although color detection for SMap element is more difficult with variable length cross-sections, the rest of the algorithm is suitable for point light sources as well.

Terrain size	**512x512**			**2048x2048**		
Algorithm	**Min Fps**	**Max Fps**	**Avg Fps**	**Min Fps**	**Max Fps**	**Avg Fps**
No shadow	57.49	80.82	69.08	28.64	37.29	30.94
Full scan	4.13	15.63	8.18	1.25	4.70	2.32
Binary search V	31.08	39.29	34.53	2.10	2.14	2.12
Quad search V	32.45	40.00	35.80	5.04	5.21	5.10
Parallel quad V	37.85	47.72	42.61	9.27	10.57	10.02
Precompute SMap	50.44	65.05	58.65	21.07	35.18	26.24

Table 1. *Performance results.*

12. REFERENCES

[App68a] A. Appel. Some Techniques for Shading Machine Renderings of Solids. *Proc.AFIPS JSCC*, vol. 32, pp. 37-45, 1968.

[Ath78a] P. Atherton, K. Weiler, and D. Greenberg. Polygon shadow generation. *Computer Graphics (SIGGRAPH '78 Proceedings)*, vol. 12, 275-281, 1978

[Bli88a] J. Blinn, Me and My Fake Shadow. *IEEE Computer Graphics and Applications*, 8(1), 82-86, 1988.

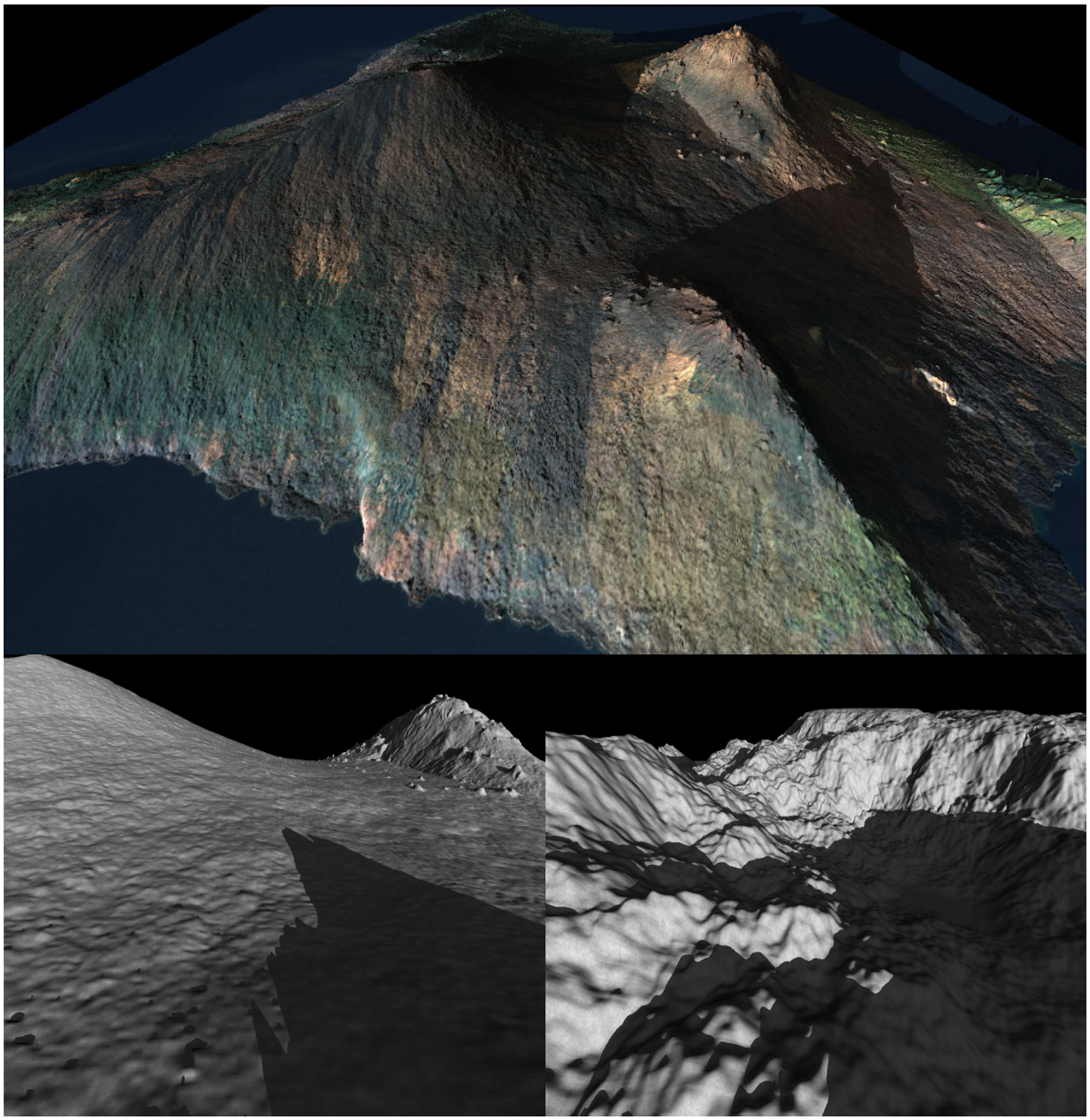
[Cha04a] E. Chan and F. Durand. An efficient hybrid shadow rendering algorithm. *Proceedings of the Eurographics Symposium on Rendering*, pp. 185

[Cro77a] F. Crow. Shadow algorithms for computer graphics. *Computer Graphics (SIGGRAPH '77 Proceedings)*, vol. 11, 242-248. 1977.

[Hei91a] T. Heidmann. Real shadows real time. *IRIS Universal*, 18. 1991.

195, 2004.

[Sta02a] M. Stamminger and G. Drettakis. Perspective shadow maps. *Proceedings of ACM SIGGRAPH 2002*, 557-562. 2002.

[Wil78a] L. Williams. Casting curved shadows on curved surfaces. *Computer Graphics (SIGGRAPH '78 Proceedings)*, vol. 12, 270-274, 1978.

[Woo90a] A. Woo, P. Poulin, A. Fournier. A Survey of Shadow Algorithms. *IEEE Computer Graphics and Applications*, 10(6):13–32, November 1990.