Volume 0 (1981), Number 0 pp. 1–8 COMPUTER GRAPHICS *forum*

GPU Rendering of Relief Mapped Conical Frusta

D. Bhagvat, S. Jeschke, D. Cline and P. Wonka

Arizona State University

Abstract

This paper proposes to use relief-mapped conical frusta (cones cut by planes) to skin skeletal objects. Based on this representation, current programmable graphics hardware can perform the rendering with only minimal communication between the CPU and GPU. A consistent definition of conical frusta including texture parametrization and a continuous surface normal is provided. Rendering is performed by analytical ray casting of the relief-mapped frusta directly on the GPU. We demonstrate both static and animated objects rendered using our technique and compare to polygonal renderings of similar quality.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Display algorithms I.3.7 [Three Dimensional Graphics and Realism]: Color, shading, shadowing and texture

1. Introduction

Today the data bandwidth between CPU and GPU is often a bottleneck for hardware rendering performance. Additionally, despite rapid increases in capacity, GPU memory usage remains a concern for complex scenes. One solution to the bandwidth problem is to generate low level graphical primitives (triangles, points and lines) on the GPU in a geometry shader or tessellator. Thus, the CPU can process and send some high-level object description to the graphics board where the GPU does the bulk of the modeling and rendering work.

Another approach to rendering once again defines high level primitives to allow fast data transfer between the CPU and GPU. However, instead of approximating the primitives with a large number of triangles, the GPU directly renders these high level primitives by ray casting. The contribution of this paper is to present such a representation that allows for interactive high quality renderings of complex objects while keeping CPU workload and CPU to GPU communication very low. We restrict our representation to objects that are described as skeletons, consisting of line segments connected by joints. Objects falling into this category include trees, snakes, columns, and other objects composed of rotationally symmetric segments. The surface around each segment is algebraically defined as a conical frustum, similar to the cone spheres of Max [Max90]. The CPU sends this "lightweight" data structure to the GPU and the geometry shader generates some bounding geometry around each segment that is rasterized. Finally, the pixel shader computes the intersection between the algebraic surfaces and the current viewing ray, and shades the pixel accordingly. The method also provides consistent surface parametrization for texturing, as well as surface normals for plausible shadings.

We extend the conical frustum-based representation to include *relief mapping* to add fine-scale surface details. At the same time, the output-sensitive nature of ray casting largely decouples the rendering speed from object complexity, often resulting in higher frame rates compared to polygonal renderings of the same quality. We show examples of the technique applied to static and animated objects.

1.1. Related work

Several methods for rendering algebraic surfaces using GPU based ray casting have been published recently, among them piecewise polynomial surfaces using Bezier tetrahedra [LB06], radial basis functions defined over point sets [CBC*01] and implicit surfaces defined over discrete voxel grids [HSS*05]. Sigg et al. [SWBG06] present a GPU ray casting framework for rendering quadratic surfaces that is similar to our work; they consider spheres, ellipsoids and cylinders, whereas we consider conical frusta. However, unlike Sigg et al., we define a parametrization and map textures and relief maps onto our surfaces.

Mapping height fields (*relief maps*) onto objects for increased geometric detail has a long tradition in computer

Published by Blackwell Publishing, 9600 Garsington Road, Oxford OX4 2DQ, UK and 350 Main Street, Malden, MA 02148, USA.

graphics. In early work, Cook [Coo84] displaced the vertices of microtriangles. Later, Meyer and Neyret [MN98] achieved real time rendering of complex surface offsets with a slicing-based method. Lee et al. [LMH00] present a geometry-based framework that represents displaced surfaces as offsets from subdivision meshes.

In recent years, relief mapping has became more practical because of the advent of GPU ray casting [Tat06, BD06, POC05]. A fast but approximate method (i.e., object silhouettes remain without details) was presented by Policarpo [POC05] in 2005. This method calculates an entry and exit point in texture space and marches the ray until its *height* is smaller than the one stored in the relief map at that position. This work was later improved by locally approximating the object shape using quadrics [PO06, OP05]. A more accurate technique is to extrude a given object mesh along the normals, thus forming a prism for each triangle. All prisms together form an object *shell* [HEGD04, PBFJ05]. Typically, each prism is split into three tetrahedra that are rendered separately. This allows for better silhouette representations at the cost of increased overdraw. While simple, the tetrahedral mapping of the prisms leads to unwanted distortion in the displacements. Jeschke et al. [JMW07] present a high quality method that does not rely on tetrahedra, but the ray casting step is more complex, making the method slower.

This paper brings the ideas of ray casting algebraic surfaces and relief maps together. By defining a consistent parametrization for objects consisting of implicitly defined conical segments we can provide geometric details for algebraic surfaces. At the same time, common problems for relief mapped polygonal models are avoided. Because the conical segments are implicitly represented, we can provide high quality relief mapped surfaces (including object silhouettes) while avoiding the high overdraw of previous shell-based methods.

2. Skinning a Skeleton with Conical Frusta

This section describes our skinning procedure, which produces a set of conical frusta suitable for GPU ray casting. We assume as input a list of line segments that are connected with joints, along with a desired radius to define the thickness of the object at each joint. We begin by defining a conical frustum representation for object segments and provide definitions for their parametrization and surface normals. Then we describe a ray-frustum intersection routine and a tight bounding volume definition needed for an efficient GPU implementation. In Section 3 we extend our ray casting model to include relief mapping, which allows the algorithm to render complex surface details in real time.

2.1. Conical Frusta

A right circular cone centered around the y axis, as shown in Figure 1, can be defined as follows:

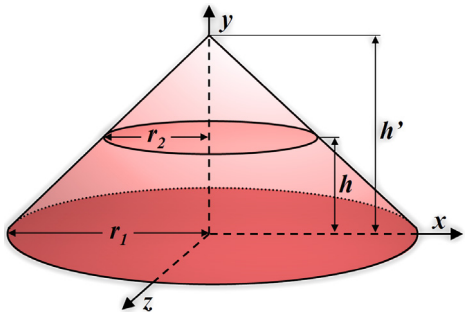

Figure 1: *Basic cone definition.*

$$(h')^2(x^2+z^2) = r_1^2(y-h')^2 \tag{1}$$

where h' is the cone height and r_1 is the radius at $y=0$. A conical frustum is defined by cutting off the cone at $y=0$ and $y=h$. Given the height of a conical frustum, h, and the radii at the two ends, r_1 and r_2, the height of the corresponding cone h' can be computed as

$$h' = \frac{r_1 h}{\|r_1 - r_2\|}. \tag{2}$$

2.2. Connecting Conical Frusta

To skin a complete skeleton using conical frusta, we need to seamlessly fit adjacent conical segments together. Our renderer handles this by defining a unique *cutting plane* for every joint, as shown in Figure 2. We start by defining a *medial plane*, which is formed by taking the cross product of the medial axes of the two participating segments:

$$\overrightarrow{N_m} = \frac{\overrightarrow{M_1} \times \overrightarrow{M_2}}{\|\overrightarrow{M_1} \times \overrightarrow{M_2}\|} \tag{3}$$

where $\overrightarrow{N_m}$ is the normal of the medial plane, and $\overrightarrow{M_1}$ and $\overrightarrow{M_2}$ are the normalized medial axis directions of the frusta (If the medial axes are colinear, we substitute one of the coordinate axes for $\overrightarrow{M_2}$). Within the medial plane, the left and

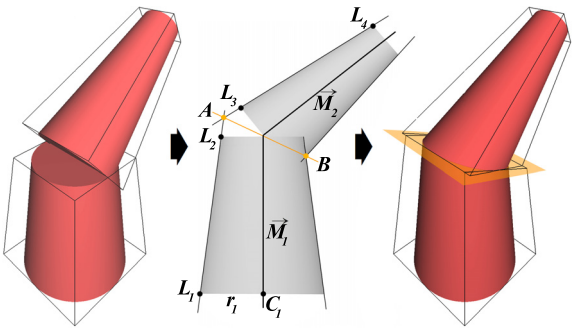

Figure 2: *A cutting plane is computed in the plane defined by the two segment's medial axes.*

right cone boundaries (shown in Figure 2, middle) intersect at two points A and B. The point A is the intersection of the lines that connect points L_1 to L_2 and L_3 to L_4, defined as offsets from the medial axes of the two frusta. For example, L_1 is given by

$$L_1 = C_1 - r_1(\overrightarrow{M_1} \times \overrightarrow{N_m}) \tag{4}$$

where C_1 is the base point of the bottom frustum. We perform the calculation to find A directly in the medial plane rather than in 3D. B is found similarly. Given A and B, we can construct the cutting plane between the two frusta to be perpendicular to the medial plane and include A and B:

$$\begin{aligned} \overrightarrow{N_c} &= (A-B) \times \overrightarrow{N_m} \\ \overrightarrow{N_c} \cdot (P-A) &= 0 \end{aligned} \tag{5}$$

$\overrightarrow{N_c}$ is the normal of the cutting plane, $\overrightarrow{N_m}$ is the normal of the medial plane and P is a point that may lie on the cutting plane.

Cutting a cone using the medial plane construction above results to an elliptical cross section on each of the cones. By construction, the major axes of the two elliptical cuts lie on the medial plane, and are identically defined by the segment AB. In general, however, the minor axes of these ellipses do not meet seamlessly. We propose the following iterative solution to this problem: if a new segment is connected to an existing one, we compute a cutting plane as described above and measure the misalignment for the minor axes of the ellipses formed by the cut. The base radius r_1 of the new cone is then adjusted accordingly and the cutting plane is recomputed. This is repeated until the minor axes match to within some tolerance. Once the major and minor axes coincide the elliptical caps will match perfectly since they have precisely the same major and minor axes. Note that this operation does not change the overall shape of the object since the bottom radius of the conical frusta remain constant. In practice we found this extra fitting step to be unnecessary since the seams between segments are generally not visible (for example, see Figure 4).

2.3. Frustum Parametrization

A (u,v) parametrization for a segment surface is required for texture mapping, normal mapping and relief mapping. We define u over the angle around the segment medial axis as shown in Figure 3. For a given point $P = (x_p, y_p, z_p)$, u is calculated by

$$u = \frac{1}{2\pi} \tan^{-1}\left(\frac{x_p}{z_p}\right). \tag{6}$$

Depending on their local coordinate systems, the u coordinates between two joined frusta may not match. In this case, we rotate the texture frame of the top frustum to bring the u parameters in line at the cutting plane. First, we calculate the $u = 0$ point on the cutting plane for the bottom frustum. Next, we calculate the u parameter for this same point as measured from the top frustum, u_t. During rendering, we subtract u_t from the u coordinate of the top frustum to align the u axes of the frusta.

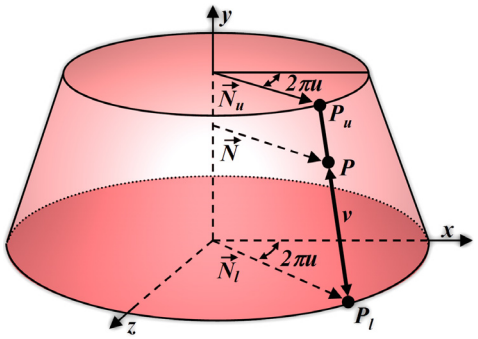

Figure 3: *Conical frustum parametrization: u is defined around the medial axis y, and v spans from the lower to the upper cutting plane.*

The v coordinate spans along the cone between the two cutting planes. To solve for v, we compute two points P_l and P_u on the two cutting planes at the same u coordinate as P (see Figure 3). These points can be computed as the intersection of the two cutting planes and the line through P and the cone apex $(0, h', 0)$. v is computed from the distance ratio between P, P_l and P_u:

$$v = \frac{\|P - P_l\|}{\|P_u - P_l\|}. \tag{7}$$

Although we define the parametrization from 0 to 1, it can be rescaled if a texture should span multiple segments. A twist in the parametrization can be handled by adding $v\theta$ to the u coordinate, where θ is the twist angle along the segment.

2.4. Surface Normals

Segment surface normals are needed for shading calculations and relief mapping. However, the normals defined by the conical frustum surface are not continuous across adjacent segments, which is desirable for smooth shading. To remedy this problem we define new surface normals that are C^0 continuous across segments as follows: Normals on the cutting planes between segments are defined to face away from the base and top point of the medial axis. Normals in the middle of a segment are interpolated using the v parameter (Equation 7). Referring to Figure 3, after computing P_l and P_u as described in Section 2.3, the lower and upper normal directions are computed as $\overrightarrow{N_l} = P_l - (0,0,0)$ and $\overrightarrow{N_u} = P_u - (0,h,0)$, respectively. After normalizing $\overrightarrow{N_l}$ and $\overrightarrow{N_u}$ we compute the final normal $\overrightarrow{N}$ by blending between the two based on v:

$$\overrightarrow{N} = (1-v)\overrightarrow{N_l} + v\overrightarrow{N_u}. \tag{8}$$

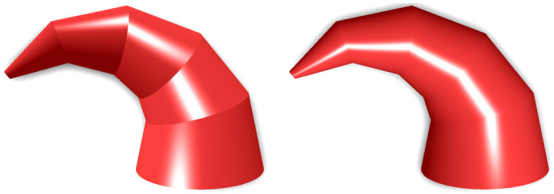

Figure 4: *Renderings with actual surface normals (left) and our interpolated normals (right).*

Even though normals defined in this way deviate quite a bit from the actual surface normals, they tend to work well for shading as well as relief mapping purposes. Figure 4 compares glossy renderings made with the actual surface normals to our interpolation scheme. Note that our normals provide a visually smooth surface.

2.5. Ray Casting Conical Frusta

The goal of the ray casting step is to find the intersections of a given viewing ray and a conical frustum. First, the viewing ray is transformed into the local *xyz* coordinate system of the segment as defined in Section 2.1. Let the viewing ray V be defined in the segment coordinate system as:

$$V(t) = E + t\vec{D} \tag{9}$$

with E being the eye point and D being the viewing direction. Inserting 9 in 1 and solving for t results to

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{10}$$

where

$$\begin{aligned} a &= \vec{D}_x^2 + \vec{D}_z^2 - \vec{D}_y^2 (\frac{r_1}{h'})^2 \\ b &= 2E_x\vec{D}_x + 2E_z\vec{D}_z - 2\vec{D}_y(E_y - h')(\frac{r_1}{h'})^2 \\ c &= E_x^2 + E_z^2 - (E_y - h')^2(\frac{r_1}{h'})^2. \end{aligned}$$

If $b^2 - 4ac$ falls below 0 or a becomes 0, the viewing ray does not hit the cone and the respective pixel can be discarded. Back substituting t into 9 provides two intersection points. We pick the one with the smallest t, i.e. the nearest to the eye point. The point is tested to see if it lies between the two cutting planes of the segment using a simple dot product. If the test fails the pixel is discarded. Otherwise we compute the respective texture coordinates and normals and render the point.

2.6. Bounding Primitives

To initiate ray casting, the renderer rasterizes a low polygon count bounding primitive for each conical frustum, in our case a pyramidal frustum. The side planes of the pyramid are defined by the points $(\pm r_1, 0, \pm r_1)$ and $(\pm r_2, h, \pm r_2)$. Each of the four points at $y = 0$ forms a line with its counterpart at $y = h$. These lines are intersected with the oblique cutting planes (see Section 2), which results in an obliquely cut pyramidal frustum that tightly covers the conical frustum as is shown in Figure 2, right. The renderer rasterizes this structure as 12 triangles, providing the pixel shader with all required fragments for ray casting.

3. Relief Mapping Conical Frusta

While the conical frusta define basic object shapes, the need for more detailed object surfaces calls for extending the proposed representation. Recently, *relief mapping* using GPU ray casting was presented for triangle meshes (see Section 1.1) and this technique fits nicely into the framework presented so far.

The first step is to define a shell-like volume around the cone where the height field is mapped to. The idea is to extend the representation by an additional conical frustum per segment, thus giving two concentric conics. Assuming the *inner* frustum is given, we add the desired shell thickness to the two radii r_1 and r_2 (see Section 2) to form the *outer* frustum. The relief texture is mapped onto each frustum using the parametrization described in Section 2.3. Note that in this way, any (u,v) coordinates are also exactly mapped along the normals as defined in Section 2.4, thus defining a class of concentric cones with equal parametrization. The *relief height* w is defined as $w = 0$ at the inner cone and being $w = 1$ at the outer one.

The principle of relief mapping is to march the viewing ray through this shell and calculate the closest intersection with the relief map. In each step the current ray height is compared against the respective relief height stored in a texture. First we compute the ray *entry* and *exit* points as the viewing ray intersections with the outer conical frustum (see Section 2.5). This provides the ray interval of interest. It is clipped against the cutting planes that bound the segment.

Ray intervals form *curves* in parameter space, similar to the triangle case [JMW07]. Consequently, to be accurate the (u,v) parametrization has to be computed separately for each sample point along the ray. First we have to compute the current ray height w which is the distance ratio from the current point P to the inner and outer cone. Fortunately, since only the ratio is of interest and both cone surfaces are parallel, the computation can be done in the plane $y = P_y$. The distance d between P and the medial axis is then $d = \sqrt{P_x^2 + P_z^2}$. The radius of the inner and outer cone at P_y are computed given the base radius r_1 and height h' for each as:

$$r = \frac{r_1(h' - P_y)}{h'}. \tag{11}$$

The current height w is then computed as

$$w = \frac{d - r_i}{r_o - r_i} \tag{12}$$

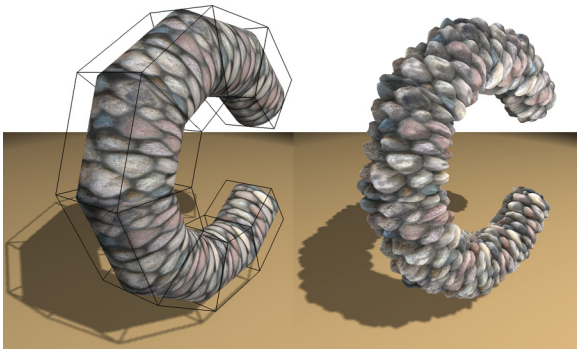

Figure 5: *Comparison between simple texture mapping (left) and relief mapping.*

where r_i and r_o are the radii of the inner and outer cones at height P_y. Comparing w against the height stored in the relief map requires texture coordinates for P. We compute these by constructing a new cone with P lying exactly on its surface by using the computed w value and calculating the (u,v) parametrization for that cone as described in Section 2.3. Finally, the height value is fetched from the relief map at position (u,v) and compared against w. If the latter is smaller, we have found an intersection and can shade the point. If no such w can be found, the pixel is discarded.

Note that r_i and r_o can be linearly interpolated between ray entry and exit point, which speeds up the computation. On the other hand, computing (u,v) as described above is exact but computationally expensive. Fortunately, since u is the angle around the y axis it can be computed quickly in the xz plane of the current sampling point. We approximate v by linear interpolation between the ray entry and exit point. This approximation provides exact results if the two cutting planes are parallel. If the angle between successive segments is less than 45 degrees, artifacts (relief "swimming") hardly become noticeable. Segments can be subdivided for larger angles. Figure 5 shows an example for a relief mapped object.

It is interesting to note that ray marching acceleration techniques like sphere or cone tracing were not efficient for the proposed relief mapped cones. We attribute this fact to the required dependent texture reads that only pay off if many uniform sampling steps can be omitted as in terrain rendering. Instead, we successfully used uniform sampling followed by a bisectional search for accuracy improvement, similar to Policarpo et al. [POC05].

4. Implementation and Results

We have implemented relief mapped conical frusta using C++ and GLSL. Our program computes the frustum bounding geometry (12 triangles per segment) on the CPU. During rasterization the fragment shader computes the ray conic intersection for each pixel, optionally with relief mapping. Illumination can then be computed using standard techniques.

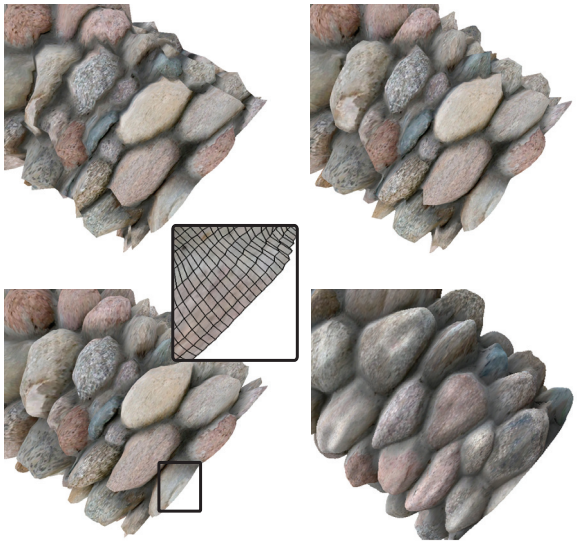

Figure 6: *Close-ups of the scene in Figure 5 with increasing number of triangles: Upper left 16k, upper right 64k, lower left 256k. The lower right shows a similar viewpoint of the same scene rendered with ray casting.*

To get an initial quality assessment between our algorithm and polygon rendering, we rendered several relief-mapped scenes with our ray caster and different amounts of polygonal subdivision. Figure 6 shows an example from our experiments. Notice that in this example, at least 256k polygons are necessary to achieve similar quality to ray casting.

We also modeled a number of plants and other objects to demonstrate the range of possible shapes that can be modeled with conical frusta. Figures 7 through 11 show several scenes that we composed. The conical frusta proved to be a flexible modeling tool. For example, complex structures such as the well in Figure 10 and palm tree leaves in Figure 11 were easy to create.

Table 1 shows model statistics and rendering performance for these scenes on an Intel Xeon running with 2.66GHz and an NVidia 8800 GTX graphics card. We measured the frame rate at 1024x768 pixel resolution with each model spanning most of the viewport. Frame rates for geometry subdivision are based on subdividing the model until similar quality to the ray casting results is achieved. As shown in the table, a few polygon subdivisions usually suffice to attain similar quality in the non-displaced case, but more subdivisions are required to equal the quality of ray casting for relief-mapped objects. (The leaves of the coconut and the pine trees are not relief mapped nor are all objects in the well scene. Also, the last four scenes in the table were rendered with shadow maps.)

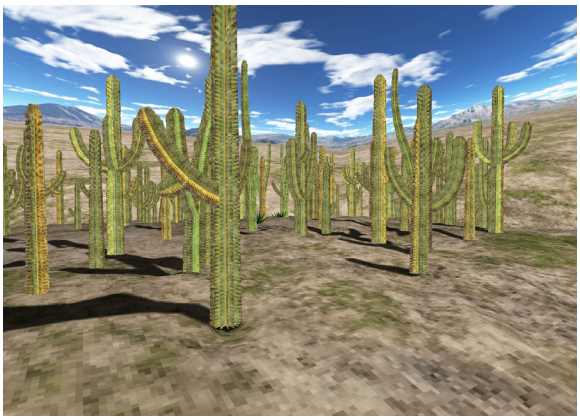

Figure 8: *Scene with 100 dancing cacti.*

Since the ray caster is fill limited and polygon rendering is transformation limited, the trade-off between polygons and ray tracing depends highly on the number of rendered pixels. However, the frame rate is not always exactly correlated with the number of pixels rendered. For example, we have obsered that for full screen close-up views the frame rate may actually be higher than for overviews since ray casting texture reads become more local and therefore more cache friendly. In general, we only see a performance benefit in the case of relief mapping, since the GPU is optimized for very high polygon loads. Polygon rendering could be improved by employing LOD techniques, but the ray caster can maintain high quality at interactive frame rates without the need to resort to LOD algorithms that require more memory and programming effort.

The conical frustum representation is also quite compact. The bulk of the memory overhead lies in the relief map, which can be reasonably stored as 8 bit values, whereas the vertices for polygons at the same sampling density will take twelve times as much space without considering connectivity. Furthermore, a relief map is more amenable to reuse within a scene than a list of vertices, so the actual memory savings may be even higher in some cases.

For animated objects, the splitting planes between segments have to be recomputed in each time step. This can be done efficiently even for large scenes because of the sparse representation. Figure 8 shows 100 individually animated cacti that are rendered interactively. We maintain interactivity even though the splitting planes are computed on the CPU. This operation could be moved to the vertex shader for added speed.

4.1. Limitations

Our representation has a number of limitations. First, objects are by definition composed of piece-wise linear segments. This makes the representation of models that are smooth in the medial-axis direction inefficient. While the presented lighting model and the relief mapping can alleviate this problem, an interesting direction of future research is to provide C^1 continuity between adjacent segments. If the angle between two segments is too large, the top and bottom caps may intersect. Currently we do not handle this situation. Branching structures are another issue, since they cannot be represented seamlessly. It would be interesting to explore using the relief map to provide continuity in these cases. Another limitation is that the cones defining our objects must be round. Nevertheless, the relief map can be used to increase the shape variety. Allowing other implicit surfaces like ellipsoids may further widen the range of representable objects.

5. Conclusion

We have presented an approach to skin skeletal objects using C^0 continuous conical frusta and render them directly on the GPU using ray casting. This provides minimal workload for the CPU and reduces communication between CPU and GPU as well as low memory requirements. We developed a parametrization for the frusta needed for texture mapping and defined consistent surface normals. The representation was extended to include relief maps. Here the implicit surface definition allows efficient high quality renderings of static and animated objects. In the future we want to develop techniques that increase the range of objects that can be represented using conical frusta.

Acknowledgements. We would like to acknowledge financial support from NSF and NGA.

References

[BD06] BABOUD L., DÉCORET X.: Rendering geometry with relief textures. In *Graphics Interface '06* (2006), pp. 195–201.

[CBC*01] CARR J. C., BEATSON R. K., CHERRIE J. B., MITCHELL T. J., FRIGHT W. R., MCCALLUM B. C., EVANS T. R.: Reconstruction and representation of 3d objects with radial basis functions. In *Proc. of SIGGRAPH '01* (2001), pp. 67–76.

[Coo84] COOK R. L.: Shade trees. In *Proc. of SIGGRAPH '84* (1984), pp. 223–231.

[HEGD04] HIRCHE J., EHLERT A., GUTHE S., DOGGETT M.: Hardware accelerated per-pixel displacement mapping. In *Graphics interface '04* (2004), pp. 153–158.

[HSS*05] HADWIGER M., SIGG C., SCHARSACH H., BÍLUHLER K., GROSS M.: Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum 24*, 3 (2005), 303–312.

[JMW07] JESCHKE S., MANTLER S., WIMMER M.: Interactive smooth and curved shell mapping. In *Rendering Techniques 2007* (6 2007), pp. 351–360.

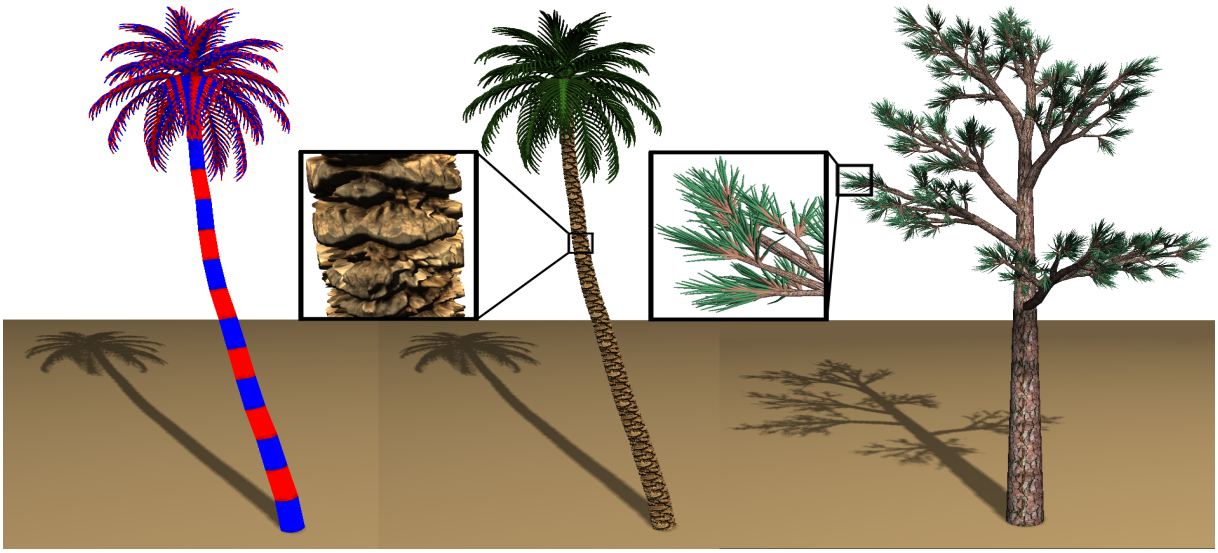

Figure 7: *Coconut tree consisting of 3713 segments (left) rendered as relief mapped cones (middle). Right: Pine tree consisting of 16997 segments.*

Model	# segments	RC (fps)	RC+R (fps)	GS (fps/sub)	GS+R (fps/sub)
Coconut tree	3713	83	71	240 / 1	63 / 2
Pine tree	16997	34	16	55 / 1	15 / 2
Elm tree	1833	193	20	375 / 1	102 / 2
Aloe Vera	229	171	20	820 / 2	53 / 4
Saguaro	30	305	105	830 / 2	54 / 4
Well	8	303	105	296 / 2	21 / 4
Desert scene	799	421	151	1100 / 0	79 / 2
Oasis scene	18557	46	31	174 / 0	13 / 2
Dancing Cacti	2158	35	33	170 / 0	12 / 2

Table 1: *Statistics for the test models. "fps" denotes frames per second and "sub" the number of polygon subdivisions. "RC" indicates ray casting, "GS" indicates geometric subdivision, and "+R" means that relief mapping was applied. Before subdivision the renderer rasterizes 16 triangles for each conical segment, so the number of triangles rendered for a given subdivision level is* $16n \times 4^m$ *where n is the number of segments and m is the subdivision level.*

[LB06] LOOP C., BLINN J.: Real-time gpu rendering of piecewise algebraic surfaces. *ACM Trans. Graph. 25*, 3 (2006), 664–670.

[LMH00] LEE A., MORETON H., HOPPE H.: Displaced subdivision surfaces. In *Proc. of SIGGRAPH '00* (2000), pp. 85–94.

[Max90] MAX N.: Cone-spheres. *Proc. of SIGGRAPH '90 24*, 4 (1990), 59–62.

[MN98] MEYER A., NEYRET F.: Interactive volumetric textures. In *Rendering Techniques (EGSR)* (Jul 1998), pp. 157–168.

[OP05] OLIVEIRA M. M., POLICARPO F.: *An efficient representation for surface details.* Tech. rep., 2005.

[PBFJ05] PORUMBESCU S. D., BUDGE B., FENG L., JOY K. I.: Shell maps. *ACM Trans. Graph. 24*, 3 (2005), 626–633.

[PO06] POLICARPO F., OLIVEIRA M. M.: Relief mapping of non-height-field surface details. In *I3D '06* (2006), pp. 55–62.

[POC05] POLICARPO F., OLIVEIRA M. M., COMBA J. L. D.: Real-time relief mapping on arbitrary polygonal surfaces. In *I3D '05* (2005), pp. 155–162.

[SWBG06] SIGG C., WEYRICH T., BOTSCH M., GROSS M.: Gpu-based ray-casting of quadratic surfaces. In *Symposium on Point Based Graphics* (2006), pp. 59–65.

[Tat06] TATARCHUK N.: Dynamic parallax occlusion mapping with approximate soft shadows. In *I3D '06* (2006), pp. 63–69.

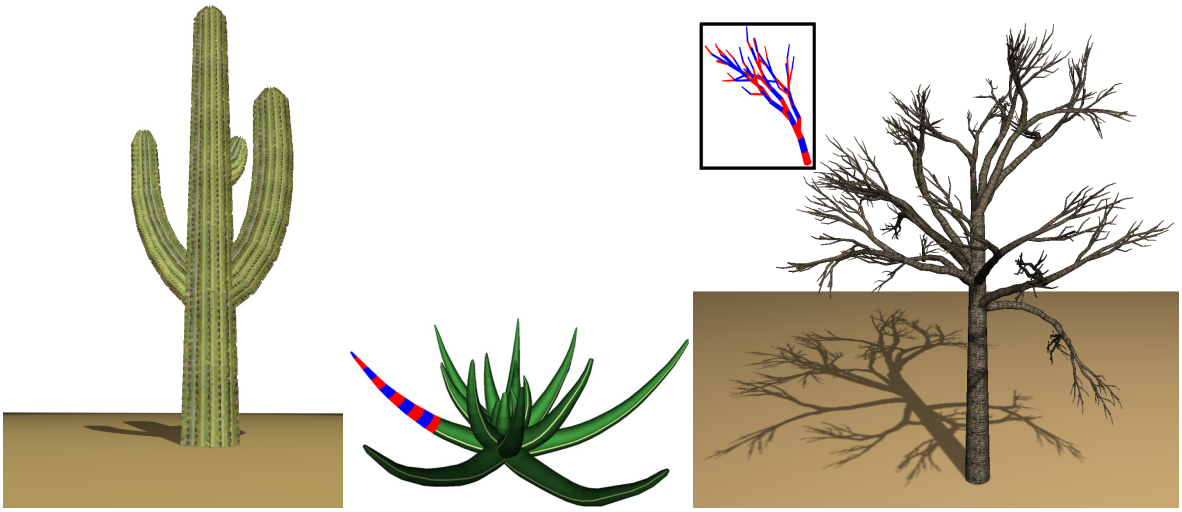

Figure 9: *Left: Saguaro cactus defined by 30 segments. Middle: Aloe vera modeled using 229 segments. Right: Elm tree modeled as 1833 segments.*

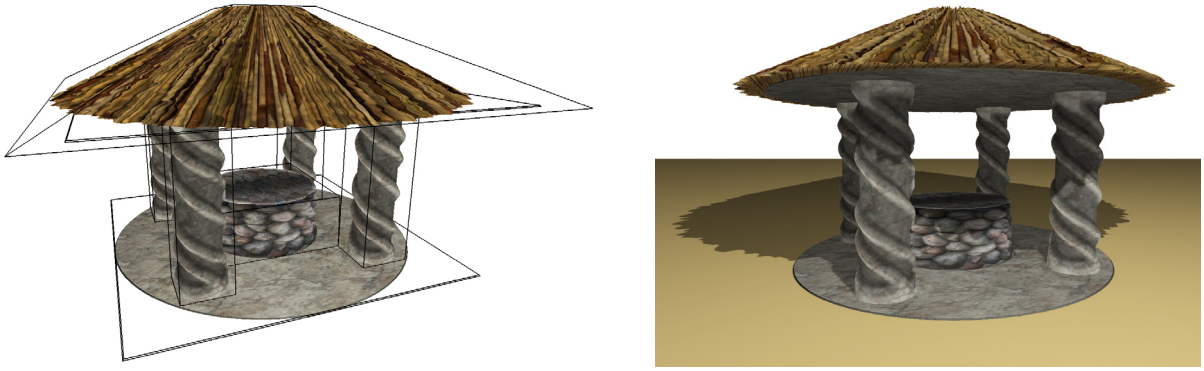

Figure 10: *Left: Well scene modeled with only 8 conical segments. The well, the columns and roof are relief mapped. Right: a second view.*

Figure 11: *Left: Oasis scene consisting of 18557 segments, most of them modeling the leaves. Right: Desert scene consisting of 799 conical segments.*