EUROGRAPHICS 2003 / P. Brunet and D. Fellner
(Guest Editors)

Volume 22 (2003), Number 3

Shadow Volumes on Programmable Graphics Hardware

Stefan Brabec and Hans-Peter Seidel

MPI Informatik, Saarbrücken, Germany

Abstract

One of the best choices for fast, high quality shadows is the shadow volume algorithm. However, for real time applications the extraction of silhouette edges can significantly burden the CPU, especially with highly tessellated input geometry or when complex geometry shaders are applied.

In this paper we show how this last, expensive part of the shadow volume method can be implemented on programmable graphics hardware. This way, the originally hybrid shadow volumes algorithm can now be reformulated as a purely hardware-accelerated approach.

The benefits of this implementation is not only the increase in speed. Firstly, all computations now run on the same hardware resulting in consistent precision within all steps of the algorithm. Secondly, programmable vertex transformations are no longer problematic when applied to shadow casting objects.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture; I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

1. Introduction

Crow's shadow volumes [5] is one of the most popular algorithms for shadow generation. Especially for real time application it is the de-facto standard way for precise, high quality shadows. This is due to the fact that shadow information is generated in object space, meaning that shadow information is available for every window-space pixel. Achieving information that precise is hardly possible with a sampling based method such as shadow maps [13].

But this accurate shadow information does not come for free. Generating the necessary silhouette information can put a heavy load on the CPU, and rendering the extruded shadow volumes on graphics hardware can easily exhaust fill rate capabilities.

Another problem is the hybrid nature of this algorithm. In order to produce accurate shadow volumes both the CPU and the graphics hardware have to be synchronized. This not only refers to the data transfer, but also, most importantly, a consistent numerical precision during all calculations. Nowadays this *perfect* synchronization becomes even more important. Recent graphics hardware exposes powerful programming features that allow nearly arbitrary operations on both vertex [9] and pixel data. When using these programmable features in conjunction with shadow volumes the silhouette extraction performed on the CPU becomes problematic: All vertex transformations computed on the graphics hardware, which are relevant for shadow casting objects, need to be simulated on the CPU in order to achieve the same results. This is not only a very time consuming task (e.g. imagine a procedural displacement shader applied to a highly tessellated object) but also makes programming with shadow volumes a nightmare, since every shadow object needs to be handled differently.

In this paper we address these issues and present a method for implementing the whole algorithm on the graphics hardware. Migrating the silhouette extraction to graphics hardware solves a number of issues:

- All calculations are performed on the same hardware, resulting in consistent precision.
- Shadowing objects can be used with programmable vertex processing in the same manner as with fixed transformation processing.
- Applications gain more CPU time, since resources which were formerly dedicated to silhouette extraction are released.
- Rendering with shadow volumes no longer needs to be

synchronized with CPU and graphics hardware, minimizing potential idle time on both processing units.
- Silhouette extraction is performed on large chunks of data in parallel (bulk processing), which results in enormous speed up.
- Using the shadow volume approach in an application becomes very simple since only trivial, local preprocessing of the objects needs to be performed. This is especially important for *general* scene graphs, where silhouette extraction would require traversal of all possible transformation and deformation nodes.

The remaining part of the paper is structured as follows. First we briefly review relevant work on shadow algorithms. Then, in Section 3 we explain stencil shadow volumes, which is the basis for our implementation, in more detail. The migration of silhouette extraction from CPU to programmable graphics hardware is presented in Section 4. In Section 5 we discuss some of the implementation details. Section 6 shows some example scenes rendered with our hardware-based implementation.

2. Previous Work

Since there are a huge number of publications dealing with shadow generation, we only briefly review those relevant for our implementation. For more information we recommend Woo's survey on shadow algorithms [14] as a starting point.

In 1977, Franklin C. Crow presented the shadow volume algorithm [5]. In his paper he describes how shadowed regions can be identified by extruding silhouette edges (with respect to the light source) to form semi-infinite volumes. A simple *point-inside-volume* test can then be used to check whether a given surface point is in shadow or lit. Bergeron [1] presented a general version of Crow's algorithm which also is capable of handling non-closed objects as well as non-planar polygons.

A hardware-based approach computing the *point-in-volume* test using the stencil buffer was presented by Heidmann [7]. His approach will be explained in detail in Section 3.

Recently, an alternative to stencil-based counting using alpha blending was proposed by Roettger et. al [12]. They replace increment and decrement operations by multiplications, which under certain conditions is even faster than using the stencil buffer. The main benefit of their method is that it works for graphics cards that do not support hardware-accelerated stencil testing.

Everitt et. al [6] came up with a bullet-proof implementation of stencil based shadow volumes for hardware-accelerated rendering. They solve the problem of non-closed shadow volumes due to near/far plane clipping by inverting the stencil count scheme, an unpublished idea by Carmack [2], and moving the far plane to infinity. This way, shadow volumes are always closed, and artifact-free shadows can be generated. The tutorial by Lengyel [8] provides some more implementation details and shows how fill rate problems can be reduced by using attenuated light sources.

Also the problem of optimizing the expensive step of detecting silhouette edges, both in the context of shadow volumes as well as for other application like non-photorealistic rendering (NPR), has led to a number of algorithms.

Goodrich [11] explained how the *dual space* of a primal space can be used to detect silhouette edges. In dual space, the light position becomes a plane, whereas edges stay edges. Those edges that intersect the plane are detected silhouette edges. Although this method sounds complicated at a first sight, it can greatly take advantage of temporal coherence, e.g. for moving light sources.

A spatial data structure for shadow volumes was introduced by Chin et. al [3]. They modified the well-known *binary space partitioning* (BSP) scheme, so that for every light source a BSP tree is generated that represents the shadow volume caused by the polygons facing towards the light. The algorithm was further improved by Chrysanthou et. al [4] to handle dynamic scenes as well.

A hardware-accelerated silhouette detection method was presented by McCool [10]. Here shadow edges are extracted by using a depth map and an edge detection filter. With this method an optimal shadow volume (non intersecting volumes) can be generated. However, the quality of the generated shadows may suffer from sampling artifacts due to the limited depth map resolution.

As graphics hardware becomes faster and faster, especially in terms of fill rate and vertex processing, a simple brute force approach is becoming popular again. Instead of detecting silhouette edges, one can also consider individual triangles as shadow casting objects and generate a shadow volume for each. This method is simple to implement but is only useful for very few, coarsely tessellated objects.

3. Stencil Shadow Volumes

Heidmann's stencil shadow volumes algorithm [7] starts with the detection of possible silhouette edges. For simplicity, we assume that all shadow casting objects are closed triangular meshes (2-manifold) for which connectivity information is available.

To test whether a given edge is a silhouette edge we check if the edge connects a front- and a back-facing triangle, with respect to the light source. This is illustrated in Figure 1. Triangle orientation can easily be checked by taking the dot product of the face normal and the vector to the light source. If this dot product is negative, a triangle is back-facing with respect to the light, otherwise it is front-facing. Repeating this for all edges, we obtain a set of silhouette edges that form closed loops.

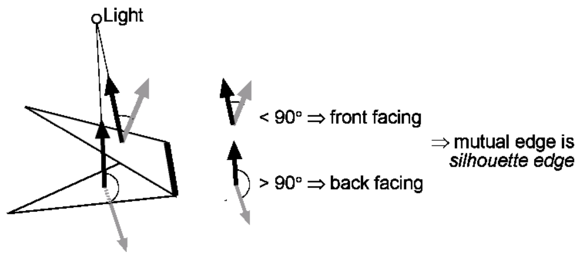


Figure 1: *Silhouette edge detection.*

Next we extrude these silhouette loops to form semi-infinite volumes. For each silhouette edge a quadrilateral is constructed by taking the two original vertices of the edge and two vertices which are computed by moving the original vertices far away to infinity along the ray originating from the light source through the vertex.

Together with the object's front facing triangles, these quadrilaterals bound all regions in space which are in shadow. In order to check if a given point is in shadow all we have to do is determine if the point lies outside of all shadow volumes.

This information can be easily obtained by following a ray from the viewer to the surface point and counting how many times we enter or leave a shadow volume boundary polygon. This counting scheme is illustrated in Figure 2. Here shadow

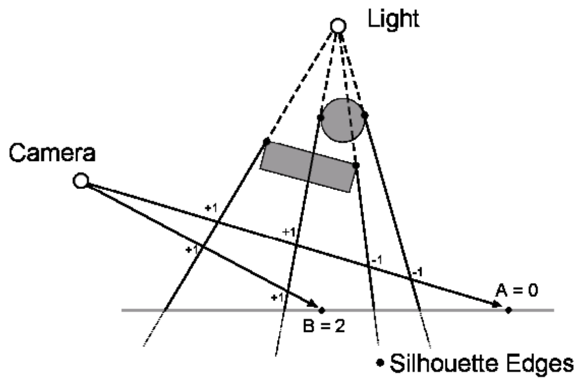


Figure 2: *Inside-outside test.*

volumes have been generated for a sphere and a box illuminated by a point light source. While following the ray from the viewer to surface point *A*, we count how many times we enter (increment) and leave (decrement) a shadow boundary. The final counter value of 0 indicates that the surface point is lit by the light source, since we have left the shadow regions as many times as we entered them. Counting shadow boundaries for surface point *B* yields a value of 2, since the point is inside two shadow volumes (sphere and box).

Implementing this test using ray tracing would be a very time consuming task. Heidmann showed that this simple in-out counting can be performed on graphics hardware using the stencil buffer. First, the stencil buffer is initialized to zero (all pixels lit). Next, the whole seen is drawn as seen by the camera. In this step only depth information is relevant so color channels and all lighting and shading computations can be disabled.

The actual counting can then be achieved by disabling depth buffer writes and rendering all shadow volume quadrilaterals. In this step the stencil operation is setup in such a way that front-facing quadrilaterals (with respect to the viewer) increment the stencil value at the window space position for all pixels that pass the depth test. Similarly, all pixels that pass the depth test and belong to a back-facing quadrilateral will decrement the stencil value. On modern graphics hardware this can be implemented in a single rendering pass (two-sided stencil testing) whereas on older hardware separate passes for front- and back-facing quadrilaterals are needed (single stencil operation). Changing the counter value based on the front- and back-facing information requires a consistent winding order when constructing the quadrilaterals. This can for instance be achieved by sticking to the vertex order of the front-facing triangle (with respect to the light source) adjacent to the silhouette edge.

In a final step, the scene is rendered once again, this time with lighting and shading turned on. During rendering we set up the stencil test such that only those pixels whose corresponding stencil value is zero will pass through.

Although the basic method is simple to implement, there are several problems, e.g. due to near plane clipping or counter overflows. These problems and possible solutions are described in detail in [6].

4. Shadow Volumes on Programmable Graphics Hardware

4.1. Motivation

Recalling the different steps in Section 3 it becomes clear that silhouette detection and shadow volume extrusion are the only steps that still have to be performed on the CPU. This can not only become a bottleneck if shadow casting objects are highly tessellated, but is indeed problematic if the input geometry will be deformed by the graphics hardware. Current state-of-the-art graphics boards provide powerful, programmable vertex processing units (vertex programs) [9] which can be used for nearly arbitrary geometric transformations, e.g. displacement mapping or matrix palette skinning. Using vertex programs in conjunction with shadow volumes requires the CPU to emulate all vertex processing in the same way as it is actually done by the graphics hardware.

As a consequence, detecting silhouette edges is no longer a trivial and fast operation since vertices and face normals have to be re-calculated at every frame in the worst case.

Also numerical differences can lead to strange artifacts, e.g. light leaks.

This is due to the fact that numerical operations are not guaranteed to yield the same result on both processing units (the CPU and the graphics hardware). As an example, the result of calculating $sqrt(x)$ can differ significantly since CPU and graphics hardware may use different approximations.

Another bottleneck when using the common hybrid approach is that CPU and graphics hardware need to be synchronized such that all shadow volumes are generated when the graphics hardware is ready to render them. Especially in applications like games the CPU is more and more dedicated to handle input events, artificial intelligence or sound, and all graphic-related work should ideally be done by the graphics hardware. Therefore keeping these two processing units asynchronous reduces potential idle time on both.

In the following sections we will show how silhouette detection and shadow volume generation can be implemented on programmable graphics hardware, which solves the described problems.

4.2. Silhouette Detection

In the first step of our hardware implementation we need to bring the actual geometry and the light sources into a common coordinate system. We will choose to transform both to world space, which is view-dependent and also, with respect to the scene geometry, reusable for different light sources.

Since all graphics cards perform a combined transformation which also includes the viewing transformation, we begin by setting the viewing matrix to identity. This way every vertex is transformed to world space. Secondly, every vertex in the scene is assigned a unique index number, so that it can later be referenced by its index. For objects which are referenced multiple times, e.g. an object placed in the scene at different locations, we need to ensure that vertices with different transformations also obtain different indices.

With these vertex indices we are now able to dump the world space coordinates of each vertex to the graphics hardware. The result of this step is a texture, in which each texel (x,y) stores the world space positions of one vertex. The mapping of a vertex to its position in the texture is defined by the vertex index. To retain as much precision for the vertex positions as possible, we use a 4-channel (RGBA) offscreen buffer with floating point precision as an output buffer.

We will now explain how this step can be implemented with the help of a vertex program: Instead of rendering filled primitives, like triangles or quadrilaterals, only the vertex itself is rendered as a point. We use the vertex program to compute the position (x,y) in the output buffer from the vertex index (passed along as a vertex attribute) and specify the result as the output position for the vertex. This way each vertex gets rendered as a single pixel at the position (x,y). The final task now is to set the color at position (x,y) to the corresponding vertex's world space coordinates. Since the vertex color output of a vertex program gets clamped to $[0\dots1]$, we output this value using one of the unclamped output registers, e.g. one of the 4D texture coordinates, and then map it to the color register in a fragment program. Note that the vertex's world coordinates include all vertex transformations (i.e. modeling or procedural transformations). This step is graphically explained in Figure 3 (Step 1).

The next task in our algorithm is the classification of possible silhouette edges. Assuming that all meshes used in the scene are well-modeled (2-manifold), meaning that there are no open edges and every edge connects exactly two triangles, the silhouette test only needs four vertices and the light source position(s) as input data. Two vertices are used to locate the edge itself whereas the remaining two are used to construct the two triangles that meet at the given edge.

Recalling Section 3, the silhouette test consists of checking the front-/back-face condition of the two triangles with respect to the given light source.

Our implementation is therefore straight forward: Given the connectivity information (edges) for all meshes we also assign all edges an unique identifier (index), used for later referencing. Since connectivity and index numbers remain constant, this can be implemented as a pre-processing step.

To detect silhouette edges, we use a brute-force approach that tests every edge in every frame during runtime (except for simple cases, where scene and light sources remain constant). Doing this on the host processor can be quite expensive, but on the graphics card, which can be seen like a SIMD-like (single instruction, multiple data) processor, this is an efficient operation running in parallel.

Like in the world space transformation step, we render all edges as single points. As before, these points represent the index number of a given edge, and have no real geometric meaning. Along with the index number describing the position where to store the result of the edge computation, we also pass all relevant input data for the given edge as additional per-vertex (point) attributes. As stated before, this input data consists of a total of four indices referring to the points that make up the two adjacent triangles. Since the light source position remains constant for all edges, this parameter is set globally.

Testing if a given edge is a silhouette edge is now trivial: We bind the dumped vertex positions as a 4-component floating point texture map and use the four indices to get the world space coordinates of all points. Since texture lookups are only possible in the fragment (pixel) processing step, this has to be implemented as a so called fragment shader (also referred to as pixel shader).

For both triangles that meet at the given edge we calculate the plane equations and compute the signed distance to the light source position. If the signs of the two distances differ, the edge is marked as a silhouette edge. Since the edge's vertex ordering has to be preserved for later steps, we also compute a flag indicating whether the vertex ordering of the front facing triangle corresponds to the order of the edge's vertices.

The result of the silhouette detection, which are two binary flags, are then written to the frame buffer as a color-coded value at the edge-index position.

4.3. Generation of Shadow Volumes

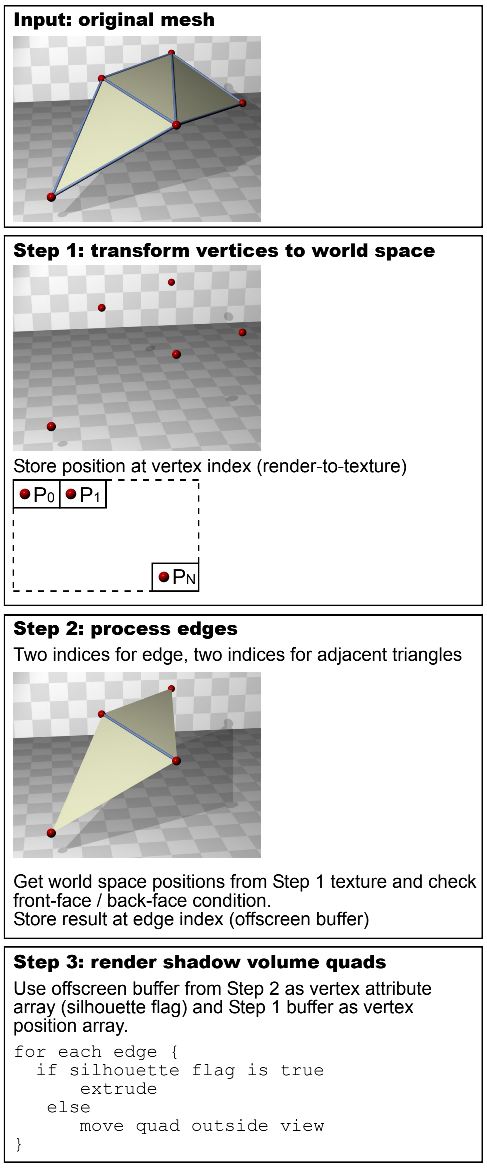
Generating and rendering the shadow volumes using the results of the previous steps is now straight forward.

In a pre-processing step we generate quadrilaterals for all edges, but instead of using the object's vertex coordinates and transformations, each vertex has a total of three indices and one flag:

- Two indices referring to the world space position of the edge's two end points.
- One index referring to the silhouette flag and the vertex-ordering for the given edge.
- A flag (yes/no) indicating whether the vertex should lie on the edge or should be extruded to infinity. Each quadrilateral has two points on the edge and two points that have to be moved to infinity.

Since we only want to render quadrilaterals for silhouette edges, the silhouette flag is used as a trivial reject. If the edge is not a silhouette edge, we move all vertices outside the viewing frustum, e.g. behind the viewer, so that the complete primitive is clipped away. For silhouette edges we either directly output the world space position, or, if the extrusion flag is true, we move the vertex to infinity with respect to the light source direction. Choosing one of the edge's vertices is based on the vertex-ordering flag, which preserves a consistent winding order.

All these steps are implemented as a vertex program and therefore are fully hardware-accelerated. The generated shadow volumes are similar to those generated on the CPU and can now be used for the stencil-based counting scheme explained in Section 3. Figure 3 illustrates the different steps of the hardware-based shadow volumes algorithm.


Input: original mesh

Step 1: transform vertices to world space

Store position at vertex index (render-to-texture)

Step 2: process edges

Two indices for edge, two indices for adjacent triangles

Get world space positions from Step 1 texture and check front-face / back-face condition.
Store result at edge index (offscreen buffer)

Step 3: render shadow volume quads

Use offscreen buffer from Step 2 as vertex attribute array (silhouette flag) and Step 1 buffer as vertex position array.

```
for each edge {
  if silhouette flag is true
      extrude
   else
      move quad outside view
}
```

Figure 3: *Workflow for hardware-based shadow volumes.*

5. Implementation

We implemented the described algorithm on an ATI Radeon 9700 card using OpenGL. This card supports all the programmable feature, like vertex and fragment programs with floating point precision, as well as floating point offscreen buffers and textures.

For the first step, we use a floating point RGBA offscreen buffer and modify all of the scene's shaders (vertex programs) such that instead of using the original vertex position the index number is used to calculate (x,y) pixel coordinates and the world space position is written out to the frame buffer. Currently this is a manual step but can simply be automated by a script that generates the modified shaders. Since only the world space position is relevant during this step, we can further optimize it by analysing which computations inside the shader affect the position and remove all other operations that are only relevant for e.g. the output color or texture coordinates.

For computing the silhouette and vertex-ordering flags, an offscreen buffer with less precision (e.g. RGB with eight bits per channel) is sufficient. By rendering all edges as points and using the offscreen buffer of the previous step as a 2D texture map, we can obtain the four world space position by sampling this texture at the exact integer positions (vertex indices). The light source position is specified as a global parameter. The fragment shader then computes the plane equation and tests for the front-/back-facing condition and the vertex ordering flag. In the previous sections we only discussed the silhouette detection for one light source. However, the fragment shader can easily compute the flags for several light sources simultaneously. The number of light sources that can be checked during this step is only limited by the maximum instruction length of the fragment shader and the number of bits available in the offscreen buffer. Since the results for one light source needs two bits, a standard RGB buffer can store the results for up to 12 light sources.

Rendering the shadow volume quadrilaterals requires the results of the two previous steps as input. Since the vertex extrusion and the trivial reject has to be performed before rasterization, this has to be implemented as a vertex program. Unfortunately, there is currently no fast path to access the required data in the vertex program directly. For our algorithm we therefore would propose one of the following features:

- Texture access during vertex processing. First attempts in this direction are made with the release of DirectX 9's displacement mapping, but a more general lookup would be necessary for our method.
- A fast, on the card copy from frame buffer to a vertex attribute array. This could be implemented as a simple copy operation, or ideally as *copy-by-reference* similar to the *render-to-texture* functionality.

Due to the lack of the proposed features, we are forced to use a very slow mechanism that transfers the data from the two buffers to host (CPU) memory and immediately downloads the same data as vertex attribute data for our current implementation.

6. Results

Figure 4 shows two example scenes with shadow volumes generated using our hardware approach. For both scenes, three light sources are used and silhouette edges are detected for all lights simultaneously. In Figure 4 (a) the vertex texture has a size of 128×128 pixels, needed to store the world space positions of the 9326 vertices. The edge buffer has a size of 256×128 which is enough to store the silhouette flags for the 27627 edges. Figure 4 (b) has a vertex texture of size 64×64 pixels (3298 vertices) and an edge buffer of size 128×128. The scenes were rendered at a window resolution of 512×512 on an AMD Athlon 1GHz machine equipped with an ATI Radeon 9700 card. For both scenes we obtain frame rates about 20 fps. The main bottleneck here are the frame buffer read backs. With all steps running on the hardware (as proposed in Section 5) we expect our method to run considerably faster. Our current implementation should therefore be seen as a proof-of-concept.

Figure 5 (left) shows a more complex example illuminated by three light sources. Here the geometry of each of the three spheres is displaced by a procedural noise shader, implemented as a vertex program. Detecting silhouette edges for this scene on the CPU would be very difficult since the vertex program would need to be evaluated on the CPU in order to obtain world space coordinates. Detecting silhouette edges with our hardware method is as simple as for the previous scenes. Only small modifications to the noise shader were necessary which ensure that for each vertex the world space position is passed as a result and the index becomes the vertex's pixel position. Here the vertex texture has a resolution of 64×32 (1638 vertices) and the edge buffer has a resolution of 128×64 (4896 edges).

Since the procedural noise shader only computes new vertex positions, the vertex normals no longer correspond to the actual geometry. Therefore the shading of the three objects looks unrealistic. To avoid strange artifacts we also disabled self and global shadowing for the three objects. With proper shading normals there would be a smooth intensity transition into the shadow region.

Figure 5 (right) shows the silhouette edges detected for one light source (yellow) as well as the generated shadow volumes for all three lights (red).

7. Conclusions and Future Work

In this paper we have shown how to perform the silhouette detection step of the shadow volume algorithm in hardware. The benefits of this approach are not only the gain in speed, what is most important is that shadow volumes can now easily be generated for geometry that is transformed by programmable vertex engines, as shown in the procedural noise example. A drawback of the algorithm is that it processes all edges and rejects non-silhouette edges at the extrusion stage, which is the very last step of the algorithm. This may put a very high load on the GPU but on the other hand, this is in most cases still faster than processing edges using the CPU.

The algorithm itself relies on capabilities available on recent graphics cards: programmable vertex and fragment units, floating point buffers, as well as floating point textures.

An important feature which is currently missing is a fast way to use the contents of a buffer as input for a vertex program, needed when rendering the shadow volumes. We are confident that future drivers will provide a more general memory management functionality. First efforts in this direction are already visible with the upcoming OpenGL 2.0 specification.

In this paper we did not address the problem of shadow volumes that intersect the near or far clipping plane. A solution to this was presented by Everitt et al. [6]. As future work we would like to investigate how those special cases can be detected and efficiently processed using our algorithm.

Another possible application for the silhouette detection presented here is in the context of non-photorealistic rendering (NPR). Here the silhouette information could be used to achieve toon-like or pencil drawn shading effects on a per triangle basis, rather than using image-based techniques.

References

1. P. Bergeron. A general version of crow's shadow volumes. *IEEE Computer Graphics and Applications*, 6(9):17–28, 1986.
2. John Carmack. John carmack on shadow volumes. Available from developer.nvidia.com, May 2000.
3. Norman Chin and Steven Feiner. Near real-time shadow generation using bsp trees. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, volume 23, pages 99–106, July 1989.
4. Yiorgos Chrysanthou and Mel Slater. Shadow volume bsp trees for computation of shadows in dynamic scenes. *1995 Symposium on Interactive 3D Graphics*, pages 45–50, April 1995. ISBN 0-89791-736-7.
5. Franklin C. Crow. Shadow algorithms for computer graphics. In *Computer Graphics (SIGGRAPH '77 Proceedings)*, pages 242–248, July 1977.
6. Cass Everitt and Mark J. Kilgard. Practical and robust stenciled shadow volumes for hardware-accelerated rendering. Technical report, NVIDIA Cooperation, March 2002. Published online at developer.nvidia.com.
7. T. Heidmann. Real shadows real time. *IRIS Universe*, 18:28–31, November 1991.
8. Eric Lengyel. The mechanics of robust stencil shadows. Tutorial available on www.gamasutra.com, Oct 2002.
9. Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 149–158. ACM Press / ACM SIGGRAPH, August 2001. ISBN 1-58113-292-1.
10. Michael D. McCool. Shadow volume reconstruction from depth maps. *ACM Transactions on Graphics*, 19(1):1–26, January 2000.
11. Mihai Pop, Christian Duncan, Gill Barequet, Michael Goodrich, Wenjing Huang, and Subodh Kumar. Efficient perspective-accurate silhouette computation and applications. In *Proceedings of the seventeenth annual symposium on Computational geometry*, pages 60–68. ACM Press, 2001.
12. Stefan Roettger, Alexander Irion, and Thomas Ertl. Shadow volumes revisited. In V. Skala, editor, *Proc. WSCG '02*, pages 373–379, 2002.
13. Lance Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 270–274, August 1978.
14. Andrew Woo, Pierre Poulin, and Alain Fournier. A survey of shadow algorithms. *IEEE Computer Graphics & Applications*, 10(6):13–32, November 1990.

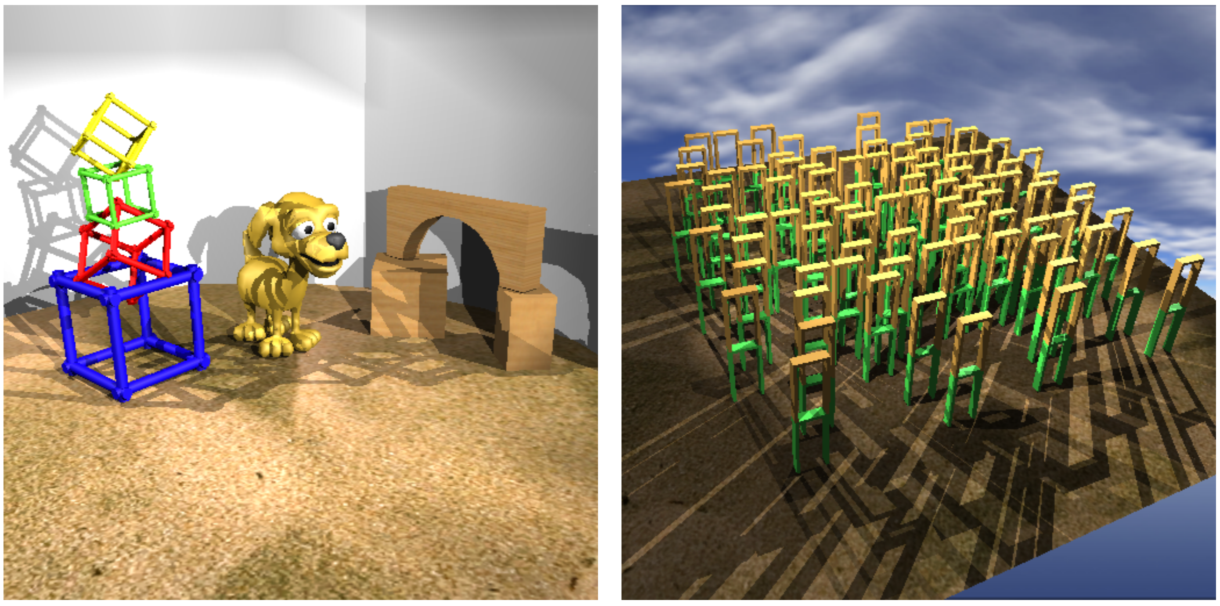

(a) (b)

Figure 4: *Two examples scenes with shadows from three light sources.*

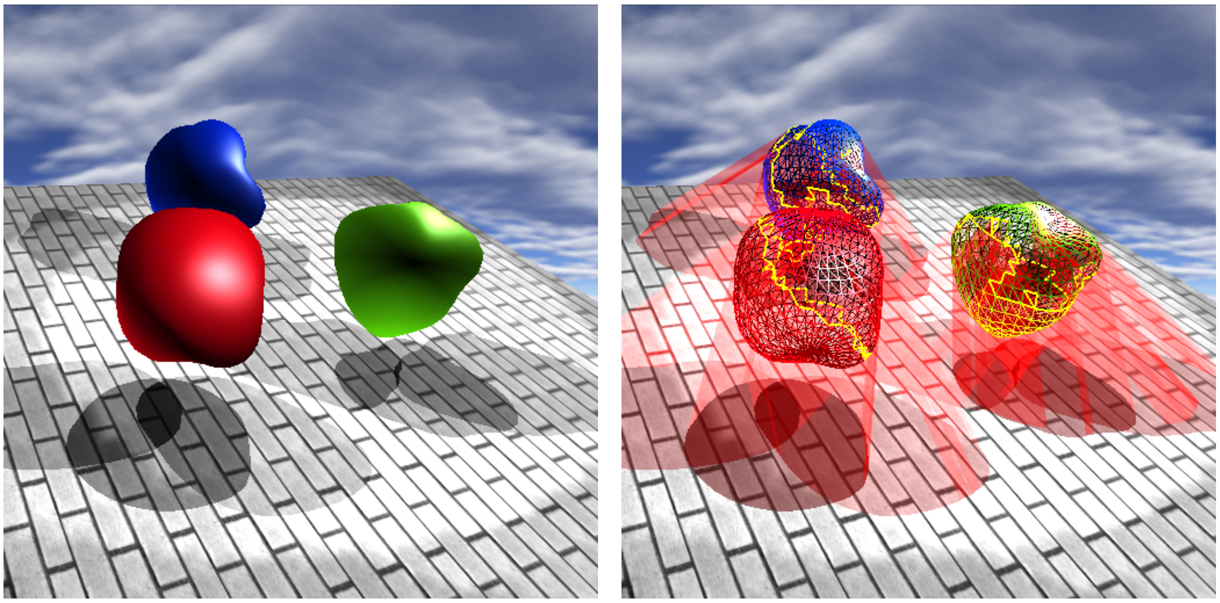


Figure 5: *Scene with three light sources. The three objects are simple spheres (1000 triangles) deformed by a procedural noise shader, implemented as a vertex program. The image on the right shows the silhouette edges for one light source and the extruded shadow volumes for all lights.*