Valient, M., "Hardware Generated Object Silhouettes". G - Journal for Geometry and Graphics, No 3, Volume 2, Slovak Society for Geometry and Graphics, 2005, ISSN 1336-524X.

Hardware Generated Object Silhouettes

Michal Valient
valient@dimension3.sk

Abstract
In this article, we present a high performance algorithm that renders thick object silhouettes completely on the GPU using only two passes. We present a new structure called topology aware mesh, which allows us to overcome the lack of global adjacency information during the vertex processing on GPU. Topology aware mesh stores complete face adjacency information for each vertex and allows edge computations to be performed on graphics hardware. Topology aware mesh consumes about 0.77 times the storage of the original mesh when using 16-bit precision and stream frequency capability of current hardware and it is created only once during the linear-time pre-processing step. Our approach can be used as a replacement for many CPU based silhouette detection algorithms in NPR or shadow rendering.

Key words: silhouette detection, silhouette rendering, GPU

1. Introduction and previous work

Silhouette detection and rendering plays an important role in many real-time non-photorealistic rendering algorithms as well as in some other active areas like the real-time shadow detection. Methods can be divided into the image space algorithms and geometric algorithms.

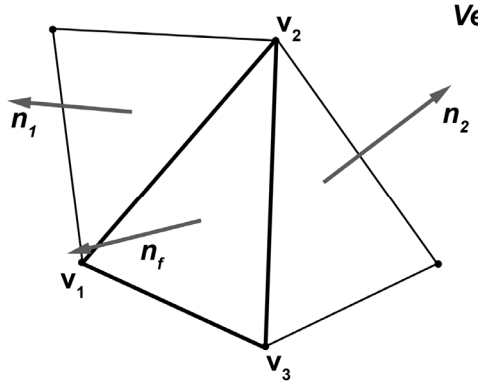
Image space algorithms use the frame buffer information to determine the feature edges. Mitchell et al. [1] use the graphics hardware to determine edges examining the scene depth buffer and world space normals rendered into the texture. These algorithms can be quite expensive because of higher fill-rate and they can operate only on pixel precision.

Geometric algorithms use the source geometry to identify the edge features and then render the silhouettes. These algorithms can provide sub-pixel level of information to the silhouette. The bottleneck of these algorithms might be the edge detection on CPU. The GPU based approaches have to overcome the limits of the stream-based architecture where shaders can access data only for one vertex or pixel. Card and Mitchell [2] and Gooch [3] introduce the hardware-based approach by packing the edge information and adjacent face information into each vertex. This allows rapid feature edge identification. McGuire and Hughes [4] have extended this idea and provide three pass algorithm for silhouette detection using the edge mesh presented in their paper. The main bottleneck of the edge mesh approach is that it requires about 9 times of the memory of the original mesh and the rendering runs 30 times slower when compared to the rendering of the scene alone. It requires separate pass to fill the possible gaps in the silhouettes and it can fail to so in some cases.

We propose different geometry based approach. Our approach produces meshes that have the same number of faces as the original geometry. The algorithm generates the silhouette edges completely in the pixel pipeline using scaled screen space representations of the original faces. Our approach uses only 0.77 times the memory requirements of the original mesh.

2. Topology aware mesh

Current graphical hardware does not provide topology information in the vertex shader. Vertex Shader 3.0 enabled hardware, like the GeForce 6800, supports texture reads in the vertex shaders and there is a possibility to store topology information in the textures, but this approach is limited by the texture read performance and it is not supported by any other current hardware. We decided to limit our approach to widely available Vertex Shader 2.0 enabled hardware. We have introduced a new mesh structure called topology aware mesh (TAM) to overcome the lack of topology information.



Vertex data for the face:

position	v_1	v_2	v_3
topology	v_1	v_1	v_1
	v_2	v_2	v_2
	v_3	v_3	v_3
	n_1	n_1	n_1
	n_2	n_2	n_2
	$-n_f$	$-n_f$	$-n_f$

Fig. 1. Structure of the topology aware mesh for triangle $v_1v_2v_3$. The table shows structure of vertex data for three vertices of TAM, which generate one face. Columns represent three items of the vertex stream and rows represent the information stored in each item. Negated face normal $-n_f$ is used for the edge v_1v_3 that is without a neighbour face.

Topology aware mesh is generated once for each object in the scene and it is valid while the topology of the original object remains unchanged. Hardware friendly vertex streams are being used to store the TAM structure. Each item of the vertex stream represents one vertex and three successive items represent one face. In our approach, a new TAM face is created with three unique new TAM vertices for each triangle of the original mesh and placed on the same position. We construct each TAM vertex to contain seven fields of data with information about entire face and three adjacent faces. One field is filled with the vertex position. Three fields are filled with position of all three vertices of the corresponding face. The last three fields are filled with normals of the adjacent faces. We store the negative value of the face normal for each triangle edge which does not have a neighbour. This ensures that such edge is always considered a part of the silhouette during the computation. The TAM face structure is shown on Figure 1. The first position field is used to produce screen space coordinates during the rendering. The information about normals is used to identify silhouette edges in the vertex shader and the three position fields are used to draw correct edges during the rendering. We describe the usage and rendering more precisely in Section 3.

2.1 Memory requirements

McGuire and Hughes [4] introduced a structure called edge mesh where they have used four vertices for each edge of the original mesh. Using half-precision in the data, this mesh consumes about 9 times the storage of the original mesh for typical closed meshes. McGuire and Hughes presented that the number of edges E is approximately equal 3 times the number of vertices V for a typical closed mesh.

In our approach, three new vertices for each face of the original mesh are created as can be seen on Figure 1. Each vertex contains seven vector fields and each vector has three components. The vertex consumes 84 bytes of data at full precision (where one component occupies 32-bits). If we store the three positions and normals in half-precision (16-bits per component), each vertex would contain only 48 bytes of information. Original mesh usually contains at least 44 bytes of information for each vertex (position, normal, tangent, texture coordinates). If every point of the original mesh is unique (the worst case) then the topology aware mesh consumes only 1.09 times more memory. For typical closed meshes with F faces, where E is about 3V, we get approximately 3F = 6V (knowing that 3F = 2E). In this case our mesh consumes about 1.53 times more space than the original mesh (for which we account also indexing information which requires additional 6F bytes for meshes up to 65535 vertices; larger meshes would require 12F bytes of indexing information).

Since information stored in the three vertices of a face is the same except for the position field, we can use the stream frequency feature of the modern hardware to reduce the memory consumption. Stream frequency of N means that each item in the vertex buffer is treated as N equal consecutive items. This feature allow us to store the equal information only once for the entire face. Such mesh would then consume only 0.77 times the storage of the original mesh (if E were about 3V).

3. Silhouette rendering

We render object silhouettes in two passes. The first pass uses the original geometry to render the depth information of visible pixels into the texture. We use this information to determine the edge visibility in the second pass. Depending on the application, the first pass can be also used to render the final scene without the edges, because it does not require special geometry or processing.

The second pass uses TAMs to render the silhouette edges of required thickness. The key feature of our algorithm is that it does not introduce new faces. Therefore, the vertex shader has to scale the screen representation of each face to contain all required silhouette features after the rendering. The pixel shader has to clip all pixels that are not part of the silhouette. We shortly describe the vertex and pixel stages of this pass and then provide detailed description of some steps.

3.1 Vertex processing

The vertex shader first detects the silhouette edges of the face. It computes the normal vector of current face n_F using the vertex position fields and computes the orientation to the eye vector v_{EYE}. For each edge, it also computes the orientation of adjacent face using normal vector of adjacent face n_E and compares it to the orientation of the current face. Edge of the triangle is marked as a part of the silhouette, only if the following condition is met

$$\text{sign}(n_F \bullet v_{EYE}) \neq \text{sign}(n_E \bullet v_{EYE}). \quad (1)$$

Triangle that does not contain silhouette edge is placed in front of the near clipping plane and collapsed into a single point – thus it is automatically rejected by the graphics card before entering the costly rasterization process.

3.1.1 Triangle shape manipulation

The face with silhouette edges is transformed to the screen space and processed in several successive steps. Following computations are performed in the screen space.

If the face contains only one silhouette edge, then it is modified to cover the smallest possible screen area. This optimization minimizes required pixel processing for the face. The modification moves the non-silhouette vertex close to the first silhouette vertex and makes one non-silhouette edge pixel wide and orthogonal to the silhouette edge.

We find the centre of the incircle $v_{incenter}$ for the face and radius of the incircle w_O. We identify the minimum silhouette width w_S, which is chosen conservatively to be always bigger than the expected width of the silhouette (specified as a rendering parameter). We compute the w_S by adding two times the length of the pixel's diagonal to the expected width. The reasons for such approach are the DirectX rasterization rules. DirectX renders a pixel only if the centre of the pixel lies inside the triangle. We make sure that centres of pixels at the silhouette borders are inside the triangle and they are properly rasterized by adding two times the length of the pixel's diagonal to the expected silhouette width. The precise shape is then computed later in the pixel shader. We define w_{SC} as a half of the w_S. Equation 2 is used to compute the factor required to scale the triangle so that each silhouette is completely covered.

$$f_{SCALE} = (w_O + w_{SC})/w_O \tag{2}$$

We scale the triangle using the scaling centre in the $v_{incenter}$ and the scale factor f_{SCALE}. The process is illustrated in the Figure 2.

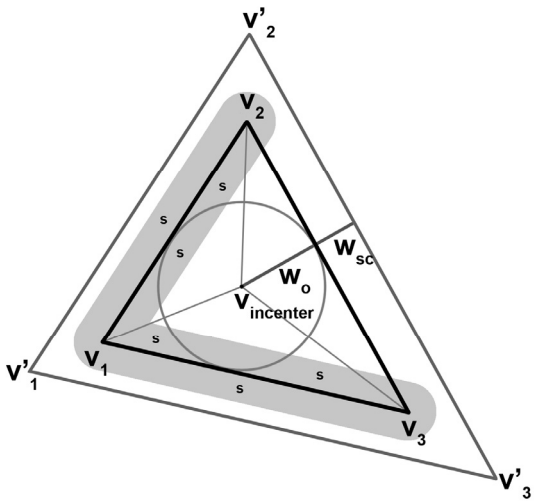


Fig. 2. Scaling of the triangle in the vertex shader. Triangle $V_1V_2V_3$ represents the original triangle. Triangle $V'_1V'_2V'_3$ represents triangle scaled using centre in incentre and scale factor $(w_o+w_{sc})/w_o$. Grey area marked with s represents the final silhouette.

Since the incentre computation and scaling is performed in the image space, we have to deal with a situation when the triangle degenerates after the perspective projection. Such triangle would not be rendered at all because it is not expanded by our algorithm. Vertex shader solves this case by moving one vertex of such triangle slightly in the reverse direction of the normal that belongs to the adjacent face of the opposite edge. We chose to pick the vertex that does not belong to the longest edge for this operation, but any vertex can be chosen as long as the original triangle remains inside of the new triangle. In fact, our implementation performs this correction step for all faces, where inradius w_O is smaller than the half of the diagonal of the pixel. The scale factors for such faces could become extremely large and the overflow errors might lead to incorrect rendering. It also helps to render very small triangles more efficiently because uncorrected triangles would cover many pixels forming a large spike after the expand operation. Most of these pixels would have to be

rejected using costly per-pixel computations. The correction step is performed before the scaling computation.

At the end, vertex shader sends original image space positions and depths of three vertices of the face and a flag that identifies each silhouette edge to the pixel shader.

3.2 Pixel processing

The main purpose of the pixel shader is to reject all pixels that do not belong to the silhouette. It consists of three major steps. Pixel is clipped if it does not belong to the thick silhouette because its distance to the nearest triangle silhouette edge is bigger than half of the required silhouette width. We find the nearest silhouette edge point for current pixel and use depth buffer information from the first pass to determine if the edge point is visible. We clip the pixel for each invisible point. Figure 3 shows different areas of the silhouette.

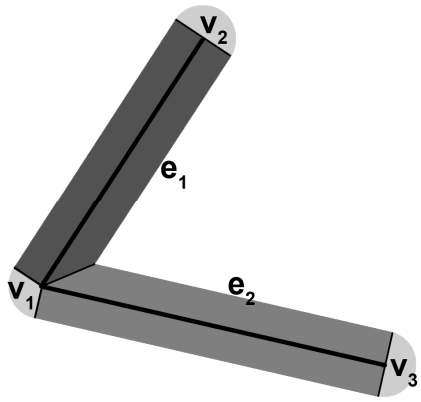


Fig. 3. Different areas of the silhouette edges. Distance to the edge vertex is used in light grey areas marked as v_1, v_2 or v_3. Smallest distance to the nearest edge is used in areas marked as e_1 and e_2.

This approach effectively renders a capsule along each edge. We disable the z-buffer during rendering because we perform a custom depth test. To ensure correct silhouette rendering on overlapping areas we enable the alpha test to blend the capsules correctly together. Alpha channel of each silhouette pixel is filled with the distance to the nearest silhouette edge and the frame-buffer alpha test is set to LESSEQUAL mode. Only pixels with smaller distance to the silhouette edge are rendered in overlapping areas.

4. Results and discussion

We have tested our implementation of the algorithm on the machine with Pentium 4 2.4GHz processor and 1GB DDRAM. The testing card was ATI Radeon X800XT Platinum edition with 256MB of DDRAM. The algorithm uses pixel shader version 1.1 for the first pass and pixel shader 2a for the second pass. Vertex shader for the second pass generation uses 123 instructions and pixel shader uses 70 instructions. The results can be found in Fig. 4.

We tested our method on the scene with 30 000 visible polygons and we discovered that the second pass, which uses TAM meshes, is about 4.7 times slower than the first pass (running at 1100FPS) for silhouettes with the width 2 pixels and about 8.8 times slower for the 6-pixels wide silhouettes. The algorithm still produces high frame rates and it is suitable for frame rate intensive applications such as computer games.

We have observed that the performance hit can be higher for the scenes where a few big silhouette triangles cover most of the scene. In such case the costly pixel shader is executed

for most of the pixels (even when a lot of pixels got clipped in the result). Our algorithm can be extended to change the TAM meshes for such big triangles. Instead of a single triangle we generate three tiny triangles, one for each edge of the original face, and modify the normals in TAM vertex structure so that only original edge could be identified as a silhouette. This straightforward approach noticeably lowers the fill-rate problem, but we consider it application dependent because the triangle modification depends on a concrete scene and a camera configuration. Therefore, we did not include it in our algorithm, but we propose it as a suitable solution for frame rate intensive applications such as games, where these conditions are known at the design time. The number of TAM faces can be lowered during the pre-processing for all meshes that would not be deformed. We can identify and remove all faces that cannot contribute to the silhouette.

Our algorithm can be adapted for animated meshes (i.e. matrix skinning) but this information has to be specified for each vertex of the TAM. This would of course slightly increase the memory requirements of the mesh and the cost of vertex processing of such vertex. We plan to work in this area in the future as well as we plan to optimize the silhouette rendering shaders to utilize parallel vector computations of scalars for better performance and to lower the number of instructions to fit into the Pixel shader 2.0 limits.

We did not specify other rendering information than the alpha value for the silhouette rendering because this depends on the application. The non-photorealistic rendering might require the additional pixel shader code which computes the coverage of the pixel by the edge and uses this information to approximate anti-aliased silhouette edges, and it might even replace the LESSEQUAL alpha test with other blending type. We successfully use this algorithm to rapidly generate soft penumbra shadows. The thick silhouette represents the penumbra area in the shadow map and it contains the information about distance to the silhouette edge and the depth of the silhouette edge. The advantage over the image space approach is that our silhouettes can hold information at sub-pixel levels. Our approach also rapidly accelerates the original shadow silhouette maps algorithm [5] by replacing the slow CPU-based silhouette detection.

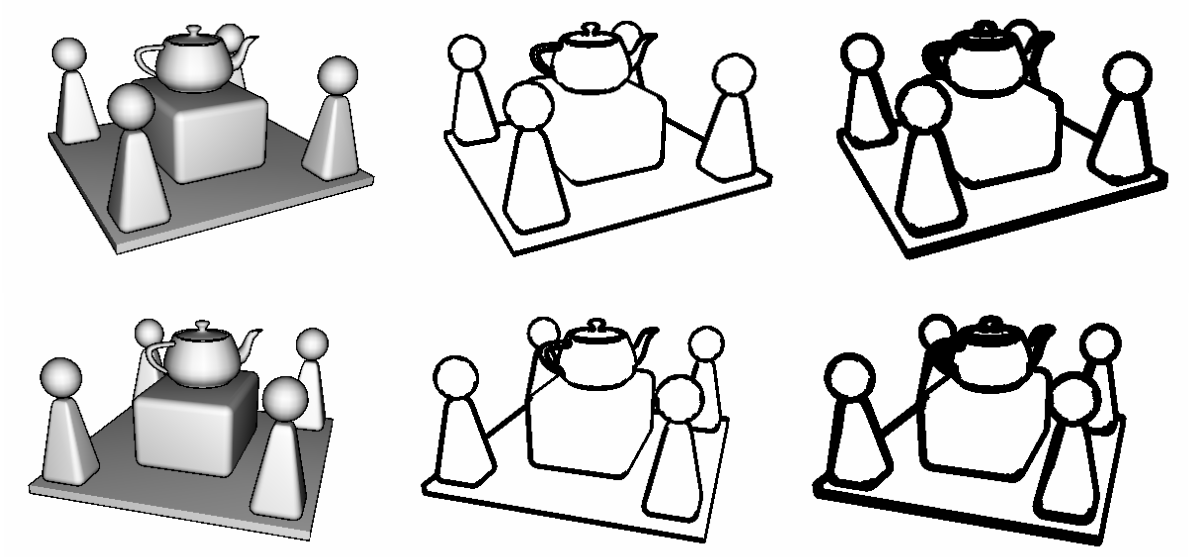


Fig 4: Examples of the detected silhouettes on the teapot scene (9217 triangles). Left column shows shaded scene combined with silhouettes with the width 2 pixels. Middle column shows only the silhouettes of the same scene with the width 4 pixels and the right picture shows silhouettes with the width 8 pixels.

5. Conclusion

We have presented an algorithm that renders object silhouettes in two GPU passes using proposed topology aware mesh structure. It uses single pre-processing CPU pass and then runs completely on the graphics hardware. Our approach allows rendering of sub-pixel precise information in the silhouettes because of its geometrical basis.

There are some drawbacks of our algorithm. It adds additional mesh structure that requires about 0.77 times of the original mesh memory usage and additional vertex and pixel processing that is however well handled using latest generation of graphics cards.

6. Acknowledgements

We would like to thank Andrej Ferko and Pavol Eliáš for their valuable comments and corrections. This research was supported in part by the VEGA grant reg.no.1/0174/03.

7. References

[1] MITCHELL, J., BRENNAN, C., CARD, D. *Real-Time Image-Space Outlining for Non-Photorealistic Rendering*. SIGGRAPH 2002 Sketch. http://www.ati.com/developer/SIGGRAPH02/SIGGRAPH2002_Sketch-Mitchell.pdf

[2] CARD, D., MITCHELL, J. Non-Photorealistic Rendering with Pixel and Vertex Shaders. In ENGEL, W. *ShaderX: Vertex and Pixel Shaders Tips and Tricks*, Plano, TX: Wordware Publishing, May 2002, ISBN 1556220413.

[3] GOOCH, B. in *Theory and Practice of Non-Photorealistic Graphics: Algorithms, Methods, and Production Systems*, course organized by M. C. Sousa. SIGGRAPH 2003, Course notes 10.

[4] MCGUIRE, M., HUGHES, J., F. Hardware-determined feature edges. In *Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*. New York, NY: ACM Press, 2004. ISBN 1-58113-887-3, p.35-147.

[5] SEN, P., CAMMARANO, M., HANRAHAN, P. Shadow Silhouette Maps. In *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, vol. 22, no. 3. New York, NY: ACM Press, July 2003. ISSN 0730-0301, p.521-526.