5.1

Cartoon Rendering: Real-time Silhouette Edge Detection and Rendering

Carl S. Marshall, Intel Architecture Labs

carl.s.marshall@intel.com

Silhouette detection and rendering is a key component for adding a stylized look to 3D cartoon rendering. The basic concept of silhouette edge detection is to find the important edges that depict the outline of a model. Many cartoon animators illustrate these silhouettes by tracing a black outline around the borders of a model. This cartoon rendering gem describes several silhouette edge detection techniques: an edge-based detection method, a programmable vertex shader technique, and an advanced texturing technique. The advantages and disadvantages of each technique are highlighted.

Inker

The terms *inking* and *painting* come from the artist's traditional cartoon eel creation process. Inking a eel is the process of drawing the outlines of the characters and objects in the scene, while painting is the process of coloring or filling in the interior of the outlines. This gem focuses on the inking process of cartoon rendering. Adam Lake's gem on "Cartoon Rendering Using Texture Mapping and Programmable Vertex Shaders" covers the painting process. Together, these two techniques form a complete cartoon rendering engine. Figure 5.1.1 demonstrates the inking and painting process on a 3D duck model.

The general inking process is comprised of two parts. The first part is the *detection* of the silhouettes, and the second is the *rendering* of the silhouettes. This gem describes this inking process using several techniques that can all be performed in real time on both manifold and nonmanifold meshes.

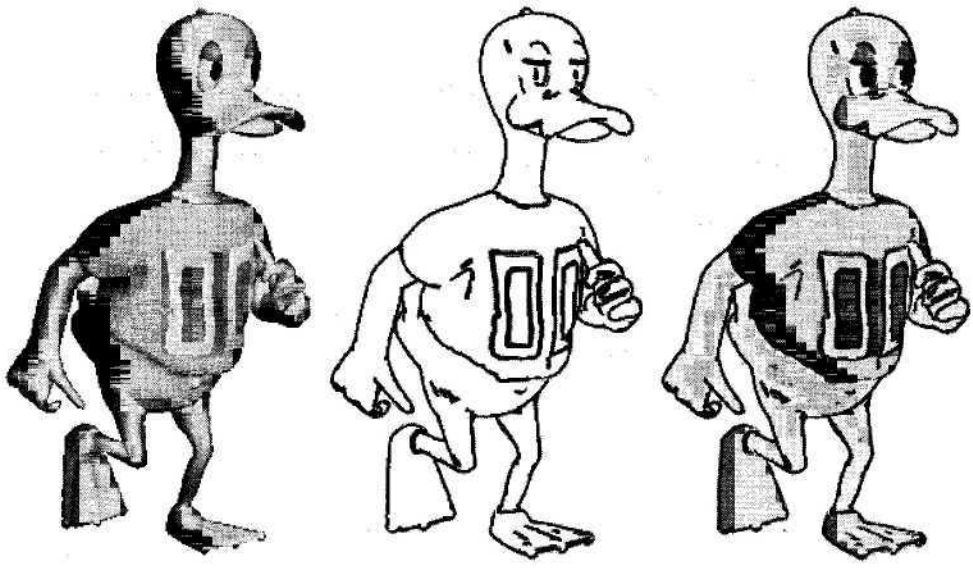

FIGURE 5.1.1 *A gouraud-shaded duck (left), inked duck with faces colored as background (center), and a flat-shaded duck using the painter (right).*

Important Edges

Silhouette edge detection (SED) is the main component of the inker. Along with the detection and rendering of silhouettes, there are other important edges of the model that artists tend to highlight: the *crease* and *boundary* edges. As described in the introduction, silhouette edges are the edges that form the outline of a model, but can also contain some internal edges, as shown in Figure 5.1.2. One aspect that is very important to silhouette edge detection is that silhouettes are view dependent. This means that the silhouettes have to be re-detected and rendered with each change in the movement of the camera or model. Crease angle edges are those edges whose angle

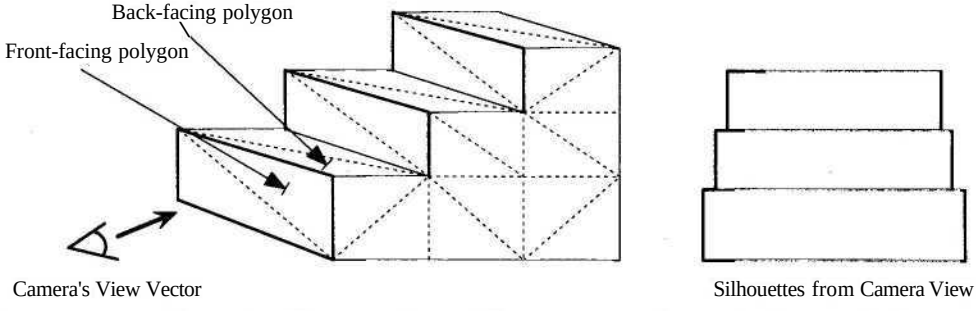

FIGURE 5.1.2 *Shows the silhouettes detected from a camera facing the first step (left). (Right) shows the silhouettes from the camera's view of the stairs.*

between its two neighboring faces is within a given threshold. This is called the dihedral angle. A user-defined threshold is used to detect crease angles. The boundary edges are edges that border a discontinuity in the mesh. A discontinuity in the mesh occurs where two faces that border an edge contain the same vertices for that edge but have one of the following differences: texture coordinates, materials, or normals.

Silhouette Edge Detection Techniques

Why are there so many different silhouette edge techniques? For starters, there are several rendering APIs that have certain advantages and disadvantages as far as rendering performance, features, and functionality on various video cards. This gem describes a summary of experiments on many of the inking methods. Texture mapping features, line segment rendering, and polygon offsetting are just a few of the features that vary dramatically across hardware and APIs.

Edge-based Inking

The term *edge-based* is used to describe a technique that analyzes the edges of the polygons of the model to detect and render its important edges. The full silhouette of the model along with accurate silhouettes and crease edges are guaranteed with this method. Other inking techniques imply silhouette and crease edges through an estimation of surface curvature. The edge-based technique consists of two parts: a *preprocess* part and a *run-time* part.

Preprocess

The first step in the preprocess is to allocate and find a unique *edge list* for the model. The three edges of each face of the model are examined individually and inserted into a hash table if they have not previously been stored. Once the hash table is complete, a compacted array can be stored with only the unique edges. An edge entry includes two vertex indices, two face indices, and a flag entry. The flag entry describes the edge as a silhouette, crease angle, boundary, and/or an unimportant edge.

To detect silhouettes accurately, face normals have to be computed. If the model is static (nonanimating), then these normals can be precomputed along with the crease angle edges. For nonprogressive meshes, the boundary edges can be processed at authoring or load time. The trade-off for accurate silhouettes at runtime for animating objects is the recomputation of the face normals every frame.

Runtime

Runtime consists of detecting and rendering the important edges of the model. V refers to the viewing vector, and Nj and N_2 are the face normals of the triangles that share the edge in consideration.

Runtime Inking

1. If animating geometry, recalculate face normals.
2. For each edge in the unique edge list,
   - Compute *V* by subtracting the viewing vector position from one of the edge vertex positions.
   - Set boundary flag if edge has one neighboring face or two neighboring faces with discontinuities.
   - If animating geometry, set crease flag if the dihedral angle between the two neighboring faces exceeds a given threshold.
   - Detect silhouette using (A/i • V) X $(N_2 \cdot V) < 0$ and set silhouette flag.
3. Iterate over edge list and render edges whose edge flag is set. Creating separate silhouette, crease, and boundary edge lists can optimize the rendering of the edges since each edge list can be rendered in a single API call. If the API you are using supports polygon offsetting and line thickness, they both give added visual appeal to the final rendering.

Step 2 is the core to the silhouette detection algorithm. By calculating the dot products between the face normal and the view vector for each of the edge's neighboring faces, highly accurate silhouettes are detected. Vertex normals could be used as an optimization, but this technique would miss silhouettes much like many of the vertex and pixel shader-based approaches. See Listing 5-1.1 for a code example.

Advantages

- Cross-platform solution.
- Completely accurate silhouette edge detection.
- Artists can choose threshold for crease angle edges.
- Visibility of edges is solved via the z-buffer.

Disadvantages

- Line thickness is only available under certain APIs.
- Must compute face normals for silhouette edge detection.
- An offset needs to be applied to the lines to avoid the model occluding the line.

Listing 5.1.1 Edge-based silhouette edge detection

```
// Detect silhouettes
Edge *pEdge;

// Iterate over all edges
for(unsigned int i=0; i < numEdges; i++){
    pEdge = &m_pEdgel_ist[i];

        //Calculate vector from eye
    pEdgeVertexPosition =
        pMesh->GetVertexPosition(pEdge->GetVertexIndex(0));
```

```
        // Subtract eyeposition from a vertex position on the edge
        viewVector =  pEyePosition - pEdgeVertexPosition;
        view/Vector.normalize();

        // Calculate the dot products from the face normal and
        // the view vector
        uDotProductl = DotProduct(viewVectpr,  *pFaceNormalA);
        uDotProduct2  =  DotProduct(viewVector,   *pFaceNormalB);

        if((uDotProductl * uDotProduct2) <= 0.0f){
             // The edge is a silhouette edge
             AddEdgeToRenderEdges(pEdge,   uNumfienderedVertices);
        }
    }
```

Programmable Vertex Shader Inking

A programmable vertex shader is an extremely flexible graphics API for modern graphics architectures. Why is a programmable vertex shader able to help with inking? Since we have shown that inking relies on the dot product of a normal with the view vector, we can pass the view vector and normals to the graphics hardware to find silhouettes.

Programmable Vertex Shader Inking Runtime

1. Set up texture pipeline via the graphics API. This requires the creation or loading of a one-dimensional texture. It works fine to load a two-dimensional texture, but only the first row will be used. In setting up the texture, the color may be varied when moving across the texture from the silhouette color, usually black, to any color desired. A 1x32 or 1x64 texture with only a few black pixels near u=0 progressing to the surface material color for the rest of the texels is recommended.
2. Load the registers of the programmable vertex shader. What you need to load may depend on your specific application and where you are doing your transformations, but normally you need to send the *world to eye and projection* matrix. Next, the *vertex position* and *vertex normal should* be sent and transformed. Then, the *eye position* is sent and used to compute texture coordinates. Since new texture coordinates are computed, they do not need to be passed to the card. However, remember to make sure that the graphics unit expects texture coordinates to be generated for the model.
3. Perform the transformation of the vertex position from model to homogenous clip space.
4. For each vertex, create a view vector by subtracting the vertex position in world space from the eye vector. Compute the dot product between the vertex normal and the view vector, $N^* V$, shown in Listing 5.1.2. Make sure the normal and view vector are in the same coordinate frame or the results

will be incorrect. This can be done on the graphics unit or the CPU before being loaded into the constant registers of the vertex shader.

5. Store the result of step 4 as the texture coordinate for the *u* texture coordinate.

Another silhouette programmable vertex shading technique is a two-pass technique that first extrudes the vertex along its normal. Then the depth buffer is disabled and the model is rendering in the desired silhouette color. Next, the depth buffer is reenabled and die model is rendered in its normal state.

Advantages

- Fast. All silhouette detection processing is placed on graphics card.
- Varying line thickness.

Disadvantages:

- Less accurate. Relies on vertex normals that depend on the object's shape.
- Does not detect crease or boundary edges without special variables.
- Requires programmable vertex shader API support.

Listing 5.1.2 Silhouette shader for DirectX 8.0 vertex programmable shading

```
;-------------------------------------------------------------
; Constants specified by the application
;    c8-c11  = world-view matrix
;    c12-c15 = view matrix
;    c32   = eye vector in world space
;
; Vertex components (as specified in the vertex DECL)
;    v0    = Position
;    v3    = Normal
;-------------------------------------------------------------

;-------------------------------------------------------------
; Vertex transformation
;-------------------------------------------------------------
; Transform to view space (world matrix is identity)
;  m4x4 is a 4x4 matrix multiply
m4x4 r9, v0, c8
; Transform to projection space
m4x4 r10, r9, c12
; Store output position
mov oPos, r10

;-------------------------------------------------------------
; Viewing calculation eye dot n
;-------------------------------------------------------------
;first, make a vector from the vertex to the camera
;value in r9 is in world space (see vertex xform above)
```

```
sub r2, c32, r9      ;make vector from vertex to camera
; now normalize the vector
; dp3 is a dotproduct operation
mov r3, r2           ;make a temp
dp3 r2.x, r2, r2          ;r2^2
rsq r2.x, r2.x       ;1/sqrt(r2^2)
mul r3, r2.x, r3     ;(1/sqrt(r2^2))*r3 = r3
dp3 oT0.x, r3, v3    ; (eye2surface vector) dot surface normal
```

Inking with Advanced Texture Features

An innovative technique that can be performed fully on the graphics processor is described in detail in [DietrichOO]. This method uses a per-pixel *Ndot Vto* calculate and obtain the silhouette. An alpha test is used to clamp the values to a black outline, and a per-primitive alpha reference value is used to vary the line thickness. The basic steps are listed next, but for more details, see the reference.

Basic Steps

1. Create a normalizing cube map texture. Effectively, this creates a single texture that contains the six square faces. Each face contains a per-pixel RGB normal that represents the direction of that point on the unit cube from the origin.
2. Use view space normal texture coordinates as a lookup into the cube map texture. This outputs a RGB-encoded vector representing A^.
3. Use view space position texture coordinates as a lookup into the cube map texture. This outputs a RGB-encoded vector representing *V.*
4. Set the Alpha compare mode to LESS_EQUAL.
5. Set the Alpha reference value to 0 for thin lines, higher for thicker lines.
6. Perform the dot product on the color values from steps 2 and 3, performing *NdotV.*
7. Perform one pass with the alpha test LESS_EQUAL for the silhouettes, and then another pass with alpha test set to GREATER for the shaded object.

Advantages

- Very fast with appropriate hardware.
- Allows line thickness to vary.

Disadvantages

- Less accurate. Relies on vertex normals that depend on the object's shape.
- Requires specific thresholds for detecting crease angles, and specific alpha reference values for boundary edges.
- Requires specific hardware for speed.

Conclusion

The inking and painting techniques used to cartoon render a scene are just a couple of techniques in a larger domain called nonphotorealistic rendering (NPR). There are many other stylized rendering techniques in the NPR field with which to experiment. A few of these techniques are pencil sketching, water coloring, black-and-white illustrations, painterly rendering, and technical illustrations. Inking is a fundamental tool and is a great place to start building your NPR library. This library will help you distinguish your game from others by giving you the flexibility of alternate rendering styles that will potentially broaden the appeal of your game.

References

[DietrichOO] Dietrich, Sim. "Cartoon Rendering and Advanced Texture Features of the GeForce 256 Texture Matrix, Projective Textures, Cube Maps, Texture Coordinate Generation and DOTPRODUCT3 Texture Blending.' Available online at: www.nvidia.com/Marketing/Developer/DevRel.nsf/WhitepapersFrame. 2000.

[GoochOl] Gooch, Bruce, and Amy Gooch. Non-Photorealistic Rendering, A.K. Peters, Ltd., 2001.

[IntelOO] Intel's Graphics and 3D Technologies Web page. Available online at: http://developer.intel.com/ial/3dsoftware. 2000.

[LakeOO] Lake, Adam, Carl Marshall, Mark Harris, and Mark Blackstein. "Stylized Rendering Techniques of Real-Time 3D Animation." Non-Photorealistic Animation and Rendering Symposium, pp. 13-20. 2000. ftp://download.intel.com/ial/3dsoftware/toon.pdf.

[Markosian97] Markosian, Lee, Michael Kowalski, et al. "Real-Time Nonphotorealistic Rendering." In Proceedings of ACM SIGGRAPH 97, pp. 113-122. 1997.