Real-Time Non-Photorealistic Rendering

Presented by: Qing Hu LIAO

SOCS, McGill
Feb 1, 2005
Index

- Introduction
- Motivation
- Appel’s Algorithm
- Improving Schema
- Rendering
- Result
“Economy of line”

- A great deal of information could be effectively conveyed by very few strokes.
Introduction

- NPR often emphasizes the **visible features edges**: 
  - Silhouette;
  - Crease, border, and etc.
Introduction

- NPR often emphasis the **visible features edges**:
  - Silhouette;
  - Crease, border, and etc.

- Naïve visible-line detection algorithm (**Ray Tracing**):
  - Easy to implement;
  - But slow: $O(n^2)$. 

Morgan McGuire

Morgan McGuire
jgt 2004
Motivation

- Find and render visible feature edges in real-time.
  - use a modification of Appel’s visible-line algorithm (APPE67).
Motivation

- Find and render visible feature edges in **real-time**.
  - use a modification of Appel’s **visible-line algorithm** (APPE67).

- Why not Z-buffer algorithm?
Motivation

- Find and render visible feature edges in **real-time**.
  - use a modification of *Appel’s visible-line* algorithm (APPE67).

- Why not Z-buffer algorithm?
  - Can’t easily produce different hand-drawn styles for visible feature lines or surface:
    - Since the hand-drawn strokes have different geometry shape with the corresponding feature lines or surface.
Motivation

- Find and render visible feature edges in **real-time**.
  - use a modification of **Appel’s visible-line** algorithm (APPE67).

- Why not Z-buffer algorithm?
  - Can’t easily produce different hand-drawn styles for visible feature lines or surface:
    - Since the hand-drawn strokes have different geometry shape with the corresponding feature lines or surface.
  - Historical reason:
    - Vertex and fragment Shader (GPU) not available.
Definitions

- **Front-facing & Back-facing Polygons**
  - Determined by the sign of dot-product of normal direction and view direction.

Appel’s visible-line algorithm
Definitions

- **Silhouette Edge**
  - Adjacent to one front-facing and one back-facing polygon.
  - Edge AB, CD, KL, DF, ...

Appel’s visible-line algorithm
Definitions

- **Silhouette Edge**
  - Adjacent to one front-facing and one back-facing polygon.
  - Edge AB, CD, KL, DF, ...

- **Border Edge**
  - Adjacent to just one face.

Appel’s visible-line algorithm
Definitions

- **Silhouette Edge**
  - Adjacent to one front-facing and one back-facing polygon.
  - Edge AB, CD, KL, DF, ...

- **Border Edge**
  - Adjacent to just one face.

- **Contour Edge:**
  - Silhouette edge;
  - Border edge.

Appel’s visible-line algorithm
Quantitative Invisibility ($QI$)

- $QI$ of a point on a line:
  - The number of front-facing polygons that obscure that point from eye;

Appel’s visible-line algorithm
Quantitative Invisibility (QI)

- QI of a point on a line:
  - The number of front-facing polygons that obscure that point from eye;
  - Increase by 1 when pass behind a front-facing polygon;

Appel’s visible-line algorithm
Quantitative Invisibility (QI)

- QI of a point on a line:
  - The number of front-facing polygons that obscure that point from eye;
  - Increase by 1 when pass behind a front-facing polygon;
  - Decrease by 1 when pass out that front-facing polygon;

Appel’s visible-line algorithm
Quantitative Invisibility (QI)

- **QI** of a point on a line:
  - The number of front-facing polygons that obscure that point from eye;
  - Increase by 1 when pass behind a front-facing polygon;
  - Decrease by 1 when pass out that front-facing polygon;
  - Only change when pass behind a contour line (silhouette or border edges).
Quantitative Invisibility (QI)

- **QI** of a point on a line:
  - The number of front-facing polygons that obscure that point from eye;
  - Increase by 1 when pass behind a front-facing polygon;
  - Decrease by 1 when pass out that front-facing polygon;
  - **Only change when pass behind a contour line (silhouette or border edges).**

- A point is visible only when QI = 0.

Appel's visible-line algorithm
Complication Vertex

- **Complication vertex** is vertex on silhouette edges:
  - Through which the QI value should be computed according to local adjacent polygons.
Complication Vertex

- **Complication vertex** is vertex on silhouette edges:
  - Through which the QI value should be computed according to local adjacent polygons.
  - Examples of complication vertex:
    - Vertex K: QI change along J-K-L.
Complication Vertex

- **Complication vertex** is vertex on silhouette edges:
  - Through which the QI value should be computed according to local adjacent polygons.
  - Examples of complication vertex:
    - Vertex **K**: QI change along J-K-L.
    - Vertex **K**: no change along J-K-M.

Appel’s visible-line algorithm
Complication Vertex

- **Complication vertex** is a vertex on silhouette edges:
  - Through which the $Q_I$ value should be computed according to local adjacent polygons.
  - Examples of complication vertex:
    - Vertex $K$: $Q_I$ change along $J$-$K$-$L$.
    - Vertex $K$: no change along $J$-$K$-$M$.
  - Examples of normal vertex:
    - Vertex $E$: no change along any path.

Appel’s visible-line algorithm
Edge coherence

- **Edge coherence**: along the mesh of edges, Q1 only changes when:
Edge coherence

- **Edge coherence**: along the mesh of edges, QI only changes when:
  - Pass behind a contour edge, or
Edge coherence

- **Edge coherence:** along the mesh of edges, QI only changes when:
  - Pass behind a contour edge, or
  - Encounter a complication vertex.
Appel’s visible-line Algorithm

- Take advantage of edge coherence to compute QI value of all the vertex in the models from a viewpoint:

<table>
<thead>
<tr>
<th>No.</th>
<th>Step</th>
<th>Method</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Appel’s visible-line Algorithm**

- Take advantage of **edge coherence** to compute QI value of all the vertex in the models from a viewpoint:

<table>
<thead>
<tr>
<th>No.</th>
<th>Step</th>
<th>Method</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Detect all the contour edges of the models from a specified viewpoint.</td>
<td>Exhaustive search</td>
<td>Heavy</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appel’s visible-line Algorithm

- Take advantage of **edge coherence** to compute QI value of all the vertex in the models from a viewpoint:

<table>
<thead>
<tr>
<th>No.</th>
<th>Step</th>
<th>Method</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Detect all the contour edges of the models from a specified viewpoint.</td>
<td>Exhaustive search</td>
<td>Heavy</td>
</tr>
<tr>
<td>2</td>
<td>Compute QI for a “seed” vertex via brute-force computation.</td>
<td>Ray tracing</td>
<td>Medium</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appel’s visible-line Algorithm

- Take advantage of **edge coherence** to compute QI value of all the vertex in the models from a viewpoint:

<table>
<thead>
<tr>
<th>No.</th>
<th>Step</th>
<th>Method</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Detect all the contour edges of the models from a specified viewpoint.</td>
<td>Exhaustive search</td>
<td>Heavy</td>
</tr>
<tr>
<td>2</td>
<td>Compute QI for a “seed” vertex via brute-force computation.</td>
<td>Ray tracing</td>
<td>heavy</td>
</tr>
<tr>
<td>3</td>
<td>Propagate the seed’s QI value out through connected edges by (1): finding the intersections with contour edges in <strong>image space</strong> and (2): taking care of the complication vertex.</td>
<td>Sweep-line</td>
<td>Medium</td>
</tr>
</tbody>
</table>
Improving Appel’s Algorithm

This paper proposed some improvements over the original Appel’s algorithm and make it suitable for real-time interactive application.
Improving Appel’s Algorithm

- This paper proposed some improvements over the original Appel’s algorithm and make it suitable for real-time interactive application.
- Focus primarily on finding the silhouette edges quickly, since:
  - Directly improve step 1 of Appel’s algorithm.
  - The most important feature edges for NPR.
Improving Appel’s Algorithm

- Step 1: A fast randomized algorithm for finding silhouette.
Improving Appel’s Algorithm

- **Step 1:**
  - A fast randomized algorithm for finding silhouette.

- **Step 2:**
  - Avoiding ray tracing tests for “seed” to determine visibility of silhouette, or
  - Efficient ray tracing via “walking” or Octree Technique.
Improving Appel’s Algorithm

- **Step 1:**
  - A fast randomized algorithm for finding silhouette.

- **Step 2:**
  - Avoiding ray tracing tests for “seed” to determine visibility of silhouette, or
  - Efficient ray tracing via “walking” or Octree Technique.

- **Step 3:**
  - Check Cusp vertex along silhouette edges.
Fast Randomized Algorithm for Finding Silhouettes

- Silhouette edges form closed curves on 3D model space.
Fast Randomized Algorithm for Finding Silhouettes

- Silhouette edges form closed curves on 3D model space.

- Algorithm:
  - Weight all the edges according to:
    - dihedral angle.
    - Being silhouette edges in previous frame.
Fast Randomized Algorithm for Finding Silhouettes

- Silhouette edges form closed curves on 3D model space.

- Algorithm:
  - Weight all the edges according to:
    - dihedral angle.
    - Being silhouette edges in previous frame.
  - Choose a small fraction of edges randomly according to the weight.
Fast Randomized Algorithm for Finding Silhouettes

Silhouette edges form closed curves on 3D model space.

Algorithm:

- Weight all the edges according to:
  - dihedral angle.
  - Being silhouette edges in previous frame.
- Choose a small fraction of edges **randomly** according to the weight.
- If find a silhouette edge, trace out the entire silhouette curve.
Fast Randomized Algorithm for Finding Silhouettes

- Silhouette edges form closed curves on 3D model space.

Algorithm:

- Weight all the edges according to:
  - dihedral angle.
  - Being silhouette edges in previous frame.
- Choose a small fraction of edges randomly according to the weight.
- If find a silhouette edge, trace out the entire silhouette curve.

This algorithm can’t guarantee all the silhouette edges will be found, but in practice it is extremely efficient and effective:

- Tradeoff between accuracy and speed.
Cusp Vertex on Silhouette

- A complication vertex on silhouette curve is **cusp vertex** if:
  - Adjacent to 2 silhouette edges, one front-facing and the other back-facing.
  - Or adjacent to more than 2 silhouette edges,
  - Or adjacent to a border edge.
A complication vertex on silhouette curve is **cusp vertex** if:
- Adjacent to 2 silhouette edges, one front-facing and the other back-facing.
- Or adjacent to more than 2 silhouette edges,
- Or adjacent to a border edge.

**QI** along a silhouette curve can change at a vertex only if that vertex is **cusp vertex**.
Avoiding Ray Tracing (1)

- Compute the 2D bounding box of all the silhouette curves in image space of current frame.

- The silhouette curves which touch the bounding box don’t need ray tracing.
  - Touching point’s $Q_I = 0$. 

McGill CS767 Winter 2005
Avoiding Ray Tracing (2)

- For each connected silhouette curves, choose an arbitrary base point and assume its initial QI value = 0.
Avoiding Ray Tracing (2)

- For each connected silhouette curves, choose an arbitrary base point and assume its initial QI value = 0.
- Propagate the initial QI value along each silhouette curves.
Avoiding Ray Tracing (2)

- For each connected silhouette curves, choose an arbitrary base point and assume its initial QI value = 0.
- Propagate the initial QI value along each silhouette curves.
- Adjust the initial QI values for the whole silhouette curve according to:
  - QI >= 0
  - Occlusion relationship between the intersection edges from different silhouette curves.
Avoiding Ray Tracing (2)

- For each connected silhouette curves, choose an arbitrary **base point** and assume its initial QI value = 0.
- Propagate the initial QI value along each silhouette curves.
- Adjust the initial QI values for the whole silhouette curve according to:
  - QI $\geq 0$
  - Occlusion relationship between the intersection edges from different silhouette curves.
- For some silhouette curves, the minimum value of QI is greater than 0, and then ray tracing is not necessary.
Effective Ray Tracing

- "Walking" Algorithm:
  - Only work when all objects are in front of camera;
  - Fast but complex.

- Octree Algorithm
  - Immersive scene.
Rendering visible lines (1)

- Drawing the lines directly with slight enhancement, such as:
  - Width;
  - Color.

A mechanical part
Rendering visible lines (2)

- High-resolution “artistically” perturbed strokes defined by adding offsets to the lines:

Mechanical part rendered with sketchy lines
Rendering visible lines (3)

- Texture-mapped strokes:

A charcoal-like rendering of terrain with texture-mapped strokes

McGill CS767 Winter 2005
# Performance

<table>
<thead>
<tr>
<th>Model</th>
<th>Triangles</th>
<th>Frames/sec</th>
<th>Triangles/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two torii</td>
<td>65,536</td>
<td>30.58</td>
<td>2,004,091</td>
</tr>
<tr>
<td>Mechanical part</td>
<td>64,512</td>
<td>14.69</td>
<td>947,681</td>
</tr>
<tr>
<td>Venus</td>
<td>90,752</td>
<td>17.83</td>
<td>1,618,108</td>
</tr>
</tbody>
</table>

**Table 1** Performance of basic visible-line renderer. Times were measured on a 200 MHz Ultrasparc.
Extensions

The variation of the algorithm also could help to fast locate the other features:

- Visible crease, border edges;
- hidden silhouette.
- Visible shadow surface.
More Results

Human figure with expressive outline and shading strokes

Mechanical part with lines in varied styles

McGill CS767 Winter 2005
Recommended Reading

- "Hardware Determined Edge Features"
  - by Morgan McGuire and John F. Hughes;
  - Utilization of vertex shader on GPU.
  - The most promising technique for real-time feature edges detection.
The Ends

- Thanks for your patience.

- Question?