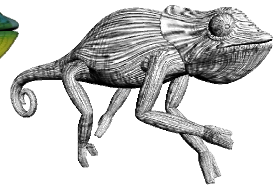
Non-Photorealistic Rendering

Mike Eißele

Institute of Visualization and Interactive Systems
University of Stuttgart

Introduction

realistic rendering



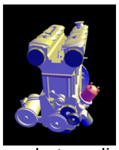
non-photorealistic

Tutorial T7: Programming Graphics Hardware | EG 03 | Non-Photorealistic Rendering Mike Eißele | VIS Group, University of Stuttgart

Introduction

- Photorealistic rendering
  - Resemble the output of a photographic camera
- Non-photorealistic rendering (NPR)
  - Convey meaning and shape
  - Emphasize important parts
  - Mimic artistic rendering

realistic rendering



non-photorealistic

Tutorial T7: Programming Graphics Hardware | EG 03 | Non-Photorealistic Rendering Mike Eißele | VIS Group, University of Stuttgart

Introduction

- Typical drawing styles for NPR
  - Pen-and-ink illustration
  - Stipple rendering
  - Tone shading
  - Cartoon rendering
- Further reading on NPR:
  - SIGGRAPH 1999 Course #17: *Non-Photorealistic Rendering*
  - Gooch & Gooch: *Non-Photorealistic Rendering* [2001]
  - Strothotte & Schlechtweg: *Non-Photorealistic Computer Graphics* [2002]

Tutorial T7: Programming Graphics Hardware | EG 03 | Non-Photorealistic Rendering Mike Eißele | VIS Group, University of Stuttgart

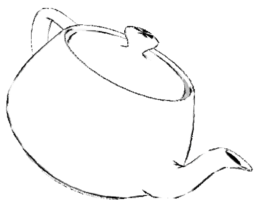
Introduction

- Focus of this talk
  - Silhouette rendering
  - Cartoon shading
  - Hatching
  - Charcoal rendering
  - Image-space filter operations
  - Dither screens

Tutorial T7: Programming Graphics Hardware | EG 03 | Non-Photorealistic Rendering Mike Eißele | VIS Group, University of Stuttgart

Silhouette Rendering

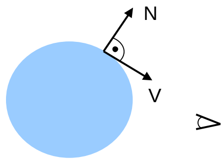
- Often necessary for non-photorealistic renderings
- Closure of the object
- Widely used for cartoon rendering



Tutorial T7: Programming Graphics Hardware | EG 03 | Non-Photorealistic Rendering Mike Eißele | VIS Group, University of Stuttgart

Silhouette Rendering

- "Manually" detect silhouette
  - Edges where adjoining faces are differently culled
- For smooth objects:
  - Normals of silhouette points and the view vector are "perpendicular"
  - Easy implementation on graphics hardware



Silhouette Rendering via HLSL Shaders

- The vertex position in world space equals -V
- Vertex Shader transfers world space position and world space normal to the Pixel Shader

```
// VERTEX SHADER:
// transform model position and output position
float3 wsPos = mul(float4(Pos,1), ModelView);
Out.Pos = mul(float4(wsPos,1), Projection);
// transform Normal with inverse Model View
float3 transNorm = mul(Normal, InvModelView);
// outpot normal to the pixel shader
Out.Normal = transNorm;
// outpot position to the pixel shader
Out.wsPos = wsPos;
```


Silhouette Rendering via HLSL Shaders

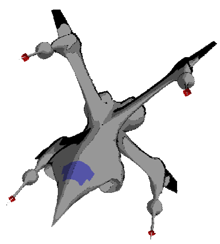
- Pixel Shader renormalizes N and V (position)
- Calculates angle between N and V
- Compare angle to threshold

```
// PIXEL SHADER:
// renormalize the normal and position vector
float3 normNormal = normalize(In.Normal);
float3 normPos = normalize(In.wsPos);
// compute angle between N and V
float angle = dot(normNormal, normPos);
// test if fragment is in silhouette and output
float4 color = 1.0f;
if (angle < 0.1f) color = 0.0f;
return color;
```


Silhouette Rendering – Demo

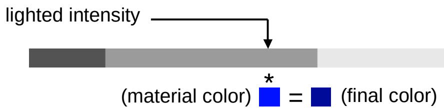
Cartoon Shading

- Black outline
- Flat or limited color shading



Cartoon Shading

- General idea from A. Lake [Lake et al. NPAR00]
- Diffuse Lighting to generate intensity
- Index in 1D Texture with lighted intensity
- Modulate texture color with surface material
- Add silhouette rendering for black outlines



Cartoon Shading via HLSL Shaders

- Same Vertex Shader as for silhouette rendering
- Vertex Shader transfers world space position and world space normal to the Pixel Shader

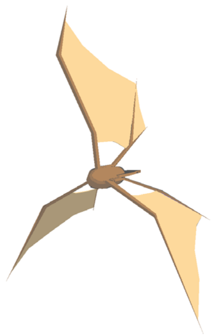
```
// VERTEX SHADER:
// transform model position and output position
float3 wsPos = mul(float4(Pos,1), ModelView);
Out.Pos = mul(float4(wsPos,1), Projection);
// transform Normal with inverse Model View
float3 transNorm = mul(Normal, InvModelView);
// outpot normal to the pixel shader
Out.Normal = transNorm;
// outpot position to the pixel shader
Out.wsPos = wsPos;
```


Cartoon Shading via HLSL Shaders

- Pixel Shader renormalizes N and V (position)
- Calculates diffuse lighting used for texture lookup
- Darken if fragment is part of a silhouette

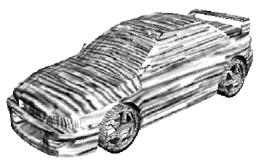
```
// PIXEL SHADER:
// renormalize the normal
float3 normNormal = normalize(In.Normal);
// calculate diffuse lighting
float lightInt = dot(normNormal, LightDir);
// lookup comic intensity texture and modulate
float4 color = tex1D(ComicTexture,lightInt);
color = color * DiffuseMaterial;
// darken color if fragment is part of silhouette
… same as silhouette rendering …
```


Cartoon Shading – Demo



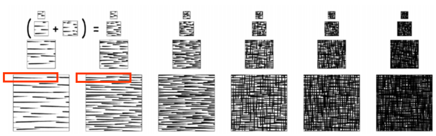
Hatching

- Stroke-base rendering of 3D models
- Density of strokes represents lighting and material
- Direction of strokes conveys the object's shape



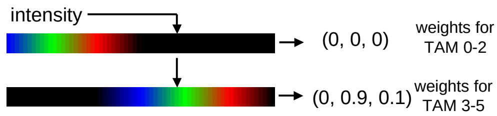
Hatching

- Method from E. Praun [Praun et al. SIGGRAPH01]
- Needs adequate texture coordinates
- Using tonal art maps (TAM) to preserve tonal and spatial continuity
- Each successive level contains all the hatch lines from the previous levels



Hatching

- All six TAM columns are encoded in two mip-map textures
- To preserve continuity, the TAMs are blended according to the lighting
- Per-pixel intensity is used for two 1D texture lookups to get the TAM weights



Hatching

- Each TAM is modulated with its corresponding weight
- The modulated TAMs are summed to produce the final color
- An additional threshold is used to filter out light gray values
- With adapted TAMs limited stippling illustrations are also possible

Hatching via HLSL

- Vertex Shader just pipes the normal and the texture coordinates of the model to the Pixel Shader
- All work is done in the Pixel Shader
- Evaluate the lighting and lookup the TAM weights

```
// PIXEL SHADER:
// renormalize the normal
float3 normNormal = normalize(In.Normal);
// calculate diffuse lighting
float lightInt = dot(normNormal, LightDir);
// load tam weights
float3 tamWeight02 = tex1D(Weight02,1-lightInt);
float3 tamWeight35 = tex1D(Weight35,1-lightInt);
```

Tutorial T7: Programming Graphics Hardware — EG 03 — Non-Photorealistic Rendering, Mike Eißele — VIS Group, University of Stuttgart

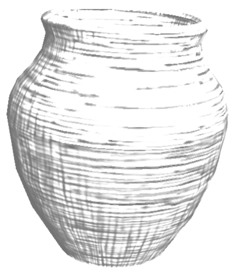
Hatching via HLSL

- Load the tonal art maps
- Modulate the TAMs and sum up the results
- Use a threshold to remove light gray values

```
// load tam textures
float3 tamInt02 = tex2D(TamTex02,In.Tex0);
float3 tamInt35 = tex2D(TamTex35,In.Tex0);
float color = dot(tamWeight02, tamInt02) +
              dot(tamWeight35, tamInt35);
if (color > GrayThreshold) color = 1.0f;
return color;
```

Tutorial T7: Programming Graphics Hardware — EG 03 — Non-Photorealistic Rendering, Mike Eißele — VIS Group, University of Stuttgart

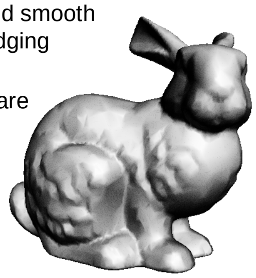
Hatching via HLSL – Demo



Tutorial T7: Programming Graphics Hardware — EG 03 — Non-Photorealistic Rendering, Mike Eißele — VIS Group, University of Stuttgart

Charcoal Rendering

- Charcoal is extremely limited in dynamic range
- Broad grainy strokes and smooth tonal variations by smudging the charcoal
- Sometimes silhouettes are not drawn to achieve the “closure effect”



Tutorial T7: Programming Graphics Hardware — EG 03 — Non-Photorealistic Rendering, Mike Eißele — VIS Group, University of Stuttgart

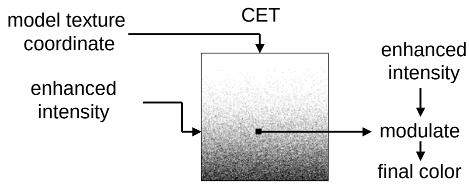
Charcoal Rendering

- Method from A. Majumder [Majumder et al. NPAR02]
- Concept is based on a contrast enhancement operator
- Uses a single contrast enhanced texture (CET) for tonal variation
- Contrast enhanced intensity is used to index into the CET for the effect of grainy strokes
- Smudging via blending the textured model with a contrast enhanced model

Tutorial T7: Programming Graphics Hardware — EG 03 — Non-Photorealistic Rendering, Mike Eißele — VIS Group, University of Stuttgart

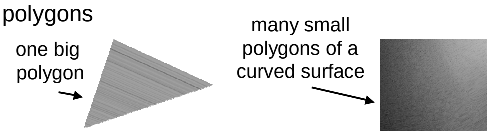
Charcoal Rendering

- To texture the model with the CET, texture coordinates are needed
- The y-component is the intensity
- The x-component comes from the model's texture coordinates



Charcoal Rendering

- Problem: CET textured polygon that is equally lit
- Needs especially arranged texture coordinates
  - Texture coordinate is either 0 or 1
  - A single polygon must not have three equal texture coordinates
- Still not fully resolved
- Good results for curved surfaces with „small" polygons



Charcoal Rendering via HLSL

- Vertex Shader just pipes the normals and texture coordinates of the model to the Pixel Shader
- Pixel Shader evaluates lighting and enhances contrast
- Texture coordinates for the CET lookup are generated

```
// PIXEL SHADER:
// renormalize the normal
float3 normNormal = normalize(In.Normal);
// calculate diffuse lighting
float lightInt = dot(normNormal, LightDir);
// enhance contrast of lighted intensity
float enhLightInt = pow(lightInt, 1.7f);
// generate texture coordinates for CET
float2 cetCoords = float2(In.Tex0.x, enhLightInt);
```


Charcoal Rendering via HLSL

- Perform lookup in CET texture
- Modulate the contrast enhanced intensity with the CET intensity to achieve smudging

```
// lookup in CET texture
float cetInt = tex2D(CetTex, cetCoords);
// modulating CET and enhanced intensity
float color = cetInt * enhLightInt;
// return final fragment color
return color;
```


Charcoal Rendering via HLSL – Demo

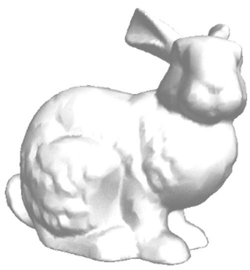


Image-Space Filter Operations

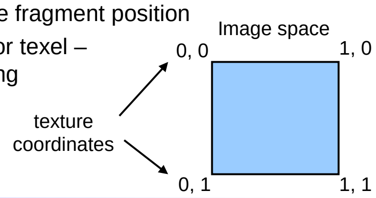
- The Computer Vision community provides many image-space algorithms
- Only simple ones can be implemented on graphics hardware
- Image-space methods are nearly always fill-rate limited

Image-Space Filter Operations

- Filter operations use data from adjacent neighbors
- Therefore attribute data must be rendered before
- On today's graphics hardware this is typically done via Render2Texture
- Texture lookup count and available texture coordinates limit the number of possible filter samples
- Many filter operations have the same general application flow

Image-Space Filter Operations

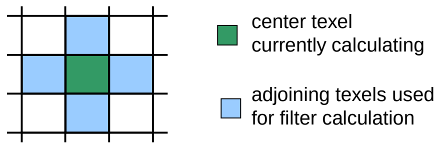
- First the attribute which should be filtered is rendered into a texture
- The filter pass just renders one quad on the entire image space
- Texture coordinates setup to represent the image-space fragment position
- Watch out for texel – pixel mapping



Tutorial T7: Programming Graphics Hardware EG03 Non-Photorealistic Rendering Mike Eißele VIS Group, University of Stuttgart

Image-Space Filter Operations

- Image-space position is used for looking up the center texel
- The address of additional texels for the filter are calculated in a Vertex Shader or a Pixel Shader
- For the address calculation the texel size must be known ⇒ resolution dependent



Tutorial T7: Programming Graphics Hardware EG03 Non-Photorealistic Rendering Mike Eißele VIS Group, University of Stuttgart

Image-Space Filter Operations via HLSL

- Vertex Shader calculates the positions of the texels and stores them in texture coordinates
- Constant register holds texel size in x and y direction
- Limit: max. 8 texture coordinate sets

```
// VERTEX SHADER:
// output position for image space quad rendering
Out.Pos = float4(Pos.xy, 1.0f, 1.0f);
// output center, left, … texel coordinates
Out.Tex0 = Tex0.xy;
Out.Tex1 = Tex0.xy + float2(-texelSize.x, 0.0f);
Out.Tex2 = Tex0.xy + float2(texelSize.x, 0.0f);
… same for upper and lower texel coordinates …
```

Tutorial T7: Programming Graphics Hardware EG03 Non-Photorealistic Rendering Mike Eißele VIS Group, University of Stuttgart

Image-Space Filter Operations via HLSL

- Pixel Shader loads all texels
- Perform arbitrary calculations based on neighboring texels
- Result can only be written to current fragment position

```
// PIXEL SHADER:
// load center, left, right, … texel
float3 center = tex2D(Attribute, In.Tex0);
float3 left = tex2D(Attribute, In.Tex1);
float3 right = tex2D(Attribute, In.Tex2);
… load other texel as well …
// perform filter operation and output result
return calcFilter(center, left, right, …);
```

Tutorial T7: Programming Graphics Hardware EG03 Non-Photorealistic Rendering Mike Eißele VIS Group, University of Stuttgart

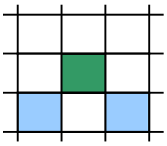
Filter Operation: Edge Detection

- Edge detection can be applied to different attributes
  - Final color (including texture and lighting)
  - Material color / object ID (raw material color)
  - Normals (in world space)
  - Depth (in eye space)
- Edge detection on a single attribute often misses some edges
- Detect edges on multiple attributes and combine
- Some graphic hardware support multiple render targets

Tutorial T7: Programming Graphics Hardware EG03 Non-Photorealistic Rendering Mike Eißele VIS Group, University of Stuttgart

Filter Operation: Edge Detection

- Can also be used for silhouette rendering
- Simplest algorithm: „Robert's Cross“



- center texel currently calculating
- adjoining texels used for filter calculation

- Build the absolute difference between every adjoining texel and the center texel
- Sum up and scale the result
- Can be applied to arbitrary scalar data

Filter Operation: Edge Detection via HLSL

– Vertex Shader calculates the texel addresses of adjoining texels

```
// VERTEX SHADER:
// output position for image space quad rendering
Out.Pos = float4(Pos.xy, 1.0f, 1.0f);
// texture coordinate for center texel
Out.Tex0 = Tex0.xy;
// texture coordinate for lower left texel
Out.Tex1 = Tex0.xy +
           float2(-texelSize.x, texelSize.y);
// texture coordinate for lower left texel
Out.Tex1 = Tex0.xy + texelSize.xy;
```


Filter Operation: Edge Detection via HLSL

– Pixel Shader loads data from all texels

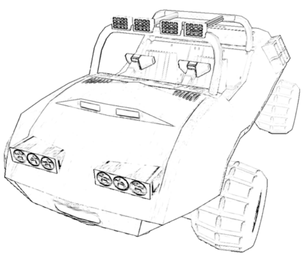
```
// PIXEL SHADER:
// define RGB weights
float3 rgbWeights = float3(0.3f, 0.59f, 0.11f);
// load texels
float3 center = tex2D(Attribute, In.Tex0);
float3 lowLeft = tex2D(Attribute, In.Tex1);
float3 lowRight = tex2D(Attribute, In.Tex2);
```


Filter Operation: Edge Detection via HLSL

– Here: transform each value to an intensity
– Evaluate the filter function
– Scale result or use a threshold

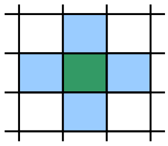
```
// transform to intensity
float centerInt = dot(rgbWeights, center);
float lowLeftInt = dot(rgbWeights, lowLeft);
float lowRightInt = dot(rgbWeights, lowRight);
// difference between low left and center texel
float diffLowLeft = abs(centerInt - lowLeftInt);
// difference between low right and center texel
float diffLowRight = abs(centerInt - lowRightInt);
// scale output and return fragment color
return edgeScale * (diffLowLeft + diffLowRight);
```


Filter Operation: Edge Detection – Demo



Filter Operation: Dilation

- Edge filter produces only “thin” edges
- Many NPR techniques require clearly marked outlines
- Use a 5-tab filter add all texels and use a threshold



- center texel currently calculating
- adjoining texels used for filter calculation

Filter Operation: Dilation via HLSL

- Vertex Shader just sets coordinates for all texels
- Pixel Shader loads texel data and sums it up

```
// PIXEL SHADER:
// load texels
float center = tex2D(Attribute, In.Tex0);
float left = tex2D(Attribute, In.Tex1);
float right = tex2D(Attribute, In.Tex2);
float upper = tex2D(Attribute, In.Tex3);
float lower = tex2D(Attribute, In.Tex4);
// sum up the data and apply a threshold
float weight = center + left +
               right + upper + lower;
```


Filter Operation: Dilation via HLSL

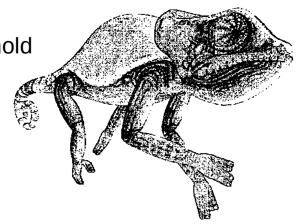
- Threshold value and default result is dependent what we want to dilate: 0.0 or 1.0
- Here we are interested in the black (0.0) values

```
// set default result (default is white)
float result = 1.0f;
// apply threshold function
if (weight < Threshold) result = 0.0f;
// output result
return result;
```


Filter Operation: Dilation via HLSL – Demo

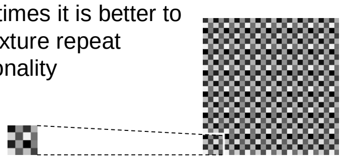
Dither Screens

- Based on idea from O. Veryovka [Veryovka et al. GI99]
- Predefine a threshold per pixel
- Use repeated threshold matrices for screening effects
- Don't use threshold matrices to avoid patterns (stippling)



Dither Screens

- Create a texture with dimensions of the output resolution
- Fill this texture with intensity encoded thresholds either via repeating dither matrices or make all dither matrices different
- Sometimes it is better to use texture repeat functionality



single dither matrix — dither screen

Dither Screens via HLSL

- Vertex Shader transforms the vertices and normals of the model
- Calculates texture coordinates for accessing the threshold texture in the pixel shader unit

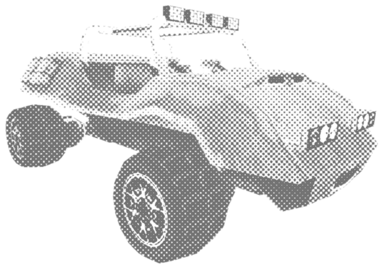
```
// VERTEX SHADER:
// transform model and output position, normal
float3 wsPos = mul(float4(Pos,1), ModelView);
float4 projPos = mul(float4(wsPos,1), Projection);
Out.Pos = projPos;
Out.Normal = mul(Normal, InvModelView);
// calculate image-space position used for
// threshold texture lookup in pixel shader
Out.Tex0.xy = (projPos.xy/projPos.w) * 0.5 + 0.5;
```

Dither Screens via HLSL

- Pixel Shader evaluates the lighting
- Reads threshold from dither screen
- Applies threshold

```
// PIXEL SHADER:
// evaluate lighting
float3 normNormal = normalize(In.Normal);
float lightInt = dot(normNormal, LightDir);
// read threshold and apply threshold
float threshold = tex2D(DitherScreen, In.Tex0);
float intensity = 1.0f;
if (lightInt < Threshold) intensity = 0.0f;
return intensity;
```


Dither Screens via HLSL – Demo



Tutorial T7: Programming Graphics Hardware · EG 03 · Non-Photorealistic Rendering, Mike Eißele · VIS Group, University of Stuttgart

Techniques we could not show

[Hertzmann SIGGRAPH98]

[DeCarlo et al. SIGGRAPH02]

Tutorial T7: Programming Graphics Hardware · EG 03 · Non-Photorealistic Rendering, Mike Eißele · VIS Group, University of Stuttgart

References

[Lake et al. 2000] A. Lake, C. Marshall, M. Harris, M. Blackstein. Stylized Rendering Techniques For Scalable Real-Time 3D Animation. In NPAR *2000 Proceedings, pages 13-20*.

[Praun et al. 2001] E. Praun. Real-Time Hatching. In Proceedings of ACM *SIGGRAPH 2001 Conference Proceedings*, pages 579-584.

[Majumder et al. 1998] A. Majumder, M. Gopi. Hardware Accelerated Real Time Charcoal Rendering. In NPAR *2002 Proceedings, pages 59-66*.

[Veryovka et al. 1999] O. Veryovka, J. Buchanan. Halftoning with Image-Based Dither Screens. In Proceedings of Graphics Interface 99, pages 167-174.

[Gooch & Gooch 2001]: B. Gooch, A. Gooch. *Non-Photorealistic Rendering*. A. K. Peters, Natick, 2001.

[SIGGRAPH 1999 Course 17] S. Green, D. Salesin, S. Schofield, A. Hertzmann, P. Litwinowicz, A. Gooch, C. Curtis, B. Gooch. *SIGGRAPH 1999 Course 17: Non-Photorealistic Rendering*.

[Strothotte & Schlechtweg 2002] T. Strothotte, S. Schlechtweg. *Non-Photorealistic Computer Graphics*. Morgan Kaufmann Publishers, 2002.

Tutorial T7: Programming Graphics Hardware · EG 03 · Non-Photorealistic Rendering, Mike Eißele · VIS Group, University of Stuttgart