Real-Time Pencil Rendering

Juan Pacheco
Digipen Institute of Technology Europe-Bilbao

Abstract

This paper contains the implementation of a non photorealistic real time rendering technique that simulates pencil drawings of 3D scenes. This implementation is based in the paper done by the POSTECH university [Lee et al. 2006] were they describe pencil rendering and the interaction among curvatures, papers and pencil materials. The objective of this paper is to be able to simulate human errors when drawing to be able to render objects that look like they were hand-drawn. This implementation contains a pipeline divided in edge detection and composition, inner surface rendering with pencil materials, a composition pass for the resulting two textures and a lighting pass for the two textures. The technique is implemented almost in GPU using OpenGL, except for a preprocessing part that is precomputed in CPU.

1 The Pipeline

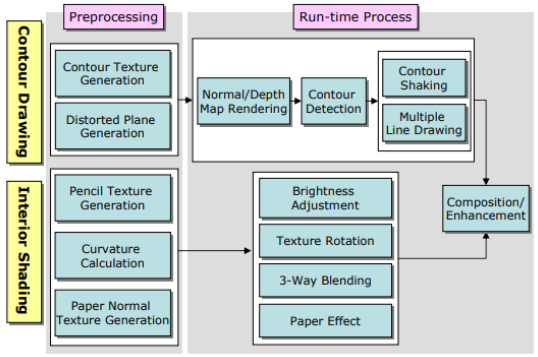

Figure 1: Original Pipeline [Lee et al. 2006]

On one hand, in the original paper the pipeline (see Figure 1) divides the work into two parts also, the contour pass and the interior pass, and also divides each part into a preprocessing step and a run-time process. On the other hand, my implementation (see Figure 2) follows almost every step on that pipeline but erases some parts. First of all, the contour texture generation does not appear in the pipeline as it is not specified in the original paper, they use it but they do not explain where it comes from, so for my technique I decided to substitute that part with the pencil texture generation. Secondly, the paper normal texture generation step that they propose, is based on generating noise from an algorithm like Perlin noise or Worley noise to generate a heightmap and finally to convert it to a normal map. In my case I decided to not implement that part as I think it was not as relevant to the technique itself, so I decided to download some normal map textures of papers for the implementation. Finally, in the interior shading part, they perform a brightness adjustment step to make distinctions between bright and dark spots by simply using a square root in the final color, however, after implementing it, I did not like the result, so I decided to leave it out.

Apart from that, my pipeline includes an extra step where lighting is computed in a simple way to make a much better looking result.

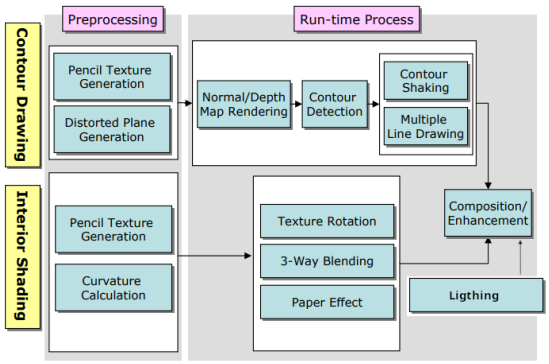

Figure 2: My modified Pipeline

2 Edge detection

First of all, in this technique we will apply and edge detection filter [Nienhaus and Doellner 2003] in image space that will use the normal and depth buffer. The idea of this filter is to detect discontinuities in the normal buffer to store them in the RGB channels of a texture and to detect discontinuities in the depth buffer to store them in the A channel of the same texture.

A	B	C
D	X	E
F	G	H

Figure 3: X Neighborhood [Nienhaus and Doellner 2003]

The original paper describes a neighborhood (see Figure 3) of 3x3 around the pixel that we are shading.
Also, three types of edges are described: the silhouette edges that are edges shared by a front-facing triangle and a back-facing one, the border edges that are edges to only one triangle, and the crease edges that are edges shared between two front-facing triangles.

The first two appear in discontinuities of the depth buffer and the last one appears on discontinuities of the normal buffer. For detecting these kind of edges the following formulas are used.

$$I_{RGB} = \frac{1}{2} \cdot (dot(A_N, H_N) + dot(C_N, F_N))$$

$$I_A = (1 - \frac{1}{2} \cdot |A_Z - H_Z|)^2 \cdot (1 - \frac{1}{2} \cdot |C_Z - F_Z|)^2$$

The equations will result in gray edges, but as later we will want to perform additive blending we will invert the RGB channel.

Finally, for the RGB channel there are two possibilities, computing it using interpolated normals or computing it using face normals (extrapolating them from the depth buffer). If we use face normals we will be sure that our geometry is right and taken into account that the curvature is analyzed with triangles itself, it makes sense to use it them, however, interpolated normals will bring details of the normal mapping to the scene, which could be desirable (see Figures 12-13).

3 Contour Shaking and Distortion Plane

When drawing the contours, one of the effects that we want to achieve is to mimic the human error of drawing the edges not perfectly straight. For this we will use the sine wave equation [Lee et al. 2006] to add errors to the UVs.

The idea is to generate something called the “Distorted plane” (see Figure 4) in a preprocessing step and then sample it when drawing the contour to take into account the generated error. For the creation of it, we will first define the parameters we want for the sine equation (amplitude, change of phase, etc.), then

we will divide the screen space into equally sized rectangles and to each rectangle we will assign the corresponding value of inputting the respective x and y to the sine wave. In this technique the distortion plane is computed in a compute shader to make it faster, but a fragment shader will work too.

The results generated by this plane when applying it to the edges (see Figure 5) make the contours to look distorted and fulfills the proposed idea.

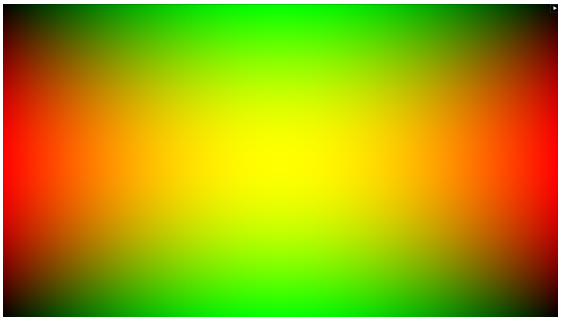

Figure 4: Distortion plane visually

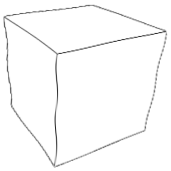

Figure 5: Effect of distortion

4 Multiple Contours

The last important part of the contours, is to imitate the tendency of drawing over and edge multiple times when using a pencil, for that, the idea is very simple. We will generate an specific amount of distortion planes and we will apply each one of those planes to the contours to generate new images. Then, we will overlap those textures with additive blending to showcase that human error (see Figure 6). The original paper recommends to choose between 3-5 planes, and after some tests, the result looks good.

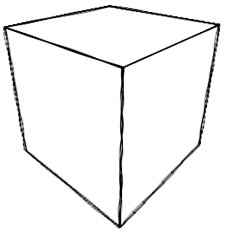

Figure 6: Multiple contours

Finally, for giving the edges that pencil material, we will access the generated pencil texture (see Section 5) each time we distort the contours at the depth we got in the lighting computation (see Section 9).

5 Pencil Texture Generation

Once again, when drawing an interior surface on paper, we tend to draw multiple overlapping strokes in the same place if we want a darker result and we will draw less strokes in sections where we want a lighter result. To modulate this behavior, we will create a 3D texture of 32 layers [Lee et al. 2006], the first one being the whitest and the latest one the darkest.

The idea is simple, we will have a single stroke texture (in my case generated by Paint), and we will draw it multiple times over each layer. However, we have to take into account that the first layer should be entirely white, so we will start drawing strokes at layer 1 instead of 0.

When drawing into a new layer, we will first copy the previous layer onto it, then we will draw a specific amount of strokes in random Y positions of the texture and we will add to them a little angle perturbation to not make them perfectly

straight. The blending of the stroke and texture goes in the following way:

```
//Get color of the previous
layer
vec3 previous_texture_color =
texture(previous_texture,
vec3(new_uvs,
previous_texture_depth)).rgb;

//Maximum stroke darkness that
can be increased
float ca =
previous_texture_color.r *
(1.f - pencil_color.r);


//If the pixel is white enough,
the darkness increase is reduced
if(previous_texture_color.r>0.9)
{
    ca *= white_preservation;
}

//Update the texture color
float new_texture_color =
previous_texture_color.r -
stroke_darkness * ca;
```

In the if statement, we can see how if the pixel is white enough, we will multiply the maximum darkness increase by a value that is a parameter that is usually between 0.1 and 0.3. We do this because usually in drawings when a zone is white after some strokes, it tends to remain white for the entire drawing.

The result generated in this step (see Figure 7) looks pretty accurate to what a real pencil material could be, but is worth mentioning that it is highly dependant on the single stroke texture that we use.

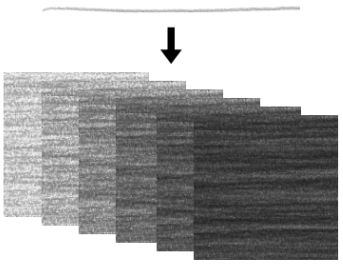

Figure 7: Pencil texture [Lee et al. 2006]

6 Curvature Analysis

When shading a surface, we tend to draw the strokes along the curvature of an object, so this section will explain how this is implemented.

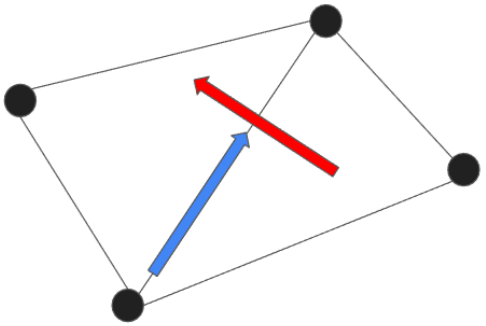

Figure 8: Curvatures along an edge

First of all, we will define that each edge will contain two curvatures, the maximum one and the minimum one. These curvatures tend to go in the direction of the edge itself and in the perpendicular direction (see Figure 8). For getting these curvatures, the model followed on this paper is the tensor field computation estimation [Alliez et al. 2003]. The idea is to estimate for each vertex of the mesh a tensor, then get the eigenvalues of that tensor and finally from those eigenvalues get the curvatures. For that, the following formula is used.

$$\boldsymbol{T}(v) = |B| - 1 \cdot \Sigma\, \beta(e) \cdot |e \cap B| \cdot \hat{e} \cdot e^{T}$$

The elements of this formula consist of:

- v : The input vertex.
- |B| : Area of the neighborhood of edges around v.
- e : An edge of the neighborhood B.
- $\beta(e)$: Signed angle between the normals shared by the edge e.
- $|e \cap B|$: Length of e inside neighborhood B.
- $\hat{e}$: Unitary vector in the direction of e.

And the pseudocode the algorithm should follow is the next one:

```
r = BBscale * 0.01;
for(vertex in mesh)
{
   Create sphere of radius r
   around vertex;
   Get edges inside the sphere;
   Compute the area of that
   neighborhood;
   for(edge in neighborhood)
   {
      Compute the summation of
      the previous equation;
   }

   Compute the tensor at vertex;
   Get the eigenvalues and
   eigenvectors;
}
```

After computing the eigenvalues we will get 3 results, the eigenvector associated to the minimum one will be the normal of the vertex, the eigenvector associated to the maximum one will be the minimum curvature vector and the other value will be associated to the minimum curvature.

Some things worth mentioning are that in the followed paper as the idea is to do an anisotropic filtering they perform a lot of steps after this one, but in my approach I decided to keep it simple and to normalize the curvatures that were computed. Then, the neighborhood that we create has the shape of a geodesic disk, so instead of computing the exact area, I did an estimation by taking the Bounding Box (BB) of the neighborhood and then compute the length of the scale and that would be the diameter of the circle. Finally, this process is very heavy computationally speaking, and it takes some time to analyze complex meshes, so an optimization that I implemented was to store the curvatures in a file and the next time the mesh is loaded it will read them from the file itself.

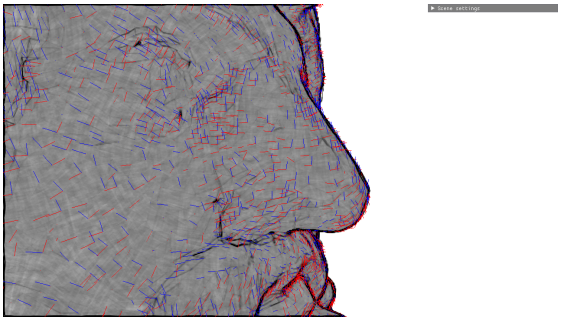

Figure 9: Curvature vectors of Serapis

7 3-Way Blending and Cross Hatching

After getting the curvature for each vertex, the idea proposed by the original paper is to draw a pencil texture for each minimum curvature of the vertices of a triangle, so we will draw in screen space the pencil texture and we will rotate it by the curvature of the respective vertex.

For that, we will assign to each vertex three curvatures, its minimum one and the other minimum curvatures of the same triangle vertices. Then we will project the

curvatures using the P * V * M matrix. Finally, we will get the angles generated by those curvatures and we will negate them to perform inverse rotations on the UV coordinates of the pencil texture to draw the pencil textures along those curvatures.

As we are drawing the pencil texture with 3 curvatures, we will get 3 directions, and if we blend them (using a weighted sum) we will get a blend of the pencil stroke in 3 different directions (see Figure 10).

After that, the possibility of cross-hatching can be implemented. Knowing also the maximum curvatures, we could do the same process but instead of a 3-way blending, we could perform a 6-way blending, so we will merge 6 different pencil textures.

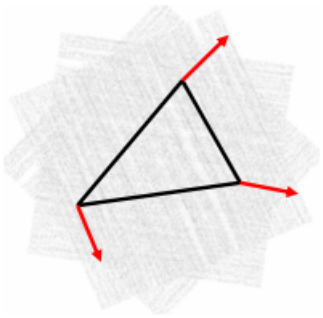

Figure 10: 3-way blending [Lee et al. 2006]

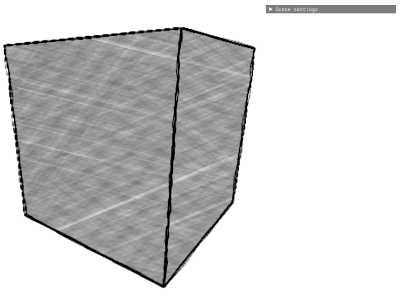

Figure 11: Cross hatching

8 Paper Interaction

This stage of the pipeline consists of trying to modulate the black lead stains that are left by a pencil when drawing on a rough paper. For that, we will get a normal map that will represent the roughness of the paper.

Once we got it, in the 3-way blending computation we will add the next formula :

$$C'_T = C_T + \mu_P \cdot dot(d, n)$$

In this formula, d represents the stroke direction, and n is the normal of the paper. With this, we achieve the effect of leaving more lead stains whenever the stroke direction is opposite to the normal of the paper. Finally, μ_P is a parameter that defines the weight of the paper (usually good between 0 and 0.1).

9 Lighting

The idea of the lighting in Real Time pencil rendering is quite simple. We will take advantage of deferred shading and we will compute two factors :

- Attenuation.
- Cosine factor.

The attenuation is computed the same way it is computed in deferred rendering is computed by :

$$CosineFactor = max(dot(N, L), 0.f)$$

The purpose of this computation is to access the pencil texture in the Z layer. If a pixel is facing directly a light we want it to look white, so it will have to access the layer 0, so when accessing the pencil

texture we will invert the given result in the lighting computation. As we are following a deferred rendering approach, this factor can be stored in a texture for optimization.

This approach works well with edges and surfaces, and it will be applied when distorting edges and in the 3-way blending of the surfaces.

10 Composition

This is the last step of the pipeline. Once we have the two textures we will have to merge them using additive blending and them invert the result one last time to get the wanted black colors. Also, we can pass the paper color as a parameter to multiply it by the merged color to not necessarily make a perfect white paper.

11 How to use the demo

The demo itself is quite simple, you can move around the scene with the mouse and the WASD keys and you have an editor window at the top right.
In that window (in the RTPR tab) you can tweak some parameters as:

- Enable/Disable
- Render max/min curvatures
- Use face/interpolated normals
- Edge overlap number
- Render contours
- Render surfaces
- Paper weight and texture
- Offsets for the lighting
- Enable/disable lighting

Then in the General tab, you will be able to change between scenes and in the Deferred rendering part, you will be able to change the number of lights.

12 Results

As we can see in the pictures, the results look very cool and pretty similar to the ones shown in the original paper. It is worth mentioning that, although it works in real-time, it looks better when the scenes are static as most of the algorithms are implemented in screen space so when moving through the scene some artifacts appear.

We can appreciate how the pencil material works and how the interaction with the paper happens correctly. Finally, we can see how although the cross-hatching works, after crossing a lot of sections the visual hatching starts to be noticed less, so that could be a little bit improved.

To see more results check from figure 12 to 15.

13 Possible Improvements

Some things that I think could be improved for my technique and that could be implemented in the future are a faster way to analyze the curvature, the addition of pencil colors to not only draw in black, the computation of the normal maps for the papers, and the implementation of shadows. Apart from that, some improvements on the cross-hatching technique could be done to be able to appreciate the result more.

Finally, other results could be explored to be able to make this technique look better in a Real-Time environment.

I am sure that all of these characteristics could be implemented in a feasible amount of time considering almost all of the work is already done.

14 Development Problems

The problems that I found when following the original papers were the next ones.

First of all the curvature analysis was a little bit difficult to understand at first because it talked about topology and I do not have a lot of knowledge in that area.

Secondly, as the papers were a little bit old, they used things of OpenGL that were not available in newer versions, so I had to rethink them to apply them in modern OpenGL

Then, as I already mentioned, the 3D texture generation is very important to have a similar result to the original paper, but still I am happy with how it ended.

Finally, some things such as the contour texture and some blendings were not defined at all, so those things had to be reimagined.

15 Conclusions

After implementing this technique, I think that it looks really realistic considering the tools that are available right now and I also think similar approaches could be expanded to develop new techniques in the non photorealistic rendering world.

Also, as already mentioned, I do not think this a good method for Real Time scenes but more for static ones, so I think it is great for programs like Blender or 3DS Max where you can do static enhanced renders.

16 References

HYUNJUN LEE, SUNGTAE KWON, SEUNGYONG LEE. 2006. Real-Time Pencil Rendering. 1-9

MARC NIENHAUS, JUERGEN DOELLNER. 2003 Edge-Enhancement - An Algorithm for Real-Time Non-Photorealistic Rendering. 3-4

PIERRE ALLIEZ, DAVID COHEN-STEINER , OLIVIER DEVILLERS, BRUNO LEVY, MATHIEU DESBRUN. 2003. Anysotropic Polygonal Remeshing.7-11a

Figure 12: Face normals

Figure 13: Interpolated normals

Figure 14: Serapis in Sponza

Figure 15: Serapis in Sponza with lighting