CS562 Final Project: Real-Time Pencil Rendering

Markel Sevilla Pelayo

DigiPen Institute of Technology Europe

Index

Contenido

1- Abstract

This report explains my journey in the implementation of a variance in real-time pencil rendering, based on the analysis conducted by Pohang University of Science and Technology located in South Korea. This project presents a modular and procedural system that tries to emulate human patterns of drawing when creating hand sketches, internal shading and erasing. Through my implementation I have tried to adapt this paper to an already created graphics pipeline with deferred shading while tackling the problems that the original implementation presents.

2- Introduction

After finishing our rendering pipeline, we were able to accurately use many techniques to create different scenes. These techniques included deferred rendering with gaussian blurred bloom, cascaded shadow maps with percentage closer filtering, decal rendering and ambient occlusion with bilateral filtering. With this project I tried to add a new mode to the already complete pipeline in which I would be able to successfully depict different hand-drawn patterns of pencil, with the possibility of expanding it to pen and ink in the future.

3- Real-Time Pencil Rendering

3.1- Pencil Rendering: Conceptually

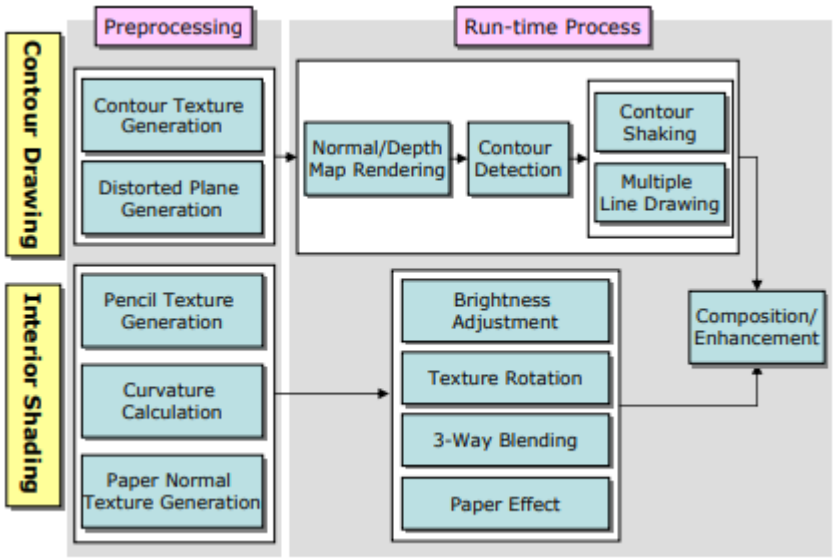
This paper tries to realistically emulate the patterns that give life to hand-made drawings by applying some techniques to the generated 3D meshes. Through these techniques we try to depict the two different parts that usually compose any pencil drawn piece: contour drawing and interior shading. We will try to create slightly inconsistent contours that subtly differ from the real edges of the 3D mesh by applying multiple shaken layers of the contours to the same image. For the interior shading we will make use of previously computed textures to give the sensation of a fake lighting effect, taking as a base a vanishing point.

3.2- Approaches to Real-Time Pencil Rendering

3.2.1- Real-Time Pencil Rendering – by POSTECH

3.2.1.1- Raw explanation of the paper- System Overview

This paper presents a system overview that separates the process into two stages: the Preprocessing and the Run-Time Process. At the same time the pipeline is differentiated into two completely independent branches that cover the Contour Drawing and the Interior Shading parts:



System Overview of the paper

3.2.1.2-Preprocessing

In the contour drawing section, the paper proposes to create a texture for the contour's color for it to resemble pencil coloring. Since contours are narrow areas, the proposed texture does not need to be of high quality, just enough to resemble the pencil tone that we want for the mapping. In the same way, we can precompute the distorted planes that we are later going to use for the multiple contour drawing and contour texture superposition.

Following this idea, for the interior shading a similar thing happens. First, we precompute the hatching textures that we are going to use to fake the shading in the later stages. The paper suggests the creation of 32 textures that go increasing in darkness by the following formula:

$$c_t' = c_t - \mu_b c_a,$$
$$c_a = c_t(1.0 - c_s),$$

In which we start with the darkness of the texture, *ct,* which depends on the index of the texture we are working on. The darkness of each pixel will depend on two factors apart from the default darkness of the texture, the darkness of the stroke itself *cs,* and the darkening factor μb. Through these factors the paper determines that the maximum amount of darkness that a pixel can be darkened for each stroke is *ca*.

In the same way to preserve white pixels and create that pencil effect, if the pixel is white enough we will maintain it as such multiplying it by the whitening factor μw.

On the other hand, to give the pencil textures the right orientation we must precompute the curvatures of the vertices and create a tensor field. This precomputation is explained in another paper [Alliez et. Al. 2003] and it describes how tensor fields work and how to generate them.

Finally, to give the paper a certain roughness and that grainy effect that the material provides we are proposed with the generation of a paper normal texture filled up with noise.

3.2.1.3 – Run-Time

For the run-time process of the contour drawing we are proposed with the rendering of normals and depth, for the latter detection of the edges and the reconstruction of the mesh itself. With this a detection of differences per pixel would be performed to extract the wanted edges based on a certain threshold.

After the contour detection, we proceed with the mapping of the edges. In the previous section we had precomputed distortion maps that will give the hand-made wavy effect to our texture. The optimum amount of distortion textures for the multiple drawing is between 3 and 5, according to the research that *PosTech* conducted. So, once we have the edge texture mapped on the different distortion maps, we just must blend them, accordingly, creating that effect of various slightly different strokes.

On the other hand, for interior shading we need to start by considering the brightness of each of the parts of the mesh. Since it's still desirable to enforce the contrast between bright and dark textures we may want to have great changes between shadow and light, however the fact is that it's more important to define the intensity variation in the brightest areas to have a better, more realistic effect. Given this case, we will use the square root of the brightness as the index for the mapping of the hatching textures. This index is the one that will fake the lighting, depending on the brightness of the pixel.

Next, similarly to the multiple contour drawing the shading effect in real drawings usually follows the geometry of the mesh we are drawing. By using the minimum curvature of each vertex we can achieve this effect simply rotating the texture in that direction. Because the geometry is defined in triangles we will use three textures for this, thus the 3-way blending. In this way, the shading will follow smoothly the geometry and the textures themselves will create soft curves that give drawings volume and shape.

Finally, to wrap everything up, we are proposed a simple idea to define the granularity of the paper, which consists of doing the dot product between the previously precomputed normal and the main direction of the strokes. This will create a sensation of roughness that complements the effect of paper well.

4- My Implementation

4.1-Preprocessing

4.1.1 – Contour Drawing

- **Contour Texture Generation:** During the implementation of this project I did not find any reason for which the texture for the contours to be generated could affect positively to the outcome. The paper itself states that this texture can be simpler due to the narrowness of the contours, that won't let the details shine. Considering this and the fact that my application is extremely modular to the point in which I constantly play with the values of the graphite's tone I deemed it irrelevant for my implementation, and chose to get rid of it. We can see the value with which I modulate the color in the next line of code:

```
vec4 newCol =  vec4( 1.f -  pencilTone * texture(edgeMap, distortedUVs).xyz, 1.0);
```

- **Distorted Plane Generation:** This is an indispensable step for the pipeline. In this pass we generate the planes that we will later use to map the detected edges and create an image in which several strokes are attempted for the same contour. For this we perform a screen space pass in which we simply create textures that contain distorted UVs stored as colors. The paper suggests to divide the screen in rectangles to localize these distortions, however when putting it into practice more segments tend to affect negatively to the look of the output. In the same way, if we choose to have different random variables for each segment, the result will deteriorate considerably.
  The distortions are bounded by a limit so that they still resemble the mesh that we are trying to draw. The following function generates the offsets that the map will store:

```
xOffset = (coeff.x) * cos( coeff.y * UVs.x + coeff.z);
yOffset = (coeff.x) * sin( coeff.y * UVs.y + coeff.z);
```

The coefficients used in these trigonometric functions are user defined parameters that multiplied by previously computed random variables generate the distortions that we look for.

4.1.2 – Interior Shading

- **Pencil Texture Generation:** This step is the most expensive one in terms of code and time, that's why it is imperative to do it just once. During this pass we compute all the hatching textures that we will later use to create a fake sensation of lighting in the drawing. Although the paper suggests us to create 32 hatching textures, in practice computing less will most probably not affect the result. Also, although we generate textures until the strokes are pitch black, the best-looking results are usually given by a maximum darkness of 40%, the last 60% of textures being too dark.

  Following the algorithm that I explained in the previous section, we would get the following code:

```
// we will check all the strokes randomly generated
for (int i = 0; i < strokes; i++)
{
    // generate a stroke from one side to another
    vec2 shadedPoint = drawStroke(UVs, vec2(0.f, strokePositions[i]), randomFactors[i].xy);

    //if we don't have a stroke in this specific uv just continue
    if( int(floor(shadedPoint.y * windowSize.y)) ==  int(floor(UVs.y * windowSize.y)))
    {
        // Compute stroke darkness modulation

        if (ct > 0.9)
        {
            ca *= mu_w; // Preserve white pixels
        }

        ct -= mu_b * ca;
    }
}
```

In which we would have our base darkness and darkening factors of the following:

```
// we check in which texture we are to detect the darkness
// 32 textures
float ct = 1.0 - (toneIndex / 32.f);
float ca = ct * (1.0 - cs);
```

```
uniform float mu_b = 0.1;  // Darkness increase factor (0.1 - 0.3)
uniform float mu_w = 0.5;  // White preservation factor (0.3 - 0.5)
const uniform float cs = 1.f / 32.f; // color of the softest stroke
```

With *toneIndex* being the index of the hatching texture that we are working on.
In the same way we select the pixel by creating a perturbated stroke:

```
// Function to draw a single stroke
vec2 drawStroke(vec2 uv, vec2 strokePos, vec2 randomFact)
{
    //take the point in which te uv would be in the x
    vec2 point = mix(strokePos, vec2(1.0, strokePos.y), uv.x);

    vec2 dir = normalize( vec2(1.0, strokePos.y) - vec2(0.0, strokePos.y) );

    vec2 perp = vec2(-dir.y, dir.x); // perpendicular perturbation :)

    //sinusoidal func for soft curve
    float perturbation =  sin( randomFact.x * uv.x * frequency * 2.0 * PI) * amplitude * randomFact.y;

    return point + perp * perturbation;  // Creates a perturbed point
}
```

For this we upload two SSBOs containing the starting positions of the strokes, and the random factors that we will use to perturbate said strokes:

```
layout(std430, binding = 0) buffer StrokeBuffer{
    float strokePositions[]; // use sssbo for hatching lines
};
layout(std430, binding = 1) buffer RandomBuffer{
    vec2 randomFactors[]; // use sssbo for hatching lines
};
```

Lastly, as a personal choice, all the textures start with the same base to ensure tileability when shading, however, as the darkness increases new strokes appear in the darkest textures. This means that if the textures start with 1000 strokes, while the lightest texture has 1000 strokes, the darkest one will display 1310, that is 1000 (the base stroke number) + 31(the index of the texture) * 10(100th of the base stroke number):

```
shader->SetUniform("strokes", hatchLines + i * hatchLines / 100);
```

- **Curvature Calculation:** This pass aims to generate the minimum curvatures of each vertex to later use them as a basis to rotate the textures. The paper does not explain much but it redirects the reader to another paper that extensively explains how to calculate the minimum and maximum curvatures of each vertex in a mesh through tensor fields. However, it proved to be harder than I thought, and I was not able to correctly implement it. In my attempts I tried to approximate it through a simplification of the tensor fields and through the descending angle algorithm without success.
- **Paper Normal Texture Generation:** This pass tries to give the final output the sensation of granularity to emulate the material of paper. For this we generate a per pixel noise texture in which we store screen space normals that define the roughness of the paper. Although in the paper we are proposed with the generation of a 2D texture for this, we will use an SSBO that serves the same purpose:

```
layout(std430, binding = 2) buffer PencilNoiseBuffer{
    vec2 pencilNoise[]; // use sssbo for hatching lines
};
```

4.2- Run-Time Process

4.2.1 – Contour Drawing

- **Normal/Depth Rendering:** Our pipeline already made use of the Geometry Buffer to store both depth and normal textures. In this case we stored them for deferred shading, but since in this case we will fake all lighting and shadowing, storing just those textures would be sufficient in any other approach.
- **Contour Detection:** For the later edge detection pass we will need to have some information to generate the contours of the mesh. In the paper we are advised to use a technique that combines both depth and normal differences as explained in [*Isenberg et al. 2003; Nienhaus and Doellner 2003]*, to detect both exterior and interior edges that are differentiated by the z-value. Although the general thought would be that just doing a simple differential between the neighboring four pixels of the center would be enough, in practice it proved to be a bit underwhelming. Due to this I chose to use an 8-direction Sobel operator to detect edges:

```
// Sample normals from the normal map
vec3 normal00 = normalize(texture(normalMap, UVs + texelSize * vec2(-1, -1)).rgb * 2.0 - 1.0);
vec3 normal01 = normalize(texture(normalMap, UVs + texelSize * vec2(-1,  0)).rgb * 2.0 - 1.0);
vec3 normal02 = normalize(texture(normalMap, UVs + texelSize * vec2(-1,  1)).rgb * 2.0 - 1.0);

vec3 normal10 = normalize(texture(normalMap, UVs + texelSize * vec2( 0, -1)).rgb * 2.0 - 1.0);
vec3 normal11 = normalize(texture(normalMap, UVs + texelSize * vec2( 0,  0)).rgb * 2.0 - 1.0);  // Center pixel
vec3 normal12 = normalize(texture(normalMap, UVs + texelSize * vec2( 0,  1)).rgb * 2.0 - 1.0);

vec3 normal20 = normalize(texture(normalMap, UVs + texelSize * vec2( 1, -1)).rgb * 2.0 - 1.0);
vec3 normal21 = normalize(texture(normalMap, UVs + texelSize * vec2( 1,  0)).rgb * 2.0 - 1.0);
vec3 normal22 = normalize(texture(normalMap, UVs + texelSize * vec2( 1,  1)).rgb * 2.0 - 1.0);

// Sample depth from the normal map
float depth00 = linearizeDepth((texture(depthMap, UVs + texelSize * vec2(-1, -1)).r ));
float depth01 = linearizeDepth((texture(depthMap, UVs + texelSize * vec2(-1,  0)).r ));
float depth02 = linearizeDepth((texture(depthMap, UVs + texelSize * vec2(-1,  1)).r ));

float depth10 = linearizeDepth((texture(depthMap, UVs + texelSize * vec2( 0, -1)).r));
float depth11 = linearizeDepth((texture(depthMap, UVs + texelSize * vec2( 0,  0)).r));  // Center pixel
float depth12 = linearizeDepth((texture(depthMap, UVs + texelSize * vec2( 0,  1)).r));

float depth20 = linearizeDepth((texture(depthMap, UVs + texelSize * vec2( 1, -1)).r));
float depth21 = linearizeDepth((texture(depthMap, UVs + texelSize * vec2( 1,  0)).r));
float depth22 = linearizeDepth((texture(depthMap, UVs + texelSize * vec2( 1,  1)).r));

// Sobel filter for normal-based edge detection
float sobelX = normalDifference(normal00, normal20) + 2.0 * normalDifference(normal01, normal21) + normalDifference(normal02, normal22);
float sobelY = normalDifference(normal00, normal02) + 2.0 * normalDifference(normal10, normal12) + normalDifference(normal20, normal22);

// Sobel filter for depth-based edge detection
float sobelXdepth = depthDifference(depth00, depth20) + 2.0 *  depthDifference(depth01, depth21) + depthDifference(depth02, depth22);
float sobelYdepth = depthDifference (depth00, depth02) + 2.0 * depthDifference(depth10, depth12) + depthDifference(depth20, depth22);

// Compute edge strength
float edgeStrengthNormal = sqrt(sobelX * sobelX + sobelY * sobelY);
float edgeStrengthDepth = sqrt(sobelXdepth * sobelXdepth + sobelYdepth * sobelYdepth);
```

The previous paper's solution provided an approach with the usage of both normal and depth rendering to generate the edges, and although my last implementation uses both, the truth is that normals are enough to accurately solve the problem. Depth provides good support for dissipation and better accuracy of interior edges, but adds the complexity of another value to tweak, and trust me, we already have lots of these. On the same note, the reality is that depth alone is not sufficient to set the contours. In practice, we would want to either use only normals, or the combination of normals and depth at the expense of more testing and extra code, but by no means would we base our detection on depth solely.

In this part we will also use light contribution for the darkness of the edges. Once we have detected the edges, we will use a simple dot product with the normal of the vertex to know the light contribution at that pixel. Although the paper proposes a simple product, we will follow the same method as in interior shading for modularity and consistency purposes, in which we add an intensity to the light. By using the adjusted brightness (e.g. the square root of the actual brightness), we will enforce the gradient difference in the lightest areas, so it is more appealing:

```
float getLightContribution(vec2 UVs)
{
    //get normal
    vec3 normal = ( (texture(normalMap, UVs) * 2 - 1)).rgb;

    vec3 lightDir = normalize(vec3( viewMat * vec4(light.dir, 0)));

    // Compute the diffuse intensity using the dot product (0,1)
    float brightness = dot(normal,lightDir) * 0.5 + 0.5;

    // Apply the square root function to adjust brightness
    return sqrt(brightness * light.intensity / maxIntensity ) ;
}
```

- **Contour Shaking + Multiple Drawing:** In this pass we will map our previously computed edges to the distortion maps that we had previously generated. The outcome will vary depending on both the distortion coefficients that we had previously chosen and the number of textures that we choose to superpose. Again, the optimal number of textures lays in the range of 3 to 5, that's why in this implementation I use 3. The main thing that

happens in this part is the mapping to the distorted textures:

```
vec2 distortedUVs = UVs + (texture(distortionMap, UVs).xy * 2 - 1);

vec4 col = texture(previousMap, UVs);

vec4 newCol = vec4( 1.f - pencilTone * texture(edgeMap, distortedUVs).xyz, 1.0);
```

In here we modulate the color of the edge with *pencilTone,* that will make the strokes darker or lighter.

In the same way, the blending of the textures happens manually. This was a personal choice because I wanted to emulate the effect of darkened pixels that happened in the hatching textures, but a common blending approach would also prove sufficient:

```
if( prev && col.r < 0.9 ) // if this pixel is already shaded we keep it that way
{
    if(newCol.r < 0.9)
    {
        FragColor = vec4( col.rgb - vec3(0.1), 0) ; // pixel is painted? make it darker
    }
    else
    {
        FragColor = vec4( col.rgb, 0); // pixel is white, paint it
    }
}
else
{
    FragColor = newCol;

}
```

4.2.2- Interior Shading

- **Brightness Adjustment:** In the same way as we did in the contour detection part, we use the light contribution with a light direction and an intensity to get the brightness at each pixel. By using the square root, we look for the effect of more differentiated light areas that scale smoothly. We need to keep in mind that this is the value that we use to map the hatching texture that will shade the pixel, so we need to map it to our local system of textures:

```
// Compute the diffuse intensity using the dot product (0,1)
float brightness = dot(normal, lightDir);

// Apply the square root function to adjust brightness
float adjustedBrightness = sqrt(brightness * light.intensity );

//choose which textue we want to use
int mappedBrightness = maxDarkness - int(floor(maxDarkness * adjustedBrightness));
```

Since in my implementation I give the possibility of changing the hatching texture's maximum darkness we will not map the brightness directly to [0, 32), instead we will use a user defined parameter to choose the mapping range, *maxDarkness.*

- **Texture Rotation + 3-Way Blending:** This pass is where the previous problem of the curvatures will affect. The principal approach for the texture rotation would be that at each vertex the minimum curvature in which the adjacent geometry changes would be stored, and we would use that per vertex curvature to rotate the textures. In the same way, this is the reason for which in the 3-way blending we superpose three textures, one per vertex curvature. However, for time and complexity reasons it was not possible for me to find an adequate approximation for the curvatures, so I will explain a simpler approach that gives decent but static results.

  Not having the per vertex curvatures directly translates into having a static 3-way blending.

This means that instead of rotating the textures with per vertex angles to follow the geometry, we rotate them by a constant factor that provides a good drawing sensation:

```
for(int i = 0; i < 3; i++)
{
    //float angle = atan(curvatures[i].y, curvatures[i].x);

    float angle = i * 3.1415f / 12.f;

    if(neg)
        angle = -angle;

    neg = !neg;

    mat2 rotation = mat2(cos(-angle), -sin(-angle), sin(-angle), cos(-angle));

    // rotate the texture to sample
    vec2 newUV = rotation * clampedUV;

    if(!useCurv)
        newUV = clampedUV;

    // darken the cross points between hatches and blend softer textures
    if(color.x == 1)
        color = texture(hatchingTexture, vec3(newUV, mappedBrightness / (i + 1)) ).rgb;
    else
        color -= (1.f - color.x) * mu;

}
```

In this case I chose to vary the rotation of the textures by 15º up and down, selecting a brighter texture for each superposed layer to give the sensation of procedural shading. Usually when we shade we tend to start with softer strokes, and end up with darker ones. By using descending lighter textures we achieve that effect. In the same way we darken the pixels that more than one stroke have shaded, to achieve the effect of superposition.

- **Paper Effect:** Finally, to give the overall image a more natural look we will make use of the previously generated paper noise SSBO. If we had the curvatures of the vertices we would be able to better represent the directions of the strokes, however, since we are using a static approach we will assume that all our hatching lines have an horizontal direction. Using the dot product of this direction and the randomly generated noise normals we create a sufficiently good granularity effect:

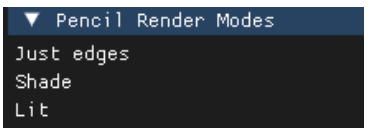
```
vec2 uvs = worldBasedUV(UVs);
int indices[2] = { int(floor(uvs.x * windowSize.x) ), int(floor(uvs.y * windowSize.y * windowSize.x)) };

// same with the normal for paper effects
if(texture(gNormal, UVs).rgb != vec3(0) && usePencilNoise)
{
    FragColor = vec4( FragColor - paperRoughness * dot(vec2(1,0), vec2( vec4(vec3(normalize( pencilNoise[ indices[0] + indices[1] ] ), 0.0 ), 0.f))) );
}
```

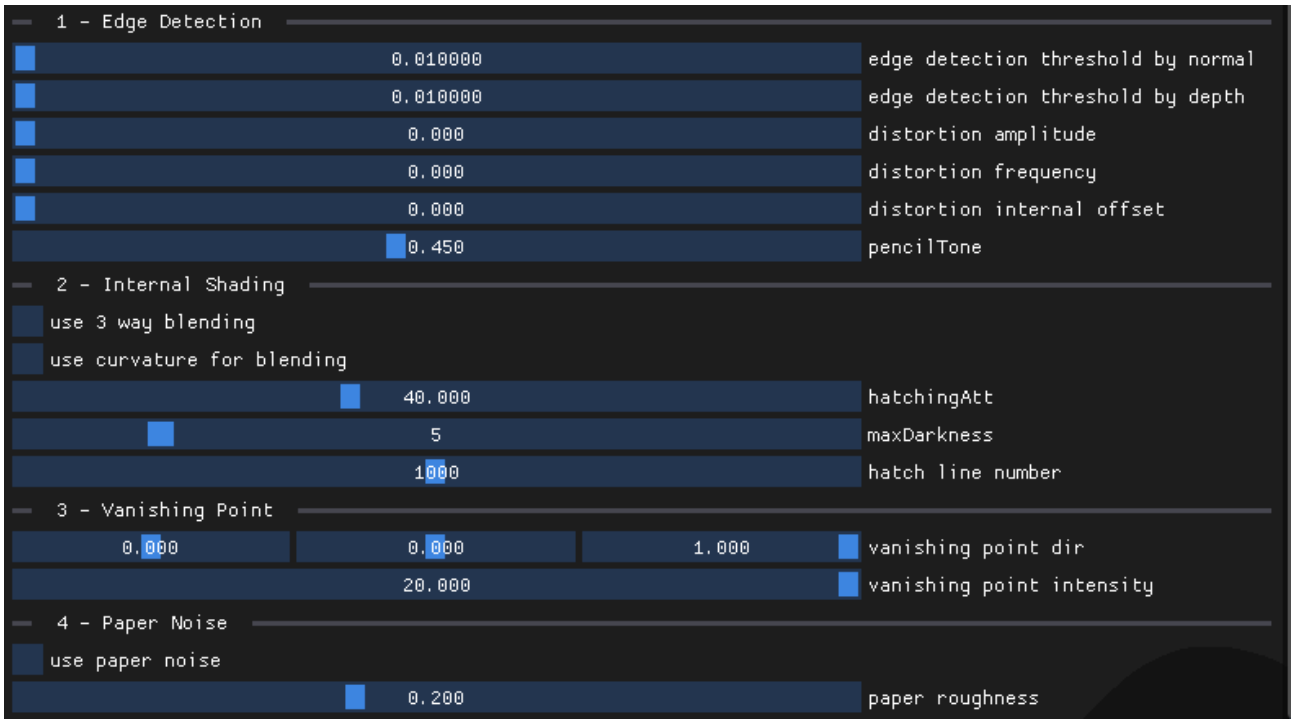
5- Outputs and Problems

5.1- Demo

This demo adds a new mode to the framework in which we will be able to tweak most of the values and test intermediate steps:



The window is further differentiated in 4 sections that control different values depending on the mode and section:

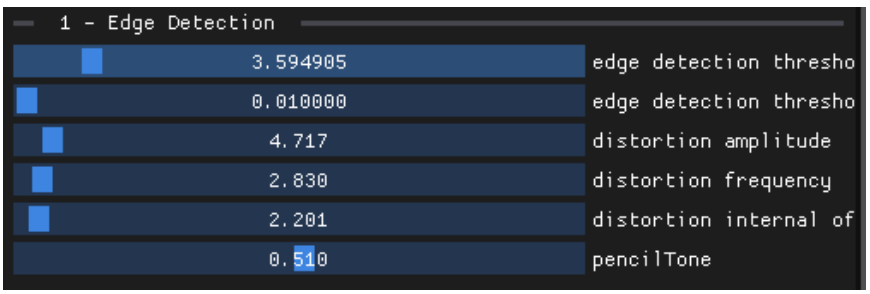


- **Edge Detection:** In this section we will be able to change the threshold of both depth and normals to determine which edges to detect. In the same way we will control the shaking coefficients for the distortion maps, that will affect the amplitude, frequency and offset of the distortion. Finally we can change the pencil tone of the edges.
- **Internal Shading:** In this section we can choose to use 3-way blending and texture rotation to see the differences between using them or not. We can tweak the hatching attenuation, that will adjust the hatching textures to the mesh until we get a result we like. In the same way we can change the base stroke number for the hatching lines, that will recompute the 32 textures. Finally, by adjusting the maximum darkness we will be able to accurately choose the darkest texture we want.
- **Vanishing Point:** In this section we can change the direction and intensity of the fake directional light that we will use to select the brightness pf each pixel.
- **Paper Noise:** Finally, in this section we can choose the roughness of the paper to get different results.

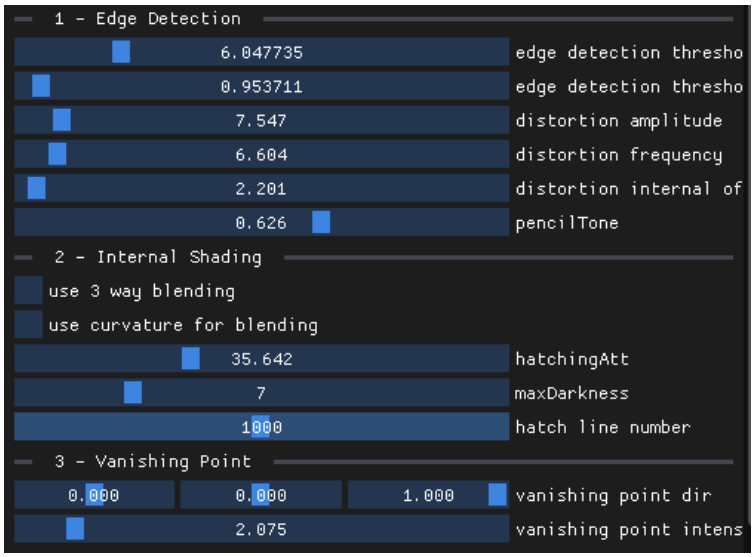
5.2- Outputs

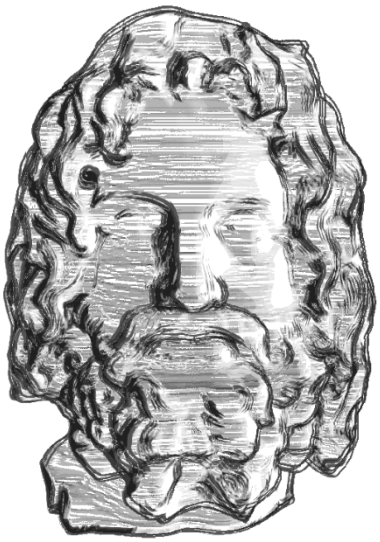
Following we will see some of the results that we can get with different values:

- **Just using edge drawing:**

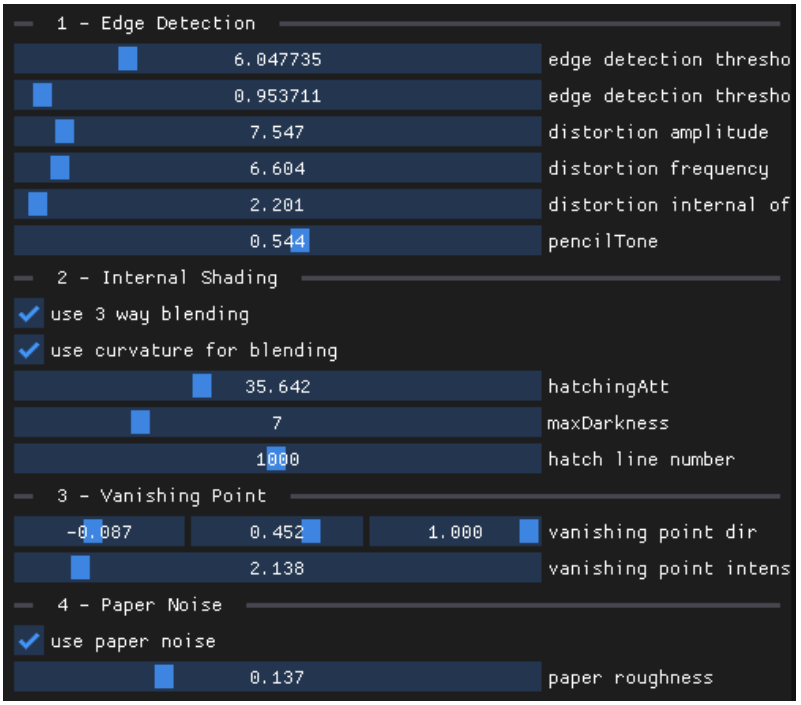


- **Using full shading without blending or paper texturizing:**

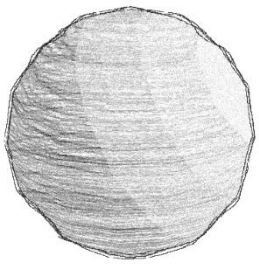


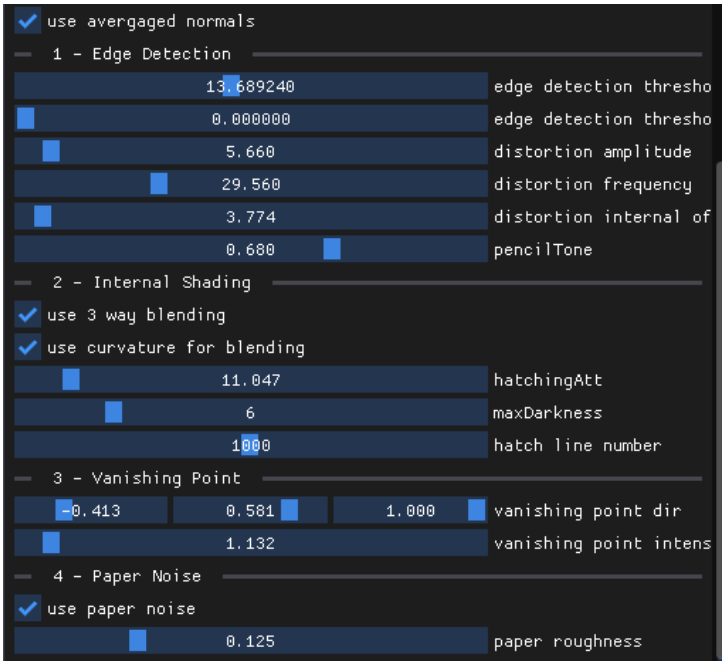


- **Using Full shading with blending and paper texturizing:**



- **Using a Sphere (with averaged normals to compensate for the simple geometry)**:





5.3- Problems

In general, the whole implementation is quite intuitive, and it is layered in a way in which the reader can program it very progressively, however, there are a few things that can cause trouble.

First, my main problem was caused by the curvatures. Although the paper of tensor fields was complete it was physics heavy, and due to lack of time and high complexity I was not able to correctly implement what they proposed. In my development time I tried to approximate it many times through a simpler application of the tensor field approach, or through the angle descent algorithm. However, my implementations did not fulfill what I was expecting, and I had to call it off.

This part was a time-hole for me, and I spent most of my time trying new things to see if I could create something useful.

Another problem was the hatching textures creation. If they are not generated with the same basis when applying them the output is not correct. The sections in which brighter textures are mapped get cut because of the inconsistencies between both. Figuring this out was quite a challenge, but the actual implementation proved to be trivial.

Finally, the mapping of the distortions and in general the mapping of the textures was tricky. Mainly because the algorithm works in screen space, and thus it is view dependent. Having to deal with this and choosing what to map through world space coordinates was not easy, however it was a matter of trial and error.

6- PROS and CONS

6.1-PROS

- It's very procedural and very modular to compute, thus we can create lots of different effects
- It is very cheap. Most things happen before the runtime application, and then it's just a matter of mapping.
- It's intuitive. Everything is very logical and tries to emulate what happens in real life
- Most things happen in screen-space; thus, they are easy to compute.
- It is very easy to expand to simulate ink, pen or charcoal.

6.1-CONS

- It can be unstable. Since distortion of maps happen in screen-space they are view dependent. This means that distortions will vary in respect to the camera, which might not be desirable.
- It is a double-edged weapon. Being very modular allows us to customize everything, however that means that to achieve a singular effect a lot of testing is needed.
- The usage for big scenes is not a good idea. The edge detection happening on a threshold in screen-space means that precision is lost with bigger z-values. The edges get blurrier and thus more unstable. Having to tweak the values for these to happen could be a challenge.

7- Future work

7.1-Ink, pen and charcoal

A great project to implement next would be to support ink, pen and charcoal drawing. This should be quite easy in terms of code. Most changes would need to happen in the modularity of the color, the way in which strokes affect the drawing and the width of the strokes themselves.

In general, it should be a matter of tweaking values and adding some new parameters to the implementation. I am sure that experimenting could give us great results.

7.2-Curvature Generation

Another future implementation would imply generating the curvature correctly. Following the tensor field application from *[Alliez et. Al. 2003]*, would give my project another turn and create a better look to the general drawing. It also is the last step that would give total completion to my implementation of the project by *PosTech*.

For this I would need to go quite deep into the previously mentioned paper and try to apply the algorithm they propose to our pipeline.

7.3- Upgrade to bigger scenes

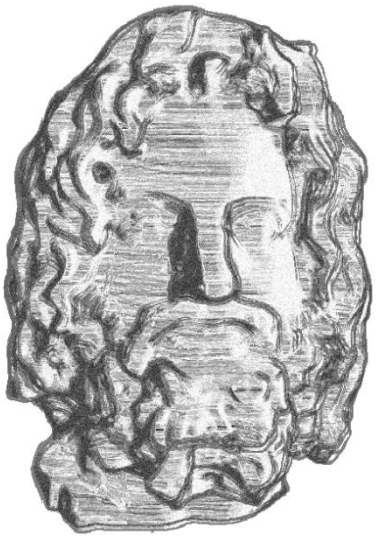
Finally, I think that a great expansion would be to upgrade the algorithm to be more stable and allow bigger scenes. A great change would be to update the algorithm, so it is not view dependent. Using object-space or world-space coordinates to generate the contours, distort them and map the textures would probably allow usage in any kind of enclosed or open scenes. I believe this would be a real upgrade to the algorithm and make it useful for lots of cases.

8- Conclusion

In general, the algorithm performs well in the cases that I have experimented with. Single objects tend to be drawn correctly, and the effects depicted are the correct ones. Since it is extremely modular and procedural it allows the user to see the results of intermediate steps such as the contour drawing. This allows us to accurately choose the values that act best to get the desired output and modulate everything to our needs.

However, this modularity is used to adapt to the needs of an object in specific, which makes it difficult to generalize it to various objects. This effect escalates to bigger scenes, which makes it very difficult to adjust to whole ambients or enclosed scenarios with lots of different objects. Furthermore, the screen-space nature of the distortions act against this generalization, making it even more difficult to create robust scenes with it.

In conclusion, the *Real-Time Pencil Rendering* implementation that I have conducted is a fast and cheap way of generating real-time hand-drawn images, and although it has great results in single objects I believe it needs to be upgraded and further explored to be used in more general cases. Considering this I hope to be able to further upgrade it in the future, as I believe it's an approach with great potential.



9-Bibliography

- **Lee, H., Kwon, S., & Lee, S. (2006).** Real-time pencil rendering. In *Proceedings of the 4th International Symposium on Non-Photorealistic Animation and Rendering* (pp. 37–45). ACM. https://dl.acm.org/doi/10.1145/1124728.1124735

- **Alliez, P., Cohen-Steiner, D., Devillers, O., Levy, B., & Desbrun, M. (2003).** Anisotropic polygonal remeshing. *ACM Transactions on Graphics (TOG)*, 22(3), 485–493. https://dl.acm.org/doi/10.1145/882262.882296

- **Isenberg, T., Halper, N., & Strothotte, T. (2003).** Stylizing silhouettes at interactive rates: From silhouette edges to silhouette strokes. *Computer Graphics Forum*, 22(3), 249–258. https://onlinelibrary.wiley.com/doi/10.1111/1467-8659.00674

- **Nienhaus, M., & Döllner, J. (2003).** Edge-enhancement—An algorithm for real-time non-photorealistic rendering. *Journal of WSCG*, 11(2), 346–353. https://www.researchgate.net/publication/2572906_Edge-Enhancement_-_An_Algorithm_for_Real-Time_Non-Photorealistic_Rendering