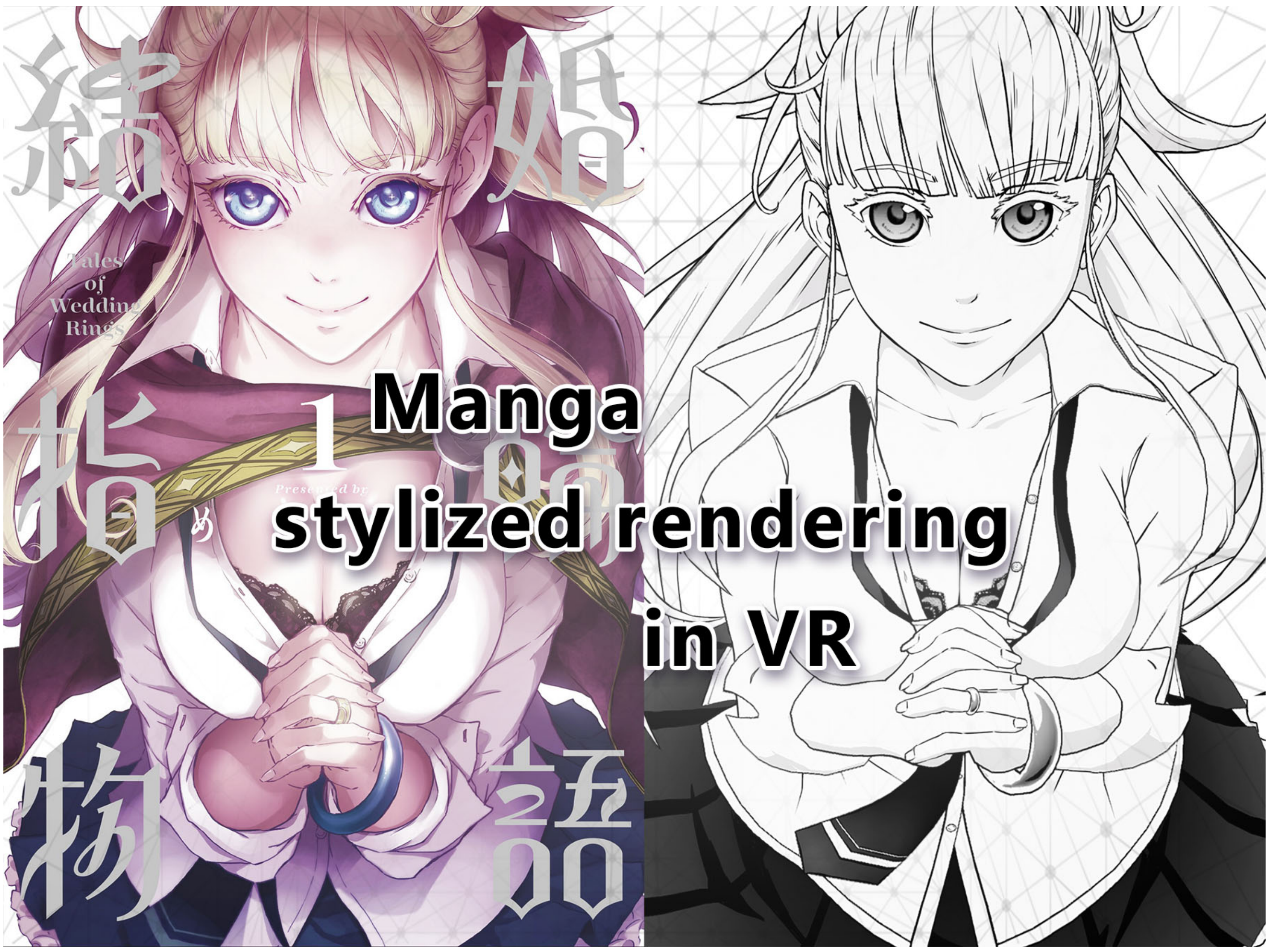
# Manga Stylized Rendering in VR

Julien Guertault, Élie Setbon, Pavel Martishevsky
*Square Enix Co., Ltd.*



*© 2018 SQUARE ENIX CO., LTD. All Rights Reserved.*

*© 2018 MAYBE.*

*SA '18 Courses*, December 04-07, 2018, Tokyo, Japan
ACM 978-1-4503-6026-5/18/12.
10.1145/3277644.3277768

## Contents

1 Introduction and context

2 Art side
- 2.1 Goal
- 2.2 Shading
- 2.3 Lines
- 2.4 Texture maps and UV management
- 2.5 What we couldn't do

3 Implementation side
- 3.1 Toon shading
- 3.2 Line detection
  - 3.2.1 Depth-based lines
  - 3.2.2 Normal-based lines
  - 3.2.3 Texture-based lines
- 3.3 Line transparency
- 3.4 Hatching
- 3.5 Half-tone and paper texture
- 3.6 Transition between frame and immersive 360°
  - 3.6.1 First prototype
  - 3.6.2 Live Windows
  - 3.6.3 Diorama effect

4 GPU Optimizations
- 4.1 Performance Constraints
  - 4.1.1 VR Performance Constraints
  - 4.1.2 Game Design Performance Constraints
- 4.2 Performance Monitoring Process
- 4.3 Small, Yet Important Optimizations
  - 4.3.1 Forcing HiZ/EarlyZ Tests for Alpha-tested Materials
  - 4.3.2 Optimizing Light Fetching and Computation
  - 4.3.3 Optimizing for Shader Occupancy
- 4.4 Extending Unreal Engine 4
  - 4.4.1 Triangle Culling
  - 4.4.2 Post-Processing Material Optimization

5 Conclusion and take away

6 Acknowledgments

# 1 Introduction and context

The research and development department of Square Enix, the *Advanced Technology Division* (ATD), has recently released a virtual reality (VR) adaptation of a manga. Since the very beginning of this project, its theme has been the question:

“ *What would it be like to be inside a manga?* ”

Manga is a strongly established narrative style, with a distinctive visual aesthetic. It typically features cinematic frames, strong black and white contrast, outlines of varying thickness, speed lines, and patterns like hatching or half-tone to compensate for the lack of color, among other elements.

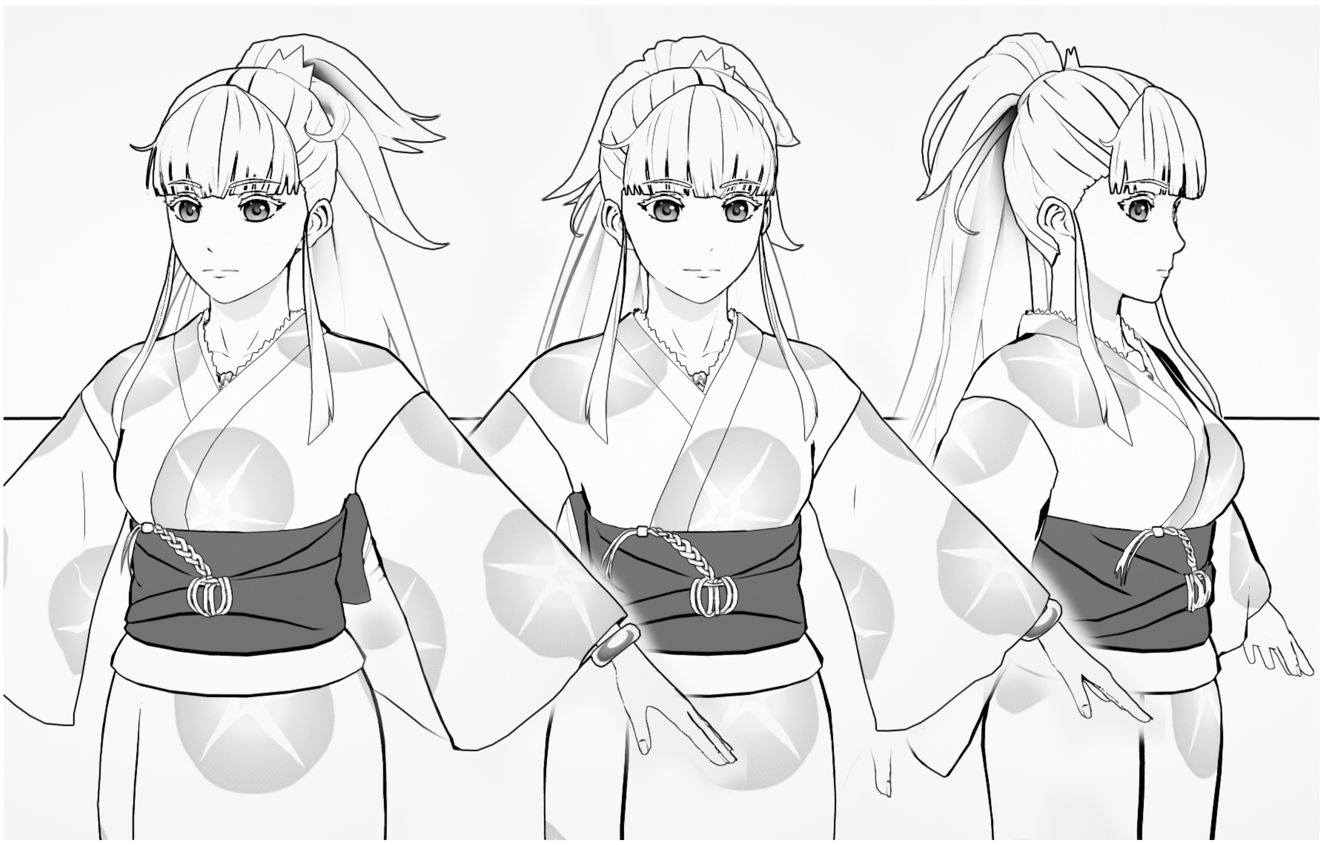
Would this style work in VR? How could page composition be translated to a 3D space? Would the outlines make sense in 3D? Would patterns still work with the pixel density of current VR headsets?

Such are some of the aspects the project was meant to tackle. In this course we offer to share some of the lessons learned through this experiment. We will concentrate on the rendering, but there are other interesting elements to consider like interaction and pacing, which fall outside the scope of this course.

# 2 Art side

## 2.1 Goal

*Manga likeness*



Every manga has its own stylization, and while there are common elements to all manga, we were interested in trying to reproduce the visual specific to the title at hand. The approach for this was to analyze all images, group out the different visual elements, and come up with shading and rendering models that can allow us to reproduce most visual cases encountered in the manga.

The shading used in manga is obviously not a realistic one, it is simplified and stylized. Because we want to stick to the visual style specific to the manga we're adapting, we ruled out a physically-based rendering (PBR) approach from the start, and decided to go with a custom non-photorealistic rendering (NPR).

## 2.2 Shading

When the characters are not just white, the main component of the shading is a grey half-tone. Tests showed that half-tone would not give good results in VR (see section 3.5), so we decided to represent it as it appears: a grey gradient depending on the light position (figure 1). Where part of the character is shown in white, and the shaded area is grey, the next step is to get some control over how much of the object is white and how much of it is going to show the grey part we talked about previously (figure 2).

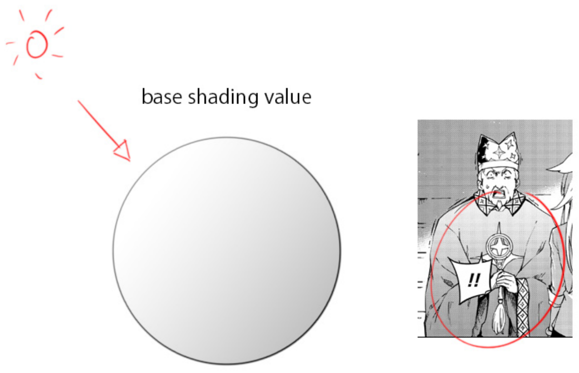


**Figure 1:** *Halftone appearing as a gradient in the original manga. © MAYBE.*

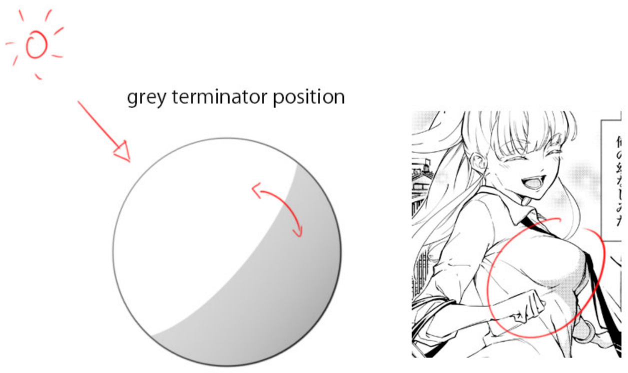


**Figure 2:** *Position of the transition between white and shaded areas.*

The grey part also appeared to differ in value from panel to panel (figure 3), so we need a control over its brightness. In some cases, the terminator between the white part and the terminator between the white part and the grey part is hard, but there are also cases of it being soft (figure 4), so we need to be able to control this aspect of the shading too.

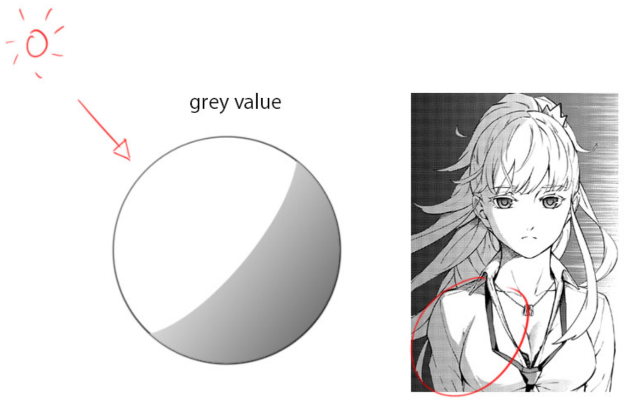


**Figure 3:** *Example of darker shading.*

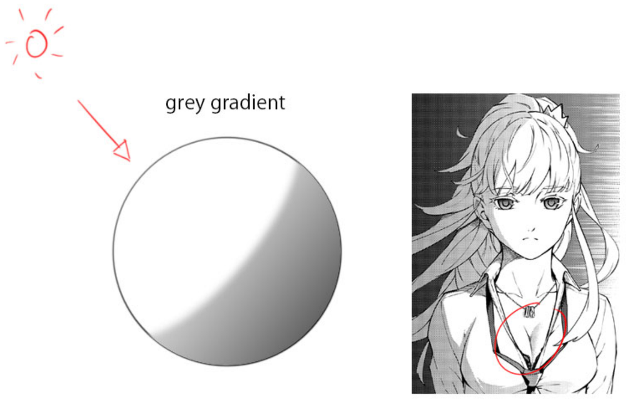


**Figure 4:** *Example of soft transition between white and shaded areas.*

On top of the white and grey parts which are the most common, we can also observe some panels showing pure black, rarely used but still present in the case of dramatic back-lit shots (figure 5). Some objects like accessories show a different type of shading, where the manga artist is mimicking materials like glass or metal (figure 6). It can be roughly reproduced using a Fresnel element.

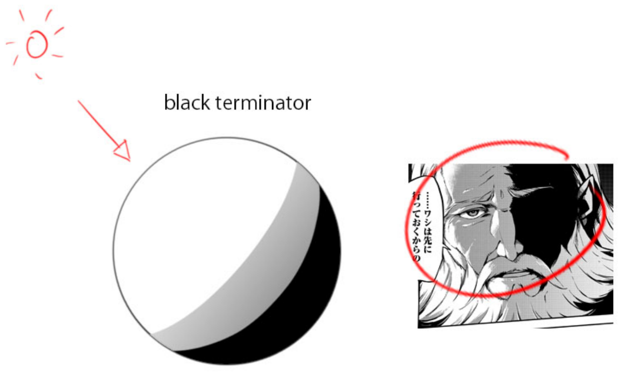


**Figure 5:** *Example of backlit scene.*

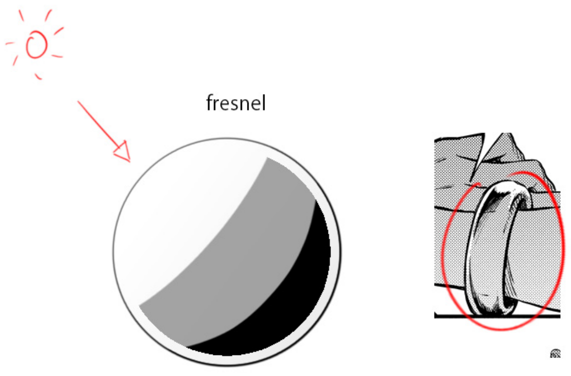


**Figure 6:** *Material of a jewel.*

Another case encountered is when, instead or on top of the grey part, the shading is represented with hatching (figure 7, figure 8).

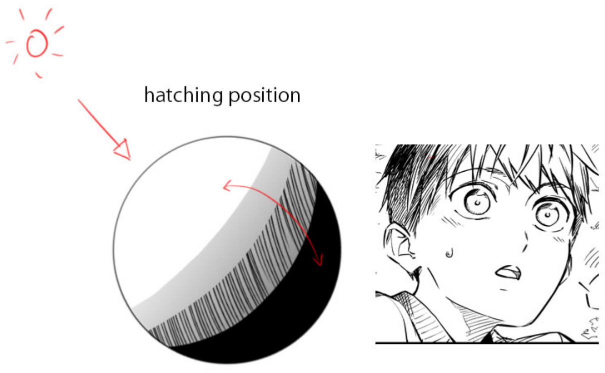


**Figure 7:** *Hatching used to represent a shadow.*

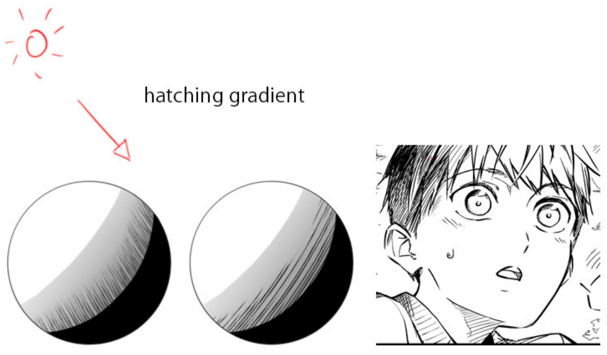


**Figure 8:** *Hatching in different directions.*

## 2.3 Lines

First off, in VR characters and assets in general need to be more optimized; in our project specifically, we are rendering the scene even more times because of the use of manga styled paneling (figure 9), a.k.a. *Live Windows*.

For that purpose, we are trying to model in an efficient way to use polygons where they are needed (figure 10). Because at this point we can't see the visual in-game result, we need to use approximation with *Maya* or *3ds Max* toon renderer and previous experience to essentially guess the result.

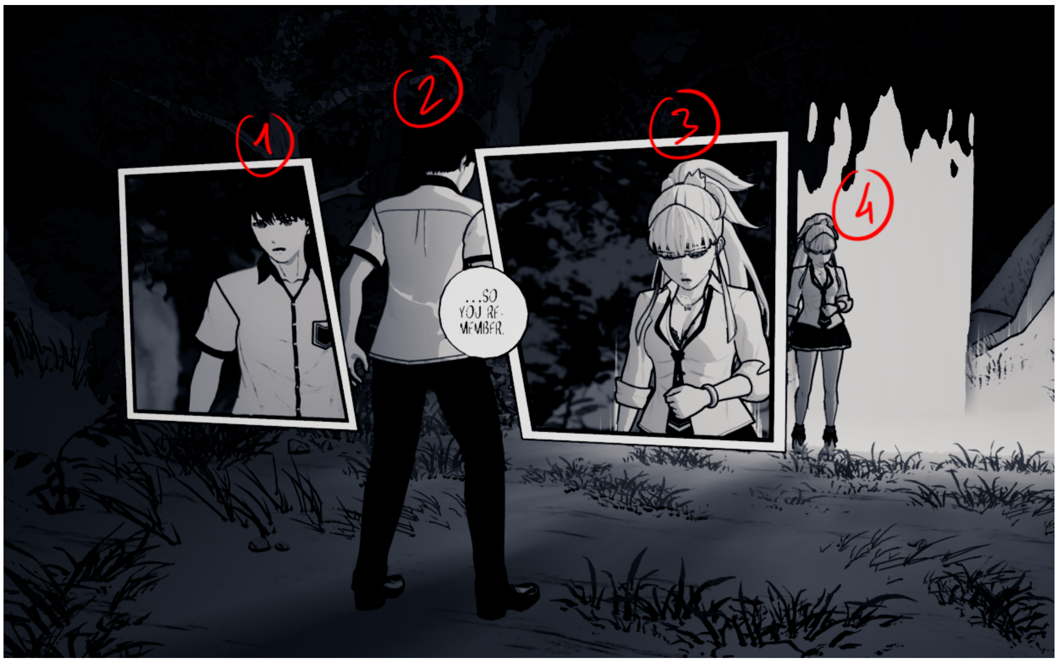


**Figure 9:** *Characters rendered multiple times through Live Windows.*

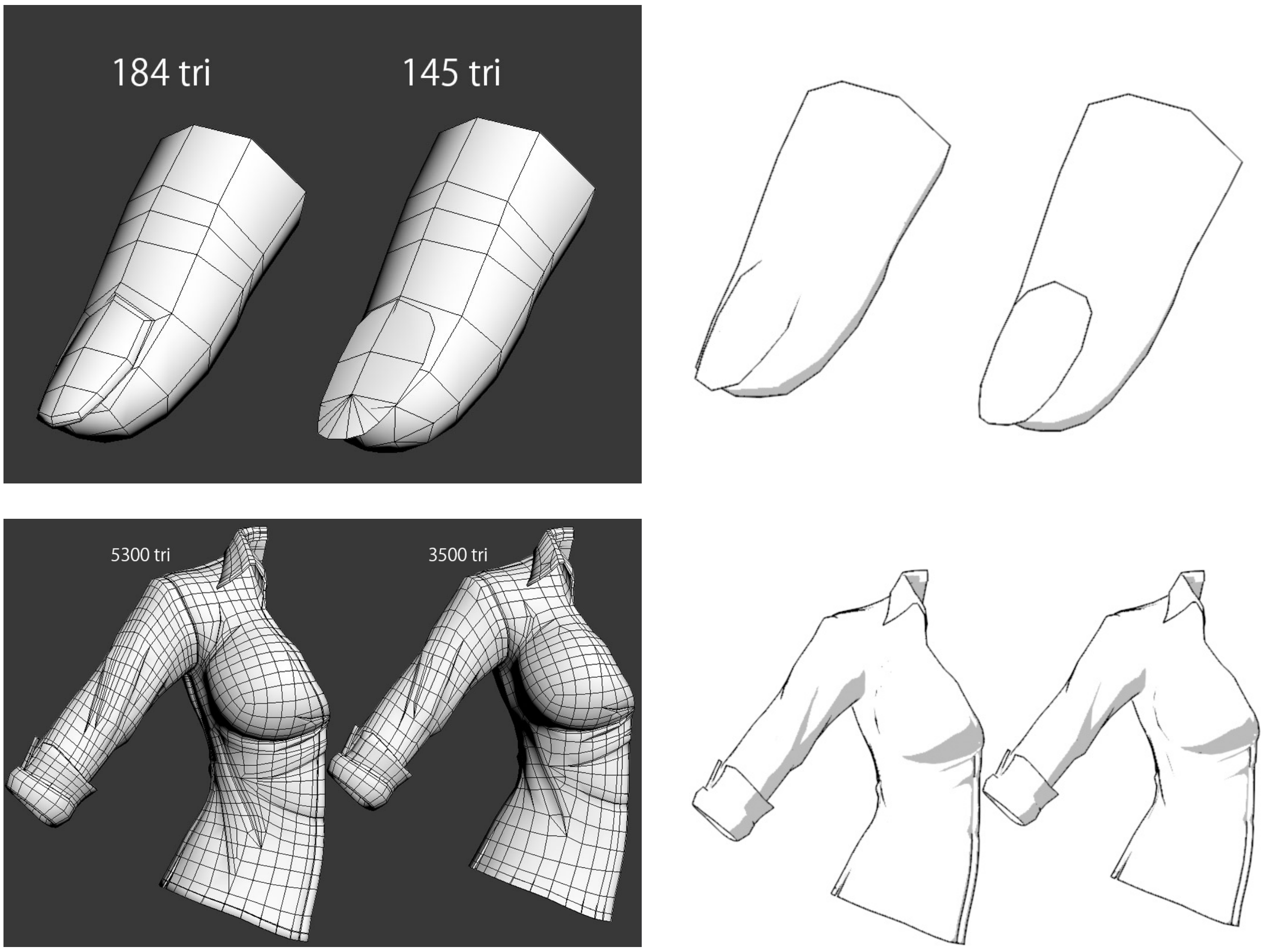


**Figure 10:** *Examples of meshes optimized to have less polygons but get a similar or better quality result.*

Now, when the modeling is done and it's first imported in the engine, that's what the character looks like:

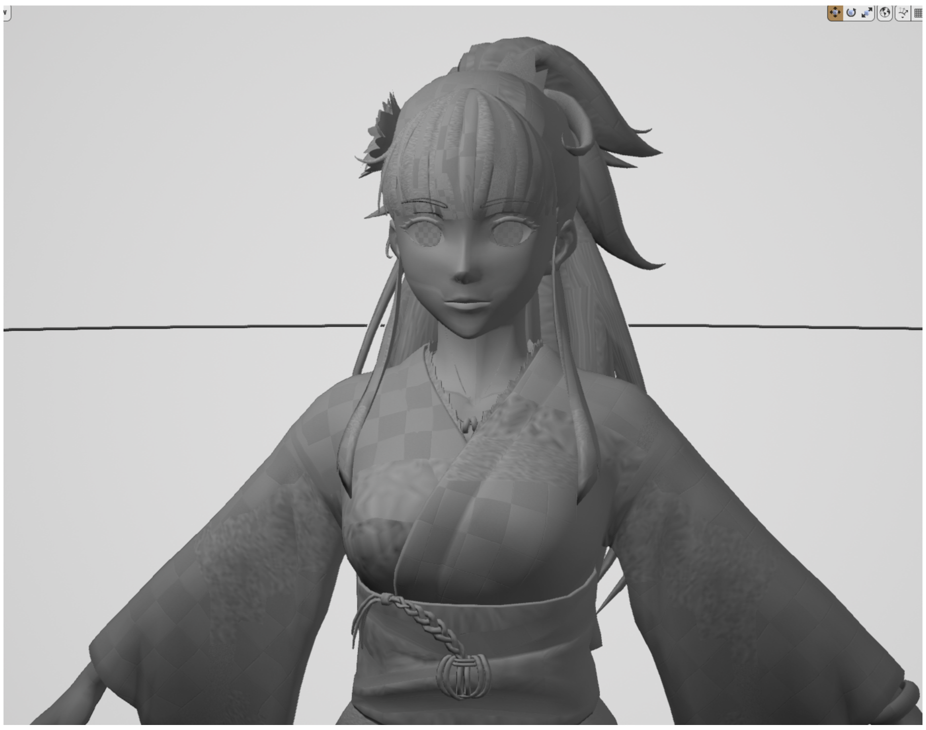


**Figure 11:** *Character mesh displayed in Epic® Unreal® Engine 4 with default shading.*

The first and main step to achieve the manga look is to import the post-process into the scene, after iterations made with the graphic programmer (section 3.2 will give more detail) we end up with this:

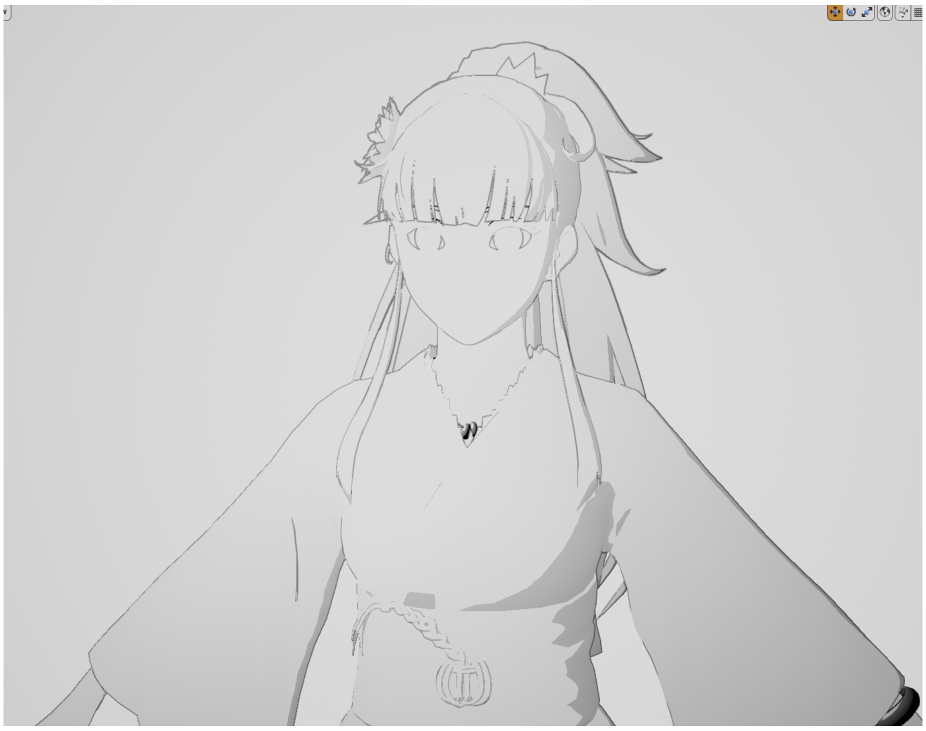
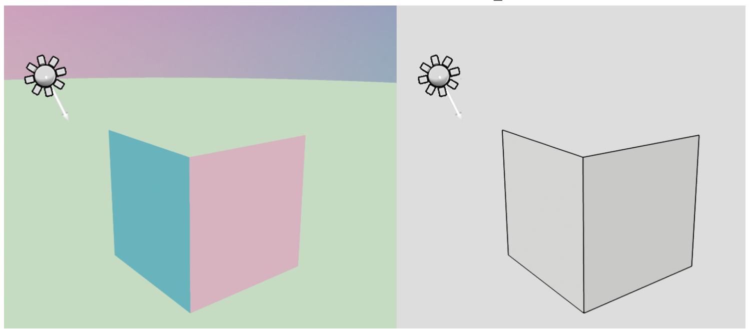


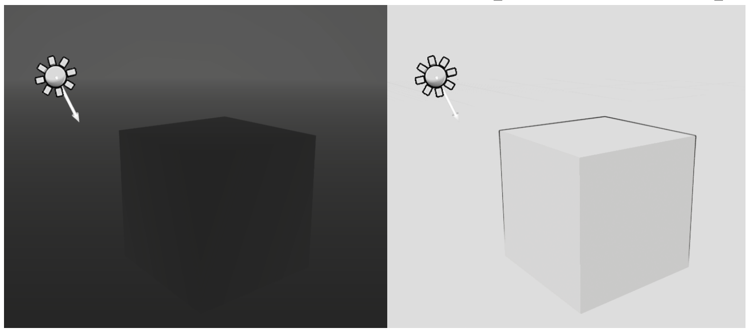
**Figure 12:** *Character mesh displayed with the manga post-process.*

A lot of complicated things happen in there, but because we are artists we will mainly keep in mind that lines are created in 2 ways:

1. Where there is contrast between 2 pixels in the normal pass



2. Where there is contrast between 2 pixels on the depth pass



The programmers exposed some parameters for artists to have control over, we can choose things like how much distance between 2 surfaces is the minimum for a line to be created thanks to the depth pass, or how much contrast on the normal pass will create a line, etc.

After tweaking those parameters we get something cleaner and already much closer to what we are looking for:

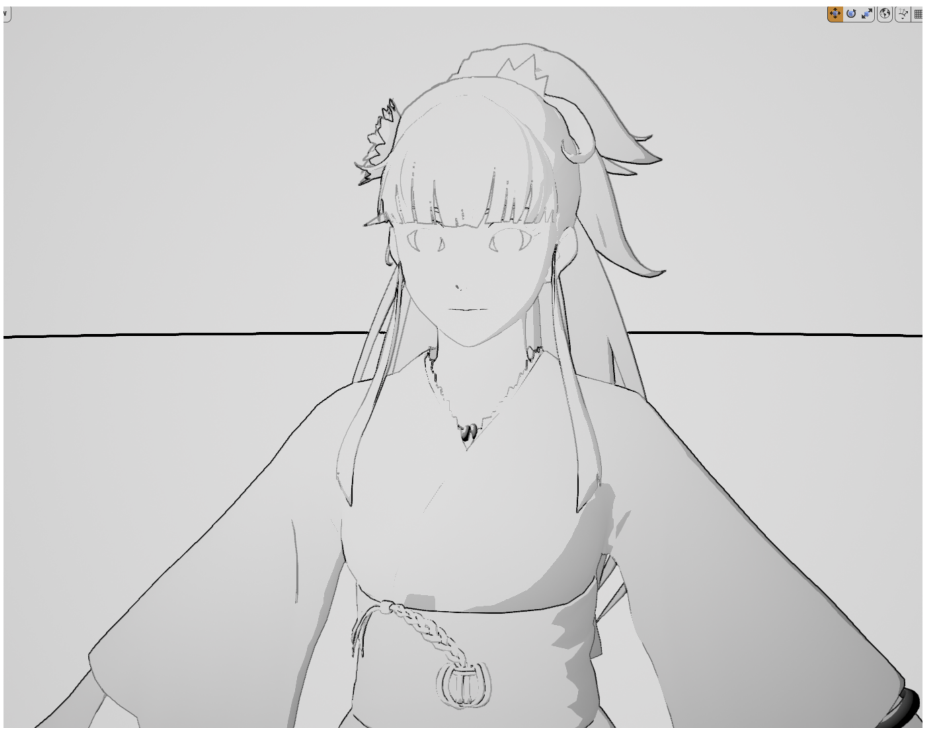


**Figure 13:** *Character mesh displayed with the manga post-process, using fine tuned parameters.*

At this point we're getting a first result, but we can influence the post-process with our modeling (figure 14), mainly adding hard edges / splitting smoothing groups, and tweaking the model topology and shapes.

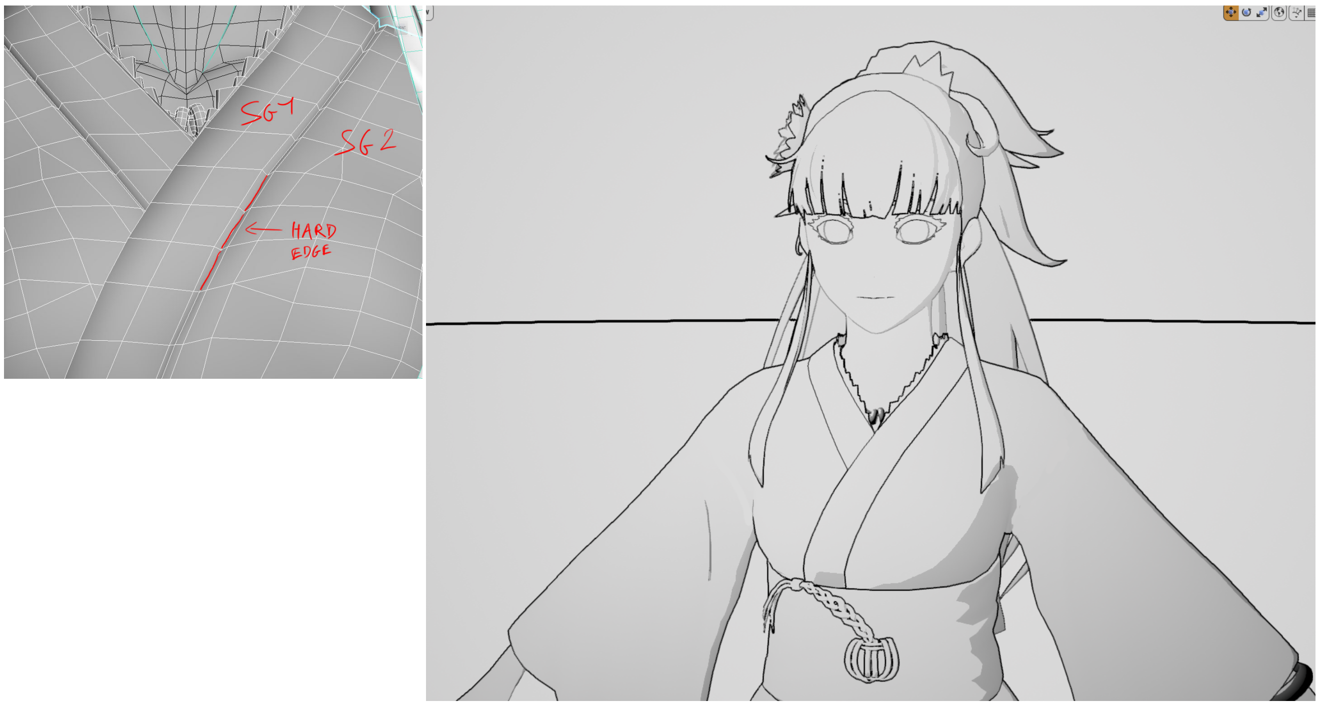


**Figure 14:** *Sharp edges added to the character mesh to help the edge detection post-process.*

At this point we made the best of the post-process, but we're still missing many lines:

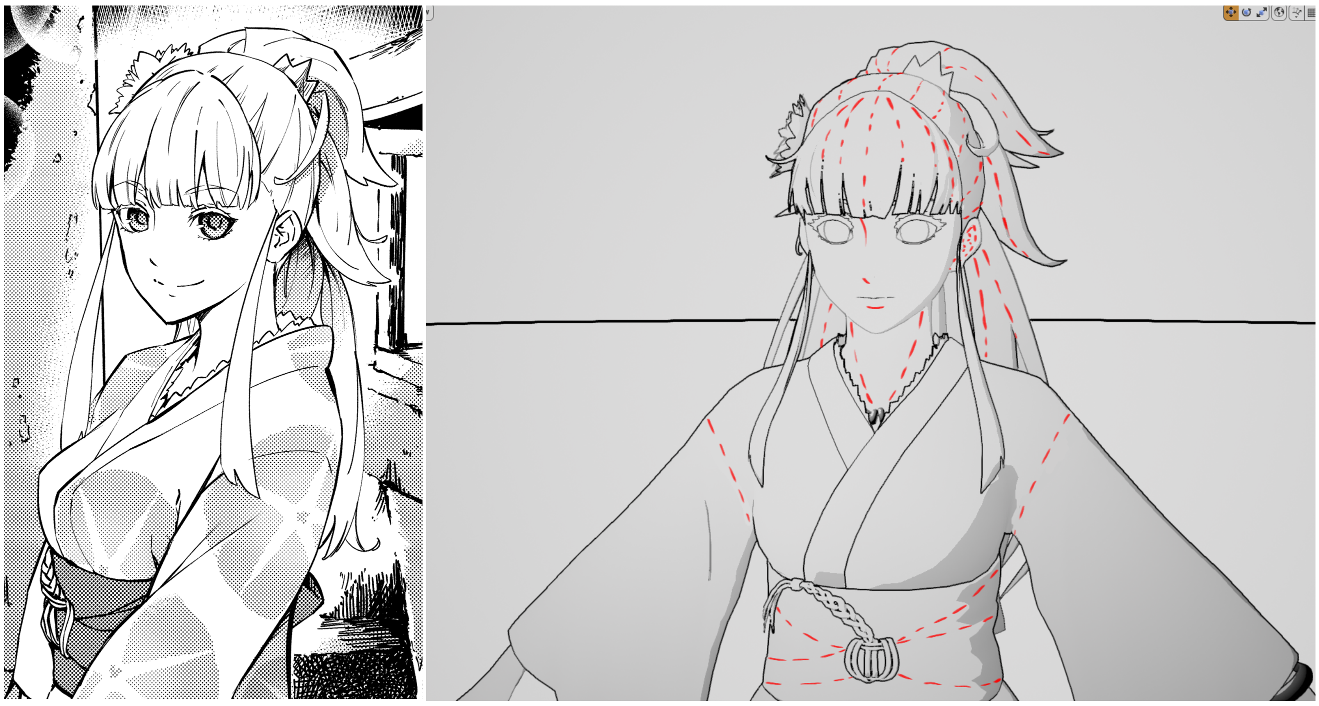


**Figure 15:** *Missing lines that are not detected by the post-process.*

We can distinguish two types of missing lines: permanent lines (figure 16) and view-dependent lines (figure 17).

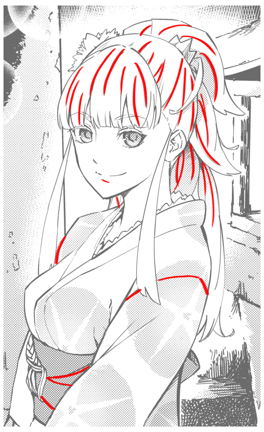


**Figure 16:** *Missing permanent lines.*

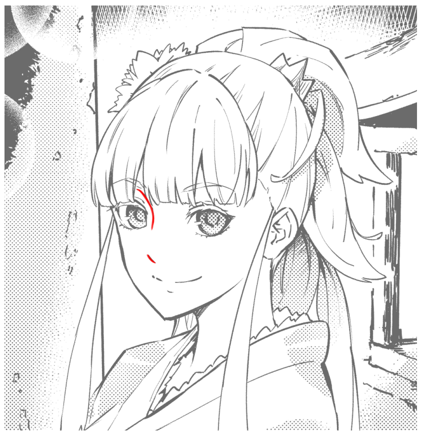


**Figure 17:** *Missing view-dependent lines.*

For the first type of line, the permanent lines, we want to just draw them in the texture. One issue though, is that we will need a very high resolution texture in order to get a clean line. To go over that problem we are using the UV shift technique. This technique was adopted from [Motomura15]'s in *Guilty Gear Xrd*, but because we don't have colors we can further optimize the texture (figure 18).

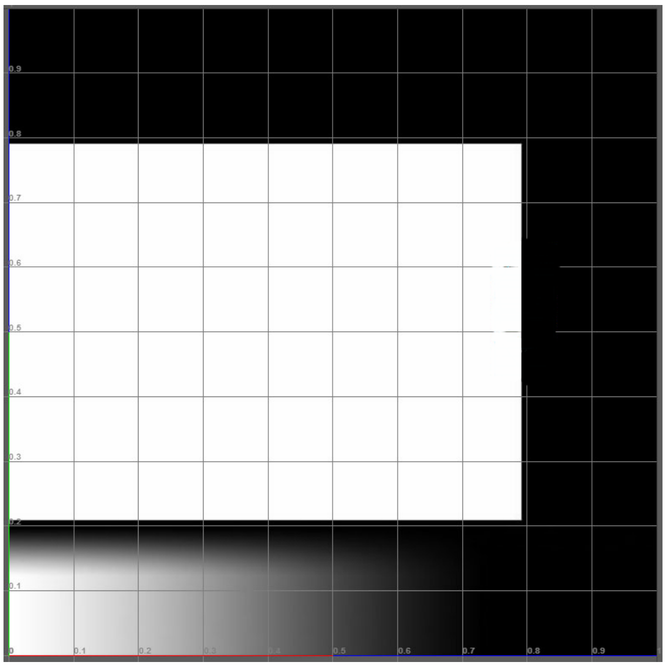


**Figure 18:** *A simple texture used for the lines.*

The next step is simple, we just need to cut out the UVs of the polygons around the edge where we want a line, and stack them onto the map (figure 19). The resulting UVs may look very nasty, but that's okay. :)

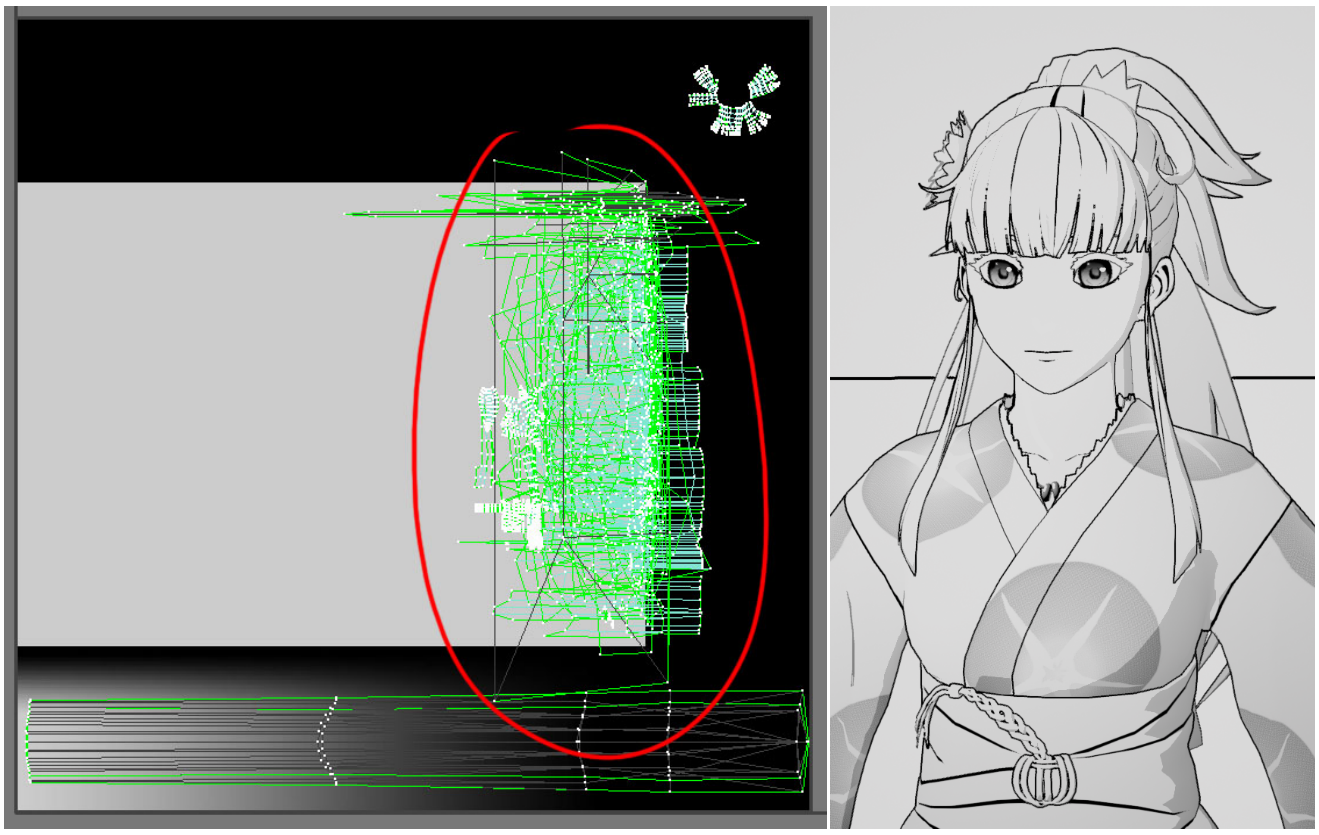


**Figure 19:** The UV mapping and the resulting lines.

For the second type of line, we need them to be visible at certain angles only, nose (figure 20), neck lines, nose bridge (figure 17), etc.

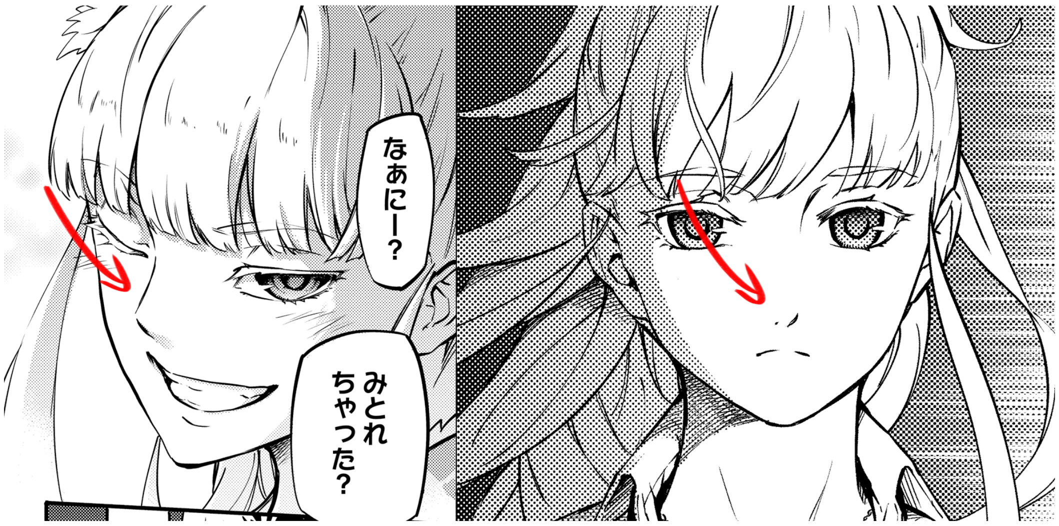


**Figure 20:** *The lines of the nose in the original manga.*

As said previously, the usual technique used for outlines is the inverted hull, we decided to use a similar technique for these lines. Instead of duplicating and inverting the whole character mesh, we're only creating small back-faced polygons (figure 21) where we need them, for these specific lines. Applying a one sided material on them allow them to appear and disappear (figure 22) depending on the angle you're looking from.

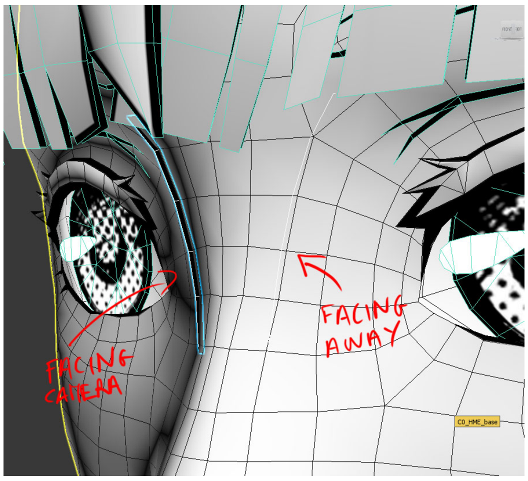


**Figure 21:** *Custom mesh topology with single-sided faces for the nose bridge.*

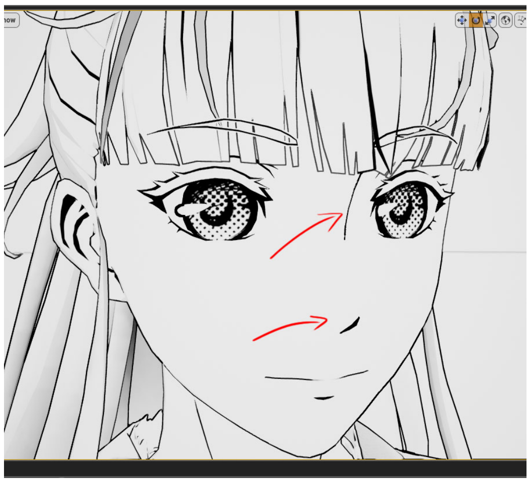


**Figure 22:** *View-dependent lines of the nose visible in the final render.*

At first we thought we could get all the lines we needed with the techniques above, but there are still cases in which lines are still not showing.

In order for us to get those lines, we decided to use what we call "line detection texture". This was a feature that we *artists* really wanted but came with an implementation cost (see section 3.2.3). This technique consists in painting the different elements on the character with different values, and for the post-process to draw a line between 2 parts that have a different value (figure 23). It's a very powerful technique that gives artists a lot of control.

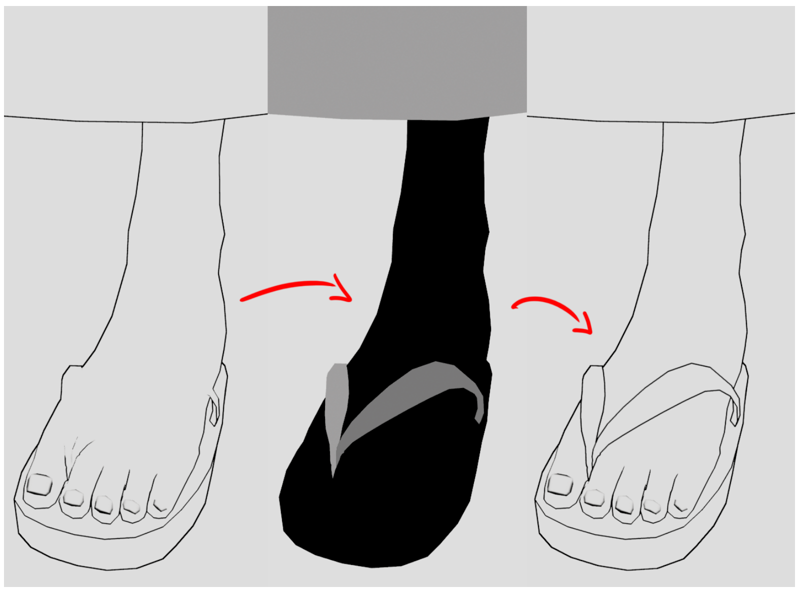


**Figure 23:** *Missing lines on the foot are completed with an additional texture to help edge detection.*

Because of the technicalities of its implementation, we only have 8 values to play with, and they have to be shared with the environment (to make sure lines are always drawn between characters and environments), so that leaves us with 4 values for the characters.

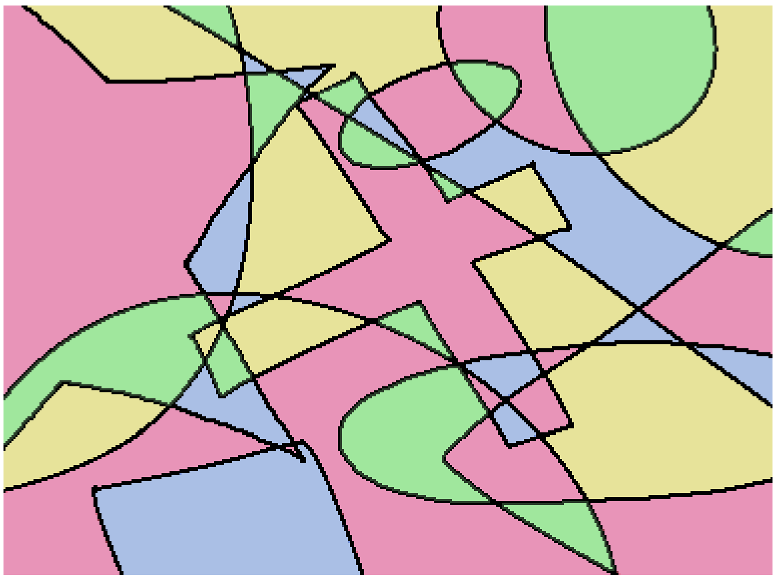


**Figure 24:** *Example of partitioning with four colors.*

In theory [Wikipedia] we should be able to separate any combination of objects with just these 4 values (figure 24). Of course and as always, it's not as easy and we do have cases where we wish that we had more values to play with

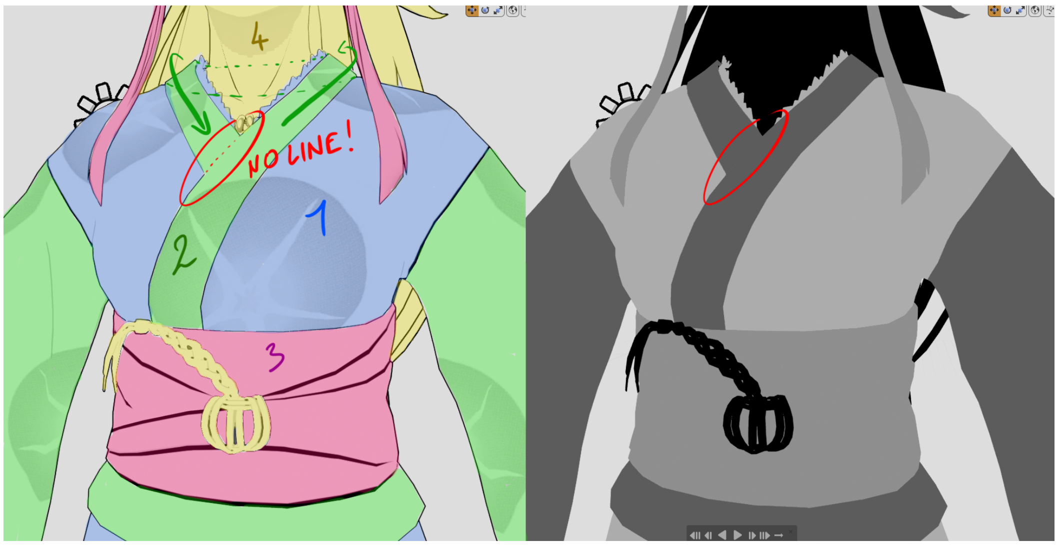


**Figure 25:** *Where the yukata collars meet lacks a line at the crossover.*

We'll see in a bit how we can hack a fix for this. In the meantime we are still able to get all the missing lines (figure 26), so at this point we're happy with the result on the lines.

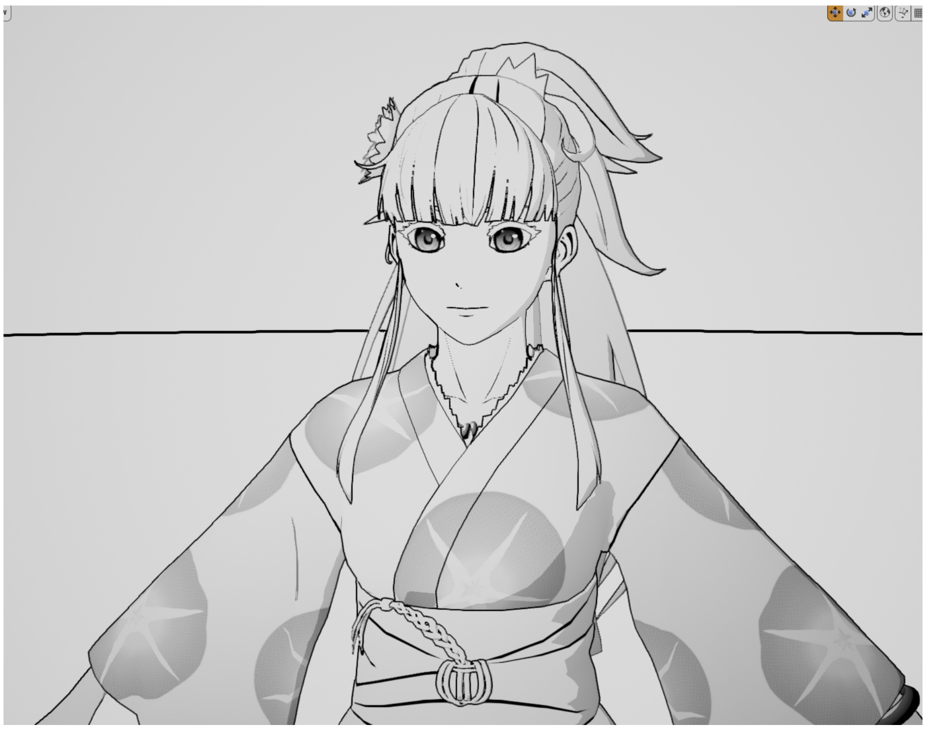


**Figure 26:** *All the wanted lines are appearing.*

Now that we have all the lines we wanted, we can see that there are areas where lines are showing but either we would want them to be more subtle or we don't want them at all:

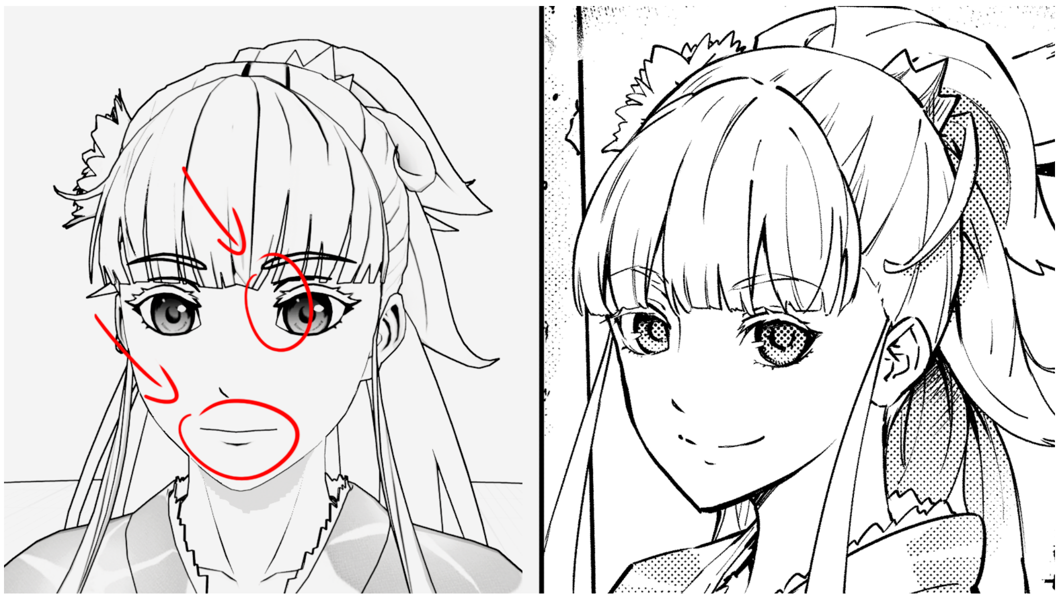


**Figure 27:** *Areas where lines should get thinner, or be removed entirely.*

In order to fix this, we are using a texture to control the line thickness (figure 28). This is pretty straightforward: a dark value indicates a thinner more faint line, and black means there is no line being drawn.

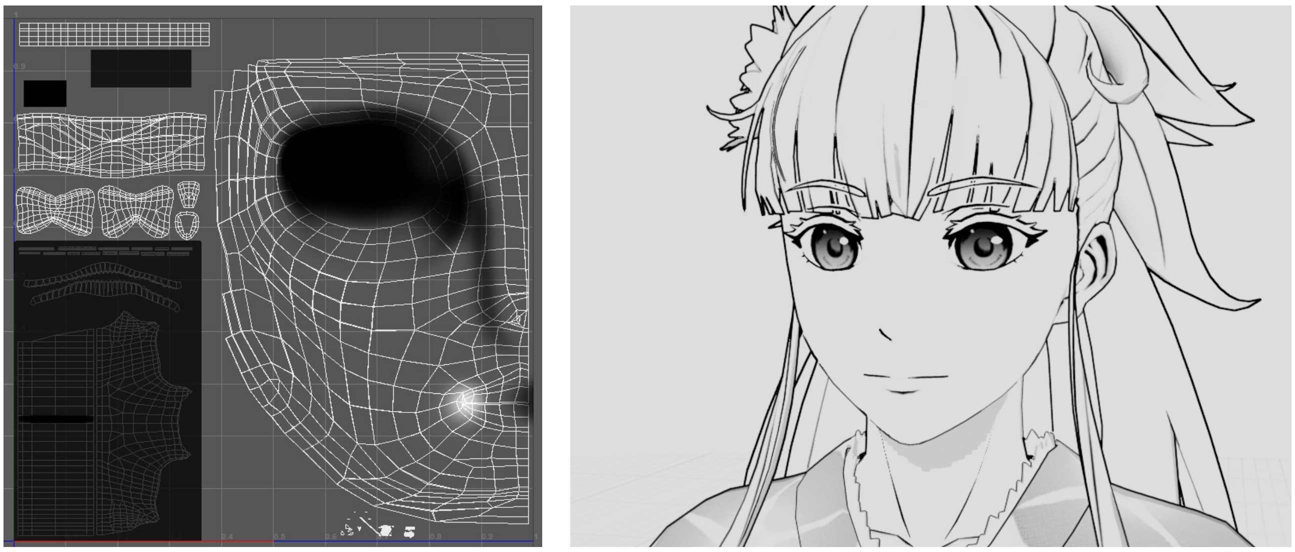


**Figure 28:** *Texture controlling the line thickness, and result with lines of varying thickness.*

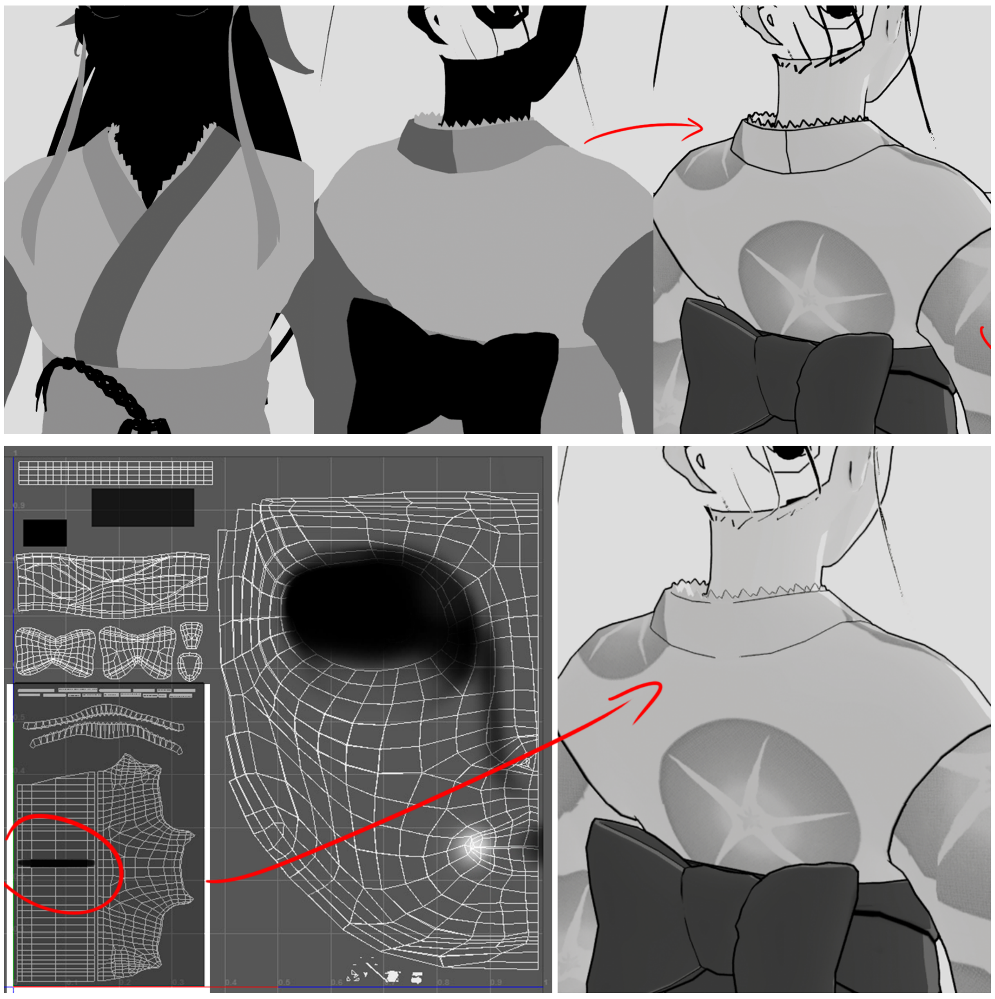


**Figure 29:** *Collar missing line issue solved with the line detection texture and (ab)using the line thickness control.*

We are also using the line thickness function in order to fix the problem we had earlier with the line detection texture on Hime's yukata (figure 29): we lay the UVs of the collar on 2 values for the line detection texture and get a unwanted line, on which we set line thickness to 0, and voilà!

## 2.4 Texture maps and UV management

As talked earlier, the goal was to optimize the texture maps as much as possible to be able to reduce the size, and also to create a base that could cover most cases and be re-usable as a standard for characters. We identified that some parts of the texture maps could be re-used throughout all the characters of the game: lines, shades, etc., and other things are more character specific.

The black parts on top and on the right side are for lines, a certain thickness is needed to allow for different line thicknesses and a corner is necessary for cases like nails or details in the ear where we need lines on both sides of the same polygon group:

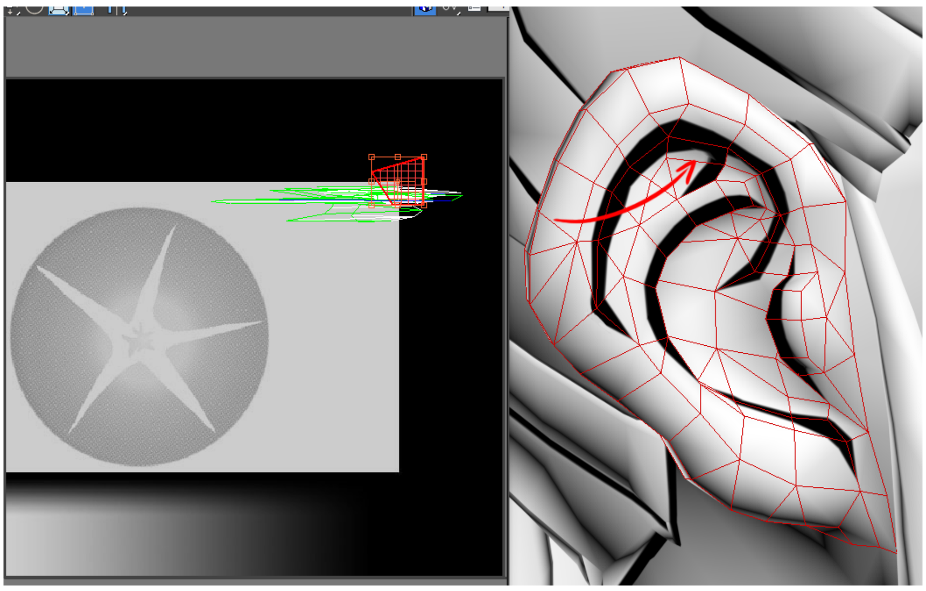


**Figure 30:** *Lines of the ear done with custom UV mapping. The corner is used to get a curved line with a T-junction.*

As you can see we first setup the UVs nicely, but since only the horizontal information is important, we then squeeze them up, in order to free space in the center.

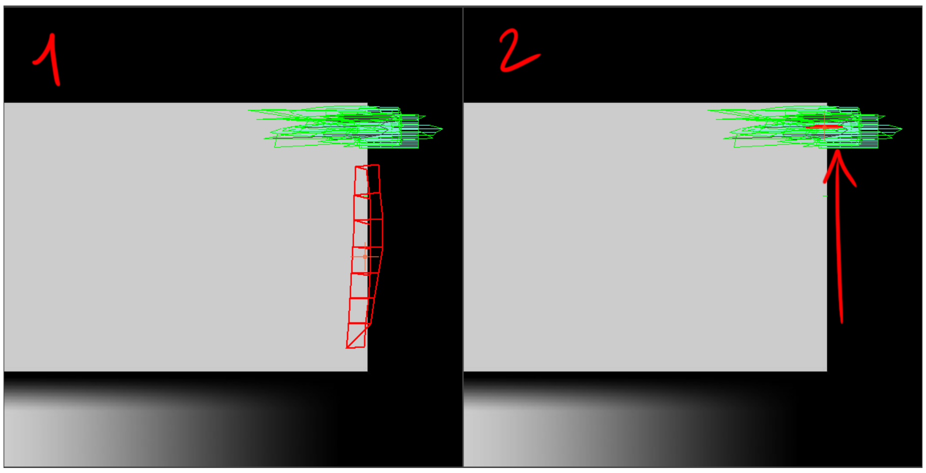


**Figure 31:** *Squeezing a mesh on the UV map to optimize texture space.*

The gradient on the bottom part is used for inside of sleeves, teeth, inside of the mouth (figure 32)... any place that needs a smooth transition between light and shade.

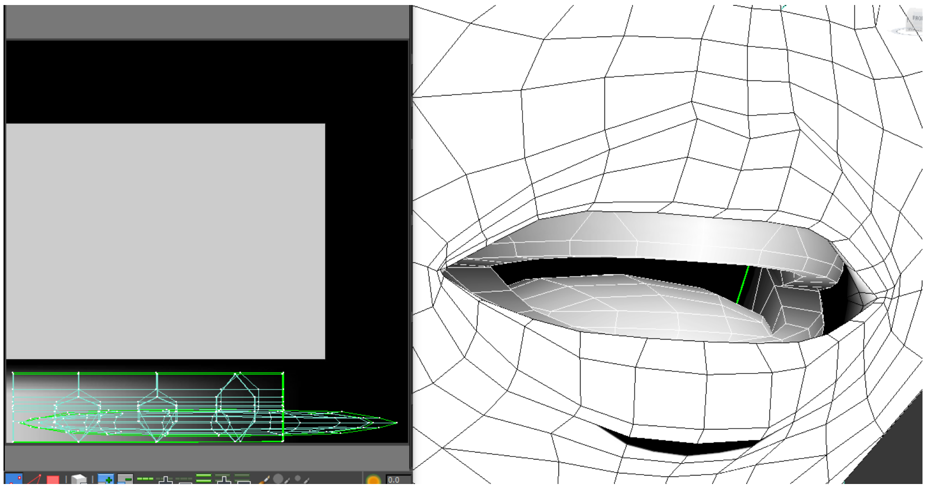


**Figure 32:** *Example of use of the gradient area for the mouth and teeth.*

The middle part is reserved for any type of pattern or details on the character, again we optimize by overlapping UV shells:

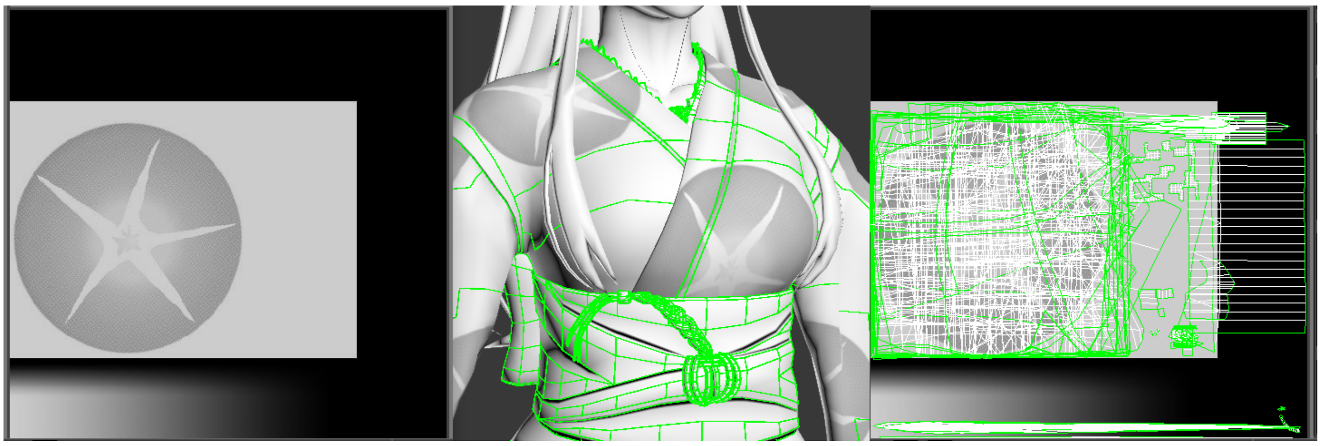


**Figure 33:** *Decorative pattern repeated on the yukata with overlapping UV shells.*

Then, we need to have different values for the different parts of the characters, because these parts also need to have lines, we have to work on these on a separate UV channel:

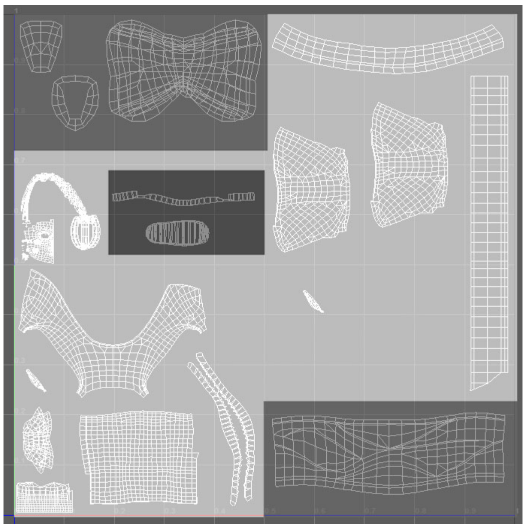
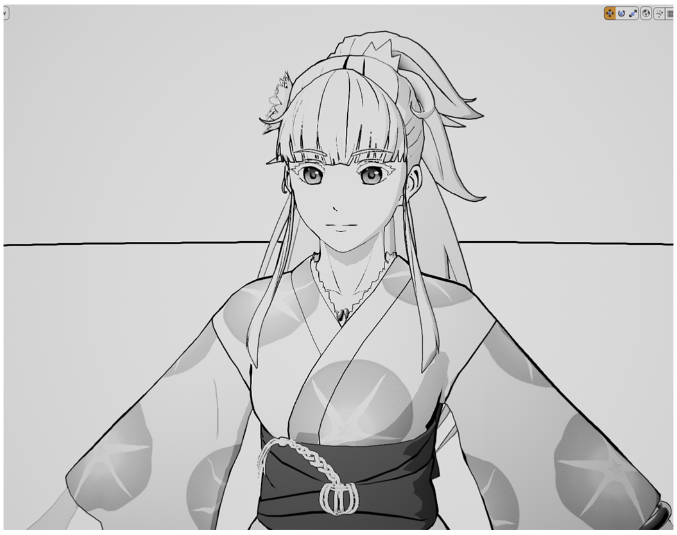



**Figure 34:** *Separate UV mapping channel.*

It might look like this texture is not optimized, but because we have a limited amount of UV sets to play with, we need to use the same UV set for several purposes. For instance here the line detection texture:

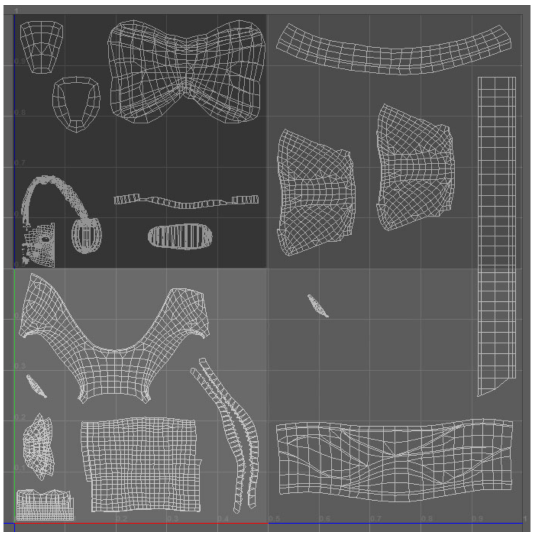


**Figure 35:** *Mesh elements grouped into the areas of the four different values for line detection.*

We also added some important self shadows, which we have to fake for performance reasons. As you can see, we also have a texture map for the line thickness, the line detection texture map, etc. In the end, we had to deal with several textures, that need to work on top of each other, so ideally we would have had just as many UV channels, but we realized we were limited to 4 channels, and it was a real challenge to organize them to be used for several texture maps (figure 37).

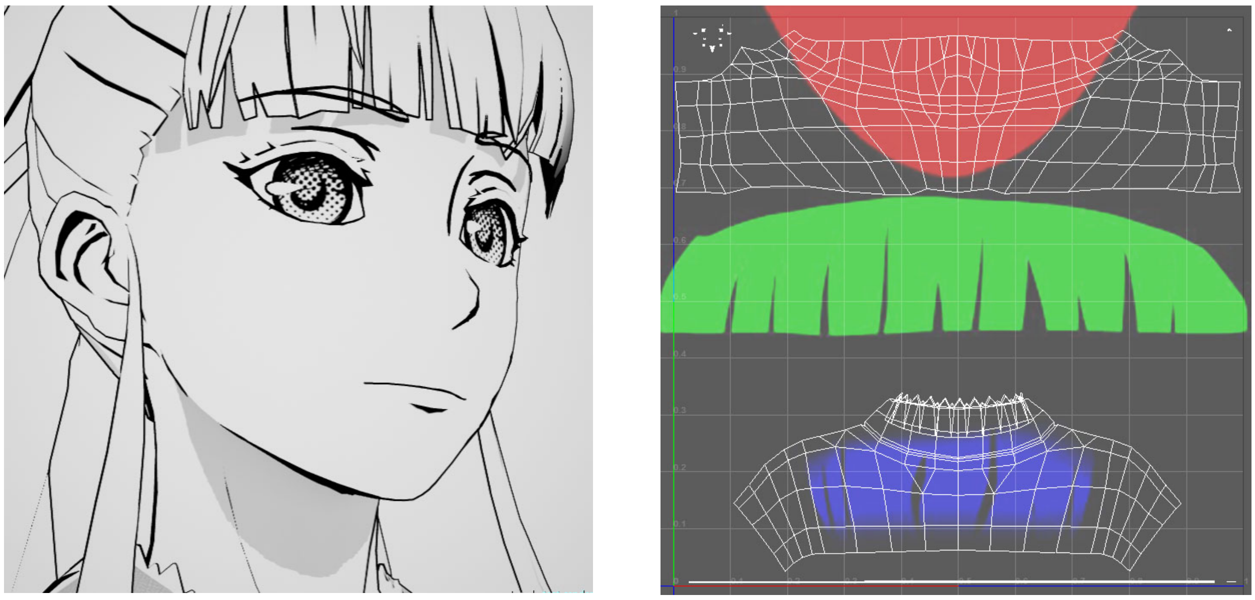


**Figure 36:** *Texture-base fake self shadows for the face of the character.*

It's still something that we would like to improve. We want to add more features on the characters, so we'll have to organize the available UV channels accordingly.

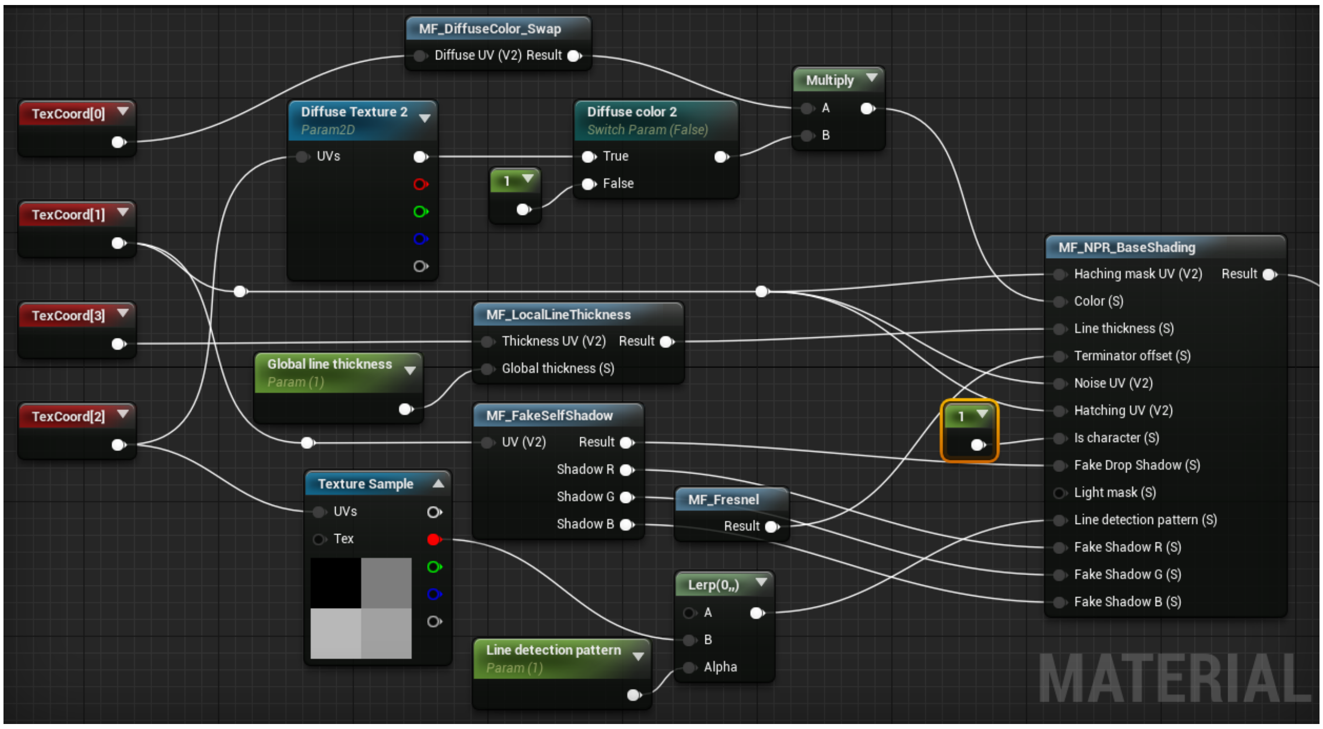


**Figure 37:** *The material uses four UV channels, although the features have eight separate UV inputs. Image from the Material Editor of Unreal.*

## 2.5 What we couldn't do

Because of time or technical limitations, there are some things that we were not able to achieve on the first episode:

### Hatching

We already implemented a feature allowing for hatching in the shadows we fake on the characters, but the result is not quite there yet.

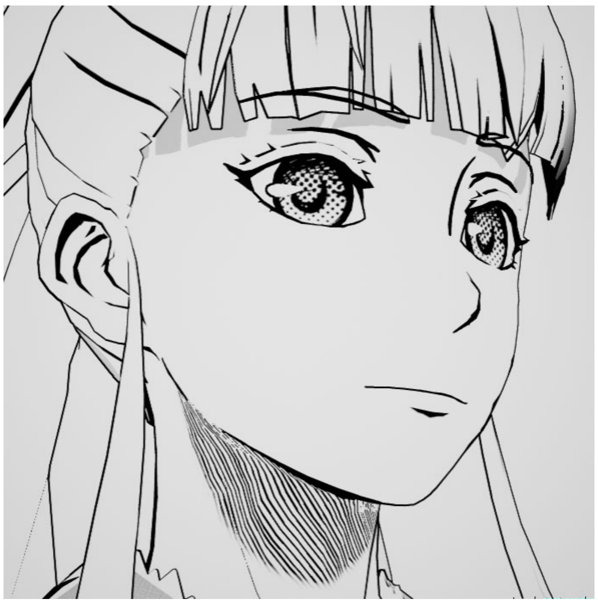


**Figure 38:** *Hatches under the character's chin.*

In this manga the hatching is usually flat and doesn't follow the volume of the object it is drawn on. Also, hatching strokes are usually a bit bent since they are drawn by hand pivoting on the wrist joint. Because the texture has to be tiled (again to avoid huge texture size and keep a clean visual) they can't have bent strokes.

### Clean line thickness

At the moment when we reduce the line thickness using the texture map we talked about above, white artifacts can be see around the thin lines (figure 39), where the line would be if it was at a normal thickness. It's not a huge deal since it's still quite small and not very visible, but that's a point to improve in the future.

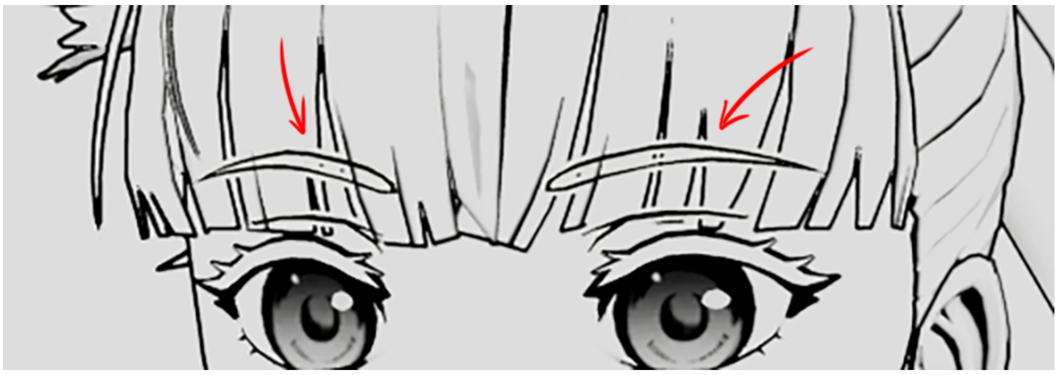


**Figure 39:** *White halo appearing on the eyebrows.*

### Distance specific visuals

In this manga (and others) when characters are far in the distance some of their features start to disappear (figure 40): nose, eyes, mouth, etc. In the future it could be a feature we want to implement, in order to both follow the manga visuals more faithfully and reduce the noise caused by small elements.

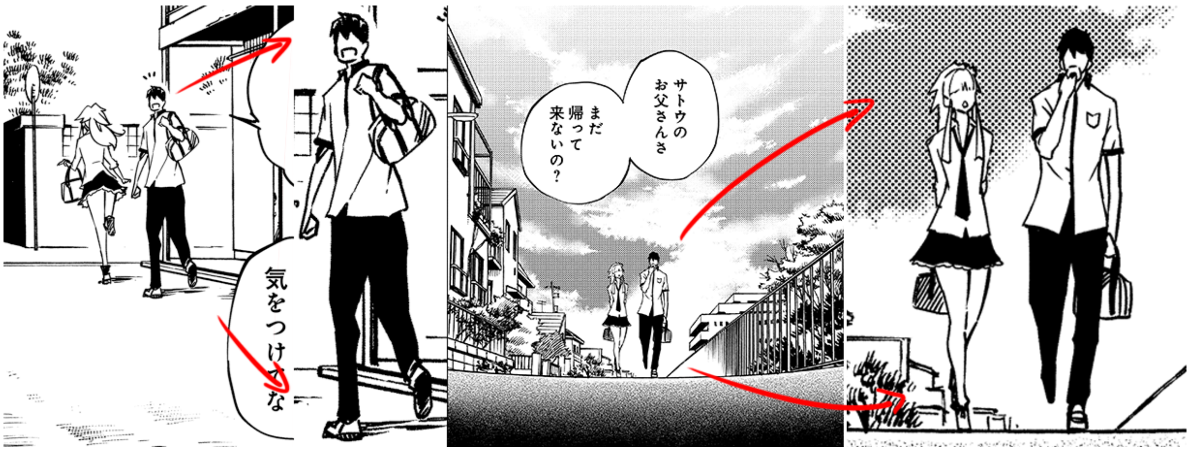


**Figure 40:** *Manga representation of characters in the distance.*

### Clean lines in the distance

At the moment the lines are resolution dependent and their thickness does not change depending on the distance from the camera (/user), because of this they become really thick when the character is small in the distance (figure 41). Combined with the point above, fixing this would improve the visuals a lot.

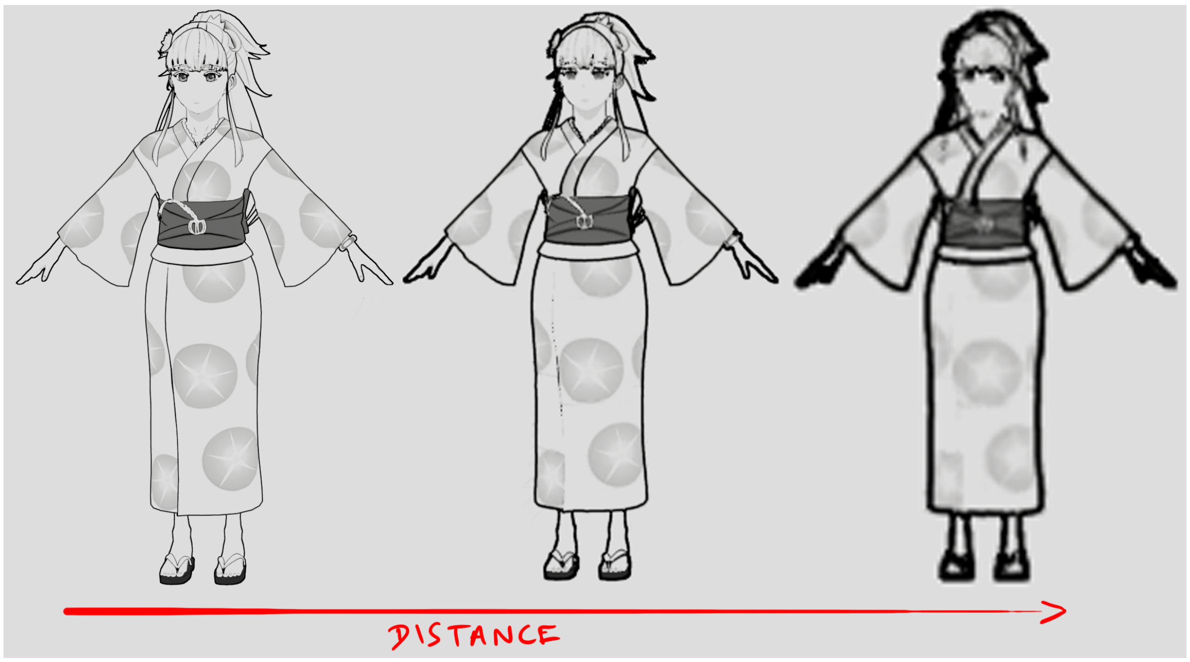


**Figure 41:** *Appearance of character lines as rendering distance increases.*

## Manga specific SD faces

In many mangas, character sometime are drawn SD or “Super Deformed”: their head gets bigger and the face and body features change (figure 42). That's something that is very specific to manga and it would add value to have it in the future.

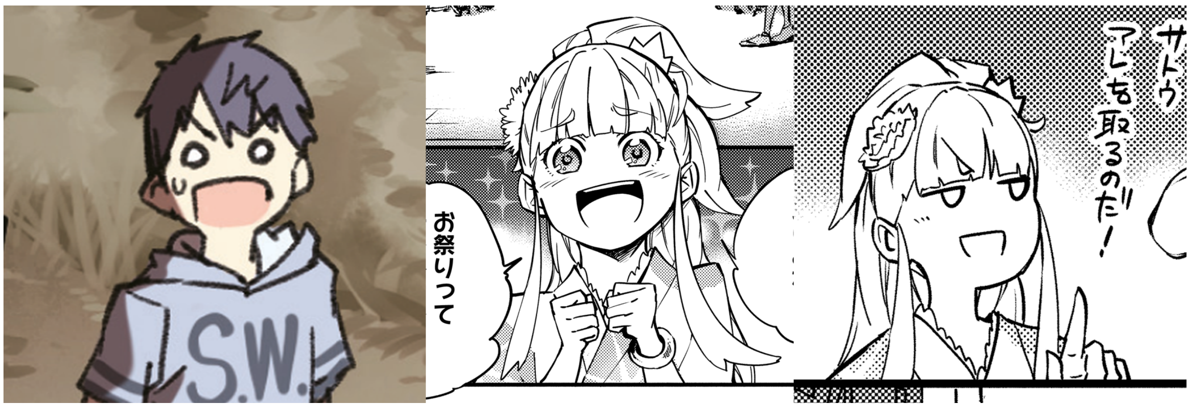


**Figure 42:** *Examples of characters drawn with a SD style in the original manga.*

# 3 Implementation side

## 3.1 Toon shading

Once the artists had described the shading they wanted and determined the controls they wanted (section 2.2), implementing it was straightforward. The function used is essentially a variation of a two-step function, with a few adjustments.

The first adjustment is the control over how fast the function transitions between steps. We described it as the “terminator sharpness”, and implemented it as a simple smoothstep, with its position and spread exposed as parameters. Attention had to be paid to corner cases, like the behavior close to the 0 and 1 or when the terminators are close to each other, so the result would match the artists' expectations.

The second adjustment is the slope of the function between the black and white terminators. The area referred to as “grey” could be uniform or a gradation, depending on the material. We implemented this as a linear interpolation between a uniform “grey value” and a wrapped diffuse (see listing 1).

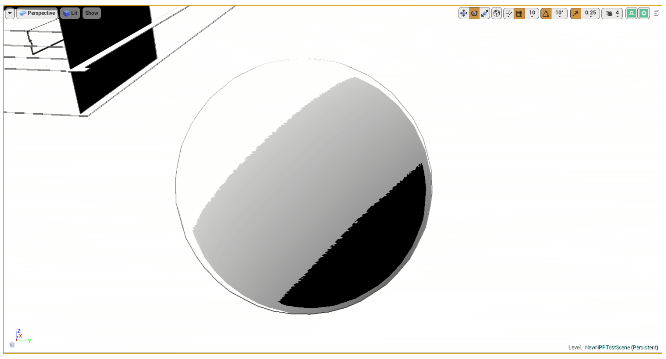
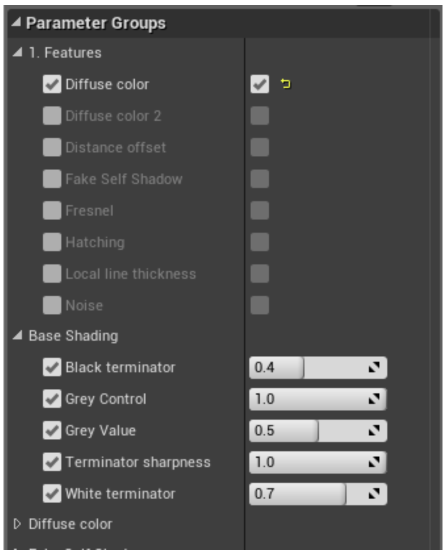


**Figure 43:** *Shading test in Unreal Engine.*

In fine the parameters exposed to artists are:

- *White terminator*: the position of the separation between the full white part and the grey part.
- *Black terminator*: the position of the separation between the grey part and the full black part.
- *Grey value*: the brightness of the grey part.
- *Grey control*: an adjustable value between a complete uniform grey and a gradient from black to white.
- *Terminator sharpness*: an adjustable value from a soft to a sharp transition between the white, grey and black parts.

Those parameters were sent directly to the shader, which was implemented in a few lines of HLSL, with some later obfuscation for optimization purposes.


Parameter Groups

1. Features
- ✓ Diffuse color ✓
- Diffuse color 2
- Distance offset
- Fake Self Shadow
- Fresnel
- Hatching
- Local line thickness
- Noise

Base Shading

| | |
|---|---|
| ✓ Black terminator | 0.4 |
| ✓ Grey Control | 1.0 |
| ✓ Grey Value | 0.5 |
| ✓ Terminator sharpness | 1.0 |
| ✓ White terminator | 0.7 |

Diffuse color

**Figure 44:** *Example of material parameters exposed to artists in the Material Editor.*

```
float Wrapped = saturate(dot(N, L) * 0.5 + 0.5);
float BlackToWhite = saturate((Wrapped - BlackTtor) / (WhiteTtor - BlackTtor));
float GreyColor = lerp(BlackToWhite, GreyValue, GreyControl);
```

**Listing 1:** *Linear interpolation between a wrapped diffuse and a defined grey value.*

The Fresnel effect was implemented by modifying the terminator position that the artist provide. We experimented with other types of effects as well, like adding noise to get a grittier terminator for example (figure 43). Implementing those in the material asset rather than directly in the base shading gave more flexibility to test different effects. However, it was important again to be careful of corner cases and keep the artist's intent in mind.

Similarly, various other features were implemented in a base material, which exposed all parameters for artists to use. The Material Editor of *Unreal Engine* allows to use "Static Switch Parameter" nodes, which can toggle parts of a material implementation at compile time. We relied on this to implement all features in a small number of base materials, with all the toggles and parameters exposed in a coherent way (figure 44).

## 3.2 Line detection

Lines as a representation for edges, borders and creases are a strong component of drawings in general and manga in particular. The base technique we used to draw such lines is a convolution-based edge detection. As mentioned in the art part however, this technique alone will not capture all the meaningful details. It has to be combined with other techniques.

There are several convolution operators for estimating the local gradient of an image. This gradient can then be used for edge detection. The [Sobel68] operator is well known, but is not the only one available: [Roberts63], [Prewitt70] or [Scharr00] have proposed similar operators.

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \times A \qquad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \times A$$

$$G = \sqrt{{G_x}^2 + {G_y}^2}$$

**Figure 45:** *The Sobel operator.*

We have decided to used the [Scharr00] operator, which has a better rotational symmetry for a similar cost.

$$G_x = \begin{bmatrix} +3 & 0 & -3 \\ +10 & 0 & -10 \\ +3 & 0 & -3 \end{bmatrix} \times A \qquad G_y = \begin{bmatrix} +3 & +10 & +3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix} \times A$$

$$G = \sqrt{{G_x}^2 + {G_y}^2}$$

**Figure 46:** *The Scharr operator.*

We implemented edge detection as a post-process in *Epic® Unreal® Engine 4*, which means the input we have at our disposal is either the geometry buffer in deferred shading, or the render buffer and depth buffer in forward shading. For performance reasons, we decided to use forward shading as soon as it was available (Unreal Engine 4.14).

The post-process applies the gradient operator to various available geometry data, filters the resulting gradients with parameters exposed to the artists, and finally combines them by keeping the maximum value. To help the artists retain better control over the lines, the post-process applies a different set of filtering parameters on characters and environment, as well as foreground and background.

Here is a concrete example with final assets. This is near final image, without the post-processing based lines.

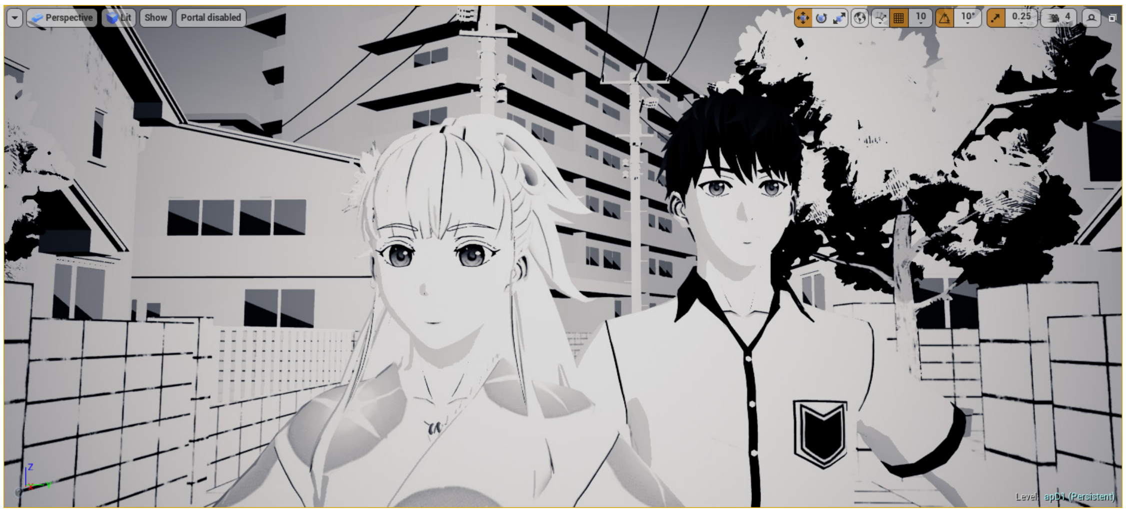


**Figure 47:** *Rendering result, without post-process based lines.*

### 3.2.1 Depth-based lines

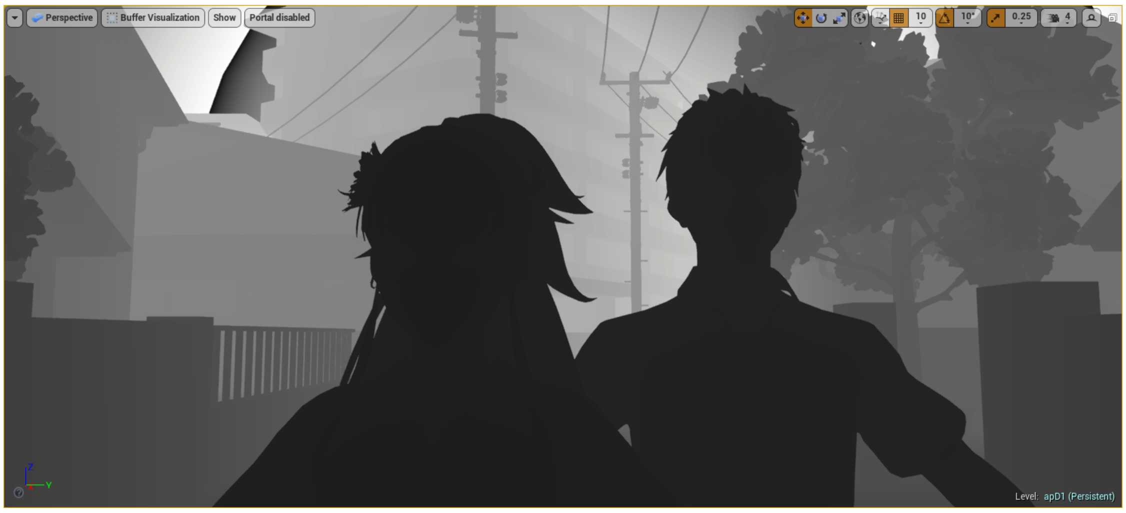


**Figure 48:** *The depth buffer.*

We start with the depth buffer. If we simply evaluate the gradient, even after some manual filtering, we cannot get a decent result. We have to choose between coarse lines between elements far apart, and fine lines on close elements, but we cannot get both.

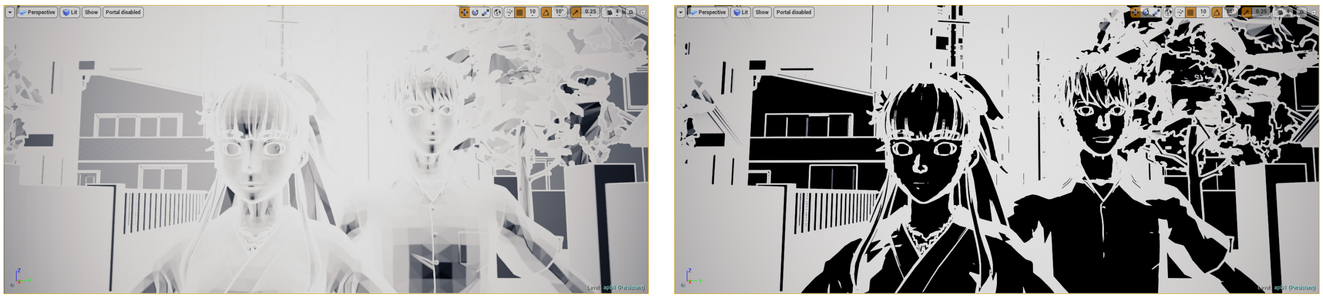


**Figure 49:** *Gradient of the depth, as is and after manual filtering.*

A first improvement we can implement is to scale down the gradient with distance: we want to retain small details close up, but only keep strong variations for elements that are far away.

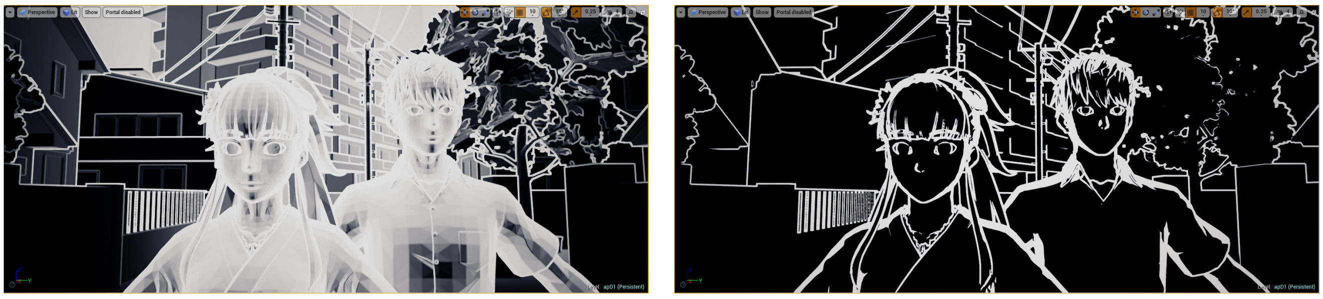


**Figure 50:** *Gradient of the depth, scaled with distance.*

This is better, but we have two problems. In the distance, some of the detail that we would have preferred to keep is now gone. For example the window frames on the left building have disappeared. This balance between too much and too little detail has been a recurring problem. Alas we haven't found an ideal solution yet.

The other issue here is the thickness of the outline at grazing angles. For example the shoulders and sleeves of both characters have large white patches, which could be interesting for certain art styles, but are not desirable for our material. We want thin lines.

Intuitively, at grazing angles the gradient will tend to be large even if the surface is featureless. We can use the surface normal to cancel this effect.

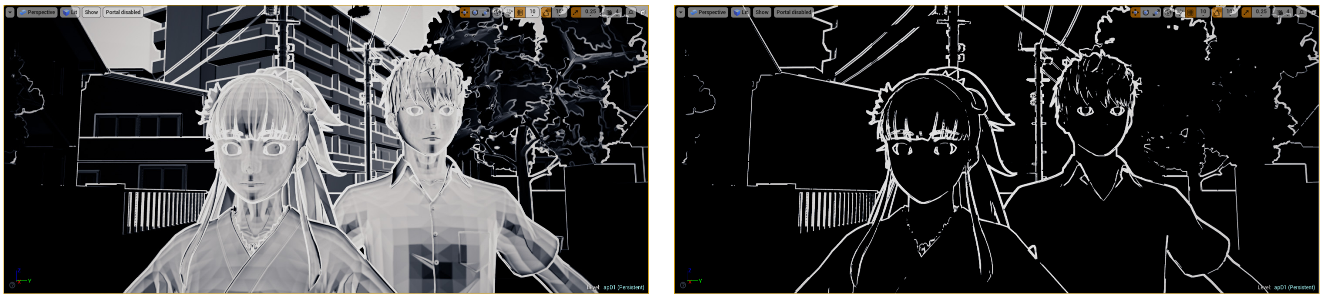


**Figure 51:** *Gradient of the depth, scaled down at grazing angles.*

We now have clean lines. However some features like the boy's hair or the electric power lines were already well defined and don't need additional lines. Moreover, some face details need to have thiner, more delicate lines. So finally, as mentioned in section 2.3, we introduce an artist-controlled local line thickness parameter.

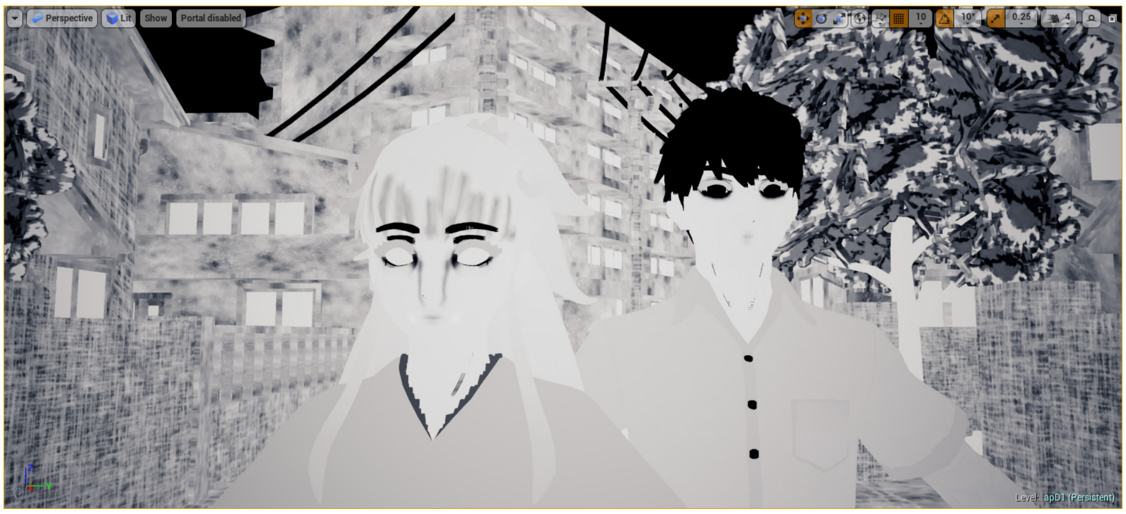


**Figure 52:** *Local line thickness.*

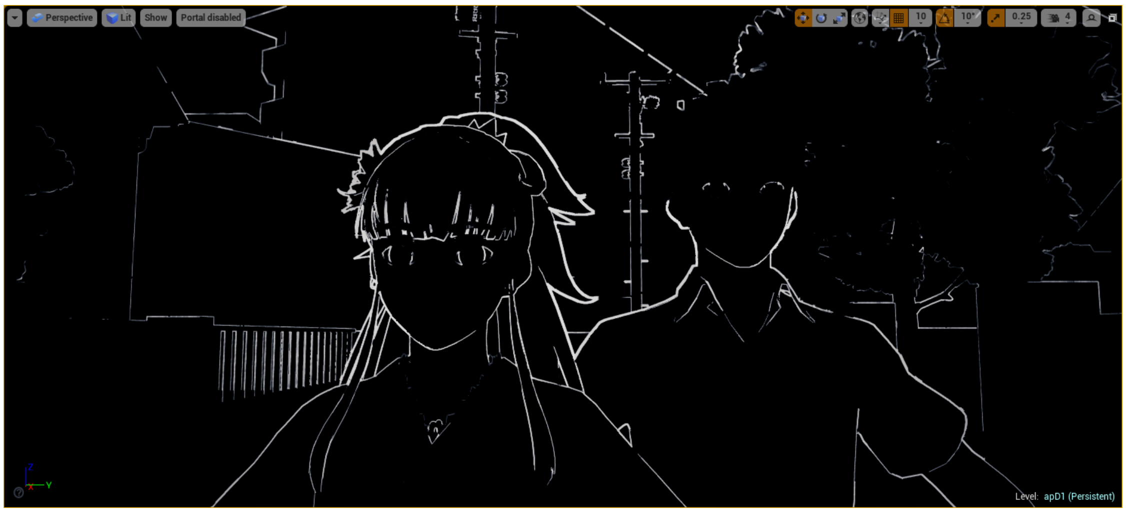


**Figure 53:** *Final lines obtained with depth, and artist-driven local thickness.*

### 3.2.2 Normal-based lines

We now have clean lines, but we are missing various features. The depth gradient tends to only highlight outlines, and miss elements like creases where we also want to see lines appear. To detect those, we need the gradient of the surface normal. We modified the forward shading path so it would also provide normals. The proper way to do that would have been to add a normal buffer. However observing that the art style didn't use the entire RGB space but only black and white, we could use two of the color buffer channels to include $N_x$ and $N_y$, which was a smaller and simpler modification to the engine than adding a new buffer.

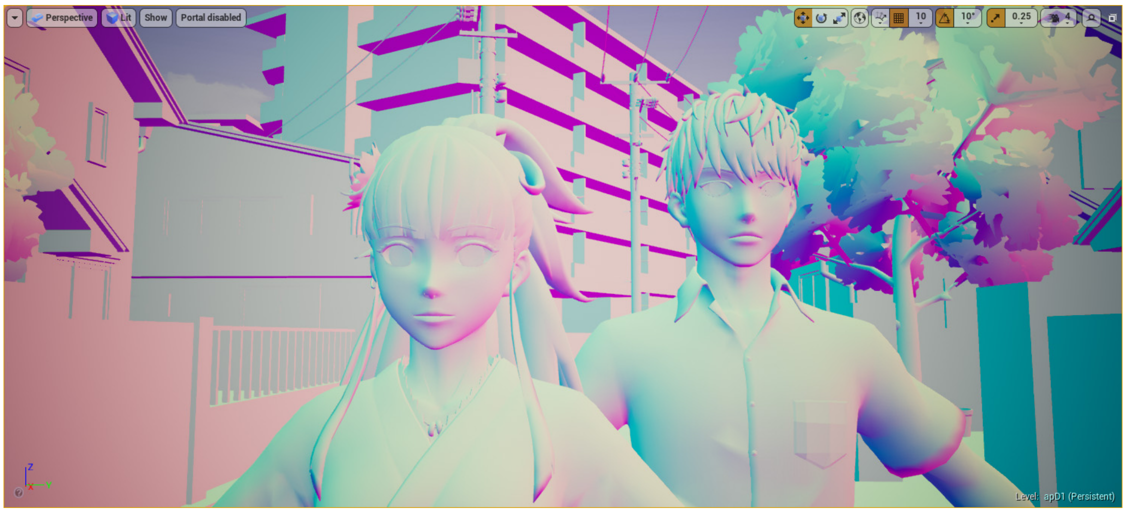


**Figure 54:** *The scene normals.*

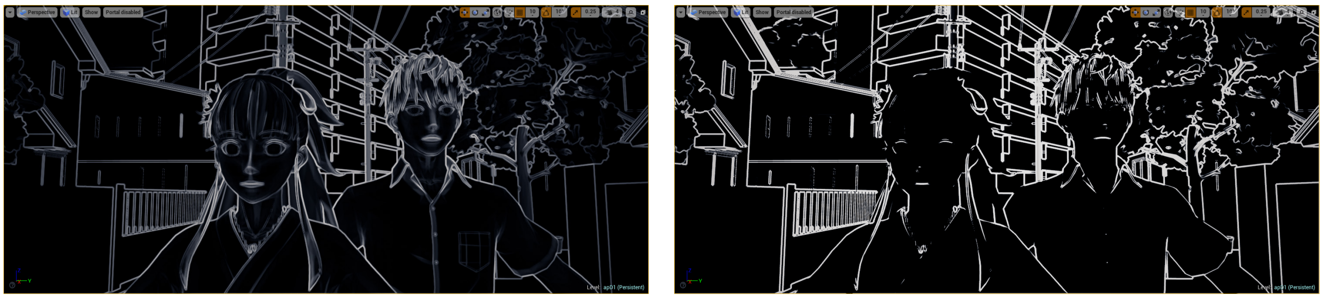


**Figure 55:** *Gradient of the normal, as is and after filtering.*

We apply the same line thickness parameter, and combine the depth-based lines with the normal-based lines. Notice how we now have lines on the mouth and window frames, and how the tree and the walls are better defined.

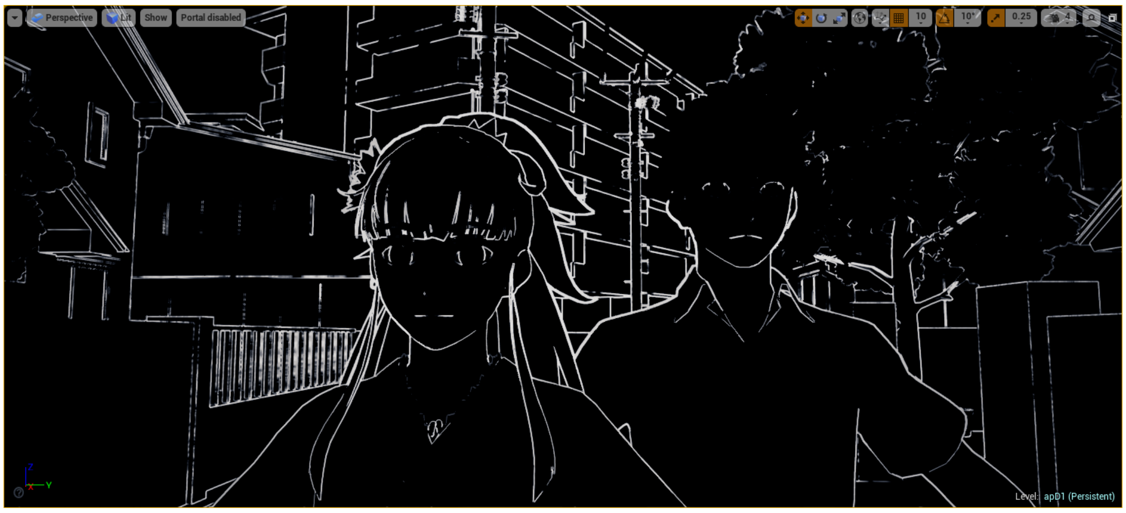


**Figure 56:** *Line detection using both depth and normals.*

### 3.2.3 Texture-based lines

Lastly, as explained in section 2.3, there are some parts where we want lines, but which both the depth gradient and normal gradient fail to detect. There are also cases where we would expect lines to be detected, but the balance between too much and too little detail doesn't give a satisfying result. For those cases we decided to introduce an artist-authored coloring.

We could manage to squeeze 8 different values in the color buffer. Artists assign those values to the mesh polygons with a pair of 2×2 textures and manual UV mapping. One texture (4 values) is used for characters, while the other one is used for environment.

Evaluating the gradient of that value allows us to add clean lines wherever needed. This technique helped artists get a lot of control over lines, with more predictable results than the normal or depth gradient, at the cost of a few bits in the render buffer and an additional convolution filter.

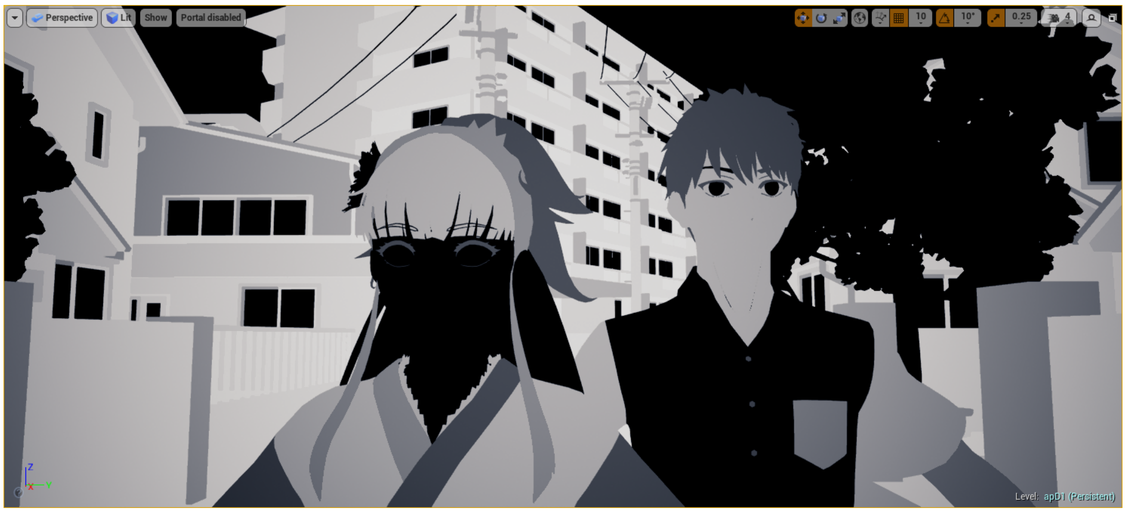


**Figure 57:** *The custom value used to add the missing lines.*

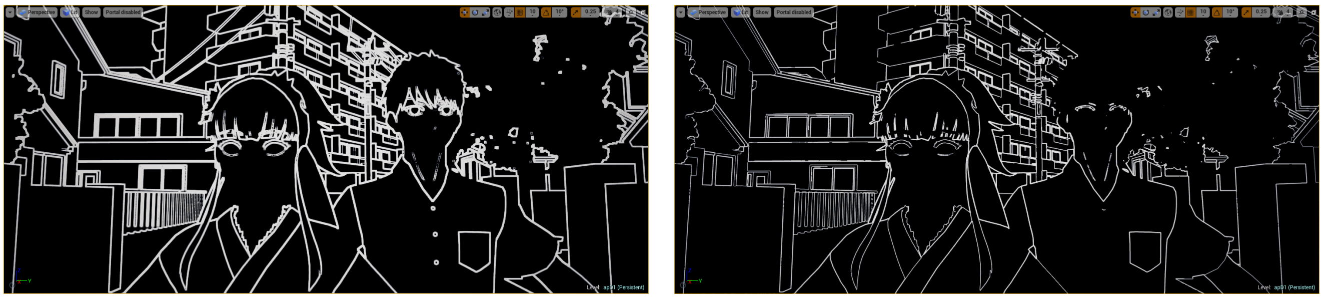


**Figure 58:** *Gradient of the custom value, and the result with local line thickness.*

Here is the final result when combining all three convolution-based lines, with parameters controlled by artists.

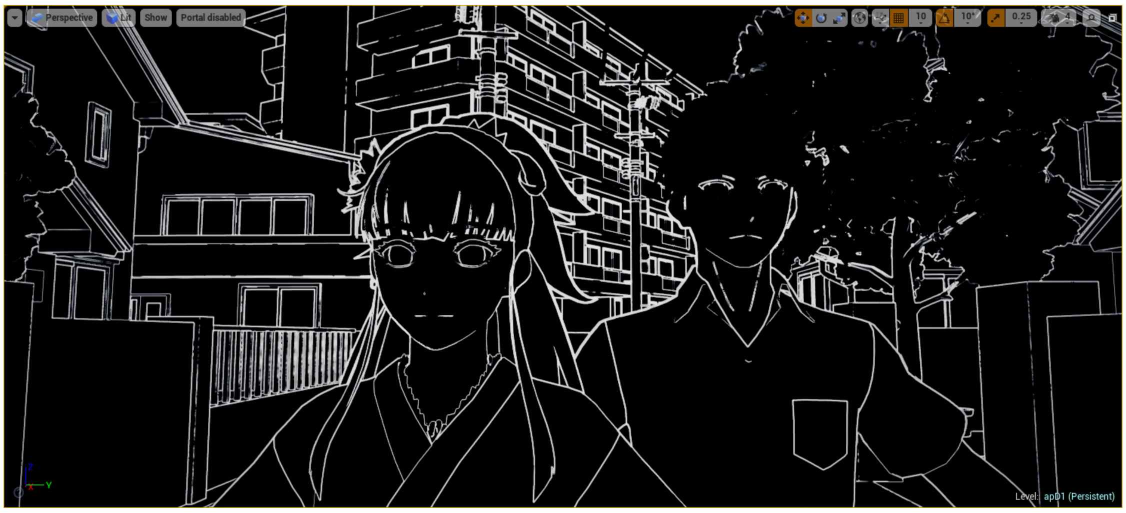


**Figure 59:** *Line detection using both depth, normals and the custom texture.*

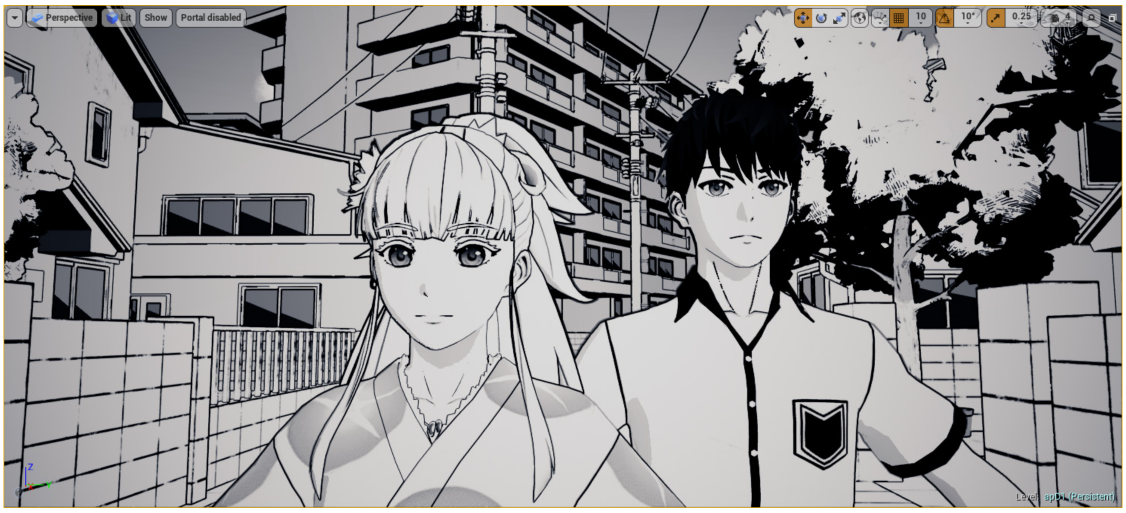


**Figure 60:** *Final result.*

## 3.3 Line transparency

A common feature of character faces in manga is the overlapping of eyebrows and hair, communicating a certain translucency of the hair. To reproduce this effect in VR, we move the eyebrow slightly toward the camera with a simple material function that offsets the vertices. Since the offset is evaluated independently for each eye in VR, the eyebrows are perceived to be in the same position as before.

Conflicting depth cues is often a source of discomfort in VR. In this case however, despite the hair being perceived to be closer than the eyebrows on top, it doesn't trigger a reaction of discomfort from the viewer.

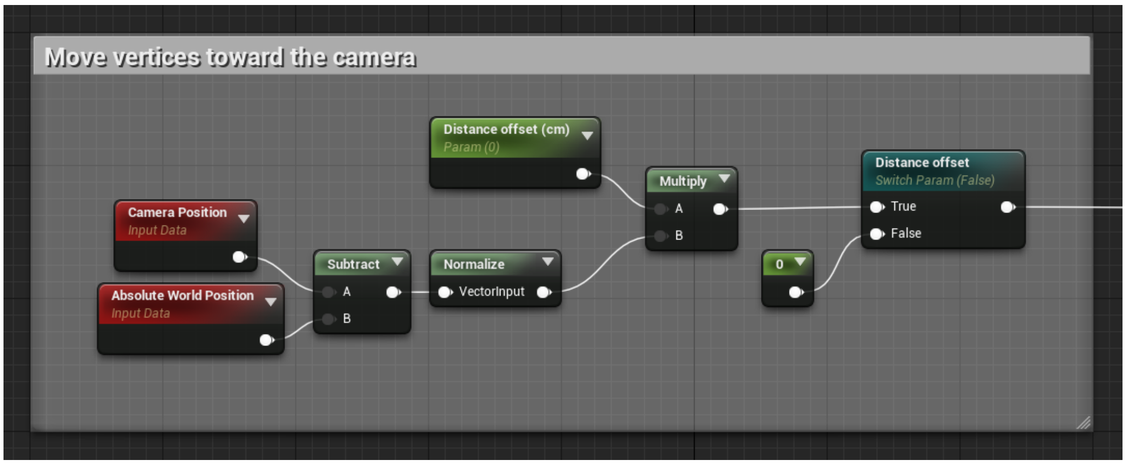


**Figure 61:** *A simple material snippet to offset vertices toward the camera, in the Material Editor of Unreal.*

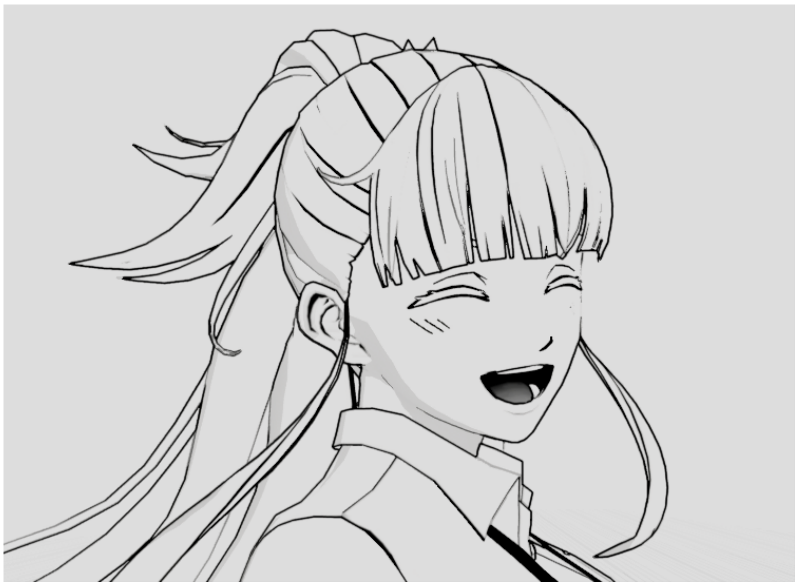


**Figure 62:** *The Hime character, without vertex offset: the eyebrows are masked by the hair fringe.*

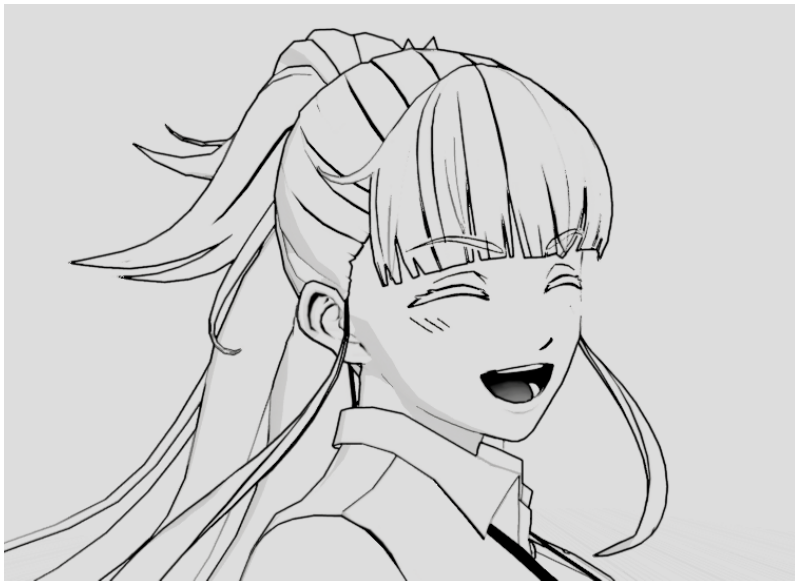


**Figure 63:** *The Hime character, with eyebrows vertices offset by 5cm.*

This simple vertex distance offset trick gives satisfying results, but it comes with limitations. The range of values that can be used for the distance is limited: if the offset is too small the eyebrows will intersect with the hair, while if that offset is too large the eyebrow might be visible through the nose in profile.

## 3.4 Hatching

As mentioned in section 2.5, the hatching present in the released version is very simple and not completely satisfying yet. Still, some of our prototyping experiments are worth discussing.

One difficult aspect of translating a hand drawn medium to VR is the change of frame of reference. Drawing is typically done in a static 2-dimensional space. VR is a dynamic 3-dimensional space. So which frame of reference should we use?

Our experiments show that using object space or regular texture coordinates of objects works very well for hatches of environments. Because the coordinates are static, the texture is stable spatially and temporally. Moreover, the artistic look is maintained and even in VR, the intent of hatches is understood just like it is in 2-dimension. The strokes follow the mesh curvature which is a desirable feature for simple geometric shapes (figure 64). For organic shapes however, as shown in section 2.5, this can be problematic (figure 38).

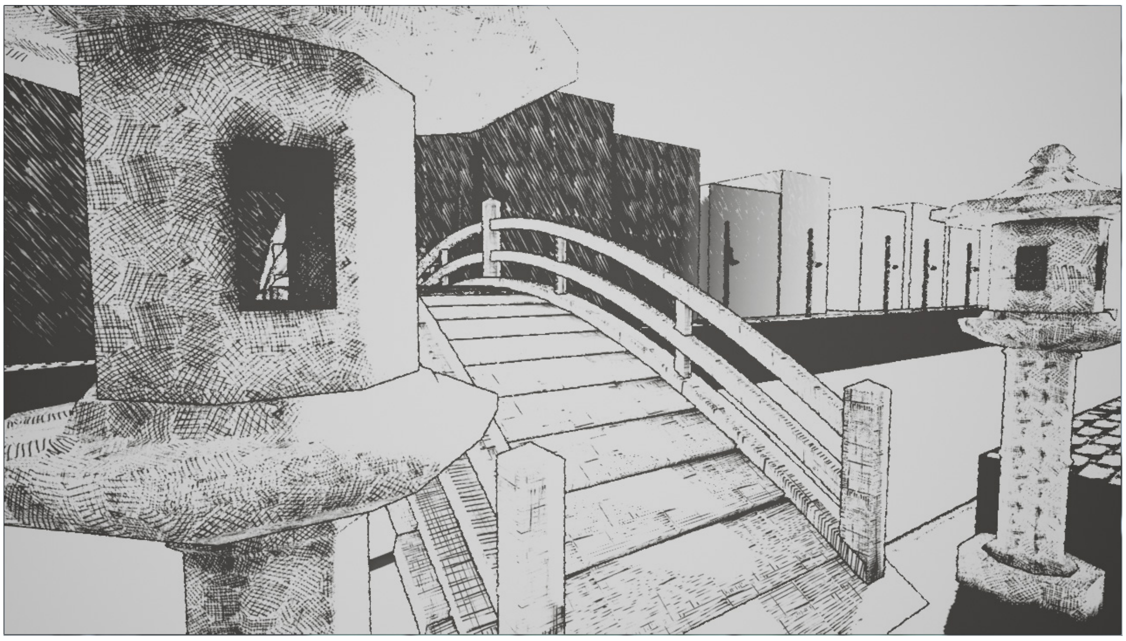


**Figure 64:** *Early prototype of hatching, with material type conveyed by different textures scanned from hand drawn hatches (some models © SbbUtutuya).*

In drawings and manga, the density of hatches tends to be uniform. In particular, it is not affected by distance. You might notice in the illustration above that the hatching of the foreground, bridge, and background have similar visual scale. This was done automatically with a simple texture coordinates trick and some blending.

Given a distance and original texture coordinates, we generated two sets of texture coordinates, and linearly interpolated between the corresponding resulting hatching patterns. As the distance doubled, so would the texture scale. Due to the visual nature of hatches, the transition from a scale to the next was natural and difficult to notice.

Finally the hatched areas can either be static or dynamic. They can be baked in the color texture, controlled by a mask (in the case of fake shadows for example), or controlled by the lighting or shadow casting (note the cast shadow on the bridge in figure 64).

## 3.5 Half-tone and paper texture

Unlike hatching, half-tone is an effect we didn't manage to implement in a satisfying way. Creating a convincing half-tone filter for the screen is simple (figure 65). The problem in VR however, is the frame of reference again.

- If the half-tone is done in screen space, it will be perceived by the viewer as a display artifact: the so called “glass-door” effect.
- If it is done in spherical coordinates, two problems happen. First, we have to map a tiling pattern onto a sphere. Second and more importantly, now the effect will be perceived by the user as an infinitely distant background: a variation of the glass-door effect. At the same time, surfaces with uniform color will now look like they are transparent.
- If it is done in world space with a tri-planar projection, a different set of problems happen. The half-tone pattern will crawl on moving objects, which is questionable and distracting. And while tri-planar projection gives acceptable results on flat surfaces, curved surfaces will show ugly patterns and Moiré artifacts.
- Finally if it is done in object space with a tri-planar projection, it fixes the crawling problem, but the undesirable mapping artifacts remain.

In the end, as mentioned in section 2.2, we decided to abandon the idea and used a gradient instead. For similar reasons, we couldn't find a convincing way to convey the idea of a paper texture.

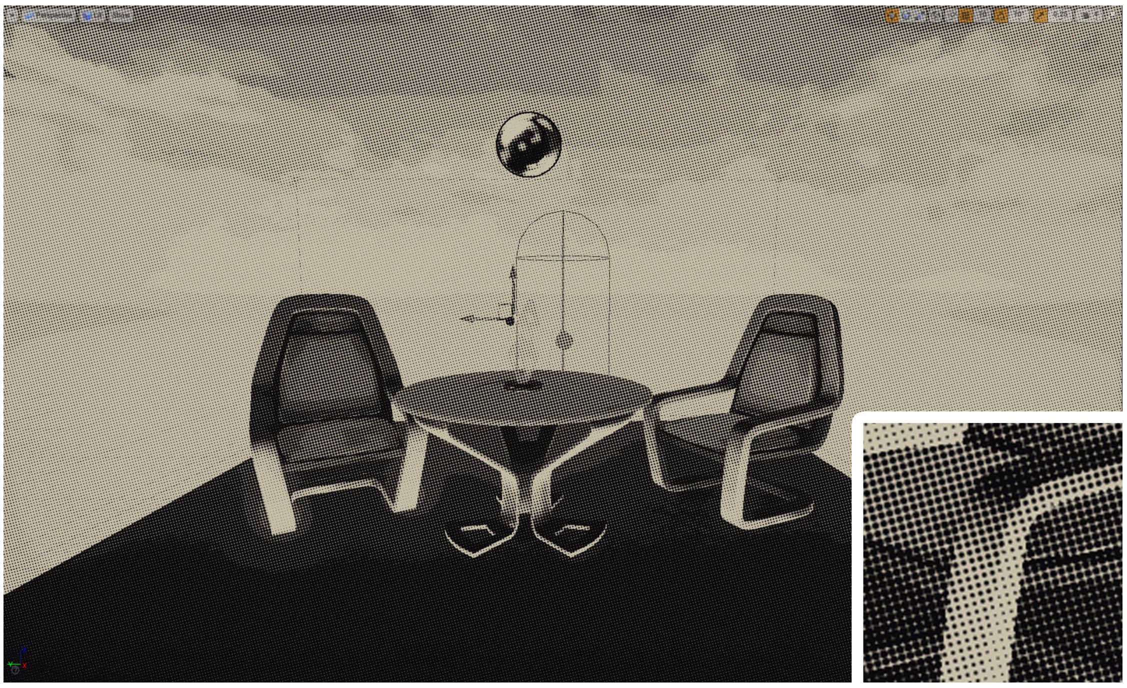


**Figure 65:** *Early prototype of half-tone filtering, applied to Unreal Editor's default scene (© Epic Games, Inc.).*

## 3.6 Transition between frame and immersive 360°

Have you ever been watching a film in a theater, and suddenly noticed and became conscious of the theater room around you? As if until a moment earlier, you were so absorbed in the film that you had forgotten where you were, the seats, the other spectators, and even the screen itself. When and how do that transition from sitting in a theater to being absorbed in a film happen?

During this project we explored various ways of creating a similar transition, from reading a manga support, to being immersed in its content.

### 3.6.1 First prototype

The very first experiment we did was very crude. It just consisted in placing a rough frame mesh, a very large white plane with a rectangular hole, in front of a 3D scene in VR. We tested how we perceived it depending on its size and position, how it affected *“presence”* and how walking through it felt.

The next experiment used shaders to create a geometry transition between a flat 2D frame and a 3D scene. A vertex shader would project the vertices of a scene on a planar frame, as if seen from a fixed point of view, while a fragment shader would discard the fragments outside the imaginary frame area. A parameter allowed to control the size of that frame and seamlessly transition between this projective transform and a regular 3D transform.

This was just an early prototype implemented quickly in the Material Editor of *Unreal Engine*, so it had limitations. In particular the vertex transform affected the lighting and caused Z-fighting. In a forward renderer though it should be possible to still compute lighting as if there was no flattening. We limited the Z-fighting problem by squeezing the scene down to depth of a few centimeters, instead of making it completely flat.

The take away of the experiment however, was the reaction of people who tested it. As they started, the scene would appear flat and limited to the bounds of the frame. The frame would then expand and as it did, the scene would gain depth, until it eventually covered the entire 360° field of view. Most people didn't notice the change of depth, and were surprised when they realized they were not just in front of a very large 2D frame, but actually inside of a fully 3D scene.

After these experiments we implemented what we called *Live Windows*.

### 3.6.2 Live Windows

The primary goal of Live Windows is to represent manga frames in the VR space. We needed a system to display separate 3D scenes in different frames. That system had to ensure the scenes wouldn't leak outside of their frame or into another scene's frame.

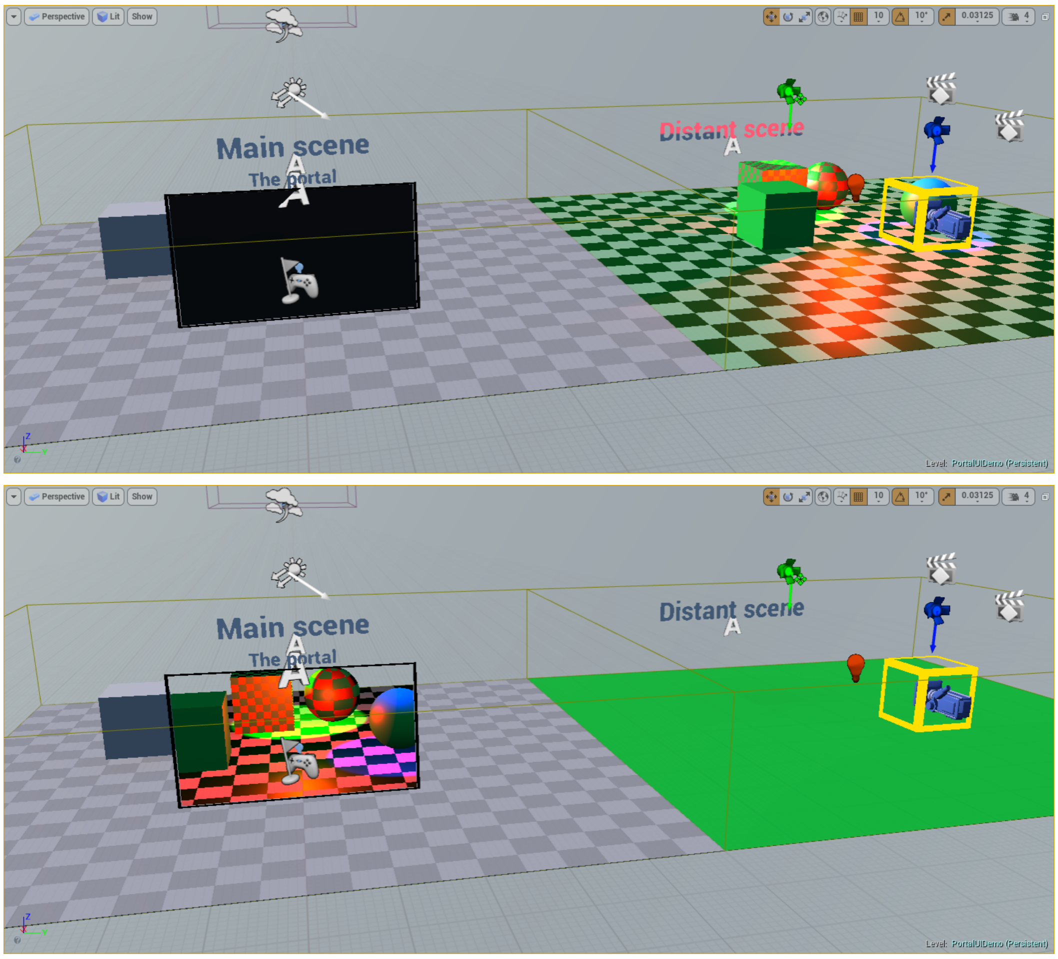


**Figure 66:** *Early rendering test of Live Window: the scene as viewed during authoring (top) and when the Live Window is active (bottom). The distant scene appears inside the frame.*

### Render-to-texture based approach

We started a render-to-texture based implementation using the `SceneCaptureComponent2D` component of *Unreal Engine*. As we did so, various issues appeared and required fixing. By default the capture applied a tone mapping, which was redundant with the main view's tone mapping, so we disabled the former. The scene in the capture didn't benefit from the temporal anti-aliasing (TAA), so we modified the capture rendering code to take the frame jitter matrix into account.

However, one persistent issue started to appear: the scene capture seemed to lag behind the main view. Not much, very little in fact, but just enough to feel a tiny spring effect when turning the head. The problem was difficult to identify because it was difficult to notice in the first place. We started looking for a culprit, created test levels meant to catch the bug red-handed, and even started to doubt some of our tools. If the bug was never visible on the screenshots, did it mean that our test was badly designed or that the screenshot was not faithful?

Finally, we identified the root cause of the problem. Because of the importance of latency in VR, sometimes referred to as “motion-to-photon” latency, *Unreal Engine* implements a clever technique so the user's head position used for the frame is as recent as possible. At the beginning of a new frame, the engine starts working with the last known head position, and prepares all the rendering: what is going to be visible, the culling, etc. Near the end of the frame, when the command list for the GPU is ready to go, the engine reads the current head position, and replaces the information in the command list before immediately sending it.

Our problem however, was that the render-to-texture, done at the beginning of the frame, was over long before the time that position update was done. At this point it looked like fixing this problem would require some major modifications in the rendering pipeline of *Unreal Engine*, so we abandoned the idea and looked for an alternative. This alternative, was the stencil buffer.

### Stencil based approach

This approach consists in marking the Live Window frames in the stencil buffer, then do several passes with different stencil tests to render the content of the different Live Windows. If this sounds expensive, it's because it is, and it required some significant effort to bring this feature within an acceptable time budget.

Here is an overview of what the rendering pipeline of *Unreal Engine* typically would do with our rendering configuration (listing 2).

1. Create views
2. Compute visibility (including frustum culling)
3. Render depth pre-pass
    1. Render opaque static objects
    2. Render opaque dynamic objects
4. Compute light grid
5. Render opaque objects
    1. Render static objects
    2. Render dynamic objects
6. Render velocity pass
7. Render translucent objects
8. Apply post-process

**Listing 2:** Typical rendering steps in Unreal Engine 4.

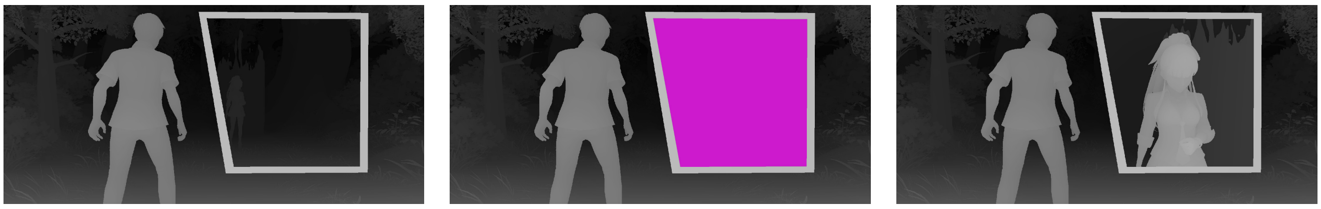


**Figure 67:** *The depth pre-pass steps with Live Windows: the main view depth is rendered (left); the Live Windows masks are marked to the stencil buffer (center); the depth inside the Live Windows is rendered (right).*

Here is how the pipeline looks with our implementation of stencil based Live Windows. First we write the corresponding masks in the stencil buffer during the depth pre-pass (listing 3 and figure 67). Doing this operation during pre-pass allows to use the depth buffer so objects can occlude the Live Windows.

```
1. Create views
2. Compute visibility
    ◦ For each LW, compute frustum culling
3. Render depth pre-pass
    1. Render opaque objects of the full-screen LW
        1. Render opaque static objects
        2. Render opaque dynamic objects
    2. Write to the stencil the masks of each LW
    3. Clear depth on the areas marked by LW masks
    4. For each LW except the full-screen one
        1. Render opaque static objects
        2. Render opaque dynamic objects
```

**Listing 3:** Live Window rendering pre-pass.

After this is done, all the rendering passes that we want to support are modified (listing 4). Instead of just rendering the main view, they iterate over the list of visible Live Windows and render the list of primitives that are visible through them with the corresponding stencil test (figure 69). The main view is also considered like a Live Window that happens to be full-screen.

```
4. Compute light grid
5. Render opaque objects
    ◦ For each LW
        1. Set the stencil test
        2. Render static objects
        3. Render dynamic objects
6. Render velocity pass
    ◦ ...
7. Render translucent objects
    ◦ ...
8. Apply post-process
```

**Listing 4:** Live Window rendering passes.

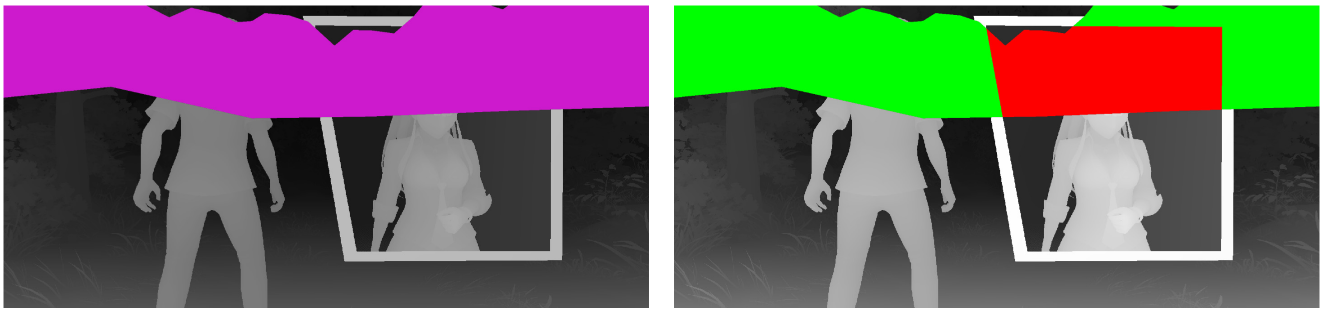


**Figure 68:** *Primitives rendered outside Live Windows with stencil testing.*

For performance reasons, we try to cull primitives as much as possible. Obviously we don't render primitives that are visible through a Live Window if that Live Window itself is not visible. We also try to do a reasonable frustum culling inside the Live Windows. Our authoring tools allow artists to use any shape they want, so we rely on the bounding box of the Live Window mesh to determine the frustum. As mentioned in section 4.2, it can be a problem if that bounding box is incorrect (this has been a source of glitches or sub-optimal performance on several occasions).

We also use that bounding box for scissor testing (figure 69). Even though the fragments outside a Live Window will get rejected by the stencil test, this test improves performance because it avoids the cost of rasterizing the triangles that are outside the Live Window. These are still high-level optimizations; see section 4 for more advanced optimizations.

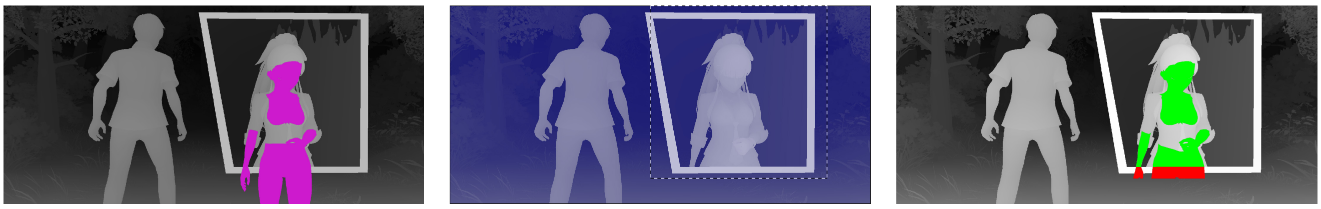


**Figure 69:** *Primitives rendered inside a Live Window (left) benefit from scissor testing (center) by avoiding rasterization ahead of stencil testing (right).*

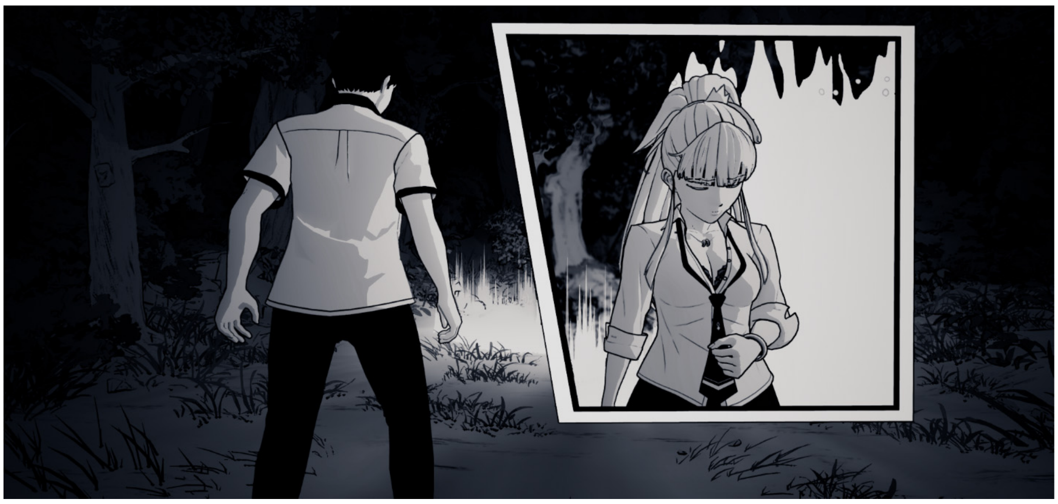


**Figure 70:** *Final result.*

By switching from a render-to-texture approach to a stencil mask approach, we lost the ability to create transparent frames. However, since we use temporal anti-aliasing (TAA), we can do some dithering based translucency, like in figure 71.

Regarding future directions, the Live Window mechanism doesn't support the flattening effect we experimented with during prototyping, but it is certainly an effect worth exploring more.

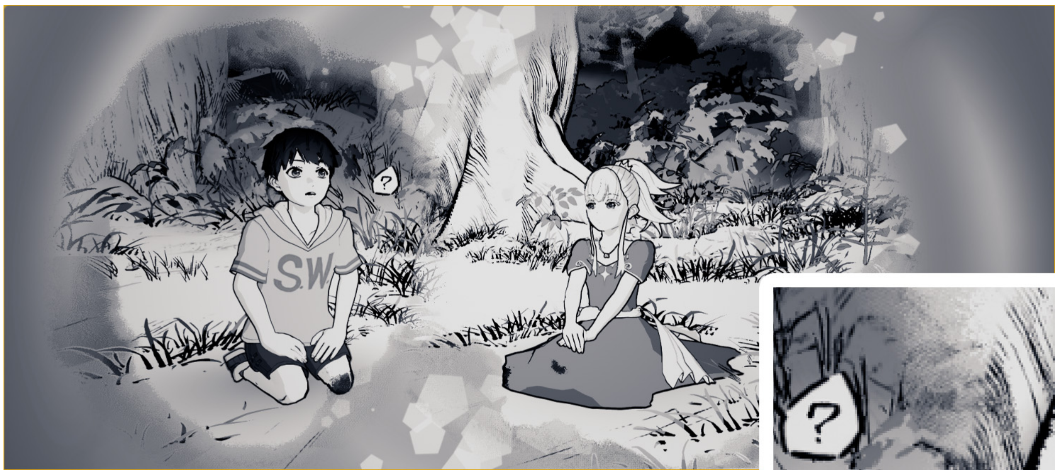


**Figure 71:** *Two Live Windows blend into each other using dithering and TAA.*

### 3.6.3 Diorama effect

In some scenes, we used a simple effect to get a powerful narrative tool: a diorama effect. In the post-process, *Unreal Engine 4* exposes enough information to know the position of a given pixel in world coordinates. We can change the color of elements of the scene based on that position. In particular, if we just turn entire parts of the scene to a uniform white, it will look like they are absent.

We used very simple distance functions to isolate elements of the scene like in a diorama. Here is a concrete example of the final render of a scene, without the masking effect.

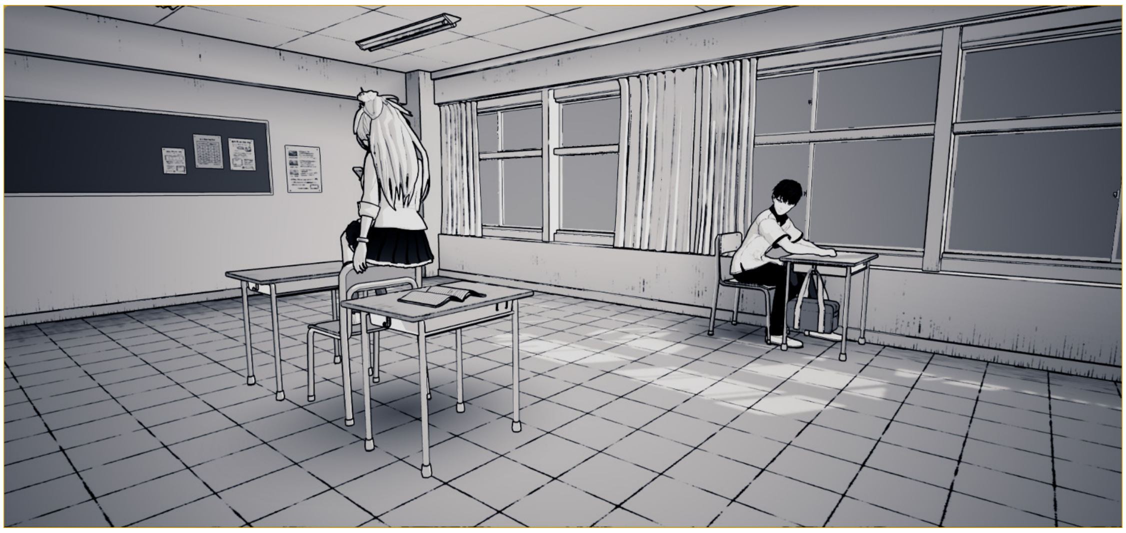


**Figure 72:** *The rendering result, without any masking.*

We can define a position of interest and compute the distance to that position, in the post-process. We can decide that the pixels beyond a certain distance will be masked out.

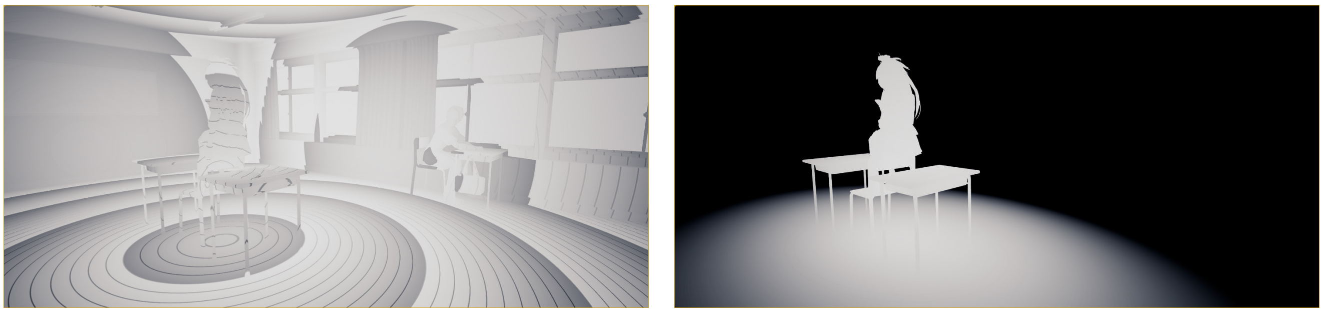


**Figure 73:** *The distance to a point, and a resulting mask.*

We can alter that distance by adding some noise based on the position. Of course the parameters of the noise are exposed to the artists.

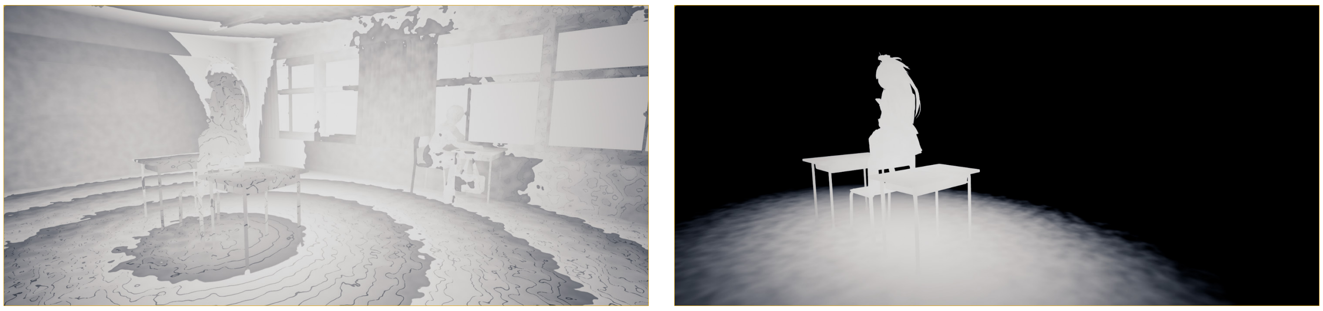


**Figure 74:** *The distance altered with noise, and the resulting mask.*

We can define a certain number of such points of interest. The computing cost will increase with the number of points, but we only need two or sometimes three of them.

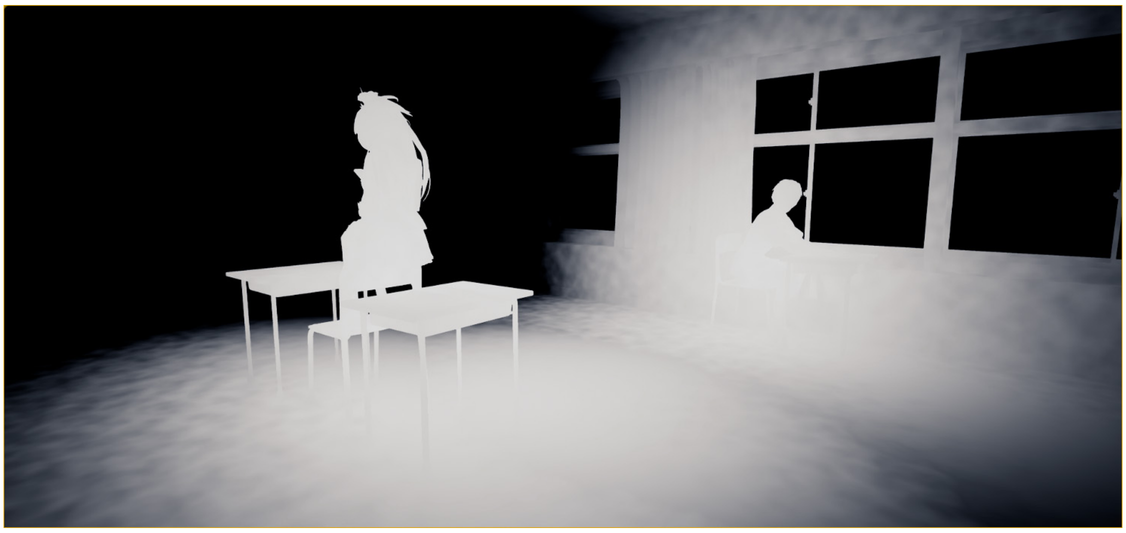


**Figure 75:** *Masking for the two characters.*

Here is the final result, as the mask is applied to the image. This simple technique works very well in VR, where is isolates the points of interest in a visually pleasing manner, and draws the attention of the viewer to them. We simply used the distance to a point, but with more complex distance functions [Quilez08] it is also possible to achieve more intricate effects.

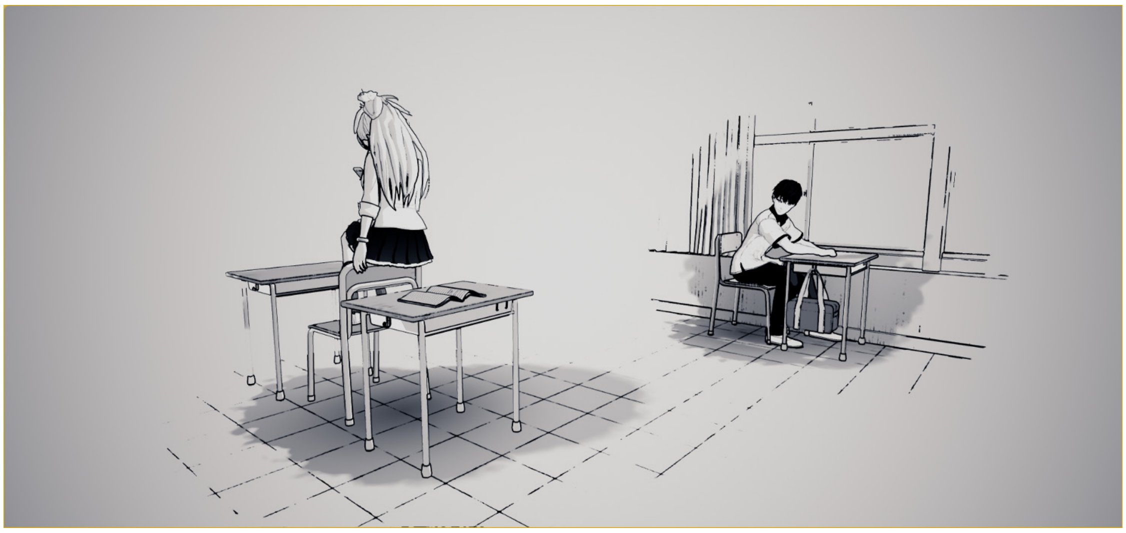


**Figure 76:** *The final result.*

# 4 GPU Optimizations

## 4.1 Performance Constraints

Let's start with an overview of the main performance constraints when dealing with VR, and continue further into the additional constraints specific to our project.

### 4.1.1 VR Performance Constraints

There are two primary performance constraints when targeting VR. The first of them is the high refresh rate of VR head-mounted displays (HMD), which is 90 Hz. This implies that we have to render a frame under 11.1ms. The second one is the high render resolution. The device has a resolution of 2160×1200, but the off-screen render target has an even higher resolution because of lens distortion. On Oculus Rift®, with 1.3 texture scale, we ended up with 3536×2064 (1768×2064 per eye) which is 7,298,304 pixels. This is a tremendous amount of work for the graphics processing unit (GPU) and a lot of memory traffic.

We found that it is safer to target 10ms per frame because of various GPU activities like frame compositing or work done for other concurrent processes running on the system. The application cannot just exclusively access all GPU resources. It should leave room for other processes to utilize GPU [Vlachos16].

We used vendor-specific profiling tools to measure correspondence between GPU time and dropped frames. We noticed that even when GPU time is in the range of 10.3 — 10.6 ms, there is a possibility of missing a frame. Alex Vlachos from Valve mentioned a similar safe zone in his “Advanced VR Rendering Performance” talk at GDC 2016, which he called a “10% idle rule” [Vlachos16]. We can confirm that we had the same experience.

### 4.1.2 Game Design Performance Constraints

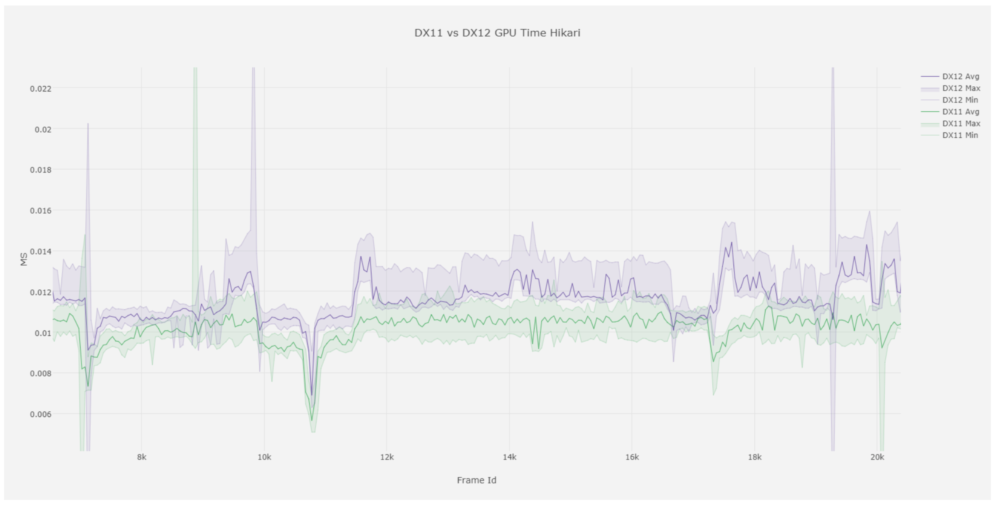
By design, we relied a lot on Live Windows which are essentially billboards the allow you to view a scene from multiple points of view at the same time. This feature created several performance challenges both on the GPU and CPU sides.

The amount of submitted geometry increased. In terms of complexity, the typical VR need of rendering a scene from both eyes was multiplied by the number of Live Windows on screen. Moreover, those views had incoherent directions and locations, and due to this fact we couldn't use a shared set of objects per frame and rely on instancing, for example, like it is usually done with a stereo pair of views. We had to cull and submit geometry per each Live Window and that increased CPU cost. We attempted to use *Microsoft® DirectX® 12* API in the *Unreal Engine 4* to benefit from parallel command list generation and potentially decrease the cost of submission of commands on the rendering thread, but we observed increased GPU frame time, and that was an unacceptable constraint for us. Furthermore, switching to *DirectX 12* late in production was a risky decision, so we stayed with *DirectX 11* despite the significant limitations such as tools we can use for profiling GPU (e.g. PIX for Windows) and optimization opportunities we had.

Live Windows affected not only the cost of scene submission on CPU but also increased GPU workload. The main reason was the reduced quality of CPU culling due to complex Live Window shapes and overlapping. As a result, the pressure on the graphics pipeline front end (geometry fetching/vertex shader executing/primitive setup and rasterization) on GPU increased too. This redundant work meant the effective utilization of the GPU decreased.

## 4.2 Performance Monitoring Process

When we started to dedicate more time to focus on performance and less on production, especially on the art side, it was essential to spend effort wisely and pick the most important directions to focus on. To our advantage, we could rely on the internal QA team in gathering necessary data to make right decisions.



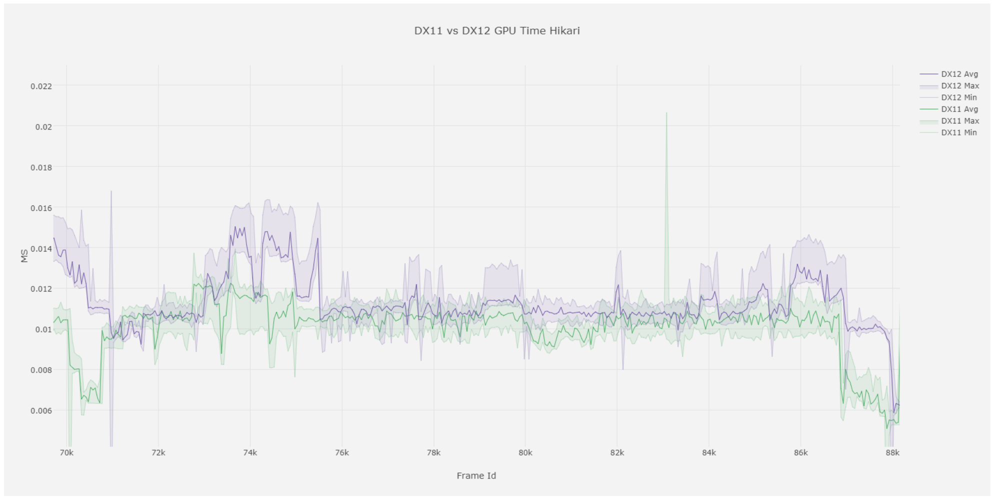


**Figure 77:** *An example visualization of the data gathered from profiling tools: two graphs show the GPU time for two different scenes under different graphics APIs.*

We asked our QA team to start capturing high-level performance metrics during play-through sessions they already had been doing on a regular basis. It is worth mentioning that in our case this was quite straightforward since our VR experience consists of animated sequences and reproducibility of the collected data was stable. Reports consisted in *.CSV* files containing various performance metrics, depending on the tool, for a particular level or set of levels. Having regular reports was important for us for several reasons:

- We used collected data to prioritize areas of detailed investigations by engineers.
- We tracked how various optimizations impacted the overall situation with performance, especially content-related optimizations.
- We could see performance regressions early if any of them occurred.

It took us some time to come up with a set of tools to collect performance metrics we can rely on to notice regressions or improvements easily across the entire VR experience. We used an automated *Python*-based tool (figure 77) to visualize data gathered with per-frame frequency and reduce it to something more readable.

It wasn't enough for us to use just CPU and GPU frame time, so we garnered a wider variety of performance metrics, especially some lower level metrics we were interested in. Some examples of high-level performance metrics we found useful to analyze included:

- CPU Time / GPU Time
- Number of dropped frames
- Compositor CPU / GPU Time
- Compositor GPU end-to-VSync

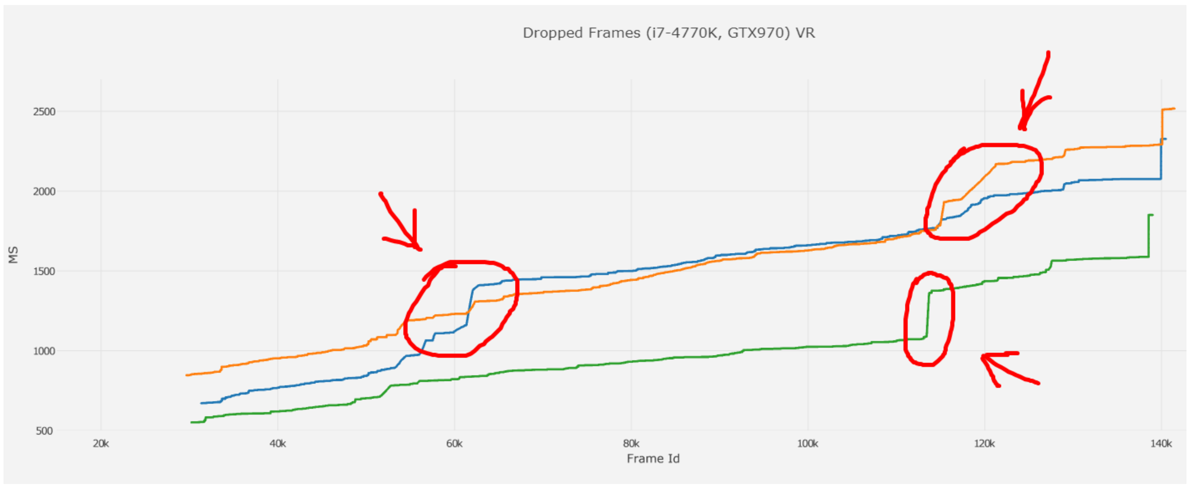


**Figure 78:** *A screenshot illustrating abrupt intervals/discontinuities on dropped frames graph.*

The most important metric was the number of missed frames (figure 78). We used this metric to identify abrupt intervals or discontinuities, which were a sign of regions where we had consecutively missed frames for a significant portion of the time. These were top priority places to fix, so engineering efforts were spent to investigate those issues, address them. If those performance problems were coming from the content side, a detailed feedback could then be provided to artists on how to avoid them.

However, we felt that doing just regular performance profiling wasn't enough. The results were not directly interpretable by the team and required engineering efforts to be translated into specific action points for the team. Although this is a usual approach when dealing with complex issues, for smaller but frequently appearing ones this could become a bottleneck, which we wanted to avoid. Instead, we aimed to reach a situation where individual team members clearly understood how they could contribute to optimizations. Communicating the importance of performance to the entirety of a team and help everyone feel responsible was surprisingly difficult. To facilitate our goals, we relied on visualization/debug tools in the engine and dedicated efforts to make sure that each individual team member could visualize and combat performance-related problems.

### Content Related Optimizations

After collecting and analyzing data from multiple problematic regions in our VR experience. We come up with groups of common issues on the content side:

- Too large local lights (wide solid angle for spotlights, large radii for point-lights and spotlights).
- Static objects with inadequate batching which increases the size of the resulting bounding volume, leading to sub-optimal frustum culling (figure 79).
- Too conservative LODs with the amount of reduction less than 25% per level.
- Excessive use of transparency cause overdraw.

**Figure 79:** *An example of inappropriate static batching. The distance between object is around 100-150m.*

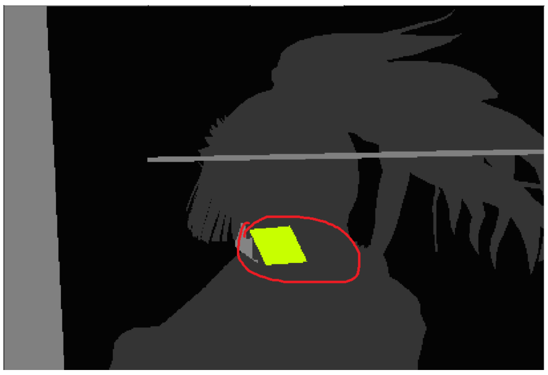
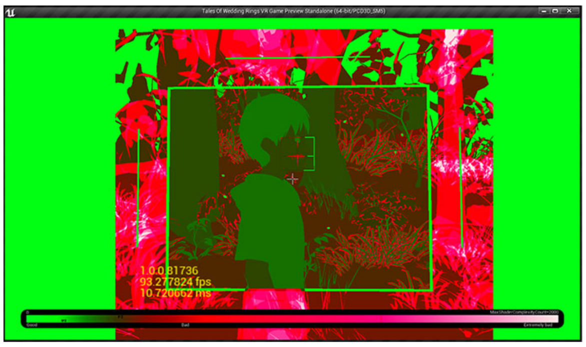


**Figure 80:** *8k polygon billboard in wireframe debug view.*


Tales Of Wedding Rings VR Game Preview Standalone (64-bit/PCD3D_SM5)
Good
Bad
Extremely bad
MaxShaderComplexityCount=2000

```
1.0.0.81736
93.277824 fps
10.720662 ms
```

**Figure 81:** *Scissor region too conservative as a result of incorrect bounding box.*

For some of those issues (like too conservative mesh LODs, transparency overdraw, or light complexity) it was essential that the art team members could use debug views to detect any content-related issue in a particular level. Some of the issues found by the art team can be seen in figure 80 and figure 81

## 4.3 Small, Yet Important Optimizations

In this section, we present three straightforward optimization cases which didn't require too much work but required careful investigation of the data from profiling tools and double-checking assumptions.

### 4.3.1 Forcing HiZ/EarlyZ Tests for Alpha-tested Materials

We started this optimization after noticing that HiZ/EarlyZ tests didn't work for alpha-tested materials. The evidence was in the number of executed fragment shaders comparing to the number of output fragments. This observation was unexpected because we used read-only depth and stencil states, and even though fragment shaders contained `discard` instruction, HiZ/EarlyZ should be enabled. However, in practice, it wasn't the case. We verified our assumption by annotating entry points with the `[earlydepthstencil]` keyword, which forces HiZ/EarlyZ tests, and measured performance again. The change in cost together with the ratio of the shaded fragments to the written fragments (1.0) helped to confirm that everything started to work correctly. This relatively small change helped us to reduce the cost of the base pass for some scenes dominated with foliage geometry.

Having cheap alpha-testing with HiZ/EarlyZ enabled was the critical feature for us because of two main reasons: we used alpha-testing for materials of low-detailed foliage LODs, and we relied on HiZ/EarlyZ to avoid expensive fragment processing outside of screen regions marked by a particular Live Window with a stencil bit.

### 4.3.2 Optimizing Light Fetching and Computation

As we mentioned earlier, we use the clustered forward renderer [Olsson12] [Persson13] in *Unreal Engine 4*. There are multiple reasons behind that:

1. Keep a possibility of comparing quality/cost between MSAA and TAA.
2. Avoid the bandwidth cost induced by writing a G-Buffer and re-reading it during a light pass in a classic deferred renderer. It's important to note that despite the fact that we have a NPR style, the number of material properties is still quite high. So evaluation of the lighting function right in the shader was a good option.

Clustered forward renderer elegantly handles light complexity. However we had an additional restriction from the Live Window system: we used a frustum voxel grid for the main view to store lights from multiple light windows, and that means that for all views we had some false-positive lights to process in a light loop.

### Eliminating Redundancies

We started with an assumption that this could affect performance significantly. To verify that, we added a simple debug view in *Unreal Engine 4* to help visualize the ratio between all lights and "valid" lights for the view.

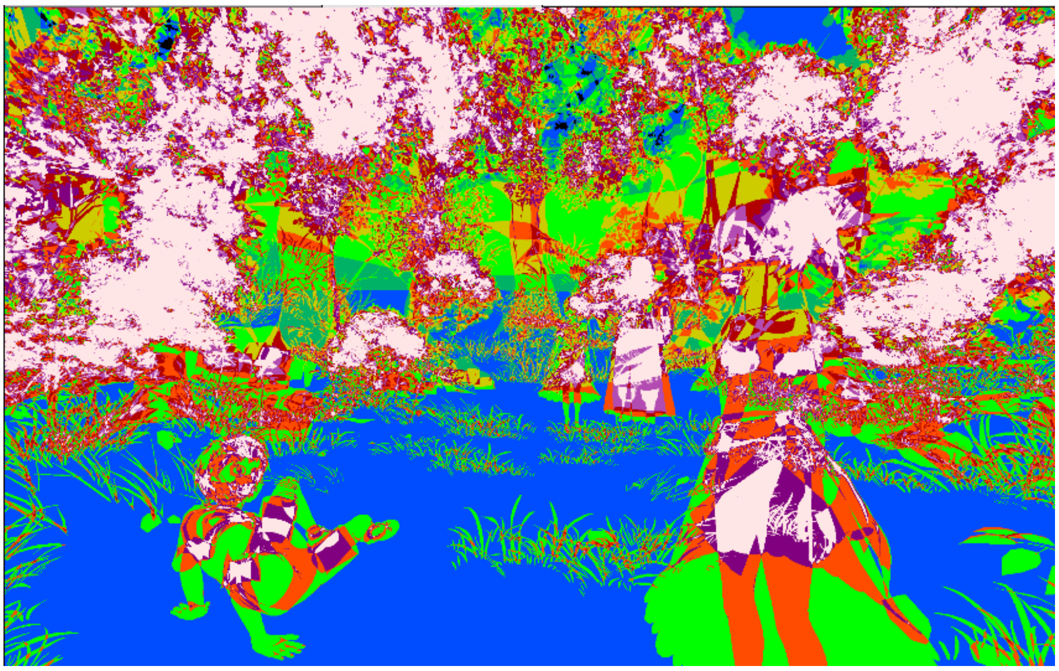


**Figure 82:** *Example of debug visualization of light overdraw.*

Using this debug visualization we confirmed that for some scenes this ratio was bigger than the ideal 1.0 and the initial assumption was confirmed. We didn't spend too much time redesigning the way lights were stored per Live Window. Instead, we just carefully branched out unnecessary lights.

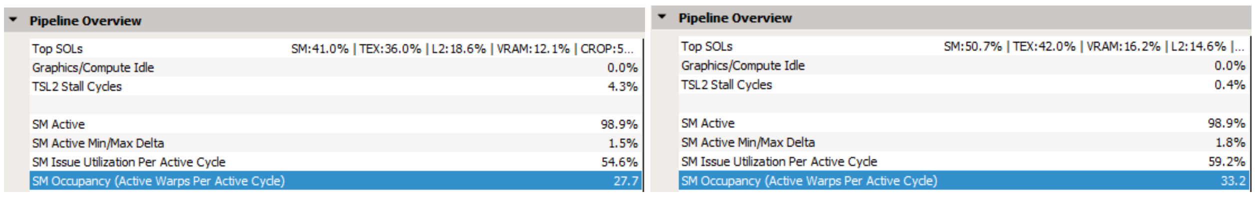
Having just a branch in a shader was not a perfect solution from a performance point of view because we still iterated over the entire chain of redundant light structures fetching the validity bitmask, but the fix itself was very fast with close to zero time investment.

### 4.3.3 Optimizing for Shader Occupancy

In our case with NPR rendering, having a full-featured Base Pass shader was overkill. We wanted to make sure we didn't pay for things we didn't use. Even if some graphics features are disabled in the shader at runtime (by dynamic or static branching), but not compiled out entirely, they still have a negative impact on performance by increasing overall shader resource usage (registers/shared memory). On the majority of modern GPUs, the increased usage of shader resources leads to decreased occupancy as the shader units stay under-utilized, especially if there are no other workloads executed asynchronously.

We looked into a profiler and found out that average occupancy for a base pass was a bit less than 28 warps on the *NVIDIA* hardware (out of a theoretical max 64 according to [Bavoil18], which is 43%). Unfortunately, there is no single answer which tool to use for that purpose. We tried different GPU profilers on PC under conditions where the surface in question covers a signification portion of the screen to make sure we measure a specific shader permutation, don't have any bottleneck somewhere in front of the graphics pipeline and generate enough fragment work to saturate the GPU. On top of that, having a shader analyzer to confirm that run-time occupancy is similar to theoretical is a plus. Also, it helps with iteration times when trying shader code transformations.

Doing several experiments with carefully compiling out several features we managed to bring occupancy to an average of 34 warps, which is 53% of maximal occupancy, and the overall increase was around 25%.



| Pipeline Overview | |
|---|---|
| Top SOLs | SM:41.0% \| TEX:36.0% \| L2:18.6% \| VRAM:12.1% \| CROP:5... |
| Graphics/Compute Idle | 0.0% |
| TSL2 Stall Cycles | 4.3% |
| | |
| SM Active | 98.9% |
| SM Active Min/Max Delta | 1.5% |
| SM Issue Utilization Per Active Cycle | 54.6% |
| SM Occupancy (Active Warps Per Active Cycle) | 27.7 |

| Pipeline Overview | |
|---|---|
| Top SOLs | SM:50.7% \| TEX:42.0% \| VRAM:16.2% \| L2:14.6% \|... |
| Graphics/Compute Idle | 0.0% |
| TSL2 Stall Cycles | 0.4% |
| | |
| SM Active | 98.9% |
| SM Active Min/Max Delta | 1.8% |
| SM Issue Utilization Per Active Cycle | 59.2% |
| SM Occupancy (Active Warps Per Active Cycle) | 33.2 |

**Figure 83:** *A screenshot from NVIDIA® NSight™ Graphics showing counters before and removing unused shader code paths.*

## 4.4 Extending Unreal Engine 4

In this section, we cover two examples of how we extended *Unreal Engine 4* to tackle performance issues we had.

### 4.4.1 Triangle Culling

Having multiple Live Windows on the screen requires rendering a scene from multiple points of view. Live Windows usually occupied only some portion of the screen. That means the majority of shaded surfaces is not visible, so the GPU processes geometry and shades fragments which don't contribute to the final image. To eliminate redundant shading, we relied on Hi-Z/Hi-Stencil and Early-Z/Stencil tests, but for the graphics pipeline frontend we didn't have many options to get rid of unnecessary processing such as fetching geometry data, executing vertex shader, assembling primitives and rasterizing them. On the other hand, removing this work means not only saving GPU cycles on processing primitives which won't generate visible fragments, it also means removing the bottleneck preventing the GPU from executing more fragment work for surfaces visible on the screen.

Usual per-object frustum culling was just not good enough here. We also noticed that *Unreal Engine 4* does culling per primitive which means complex objects with multiple static meshes were hard to reject. As a result, we had scenes where the number of rendered triangles was as much as 3 times bigger than the number of triangles sent for rasterization. So we started with verification of our hypothesis and performed polygon reduction in several scenes to see if our reasoning was correct.

### Practical Challenges

Before implementing triangle culling, we came up with several restrictions to stay mergeable with newer versions of the engine:

- Avoiding excessively invasive changes to the rendering pipeline, we wanted to keep code changes at a minimum and possibly re-use it to structure culling submission like regular *Unreal Engine 4* passes.
- Avoiding changes to asset build pipeline / stored format of geometry.
- Support *DirectX 11* as a base API

### Submission Structure

*Triangle culling* pass sits at the beginning of the frame (see figure 84). We re-used the engine's submission logic to traverse static and dynamic objects to collect only those of them for which we want to do triangle culling. This traversing is essentially a pass-through pass which skips any real rendering commands/state setup and does only data extraction (mainly vertex/index/constant buffers) from relevant meshes.

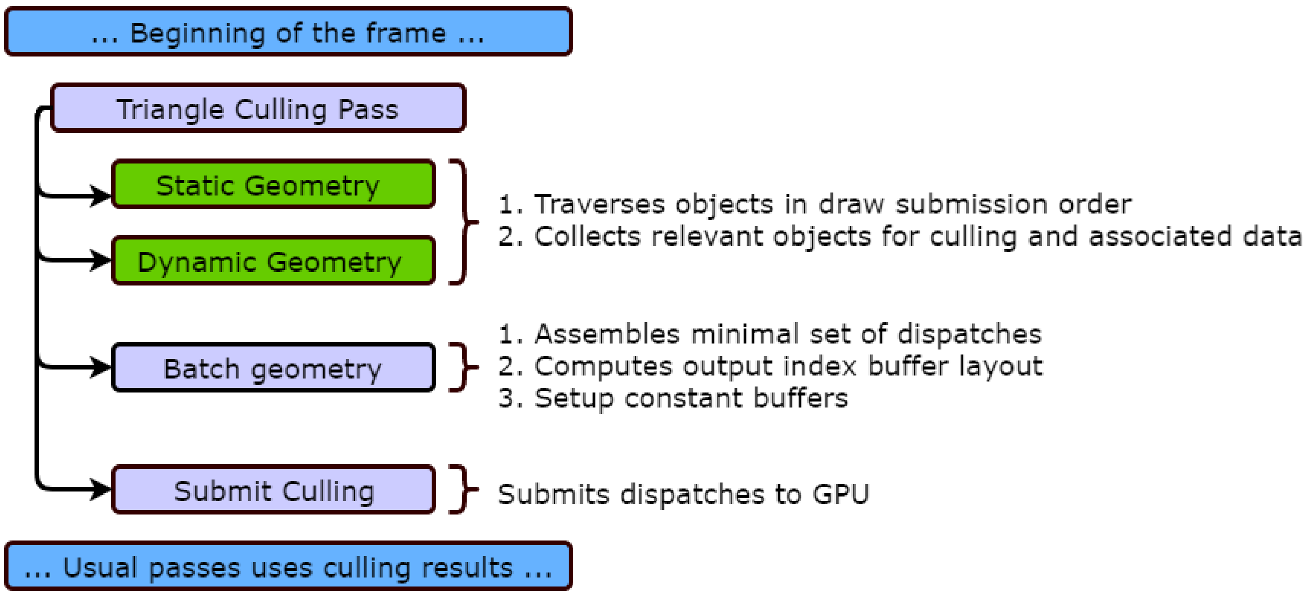

... Beginning of the frame ...
Triangle Culling Pass
Static Geometry
Dynamic Geometry
1. Traverses objects in draw submission order
2. Collects relevant objects for culling and associated data
Batch geometry
1. Assembles minimal set of dispatches
2. Computes output index buffer layout
3. Setup constant buffers
Submit Culling
Submits dispatches to GPU
... Usual passes uses culling results ...

**Figure 84:** *A high-level scheme of triangle culling pass. Green nodes illustrate stages where existing code is re-used as major part.*

After collecting all the necessary data, we batch together meshes which share index/vertex buffers, transform and an identifier of the Live Window from which this object is seen to minimize the number of dispatches. Additionally, we compute a location of culled index data in the global output buffer.

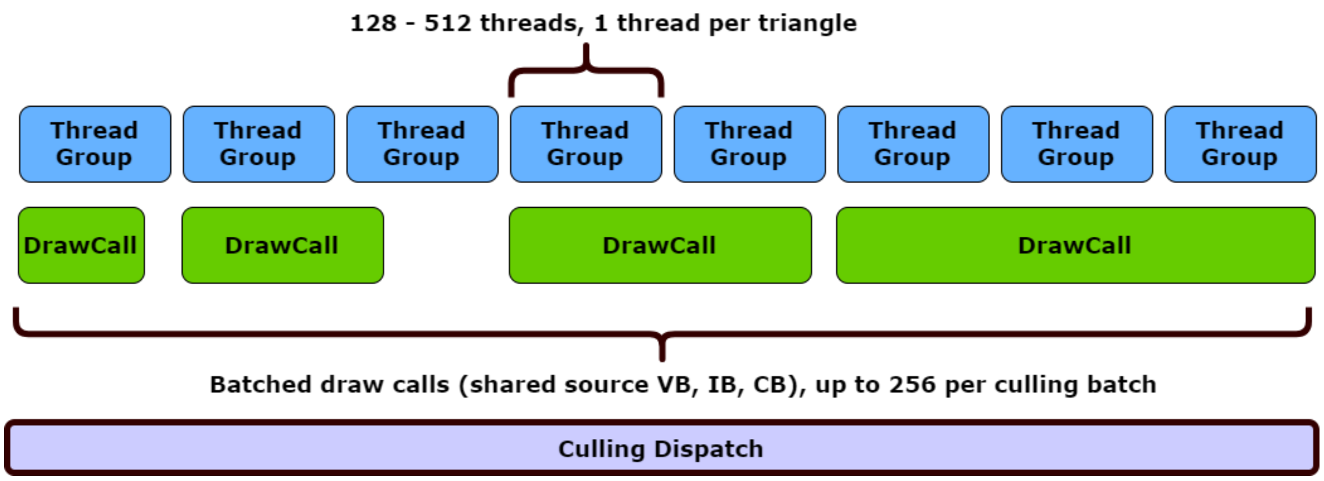


**Figure 85:** *Structure of a single Dispatch call and relationship between individual meshes used for draw calls and culling dispatches.*

After that, all dispatches are submitted. The structure of each dispatch can be seen in figure 85. Each dispatch executes a certain number of thread groups (each with size 128-512 threads) required to cull triangles of each mesh in a batch. Each mesh is essentially a draw call. We don't allow any thread group to cull triangles from any two or more different meshes. This restriction exists because we need to store the backward mapping from a thread group index in a dispatch to a mesh index in a dispatch which is used to fetch per-mesh data in the shader. After the culling pass finishes, all consequent passes use `DrawIndexedIndirect` for meshes passed triangle culling. One caveat is that it is not possible to eliminate empty draw calls and by submitting them we still create pressure on the graphics command processor on GPU.

Although this technique initially supported only static meshes, mainly because of simplicity of vertex transformation, we found a way to use it with skinned meshes as well by enabling GPU pre-skinning in *Unreal Engine 4*.

## Overview of Culling Techniques

We do several tests per triangle on GPU [Haar15] [Wihlidal16] [Engel16] to reject those which doesn't contribute to the final image:

- Backface triangle rejection
- Near plane rejection
- Screen/Live Window area rejection
- Conservative small triangle rejection

We didn't use clustered culling mainly because it requires `ExecuteIndirect/MultiDrawIndirect` from the API side, but we needed to support *DirectX 11*.

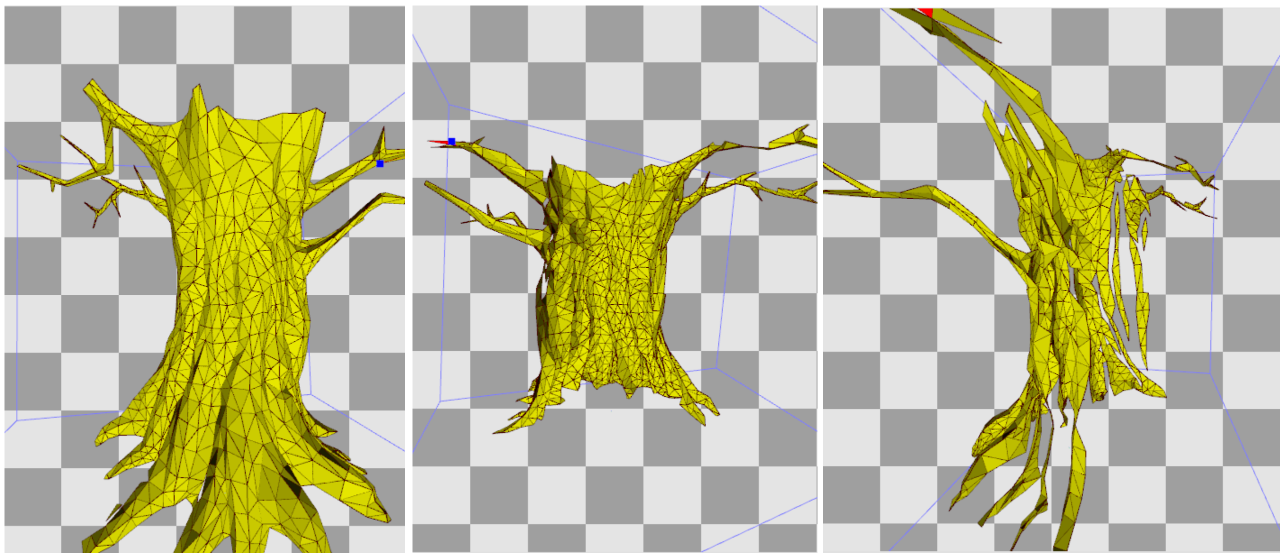


**Figure 86:** *A debug visualization from RenderDoc showing removed triangles on the front, back, and side views*

Backface culling [Olano97] is one of the most critical tests. It allows us to reject ~50% of triangles (figure 86), so we do it first in the chain of per-triangle tests. The second most important test we do is a frustum test in normalized device coordinates (NDC). It is especially helpful with rejecting triangles rendered into Live Windows. They usually occupy only a small portion of the screen, so the number of rejected triangles is usually high. For Live Windows we compute an aligned rectangle in NDC which is used later to cull triangles which don't fall into its area (figure 87).

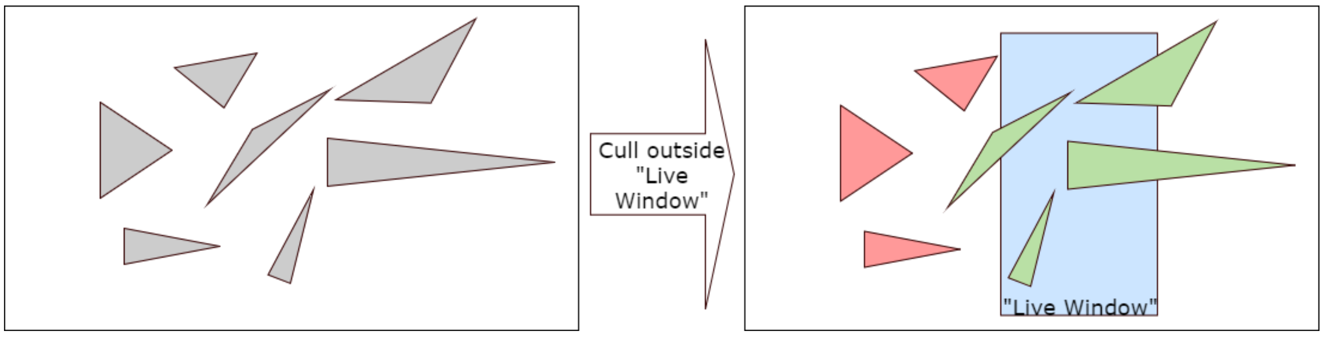


**Figure 87:** *Rejection of triangles outside of the screen-space projected rectangular area of a Live Window.*

Additionally, we reject triangles which are in front of the near camera plane and triangles which don't cover any subsample. The last mentioned test is conservative as it checks if any subsamples are inside of triangle axis-aligned rectangle.

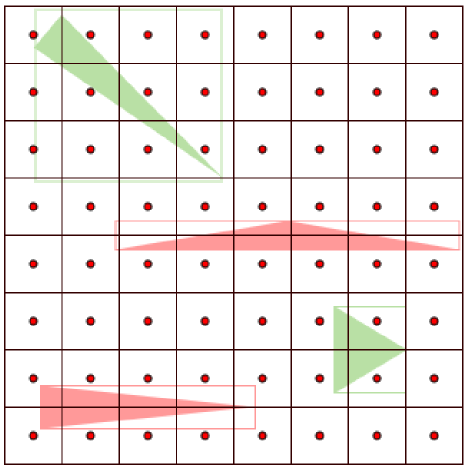


**Figure 88:** *Conservative test for small triangles: only triangles with bounding rectangle dimension less that subsample distance are rejected.*

There are a lot of cases where this test doesn't work (figure 88) but we keep this test because it is simple to do. All described tests are performed twice for stereo view in VR. The passing criterion is that no rejection happened for both eyes in one test. Overall, the worst situation for all tests is when there are a lot of stretched long triangles which can't be handled well. This technique helped us save around 0.7—1.5 ms on several scenes where we were able to achieve good batching to saturate GPU and decrease the number of compute shader dispatches. For some scenes enabling it led to a net loss because the cost of triangle culling pass was greater than savings from the subsequent draw passes.

### 4.4.2 Post-Processing Material Optimization

We relied heavily on *Unreal Engine 4* post-processing materials to prototype various full-screen effects to achieve manga look, including the line detection mentioned in the section 3.2. Although this approach was convenient for technical artists, complexity had multiplied quickly, and we ended up with 3 ms full-screen pass shader handling many different aspects in full-screen.

Profiling showed a couple of big issues: low occupancy and lots of memory stalls. Memory stalls and poor cache utilization were expected issues because of the resolution (almost 4k) and many samples from intermediate full-resolution buffers to perform the line detection: both depth buffer and additional 64-bit per fragment render target were sampled with the 3×3 kernel to get normals and additional information for line detection. Poor occupancy was a result of the overall shader complexity: the material supported many features. To overcome this we used static switches where it was possible, but we still had the worst case permutations with several features enabled at the same time.

Ideally, we wanted to avoid too many samples per-fragment for edge detection by porting this to compute shader and re-use samples within thread groups, but we had just the material graph. So we split this port into two stages.

## Stage 1. Custom Expressions

We started by moving the line detection part of the material's node-based graph to HLSL code. We exploited “Custom Expression” nodes in *Unreal Engine 4*, which allow to call HLSL code directly from a material graph. An example of the transformation can be seen in figure 89.

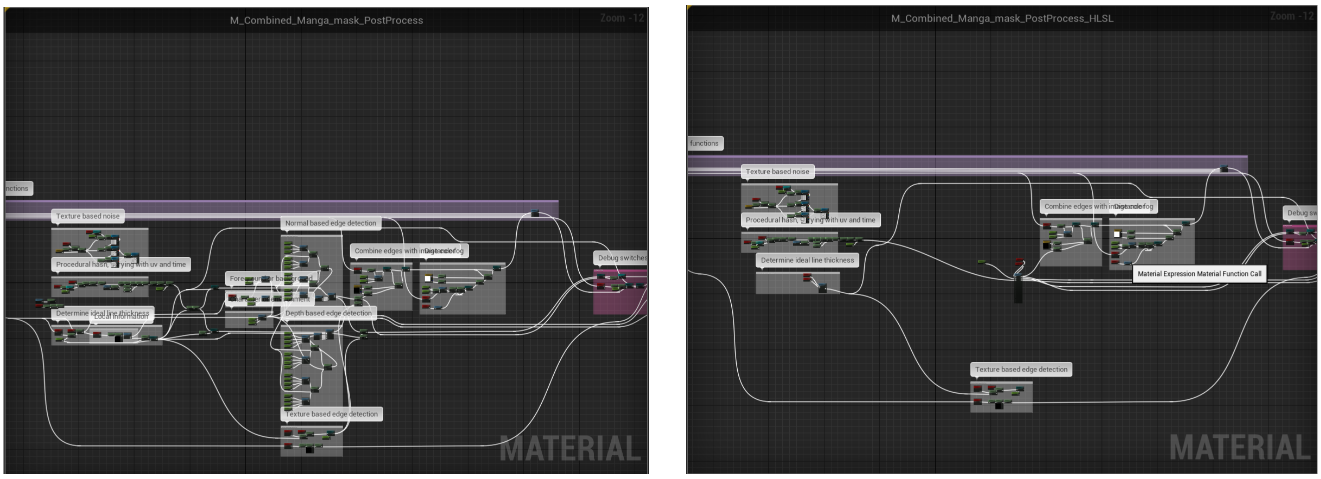

M_Combined_Manga_mask_PostProcess
Texture based noise
Procedural hash, varying with uv and time
Normal based edge detection
Combine edges with image color fog
Determine ideal line thickness
Local information
Depth based edge detection
Texture based edge detection
Debug switches
MATERIAL
M_Combined_Manga_mask_PostProcess_HLSL
functions
Texture based noise
Procedural hash, varying with uv and time
Determine ideal line thickness
Combine edges with image color fog
Material Expression Material Function Call
Texture based edge detection
MATERIAL

**Figure 89:** *Simplified post-process material graph.*

Moving line detection to HLSL allowed to experiment on micro-optimizations and have better control over code generation because *Unreal Engine 4* pulls in custom HLSL code into the final source without any modification. The only thing to be aware of is constant (or uniform expression) folding which happens during the transformation of a node-based graph into HLSL code. This mechanism helps to reduce the number of instructions for cases where user-defined parameters are used in arithmetic operations, and results of those operations stay constant across all shader invocations. For example, if two user parameters are added, it's possible to precompute them and store the result as one constant. However, this doesn't work with code referenced by “Custom Expression” nodes. So, it's better to precompute all expressions referencing user parameters in a material graph and pass results of those transformations to “Custom Expression” node as parameters.

After transforming the most expensive part of the node-based material graph to HLSL, pulling up constants for the automatic folding and performing micro-optimization, we achieved around 2.4 ms for the entire full-screen pass by reducing ALU count and increasing occupancy. The pass became entirely memory bound. By doing these optimizations, we got not only the performance improvement but also got a verified and ported to HLSL version of our line detection technique.

Moreover, it was possible to extract it as a new pass to be implemented as a compute shader, in order to utilize the LDS for sharing sampled data across a single thread group.

## Stage 2. Compute-Based Pass

When the line detection was part of the pixel shader, it implied sampling one fragment nine times because of the 3×3 kernel (figure 90 (a)). In our case, we sampled both depth buffer and the render target storing normals and additional artist-controlled values for the line detection. That means 18 samples (64 bits of data each) per output fragment in the full resolution. With the compute shader, on the other hand, it was possible to sample data once for the entire thread group and reuse it between threads through the shared local memory (LDS). In Figure 90 (b), we show a border of additional fragments outside of the 8×8 thread group to sample to make sure we have available data for the 3×3 kernel for the every output fragment. Additionally, in figure 90 (c) we show the boundaries of four 8×8 thread groups and intersections of their sampling areas. It illustrates that each fragment is sampled at most 4 times (corners) or even just 2 times (a boundary of two groups). The majority of fragments are sampled once.

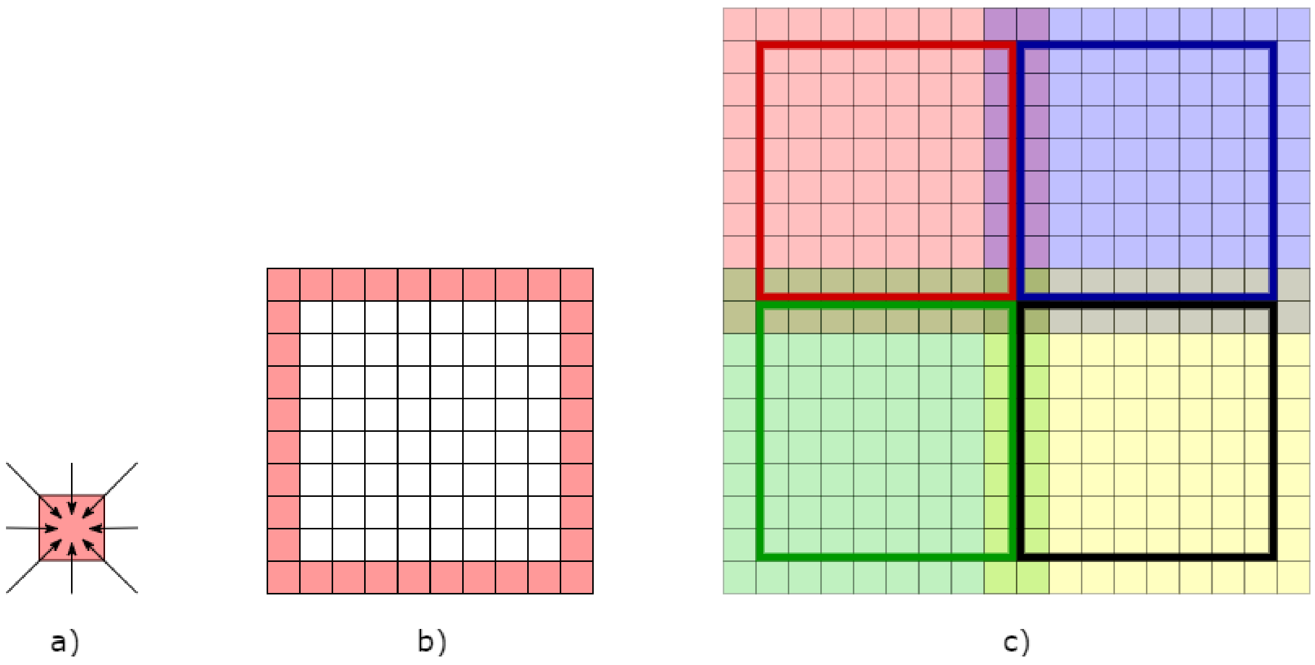


**Figure 90:** *a) Each fragment of the source render target is sampled 9 times when the pixel shader process 3×3 kernel for each destination fragment. b) Fragments sampled outside of the 8×8 thread group in the compute shader. c) Sampling areas of 4 adjacent thread groups and fragments sampled multiple times.*

This reduction in the number of samples was an important point, but not the only one. We took extra care to ensure that there are no LDS bank conflicts, and usage of shader resources (VGPR, LDS) doesn't limit occupancy of the shader. The structure of the shader became somewhat complicated because of the LDS management and packing of the sampled data, but we were able to achieve 1.2—1.4 ms for the full-screen pass handling the line detection only. We ended-up with 16×16 sized thread groups and noticed better cache utilization if the entire pass works in stages, for example, processing half of the screen at once.

The last unsolved issue before enabling compute shader-based line detection was the need to preserve all user parameters for this technique which were already set up through material instances. We used a small trick here: we just disabled the line detection functionality in the node-based material by adding a dummy expression which uses all user parameters (uniforms) required for the line detection (to preserve dependencies and avoid compiling them out) and disabled it with a static branch with a uniform parameter condition that is never enabled (see listing 5).

```
cbuffer AutoGeneratedCB
{
    bool AlwaysDisabledParameter;
    float UserParam0;
    float UserParam1;
    float UserParam2;
    float UserParam3;
}

float3 ShaderOutput = ...;

[branch] if (AlwaysDisabledParameter)
{
    ShaderOutput *= saturate(UserParam0 * UserParam1 * UserParam2 * UserParam3);
}
```

**Listing 5:** *Code never executed and forcing an input dependency on purpose.*

Because the layout of the auto-generated constant buffer was the same for all the material instances, we just dumped offsets of user parameters required for the line detection and declared buffer with the same layout in the compute shader. After that, we used the same material buffer for graph-based post-processing pass which handled everything except the line detection and for the compute shader-based optimized line detection (figure 91).

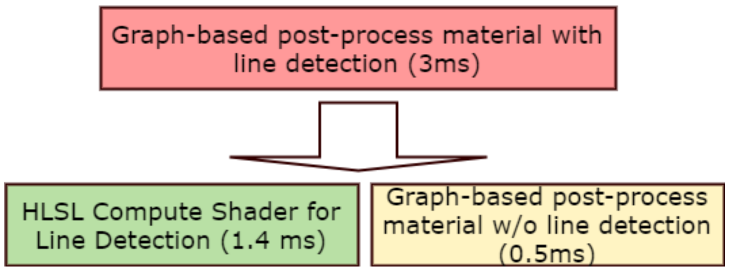

Graph-based post-process material with line detection (3ms)
HLSL Compute Shader for Line Detection (1.4 ms)
Graph-based post-process material w/o line detection (0.5ms)

**Figure 91:** *Split of original graph-based post-process material.*

The remaining part of the graph-based post-process material was much cheaper (~0.5 ms), so together with the compute-based line detection it was less than 2.0 ms which was significantly less expensive than the original 3.0 ms we started from.

## 5 Conclusion and take away

This document is a recollection of some of what we have learned in the process of replicating the visual style of a manga. We hope that readers will find it useful and that it will help them explore further or experiment in other directions.

On our side, while we are satisfied with the result we achieved, we think there is certainly room for improvement. We are considering how to further increase the quality of our rendering and the efficiency of our tools. This includes line detection, hatching, artist control and workflow in general.

The identification of the visual elements of the manga and their break-down into a list of features from the start was key to implementing the shading and offering artist-friendly controls. A successful implementation depends on the artists and engineers communicating clearly the results they want to achieve.

One constant observation is that line detection can yield sometimes unwanted, unexpected or counter intuitive results. Maintaining a debug visualization of the intermediate results is important to help the artists understand the reason when the lines they get are different from what they want. That understanding is necessary so they can decide what to do, whether it is tweaking a parameter to help line detection, or use a different solution like a geometry or texture based approach.

Finally, developing for the virtual reality medium is difficult and unforgiving. It's a platform where certain camera moves, or even a simple rendering bug, can lead to actual, physical sickness. Furthermore, it requires working within a very tight frame budget, which can translate to drastic limitations.

We have noticed that is can be difficult for the team to recognize those constraints and understand how to work with them. Unfortunately ad hoc and late performance improvements often come at the cost of sacrificing visual quality. Optimizations done at the end of production cannot have the same impact as planning and budgeting from the beginning. So it's important to establish a proper understanding for everyone involved and arrange all aspects of the production with those limitations in mind.

## 6 Acknowledgments

We would like to thank Patrick Davis and Brian Wanamaker for reviewing this document, Mike Gränz for his additional help, and Rémi Driancourt and Takako Yamashita for their support in organizing this course.

We also extend our thanks to MAYBE for providing the manga source material this work is based upon.

## Bibliography

[**Bavoil18**] Louis Bavoil. 2018. The Peak-Performance Analysis Method for Optimizing Any GPU Workload. URL: https://devblogs.nvidia.com/the-peak-performance-analysis-method-for-optimizing-any-gpu-workload/

[**Engel16**] Wolfgang Engel. 2016. The Filtered and Culled Visibility Buffer. In GDCE'16. URL: http://www.conffx.com/Visibility_Buffer_GDCE.pdf

[**Haar15**] Ulrich Haar and Sebastian Aaltonen. 2015. GPU-Driven Rendering Pipelines. In *SIGGRAPH Course: Advances in Real-Time Rendering in Games*, ACM SIGGRAPH. ISBN: 978-1-4503-3634-5. URL: https://doi.org/10.1145/2776880.2787702

[**Motomura15**] Junya Christopher Motomura. 2015. GuiltyGearXrd's Art Style: The X Factor Between 2D and 3D. In GDC'15. URL: https://www.gdcvault.com/play/1022031/GuiltyGearXrd-s-Art-Style-The

[**Olano97**] Marc Olano and Trey Greer. 1997. Triangle Scan Conversion using 2D Homogeneous Coordinates. In *Proceedings of the ACM SIGGRAPH/Eurographics workshop on Graphics Hardware*, pp. 89—95. ISBN: 0-89791-961-0. URL: https://www.csee.umbc.edu/~olano/papers/2dh-tri/2dh-tri.pdf

[**Olsson12**] Ola Olsson, Markus Billeter, and Ulf Assarsson. 2012. Clustered deferred and forward shading. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, Eurographics Association, pp. 87—96. ISBN: 978-3-905674-41-5. URL: http://dx.doi.org/10.2312/EGGH/HPG12/087-096

[**Persson13**] Emil Persson. 2013. Practical clustered shading. In *SIGGRAPH Course: Advances in Real-Time Rendering in Games*, ACM SIGGRAPH. ISBN: 978-1-4503-2339-0. URL: https://doi.org/10.1145/2504435.2504450

[**Prewitt70**] Judith M.S. Prewitt. 1970. Object enhancement and extraction. Picture processing and Psychopictorics, vol. 10, no. 1, pp. 15—19.

[**Quilez08**] Iñigo Quilez. 2008. Distance functions. URL: https://iquilezles.org/www/articles/distfunctions/distfunctions.htm

[**Roberts63**] Lawrence G. Roberts. 1963. Machine perception of three-dimensional solids. Doctoral dissertation, Massachusetts Institute of Technology.

[**Scharr00**] Hanno Scharr. 2000. Optimale Operatoren in der Digitalen Bildverarbeitung. Doctoral dissertation. URL: http://archiv.ub.uni-heidelberg.de/volltextserver/962/

[**Sobel68**] Irwin Sobel and Gary Feldman. 1968. A 3×3 isotropic gradient operator for image processing. *presented at a talk at the Stanford Artificial Project*.

[**Vlachos16**] Alex Vlachos. 2016. Advanced VR Rendering Performance. In GDC'16. URL: https://www.gdcvault.com/play/1023522/Advanced-VR-Rendering

[**Wihlidal16**] Graham Wihlidal. 2016. Optimizing the Graphics Pipeline With Compute. GDC'16. URL: https://www.gdcvault.com/play/1023109/Optimizing-the-Graphics-Pipeline-With

[**Wikipedia**] Wikipedia. Four color theorem. URL: https://en.wikipedia.org/wiki/Four_color_theorem