Steels Rufus

# Cross Hatch 3D Art Style

Graduation work 2022-2023

Digital Arts and Entertainment

Howest.be

## CONTENTS

## ABSTRACT

In this paper we will take a deeper look into hatching shaders. most of the hatching shaders today use textures, but the drawback of these shaders is the fact that they cannot be used for the contour hatching style unless the shapes are unwrapped in the model. And even then the textures can only be lightened and darkened, but the thickness of the lines will not change.

The main problem being solved in this paper is the problem of finding an alternative solution for drawing the lines in a way that they follow the shape they are drawn on without the use of external textures.

## INTRODUCTION

The topics discussed in this paper concern the idea of taking the hatching/etching art style developed through history, and converting it to something suitable for the 3D medium.

One of the advantages of 3D art is the fact that the light can easily be manipulated. If this can be accomplished nicely with a shader, it would mean that an artist can change the look of an 'image' very easily by simply changing some variables in a scene.

So now the question arises: 'Is it possible to create a hatching shader in Unity that can easily be manipulated?'. There are however some conditions the shader should meet before it can be called a success.

Does the shader look good while the object is moving, or while the light source is moving? Does the hatching look natural on the objects? Are there no weird seams on the objects? Does the scene look like it is a hatched drawing? Is the shader easy to implement without too much additional tasks to perform beforehand?

To know what is important for creating this shader, research needs to be done first.

# RELATED WORK

This chapter talks about the research that has been done during the writing of this paper.

Research has been split up in two main categories.

The first one is doing research on the art style of hatching. To know how to recreate the art style, there must first be an analysis of the art style. This way the rules of how these drawings are made can be laid bare and can be communicated into a 3D world.

The second category is more on the technical side as to what is possible with shaders in Unity. What data is available in the mesh and how this data can be used to create a shader that looks like a hatched drawing. Of course there will be more information about the process of learning how to write shaders as well but this will be discussed in further detail in the chapter “Experiments & Results”.

# 1. HATCHING

## 1.1. HISTORY

Hatching is a drawing method that came to exist over the course of history. It was used mostly for making drawing that could be printed. Since the printing press could only print in binary values, the artists needed to find a way to incorporate shadows, light, shapes into their drawings. They did this by making etchings that were shaded using thin lines.

These thin lines were then used to start making more and more detailed images. New techniques were invented to convey different things, contours were being followed, sometimes contours were not being followed to indicate some objects that were in the background or just to indicate objects that were not the subject of the drawing. People found new and innovative ways to use these lines to create true pieces of art.

This then evolved into a modern art style that people still use and is called hatching.

The way these prints were made in the old ages was by etching a piece of metal or carving a piece of wood. These plates were called etches. An etch was then used in a press to make multiple prints.

These etches were made by taking a metal plate (usually copper) and placing an acid resistant layer of wax over this plate. Then the artist carved the wax in the spots where the metal should be corroded. Then finally the wax coated plate is placed in an acidic bath. The acid corroded the metal in the places the wax had been carved away and now the plate could be coated in ink. Placing these etches in a press and pressing down on a piece of paper made it possible to create multiple copies of the same etch.

Sometimes there were however small errors, like places where the acid got through the wax, or places where the acid started to slip under the wax because it was in the acid for too long. These artefacts are desirable these days because they are seen as properties of the medium.

In the modern time these prints are still being collected and they are very valuable. Depending on how old they are they are considered Old Master Prints. The prints don't even need to be of great artistic quality to be worth a lot. Sometimes the prints were mass produced, but even then, owning one of these is very rare.

Because the style is done with a printing press, and only has two values, black and white, the modern art style has kept the limitation of using exactly two values. Therefore it is often done by using special pens that don’t change in color by changing the pressure, but change line thickness instead.

## 1.2. CROSS HATCHING BASICS

Hatching is a technique used to shade a drawing. Parallel lines are drawn to show where darker areas are on a drawing. This can either be because of more shadow or because a darker material must be displayed. Hatching can be used to give depth to your drawings in an elegant way.

To make sure that the shapes are still legible the lines should follow the contours of the object you are shading. A good example of this is a shaded sphere, none of the lines are straight. They are all curved to follow the curves of the sphere.

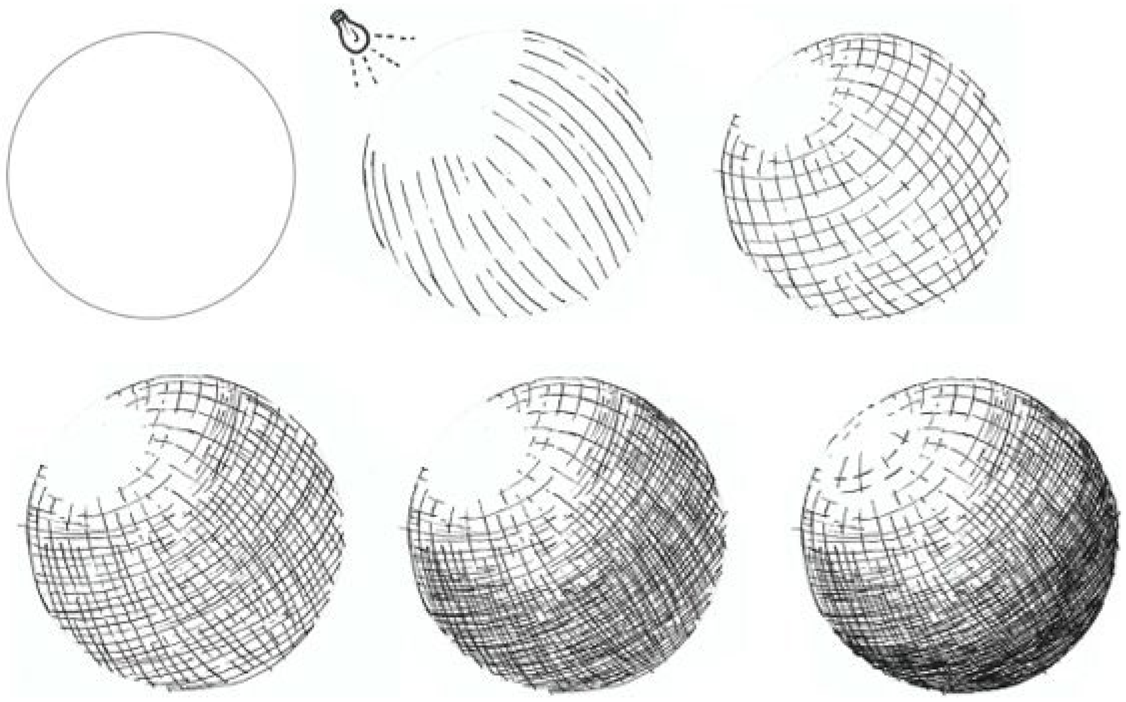


Figure 1: Hatching a sphere

It is easiest to start with the darker areas and work your way up to the lighter areas. The darkest can be filled in completely with black, and then the lighter the color the less dense the hatching. For the darker but not quite black areas it is best to cross the lines to create a cross hatch. This creates a dark color.

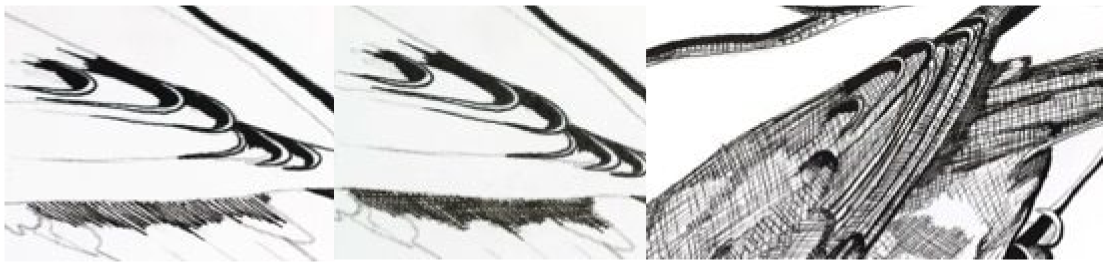


Figure 2: Cross hatching order

## 1.3. TYPES OF HATCHING

There are several types of hatching to pick from. The most common ones are parallel hatching, cross hatching and contour hatching. Oftentimes combinations of these types are also used. Of course, there are more types. These are to achieve a different, more unique look for the drawings. Some of these are patch hatching, scribble hatching or tick hatching.

These are some images of the different types of hatching. They are listed as follows. Parallel hatching, cross hatching, contour hatching, patch hatching, scribble hatching and tick hatching.

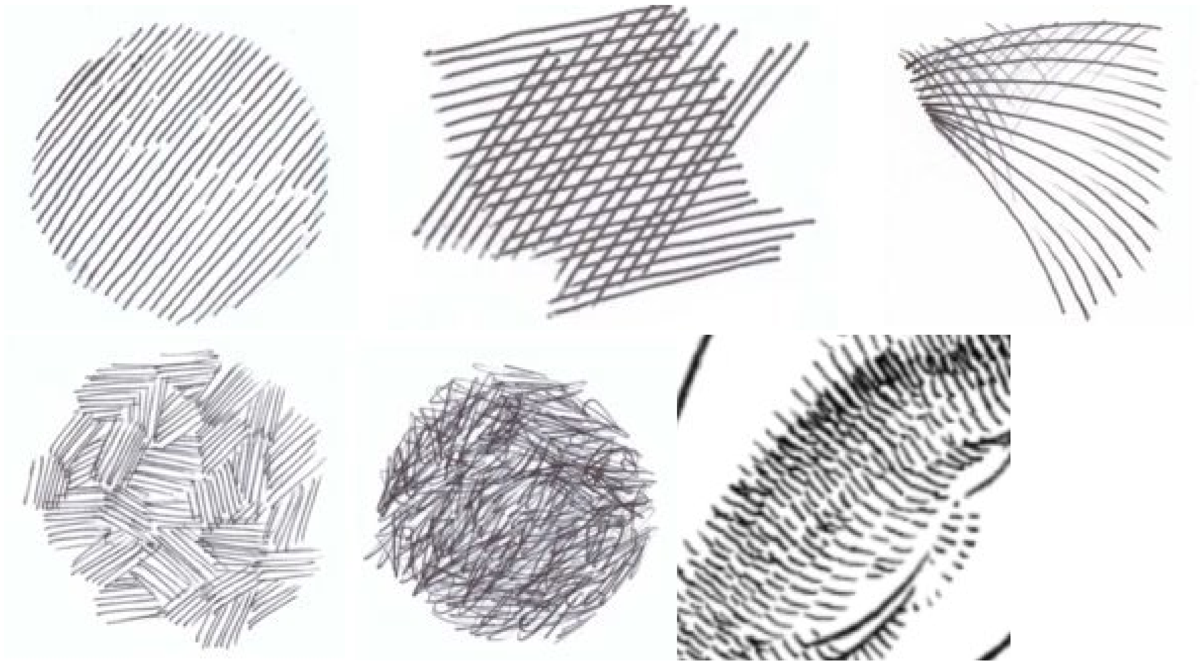


Figure 3: Types of hatching

## 2. SHADERS

### 2.1. EXISTING HATCHING SHADERS

There already exist several shaders that mimic the art style of hatching in 3D. There are multiple approaches to making a cross hatching shader.

Some shaders mimic the lines by using mathematical equations that appear and disappear depending on the light level. Others use the approach of taking different textures and drawing these with a certain opacity depending on a couple of variables. Some shaders layer textures overtop each other to create darker areas. Some combine these ideas where they procedurally generate several textures to then pick from in runtime.

One of the problems with the shaders in which textures are needed is that it is very hard to have a contour hatch. The textures can be used in screen space, to just overlay them overtop each other depending on the how dark the area is. But this automatically means that contour hatching is impossible because no attention is turned towards how the shape is actually laid out.

Another way of using these hatches is by UV-unwrapping the model over the texture and changing the strength of how much the texture is displayed to show the lighting on the model. The problem with this is that it is near impossible to make sure that the lines match up on areas where seams occur. Another drawback in this case is that the only thing that changes is the darkness of the texture, so this goes a little bit against the idea of hatching since there are gray values being displayed. The lines also do not change in thickness depending on the light level. This does not mean that these shaders are bad, they look very good and are a good choice for hatching shaders in styles like patch hatching.

Therefore this paper will discuss a method that does not use external textures. The goal is to create a shader that uses a contour hatching style. The big problem with the types of shaders that currently exist is the fact that it is hard to find out in which direction the lines should run on a specific object. So the big focus will be finding a way to draw the hatching lines in a manner that they follow the shape they are drawn on.

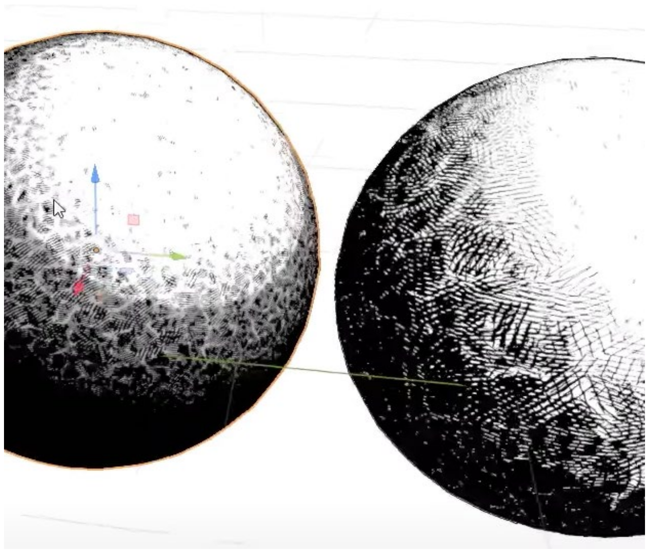


Figure 5: Shader by Kristof Dedene

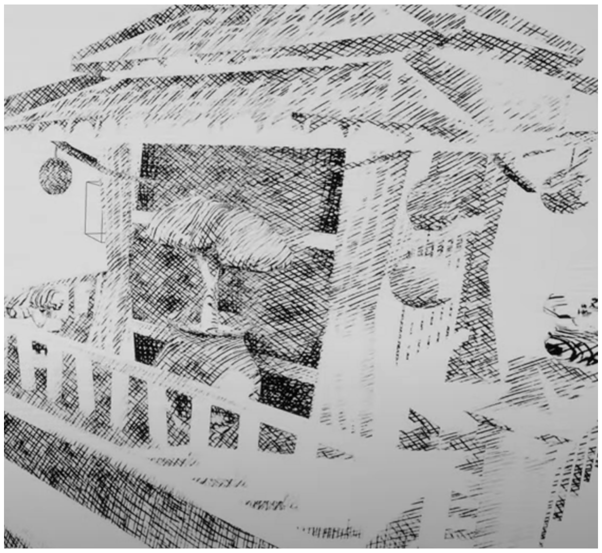


Figure 4: Shader by Kamil Hepner

As you can see in the see in these examples the hatching does not follow the shape of the objects at all. A lot of lines are drawn and the patch hatching technique is well represented. But some of the shapes are relatively hard to read, and the second shader is not very well equipped for moving around the scene. This can be seen on the video I have taken this example from.

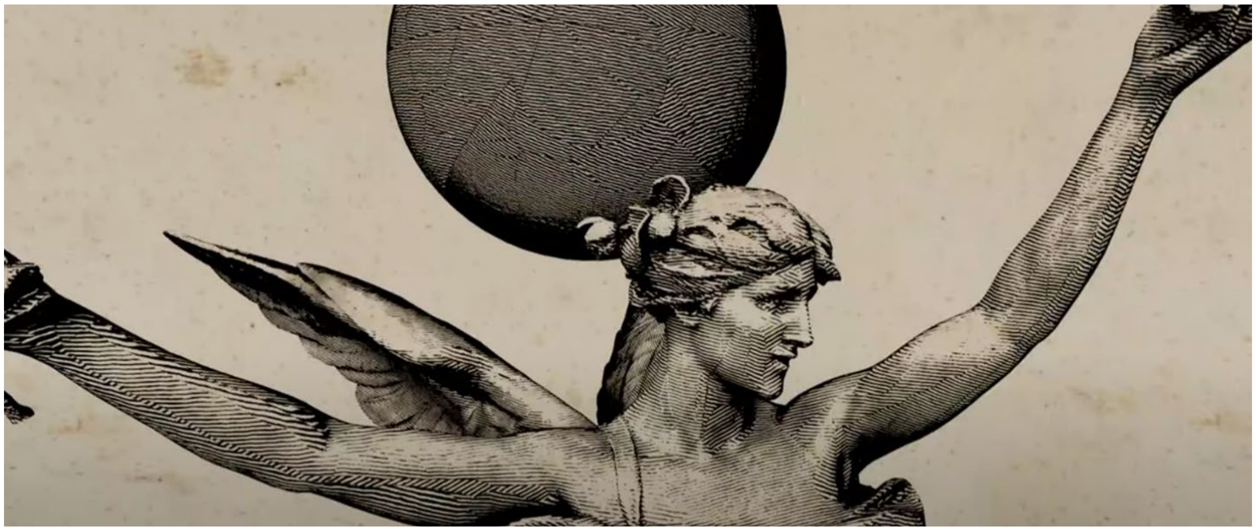


Figure 6: Shader by Ocean Quigley

In this last example you can see that the author tried to change the direction of the lines in a way that they would follow the object at least a little bit. The effect does not however yield the results you would want from a contour hatching shader.

## 2.2. VERTEX DATA IN UNITY

To know how to make shaders I will need to know which data is available to use. Meshes contain a lot more data than just position. First, I will look at what data is available in Unity's ShaderLab.

In total there are 8 variables in the vertices:

POSITION
NORMAL
TEXCOORD0
TEXCOORD1
TEXCOORD2
TEXCOORD3
TANGENT
COLOR

There are several sizes to these variables. POSITION, TANGENT, and COLOR are typically float4 variables (a 4-dimensional vector). The NORMAL is usually a float3 and the TEXCOORD variables are either a float2, float3 or float4.

Some of these variables will come in very handy while writing a shader. The normal shader is one that is used very often in shader situations to calculate how light an area will be depending on the direction of the light. The position is also used to take this position and know where to draw something on the screen. TEXCOORD0 - 3 are used for UV mapping.

Of course, not every one of these datapoints is used in every mesh/shader. So, if we would need some more information, we could store it in a variable that is not being used in the current situation. But we should also consider that we need enough space to store the correct amount of data.

In our case we will be storing a direction in which the hatch will go. A direction is a vector with 3 variables, x, y and z. This means that we need a variable of at least 3 floats in the vertex. Since the TEXCOORD variables can also be a float2, we should probably take the COLOR. This one is often used to store extra data.

Another useful thing in the ShaderLab is the fact that we can add additional data into the vertex to make the shader a little bit more performant because calculations will be done in a per vertex fashion instead of a per fragment (comparable to per pixel) fashion.

As you will be able to read later on, At the end of the project I had to move the shader towards a ShaderGraph shader. This was because the shader I had written up to that point had been an unlit shader which conflicted with the outlines I was trying to add.

One of the downsides with this shader was the fact that I had a little bit less control over which data was calculated on a per vertex basis and which values on a per fragment basis.

## RESEARCH QUESTIONS

"How can I create a hatching shader in Unity?". This is the main question I will be answering, but during this project I have found a couple of obstacles I will need to conquer. I have split these up in 3 larger categories:

**Hatching direction:**

1. How can the hatching direction of a separate area of an object be chosen?
2. How can I make sure that the transitions between the hatching directions still look good?
3. How can this direction of the hatch be baked into the mesh?
    3.1. Can a rig be used to bake the data into the mesh?
4. Are the techniques mentioned worth using?

**Light sources and moving hatches:**

1. Can the light level be taken into consideration while still displaying the hatching nicely?
2. Does the hatching look good if the object is moving?
3. Does the hatching look good if the light source is moving?
4. Does the hatching look good if the object is changing shape (e.g. moving bones for animating)?

**Influence of outlines:**

1. Are outlines necessary for a hatching art style?
2. Can outlines be implemented easily?

## EXPERIMENTS & RESULTS

### 1. HATCHING

Experimentation is important to gain a better understanding of the rules of hatching. What works and what does not. How shadows and light are visualized.

For this reason, some experiment-drawings were created. Drawings were made using the cross-, parallel-, and contour hatching techniques. The first drawings were made using pencil. This is not exactly ideal for hatching but it is a good starting method for a beginner-hatcher because it is easier to shade using pencil. Some techniques were followed from a 'cross hatching for beginners' video.

The first drawings were sphere drawings. This is because this is an easy shape to begin with and it still has enough variation to clearly see the difference between hatching methods. The first technique used was the contour hatching technique, in which the lines followed the shape of the object. The second drawing was made using the cross hatching technique. Which did not look quite natural and frankly, a little weird, The second iteration of the cross hatching technique used a finer hatch. This version came out better than the first one. The shape could still be read even though the contours were not emphasized by form-following lines. The cross hatched shape had a more stylized feel, while the contour hatch looked more realistic. Of course, the cross hatch is not perfect but that is not necessary for an experiment either. The important part is to convey different feelings of the drawings using different hatching techniques. Using cross hatching without trying to mimic the shape with the lines can still be very beautiful and definitely has its place.

Since the decision had been made to use contour hatching, another shape could help in understanding how exactly it worked. Therefore another experiment was done using a cylinder as reference. This clarified the fact that differences in depth, like a concave part or a hole showed more contrast between the separate faces. This contrast can be amplified by adding outlines.

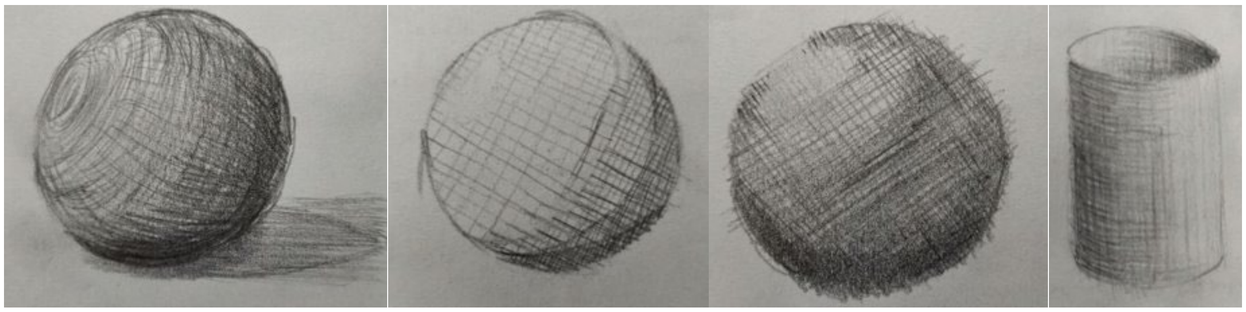


Figure 7: Experiments, Contour, Fine Cross, Cylinder

There was another hatching technique that had not been considered yet. The tick hatching technique could also be combined with the contour hatching, in which the ticks would follow the shape of the object. The following drawing was done with a ballpoint pen because it doesn't quite have the value range of a pencil. This was more in line with what hatching is at its core.

The tick-contour hatching had another unique look to it. It was a type of blend between the stylized and the realistic looks that came from the cross hatching and contour hatching respectively.

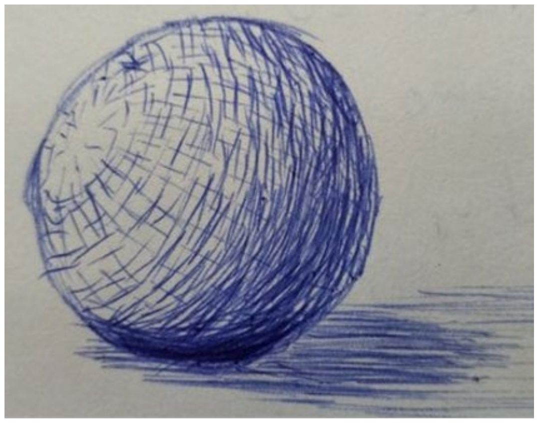


Figure 8: Tick hatching

Now I wanted to take a closer look at how cast shadows and the light that remains therein could be visualized. Again using a ballpoint pen.

I studied some hatched drawings that I had found and looked at how the lighter areas were displayed. This was often done by 'pausing' the hatch in areas of cast light. One of the more interesting parts to me was when the artist wanted to convey a dark room with a streak of light the light was exaggerated by removing the sharp edges.

As you can see, I tried to do the same in the second drawing. The line of the corner between the floor and the left wall can be seen in areas of darkness, but the line disappears completely in areas of high light. This gives a blinding effect telling the viewer that the contrast between the dark room and the light cast from outside is very high.

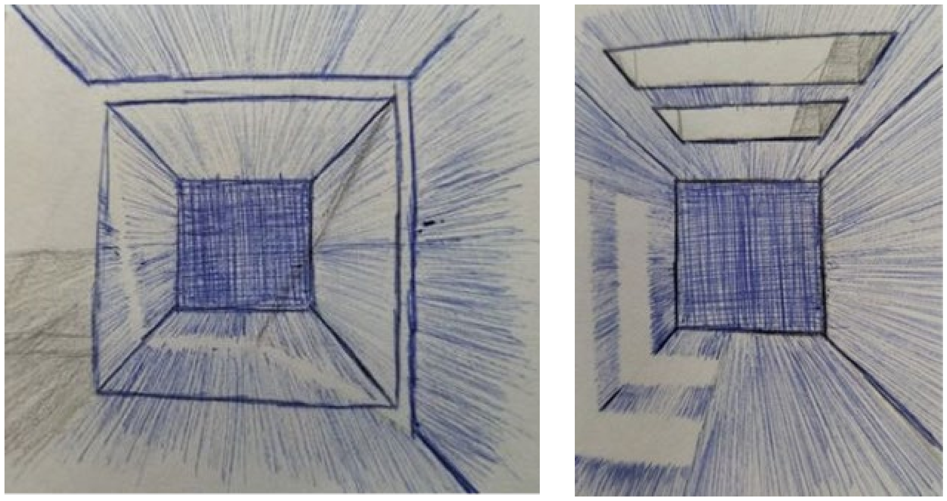


Figure 9: Light studies

# 2. SHADERS

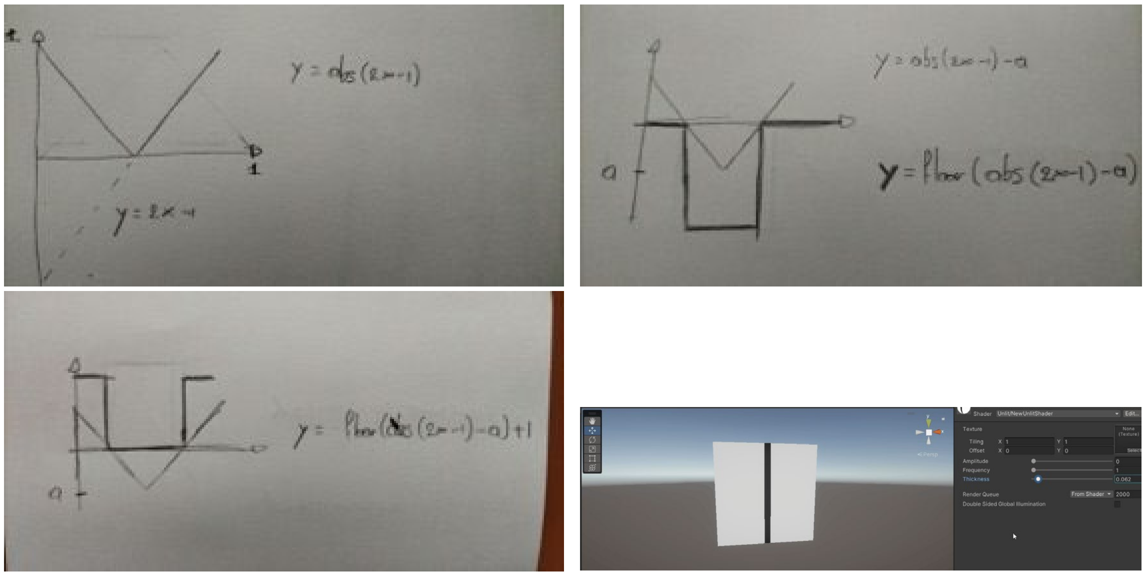
## 2.1. FIRST SHADER

After the research I tried to take a more hands on approach on creating shaders. I decided to start with HLSL (High Level Shader Language) in Unity. I could do this via the ShaderLab in Unity. I chose this over the shader graph because in my experience I feel like I personally have more control over what I can do in code then in a node-based system. I Had the same problem with prototyping in Unreal a couple of years ago, so I wanted to avoid having a similar problem this time around.

I followed a video about shader basics to create a simple shader. I used it more to gain experience in what functions can be called and how to write shaders in general than to follow along and create a similar shader.

What I tried to achieve by the end of this session was a line which could be thicker or thinner depending on a certain value.

I did some calculations to mathematically draw a line. I used a sawtooth shape that you can move up and down using a value, after which I floored the function and inverted it to make it so the value it outputs are always 1 or 0. Then I used the x of the UV coordinate (which is normally also between 0 and 1) to display the line on the object. Later on, the value can be calculated internally so the thickness of the line varies depending on the calculation of the light present on the surface.



Shader Unlit/NewUnlitShader
Texture None (Texture)
Tiling X 1 Y 1
Offset X 0 Y 0
Amplitude 0
Frequency 1
Thickness 0.062
Render Queue From Shader 2000
Double Sided Global Illumination

Figure 10: First shader and calculations

## 2.2. IMPROVING THE FORMULA

Now that I have a line I can change the thickness of, I need to take the next step. The next step being that I need to draw multiple lines on an object to create a hatching pattern.

I did this by changing the formula to use a modulus and multiplying the function so the function repeats at a frequency you can change by a single variable f. By multiplying by 2 and taking the modulus of 2 the function repeatedly drops when the value reaches 2. This way we can change the frequency of the function by changing the slope of the function. After this we can do the same thing as we did before to create the lines .

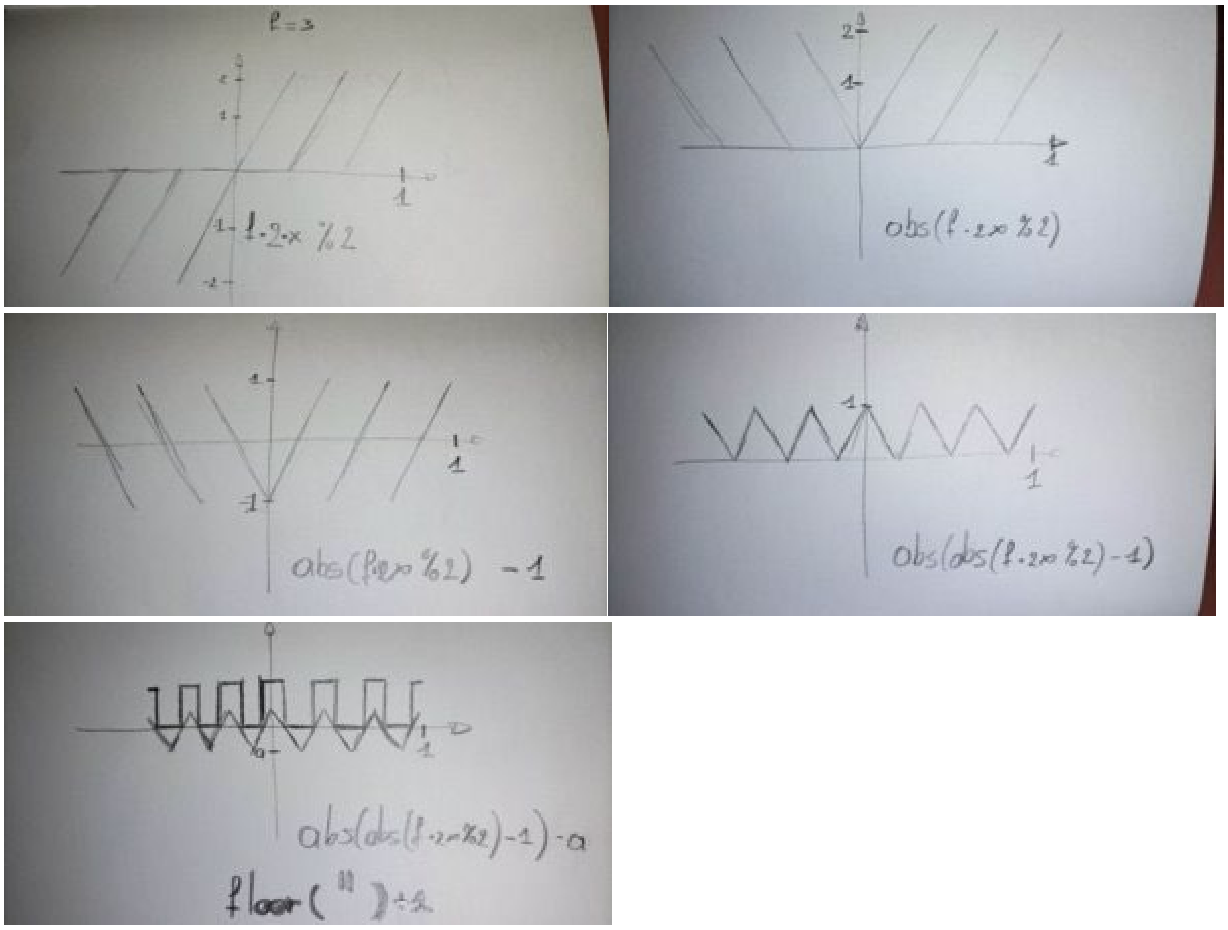


Figure 11: Calculations for recurring lines

## 2.3. IMPLEMENTING THE EQUATION

Now we have the formula to draw the lines, but we still need a variable we can link to the X axis of the function. To take an easy example we can use the local y position of the vertices of the mesh. I will first make a shader that represents the positions of the vertices. The X is the red value, Y the green value and z the blue value.

Now that we have the local x, y and z value of every point on the surface of our mesh, we can use these values as a variable in our previously calculated function. In the example below I have simply plugged the local y-position of each point on the mesh in the equation. And I have used the resulting value as the color black on the mesh.

The resulting equation is:

Value = floor (abs(abs((f * 2 * localPosition.y) % 2) - 1))) – a

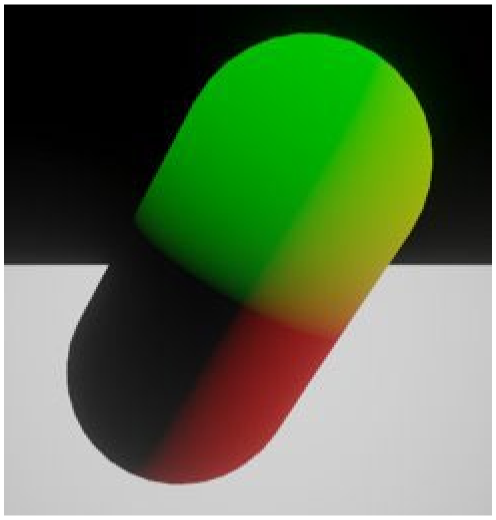


Figure 12: Local position

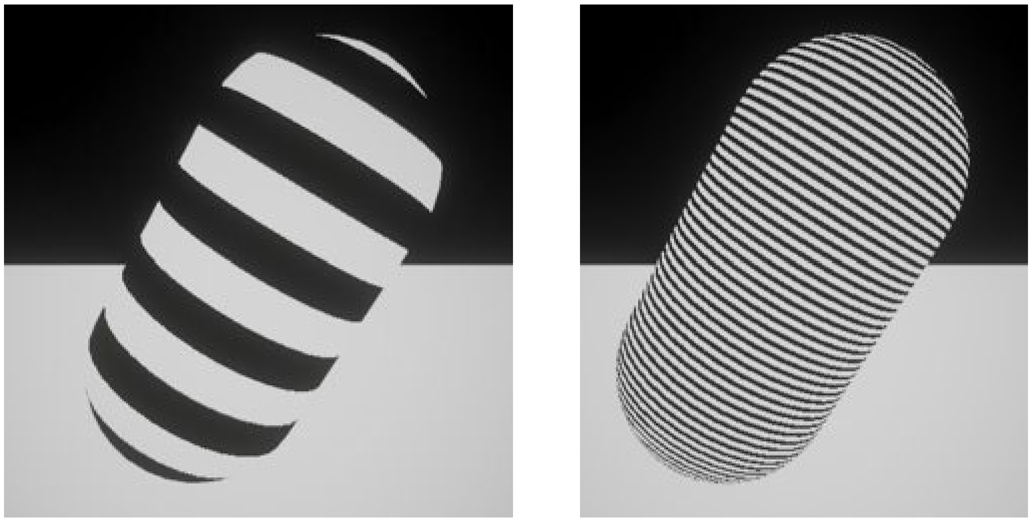


Figure 13: Applied equation, higher frequency

## 2.4. LIGHTING INFORMATION

So there we have it, now there is a way of showing lines on a mesh mathematically. We can also change the thickness of the lines by changing a single variable 'a'. The next step is to link this variable to the actual light level that is present on that certain spot.

For this we will need a way to know what the actual light level is at that spot. How do we find this? Well depending on how far we want to go in this the method can become very complex. Will the mesh take other meshes' cast shadow into account? To what extent will each method influence the performance of the shader?

To begin with we will create a very simple way to visualize the lighting on a mesh. We will take the dot product of the normal at that specific point on the mesh and the direction of the light present in the scene. This is an easy way that still looks good for what we currently have. First I created a shader that visualizes the normal vectors on each point of the shape. Then I went further to create the first shader that takes into account lighting information.

The dot product of 2 unit vectors is a value between –1 and 1. if the value is 1 this means that the angle between the 2 vectors is 0 degrees. If the value is 0 the angle is 90 degrees and if the value is –1 the angle is 180 degrees. This is a very easy way to calculate how light a point of a mesh should be because a value of 1 signifies the highest light value possible of the light source, and a value of –1 means the lowest light value. This means that if we just place these values as the color on the mesh we should get a shape that has a very clear form because of the shading.

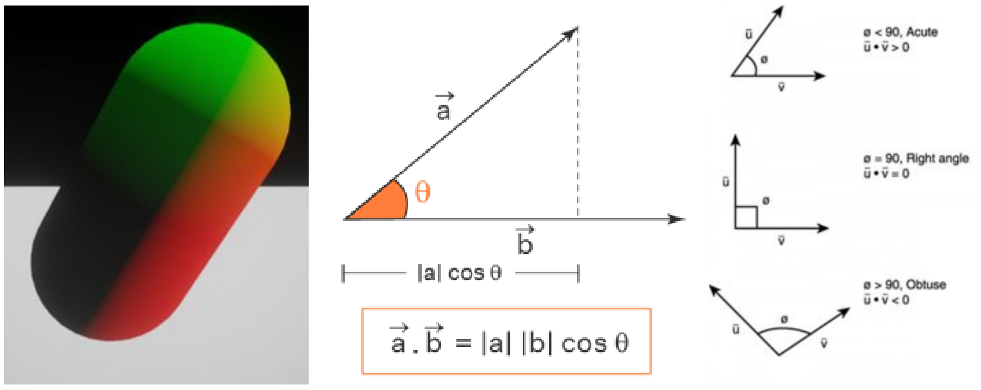

$\vec{a}$
$\theta$
$\vec{b}$
$|a| \cos\theta$
$\vec{a} . \vec{b} = |a|\,|b| \cos\theta$
ø < 90, Acute
$\vec{u} \cdot \vec{v} > 0$
ø = 90, Right angle
$\vec{u} \cdot \vec{v} = 0$
ø > 90, Obtuse
$\vec{u} \cdot \vec{v} < 0$

Figure 14: Normals, dot product

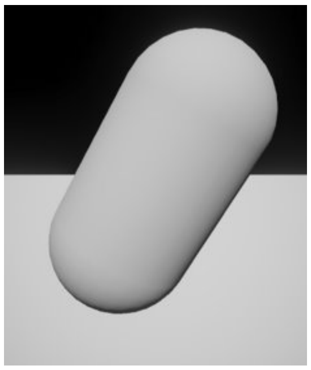


Figure 15: Smooth shading

## 2.5. ADDING LIGHT TO THE LINESHADER

Now we have a shader that draws lines, and a shader that shows the shading of an object in the current light. Now all we need to do is to combine these 2 to create a shader that changes a hatch depending on the light level.

The value we gain from the shaded material we have created is a value between –1 and 1. If we can just scale this to a variable between 0 and 1, we can use this value for the thickness of our lines.

It would be nice if we could change the softness of the hatched shadow and the size of the black area. This way we could tweak some variables to change the look of the hatch. If we just add a variable to the function that gets multiplied by the sawtooth-function we already have, we can change the range over which the shadow goes from black to white, and so change the softness of the generated shadow.

Of course, now the entire shape gets darker so we will just need an offset of the entire sawtooth function to change the size of the highlight. This way we can have a smaller area of transition while still having a shape that is as light as you want.

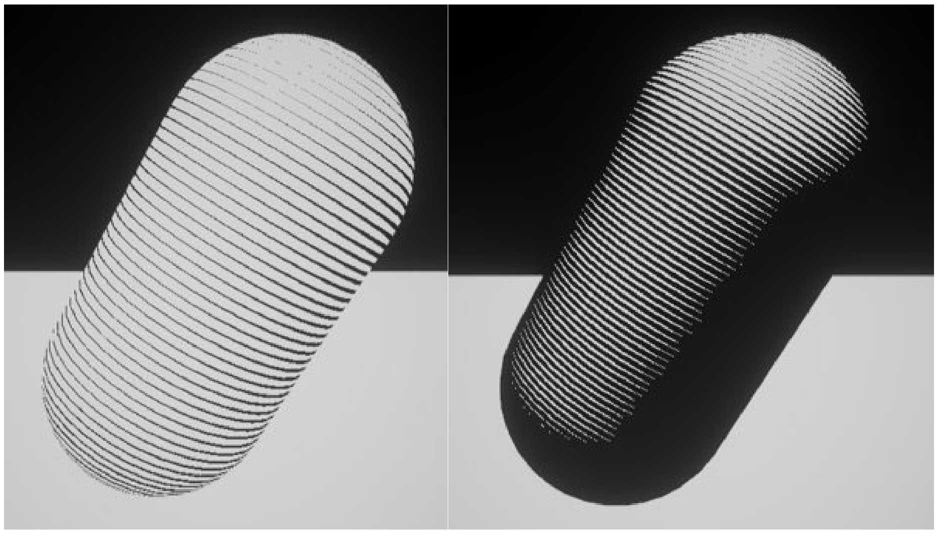


Figure 16: Lit lines, lower softness

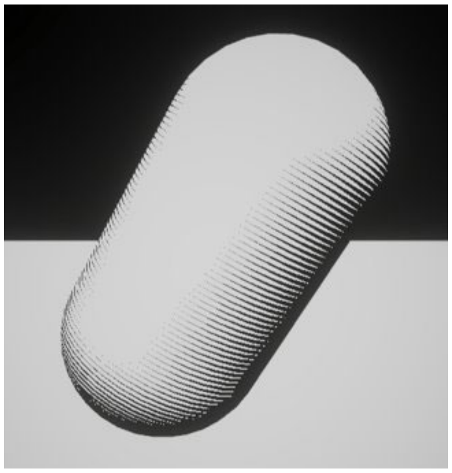


Figure 17: Larger highlight

## 2.6. INTRICATE OBJECTS

Now that we have something that interacts with light and looks interesting at least, we can try to add the shader to other geometry. This way we can see if it looks good on other, less simple objects. And if the shapes are still readable etc.

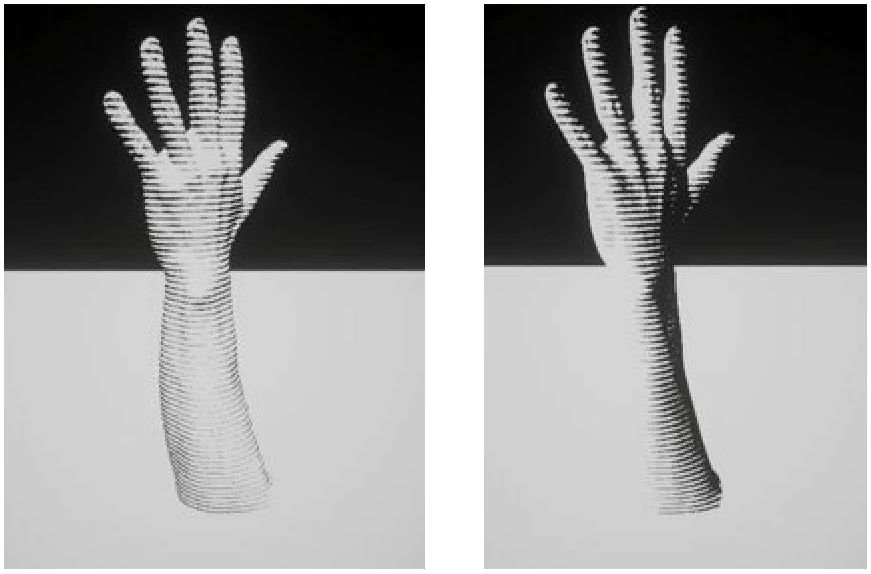


Figure 18: Arm with shader, tweaked variables

We can see that the shader works, but we can also see that there are certain flaws. The arm itself looks nice, the lines follow the object nicely and the shapes and depth are still very readable. The problem arises when the shader reaches the hand. The lines look like they are just cutting the hand and it has more of a holographic feel than a hatched feel. On the fingers the problem becomes the clearest, especially the pinky finger and the thumb. The lines are not following the shape of the model anymore, they look weird and don't feel quite right.

Therefore, we will have to be able to change the direction of the hatch depending on which part of the object we are looking at.

## 2.7. FINDING THE HATCH DIRECTION

Now to tackle the problem of finding the correct direction the hatching lines should follow. There are a couple of ideas I had to try to figure out this problem. The first way is to calculate the lines by creating several parallel planes that 'emit' from the light source. Then the intersections between the mesh and the planes would be used to determine the direction. The spacing of the planes would determine the spacing of the lines. A drawback of this method is that even though the lines follow the contours of the shape, the lines would all be drawn parallel to each other. Which would make it so there is an angle from which the entire shape would seem flat. This also seems like a method that is very demanding for the computer.

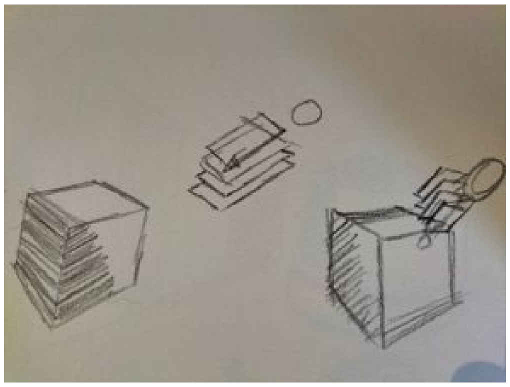


Figure 19: Direction idea 1

Another possibility is by using unused vertex data like vertex color. Color is a 4-dimensional vector (R, G, B, A). If the color of the vertices are not used, other information could be stored in this data slot. For example, another vector (X, Y, Z, W). This information could be used to show the direction of the vector that is perpendicular to the 'plane' the hatching line makes.

But there is a drawback with this method too. This way the shader could not just be added to any model, but the model needs to be prepared first. This can be done manually or by writing a tool for some modeling software.

Another way to possibly do this is by reading the rigging information. Every vertex is connected to a certain part of a rig, and if we can get the direction of the bone the vertex is connected to, we could use this information to determine the direction of the hatching lines.

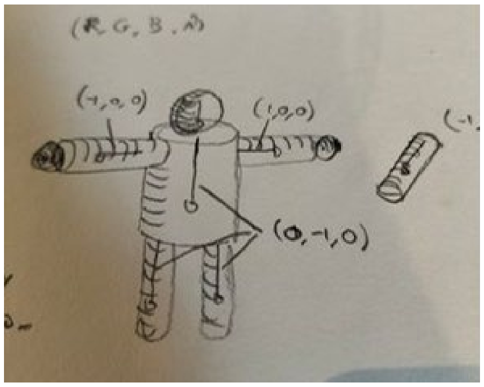


Figure 20: Direction idea 2

## 2.8. CHANGING THE HATCH DIRECTION

Now that we have a way of drawing lines on a mesh, we will need a way to change the orientation of the lines that get drawn on the mesh.

Right now, we take the local value on the y-axis to put into the equation. All we need to do is find a way to put the value on any axis into the equation. To do this we can take the dot-product of the local position and the vector around which we would like to draw our lines. The dot-product projects the local position onto the vector and the value it returns is the value of our point projected onto that vector.

This way we can change the formula to incorporate the lines being drawn around a vector that we can enter ourselves:

Value = floor (abs(abs((f * 2 * **dot(localPosition, directionVector))** % 2) - 1))) – a

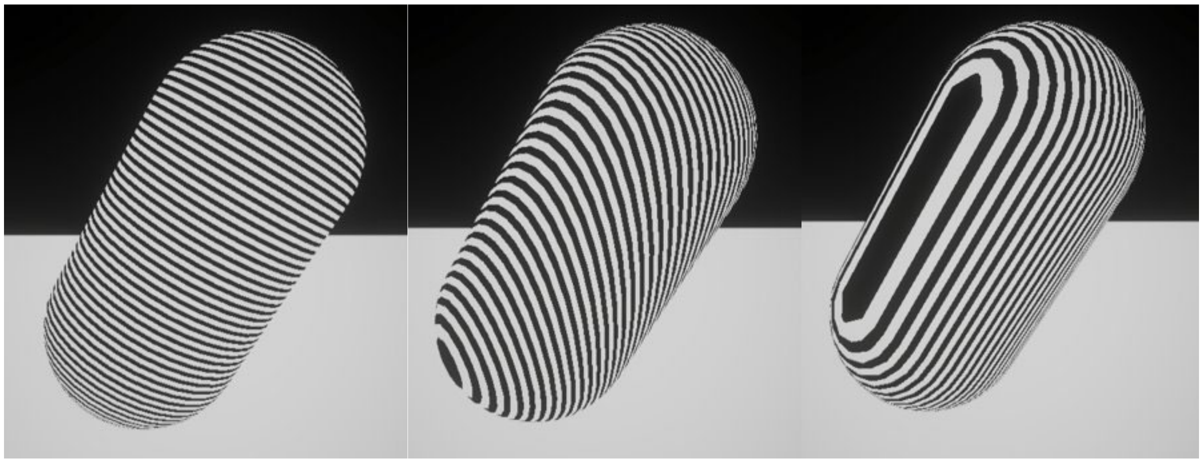


Figure 21: Rotating direction

## 2.9. CHECKING FOR SEAMS

Of course, we will have to make sure that when the same model has different directions, the transitions between these directions also doesn't cause too many problems like seams for example.

We can do this for now by making the direction change depending on a value. We will pick the local y-position of the point on the mesh.

First, we will need to generate a vector that rotates more the higher the local y-position is. We can do this by using trigonometry.

We can generate a direction by using a sin and cos function. If we set the x-value of the direction to sin(y-local), the y-value of the direction to cos(y-local), and the z-value to 0, we have a vector that rotates around the z-axis clockwise the higher it goes on the y-value and counterclockwise the lower it goes on the y-value.

And as we can see there are no seams or weird transitions.

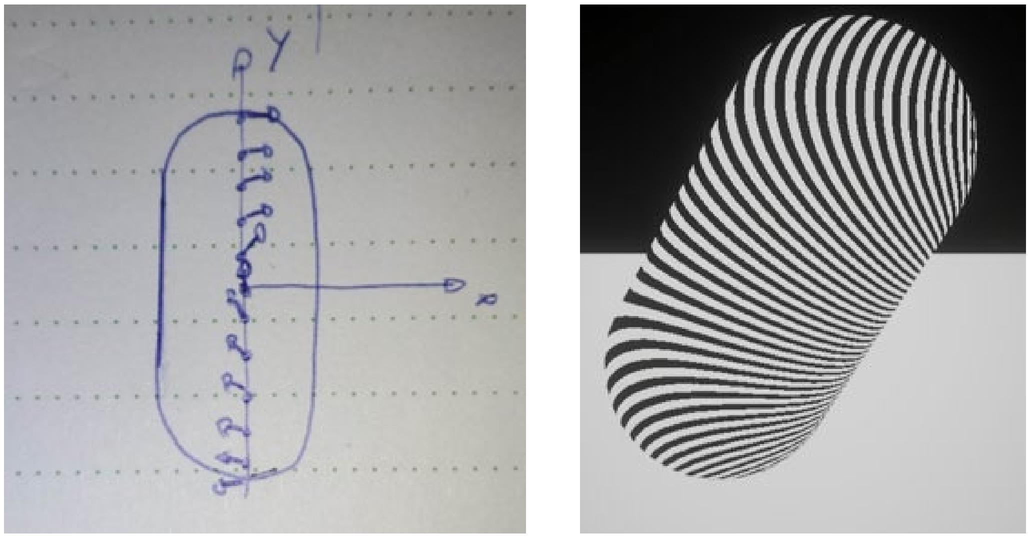


Figure 22: directional transition

## 2.10. ADDING RIG DATA TO THE MESH

What we need to do now is find a way to tell the shader what direction to use on which part of the mesh. The Idea I had for this was that I rig the mesh I wanted to use and write the directions of the bones to the color value of each vertex. It should also consider the weights of the rig to ensure smooth transitions between the different directions.

I have split this task up in several parts:

Firstly, I will need to learn how to rig models. I have decided to learn to rig in blender. This is primarily because I know the software will always be available seeing as it is free. I looked up some videos on how to do it and went ahead and gave it a try. I decided to us e model of a hand because the geometry did not seem too hard to rig, and there were a lot of edge cases where the geometry split in different ways (at the fingers for example).

Now that I have learned how to place a rig in the mesh, I just need to link the 2. this can be done by using the weight painting tool in blender. Another option is to just use the auto rigging function. Which is the option I went with because it worked very well and didn't take a lot of my time. Which is one of the goals I want to achieve with this shader, that it does not need too much preparation of the mesh for it to work.

When this has been completed, I need to find a way for this data to be available in the shader. Which is why I need a way for this data to be stored in the mesh instead of in a combination of the mesh and the rig. I can do this by storing the direction of the bones in the mesh in the color property of the different vertices. Instead of manually setting all these values I wanted to make a tool that took the mesh and the rig I just created to generate these values in the correct way.

Finally, I just need to find these values in the shader and use them to set the direction of the hatch.

## 2.11. RIGGING

The first thing I need to do is find a nice model to use. I found a free asset for game ready arms, so I decided to take them and cut off the parts I didn't need to achieve the model you can see below.

Next, I placed the bones of the rig inside the model and moved the joints until I was satisfied with where every bone lined up. Then I needed to skin the mesh to the rig. There is a tool in blender that automatically sets the weights and bone references, so the skinning process becomes a lot easier. After that, I could move the bones and the mesh followed. The skinning was done well, and everything looked about right to me.

I changed the pose of the hand into something that something else to make sure that it still looked good even after being moved. And as you can see below the result looks pretty nice.

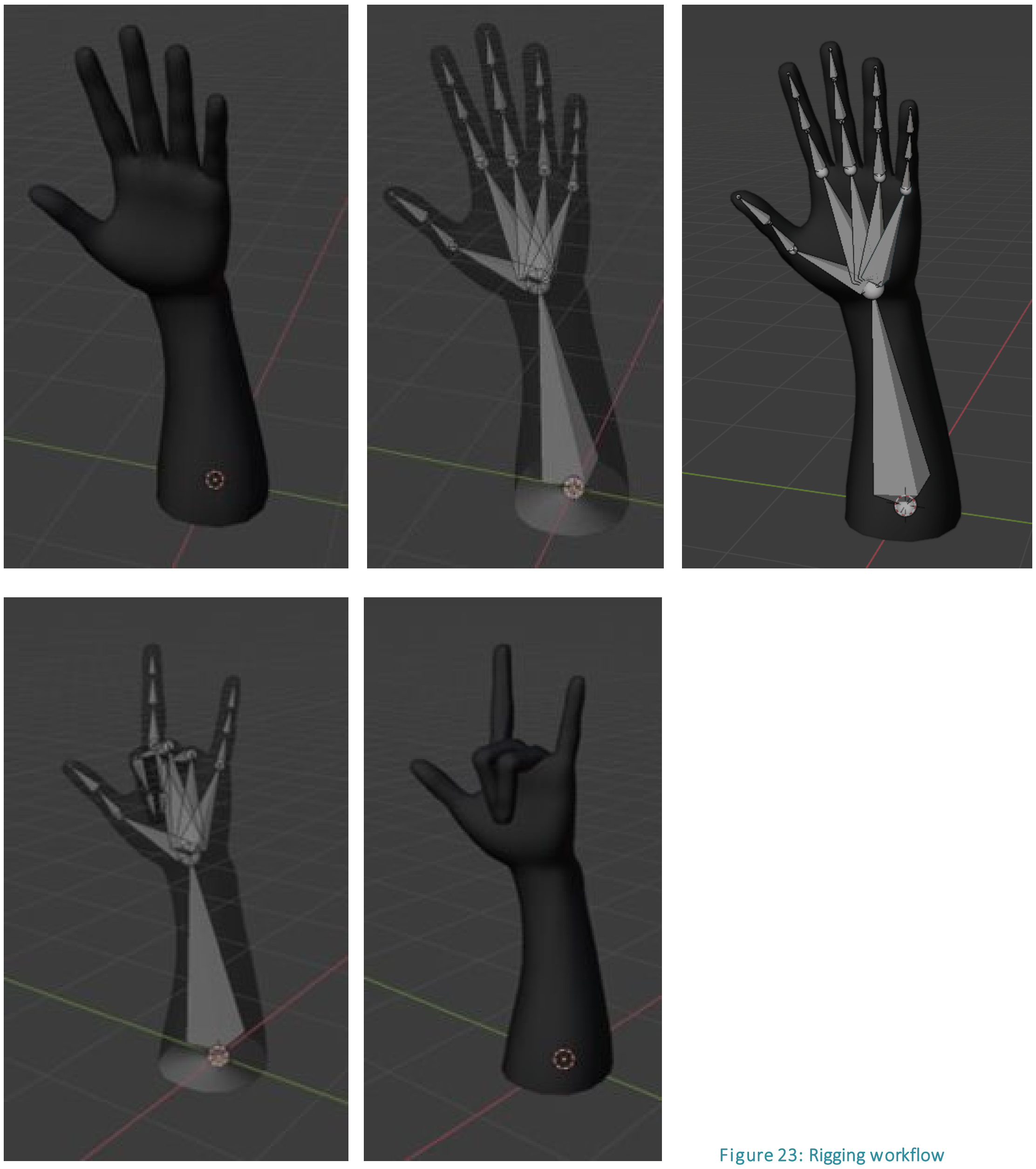


Figure 23: Rigging workflow

## 2.12. BAKING THE BONE DIRECTION INTO THE MESH

Now that I had a mesh that was completely rigged, I wanted to convert this rig data into data that was available in the mesh itself.

What I needed for the shader is comparable with a flow map. Except for the fact that I did not put the data into a texture but I would store the data in one of the vertex variables. Since I did not use the vertex color I decided I was going to store the rig data in the vertex color slot.

Now what exactly do I want to store?

Well, since I am trying to find a solution for the hatch direction issue, I want to have some way to store the direction in the vertex. I rigged the model because the direction I need is basically the direction of the bones. The shader is written in such a way that the lines would form "rings" around the direction, so if the finger would tilt left, the rings would tilt with it.

Since the model and the rig were already in Blender, I decided to learn how to write scripts in Blender. The syntax was relatively easy to learn, but learning how to access or where to find the data from either the rig, the mesh, or the individual vertices, was a lot bigger of an obstacle. Then there was the small issue that Blender uses a right handed coordinate system while unity uses a left handed one. Also in Unity the Y-axis is up while in Blender the Z-axis is up. These issues were easily solved by storing the direction as (x, -z, y) instead of simply (x, y, z).

Below you can see some examples of the direction errors occurring without taking these changes into account.

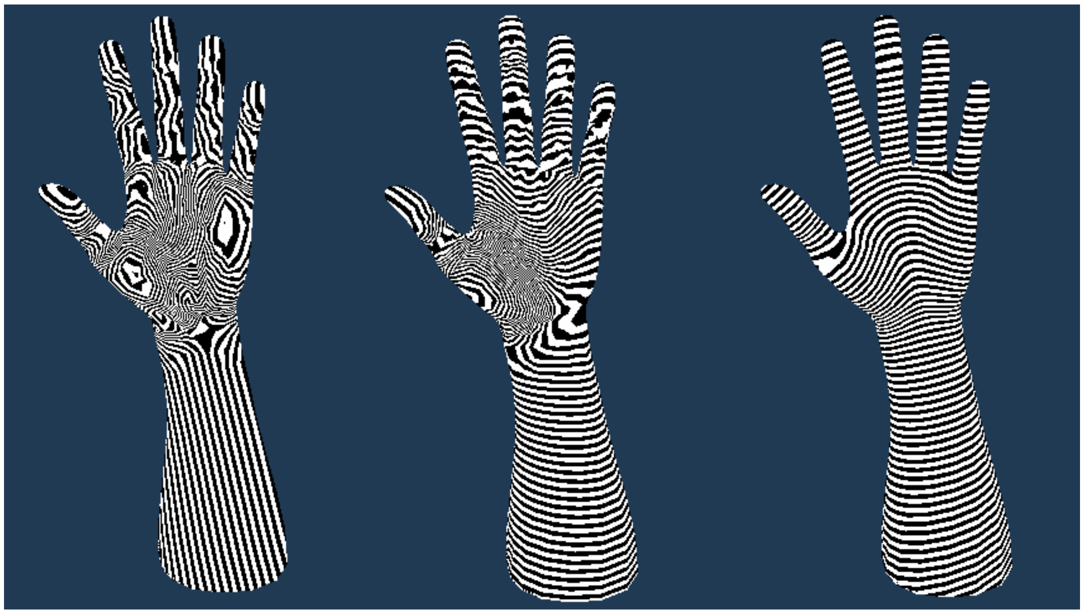


Figure 24: Wrong, Wrong, Correct

Now the hard part. Converting rig data into a simple per vertex color variable. Since the color is basically a vector with 4 variables (R, G, B, A), a direction can easily be stored inside it (X, Y, Z, W). To store the correct data I looked at each vertex and looked for the weight of each bone on that vertex. Then I took the difference of the ending position of the bones and the starting position of the bones (the direction of the bones) multiplied it by the weight of that particular bone on the current vertex and added all those up.

Using the weights made it so the transitions were smoother from one part of the mesh to another.

One more thing I decided to do was to change the data a little bit. Since color values are always positive and never larger than 1, this means that the direction could also not be negative, which is a problem in this case. I remapped the data of the values from (-1, -1, -1, -1) - (1, 1, 1, 1) to (0, 0, 0, 0) - (1, 1, 1, 1). This made it so that a null vector in the shader was not (0, 0, 0, 0) but rather (0.5, 0.5, 0.5, 0.5). I just had to make sure to account for this in the shader itself too.

Now I had a mesh where the correct direction was stored in the vertex color slot of the vertices. In the Image you can see the vertex colors, and translated to lines visualized compared to a shader that does not consider direction.

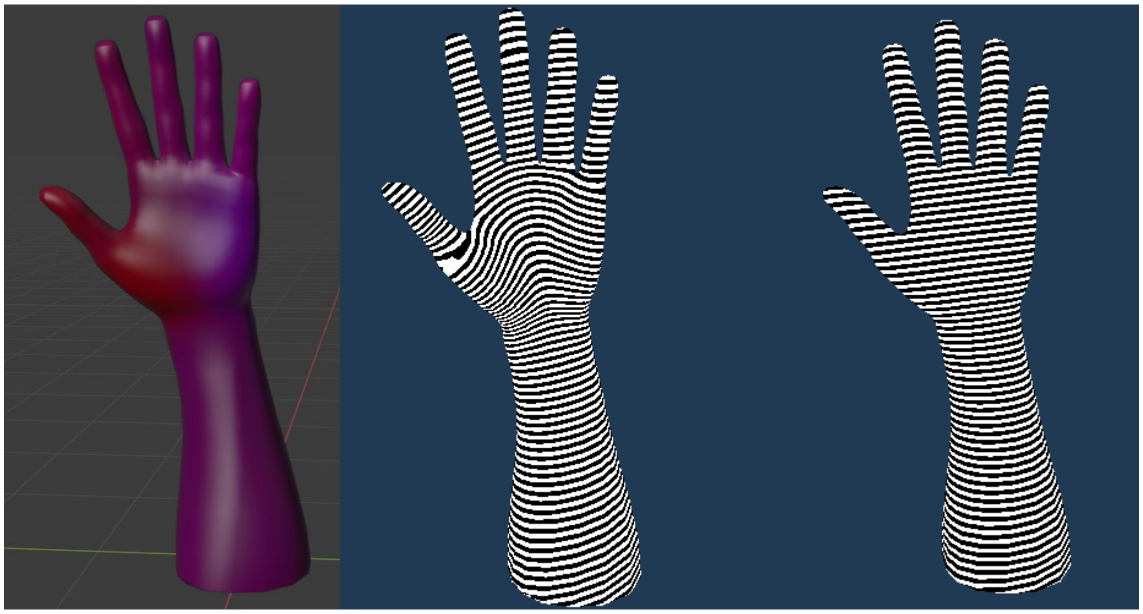


Figure 25: Color, Directed lines, Normal Lines

Another thing I had to take into consideration was the fact that not every file type stored the correct data. I tried FBX, but the normals were not correctly stored. I tried OBJ but these did not store color data seemingly at all. But then finally I found one that seemed to store exactly what I needed, in the format I needed, The DAE file Format.

Here is an example of how the model looked with the shader in the wrong file format.

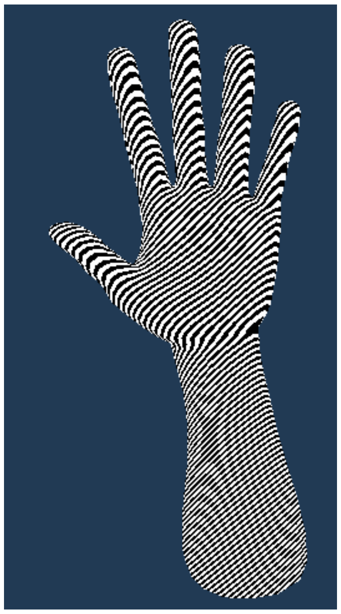


Figure 26: OBJ

Because I had a little bit of trouble with the conversion from the rig data to mesh data, I had planned a meeting with one of the professors at school, Stijn van Coillie. After some discussion I was able to try some new things and got it working correctly not much later.

He had given me some ideas about what other things were possible in shaders. He told me about the existence of flow maps, which could also be used. The problem with this is that the models would have to be unwrapped as well. Which I was trying to avoid because I wanted the workflow of this shader to be relatively quick.

Another thing he told me about were geometry shaders. In geometry shaders you have more information about nearby vertices which would make it possible to calculate these directions in shader instead of baking them manually. This would make it so there is even less preparation necessary on the models, but geometry shaders are very demanding. Since I had already found a way of doing what I needed to do I decided to leave the geometry shader alone and put it on the list of things to possibly revisit later on in the development of the shader.

## 2.13. RIM LIGHTING

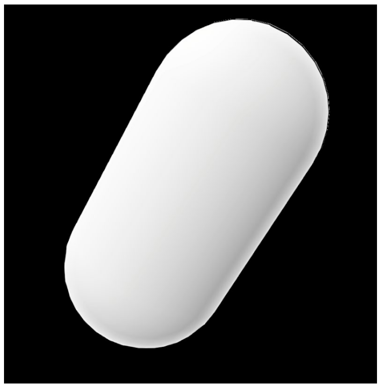
Since the shader uses only black and white, the chance exists that the object is black and the background is black as well. This makes it so that there is no contrast and the shape can barely be read.

To combat this I decided to add a rim light effect to the shader. A rim light is an area around the edges where the object is brighter. This makes it so the shape of the object is stays legible in most situations.

To implement the rim light I needed to dive into the shader again. Rim lights are typically implemented by comparing the normal of the object against the view direction of the camera, and this is exactly what I did as well.

We can use the dot product of the view direction and the light direction as a factor with whose absolute value we can multiply the overall light level of the model. Because if the dot product is 0 this means that this is a normal that is perpendicular to the camera. This basically means that on a rounded object this is the edge. We can multiply the dot product with another value to change the strength of the rim light.

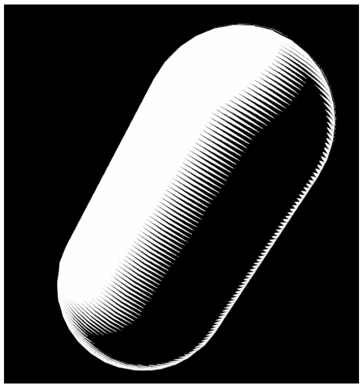
To visualize what the dot product does I wrote a shader of a smoothly lit material with a rim light to visualize what area it would affect.



You can see on the edges of the model that they are a little bit brighter. This can be seen a lot better on the darker side of the object because this is the part where the light has been affected however it has barely any effect on the lighter side of the object.

Figure 27: Smooth rim light

Now that we had seen the impact on a normal lit material it was time to implement it in my own shader and see if it had the same effect.



As you can see it definitely helps with the contrast issue. In general the hatching shader is darker but that is because the values have been tweaked so the 'gray' area is squashed down into a smaller portion of the model.

Figure 28: Hatched rim light

## 2.14. OUTLINES

Having gotten as far as I had made me realize that even though the shader worked and clearly conveyed how the light was set in the scene, that it was still missing something. I did not quite realize what it was until I started the case study.

The case study is basically a recreation of a drawing made by a hatching artist (McBess). Later on I the paper you will be able to read more about the case study.

After I had chosen an image to recreate I started modeling one of the props in the image. This is where I realized that even though the shadows are visualized nicely in the shader, the image was not complete without the outline. There are drawings that do not need outlines, but this is because there is a lot of stuff going on in the background that creates enough contrast between the two. This is something the shader can visualize well. But seeing as the piece I am trying to recreate is a piece in a more cartoony style, the outlines made the drawing what it was.

Initially I had the idea of doing something similar as I did with the rim lighting, where this is how the edges would be detected. This Idea however started to fall apart quickly when I realized that this did not work at all with models with sharper edges. Though these were actually the edges that benefitted most from the outlines.

So I started doing more research on how outlines are done in shaders. It turns out not to be as simple as just adding some lines of code to the shader I had at that point.

Apparently outlines are a screen space effect because the points where the outlines should be cannot be calculated when there is no information about the neighboring vertices. Which in a non-geometry shader is the case.

After reading this article about methods of doing this I found out that usually a special texture is generated. This is not a texture like a texture you would use to texture a model. But rather an image of what is on screen. This does not necessarily mean that it shows exactly what you see on the screen but can also be an image with a 'filter' over it. Then this image can be used to calculate where the 'hard edges' are. These can then be amplified with a line.

If you think about it to achieve a line on a hard edge you need to draw a line where there is a lot of contrast in the normals. But if you would have two separate planes identically oriented, but the one is a bit behind the other, no line would be drawn because on the texture these planes would have the same color. This means that you will also need to take into account how deep each plane is from the camera. Now the difference in depth can also be visualized by a line.

So what we need is a normal texture and a depth texture. Luckily you can find online how to generate both these textures. And even better there is a way of combining these two textures into one texture called the depth-normals texture

After having done this all that was left to do was using an edge detection algorithm to draw the outlines overtop this depth-normals texture. Several techniques to do this could also be found online so I followed a tutorial. You can follow this link to see how the method works.

Now that the shader had outlines, it could be considered finished as far as my purposes were concerned.

# CASE STUDY

## 1. INTRODUCTION

In this chapter We will discuss the making of the case study. For my case study I decided to use a drawing made by the artist “McBess” and recreate it in 3D. It would use my shader and the goal is that it would look somewhat like the original.

I chose McBess as an artist because of the way my shader was turning out. The style my shader could deliver was comparable to the style McBess had in his drawings. So I started work on some props that were present in the scene and tried using my shader on them.

I will talk about the process of working on the case study and the problems I ran into.

Some information will be given about how to start work on new models that use this shader. What some of the drawbacks are, and how I have found those through working with it to create a scene.

## 2. STARTING THE STUDY

### 2.1. REFERENCE

The reference I used for my case study is the following image of the artist McBess.

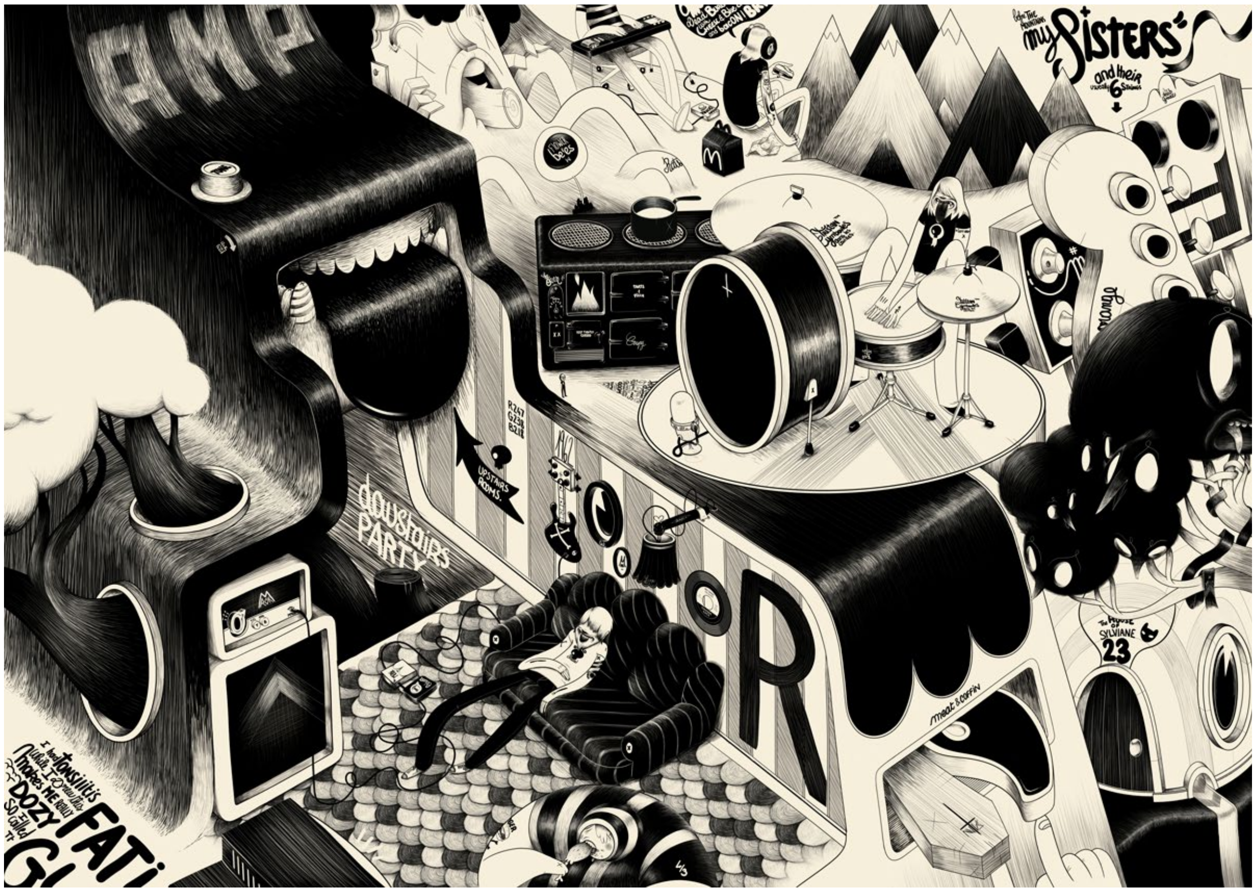


Figure 29: McBess - Dark Twisted Toons

This was primarily because in the shader I currently had a lot of the shading was very similar to the style McBess used. This was in large part due to the fact that McBess uses the contour shading technique for a lot of the objects in his drawings.

## 2.2. FIRST PROPS

The first thing I started with was the bass drum you can see in the reference Image above. I did this because it was a very simple shape and the color variables could easily be set manually, making it a lot quicker of an object to create because I would not have to go through the process of rigging and running the python script.

After I had done this the result I had was certainly promising, but it did not nearly look like what I was expecting. This was largely due to the fact that there were no outlines at this point.

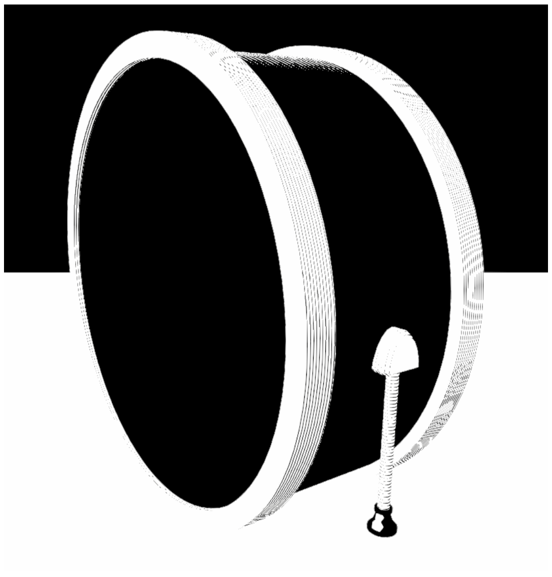


Figure 30: Object - no outlines

As you can see there is no contrast between some parts of the object and the background. It also does not have the same feeling as the drum has in the original drawing.

One of the big problems I had with the outline shader, is the fact that for some reason any objects with my shader on it were not displayed on the depth normals texture talked about in the chapter about the outlines in the previous part of the paper. This meant that the outlines were not drawn on these objects either because the outlines are drawn overtop this texture.

The reason they were not displayed is because the shader I wrote in ShaderLab was an unlit shader. This made it so while drawing the depth texture those objects were not displayed because Unity uses a shadow casting system to generate this depth texture. Since no shadows were cast on these objects no depth data was recorded and thus these objects were invisible to the depth texture.

What I did to fix this is rewriting the entire shader in a lit ShaderGraph shader. This was not as big as an undertaking as it sounds, because a lot of the features are just extensions on the same equation. After I had done that the shaded objects were correctly displayed on the texture and so the outlines were finally drawn on the objects correctly.

After some trying and failing eventually I found a way to visualize outlines on the scene. The result that followed was this.

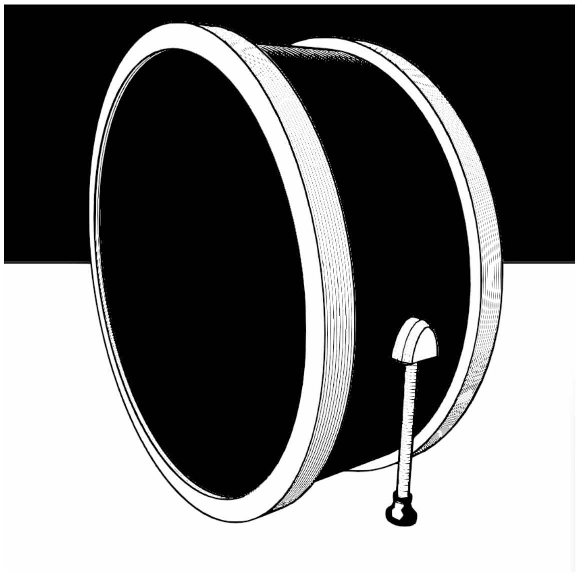


Figure 31: Object - Outlines

As you can see the shader now looks a lot more like the drawing I am trying to recreate. Now that I had a simple object successfully looking like an object from the drawing I was ready to start working on the rest of the scene.

## 3. SETTING HATCHING DIRECTIONS

One of the more important things in this project is the fact that the hatching direction is present in the shader and looks right. But how do you actually set this direction?

There are two options. Either one works just as well but they both have different use cases.

The first one is the simplest one, but this one is only handy in the cases where you want to set the direction of simple objects where there is not too much changing of the direction. Like the drum or a simple sphere.

In this method you will set the direction of the hatch manually. You can do this by doing a simple calculation.

If you take a cylinder and you want the hatching lines to basically create circles around the x-axis, you would set the vertex color to (1, 0, 0, 0). But as you may recall we remapped these values to a range between 0 and 1 instead of a range between -1 and 1. So the resulting value would be (1, 0.5, 0.5, 0). The 0 in the a channel of the vertex color does not really matter at all because we do not use this one in the shader seeing as a direction is only a three dimensional vector. Maybe as an extension of the shader this could be used as another variable to make some parts of the mesh lighter then others for example. Setting these variables can easily be done using several programs like maya or blender. Below you can see an image of a vertex colored object.

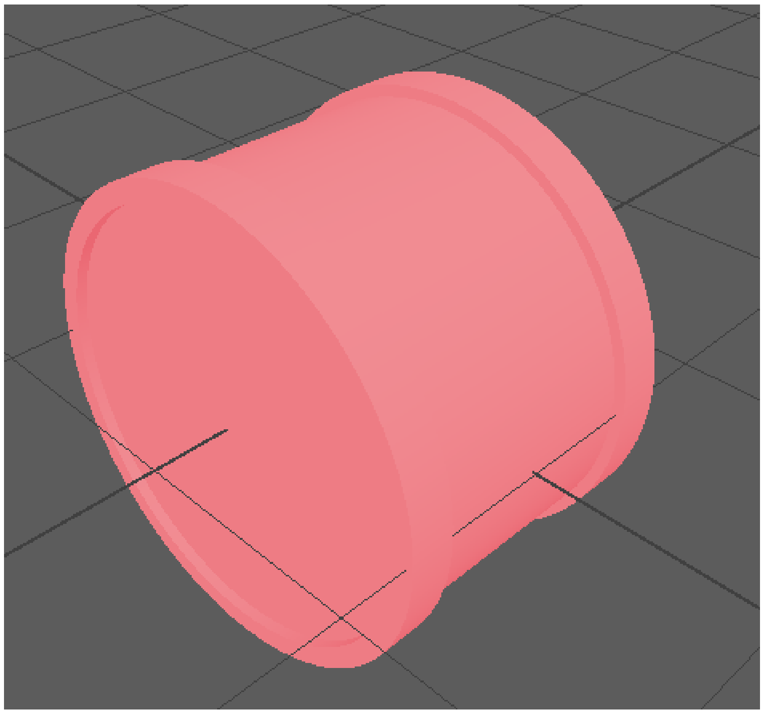
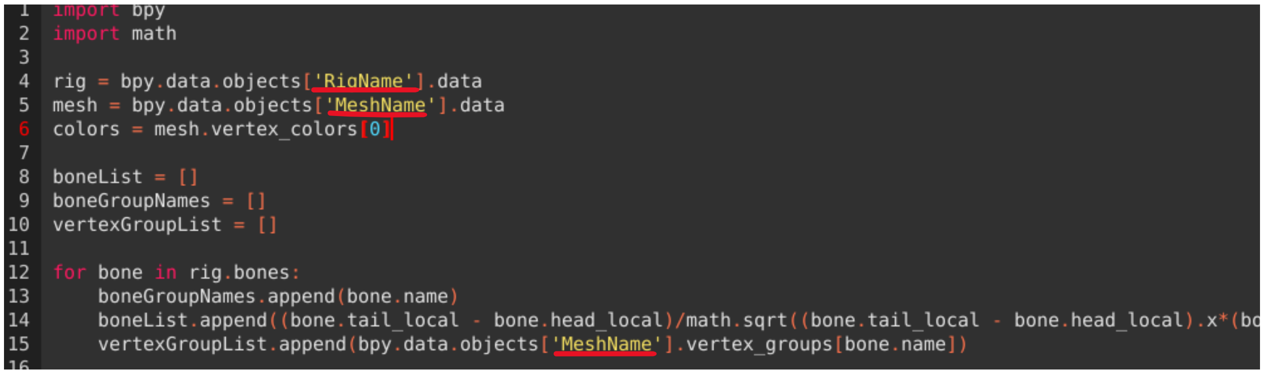


Figure 32: Color manual

In the other method this is a bit more work, but it is perfect for shapes that are a little bit more intricate or that have a lot of change in the hatching direction.

The workflow is as follows. First you rig the model via your program of choice. In my case this was blender. When you are content with the way everything moves and you feel like there are no weird parts in the rig.

Now you can import the rigged mesh into blender (if it is not there already) and you can run the script I wrote. There are a few lines you will need to change depending on what your objects and your rig are named to make sure that the script knows which object to use as the mesh and which object to use as the rig.



```
import bpy
import math

rig = bpy.data.objects['RigName'].data
mesh = bpy.data.objects['MeshName'].data
colors = mesh.vertex_colors[0]

boneList = []
boneGroupNames = []
vertexGroupList = []

for bone in rig.bones:
    boneGroupNames.append(bone.name)
    boneList.append((bone.tail_local - bone.head_local)/math.sqrt((bone.tail_local - bone.head_local).x*(bo
    vertexGroupList.append(bpy.data.objects['MeshName'].vertex_groups[bone.name])
```

After running the script and setting the mode to vertex paint you will have an object that looks like this.

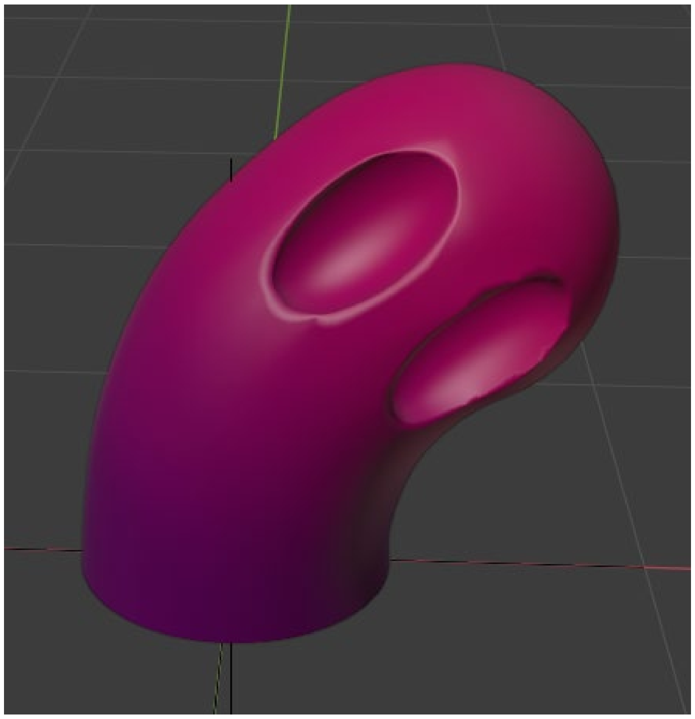


Figure 33: Color via rig

You can check if this is correct by looking at different parts and seeing if the color lines up with the axis color. In this example blue is up, and at the bottom, where the bone is pointing upward, the model is more blue than it is at the top part, where it is tilting in the x-direction, or as you can also see on the image, the red direction.

As you can see on the following images the lines are now nicely following the shapes. You can see that the outlines are not working but that is because the flat line shader is still an old shader that still uses ShaderLab.

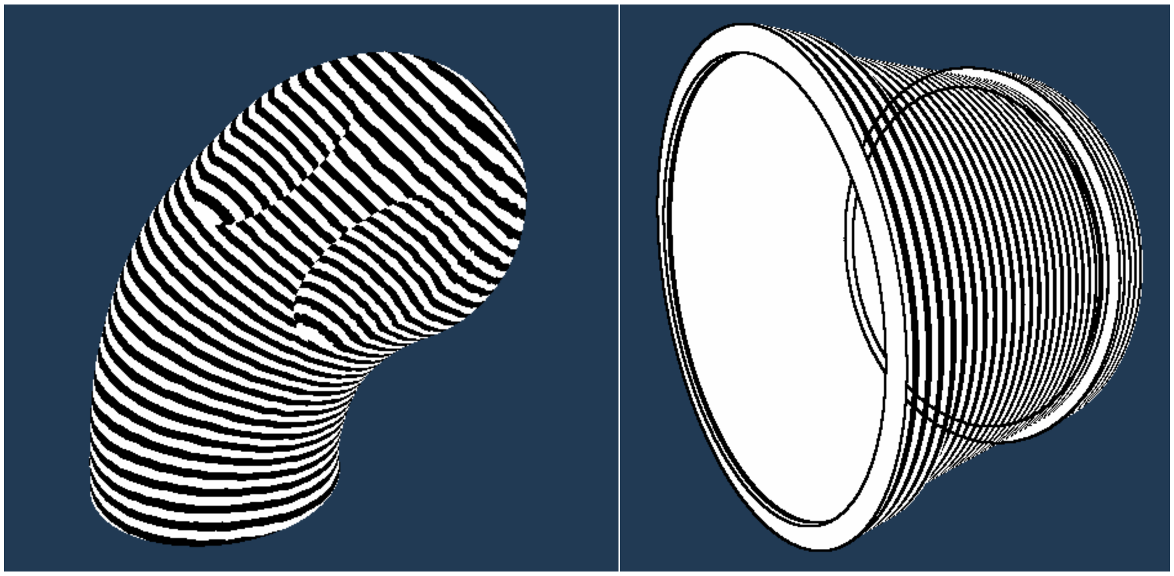


Figure 34: Lines rigged, manual

## 4. CREATING DIFFERENT MATERIALS

Since I made it a priority to be able to change a lot of settings easily, it was possible to change the way a material looked drastically by only changing a few variables. Of course I could have white or black materials that interacted with light and had a shadow zone and a light zone, but I could also create gray shades that became darker if the light was not hitting them directly.

By creating different materials using the same shader I was able to create a scene that had a lot of variation in value even thought the only two colors being used were black and white. A fun example of this is the tablecloth I made. You can see that there are different shades of gray that are created solely with black lines. And at the parts where the tablecloth has been folded the thickness of both colors is different because the light is different too, but you can still see the different squares.

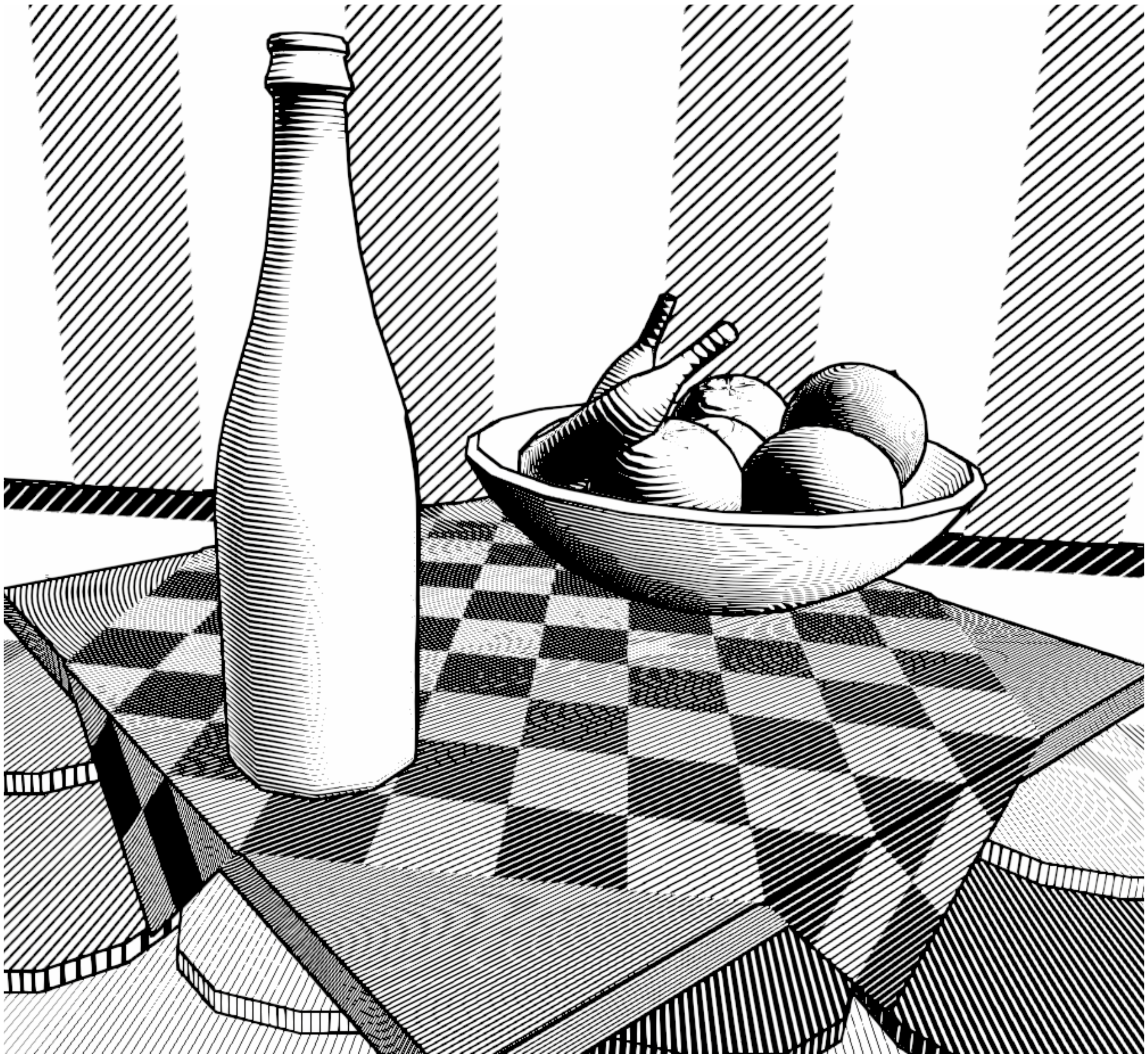


Figure 35: Different shades

## 5. CASE STUDY SCENE

When I started to create rest of the scene I started by creating the larger piece that presented as the pedestal on which the props rested.

A lot of this object looked black because of the fact that the object was covered by something I assumed represented a black carpet. The rest of the object was completely white. This made it so I did not quite feel like my shader was showing its full potential. Because there was primarily areas with very large contrast and the Idea of a hatching shader is to show the illusion of gray values.

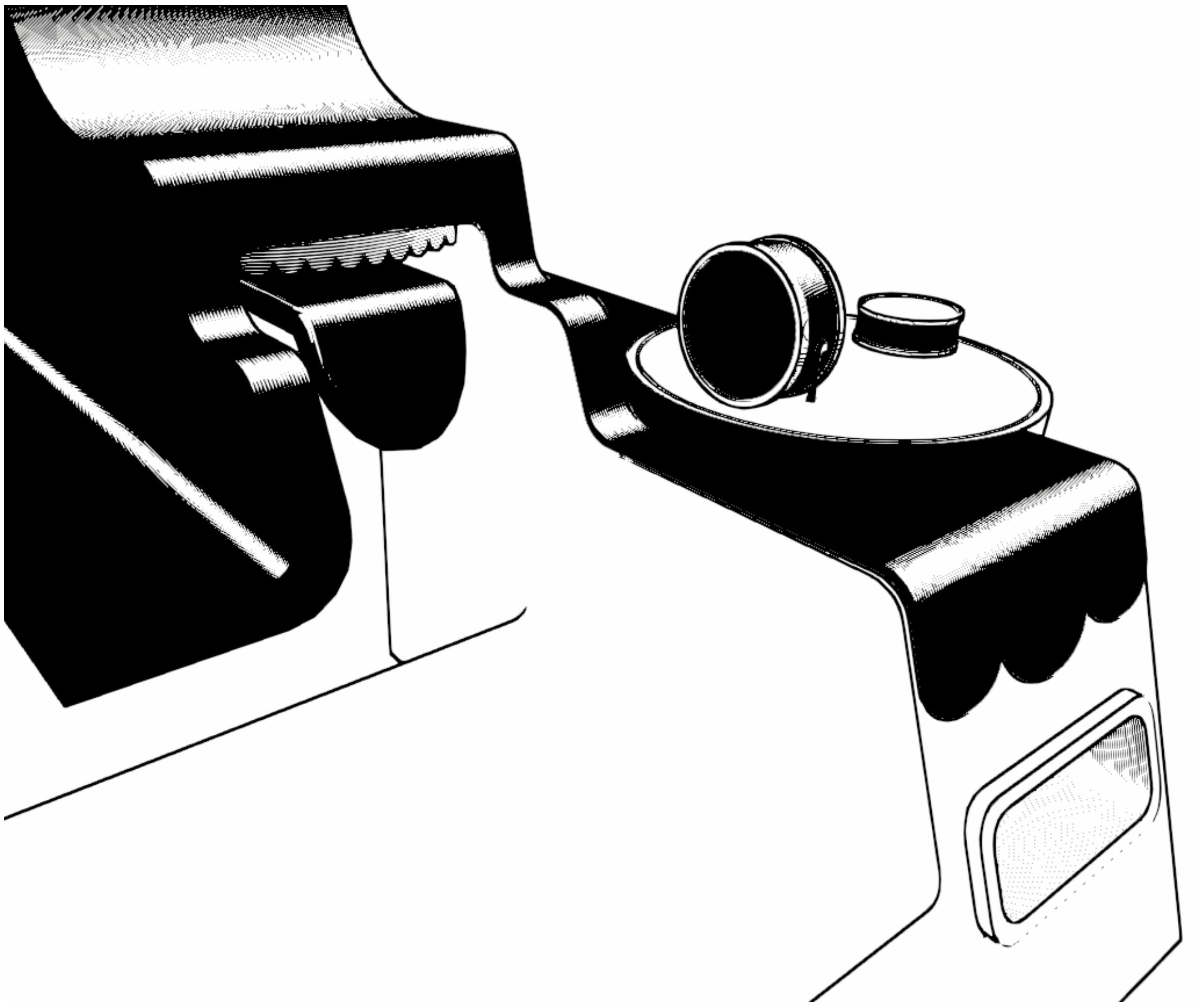


Figure 36: Case study start

This is where I decided to start working on more props. This was to be able to create something to break up the larger white or black areas. Since I felt like my shader was useful in several different ways I decided to create props that were not necessarily present in the drawing, but rather props that would look good using my shader.

Another thing was the fact that I had not quite shown a lot of gray materials in the scene. So I made the stripes on the wall, the tiles on the floor, and recolored another wall to break up the diorama and make it look more interesting.

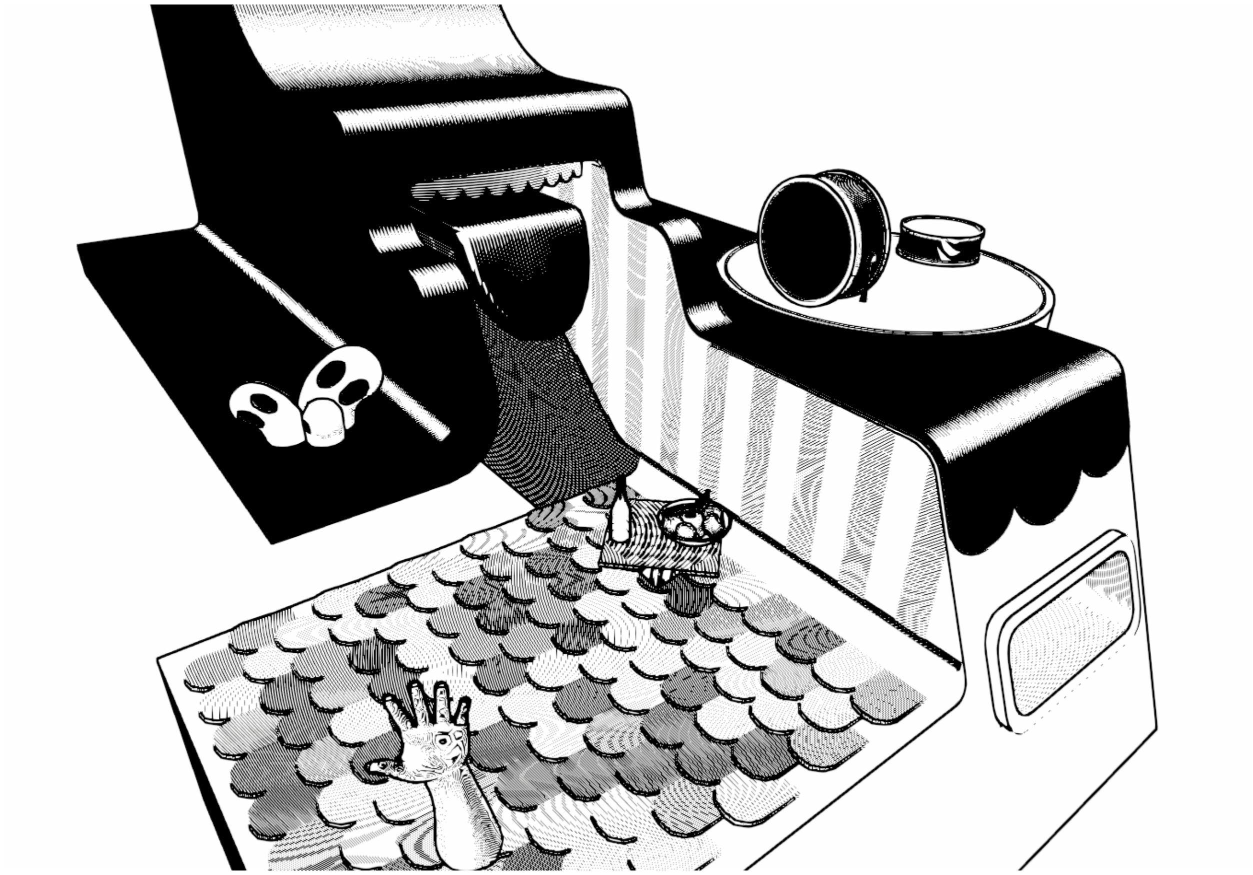


Figure 37: Case study full

As you can see the scene looks a lot more interesting already. One problem that I have noticed here however is the fact that because the render is on a computer screen, it is hard to see the lines when the view is zoomed out as much as it is. Because of this I feel like the shader is a lot more suited for scenes that are zoomed in a bit more than the views that are more of an overview.

## 6. CREATING AN INTERACTIVE EXPERIENCE

Since there was such a large difference in how good the scene looked depending on the viewing distance I decided to make the scene an interactive experience.

I did this by making the observer be able to toggle between a couple of set pieces he can look at. You can then cycle between the objects and return to an overview. When there is an object in focus you can also rotate around it to find an angle more to your liking.

Another thing you can do is rotate the light source. This was done to make sure that one of the stronger elements of the shader can easily be seen. The changing light levels that smoothly go from one to another. This way you can quickly create a cool view by rotating either the camera or the light source.

This solved the problem of only being able to see objects from a distance in the overview. And there could be close ups of several objects in the scenes because the shader works better from a shorter distance due to the anti-aliasing on a computer screen.

I made it so the camera cannot rotate around the overview because I wanted it to still be recognizable from the image, and the overview is also not quite as beautiful from every angle.

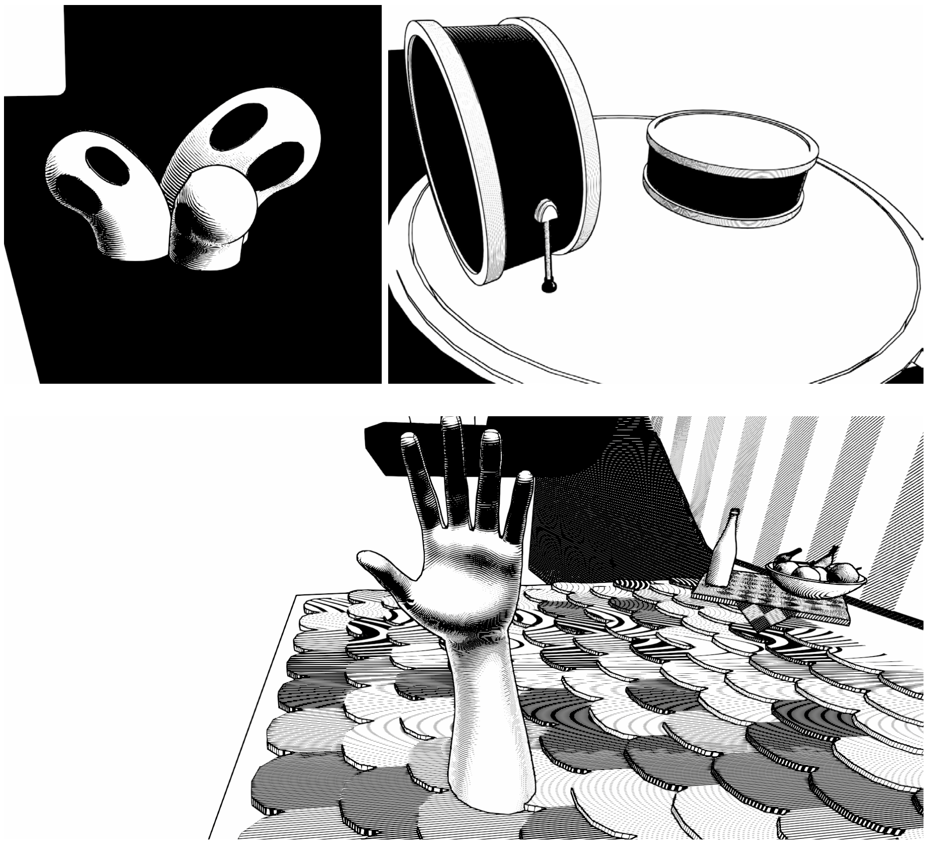


Figure 38: Case Study focuspoints

## DISCUSSION

In general the project went relatively smoothly. Of course there was the occasional hiccup, but nothing major.

The first part I had actually struggled with quite a bit is the most important part of the paper. Finding a way of figuring out the direction the lines should follow. The method that I used in the end of the project was based on an Idea that came relatively quickly in the project. The problem was however the execution of this idea.

This was probably due to my lack of experience in a lot of the things I needed for this method to work. The first one was that I needed to know how to rig, which was new to me. Then I needed to write a python script, which was new to me. And finally I needed a way to test if the directions were actually correctly stored by creating a shader, in which my experience was very limited.

The combination of these made it so it was very hard for me to find out where the problem lied when one arose. Making it fairly hectic in my mind at some points during the development of the shader.

Luckily I really liked working on the project because it felt like making a puzzle and making all pieces fall in their place. I fear that if I did not like the project I would have had a hard time staying motivated to continue working on the shader.

The second panic moment I experienced was when I wanted to add the outlines. I had not expected it to be quite as much work as it turned out to be. There were also a lot of things going on in the background that I did not entirely understand yet making it feel out of my control when something was not working.

When I was then forced to move the shader over to a ShaderGraph shader I had another moment where I did not want to continue. This feeling quickly faded however when I had just started on rewriting the shader since it was not quite as much work having already done all the calculations. It was just a different way of implementing the equations.

One of the things that I would have liked to be able to avoid was the amount of preparation necessary on the models before the shader works. Preferably I would just give the model a material that uses this shader and it would work. But the directions should be baked into the mesh through one of the two methods mentioned a couple chapters back. It does take less time then to UV unwrap the entire model over a hatching texture however. And using this method you also avoid seams if the model itself is prepared correctly.

This counts for the outlines as well. These outlines are not part of the shader placed on the models but rather is its own screen space shader. But these are not easy to install since you need to add some things to the renderer you are currently using to render your scene in Unity.

In general however I feel like the problem of figuring out in which direction to draw the hatching lines is a very topical one in the relatively niche world of hatching shaders. And on that level the shader is definitely a success.

## CONCLUSION

To form a conclusion we should look back at what we were researching as a research question. We split these up in three categories.

The first one of these was focused primarily on how to figure out the directions of the hatch and how to store and display those.

The direction of the lines can be store as a vector that is basically pointing in the direction around which the lines should form 'rings'. And since we are using a shader that calculates which part of the mesh is supposed to be black or white through mathematical equations, we have eliminated the occurrence of seams if the model is prepared correctly. Since the shader can access different vertex values from the vertices and very few of them are used, we can use the vertex color to store the direction of the hatching lines because a color has 4 values (R, G, B, A) and a direction only needs three (X, Y, Z). The alpha channel from the color is an extra that could later be used to store something extra, like inverting the colors or a light value multiplier for example.

The directions can also be extracted from a rig, that is used to rig the object. If this object has been rigged nicely and with enough detail the vertex colors will transition smoothly, which means that the hatching lines will too.

In general this is a very good technique to create a contour hatching shader. If you are trying to create a different style of hatch like, a patch hatch or a cross hatch you are probably better off using a different shader like one of the currently existing ones that use textures to overlay on top of each other.

The second category is about lighting and movement in the shader.

The shader reacts to directional light like any other would. If the light source moves or the object moves the shader looks nice and the transitions of the shadows, or the movement of the lines happens smoothly and looks natural. They move how you would expect a hatch to move if a 2D hatched drawing were to be animated.

One of the problems at this point in development of the shader is however if the object the shader is on changes shape. This could happen through the movement of the bones since the more complex models will be rigged anyways. Or having a model that can change shape in another way. The stored directions will be the direction of the original bone instead of the current direction of the bone which means that if a finger points up, and is then moved to point sideways, the hatch will still think it points upward. If the entire model is rotated however the hatch will still work since the lines are calculated locally.

The final category contains the outlines.

The outlines completely change the way the shader looks, and for some styles this is definitely necessary (Like in the style of the McBess drawings) While the shader can still stand on its own without the outlines for more realistic styles. The drawback of these outlines is however that they are relatively annoying to install. They must be added to the render pipeline used by unity directly and are not just some code to add in your shader since they are a screen space effect.

If we look at the shader in a more general sense it is not perfect, but it definitely carries its own weight in the world of hatching shaders. Which I will consider a success.

## FUTURE WORK

If I decide to continue work on this shader, which is definitely possible, I could still add a couple of different things.

The first one I think about is making it possible for the shapes to morph, and having the hatch still follow the objects nicely. This would make it possible to make animations with this shader. Right now you would get a very weird effect if you made objects change shape.

Another thing is that I might add some more details to the lighting of the shader. Right now I have implemented a single directional light and a rim light effect. I could make it so more directional lights can be considered, or making it so the intensity of the light matters. I could add specular lighting… The list goes on.

Since the lines drawn by this shader are now very “perfect”, the shader loses the hand drawn feeling of the actual cross hatching technique. If I could find a way to make the lines more random, it might have a better effect.

I could add something to the extra data slot I have left in the alpha channel of the vertex color. I could make it so there is a light level multiplier for if I want materials with a gradient, or I could make it a toggle that tells the shader that the colors should be inverted…

I could try to add different hatching styles, like patch hatching or I could add a secondary array of lines to create a cross hatch. These ideas are however lower on the priority list because these can already be achieved by other shaders.

As you can see there is a lot of opportunity for expansion on this project. Of course these types of projects are never finished, but it is easy to find expansions for this particular one.

## BIBLIOGRAPHY

Cross hatching basics - https://thepostmansknock.com/the-beginners-guide-to-crosshatching/#:~:text=Crosshatching%20is%20a%20technique%20that,and%20the%20right%20is%20red.

Types of hatching (ink) - https://johnmuirlaws.com/hatching-and-crosshatching-technique/

Types of hatching (ink) - https://www.craftsy.com/post/hatching-and-cross-hatching/

Hatching techniques (ink) - https://thevirtualinstructor.com/hatchingcrosshatching.html

Hatching for beginners (Pencil) - https://www.youtube.com/watch?v=RmV5BcDpkpk

Types of hatching (ink) - https://www.craftsy.com/post/hatching-and-cross-hatching/

Shadow hatch study - https://www.researchgate.net/publication/239761619_Learning_Hatching_for_Pen-and-Ink_Illustration_of_Surfaces

Why hatching - https://www.britannica.com/art/hatching-drawing-technique

Etching - https://en.wikipedia.org/wiki/Etching

Old Master Prints - https://en.wikipedia.org/wiki/Old_master_print

Shader Basics - https://www.youtube.com/watch?v=kfM-yu0iQBk

Hatching shader: https://www.artstation.com/artwork/yJqYq3

Procedural hatching/manga shader - https://www.youtube.com/watch?v=2ZR5XIjBmho

Hatching shader in blender - https://www.youtube.com/watch?v=508pwYME-w4

Hatching shader in unreal - https://www.youtube.com/watch?v=18U50KPdD2A

Rigging tutorial - https://www.youtube.com/watch?v=EVBseo4YLa4

Skinning geometry and how it is stored - https://sites.google.com/site/artemscode/asset-pipeline/15-skinned-geometry

Understanding skinning - https://www.pluralsight.com/blog/film-games/understanding-skinning-vital-step-rigging-project#:~:text=What%20is%20Skinning%3F,model%20and%20move%20them%20accordingly.

Blender scripting basics - https://docs.blender.org/api/current/info_quickstart.html

Accessing al vertices in Blender via Python scripting - https://blender.stackexchange.com/questions/909/how-can-i-set-and-get-the-vertex-color-property#comment10613_911

Blender Loop indices vs Unique indices - https://blender.stackexchange.com/questions/119599/vertices-loop-indices-vs-unique-indices

Math functions in Python - https://www.geeksforgeeks.org/python-math-function-sqrt/

Outline Shader - https://alexanderameye.github.io/notes/edge-detection-outlines/

Outline Shader - https://roystan.net/articles/outline-shader/

Toon Outline Shader - https://www.youtube.com/watch?v=fW-5srSHDMc

Unity Reference - https://docs.unity3d.com/Manual/

## IMAGE LIST