3D Postprocessing Shaders for Rendering Buildings

by Leo Traub

Interdisciplinary Project
Lehrstuhl Architekturinformatik
TUM

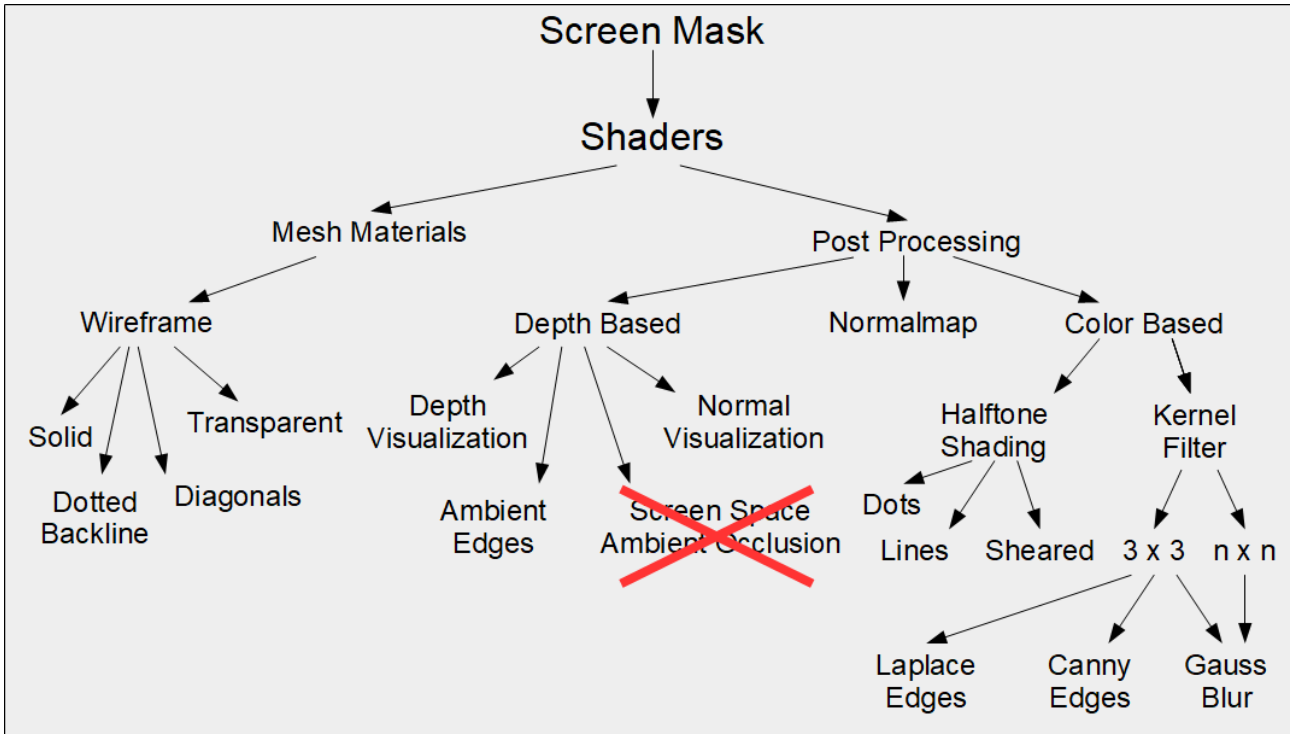
Structure

1. Description

The goal of my interdisciplinary project (IDP) was to visualize the geometry of buildings using 3D shaders and postprocessing techniques. In architectural visualization it is important to highlight the edges of the 3D geometry, so that the shape can be seen clearly. The realtime rendering engine that was used is Unity3D. In a future project the post processing shaders might be translated to OpenGL, so that they can be integrated with the CDP.

2. Implementation

Most of the shaders, implemented in my project, are post porcessing shaders. This means, that they are applied after the 3D view of the virtual camera has already been rendered into a texture, which is then modified by the post processing effect. Some of these shaders only make use of the color channel, while others are using additional textures provided by the camera. These are the depth texture and the depth normals texture, which are very useful as we can extract geometric informations from them.
For the desired wireframe effect, it was also necessary to implement a 3D shader for meshes, which has to be set to every mesh separately and is therefore harder to apply to an entire scene. But of course there are also some advantages of using mesh shaders or even combining them with post processing.
Another way to highlight certain parts of a building is, to only render that part with a special effect, while the rest is rendered without any effect. Therefore a mask is drawn on the entire screen before the shaders are calculated, which selectively applies the effects.



2.1. Post Processing

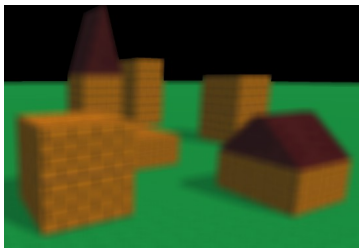
Post Processing refers to any kind of image effects, that can be applied to an image after rendering. For the project I have only investigated effects, that are useful to visualize buildings in the context of architecture.
In Unity you write the effect as a shader, so that the computations are performed on the GPU, which is more efficient. Many effects are using pixel distances on the screen in theory, but in a Unity shader, you are working with UV coordinates of the screen rect, so many of the shaders need to be used with two values, in order to remap the UV coordinates to pixel distances: The Aspect Ratio, which is 1.78 for 16:9 and the distance between two pixels in UV space.

2.1.1 Kernel Filter

One of the most common ways to manipulate an image, is applying a filter kernel to the image. A filter kernel is a NxN matrix of numbers, often 3x3. Beginning with the bottom left pixel, each pixel of the resulting image is calclulated using the color values of the source image. The pixel that has to be calculated is treated as the center pixel in a way, that each value of the filter kernel is multiplied by the color value of the pixel with the same offset to the center pixel like in the matrix.
The kernel filter is then moved all across the source image to calculate the result.

In theory this sounds rather mathematical but if you look at the example of a simple box blur filter:

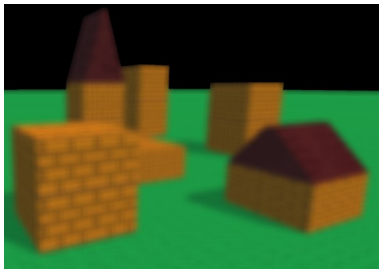
$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$



To blur the image, every pixel of the result is composed of the surrounding pixels. The value is calculated by simply summing up the surrounding color values (each multiplied by 1), and later dividing the result by 9, in order to normalize the result again.

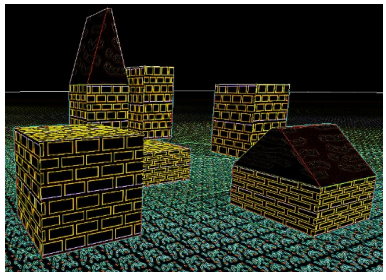
Using this approach, several rather basic effects can be implemented only by setting up the matrix with the right values. Because of this reason the concept is so powerful. To get a smoother blur effect, you can use a gaussian filter kernel. If you scale the dimensions of the matrix to for example 5x5, then the blur effect does use an even bigger area around a pixel and therefore looks better:

$$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

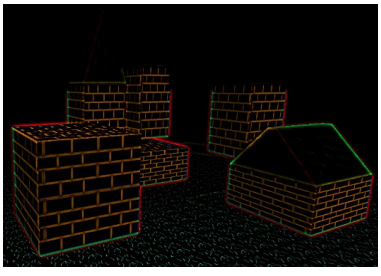


In our case we are using an edge detection filter known as the Laplace filter, which can be implemented as a 3x3 or 5x5 matrix:

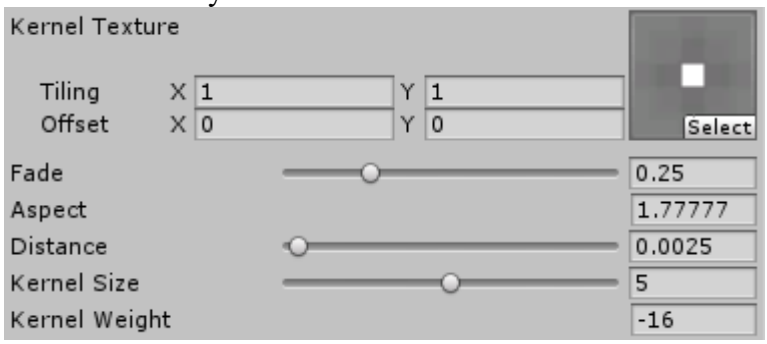
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$



$$\begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix}$$



In Unity3D we can not assign an arbitrary matrix to a shader, but for a 3x3 matrix, we can simply assign three vectors to a shader, that are then interpreted as a matrix. For a 5x5 or even larger matrix, it was necessary to get tricky. I came up with a way, where you can assign a kernel texture to the shader, that is then interpreted as a matrix. Each pixel value of the kernel texture is a greyscale representation of a number, so the values are between zero and one. The greyscale value of 0.5 is then interpreted as a matrix value of zero and you have to set a multiplier to the entire matrix to reach the desired values of your filter.



Setup for 5x5 Laplace Shader with kernel texture

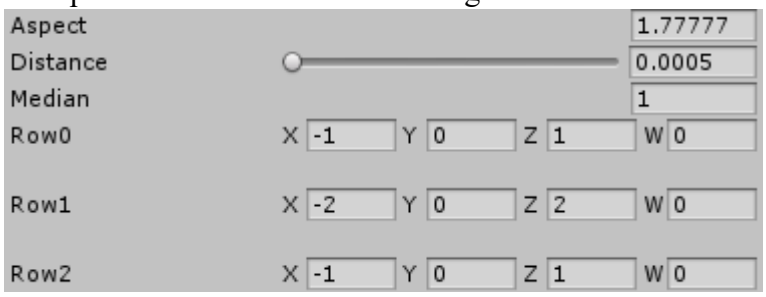
The minimum import size of a texture in Unity is 32x32, and if you try to import smaller textures (for example for a 5x5 kernel), the sampling does not work correctly. Therefore you should scale an image up to 32x32, after you have setup the pixels for the matrix. Still the pixel values are not always precise after sampling. There is a finetuning parameter for the kernel to counter this effect.

Also you have to manually set the size of the matrix, so that the shader knows, where to sample the matrix values from the texture. You have to be careful with this value, as it exponentially increases the number of texture samples, that have to be performed by the GPU.

As a result the 3x3 edge detection is very sensitive to thin edges and also amplifies noise, while the 5x5 edge filter can also find thick edges and weighs the amplitude by the strength of an edge. What is also nice, is that for Gauss Blur the mathematical distribution is a continuous function, that can be baked into a smoothed texture. This way you can use the same continuous texture independently from the matrix size, that you want to blur with.

2.1.2 Canny Edges

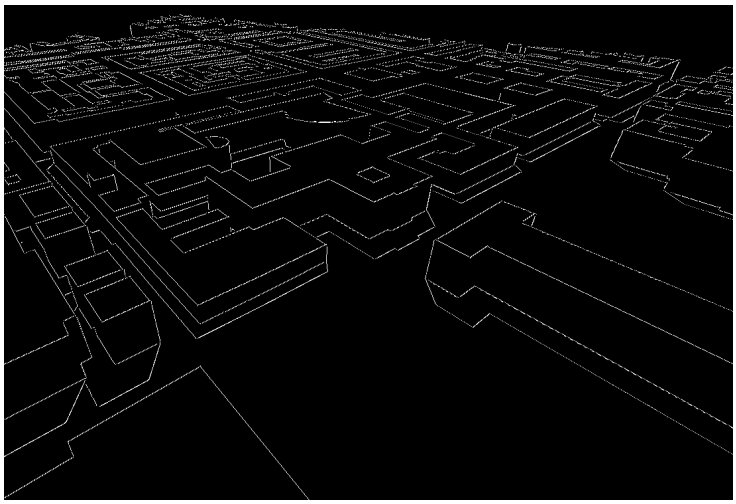
The Canny Edge Detection algorithm is a very powerful way to visualize edges, as it selectively only uses the strongest edges and reduces them to a thickness of one pixel. It makes use of two filter kernels called the Sobel-Operators: One for horizontal edges and one for vertical.



Horizontal Sobel Operator Setup in the 3x3 Kernel Shader

From these two textures the direction of an edge can be estimated at any point on the screen. If you have the direction of an edge, you can perform an action called the Non-Maximum Suppression, which eliminates any pixel on the edge, that does not have the maximum value. This action gives every edge the same thickness.

To give the outcome an even more consistent look, a Hysteresis Treshold operation is performed at the end of the algorithm. That way any edge value beneath the treshold is set to zero, while any value above the treshold is set to one.

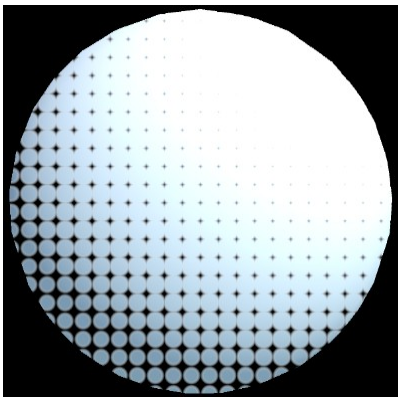


In the end you have very sharp and clear edges, which can be combined with other shaders or rendered on top of them very well. Also the shader can be calibrated so sensitive, that it even finds the edges of shadows, which can also give the result a unique look.

2.1.3 Halftone Shading

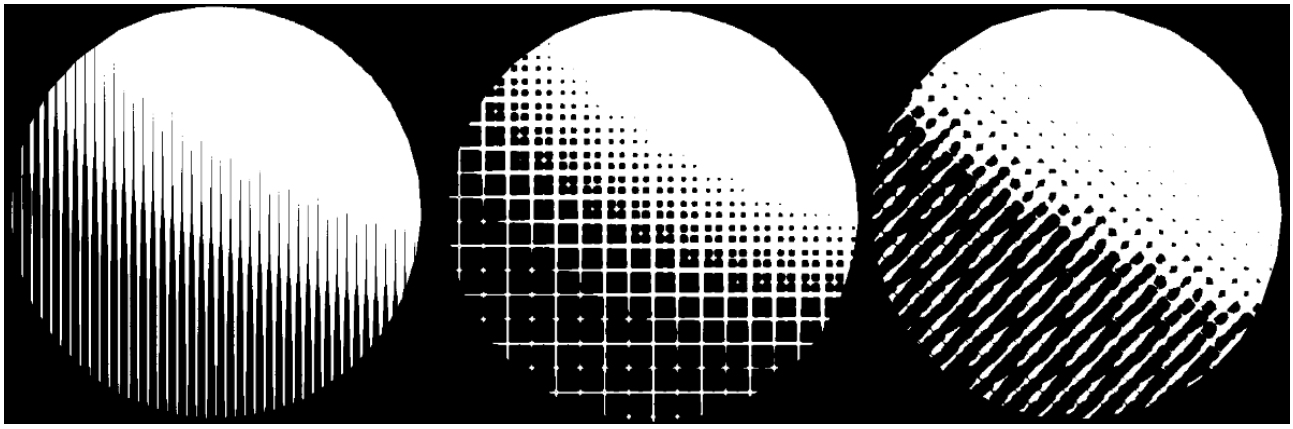
Halftone Shading is a more artistic way to display a scene. The main reason for implementing this shader in an architectural context was to make a 3D model look like it has been drawn by hand. In the end an algorithm can never replace a good old hand drawing, but there are still some advantages.

The simplest approach of Halftone Shading is the Dotted Halftone Shading. For this approach you slice the screen into small regions of for example 10x10 pixels, and for each of the reasons you calculate the average greyscale value. The you draw a circle inside this region of the screen, with a radius that depends on the greyscale value.

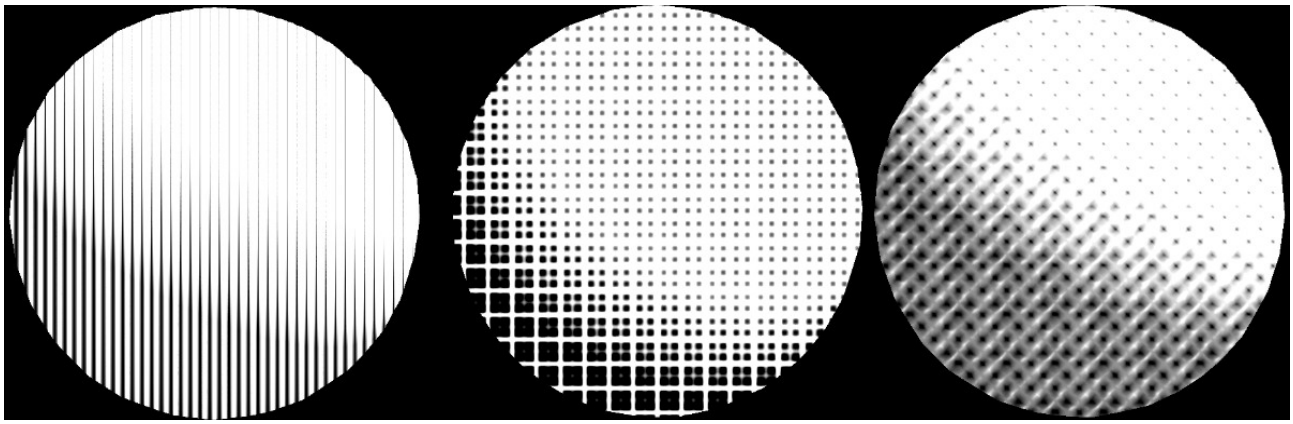


Circles are easy to calculate mathematically, but if you want to use other shapes for the shading, you quickly need the possibility to dynamically assign textures to the shader, in order to experiment and find out, what looks best.
The chosen texture defines a treshold and is tiled accross the screen. For every pixel of the source image the algorithm evaluates wether the greyscale value of the screen is above or below the greyscale value of the tiled texture. This way a pixel of the result can either be black or white in

theory, but for my approach I tried out two different approaches: One where actual tresholds are used and one where the treshold defines a soft border , so that the shading does not look so noisy, but smoother.



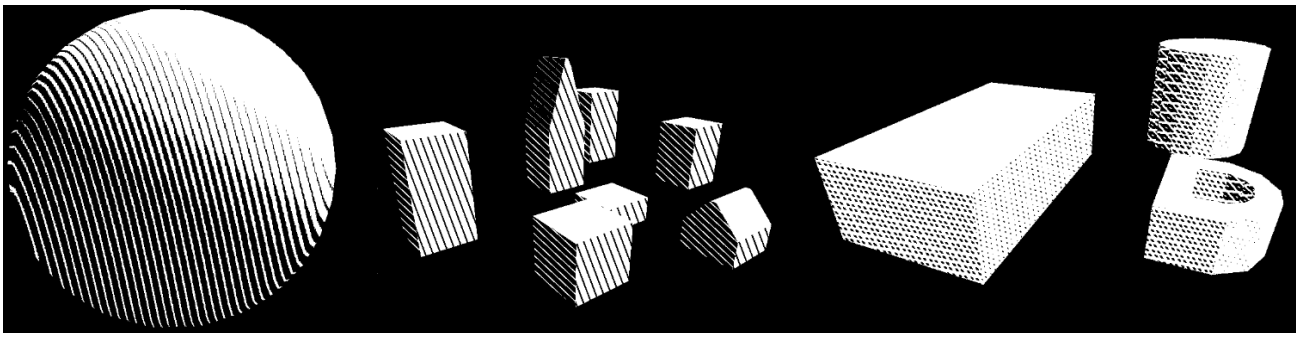
Treshold Halftone Shading



Dynamic Halftone Shader with smooth borders

The outcomes already look pretty nice on single frames, but as soon as motion comes into play, the effect looks weird. One major drawdack is, that you can see the tiles of the texture all across the screen, which is even more obvious, when you move the camera, and the white pixels stay in the same place.

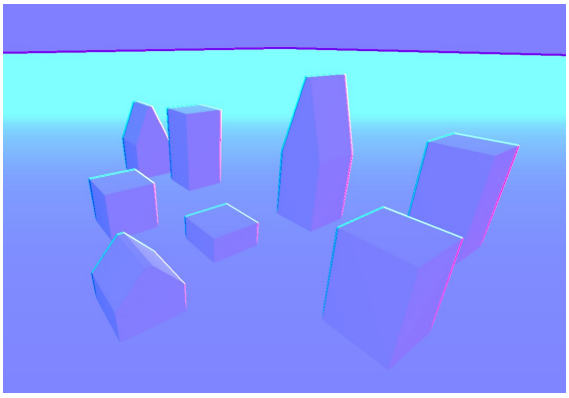
The last aspect of the shader was optimized for the visualization of 3D geometry: The shearing factor. The idea is simple: The texture is sheared (meaning the UV corrdinates are manipulated) depending on the screen space normals. This way you can not only hide the tiling effect, but you can also see the different normal directions of walls better and therefore the edge between those walls.



For Virtual Reality another version of the shader was implemented, that is not a post processing effect but a mesh material, that bakes the halftone effect directly onto a mesh (see 3.3.2).

2.1.4 Normalmap

The Normalmap Shader was some kind of lucky coincidence. Actually I wanted to implement a shader, that computes the normals from the depthmap of the current view. At that point I did not know, that you can simply access the screen space normals without having to compute them. In the end I implemented a shader, that generates a bumpmap of the current view at runtime. A bumpmap can also be called a normalmap, but it is in fact not a map of the normals.

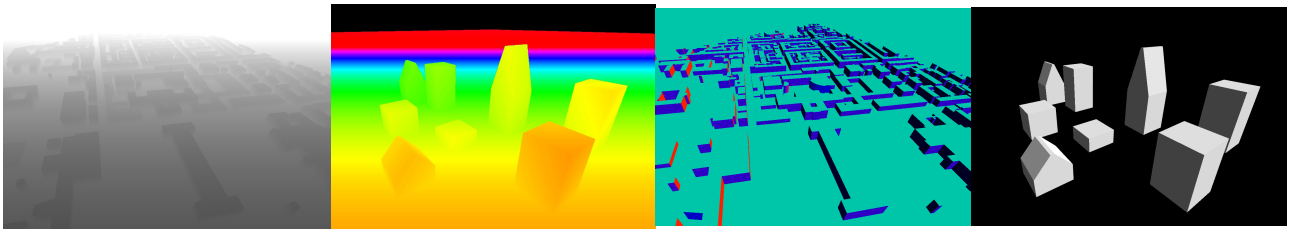


Normalmap generated from view

This shader is not very useful as a post processing effect, but could be really useful for a mesh shader, where you directly want to compute the normalmap from the texture, to enhance your lighting computations.

2.1.5 Camera Depth

In Unity you can directly access the depth map from any camera, that is currently rendering. The depth map stores informations about how far an object is away from the camera, for each pixel. The values are very precise floats ranging from zero to one, but when directly mapped to a greyscale image, the data is compressed to 256 different depth levels. You can also map the values to a color gradient, like for example a rainbow texture. This way you don't lose so much information. Another dataset you can directly access in Unity, are the depth normals. These are simply the normals of any surface in view space. As you typically get a very colorful and bright image from the depth normals, it might be useful to map them to greyscale values.

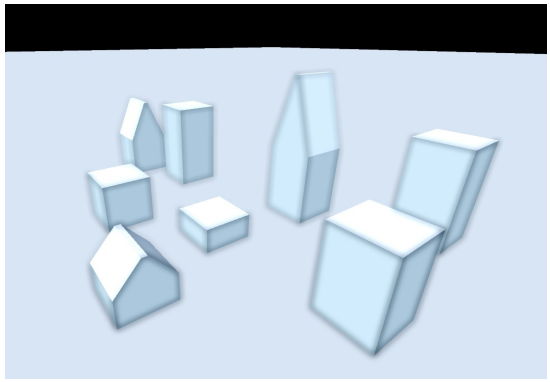


Depth greyscale (1), Depth gradient (2), Depth Normals raw (3), Depth Normals greyscale (4)

Both textures, the depth map and the depth normal map, are not very good to visualize architecture, if you use them raw, but they can both be used very well in combination with other effects. Especially the depth normals are providing a lot of informations about the geometry and edges, which we want to visualize in architectural renderings.

2.1.6 Ambient Occlusion

Ambient Occlusion is a shadow like effect, that occurs, when two surfaces are close to each other, because light can not shine so bright into narrow spaces. A variant of the effect called Screen Space Ambient Occlusion (SSAO) can be implemented using the depth data and the screen space normals and is often used to simply make scenes in video games more realistic, as it is very efficient compared to other effects.
Unity already provides a SSAO effect as a part of their post processing stack, which is sadly not customizable at all, as you don't really get any insights on how the effect is exactly implemented. When trying to implement the effect on my own, I ran into a lot of problems, which simply made it impossible to get the desired results. The problems were related to transformations from linear eye space to screen space or to world space. There was simply no setup, that could satisfy the constraints for SSAO.
In the end I worked out some effect, that looks a bit like SSAO and also visualizes edges, but not in a very realistic way. For real Ambient Occlusion the shadows should only be drawn between surfaces, that are forming a concave structure and not on convex edges. In my approach, the algorithm does not distinguish between those edges and simply adds shadows to any kind of edge in the scene. I called it Ambient Edges.



If you want real Ambient Occlusion in Unity, you should probably just use the provided effect, as it is not worth going through all the work, that is need to get similar results.

2.2. Mesh Materials

Post processing is not the only way to highlight the geometry of a building. In fact you could think of a a lot of different approaches to interesting shaders. As the focus of my IDP was to work with post processing effects, I did not dive very deep into these topics, but there was one effect, that is definitely needed in architecture and that can simply not be achieved with post processing. The wireframe representation of a 3D mesh is very useful, when it comes to visualizing buildings. Also there are some post processing shaders, that work better with the right materials on the meshes, before applying the effects. Therefore it was also possible to come up with shader solutions.

2.2.1 Default Shading

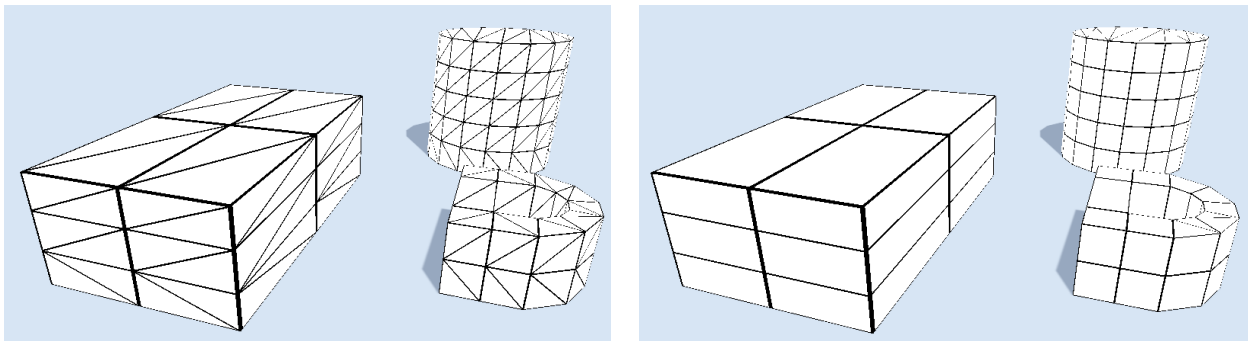
It was necessary to write a very simple default shader for meshes, as the Unity default shader is creating some weird artifacts, of you look at a surface from a close angle. As a result of error propagation, these artifacts were causing a lot of noise for the Laplace or Canny Edge Detection.

Although it is necessary to calculate the lighting on a mesh for a lot of post processing shaders, it can be rather bad, if you have advanced effects like specularities on a mesh. Therefore I implemented the most simple mesh shader you could think off, so that it works well with all the different post processing techniques. Also you should sometimes turn off "receive shadows" in the MeshRenderer Component, in order to prevent self shadowing artifacts.

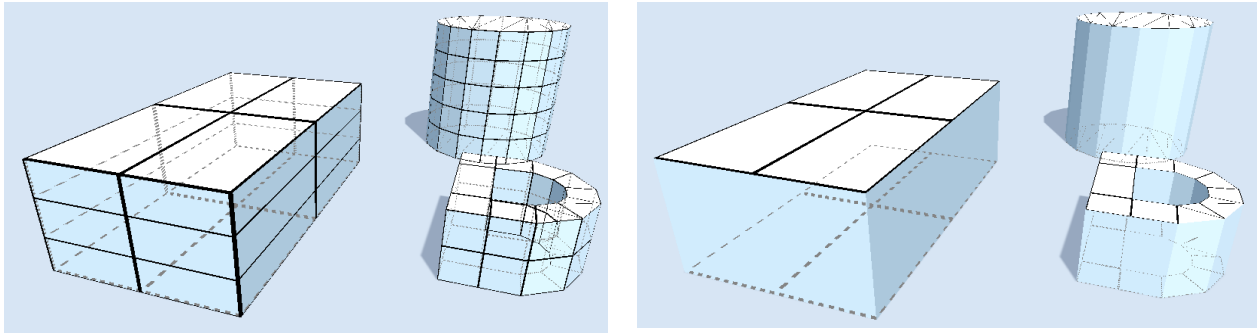
2.2.2 Wireframe

Wireframe shading is a well known approach to showing the geometry of a 3D Mesh. The simple idea is, that the edges of the mesh are drawn with another color than the faces. This way you directly see the triangles, that any three dimensional shape is composed of. You can also make the faces transparent, so that you can see the edges on the back of the mesh and you can give those edges a different color or make them dotted in order to clearly keep them apart from the front edges. In the end, you get a very invormative representation of the geometry, that you want to visualize. Of course this effect is not used for realistic renderings, but for precise sketches or renderings you want to work with.
One problem of the effect is, that it shows every edge between triangles, and not only the edges between quads. The drawback is, that you often have too many edges, that make it more difficult to interpret the image. Luckily it is possible to switch off the diagonal edges of quads, if the mesh is stored in the right way and the polygons are defined as quads.
Another interesting option is a filter, that discards all vertical edges from the mesh, so that the wires are only drawn on horizontal faces in the end. For architecture this can be used in order to only highlight the outlines of a building.
In the end I came up with a wireframe shader, that combines all the described options.

In order to work in combination with the post processing effects, I have set the scene up in a way, that every mesh has a duplicate with the wireframe material applied. The wireframe effect camera then only renders these objects filtered by object layers.



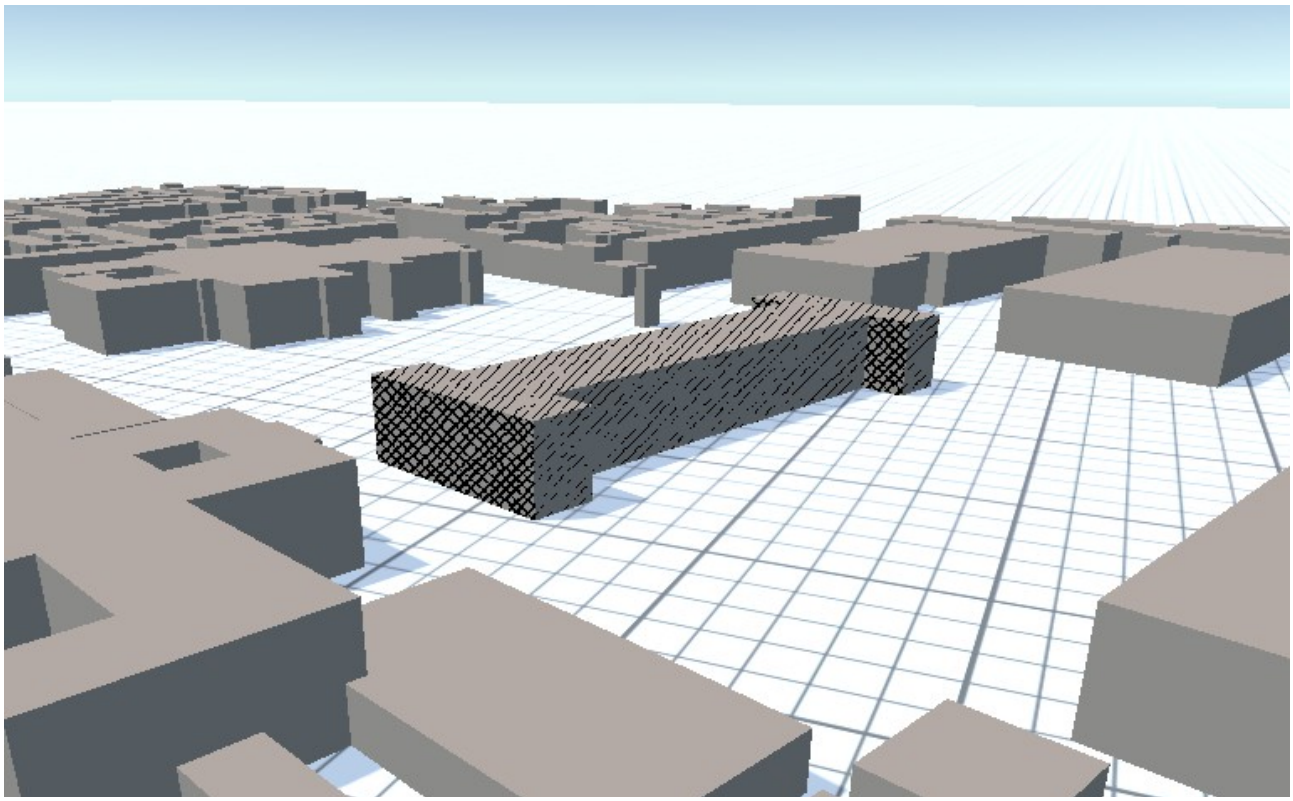
Wireframe with diagonals (left), without diagonals (right)



Wireframe transparent with dotted backline (left), only horizontal faces (right)

2.3. Screen Mask

In the context of the CDP you often want to highlight one building in between a lot of other buildings. Therefore it is very useful to only apply the desired post processing effect to that single building.
While it is very simple to exchange the mesh material of a single object, it is rather diffcult to only render a certain screen area with a post processing shader, as post processing typically just uses the entire screen texture, that is rendered by a camera and does not distinguish between objects in the scene.
In my approach I am using a setup, where you have two cameras, that are rendering the scene: One camera simply renders everything without any effects while the other camera only renders the desired object, in order to a apply a shader to it. The rest of the second rendertexture is transparent, so that you can simply place it on top of the default view. It somehow works like a mask, that is applied to the entire screen.
You can control the mask, by changing the layer of an object in Unity. The second camera only renders that layer. The rendertextures are then layered in the UI using rawimage components. There is a little drawback to this approach, because you have to set the size of the rendertextures to your screen size before you can use it (1366x768 at the moment).
You could probably also write a post processing shader that directly combines the textures, without having to use UI components, but that was not so simple to do in a dynamically exchangeable way.



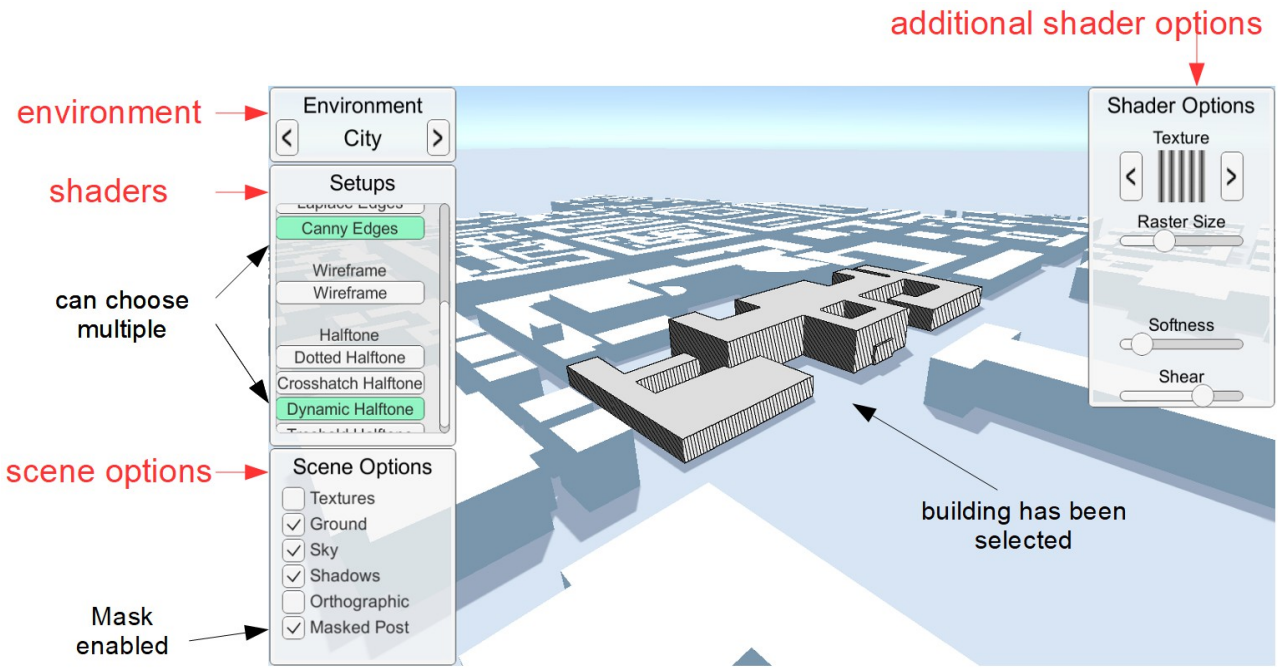
Single building maksed with crosshatch halftone

3. Testing

For testing the shaders, I have built a dynamic demo scene, where you can switch between all of the different shaders and have a lot of different environmental and scene options.
Also some shaders turned out to work even better in combination with other shaders, in order to highlight certain aspects of 3D meshes better.
In Virtual Reality some of the shaders behave differently, as Post Processing isn't as easy in VR.

3.1. Demo Scene

The demo scene consists of four different menus: One to choose the current environment, one for the desired shader, one with scene options and for some shaders, you are also provided with shader specific options. You can navigate and change your view by using keyboard and mouse input.



Demo Scene (City view tih treshold halftone shading)

For the environments, you can choose between different kinds of buildings with different degress of detail. The default scene is a city view with the coarse geometry of the district around the TUM building in Munich. This view was generated from open street data and represents, what the view in the CDP also usually looks like. Then there is a scene with a detailed 3D model of a building and one scene with a default tree by Unity. These two scenes should demonstrate how the shaders behave in rather realistic environments. And there are also some scenes with very undetailed buildings for testing cases, where for example one building is round and another one has a hole (for wireframe testing). The last scene simply contains a sphere which can be used to test different halftone shading options.

For the Shaders you can choose between all the options described in 2. In the desktop scene You can select multiple options, that are layered on top of each other, so that the last clicked effect is on top of all the others.

For the scene options, you can switch off things like the ground, shadows, the sky, or the textures of the models inside the scene. Some shaders simply work best with the right setup here, as the

shadows are sometimes generating misleading informations or the sky gives a better contrast if it is simply black. There is also an option to set the camera view to orthographic and the option to use a post processing mask like described in 2.3. To do so, simply click on an object, while the option is enabled. That changes the layer of the selected object.

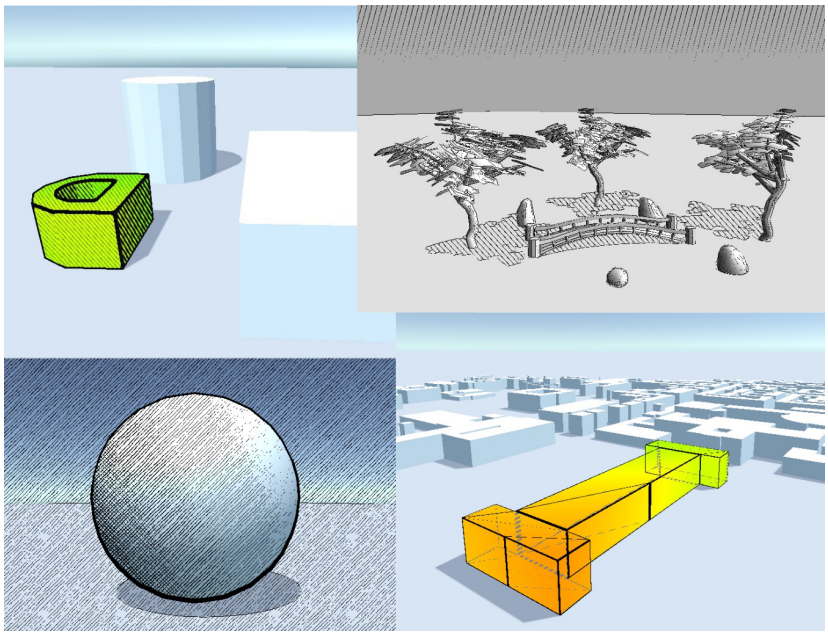
There are shader options for three different shaders at the moment: The wireframe shader, the dynamic halftone shader and the treshold halftone shader. Of course there are a lot of different options for the other shaders too, but most of these options don't have to be changed at runtime. Something like the texture or the raster size of a halftone shader is just very practical to be used at runtime, where you directly see the impact of different options. Also some values need to be calibrated in order to get the perfect screenshot.

3.2. Combinations

In order to dynamically combine any of the shaders with another, I came up with a system, that allows you to layer the different post processing effects like in photoshop. Each effect camera renders its image separately and the processed results are then layered on the UI. For this approach every shaders needs to support transparency to some extend. For example for the Canny edge detection filter it is important, that only the edges have full opacity, so that you can layer this effect on top of the scene.
With these properties you can differentiate between two classes of effects: Effects that can serve as a base layer, and effects, that can be layered on top of this base layer.
The base layer can be a layer with no effect at all, the depth view or the depth normals view. On top of these three you can render anything else, as the other ones all support transparency.

Another advantage of my approach is, that the masking option as described in 2.3 still works for any combination, so this is not making things any more complicated.

A drawback is, that the combinations don't work in Virtual Reality, as you don't have any screen space UI in VR. It is still possible to hardcode any desired combination in a dedicated shader, if needed. The desktop scene is a perfect playground to play around with combinations, so that the right one can be found for VR.



some example combinations

3.3. Virtual Reality

For Virtual Reality I simply duplicated the demo scene together with its UI. There is no implementation for VR controllers. You can only switch between options on the desktop UI. Some of the features had to be switched off. For example you cannot use an orthogonal view in VR, as that would simply not make any sense. You can not navigate with the keyboard any more, but simply by walking around. As an alternate feature you can shrink and expand the scene around you using the +/- keys on the keyboard, as you can explore the architecture from different angles this way.

3.3.1 Disadvantages

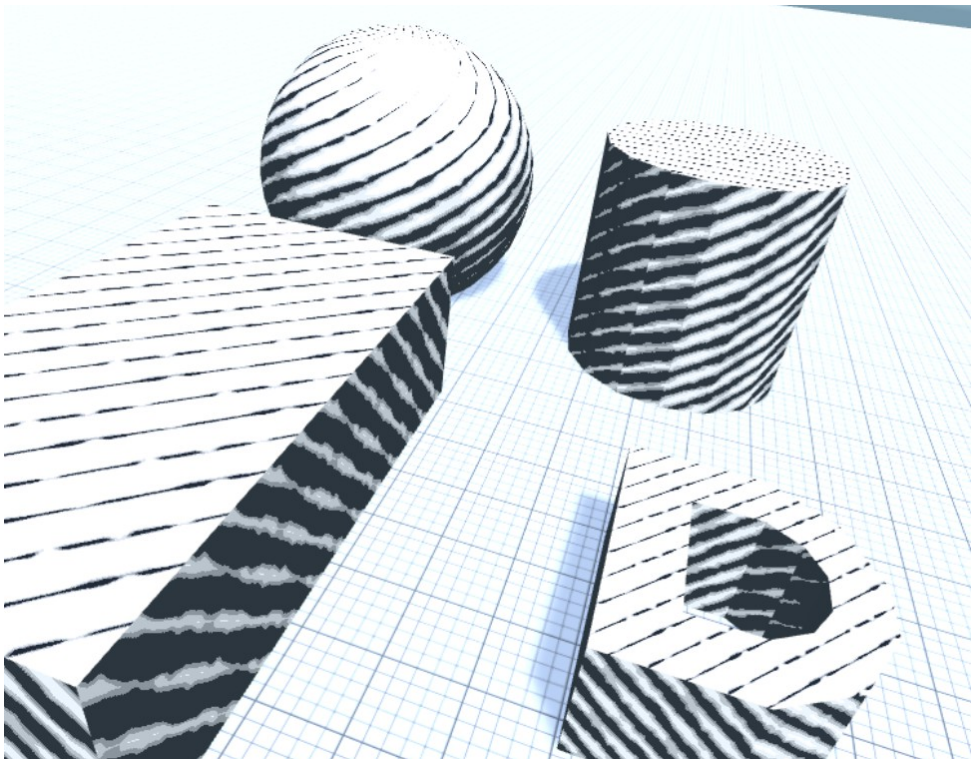
When testing the shaders in VR it quickly became clear that some approaches won't work. Any kind of halftone shading is a total mess in VR, because you have the static halftone grid directly in front of your eyes. There is no screen space UI in VR, so my approach of layering shaders doesn't work in VR and also there is no chance, that the post processing mask would work the way, it is implemented right now.
On top of all these drawbacks another problem became clear: In Unity you generally don't have any Anti-Aliasing (AA) after a post processing effect is applied, which isn't any problem, if you are seeing the effect on a display with high resolution. But typically the resolution of the display inside a VR headset isn't as high. Therefore you can clearly see the pixel differences in front of your eyes, if there is no AA, which gets really disturbing after some minutes in the VR world. A solution would be, to write an own AA algorithm as a post processing effect. There is currently a new approach of AA called FXAA, that would have the right properties for this solution, but after some investigations, it turned out that FXAA is a really long and complicated algorithm, that cannot simply be described or implemented. I also tried importing external versions of the algorithm from the internet, but none of them worked for VR.
For some of the post processing shaders I came up with a quick workaround. If you use a simple 3x3 Gauss blur, it also makes the pixel differences smoother, which can be interpreted as some kind of AA. In fact this approach only works in a really small dose, as it also makes everything blurry, and does not only smooth the edges, that need to be smoother.

3.3.2 Advantages

With all these disadvantages also comes one great advantage: None of the problems described above exist for the Wireframe shader, as it is not a post processing effect, but a material shader for meshes. You can assign the shader to a single mesh, which is equivalent to the masking feature of my post processing effects, and you have AA for the Wireframe shader, as it is not necessary to render it with post processing.
The effect even benefits from the usage in VR, as it usually gets very messy, when you use the transparent mode with dotted backfaces turned on. There are simply too many lines. But when having two separate eye views while seeing the effect in VR, you can clearly see which line belongs to which building, as you have some natural eye depth then.

The problem with the halftone shaders not working at all, was also solved by implementing another version of halftone shading, that is not applied as a post processing effect but as a material effect. Similar to the wireframe shader, you can assign the halftone material to whatever mesh you like. The effect is basically interrupting the lighting pass of a usual mesh shader and applying the halftone effect based on the light calculations, so that the darker shading is seen where an object is

in shadow. As it is not a post processing effects, most of the described disadvantages are irrelevant, like for the wireframe shader. The rastered texture, that is used for the halftone, is now baked directly onto the mesh and does not depend on the screen coordinates, which makes it far easier to look at the effect in VR. The hard edges of the halftone shader are now rendered with AA and also more convenient to watch, but to make it look even smoother, there is an option to render the shadows with a rather smooth step gradient.



Halftone mesh shader using the lighting pass for VR

4. Evaluation

The project was challenging at some points, because writing shaders (especially post-processing shaders) is not easy to learn and very hard to master. In architecture you mostly need to visualize the edges of your geometry but also the curvature needs to be presented intuitively.
The implementation proves that it is not only possible to set up a very specialized combination of shaders for a certain scenario, but that it is also possible to arbitrarily combine any kinds of shaders, as long as the application is running on a 2D display. As soon as you enter the VR world, it turns out, that it is much harder to work with post-processing effects. In Unity you can not really use your own post-processing effects in VR, but should rather stick to the ones provided by the Engine. Of course it would be possible to reach that point of beauty and effectiveness, too, but you would need to spend a lot of time until you actually get there.
Personally I have already learned a lot about shaders in this IDP and I am rather proud of the results. One good thing about shaders is, that they are not only bound to one application, like in this case architecture, but they can also be used in a lot of different scenarios. As well as I might use them in future projects, I hope the CDP will also benefit from my work and the shaders will enhance the representation of buildings.