SSAO and Paint Mapping, an EDAN35 project

Mattias Gustafsson[1] Markus Olsson[2]
Lund University
Sweden

Abstract

In this paper we discuss the implementation of two effects; screen space ambient occlusion and screen space paint mapping. We will see that the effects can look great but note that there are limitations to both. The ambient occlusion effect is based on a common version while the paint effect was implemented without preexisting guidelines.

1 Introduction

For the project two different screen space effects were created; ambient occlusion (SSAO) and a custom effect we will refer to as paint mapping (or SSPM). We intended to first make the SSAO effect and draw it without any textures to highlight the shadows created from the effect, then add colored paint splats on the white surfaces.

In the following sections we will discuss the algorithms used for each effect, the results as well as the problems we ran into and the solutions we found.

2 Algorithms and applications

Below is an explanation of how each effect was created. More detail as to why some choices were made will be given in section 4.

2.1 Screen space ambient occlusion

SSAO is a commonly used effect in real-time graphics. It was first implemented by Crytek for the game Crysis. Several versions exist today[3].

It works by saving all the depth values of the scene in a buffer. Then computing the occlusion of each pixel by transforming it back into world space and comparing the depth of the pixel with the depth of a few nearby points. The nearby points can be chosen in a few different ways. The original Crytek solution sampled points from a sphere around the pixel. Our solution used a hemisphere oriented along the normal of the surface. The sample points from the hemisphere are chosen randomly at startup and the hemisphere is then rotated randomly at each pixel using a noise texture. A point is occluded if the depth in the buffer is lower than the depth at the point and a range check is passed. This range check makes sure the difference between the depths is not too great and is there to prevent an object from occluding other objects too far behind it. The occlusion of the pixel is calculated as an average of the occlusion of all the sample points. The occlusion is then blurred to reduce noise. The blur was created by averaging together nearby pixels in the occlusion buffer.

2.2 Screen space paint mapping

Paint mapping appears in numerous games in varying forms. Some allow the player to paint on surfaces (for example Portal 2, Splatoon), while others use it for decals and clutter to get more varied environments. A common way to create these effects is to UV-map the geometry to a paint map (similarly to how pre-generated lighting is done where a light map is used). However, for reasons explained in the discussion section this is not how our implementation worked.

Because we use a screen space effect we need to store the paint data separately. This is done with an array of points and vectors as each paint shot fired by the user is represented by the position of the user at that time and the direction of the shot. The color of the paint is stored as well.

The effect is created through several render passes through two shaders. The first shader used calculates what parts of the screen are colored by a paint shot. This shader is run once per paint shot. The second shader then takes the image created by the first and applies a filter to create the “splat” shapes. The resulting image is combined with the ambient occlusion and rendered to the screen.

2.2.a Paint Calculation

To decide if a pixel should be colored, we transform it into world space coordinates and compares its distance to the paint shot vector is less than a paint range uniform. If it is in range it is colored in the paint color and the alpha is set to one over the distance from the vector. We also do a check to see if the dot product between the paint vector and the vector between the pixel and firing position is not smaller than 0. Otherwise the pixel is in the other direction compared to the shot and should not be affected. We use a custom blend mode to mix the new color with any existing color from previous shots without overriding it (normal blend for color, additive blend for alpha).

2.2.b Splat Filtering

This step was created to give the color marks a more paint-like appearance. The shader compares the alpha value of each pixel on the screen with a texel in a noise texture. If the alpha is above the texel intensity, the pixel

[1] dat13mgu@student.lu.se
[2] dat13mol@student.lu.se
[3] We based our version on the ones explained in http://john-chapman-graphics.blogspot.se/2013/01/ssao-tutorial.html and http://learnopengl.com/#!Advanced-Lighting/SSAO.

is drawn at full alpha, otherwise it is discarded. The pixels are converted to world space coordinates before they are mapped. This is in order to keep the mapping consistent as the user moves around (which changes the screen).

3 Results

Below, we see an example of each effect. In Figure 1 in the appendix, the scene that is rendered is completely white and solely relies on SSAO for distinguishing objects. As can be seen, the effect gives a lot of detail to the scene. In Figure 2 several paint shots have been fired, showcasing the paint splats created with SSPM.

The performance of our SSAO algorithm is good. It was able to maintain 60 frames per second in the Crytek Sponza scene. The cost of SSPM scales with the number of paint shots fired. When using SSAO and SSPM 60 fps could be maintained with up to 40 shots fired on the computers in Uranus and dropping slightly when going beyond 40. On a more powerful machine, 60 fps could be kept effortlessly.

SSAO is a very useful effect, as seen by the number of games that are using it. SSPM however has a few problems, like paint going through walls. An alternative implementation to the effect is desirable if you want to use it in a game. With that said, the effect is useful in that it is very simple and requires little memory resources. That it exists in screen space means it can be added to any scene without modifying it.

4 Discussion

In this section we will go into more detail as to why we made certain choices and what we found by doing so.

4.1 Screen space ambient occlusion

While doing the project a few different variants of SSAO were tried before the final version was reached.

Initially the points were sampled from a sphere and no randomization was used. Sampling from a sphere instead of a hemisphere means that convex surfaces will be brighter than flat surfaces. This is unrealistic because all surfaces are flat if looked at with enough magnification, so light with an angle to the surface normal that is larger than 90 degrees will always be occluded. Not using randomized sample points will cause the occlusion effect to look very banded as surface angles will cause more or less points to be occluded.

In most implementation tutorials of SSAO the range check is implemented by taking the difference in depth and checking if it is greater than a constant value. This is not good since the depth is not scaled linearly, the same difference in depth means different real world distance depending on how close to the camera the objects are. This means that the range check will stop working on objects far away and all occlusion will disappear when the camera gets too close. This was fixed in our implementation by linearizing the depth values before checking the difference.

We also found that if there was no bias it was easy to get a lot of flickering as distances in depth between some points could be incredibly small. On the other hand, a large bias value would drastically reduce the detail level of faraway objects, making them near invisible since we use SSAO as our only way to differentiate objects. As a result, we had to use a smaller bias than one would normally use if diffuse textures were also rendered.

A side effect of using a noise texture for sampling points is the pattern that appears on the screen. To compensate for this, we blur the texture. Normally the blurring is fairly intense as the shadows don’t need to be sharp when textures are applied but, again, without them we found that we had to use less blur or the screen would look too smeary.

4.2 Screen space paint mapping

Right from the start we faced several challenges with the paint effect. The commonly used approach to use a paint map for the entire geometry was no option as we had no good way to map the geometry correctly. We also lacked means to do collision checking which prevented us from placing decals on the hit area (decals would also be tricky when geometry met as we would have to put it on several different objects). Our solution to this issue was to treat the paint as cylinders which would reach far into the geometry. We could see if a pixel was colored by checking if it was inside the cylinder or not. The drawback of this was that we would paint geometry behind objects but we could ignore that as long as we chose not to show it (an alternative method could be to create a texture for each paint shot, similar to how shadows were handled in assignment 2).

Our paint mapping went through several iterations before we decided on the screen space solution. We initially tried to give each object its own paint texture. Because each object has existing UV-coordinates used for its diffuse texture (among others) we thought we could use those to draw to and from a paint texture. However, it turned out several points in the geometry mapped to the same texel which caused our paint marks to cover large areas they should not. For example, the walls repeat their texture over and over but is just one object, the same goes for the archways which again are just a single object with a lot of geometry.

This left us with two choices: to create our own geometry or try a different approach. We tried both but in the end we settled on the screen space effect as it would let us keep the Sponza scene. The new effect turned out to be fairly straightforward and simple to implement, though not without issues.

The biggest issue with our screen space paint mapping is that we need to run the paint shader once per paint shot per frame. This severely limits how much paint we can have at once. We solved this issue by keeping only the 64 latest paint shots, enough to paint a decent amount of geometry without major slowdowns.

Another issue was that new paint circles would override old paint, with alpha enabled this would create ugly white circles where the paint should blend together but would not because we painted over the old paint with new, less opaque paint. The solution to this was to, as stated in the Algorithms section, use a blend mode to mix the values.

We wanted the color to look like paint marks rather than circles which we did by cutting away parts of the color with a noise texture. This turned out to work really well but there was one problem: we couldn't just map a pixel on the screen directly to a texel in the noise texture or the paint would animate when the user moved as the values of the pixels changed. We fixed this by taking the world position of the pixels, getting points which wouldn't move with the camera.

5 Conclusion

SSAO exists in many forms and is widely used in games. It is an effect you often don't realize unless you get to compare the difference in the scene. Our version is perhaps one of the more common implementations that handles most cases well. Sadly, the inherent drawback of screen space processing, that there is only so much information about the environment available, limits what we can do in certain cases where the shadow caster is hidden behind other geometry.

The SSPM effect looks great but the limited paint count and lack of collision checks makes it a poor choice in situations where the environment cannot be designed to hide these flaws. On the other hand, the effect can easily be added to any scene without modifying it which makes it a good tool for testing the effect. This could be useful if a developer is considering to add dynamic paint and wants a quick example of how it could look.

Appendix:

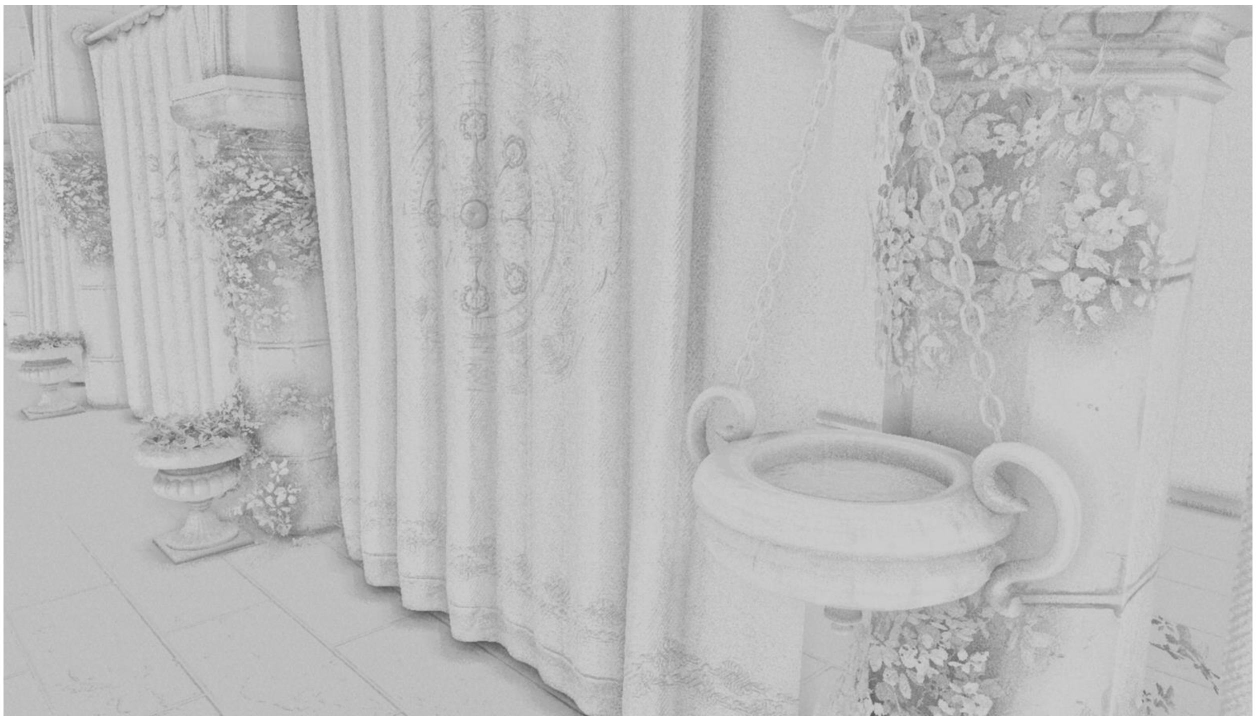
Figure 1: *The Sponza scene with only SSAO.*

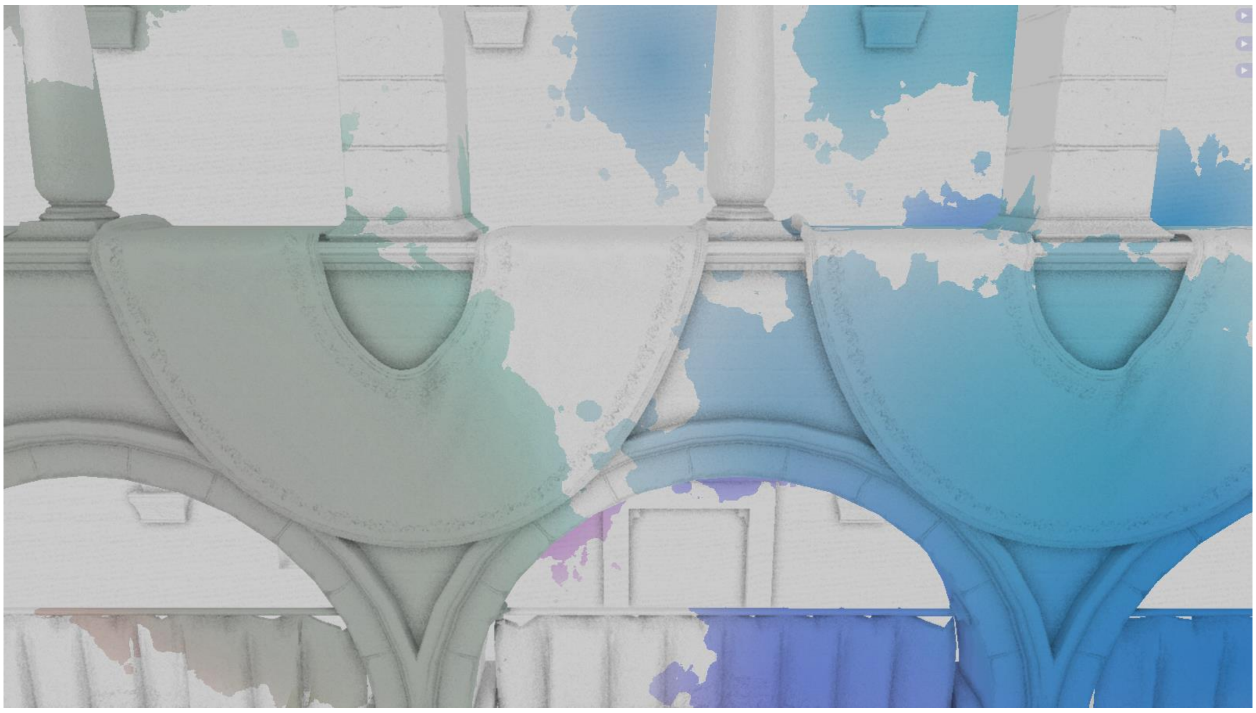
Figure 2: *Detailed paint splats can be achieved with seamlessly blended colors.*