# Screen-Space Secondary Lighting

**Dale Whinham**

Supervisor: Dr. Gary Ushaw

School of Computing Science
Newcastle University

This dissertation is submitted for the degree of
*BSc. Computer Science (Games Engineering)*

Word count: 12,047

May 2017

# Declaration

I declare that this dissertation represents my own work, except where otherwise stated.

Dale Whinham
May 2017

# Acknowledgements

I would like to extend my sincerest thanks to Dr. Gary Ushaw, Dr. Graham Morgan, and Dr. William Blewitt for their valuable support and assistance provided during the course of the project and the writing of this dissertation.

# Abstract

Simulating secondary lighting (reflected light) within a computer-generated 3D environment is a very challenging and computationally expensive problem to solve. The canonical solutions involving raytracing produce the most convincing results, however the high complexity of raytracing makes it unsuitable for the real-time rendering performance we require in today's fast-paced video games.

In this dissertation, we investigate and implement an advanced rendering technique called *screen-space ambient occlusion*, which seeks to approximate a secondary lighting effect in linear time, whilst still maintaining an acceptable level of realism. Through performance analysis, we find that this technique is an efficient and computationally inexpensive addition to a typical game engine, which makes it a good candidate for greatly improving the visual fidelity of the simulated 3D environment without severely impacting the overall performance of the game. We also show that the effect is achievable on current-generation smartphones with very promising performance figures.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In this chapter, we will set the scene for the project, introduce the hypothesis, and detail our proposed solution to the problem in the form of an overall aim, and set of individual objectives.

## 1.1 Setting the Scene

In the video games industry, computer-generated graphics play a critical role in shaping the overall experience of the end product. Modern video games often aim for a high level of realism and therefore make use of many different kinds of rendering algorithms to recreate the way things appear in the real world. In rendering, the simulation of the way light behaves allows us to create effects such as reflections, refractions, and shadows. Combining these effects adds additional perceived realism to a computer-generated scene.

These rendering effects come at a considerable computational cost, and most of the time a specialised piece of hardware dedicated to graphics rendering is used, the *graphics processing unit* (*GPU*). This has two advantages - it accelerates generation of the rendered output, and it keeps the system's *central processing unit* (*CPU*) free to take care of other tasks.

However, even with dedicated hardware to take care of rendering, we still need to be concerned with the efficiency of the rendering algorithms we use. In a video game, frequent slowdowns during gameplay are detrimental to the experience and can make a game less enjoyable, or more difficult to play.

## 1.2 Hypothesis

For many problems in computing, a compromise can be made between accuracy and performance, and this can make infeasible tasks a possibility. From our background research, we deduce that this idea can be applied to rendering algorithms, and so it follows that many expensive rendering effects can be replaced with less costly approximations, where the effect is considered "good enough" for inclusion in a video game.

In this dissertation, we investigate the implementation and performance of an advanced rendering technique known as *screen-space ambient occlusion* (*SSAO*). This algorithm attempts to approximate the effects of the much more expensive canonical ambient occlusion effect. Ambient occlusion adds depth to a computer-generated scene by darkening surfaces that are less exposed to light, as one would expect in the real world.

## 1.3 Aim and Objectives

The original aim of the project was to implement and analyse *two* screen-space rendering algorithms, however we realised midway into the project that this would be quite an ambitious goal considering the time constraints. The objectives were amended such that only the first algorithm would be implemented, and instead we decided to analyse the performance of the first algorithm in more detail.

The new aim is to implement the SSAO algorithm, and determine its suitability for use in video games, commenting on the ease of implementation, the performance impact it has on the rendering process, and the visual quality of the output. We decided to take the opportunity to attempt to bring the algorithm to a second, less powerful platform, and if successful, compare how well the algorithm scales.

This will be broken down into the following objectives:

1. Prepare a suitable graphics framework using *OpenGL* and *C++* for the implementation of the screen-space rendering algorithm.
2. Research the use of SSAO algorithms - calculating lighting as part of a post-processing stage in which only the visible pixels are considered.
3. Implement the SSAO algorithm.

4. Evaluate the performance:
   (a) Use a number of test platforms representing typical hardware used for playing video games.
   (b) Measure the impact on the rendering process in terms of time spent performing the calculations.
   (c) Adjust a number of tuning parameters the algorithm offers, commenting on how they affect the performance and visual fidelity.

## 1.4 Dissertation Outline

The dissertation is structured as follows:

**Chapter 1** This chapter - introducing the topic area, our hypothesis, aim, and objectives.

**Chapter 2** We present our background research including references to previous work in this field. We introduce the reader to the complex computational problems faced by graphics developers in the video game industry, and how we aim to solve them using the concept of hardware-accelerated rasterisation and lighting calculations. We discuss the idea of global illumination and give an example of how ambient occlusion can improve a computer-generated scene. We then discuss how we can accelerate an approximation of ambient occlusion and review some examples of its use in video games. Finally, we discuss renderer pipeline architecture, and the theory of how screen-space ambient occlusion works.

**Chapter 3** We discuss the high-level architecture of our implementation, which target platforms we chose, and which tools and technologies we chose, including our reasoning for doing so. We then discuss each of our software components in more detail, as well as our test data and profiling procedures.

**Chapter 4** The results collected during our testing are presented in a visual form, and we discuss and evaluate our findings, commenting on raw performance as well as visual quality.

**Chapter 5** Finally, we reflect on what we have learned as a result of completing the project. We discuss some of the problems we faced and what might have been

done differently. We then bring the dissertation to a conclusion by suggesting some further areas to investigate should there be a desire to continue our research.

# Chapter 2

# Background Research

In this chapter, we will look at the background research which was carried out in support of the project. We will examine previous research in this field, and discuss how we build on it to achieve our aims and objectives.

## 2.1 Rasterisation

*Three-dimensional computer-generated graphics* (*3D graphics*) are, as a generalisation, difficult and complex to compute, presenting many challenges for computer architects, hardware designers, and software engineers alike. They can easily involve many thousands of mathematical computations involving fractional numbers, which are needed to transform positions in 3D space (also known as *vertices*) into the *two-dimensional* (*2D*) space of a monitor or television. The process of drawing a 3D representation of an object onto a 2D surface for the purposes of presenting it on a video display is known as *rasterisation*.

In a video game scenario, all of the calculations required for the rendition of a scene must be completed as quickly as possible in order for the game to appear fluid to the player - typically in well under $\frac{1}{60}^{\text{th}}$ of a second. A typical video game will have many other subsystems to update as well as the graphics, such as audio, physics, human input from a controller, and networking in the case of a multiplayer online game. Hence, the compute time available for graphics rendition may actually only be a small fraction of that $\frac{1}{60}^{\text{th}}$ of a second window.

The problem of how to process and rasterise 3D geometry quickly has been (and continues to be) solved with advancements in GPU technology. The GPU has become increasingly ubiquitous ever since the introduction of dedicated 3D acceleration hardware in consumer products in the mid-to-late 1990s. A trend which began as custom integrated circuits inside early 3D-capable games consoles, such as the Sony PlayStation and Nintendo 64, would later inspire the invention of 3D accelerator cards for desktop IBM PC compatibles. This made high-quality real-time 3D graphics in video games a possibility on ordinary household computers. As of 2017, GPUs can now be found embedded within small devices such smartphones, tablets, and handheld games consoles.

Now that GPUs are commonplace, we can easily rasterise complex geometry on average computer hardware thanks to the vast parallelism a GPU provides, which makes the process of transforming 3D positions into 2D space for display on a screen a trivial task. The focus now shifts towards the problem of approaching what is known as *photorealism* in real-time - producing computer-generated output which is difficult to distinguish from a photograph or video of a real life scene.

## 2.2 Light and Global Illumination

Photorealism in computer graphics is achieved by simulating the behaviour of light. This creates an even bigger problem for us, because light interacts with objects in many different ways - a light-coloured polished surface will reflect photons away from it, whereas a darker matte surface will absorb them. If we consider a scene with a room full of shiny objects, we would have to follow the path of each photon as it bounces from one shiny object to another - possibly amounting to thousands of bounces per photon. In real life, photons can potentially travel for a very long time and experience an enormous amount of bounces before eventually being absorbed, so this kind of simulation will typically place an artificial limit on the number of bounces to simulate.

This technique is often used in the film industry, and the process of doing so is known as *raytracing*. However, in the film industry there is no requirement for instantaneous, real-time rendering performance, because the output will be a fixed media such as film or digital video, and much more time can be afforded to perform long computations. Additionally, the input for the rendering process is known ahead of time, and so it can be distributed across several computers, also known as a *render*

*farm*. In this way, each computer can render individual frames of animation in parallel with others.

In video games, we need to find other ways to achieve similar effects, but in far less time, and with only a single machine at our disposal. Raytracing is currently not feasible for real-time rendering because of its complexity. Instead, we look for patterns and generalisations that we can use to approximate the behaviour of light under certain conditions, to create effects such as reflections and shadows.

The algorithms which help us achieve photorealism by simulating the behaviour of light belong to a family of algorithms known as *global illumination* or *secondary lighting* algorithms. They have the capability of recreating the effects of light having bounced from one surface within the scene to another, and are not limited to simulating *direct light* - light which has travelled directly from the source to a surface.

## 2.3 Ambient Occlusion

One of the algorithms belonging to the global illumination family of algorithms is known as *ambient occlusion*. This technique allows one to determine how exposed a surface is to ambient light. If a point on a surface is partially or fully blocked from receiving ambient light (occluded), it can be shaded darker than non-occluded points to make it appear more life-like.

Ambient occlusion has its origins in Miller's work on surface accessibility in 1994. Miller proposed an approach for determining the likelihood of a region to have accumulated dirt or become tarnished due to the region being a concave corner or crease. This calculation could then be used to shade the region darker accordingly (Miller, 1994).

In later years, the film production industry would begin to make use of a similar technique to determine the accessibility of a surface to light. This is evidenced in the application notes for Pixar's *RenderMan* software, which describes the technique and names it as ambient occlusion (Christensen, 2002).

Figures 2.1 and 2.2 show an example of the visual improvement ambient occlusion has on a computer-generated scene. In the second image, notice the soft shadows near the ceiling and around the shelving. This is an interesting and desirable effect to have, because it adds depth and realism to an image. Corners of geometry which would not receive direct light are softly darkened to make them appear as if they were in shadow.

**Figure 2.1** A scene from the video game *Dying Light* (2015), with ambient occlusion disabled.

**Figure 2.2** As in figure 2.1, but with ambient occlusion enabled.

To obtain a "true" ambient occlusion effect, we would need to trace the paths of light rays as they interact with all objects in the scene, and determine whether any light reflected from their surfaces reaches any point in question, taking into account the surface properties of each object involved. In a video game, even objects not visible

to the player would need to be taken into account when performing these calculations, such as geometry behind the player.

The problem here is a matter of computational complexity. Referring back to our brief discussion of raytracing, it should be clear to see that calculating true ambient occlusion would be a very costly operation for a video game to have to perform for every frame of graphical updates. Shanmugam and Arikan found this to be the case in 2007.

Indeed, for static geometry - geometry which never moves, such as terrain - we could pre-calculate the contributions each surface has toward the ambient occlusion, and store the data alongside the geometry. This is known as *baking*, and it allows data that is expensive to compute to be quickly looked up at run-time rather than computed on-the-fly. However, this is not always possible, especially for games which have hundreds of moving entities. An example of this could be a game series such as *Grand Theft Auto*, where city scenes with lots of moving traffic are commonplace, and all of the vehicles can be interacted with by the player.

The interesting nature of the problem along with the striking amount of visual improvement the ambient occlusion effect adds to an otherwise simple and "flat"-looking 3D scene are some of the main reasons why we decided to investigate it.

## 2.4 Hardware Acceleration of Ambient Occlusion

*Hardware acceleration* refers to the use of specialist or bespoke hardware to speed up a computational task. We use a GPU to facilitate the hardware acceleration of graphics computations.

With the assumption that we have a GPU available to us, it would make sense to pursue methods of accelerating the ambient occlusion effect which utilise its strengths to our best advantage. A typical GPU will excel at tasks that are "embarrassingly parallel", which means that the task is easily broken down into a large number of sub-tasks that are not interdependent of each other (Herlihy and Shavit, 2008). This is what makes the GPU well-suited for graphics. A typical 1080p-resolution monitor or television consists of over 2 million *pixels*, and a desktop computer's main CPU will typically have 4 cores, so a software rasteriser running on the CPU would potentially only be able to divide the workload by 4. As of 2017, the current generation of GPUs manufactured by NVIDIA offer a variety of execution core configurations, ranging from

640 cores in the budget model to 3840 cores in the high-end model (NVIDIA, 2017). This usage of hundreds or thousands of parallel cores means that a workload such as rasterisation on the 1080p screen is reduced to a much smaller number of parallel iterations.

In 2007, Shanmugam and Arikan introduced their approach to a hardware-accelerated ambient occlusion approximation, which used the programmable and parallel nature of modern GPUs to break down the computation into an efficient multi-pass algorithm. One pass is able to compute what they refer to as "high-frequency" occlusion, which approximates the occlusion of nearby surfaces, and a second, independent pass approximates the "low-frequency" occlusion of objects that are more distant.

The most important takeaways from the methods presented in Shanmugam and Arikan, 2007 are:

- Only buffers containing *depth data* and *normal data* - which are very likely to already exist in a typical video game engine's rendering pipeline - are required to compute the high-frequency effect.
- The complexity of computing the ambient occlusion is completely independent of the complexity of the 3D geometry that makes up the scene. This means that the performance of the algorithm will not change if there are more or less objects in the 3D scene, or if the objects are more or less detailed.
- The algorithm computes the occlusion on a per-frame basis, which makes it suitable for moving or deformable geometry, and is therefore not limited to static geometry such as terrain.
- The computations only involve data which actually has an influence on the final rendered output - i.e. visible to to the viewer. In other words, geometry which is behind the player in a video game or hidden behind other objects will not be processed unnecessarily.

To elaborate on the third point, the data used to calculate the ambient occlusion is the set of data that exists *after*:

1. The GPU has discarded position data for objects that are partially or fully outside the region of interest. This includes objects that are beyond the horizontal and vertical borders of the screen, as well as objects that are too far away from or

too near the "camera". Anything which would exist outside of the *view frustum* (see Figure 2.3) is discarded by the GPU - this is known as *clipping*.

2. The GPU has discarded image data for objects which have been determined to be hidden behind other objects. This process is known as *depth testing*, and occurs after a pixel representing a potentially visible location (known as a *fragment*) has been given its final colour. If a fragment has been determined to be behind the currently frontmost object in the scene, it can be discarded because it has "failed" the depth test.

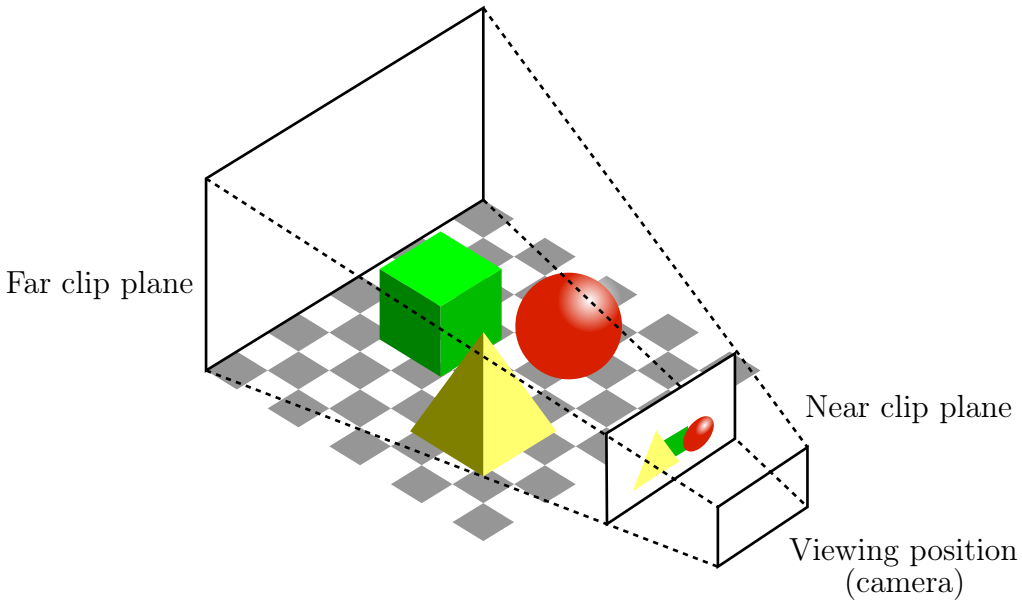



**Figure 2.3** Diagram showing the view frustum in relation to a 3D scene. The view frustum defines the region of interest - objects that are found to lie either fully or partially outside it will be clipped.

The data which remains after these two stages of the so-called *rendering pipeline* is said to be in *screen-space*, because it is associated with a visible pixel on the screen (N.B. Shanmugam and Arikan use the term "image-space" to refer to the same concept).

Though Shanmugam and Arikan detailed two separate algorithms, we will only concern ourselves with the high-frequency (near occlusion) algorithm. We will refer to this and similar techniques as SSAO algorithms.

## 2.5 SSAO: Use in Video Games

Shanmugam and Arikan's paper brought the concept of performing ambient occlusion computations in screen-space to the attention of the wider graphics community, including many video games developers, who quickly began to work on their own implementations and variations. For many, this was an exciting development in graphics, as it showed that the effect was possible to achieve in realtime on consumer-grade hardware. One such blogger and game developer referred to 2007 as "the year SSAO broke" (Shopf, 2008).

In November 2007, Crytek, a video games developer based in Germany, released *Crysis*, a well-received first-person-shooter game for the PC. The game quickly gained a reputation for its unprecedented demands on computer hardware when set to the highest rendering detail settings, due to the extremely advanced capabilities of its renderer. Crysis would go on to be used as a benchmarking tool in reviews of GPUs for several years because of this, and its reputation led to the comedic use of the phrase "but can it run Crysis?" in response to reviews of new PC hardware (Leather, 2009).

Crysis is especially notable to us, because among the many advanced rendering techniques offered by its game engine, SSAO is one of its most prominent. Additionally, Crysis is said to be the one of the first video games to ship with this rendering technique (NVIDIA, 2013).

In the months leading up to the release of Crysis, a paper was published by one of the engineers at Crytek which detailed some of new technologies found in their new game's engine. Mittring, 2007 briefly describes how Crytek engineers simplified the algorithm to a single pass (as opposed to the two passes shown in Shanmugam and Arikan, 2007) and also managed to compute their version of the effect with only the depth data that already existed in their rendering pipeline (Shanmugam and Arikan, 2007 used a combination of depth data *and* normal data). Although Mittring, 2007 spares the exact implementation details of their version of the algorithm, it was this paper that gave SSAO its popular name, and convinced the video games industry that such an effect was a possibility within video game engines.

Figure 2.4 shows a typical scene in Crysis, without SSAO enabled. Although some shadows are being computed for objects by means of *shadow mapping* (note the shadows being cast by the large tree onto the wall), the scene lacks some perceivable depth because there is little colour variation within the concave sections of the wall, and within the vegetation near the front of the scene.

**Figure 2.4** A scene from the video game *Crysis* (2007), without ambient occlusion.

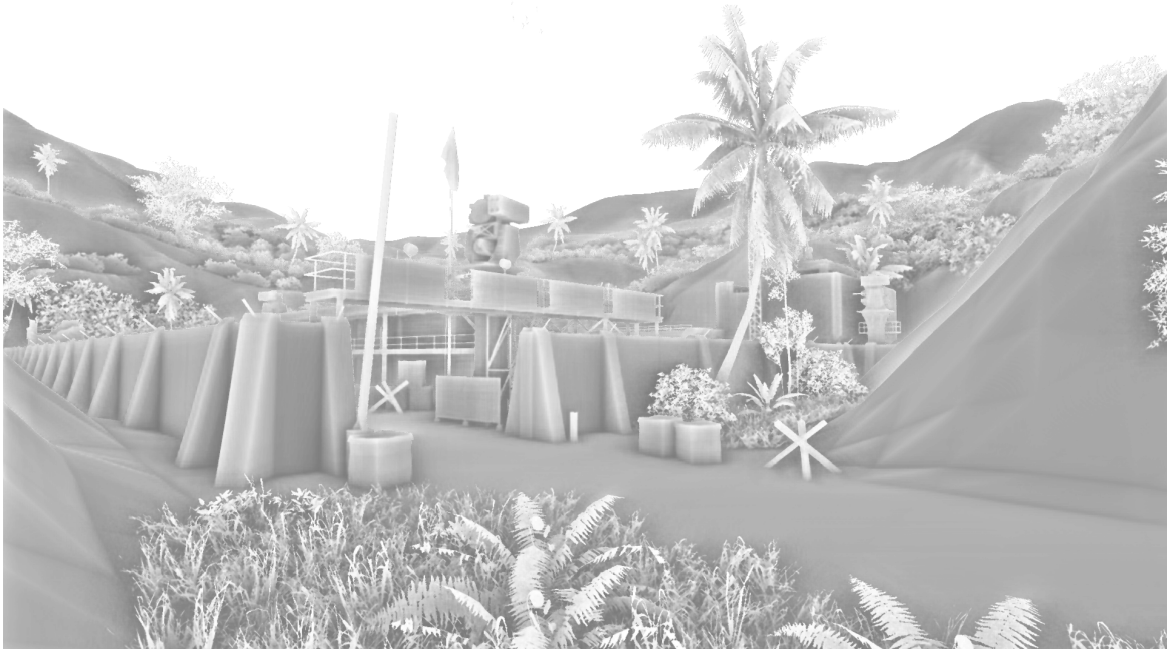


**Figure 2.5** The isolated ambient occlusion component of the scene depicted in Figure 2.4.

In Figure 2.5, we can see the results of an SSAO computation on the same scene. At this stage, no colour is involved as the output of the algorithm is a single number for each pixel in the image that represents its *occlusion factor*. Areas that are more

occluded receive a higher occlusion factor value, and this value is normalised between 0 and 1.

For the purposes of visualisation, we assume that the value 1 represents full brightness (white) and 0 represents no brightness (black). We then subtract the occlusion factor from 1 to produce the image. It is useful to note here that the amount of impact the occlusion factor has, could easily be adjusted for artistic reasons by multiplying it with a weight value, so that the effect can be emphasised or made more subtle by increasing or decreasing the darkening effect if the designer desires.

**Figure 2.6** After combining the image data shown in Figures 2.4 and 2.5 - the final composite image.

With an occlusion factor calculated for every pixel in the scene, it can then be combined with the original output in Figure 2.5 to produce the final image shown in Figure 2.6. Note how the concave wall sections now have extra perceivable depth within the corners, and that individual blades of grass and leaves within the foliage at the front of the scene now stand out from one another.

With SSAO promising to be both efficient and highly appreciable as seen in our Crysis example, it should be easy to see why it is such an interesting topic of research.

## 2.6 Deferred Rendering Pipelines and the G-Buffer

In computer graphics, we often talk about the rendering pipeline. This refers to the several processing stages through which geometry data passes in order to produce the final image. One of the simplest forms of rendering pipeline is shown in Figure 2.7, and this is known as a *forward rendering* pipeline. A forward rendering pipeline is named so because the data moves directly forward - from the application software, through the pipeline, and onto the screen in a single pass.

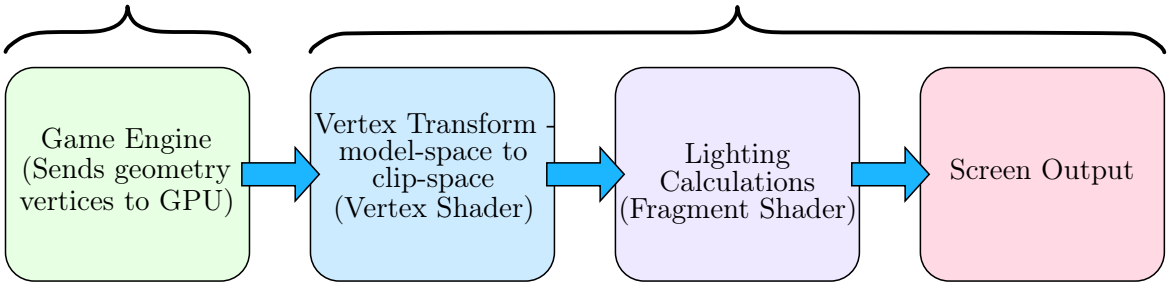



**Figure 2.7** Diagram showing a simple single-pass forward rendering pipeline.

In a forward rendering pipeline, the stages of processing are as follows:

1. The application (in our case, the game engine) sends some geometry data (vertices) to the GPU. A typical game engine will try to minimise the amount of geometry sent to the GPU in order to save memory bandwidth and increase performance as a result, because sending geometry to the GPU can be an expensive process. The game engine may use *culling* techniques to determine which geometry is not visible or is too far away - this will reduce the amount of geometry the GPU itself has to cull later in the pipeline.

2. A program which executes on the GPU, known as a *vertex shader*, processes the vertex data and applies a number of per-vertex *transforms* to it. The transforms will typically involve a number of mathematical matrix multiplications to apply translations (repositioning), rotations, and scaling to the vertices. A piece of geometry will typically have its coordinates defined in relation to the origin of a Cartesian coordinate system ($x = 0, y = 0, z = 0$), with the the origin at its centre - it can be said to be defined in *model space*. Applying a transformation matrix lets us reposition, rotate, and resize an object in relation to the game world. Hence, this stage can be said to be taking a object from *model space* to *world space*, and involves the use of a *model matrix*.

Another matrix multiplication is used to simulate the effect of a camera by moving and rotating all world geometry in relation to the viewing position - this is known as going from model space to *view space*, and uses a *view matrix*. A final matrix multiplication is then applied which defines the view frustum (or viewing volume), which is typically either a *perspective projection* or *orthographic projection*. This matrix takes the object from view space to *clip space*, as anything lying outside of the view frustum is then discarded (clipped) by the GPU. At this stage, all visible vertices will now lie inside the range $-1$ to 1 on all three axes, which is known as being within *normalised device coordinates*.

3. Once the geometry has been transformed and clipped, it is sent to a second program executing on the GPU, known as a *fragment shader*. As we mentioned earlier, a fragment is a potentially visible pixel associated with a location in 3D space. The fragment shader determines the final colour of the pixel, and this is the opportunity at which a typical forward renderer will apply some form of lighting model to calculate the brightness of the surface. An example of this is *Phong shading*, which makes use of the normal to a surface to calculate brightness based on light position, incidence angle, and reflection angle (Phong, 1975). If an object uses image-based texturing (for example, applying an image of some bricks to a surface to make it appear like a brick wall), then the GPU can sample the texture to determine the colour of each pixel.

4. If a fragment passes the depth test we mentioned earlier, the pixel is written to the *framebuffer*, and the *depth buffer* is also updated to reflect the new depth at that screen-space location. The framebuffer and depth buffer are temporary buffers whose dimensions are the same size as the screen. Often a technique known as *double buffering* is used, where two framebuffers, known as the front and back buffers, are used. The back buffer is cleared, drawn onto, and then made visible (presented) by swapping it with the back buffer once all drawing operations are complete. This helps to avoid graphical tearing, as the back buffer can be swapped with the front buffer in perfect synchronisation with the refresh rate of the monitor or television.

The advantages of a forward rendering pipeline are that they are simple to implement and have low memory requirements - we don't need any extra storage other than the screen output buffers. The game engine need only perform one rendering pass per frame, and this may be enough for the simplest rendering requirements - i.e. only a few light sources are present within the game world. However, since all lighting calculations

need to occur at once, we can easily end up with an extremely complex fragment shader stage performing lighting calculations for every single fragment, even if they end up being discarded as a result of failing the depth test. If we were to use "big O" notation for the computational complexity of a forward renderer's lighting stage, it would be $O(f \times l)$, where $f$ represents the number of fragments, and $l$ represents the number of lights (Owens, 2013).

Additionally, since we are interested in implementing an SSAO algorithm, we need to be able to capture the screen-space geometry data set, and be able to refer to it when calculating our ambient occlusion factor. As we saw in Section 2.5, Crytek was able to create the effect by only using the depth buffer, and as such, their version of the effect is possible with a simple forward renderer. However, we will discuss a more advanced version of the effect in Section 2.7 for our own implementation, and one of its requirements is that we have additional screen-space data available to us as well as the depth buffer.

One solution that gives us screen-space geometry data buffers and also allows us to perform faster lighting calculations is called *deferred rendering*. As the name implies, we leave the lighting calculations until after we have had the opportunity to perform some extra processing on the screen-space geometry data. An example of a deferred rendering pipeline is shown in Figure 2.8. Here, a two-pass approach is taken, which separates lighting calculations into its own pass. What is important here is that the second pass only operates on the set of data that has been determined to contribute to the final image - we do not suffer from wasted lighting computations in the fragment shader this time.

In the first pass, we perform most of same operations we do in a forward renderer, until we get to the fragment shader stage. Here, the fragment shader no longer performs any lighting calculations, however, it does calculate the colour of a surface as if it were at full brightness, again applying any textures as necessary. We refer to this colour as the *albedo colour*. It then outputs data to a set of buffers in GPU memory, which are stored as textures. This means that the GPU can sample them later, as it would when sampling a texture to be applied to geometry, like in our brick wall example. The data outputted to the textures can vary here, but in our example we have recorded the view-space position, the view-space normal, and the albedo colour of every pixel.

The textures used for data storage in this fashion are referred to collectively as the *geometry buffer* or *G-buffer*. With it, we can quickly look up the original view-space position, normal, and albedo colour associated with each pixel in the final image. When

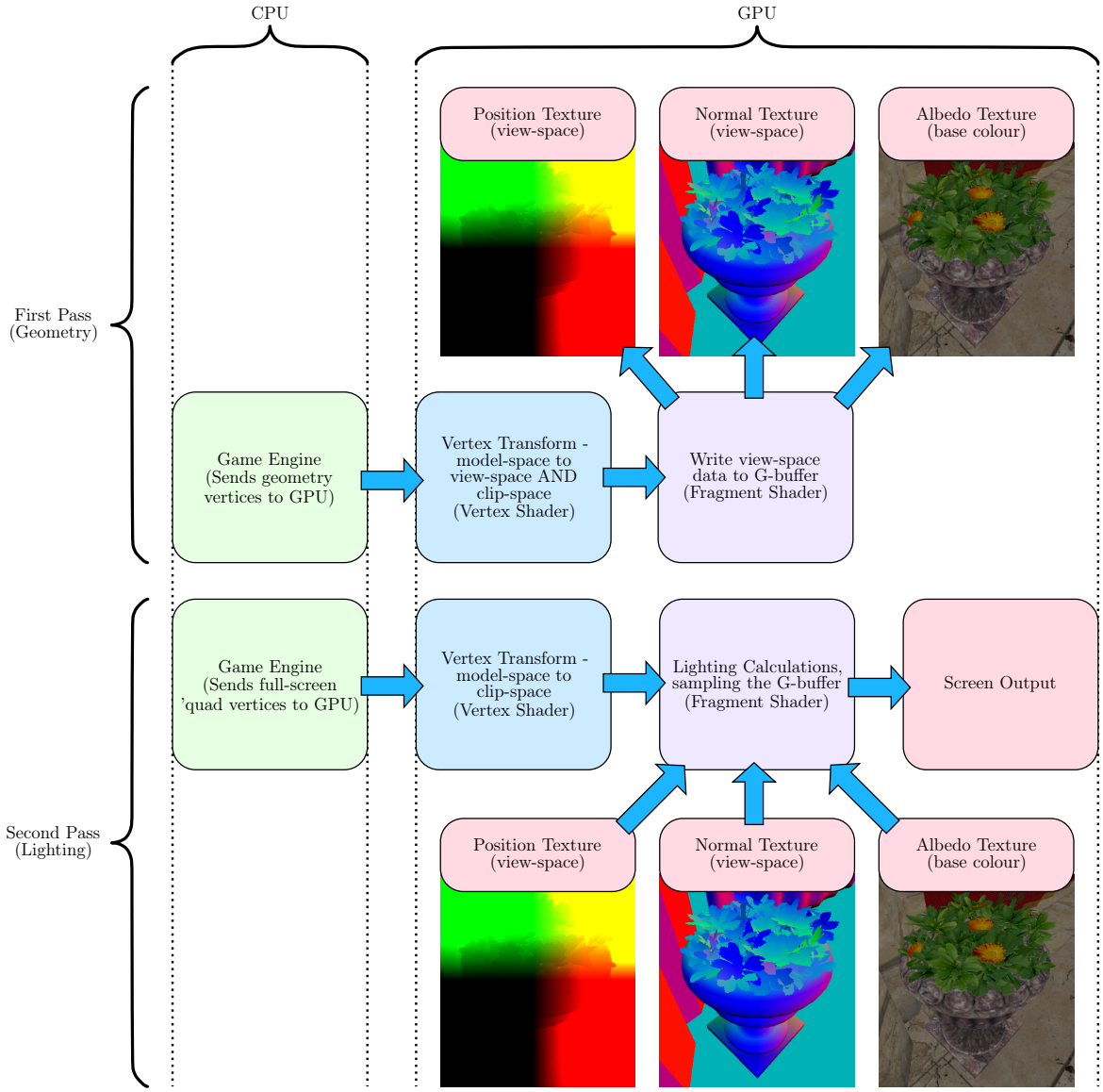



**Figure 2.8** Diagram showing a two-pass deferred rendering pipeline.

we visualise the position and normal textures of the G-buffer as in Figure 2.8, they appear as unusual colours because the red, green and blue components of the texture have been used to store $x$, $y$, and $z$ coordinates instead of actual colours. After the first pass, there is no screen output - only the G-buffer is written to.

In the second pass, we do not send the game world's 3D geometry to the GPU. Instead, we render a flat 2D full-screen rectangle (known as a *quad* in computer graphics) which spans the entire screen. The vertex shader can be very simple for this pass if care is taken - a rectangle whose bottom-left coordinates are $x = -1, y = -1$

and top-right coordinates are $x = 1, y = 1$ will require no transformation as it covers the whole range defined by normalised device coordinates on the $x$ and $y$ axes.

In the fragment shader stage, we sample the G-buffer for each pixel covered by our full-screen rectangle, and we extract the position, normal, and albedo from it. We can now perform the same lighting calculations as we did in our forward rendering pipeline, except this time, we can be assured that all lighting calculations will be used. We can say that the lighting calculations are being done in screen-space, as we are only operating on data associated with the final screen output. The computational complexity is reduced to $O(p \times l)$, where $p$ represents the number of screen pixels, and $l$ represents the number of lights (Owens, 2013).

Most importantly for us, we can insert additional passes into this pipeline that make use of the G-buffer, and extend the final pass to include data generated during our intermediate passes. This gives us our opportunity to calculate ambient occlusion in screen-space, the process of which we will describe in the next section.

## 2.7 SSAO: Theory of Operation

For a given pixel, SSAO is computed by inspecting the depth ($z$ value) of a number of randomly-generated points in clip space surrounding the position associated with the pixel. This is known as sampling, and SSAO algorithms tend to use either a sampling sphere or hemisphere of some radius. The collection of samples is called a *sample kernel*, and we generate it once during the initialisation of the application. We will go into further detail on how the samples are generated in Section 2.7.3.

Note that the more samples we use, the slower the algorithm's performance, as the performance of the SSAO stage can be said to be $O(p \times s)$, where $p$ is the number of screen pixels, and $s$ is the number of samples. We will look at ways to keep the sample count low, but still have acceptable results by post-processing the SSAO stage to remove noise caused by undersampling.

### 2.7.1 Crytek SSAO

Crytek showed that SSAO is possible by only using screen-space depth data. Figure 2.9 shows how their approach makes use of a sampling sphere to inspect the depth of some random points around a position. The more points that are determined to be

inside geometry, the higher our occlusion factor, and the darker we shade that pixel. In Figure 2.9, we can see that ten out of the sixteen sample points are inside the geometry. We know this because their depth, or $z$ value is greater than the $z$ value present in our G-buffer, which represents the depth of the geometry.

The fraction of samples inside geometry out of the size of our sample kernel gives us our occlusion factor. Referring back to Section 2.5 where we saw Crysis using the occlusion factor to darken occluded areas of a scene, we can determine a brightness factor for a given pixel using the formula:

$$\text{occlusion factor} = \frac{\text{samples inside geometry}}{\text{total samples}}$$

$$\text{brightness} = 1 - \text{occlusion factor}$$

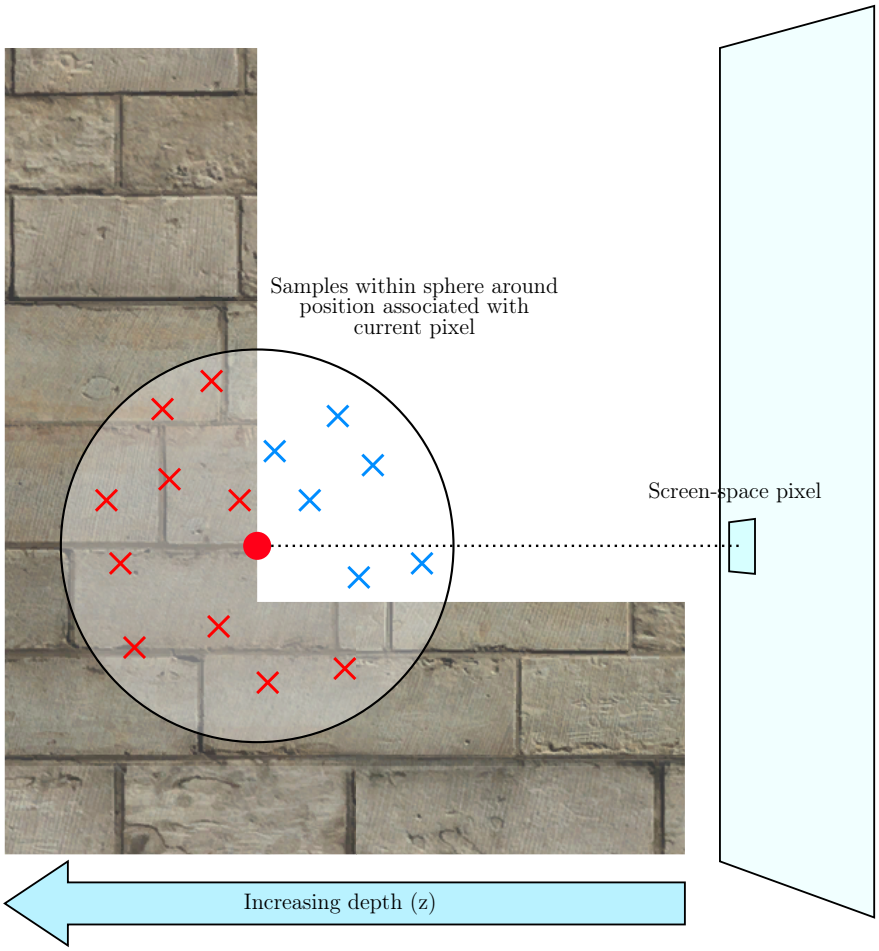



**Figure 2.9** Diagram showing Crytek's method of using a sampling sphere to calculate the ambient occlusion factor for a point.

Crytek's technique is simple and effective. When we sample corners or concave areas, we end up with a greater proportion of samples inside geometry, and the area is darkened. However, it has a characteristic which may or may not be desirable depending on the artistic goals of the designer. Looking at Figure 2.9, we realise that about half of the samples will *always* be inside geometry when testing a point on a flat surface. This has the effect of "greying" surfaces that would otherwise be fully exposed to light. Additionally, as we get closer to a non-occluded edge, the number of samples inside geometry tails off, and the edge of the surface appears lighter, with a kind of "halo" effect. Figure 2.10 shows these artifacts in more detail. In practice, these artifacts may not be noticable in the final scene, but it is interesting to note.

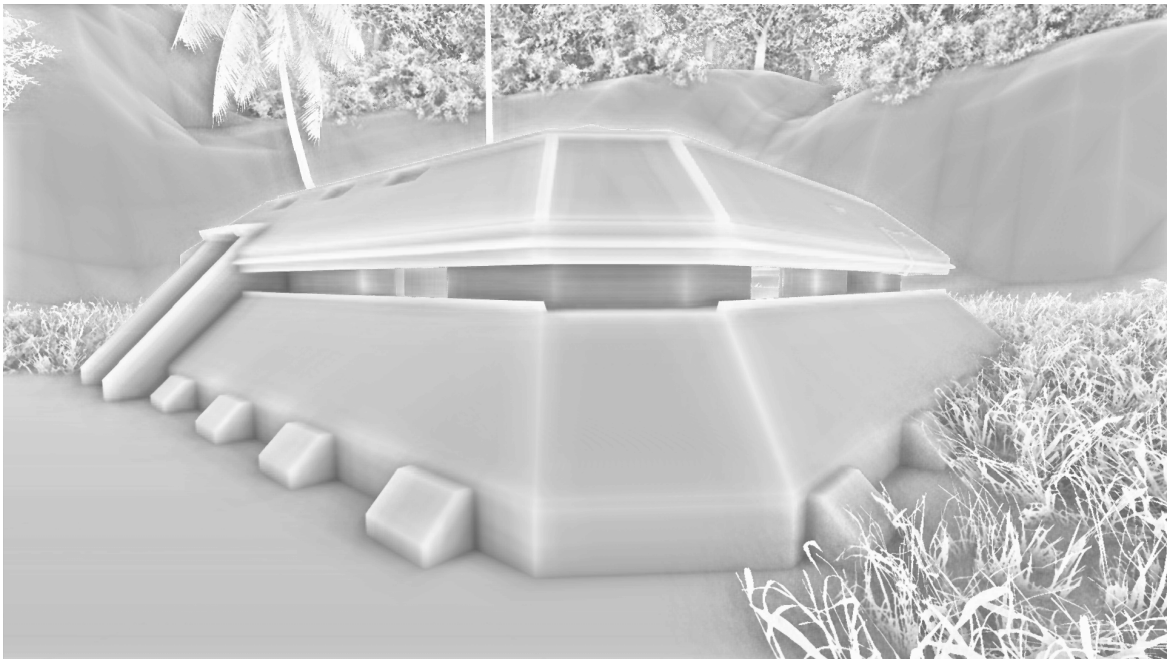


**Figure 2.10** An example of the exposed surface darkening and edge halo characteristics of Crytek's SSAO algorithm.

To avoid these artifacts, we can use an improved version of the algorithm so that a surface-aligned hemisphere is used instead of a sphere. This way, the surface we are interested in does not contribute to its own occlusion factor - we can say it no longer suffers from self-occlusion.

### 2.7.2 Hemispherical SSAO, aka. StarCraft II SSAO

In 2008, American games developer Blizzard Entertainment showed that they were able to overcome the self-occlusion problem when implementing SSAO in their video

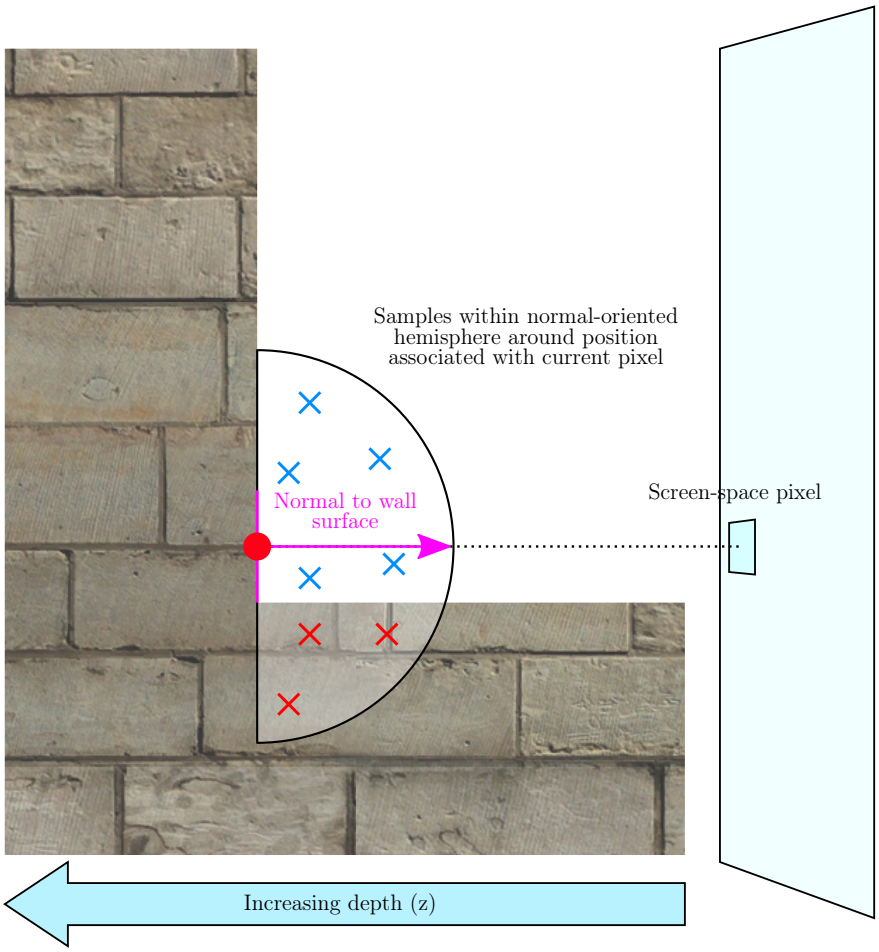



**Figure 2.11** Diagram showing alternative method of using a normal-oriented sampling hemisphere to calculate the ambient occlusion factor for a point.

game *StarCraft II* (Filion and McNaughton, 2008). In Figure 2.11 we can see how the technique has been modified to use a hemisphere. We align the base of the hemisphere such that it lies perpendicular to the point being tested, and to achieve this, we make use of the normal. In Section 2.6, we discussed the idea of placing normals into the G-buffer, and this is where having quick access to the normal associated with a pixel is very useful.

Our hemispherical sampling kernel is constructed in the same way as for a spherical kernel, except we clamp the radius on one axis to be positive-only (between 0 to 1).

We can offset a sample from our kernel by adding the sample's $x$, $y$, and $z$ coordinates to our position coordinates retrieved from the G-buffer. In order to re-orient the sample with respect to the normal at that position, achieving the effect we see in Figure 2.11, we need to employ a special transformation matrix known as a *TBN matrix*. The TBN

matrix is named so because it is constructed using a tangent, bitangent, and normal. In a similar way to how we move our game world's geometry from model-space, to view-space, to clip-space using transformation matrices, a TBN matrix moves a point from *tangent-space* to some other space. In our case, we want to move the point into view-space to match the rest of our position data.

Figure 2.12 shows our hemispherical samples in tangent space, and how the tangent, bitangent, and normal vectors are related to each other. When generating samples, we can use the $x$ axis to represent tangent, $y$ for bitangent, and $z$ for normal. The TBN matrix allows us to rotate the hemisphere such that its normal is perfectly aligned with the normal of the point we are sampling around.

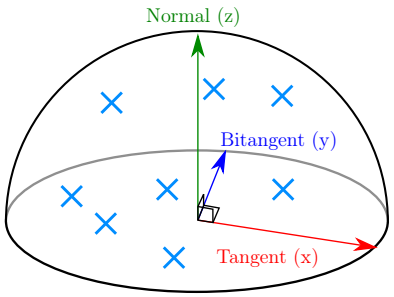



**Figure 2.12** Samples within a hemisphere, in tangent-space.

The tangent and bitangent effectively determine the rotation around the normal (or $z$ axis) for the hemisphere. We could use the same values for tangent and bitangent for every pixel as we traverse screen-space and perform our sampling, but it turns out that doing so will create undesirable artifacts such as "banding", as the same predictable pattern of sampling would be applied along a piece of geometry. To avoid this, and to smoothen out the results, we can apply a random rotation to the hemisphere around the $z$ axis for every pixel. We can store an array of rotations as randomly-generated two-dimensional coordinates within the range $-1$ to 1 on the $x$ and $y$ axes ($z$ is always 0 as this is our axis of rotation). Then, we can upload them to the GPU as a texture as shown in Figure 2.13. This way, we can quickly access them inside the fragment shader program. In our implementation, we use a $4 \times 4$ texture of rotation vectors, which is repeated ("tiled") across the screen as we perform our SSAO calculations. This technique is similar to how randomisation was achieved in Filion and McNaughton, 2008.

The rotation vector acts as a new tangent, and when constructing our TBN matrix, we use it and the view space normal to derive the bitangent, and subsequently the matrix. This is done as follows:

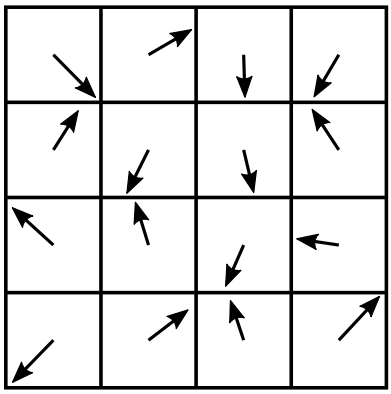


**Figure 2.13** A $4 \times 4$ texture containing a set of rotation vectors for randomising the hemispherical samples.

$$\vec{T} = \vec{R} - \vec{N} \times (\vec{R} \cdot \vec{N})$$
$$\vec{B} = \vec{N} \times \vec{T}$$
$$TBN = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$

...where $N$ is the view-space normal and $R$ is our random rotation vector. Note that when calculating $T$, we multiply $N$ by the vector *dot product* of $R$ and $N$, and subtract it from $R$. This matrix can then be used to properly re-orient and randomise any of our hemisphere samples.

Once a sample has been re-oriented and the depth at that sampling point determined, we project the sample into clip-space (by applying the projection matrix), so that it becomes associated with a nearby screen-space pixel. Finally, we examine that pixel's associated depth buffer value, and from here, as with the Crytek technique, we can determine whether the sample lies inside geometry or not.

Section B.1 contains the pseudocode for our SSAO implementation, which was based on information provided by Chapman, 2011.

### 2.7.3 Sample Kernel Generation

For generating a good set of samples inside a hemisphere, especially when the sample count is low, we decided to research a more sophisticated form of sampling which applies

a weighting such that the sample is more likely to appear near to the normal of the hemisphere. This is because occluders located near the top of the aligned hemisphere should have more of an occluding effect than objects located to the sides.

The method we thought would be most suitable is known as *Malley's method*, or *cosine-weighted hemisphere sampling* (Pharr and Humphreys, 2010). It helps give a better distributed sample set by sampling a 2D unit disc in a cosine-weighted fashion, and then projecting the samples up to the surface of the hemisphere.

As the number of samples has a direct impact on performance, we will be investigating different sizes of sample kernel and how they perform, both visually and in terms of rendering time.

### 2.7.4 Noise Reduction with Blur Filter

When calculating the SSAO values for each pixel, we will experience noise as a result of using lower sample counts. To alleviate this, we apply a simple blur algorithm that averages surrounding SSAO pixels. This process is done in a separate rendering pass in our pipeline, taking the SSAO stage as input, and producing a blurred texture as output. Figures 2.14a and 2.14b show the improvement in softness when blurring a scene with 8 samples. As ambient occlusion is meant to produce soft shadows, the blurring process helps to achieve this.

We used a simple blurring algorithm which is applied to every pixel of the SSAO pass output (Chapman, 2011). We sample a number of pixels around the current pixel and find their average, and the area of pixels processed in this way is the same size as our rotation vector texture, which should help minimise any repetition-induced patterns or artifacts in the final output. Section B.2 contains the pseudocode for the blur pass.

**(a)** No blur applied - notice the "grainy" appearance due to noise.

**(b)** Blur applied - less grainy and softer shadows.

**Figure 2.14** Comparison of SSAO before and after blur.

# Chapter 3

# Implementation

In this chapter, we will detail the architecture of our implementation and how it was developed. We will discuss the tools and technologies chosen along with the reasons for doing so, as well as the challenges faced and overcome during development.

## 3.1 High-Level Architecture Overview

We implemented a mock "game engine" to act as a graphics framework for our SSAO implementation. This software project will feature the minimum required core components of a typical game engine for implementing a deferred 3D rendering pipeline capable of supporting the hemispherical SSAO algorithm.

Figure 3.1 shows a high-level view of how the software project is structured. The software is broken down into a number of components, some of which consist of our own code, and some of which are third-party libraries which help us interact with our target platforms.

## 3.2 Hardware Platforms and Operating Systems

We decided to choose hardware platforms that are both commonplace and capable of supporting the techniques we wanted to implement. The decision was made to target previous-generation hardware as a minimum requirement, so that performance comparisons could be made between older and more current hardware.

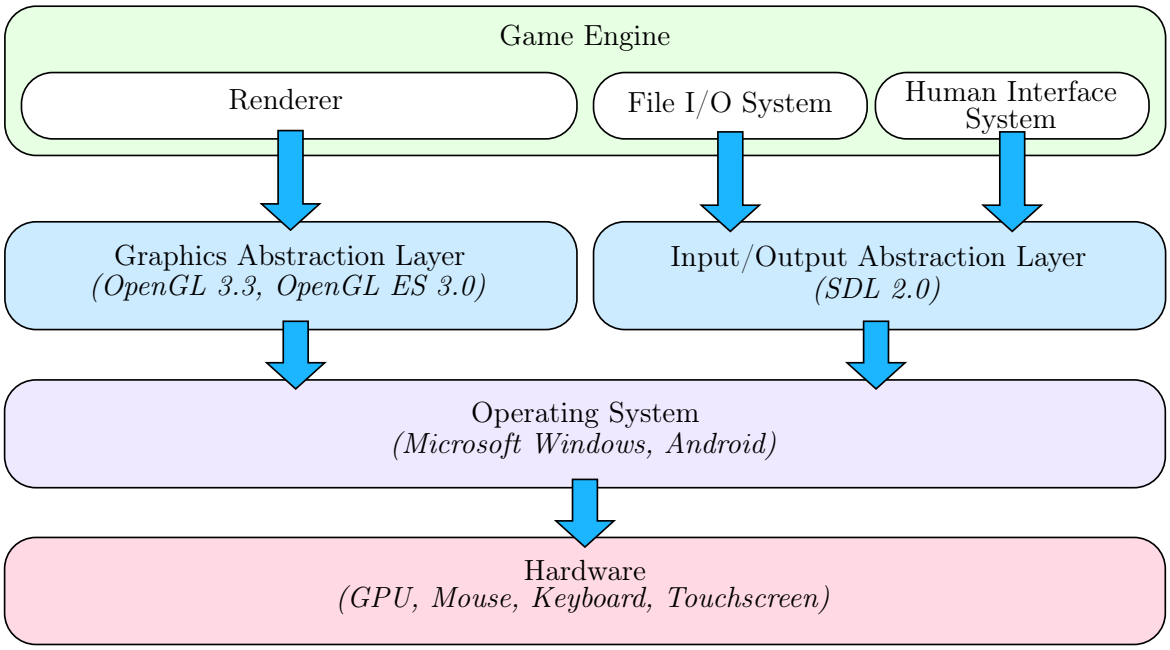



**Figure 3.1** Diagram showing the high-level architecture of the project.

Two main categories of hardware were chosen to be targeted by the project, PC and smartphone/tablet. The PC and smartphone, when combined, represent over half the market share in video games platforms - about 54% in 2016 according to McDonald, 2017, hence they are important platforms for us to consider.

Within the PC category, the vast majority of video games are played under the *Microsoft Windows* operating system with a *Direct3D 10* (or greater) compatible GPU (Steam, 2017). Direct3D is the graphics abstraction layer used exclusively under Microsoft platforms, but we opt for a similarly-capable version of the *OpenGL* abstraction layer which exists on other platforms as well as Windows. This allows our graphics code to be shared between PC and smartphone/tablet and thus compared like-for-like (we discuss this further in Section 3.4).

For the smartphone/tablet category, the *Android* operating system was found to be used on over 81% of smartphones sold in 2016 according to Gartner, 2017. The minimum version of *Android* we can target is version 4.4 (also known as "KitKat") because it is the first version of *Android* to support the kind of deferred graphics rendering we want to implement via. *OpenGL ES* version 3.0. According to Google, 2017, over 88% of *Android* devices are running version 4.4 of the operating system or greater, so we end up supporting the majority of *Android* devices in use at the time of writing.

An additional reason for choosing these platforms is that the development tools required in order to target them are freely available and easy to obtain from the vendors' websites (*Microsoft Visual Studio* and the *Android Software Development Kit/Native Development Kit* respectively).

The following devices were available to the author for testing. Devices marked with a "*" were available for more in-depth testing with a GPU debugger - we will discuss this in Section 3.9.

| | **CPU** | **GPU** | **RAM** | **OS** |
|---|---|---|---|---|
| **Custom PC (2011)*** | Intel Core i7 920 | NVIDIA GeForce GTX580 | 6GB | Windows 10 |
| **MacBook Pro (2013)*** | Intel Core i7 4558U | NVIDIA GeForce GT750M | 16GB | Windows 10 (Boot Camp) |

**Table 3.1** Table of PC-class hardware used for testing.

| | **CPU** | **GPU** | **RAM** | **OS** |
|---|---|---|---|---|
| **Tesco Hudl 2 (2015)** | Intel Atom Z3735D | Intel HD Graphics (Bay Trail-T) | 2GB | Android 5.1 "Lollipop" |
| **Samsung Galaxy S6 Edge (2015)** | Samsung Exynos 7 Octa 7420 | ARM Mali T760 MP8 | 3GB | Android 7.0 "Nougat" |
| **OnePlus 3 (2016)*** | Qualcomm Snapdragon 820 | Qualcomm Adreno 530 | 6GB | Android 7.1.1 "Nougat" |
| **OnePlus 3T (2016)** | Qualcomm Snapdragon 821 | Qualcomm Adreno 530 | 6GB | Android 7.1.1 "Nougat" |
| **HTC 10 (2017)** | Qualcomm Snapdragon 820 | Qualcomm Adreno 530 | 4GB | Android 7.0 "Nougat" |

**Table 3.2** Table of smartphone/tablet-class hardware used for testing.

## 3.3 Programming Language and Middleware

The *C++* programming language was chosen for the implementation as it allows a vast amount of middleware relevant to our project to be used. It also offers the programmer the ability to access the underlying system at a lower level than is possible in managed languages that run in a virtual machine such as *Java* or *C#*.

The language offers no automatic memory management or garbage collection as found in *Java* and *C#*, which raises the difficulty of implementation. However, the advantage to us is that the runtime performance is more deterministic as we have more precise control over resource management. These are also some of the reasons the *C++* language is used within the video games industry, as high performance and deterministic memory management is important for real-time simulations, especially on platforms with limited memory and processing power such as consoles and smartphones. As our project is trying to determine an algorithm's suitability for use in video games, it makes sense to use the programming language that the games industry prefers.

Under *Microsoft Windows*, we can use *Microsoft Visual Studio* to compile *C++* code and produce an executable application. To do the same for *Android*, we use the *Android Native Development Kit*, or *NDK*. *Android* applications are normally written in *Java* and use the *Java* frameworks provided by the operating system, however the platform also allows "native" *C* and *C++* code to be integrated into a project. The NDK comprises a cross-compiler targeting the CPU architectures found inside *Android* devices (usually *ARM* but occasionally *Intel x86*) as well as programming interfaces that allow us to access the graphics abstraction layer, human interface devices (touchscreen) and other features of the device.

For *Android*, a thin *Java* wrapper is used to allow the application to be launched like a regular *Java Android* app. The wrapper handles the proper loading and initialisation of our *C++* code and middleware, then hands control over to the *C++* code. The application then behaves just like it does in our *Windows* PC version.

The middleware chosen for the project had to satisfy the following requirements:

- Support both of our target platforms.
- Interface with a programming language that can also target both of our platforms.
- Allow as much code as possible to be shared between both platforms such that we can compare the results like-for-like and avoid implementing the same code twice.

## 3.4 Graphics Abstraction Layer

The purpose of the graphics abstraction layer is to remove the need for the programmer to understand all of the nuances and hardware-specific behaviours of the GPU, and provide a standard interface for sending instructions to it. This means that the rendering code within the application will be able to run on any GPU which supports the abstraction layer.

The abstraction layer is designed to expose as much of the GPU's functionality as possible through a standard set of publicly-accessible functions and data formats. This is known as an *application programming interface*, or *API*.

Some examples of graphics abstraction layers include *Microsoft Direct3D*, *OpenGL*, *Glide*, and recently *Vulkan*. In order for them to work, the GPU manufacturer must

supply a driver that implements them. The driver converts the requests sent to the abstraction layer by the application software into GPU-specific commands which are generally kept private by the manufacturer and are unique to the GPU architecture.

We have chosen *OpenGL* (*Open Graphics Library*) and its mobile counterpart, *OpenGL ES* (*OpenGL for Embedded Systems*). *OpenGL ES* is used for small devices such as smartphones and tablets, and is modelled on regular *OpenGL* used on the PC, albeit with reduced functionality to reflect the lesser capabilities of mobile GPUs. Whilst the two are considered separate standards, with care, code can be authored that works on both with very few changes.

*OpenGL* and *OpenGL ES* allow us to compile and execute code on the GPU by using shaders, and it defines a shader language called *GLSL* which describes the syntax, functions, and capabilities. Like *OpenGL* and *OpenGL ES*, there exist many versions of *GLSL* with different levels of functionality, and each release of *OpenGL (ES)* has a matching version of *GLSL (ES)*.

We found that *OpenGL* version 3.3 and *OpenGL ES* version 3.0 were the lowest versions of both APIs that supported our approach to deferred rendering, and had the most in common with their shader languages. Whilst *OpenGL* 3.0 on the PC could certainly support our project, we would have had to supply a separate set of shader code with slightly different syntax for the smartphone version.

*OpenGL* 3.3 is also supported on the first PC GPUs which supported *Direct3D* 10 (the *NVIDIA GeForce 8* series and the *ATI Radeon HD 2000* series), which means that according to Steam, 2017, we can support over 96% of PCs used for playing video games as of March 2017.

## 3.5 Input/Output Abstraction Layer

The input/output (I/O) abstraction layer gives us a common interface for reading and writing files from disk, and receiving input from the user by means of mouse and keyboard on the PC, or touchscreen on the smartphone or tablet. Like the graphics abstraction layer, this removes the need for the programmer to understand and write platform-specific code for accessing these features of the underlying operating system.

We require disk I/O for reading in 3D assets and textures for our renderer to draw for us, as well as the shader code which gets compiled and sent to the GPU at runtime by our graphics abstraction layer. The paths used for finding our assets differs

between *Windows* and *Android* - under *Windows* we can simply look underneath the subdirectories where our application executable is located. Under *Android*, things are slightly more complicated as our assets are compressed inside the application bundle (known as an *Android Package Kit* or *APK*) to save disk space on the mobile device. To access them, we would normally need to use *Android*'s system APIs to locate and decompress them.

The *Simple DirectMedia Layer* or *SDL* is a cross-platform middleware that provides an I/O abstraction layer for Windows, *Android*, and many other operating systems. With it, we can write the code that loads our assets once, and have it work on any operating system supported by SDL. Hence, our File I/O system within the project uses *SDL* for all operations where files are loaded from disk, and under *Android* it transparently takes care of locating and decompressing them for us.

*SDL* also provides a standard API for receiving human input from the user. In our project, we use this to allow the user to look around and move within the virtual environment using a mouse and keyboard, or touchscreen. We also allow the user to control the parameters of the SSAO algorithm using the keyboard or touchscreen in order to see the effects and measure the impact it has on performance. Again, this allows us to use the same programming interface for receiving and processing input events on both of our target platforms.

## 3.6 Cross-Platform Game Engine

With the hardware, operating systems, abstraction layers and programming language decided upon, we have all of the critical components required to implement a *game engine*. Although we won't actually be implementing a game, we still refer to the software as a game engine as it contains a lot of the core functionality that would be seen in a typical game. We use the term "cross-platform" because our abstraction layers allow the engine to run on more than one platform.

The game engine is the part that we implement ourselves in the project, and it carries out the following tasks:

- Provides a test-bed for implementing and profiling our SSAO algorithm.
- Represents the software that would be used at the heart of a typical video game.

- Initialises the platform's resources (GPU, file I/O, human interface) and gains access to them.
- Loads in our 3D assets and GPU programs (shaders) from disk and prepares them for use.
- Sets up our deferred rendering pipeline.
- Enters a loop of processing user input, updating the "game world" accordingly, rendering the 3D scene, and presenting the results to the user.
- Cleanly shuts down and releases any resources that were used at runtime when the user wants to exit the program.

## 3.7 Deferred Rendering Pipeline with SSAO

In Section 2.6, we discussed deferred rendering pipelines and how we can extend them to include additional processing before calculating the final lighting for a scene. In Figure 3.2, we show how we have added the SSAO and SSAO blur stages into our own rendering subsystem within the game engine. The rendering pipeline consists of four individual passes. For testing purposes and profiling, the SSAO and blur passes can be disabled so that we can measure the impact they have on rendering performance.

The renderer exposes the following features through the game engine:

- Renders geometry with albedo to G-buffer.
- Can toggle SSAO on/off.
- Can toggle SSAO blurring on/off.
- Allows you to change the number of SSAO samples to 4, 8, 16, 32, 64, or 128.
- Allows you to set the SSAO sampling radius.
- Can visualise the three textures of the G-buffer (position, normal, albedo) individually.
- Can visualise the SSAO component individually.
- Performs a lighting pass which illuminated the scene with 4 coloured lights using Phong shading, whilst also integrating the SSAO component.

- Can display the performance in frames-per-second averaged over the last 10 frames.

## 3.8 3D Assets Used for Testing

We decided that it would be beneficial to have the game engine load in an interesting and reasonably complex 3D scene for the following reasons:

- It would allow us to see how the SSAO algorithm behaves on various irregular shapes and surfaces.
- It would, to some extent, simulate what a typical video game might present to the player as one of its stages.
- A more complex scene would cause any errors or flaws in our SSAO implementation to be made more obvious.

The data we chose to use is a scene called the "Sponza" and was originally authored by engineers at Crytek. The scene depicts the atrium of the Sponza Palace in Dubrovnik, Croatia. The 3D geometry data and textures have been placed into the public domain for the benefit of the graphics research community by its authors. It gives us a reasonably complex set of data to work with, consisting of 262,267 triangles and several high-resolution textures. We have used an updated version distributed by McGuire, 2011 which includes some corrections to the original Crytek release.

## 3.9 Renderer Profiling Procedure

We used two methods for measuring rendering performance. The first is a basic test which simply allows the renderer to run as fast as possible and measuring how many frames-per-second (FPS) we can achieve with various settings. This test can be performed on any machine capable of running our game engine, however it does not give us the full picture of how each pipeline stage is performing individually.

In order to drill down into the rendering pipeline and measure the performance at each stage accurately, we had to think carefully about the strategy we would use. Sending rendering commands to a GPU is often performed asynchronously or in batches

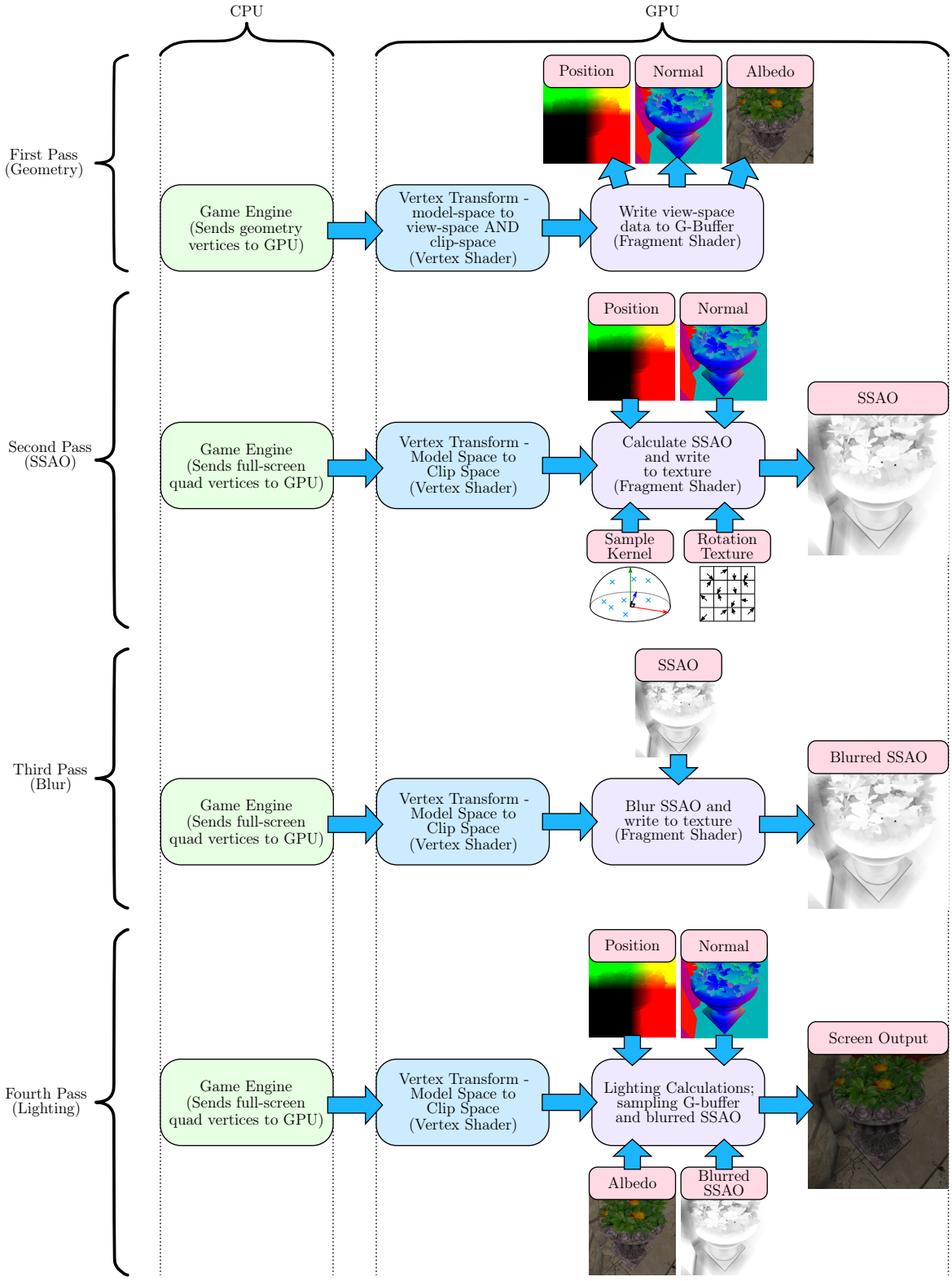



**Figure 3.2** The 4-pass SSAO deferred rendering pipeline used in the project.

by the GPU driver and/or the graphics abstraction layer. This non-determinism means that the traditional method of recording the wall clock time at two points within our code and finding the difference between them will not necessarily give us a meaningful result.

Instead, we can use specialist tools provided by the GPU vendor to retrieve more accurate timing information, because they have inside knowledge about how the implementation works, and can control what the driver is doing internally.

Our target PCs use NVIDIA-manufactured GPUs, so we can use *NVIDIA Nsight* - a GPU debugging tool which has comprehensive profiling features - to measure the performance of our renderer. Our *Android* smartphone's GPU is manufactured by Qualcomm, who provide the *Snapdragon Profiler* for debugging and profiling GPU code on devices using this family of hardware.

These tools allow us to run the program and capture a rendered frame. The capture data includes timings for all of the instructions the GPU has received, and we can get a good idea of how well our renderer is performing.

The vendor-specific and specialist nature of this method limits our ability to collect accurate measurements to devices which we are able to spend extra time with and install the debugging tools on (i.e. in our possession). However we were able to collect FPS data for a variety of other devices from willing participants.

Measurements were taken with the software compiled in "Release" mode for maximum performance (compiler optimisations enabled).

The procedure for performing a basic FPS test was as follows:

1. Run the program, and leave the camera in its initial position (avoid looking around). Leave SSAO and blur enabled.
2. Note the current SSAO sample level, and the value on the on-screen FPS counter.
3. Repeat the recording of FPS values for each SSAO sample setting until all six levels have been tested.

When using GPU debuggers, it is sometimes difficult to see which draw calls correspond to the stages of the deferred renderering pipeline. We decided to simplify the process by measuring the differences in frame timings when enabling and disabling SSAO and blur, and combined the geometry and lighting passes into one measurement.

The method used for each debugger was as follows:

1. Run the program, and leave the camera in its initial position (avoid looking around). Leave SSAO and blur enabled.

2. Note the current SSAO sample level, and use the GPU debugger to capture 50 frames.

3. Use the debugger's timeline features to determine the frame time in microseconds for the geometry, SSAO, blur, and lighting passes, averaged over a number of frames (we used 50):

    - *NVIDIA Nsight*: Perform a capture with SSAO and blur disabled, then note the *GPU Duration* recorded by the debugger to use as a baseline. Then, for each SSAO sample count, perform a capture with only SSAO enabled, and then with both SSAO and blur enabled. Use the differences between each set of timings to determine how long the SSAO and blur stages took on average, per frame.
    - *Qualcomm Snapdragon Profiler*: Perform a capture for each level of SSAO with and blur enabled. Look at the timeline to see when "rendering surfaces" are changed - these surface durations correspond to the durations of each of the 4 pipeline stages.

# Chapter 4

# Results and Evaluation

In this chapter, we present the measurements collected during the testing of our SSAO implementation and discuss the results. We also provide an evaluation, referring to our original objectives to determine whether we were successful in achieving our aim.

## 4.1 PC-Class Hardware

Two machines were tested in this category. The first, a custom-built gaming PC built in 2011, with an *NVIDIA GeForce GTX 580*, which was a high-end gaming GPU at the time of release. The second, a *MacBook Pro* from Apple, which is a laptop computer manufactured in late 2013. The *MacBook Pro* is equipped with an *NVIDIA GeForce GT 750M*, which is a GPU better suited to lighter tasks such as computer-aided design and video production work as opposed to gaming.

### 4.1.1 Frame Timings

Figures 4.1 and 4.2 show frame timings in microseconds for our SSAO renderer when run on our Custom PC and *MacBook Pro* respectively. The first thing that we notice is that the performance of the SSAO stage decreases in a linear fashion as we increase the number of samples - in other words, doubling the number of samples approximately doubles the time taken to complete the SSAO pass. Referring back to Section 2.7, this reinforces our statement about the time complexity being $O(\text{pixels} \times \text{samples})$.

When the samples are restricted to only 4 or 8, we can see that the combined SSAO and blur passes only add between 1-2 milliseconds to the total rendering time on our 2011-spec Custom PC, and 3-5 milliseconds on the *MacBook Pro* which suggests that the algorithm in this configuration is quite efficient, even on older generation or low-end hardware.

With the jump from 64 to 128 samples on the Custom PC (or above 8 samples on the *MacBook Pro*), we begin to experience a perceivable slowdown in the rendering performance. This is because we begin to exceed the $\frac{1}{60}^{\text{th}}$ of a second budget for giving our 60Hz monitor a new graphics frame in time for the next screen refresh. This suggests we are saturating the available memory bandwidth of the GPU - 128 iterations of G-buffer read operations (fetches) for every pixel incurs a severe performance penalty.

In Section 4.3, we notice that using higher sample counts yield diminishing returns after a certain point, so it doesn't make sense to use them in a real application.

We notice a possible area for improvement here - the blur stage, which seems to be quite expensive compared to the other stages, especially at 4 and 8 samples. We suggest that our blur algorithm is not very efficient because it is quite a basic implementation which duplicates its own work. Whilst it has the effect we desire, further research into blur algorithms could yield even better overall performance.

It is important to note two things:

- The *MacBook Pro*'s GPU is a mid-range laptop GPU, hence the significantly lower performance compared to a high-end desktop GPU, despite the latter being older.
- The use of *NVIDIA Nsight* to record detailed frame timings has a performance penalty, hence real-world performance without a debugger attached is slightly better, as shown in the simple FPS measurements in the following section.

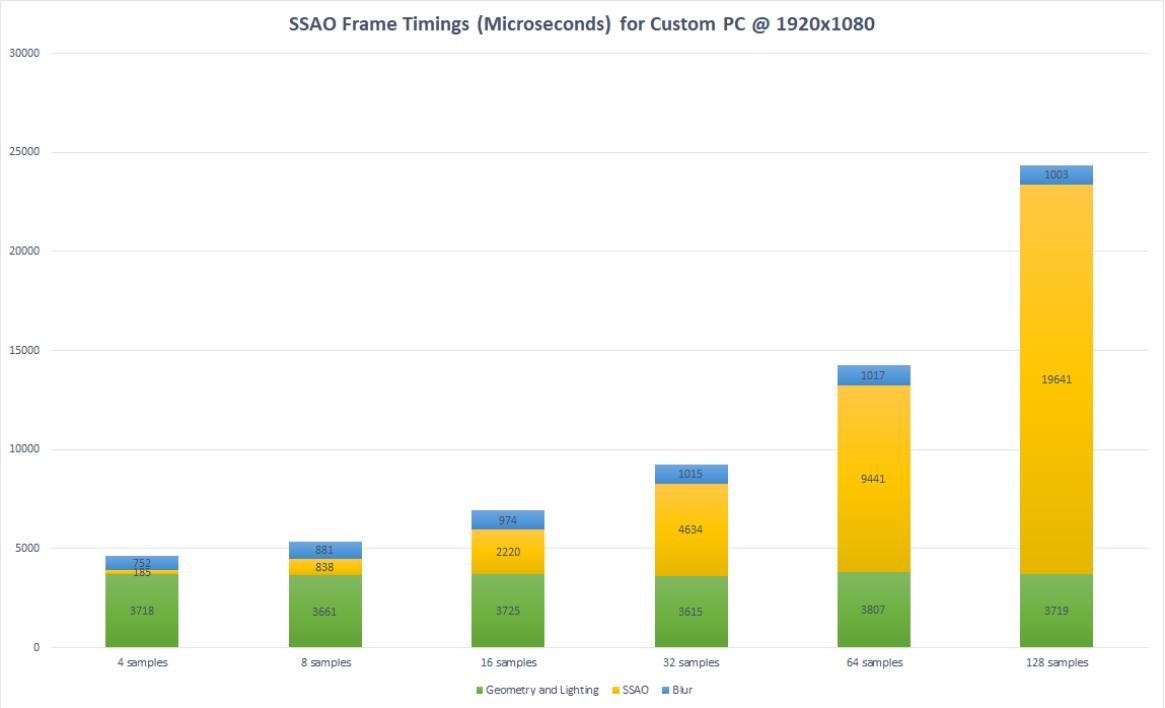



**Figure 4.1** Chart showing frame timing results for our 2011 Custom PC.

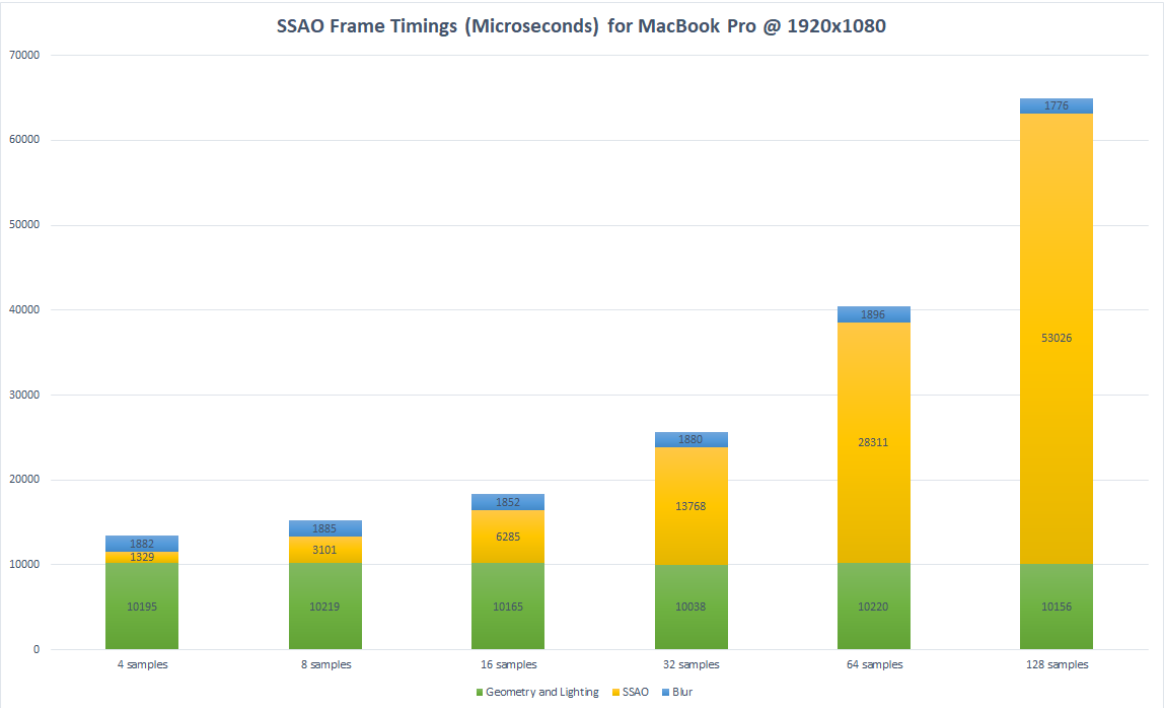



**Figure 4.2** Chart showing frame timing results for our 2013 *MacBook Pro*.

### 4.1.2 Frames Per Second

Figure 4.3 shows how FPS compares between our two PCs. We decided to try two screen resolutions on the *MacBook Pro* - its native "Retina" 2880x1800 resolution, as well as the 1920x1080 resoultion we are using for the Custom PC. Here, the screen-space nature of the algorithm is very evident as the increased number of pixels has a direct impact on SSAO performance. Using the high resolution on the *MacBook Pro* for SSAO rendering is not a very realistic option, as the performance suffers greatly, however 1920x1080 is a commonly-used resolution for gaming and the results for the lower sample counts are very good.

On the Custom PC, using any number of samples between 4 and 64 inclusive results in a fast and fluid experience, with the FPS remaining well above the 60 FPS (monitor refresh rate) target. With the *MacBook Pro*, staying within 4 and 8 samples yields the best results, although using 16 samples still gets us very close to the target at 57.5 FPS.

Given that the *MacBook Pro* is not a machine intended for gaming, its performance in this test is more than acceptable. What is most important is that the Custom PC, despite having an older-generation GPU, yields excellent performance, with SSAO still leaving a lot of headroom for further rendering effects and other game engine tasks. We are confident that newer GPUs will have no issues dealing with SSAO at higher resolutions and sample counts.

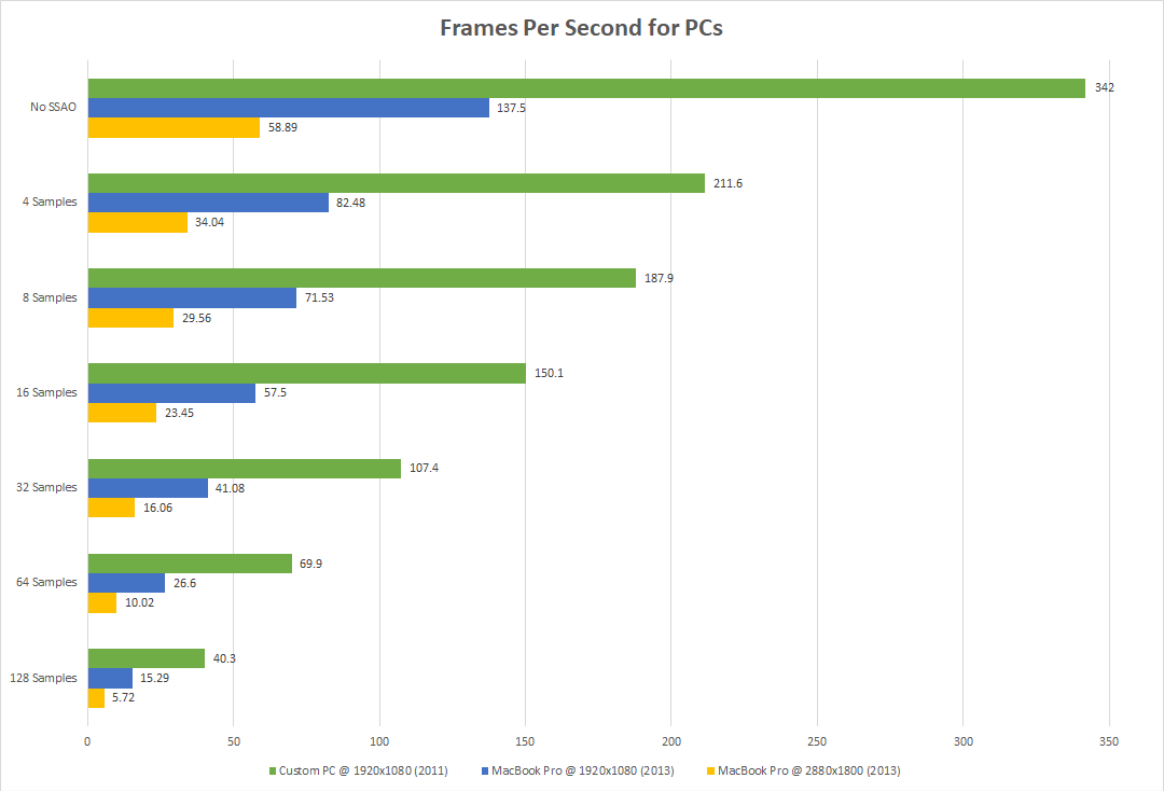



**Figure 4.3** Chart showing frames-per-second results for our PCs.

## 4.2 Smartphone/Tablet-Class Hardware

For the smartphone/tablet tests, we were only able to obtain frame timing measurements for one device using a GPU debugger. The *OnePlus 3*, released in early 2016, is considered a high-end smartphone and contains a very capable GPU for such a device - the *Adreno 530* from Qualcomm.

### 4.2.1 Frame Timings

The smartphone's frame timing pattern looks similar to what we saw earlier on the PCs. Interestingly, there is only a small relative increase in the SSAO stage duration between 4 and 8 samples. It is difficult to say whether this is error in measurement (the overheads of attaching a debugger) or perhaps there is the implicit overhead of having a deferred renderer instead of a simpler forward renderer.

Nevertheless, the results are very impressive for a smartphone, given that the device's GPU only has a fraction of the power and memory bandwidth of a PC. We notice the same relatively high duration of the blur stage as we saw on the PC, so an optimised blur algorithm could be helpful again.

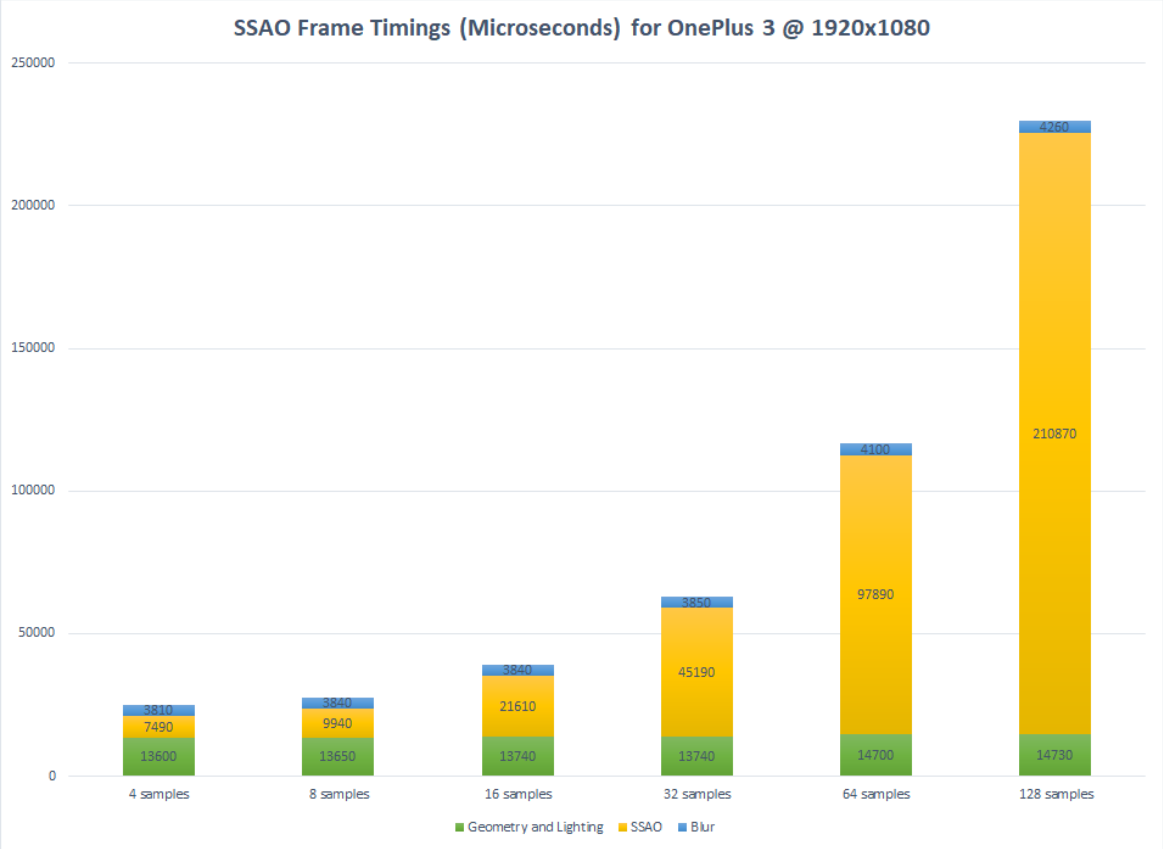



**Figure 4.4** Chart showing frame timing results for our *OnePlus 3* smartphone.

### 4.2.2 Frames Per Second

For our FPS test, we had access to a variety of new and previous-generation smartphones, and a tablet.

An interesting problem with the latest smartphones is that they often come equipped with a very high resolution screen, to improve the user's text-reading experience on physically-small devices. This poses a problem for rendering 3D graphics, as we have several thousands more pixels to compute than in earlier devices. The *Galaxy S6 Edge* is an example of a device whose rendering performance suffers as a result of its high screen resolution, as can be sen in Figure 4.5.

The *HTC 10* is a device whose manufacturer appears to have acknowledged this problem, and provided a workaround. Traditionally, smartphones do not have settings available to the user for lowering their screen resolution as found on PCs. The *HTC 10* comes with an app called *Boost+*, which features an option to lower the screen resolution to 1920x1080 when playing games.

We decided to test this option with our SSAO renderer, and as we can see in Figure 4.5, the *HTC 10* benefits from a significant speed increase after lowering the screen resolution. This echoes our findings when we compared higher and lower resolutions on the *MacBook Pro*.

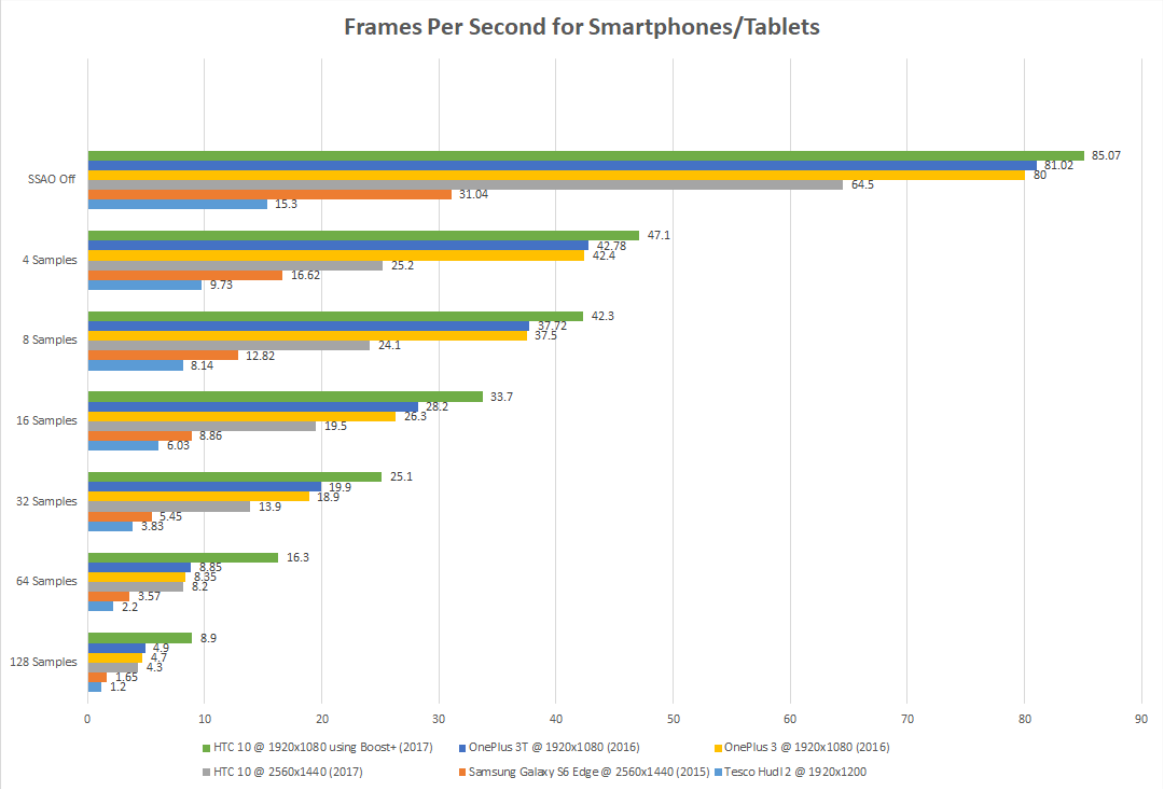



**Figure 4.5** Chart showing frames-per-second results for our smartphones/tablets.

The *Tesco Hudl 2* is a budget tablet from 2015, and its performance is poor even without SSAO enabled, which might be explained by its GPU architecture not being designed with deferred rendering in mind. It is one of the earliest devices to support *OpenGL ES* 3.0. However, its results serve to show the dramatic improvement in mobile GPU technology between 2015 and the present day.

None of the sample kernel settings allow any of our tested mobile devices to achieve over 60 FPS with SSAO enabled at the present time, but overall, the results are very

promising for future devices. 40 FPS remains an arguably acceptable frame rate, especially for smartphone-based games, so the 60 FPS target should be achievable within the next few generations of mobile GPU.

## 4.3 Visual Quality

Figure 4.6 shows the effect of SSAO on an irregular object, using 4 and 16 samples. The soft shadows in both cases offer a pleasant additional sense of depth to the image, and the concave areas of the lion object are shaded darker as we would expect. There is a clear distinction between the wall and the lion, and the arch near the top of the scene has gained a shadow in the corner.

The difference between 4 and 16 samples is subtle at first, but looking closely, we see that the shadows in the 4-sample image appear more solid and less gradual compared to the smooth fade of the 16-sample shadows. Ths is a consequence of undersampling, which causes banding artifacts, however, improving how the 4 samples are distributed will improve the appearance.

**(a)** No SSAO **(b)** 4 samples **(c)** 16 samples

**Figure 4.6** Comparison of SSAO being applied to an irregular object.

Figure 4.7 shows the isolated SSAO component after blurring for 4 and 16 samples. The undersampling artifacts are now much more obvious, especially on the surfaces of the arches. This is likely to be a result of one of the samples being very close to

the base of the hemisphere, and rounding error of the depth buffer results in the pixel being treated as occluded. This could be alleviated by carefully choosing the sample positions or tuning the random sample generator. With 16 samples, the likelihood of this occurring decreases, and the quality of the output is improved.

In Figure 4.8, we see the overall impact that SSAO has on the scene when looking across the atrium of the Sponza. Of particular note are the soft shadows added underneath the curtains and hanging tapestries, behind the planters, and under the arches.

Further images can be found in Appendix A which show how the quality of SSAO changes with the other sample counts. At 16 samples, the banding and noise artifacts become almost unnoticeable and it becomes very difficult to tell the differences between the higher sample counts. We find that 8 samples offers a good level of quality with our sample generator whilst maintaining good performance. 4 samples can be acceptable if the samples are well chosen.

## 4.4 Summary

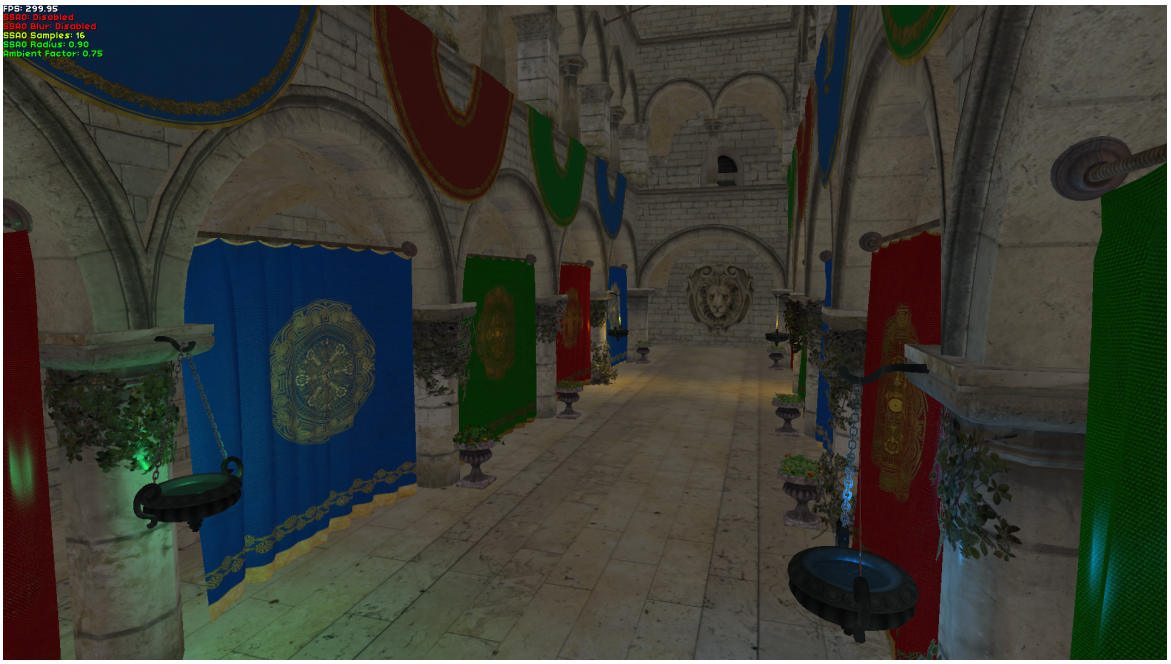
In Section 1.2, we hoped that SSAO would have a significant improvement in visual quality with a low performance impact. Our results suggest that our hypothesis has held true, especially for hardware designed with video game performance in mind. We are especially encouraged by the good performance on the latest smartphones. With some ideas in mind about how we might improve the performance, we are confident that SSAO is a technique well-suited for many platforms, including low-power handhelds, and the predictable, linear performance scaling allows it to be tuned to suit the capabilities of the taget platform's GPU.

We propose that an SSAO kernel of 8 samples on desktop PC-class hardware is a possible "sweet spot" for SSAO, providing a good balance of visual quality and performance. For those with high-end GPUs, 16 samples or higher might be implemented as an "ultra detail" setting. On laptops or smartphones, 4 samples would make the most sense with the current levels of performance we are seeing, however, if the samples are chosen carefully and well distributed, the results can be difficult to distinguish from higher sample counts. This is especially true given the lower physical screen size of portable devices.

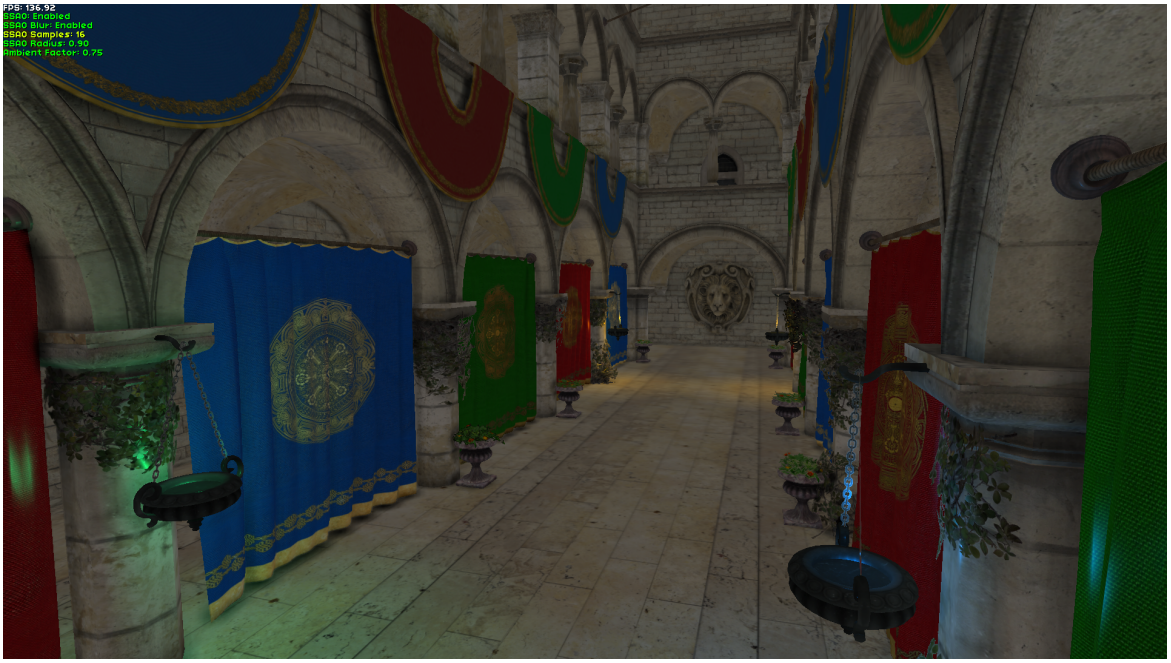
**(a)** Blurred SSAO pass using 4 samples

**(b)** Blurred SSAO pass using 16 samples

**Figure 4.7** Comparison of SSAO passes at 4 and 16 samples.




**(a)** SSAO disabled




**(b)** SSAO using 16 samples

**Figure 4.8** Comparison of complete atrium scene with and without SSAO.

# Chapter 5

# Conclusions

In this chapter, we will bring the dissertation to a conclusion and reflect on what we have learned as a result of carrying out this project. We also suggest some areas to investigate for future researchers wanting to continue from where we left off.

## 5.1 Revisiting Our Original Goals

Our aim was to determine the suitability of a screen-space secondary lighting algorithm for use in video games, by implementing it, profiling it, and commenting on how it improves the visual fidelity of the rendered output.

We believe we have managed to stay true to this aim throughout the project. Our graphics framework was successfully realised in the form of a mock game engine using the programming language preferred by many programmers within the games industry. We have successfully implemented the algorithm we chose, and shown that SSAO is an excellent choice for adding extra detail and depth to a scene - largely because of its predictable performance cost and independance of the complexity of the geometry being rendered.

We were successful in bringing the test framework up on a second, less powerful platform, and were able to profile it on a reasonably good selection of smartphones. It would have been interesting to see the performance on state-of-the-art PC-class hardware, but the author was unable to find suitable hardware in the time available.

The impact of our implementation on the rendering pipeline was successfully measured on the hardware at our disposal, and were able to test the most important

tuning parameter (sample count). It would have been interesting to see if other parameters could have any performance or visual impact, such as sample radius or using other ways to generate sample sets.

## 5.2 What We Have Learned

In this project, we learned a lot about the inner workings of a deferred rendering pipeline and why they are necessary to implement some of the more advanced rendering techniques seen in video games. We learned about the idea of screen-space computations and why they offer us predictable, linear-scaling performance.

We found the implementation of the algorithm to be challenging at first, but would recommend it to any graphics programmer wanting to progress from forward rendering to a more advanced deferred rendering pipeline, as it offers a reasonable learning curve and instantly appreciable results. We enjoyed excellent visual quality of the output and very encouraging performance numbers in our tests.

The most surprising and encouraging results we garnered from our investigation are those of the smartphones. We did not expect the smartphones to perform as well as they did due to their less powerful GPUs, energy-conscious CPUs, lower memory bandwidth, and lower execution unit counts.

In 2007, Shanmugam and Arikan said: “The results show that our algorithm is best suited for the upcoming and future hardware; we experience a worst-case frame-rate of around 17 fps in a GeForce 8800 GTX which maps to about 3 FPS on a GeForce 7950 GX2.”. At the time, they were testing their algorithm on state-of-the-art desktop PC hardware. Ten years later, we are observing very similar performance numbers on a handheld smartphone, which is an exciting prospect and certainly puts the advancements in GPU technology into perspective. For mobile devices, we would like to echo the sentiments of Shanmugam and Arikan and suggest that SSAO is best suited for current high-end devices and future devices.

## 5.3 Possible Improvements

In the previous chapter, we touched on the possibility of our choice of blur algorithm being inefficient compared to the SSAO itself. If we were to revisit the project, we would certainly want to compare alternatives, such as a separable Gaussian blur algorithm.

This type of algorithm reduces computation complexity by performing a blur in one axis in one pass, then blurring the other axis in a separate pass.

We also believe that careful choice of samples can greatly improve the quality of the output for low sample counts, which is important for the devices which only perform well at low sample counts. It may be a good idea to try hand-picking 4 hemisphere samples instead of random generation.

It should be possible to reduce the GPU memory bandwidth consumed by the renderer and increase performance by reducing the reliance on data in the G-buffer, and in turn reducing the G-buffer's size. For example, Pettineo, 2009 shows how one can reconstruct a position in 3D space from existing depth data, removing the need to have a position G-buffer.

## 5.4 Future Work

We propose the following suggestions for future work in the areas of SSAO and other screen-space secondary lighting algorithms:

- Extend testing to other video gaming platforms such as consoles (*Xbox One*, *PlayStation 4*, *Nintendo Switch*), and other smartphone/tablet platforms (*iOS*).
- Investigate performance on more current and upcoming GPUs, and look at visual quality on 4K displays, or virtual reality headsets.
- Explore better/more efficient blur methods (separable Gaussian).
- Look into more sophisticated ambient occlusion effects (horizon-based ambient occlusion).
- Investigate the implementation of other screen-space global illumination algorithms such as those that approximate additional light "bounces".

# Glossary

**2D/3D** Two-dimensional, three-dimensional.

**Albedo colour** The colour of a surface or object before lighting calculations are applied to it. Represents the object as if it were at full brightness. Sometimes also known as the *diffuse colour*.

**Ambient occlusion** The process of calculating the amount of ambient light that a point in 3D space is exposed to.

**Application Programming Interface/API** A standard set of functions and data formats defining an interface through which a programmer can access the functionality provided by a software library or piece of middleware.

**Baking** The process of encoding extra data (usually associated with lighting calculations) alongside the geometry data it applies to.

**Clip-space** The coordinate system in which all objects that remain inside it after clipping will be visible on screen.

**CPU** Central processing unit

**Culling** The process of discarding unnecessary geometry data, usually because it has been determined to outside the view frustum, or hidden behind other objects.

**Deferred rendering** A type of rendering in which lighting calculations are decoupled from the geometry transformation stage(s), and left until a later stage in the pipeline.

**Depth buffer** A memory buffer (usually in GPU memory) which holds depth data.

**Depth data** The depth ($z$) coordinate of the currently-frontmost fragment at a given screen-space location.

**Depth testing** The process of comparing a fragment's depth data to that of the current value for this screen-space position in the depth buffer. If the new fragment is found to have a higher depth value at that position, it is hidden behind the current fragment and can be discarded.

**Direct light** Light which travels directly from a light source to a point or surace.

**Dot product** A mathematical operation involving two vectors which returns a scalar value. Also known as the *scalar product*.

**Double buffering** The technique of alternately drawing graphics to two framebuffers, and swapping them in synchronisation with the monitor or television's refresh rate.

**Forward rendering** A simple type of rendering pipeline in which vertex data moves through it from start to finish in one pass - directly forward.

**Fragment** A potentially visible piece of piece of rasterised geometry which may end up becoming a visible pixel on the screen.

**Fragment shader** A program running on the GPU which processes and gives colour to fragments.

**Framebuffer** A memory buffer which is large enough to hold an entire rendered frame of graphics.

**Game engine** A piece of software which contains all of the subsystems required to support the implementation of a game. Typically architectured such that it an be reused for more than one game.

**Geometry** The data representation of one or more objects in the form of 3D polygons.

**Geometry buffer/G-buffer** A collection of buffers that store the results of a complete geometry transformation pass in memory so that they can be quickly looked-up later in lighting calculations.

**Global illumination** The family of algorithms that help simulate the physical behaviours of light.

**GPU** Graphics Processing Unit, a piece of hardware specialising in the fast rendering of 3D graphics.

**Hardware acceleration** The use of specialist or bespoke hardware to speed up complex computations.

**IBM PC** International Business Machines Personal Computer.

**Middleware** Additional software, usually provided by a third-party, which lies between our own software and the operating system and provides useful functionality.

**Model-space** The coordinate system in which 3D objects are usually designed, with their Cartesian origin at the centre of the object.

**Model/view/world matrix** The mathematical matrices that allow the transformation of a set of coordinates from one coordinate system into another.

**Normal data** A vector which is perpendicular to a point or surface.

**Normalised device coordinates** The coordinate system in which all coordinates lie between $-1$ to 1 in all axes. Obtained by applying the perspective divide to coordinates in clip-space.

**Occlusion factor** A number between 0 to 1 representing how much ambient light has been blocked from reaching a point or surface.

**Orthographic projection** A type of projection in which the distance of an object from the viewing position has no effect on the object's perceived size.

**Perspective projection** A type of projection in which objects appear to be smaller the further away they are from the viewing position.

**Phong shading** A type of light surface shading model in which light reflection angles are computed to determine the brightness of a point on a surface, based on ambient light, the diffuse colour of the object, and the specular highlights on the object (its "shininess").

**Pixels** Picture elements, the smallest addressable units in a raster image, usually a single coloured "dot" within the image.

**Quad** A rectangular graphics construct, usually made up of two triangles.

**Rasterisation** The process of converting 3D objects into arrays of shaded pixels suitable for viewing on the 2D surface of a monitor or television.

**Raytracing** An expensive, yet more physically accurate type of rendering in which "rays" of light are followed through the scene from the viewing position, and their interaction with objects in the 3D world is simulated as they bounce from one surface to another.

**Render farm** A large group of high-performance computers dedicated to the task of rendering frames of computer-generated graphics in film production.

**Rendering pipeline** A sequence of rendering stages in which data is acted on, then passed to the next stage for further processing.

**Sample kernel** An array of randomly-generated 3D points around a unit sphere or hemisphere, used for sampling positions around a point in 3D space to determine an approximate ambient occlusion factor.

**Screen-space** The domain in which geometry data guaranteed to be associated with the final screen output exists.

**Screen-space ambient occlusion** The process of calculating an approximation of ambient occlusion using only data in screen-space.

**Secondary lighting** Light which has experienced at least one reflection or bounce, and has not travelled directly from the source to a point or surface (also known as *indirect lighting*).

**Shadow mapping** The technique of generating shadows by rendering the scene from the point of view of a light source, and mapping the silhouette of an object onto a texture which can be used to simulate a shadow being cast by the object.

**Tangent-space** The coordinate system in which positions are relative to the surface of an object.

**Tangent/bitangent** In tangent space, the axes which define the two-dimensional surface plane.

**TBN matrix** A mathematical matrix used for transforming a position from tangent space to some other coordinate system.

**Transforms** Operations performed on vertices to relocate them or move them from one coordinate system to another. Transforms can include translation, rotation, and scaling operations.

**Undersampling** An undesirable effect caused when insufficient sampling has taken place.

**Vertex shader** A program running on the GPU which processes and transforms geometry data (vertices).

**Vertices** Positions or points in 2D or 3D space, usually represented by their $x$, $y$, and $z$ coordinates.

**View frustum** A volume representing the region of interest (visible region), defined by a projection matrix.

**View-space** The coordinate system in which objects are positioned relative to where they are being viewed from.

**World space** The coordinate system in which objects are positioned where they are located within the 3D world.

# References


Chapman, J. (2011). SSAO Tutorial. Retrieved November 5, 2016, from http://john-chapman-graphics.blogspot.co.uk/2013/01/ssao-tutorial.html

Christensen, P. H. (2002, April). *PhotoRealistic RenderMan Application Note #35: Ambient Occlusion, Image-Based Illumination, and Global Illumination*. Pixar.

Filion, D. & McNaughton, R. (2008). StarCraft II: Effects & Techniques. In *ACM SIGGRAPH 2008 Games* (pp. 133–164). SIGGRAPH '08. Los Angeles, California: ACM. doi:10.1145/1404435.1404441

Gartner. (2017). Gartner Says Worldwide Sales of Smartphones Grew 7 Percent in the Fourth Quarter of 2016. Retrieved April 20, 2017, from http://www.gartner.com/newsroom/id/3609817

Google. (2017). Dashboards. Retrieved April 20, 2017, from https://developer.android.com/about/dashboards/index.html

Herlihy, M. & Shavit, N. (2008). The art of multiprocessor programming. (p. 14). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Leather, A. (2009, December). Crysis - did you upgrade? Retrieved April 19, 2017, from https://www.bit-tech.net/blog/2009/12/15/crysis-did-you-upgrade/

McDonald, E. (2017). The Global Games Market Will Reach $108.9 Billion In 2017 With Mobile Taking 42%. Retrieved April 20, 2017, from https://newzoo.com/insights/articles/the-global-games-market-will-reach-108-9-billion-in-2017-with-mobile-taking-42/

McGuire, M. (2011). Computer Graphics Archive. Retrieved April 10, 2017, from http://graphics.cs.williams.edu/data

Miller, G. (1994). Efficient algorithms for local and global accessibility shading. In *Proceedings of the 21st annual conference on computer graphics and interactive techniques* (pp. 319–326). SIGGRAPH '94. New York, NY, USA: ACM. doi:10.1145/192161.192244

Mittring, M. (2007). Finding next gen: CryEngine 2. In *ACM SIGGRAPH 2007 courses* (pp. 97–121). ACM.

NVIDIA. (2017). Compare 10 series graphics cards. Retrieved April 19, 2017, from https://www.nvidia.com/en-us/geforce/products/10series/compare/

NVIDIA. (2013). Enabling ambient occlusion in games. Retrieved April 21, 2017, from http://www.geforce.com/whats-new/guides/ambient-occlusion#1

Owens, B. (2013). Forward rendering vs. deferred rendering. Retrieved April 21, 2017, from https://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342

Pettineo, M. (2009). Scintillating snippets: Reconstructing position from depth. Retrieved April 29, 2017, from https://mynameismjp.wordpress.com/2009/03/10/reconstructing-position-from-depth

Pharr, M. & Humphreys, G. (2010). *Physically Based Rendering, Second Edition: From Theory To Implementation* (2nd). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Phong, B. T. (1975). Illumination for computer generated pictures. *Commun. ACM*, *18*(6), 311–317. doi:10.1145/360825.360839

Shanmugam, P. & Arikan, O. (2007). Hardware accelerated ambient occlusion techniques on GPUs. In *Proceedings of the 2007 symposium on interactive 3D graphics and games* (pp. 73–80). I3D '07. Seattle, Washington: ACM. doi:10.1145/1230100.1230113

Shopf, J. (2008, February). 2007: The year SSAO broke. Retrieved April 19, 2017, from https://levelofdetail.wordpress.com/2008/02/10/2007-the-year-ssao-broke/

Steam. (2017). Steam Hardware & Software Survey: March 2017. Retrieved April 20, 2017, from http://store.steampowered.com/hwsurvey/

# Appendix A

# Screenshots

The following pages contain some larger, landscape screenshots from our SSAO renderer demonstration application.

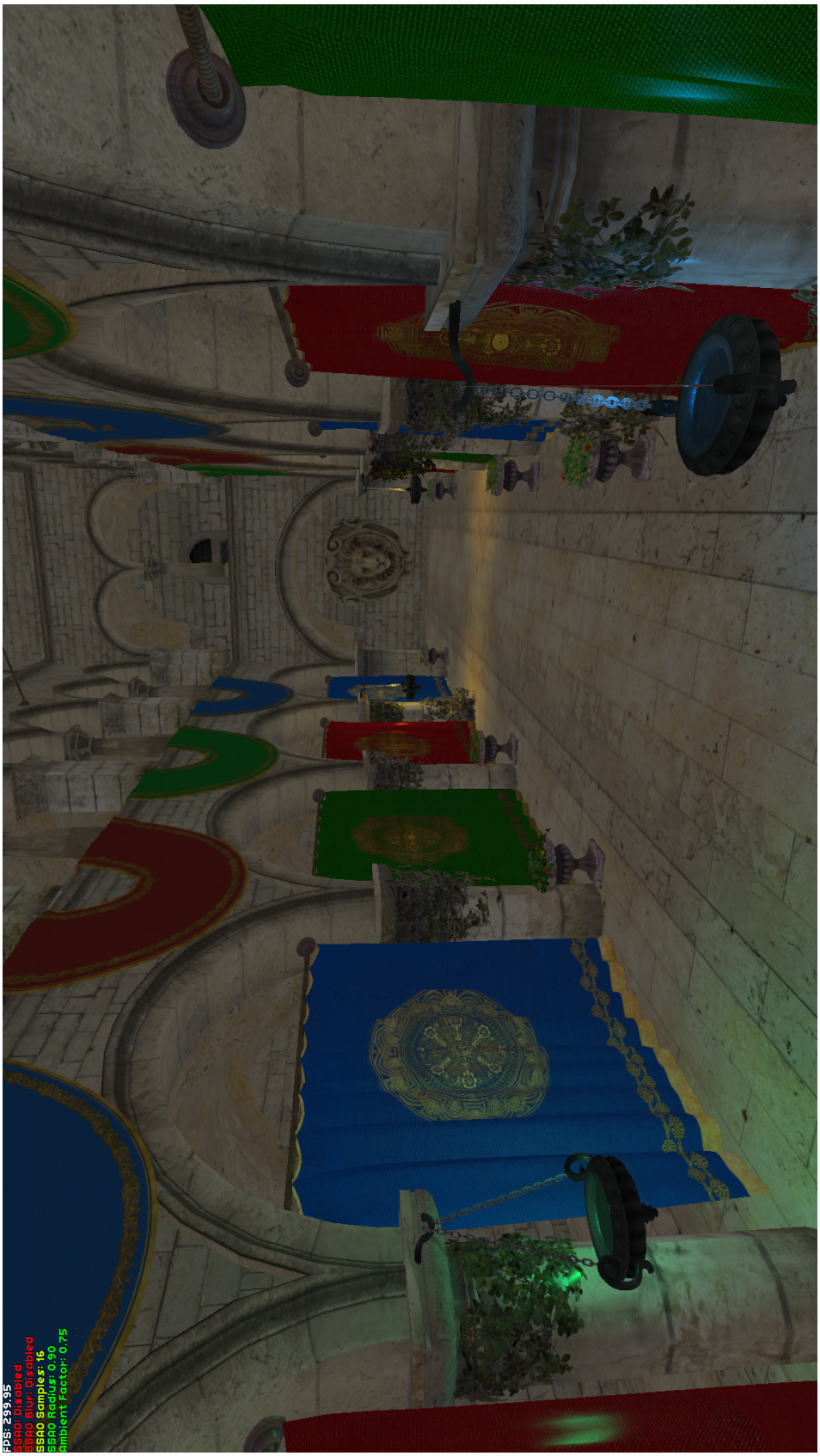



**Figure A.1** Screenshot of our SSAO demo application, with SSAO disabled.

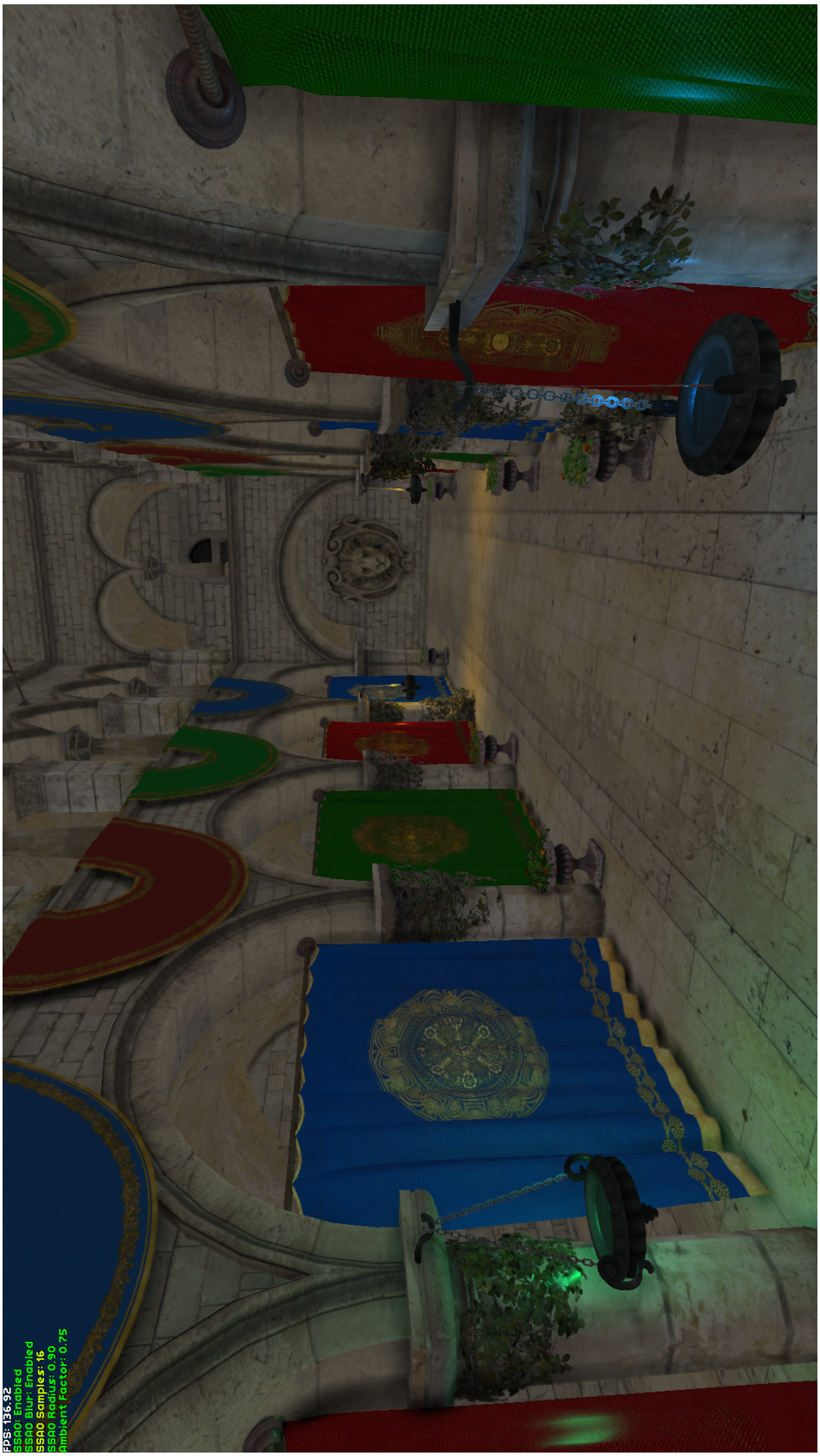



**Figure A.2** As in Figure A.2, but with 16-sample SSAO enabled.

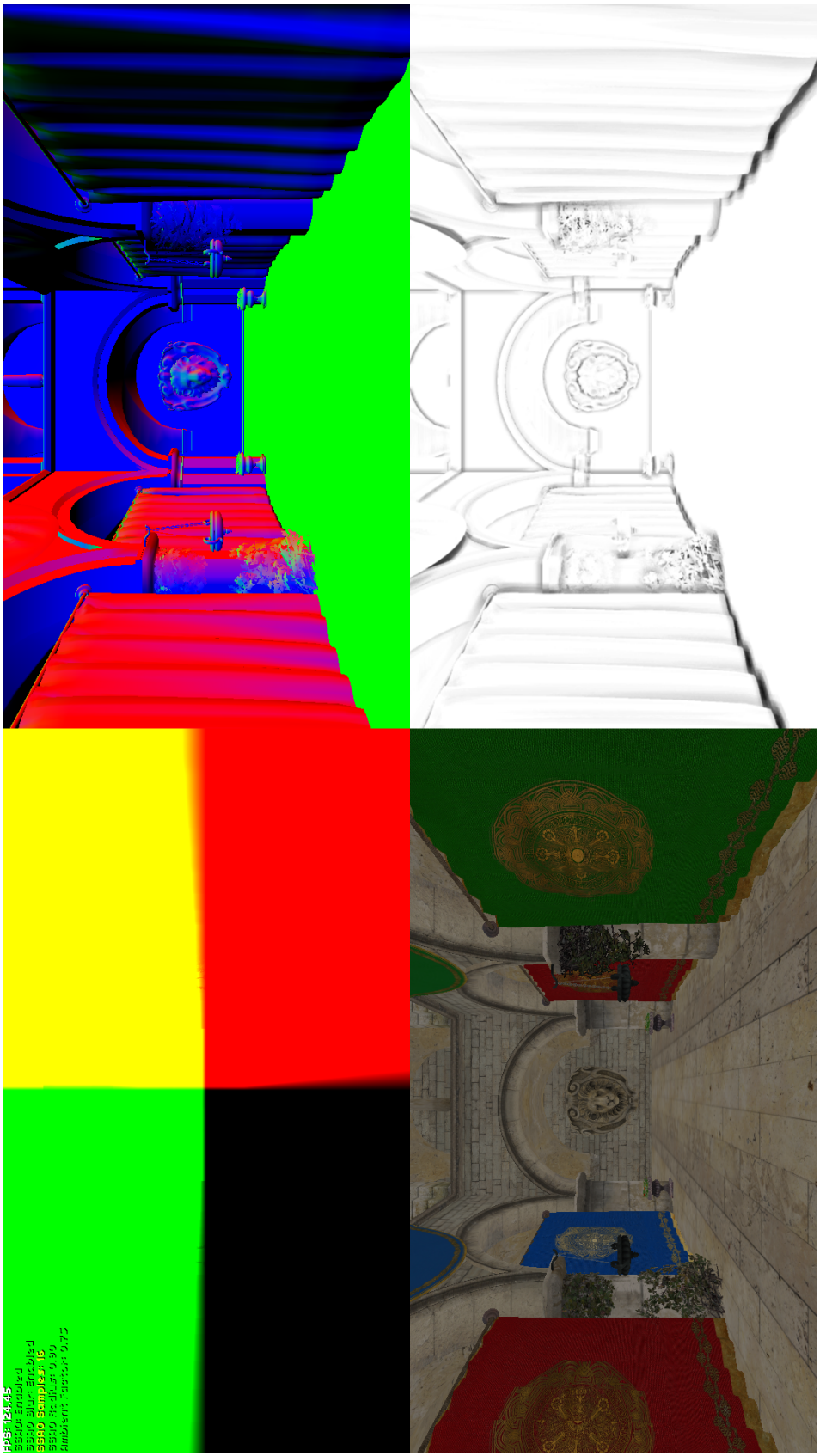



**Figure A.3** The G-buffer visualisation feature showing (clockwise from top left, with image correctly oriented) the position buffer, normal buffer, SSAO pass, and albedo buffer.

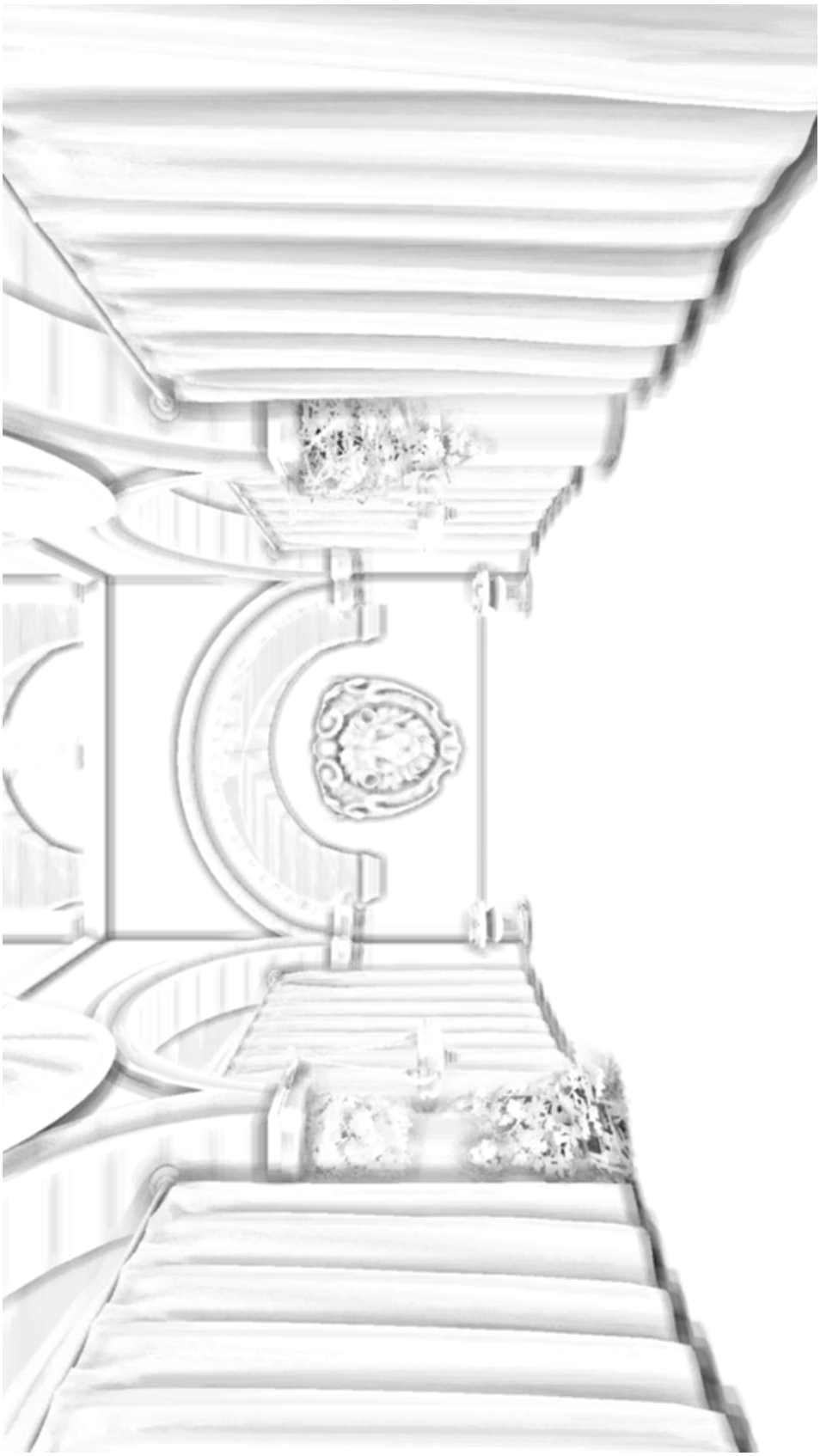


**Figure A.4** SSAO pass - 4 samples. Note the "banding" around the arches due to undersampling.

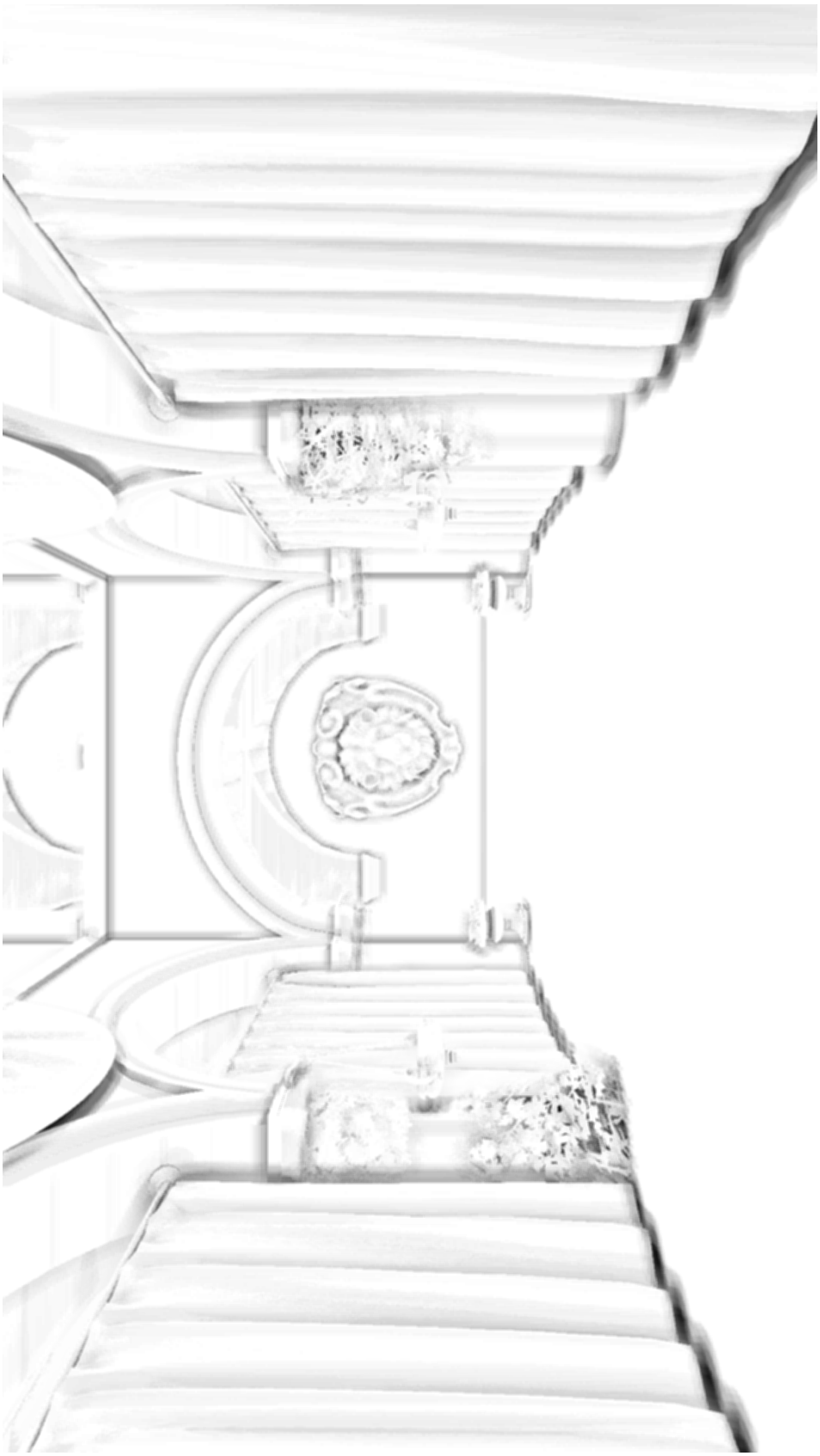


**Figure A.5** SSAO pass - 8 samples. Banding is still present, but greatly reduced.

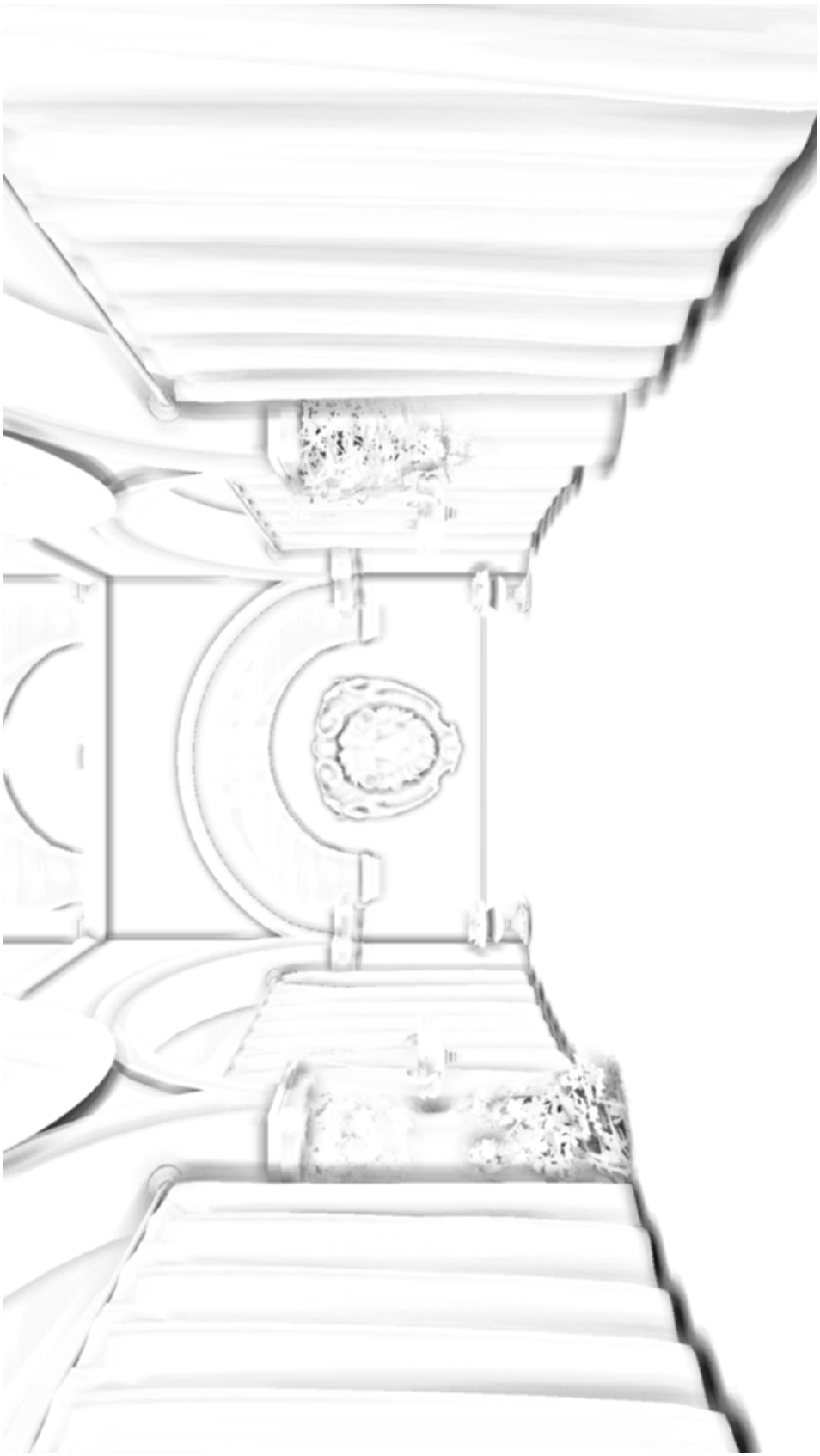


**Figure A.6** SSAO pass - 16 samples. Banding has almost disappeared.

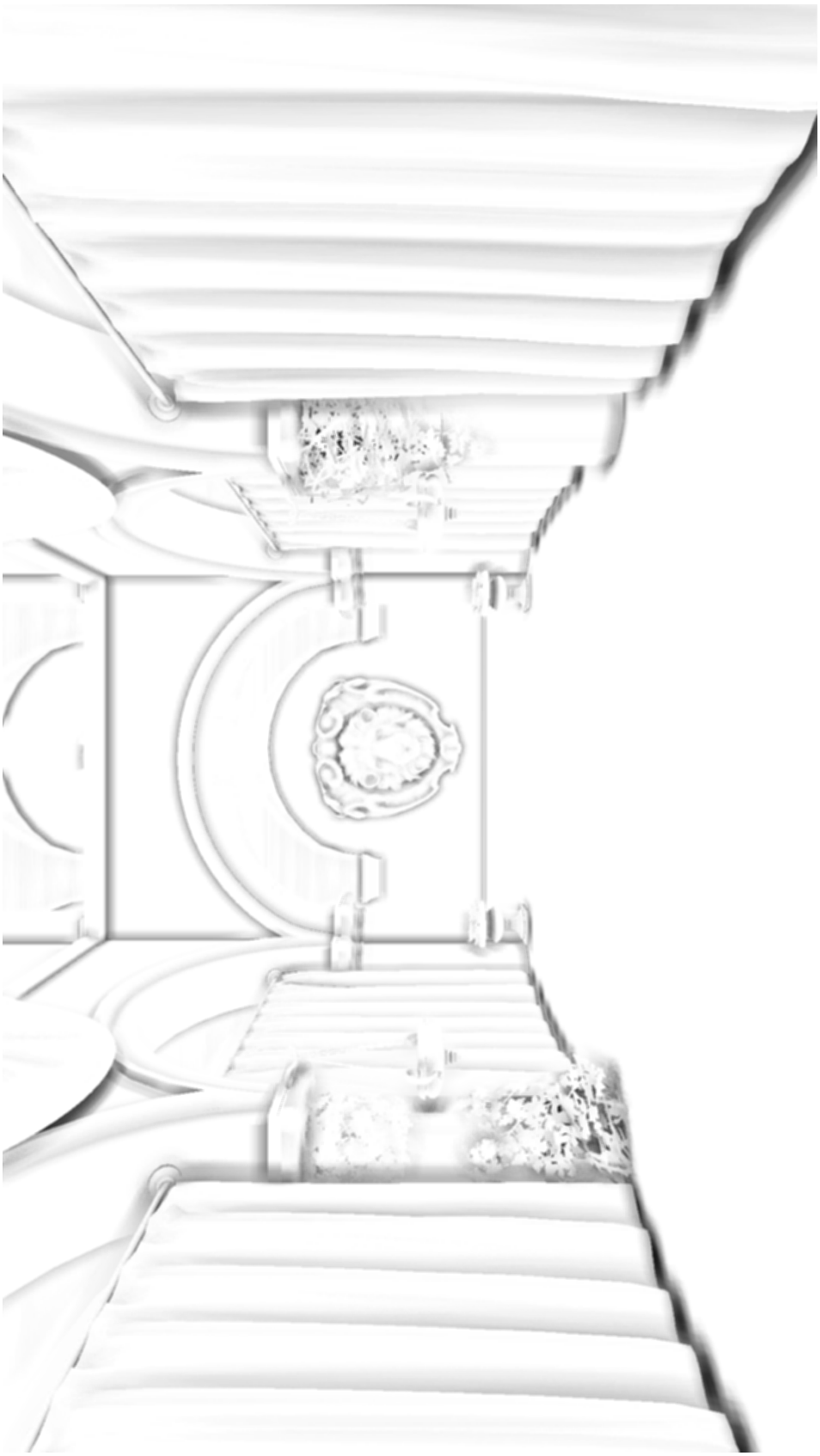


**Figure A.7** SSAO pass - 32 samples. The differences between this and the next two sample counts are now much less perceivable.

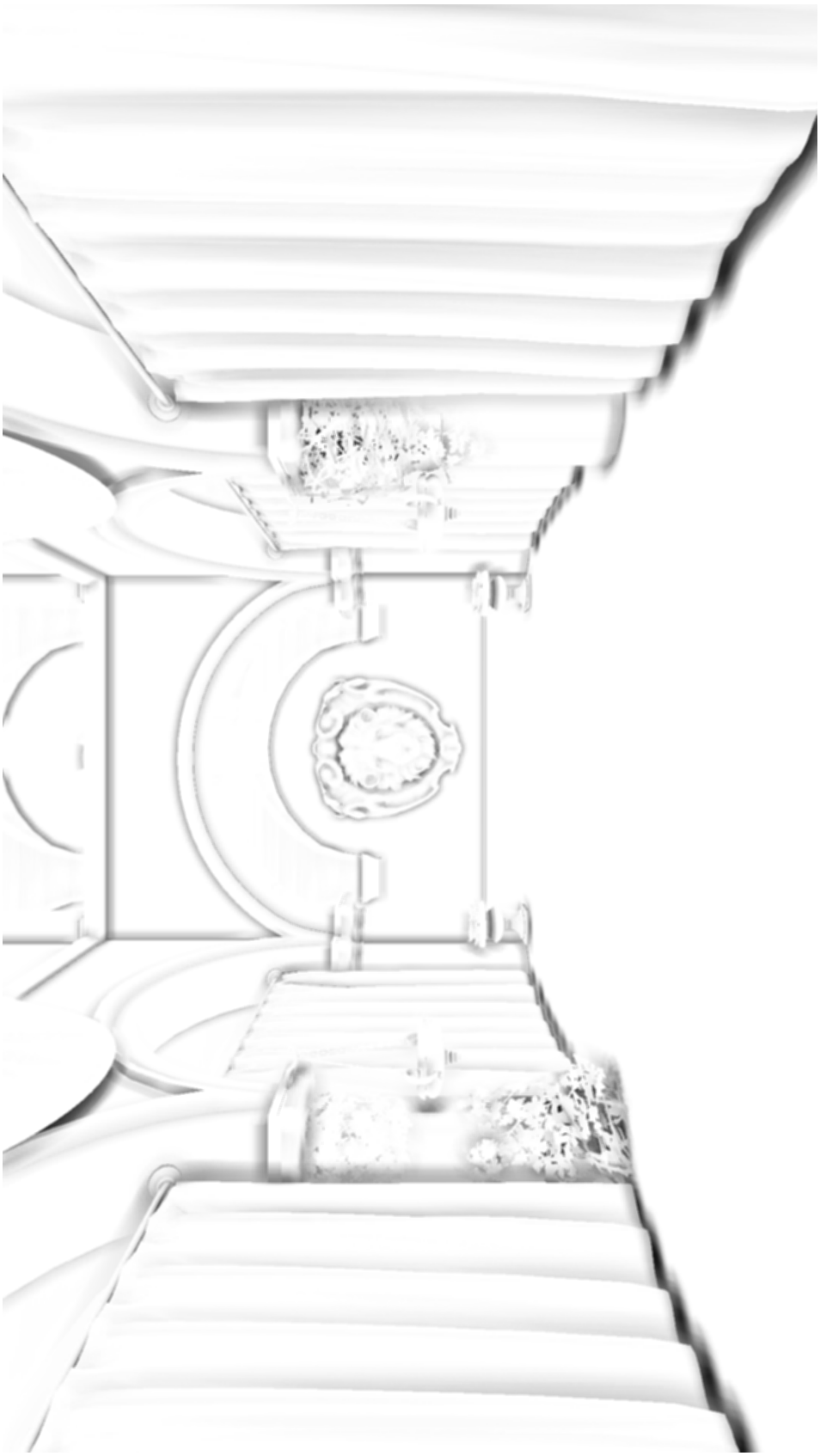


**Figure A.8** SSAO pass - 64 samples.

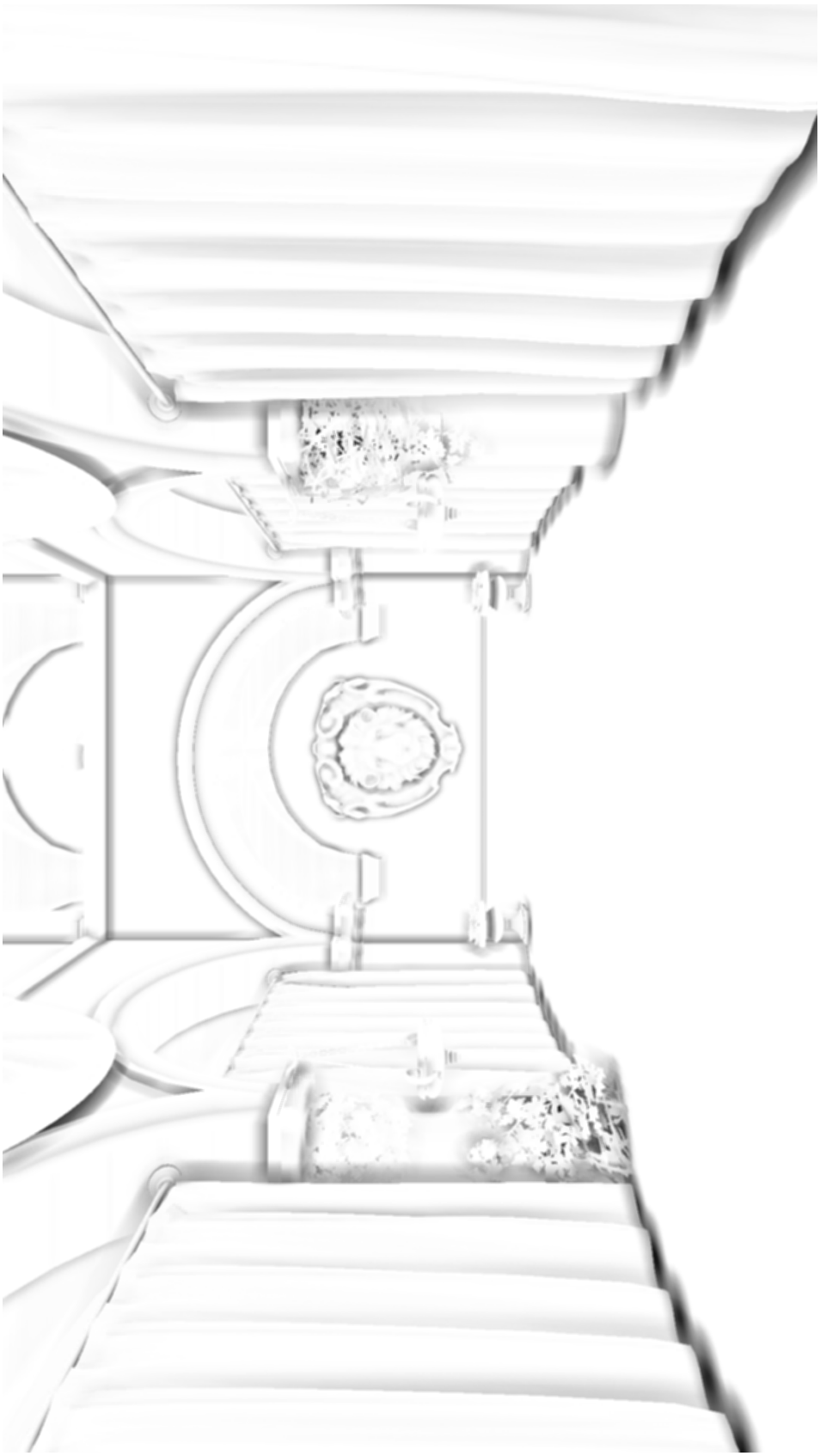


**Figure A.9** SSAO pass - 128 samples.

# Appendix B

# Pseudocode

The following pseudocode listings show our implementation of the SSAO and blur passes within our deferred rendering pipeline. In our project, they are implemented as *GLSL* fragment shaders.

## B.1 Computing Hemispherical SSAO

```
Inputs:  GBufferPositions[][]
         GBufferNormals[][]
         SSAOSampleKernel[]
         RotationTexture[]
         SphereRadius
         ProjectionMatrix

Outputs: SSAOTexturePixels[][]

Procedure SSAOHemispherical:
{
    ForEach Pixel in SSAOTexturePixels:
    {
        // Retrieve our view space positions, normals and a random
        // rotation vector from their respective textures
        Let viewPosition = GBufferPositions[Pixel.x][Pixel.y];
        Let viewNormal = GBufferNormals[Pixel.x][Pixel.y];
        Let randomVector = RotationTexture[Pixel.x modulo RotationTexture.width]
                                          [Pixel.y modulo RotationTexture.height];

        // Construct a TBN matrix (Gram-Schmidt process) to re-orient hemisphere
        // samples with respect to view space normal, and also apply the random
        // rotation from our noise texture
        Let tangent = randomVector -
                      viewNormal * DotProduct(randomVector, viewNormal);
        Let bitangent = CrossProduct(viewNormal, tangent);
        Let tbnMatrix = Matrix3x3(tangent, bitangent, normal);

        // Accumulate occlusion samples
        Let occlusionFactor = 0.0;
        ForEach SSAOSample in SSAOSampleKernel:
        {
            // Reorient sample
            Let reorientedSample = tbnMatrix * SSAOSample;

            // Move it relative to our view space position
            Let reorientedSample = viewPosition + reorientedSample * SphereRadius;

            // Take sample from view space to clip space (project it)
            Let gBufOffset = ProjectionMatrix * Vector4(reorientedSample.xyz, 1.0);

            // Apply perspective divide
            Let gBufOffset.xyz = gBufOffset.xyz / gBufOffset.w;

            // After projection, x/y/z range is between -1 and 1
            // (centre of view frustum is {0, 0, 0})
            // Transform range to 0 to 1 to put it into texture coordinate range
            Let gBufOffset.xyz = gBufOffset.xyz * 0.5 + 0.5;

            // Now get the depth component at this sample position
            Let sampleDepth = GBufferPositions[gBufOffset.x][gBufOffset.y].z;

            // Positive z is deeper into the world
```

```
            If sampleDepth >= reorientedSample.z:
            {
                // Sample is inside geometry; increase occlusion factor
                Let occlusionFactor = occlusionFactor + 1.0;
            }
        }

        // Occlusion factor becomes a fraction of the number of samples
        Let occlusionFactor = occlusionFactor / SSAOSampleKernel.size;

        Let Pixel = 1.0 - occlusionFactor;
    }
}
```

## B.2 Computing Blur

```
Inputs:  TextureCoordinates
         RotationTextureSize
         SSAOTexture[][]

Outputs: BlurTexturePixels[][]

Procedure SimpleBlur:
{
    ForEach Pixel in BlurTexturePixels:
    {
        // We blur around the same area covered by our SSAO rotation texture
        // to increase its effectiveness
        Let min = -RotationTextureSize / 2;
        Let max = RotationTextureSize / 2;
        Let texelSize = 1 / SSAOTexture.size;

        // Accumulate and average the neighbouring pixels
        Let result = 0.0;
        For x = min To max:
        {
            For y = min To max:
            {
                Let texelOffset = Vector2(x, y) * texelSize;
                Let result = result + SSAOTexture[TextureCoordinates.x + texelOffset.x]
                                                 [TextureCoordinates.y + texelOffset.y].r;
            }
        }

        Let Pixel = result / (RotationTextureSize * RotationTextureSize);
    }
}
```