NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSc THESIS

Real Time Graphics Rendering Capabilities of Modern Game Engines

Charalampos E. Alisavakis

Supervisors: **Dr Maria Roussou,** Assistant Professor
Dr George Drettakis, Inria Sophia-Antipolis

ATHENS

AUGUST 2018

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Δυνατότητες Απόδοσης Γραφικών Πραγματικού Χρόνου σε Περιβάλλοντα Σύγχρονων Μηχανών Παιχνιδιών

Χαράλαμπος Ε. Αλισαβάκης

Επιβλέποντες: **Δρ. Μαρία Ρούσσου,** Επίκουρη Καθηγήτρια
Δρ. Γιώργος Δρεττάκης, Inria Sophia-Antipolis

ΑΘΗΝΑ

ΑΥΓΟΥΣΤΟΣ 2018

BSc THESIS

Real Time Graphics Rendering Capabilities of Modern Game Engines

Charalampos E. Alisavakis
S.N.: 1115201300004

SUPERVISORS: **Dr Maria Roussou,** Assistant Professor
Dr George Drettakis, Inria Sophia-Antipolis

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Δυνατότητες Απόδοσης Γραφικών Πραγματικού Χρόνου σε Περιβάλλοντα Σύγχρονων Μηχανών Παιχνιδιών

Χαράλαμπος Ε. Αλισαβάκης

Α.Μ.: 1115201300004

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Δρ. Μαρία Ρούσσου,** Επίκουρη Καθηγήτρια
Δρ. Γιώργος Δρεττάκης, Inria Sophia-Antipolis

ABSTRACT

The subject of this thesis pertains to the more popular modern game engines as well as their capabilities concerning the rendering of 3D graphics in real time.

Despite the advances in 3D graphics rendering, modern game engines still need to face the challenge of providing high-fidelity visual results in as little time as possible. In this thesis, I investigate the current real time graphics rendering capabilities of popular game engines along with ways to extend the graphics pipeline of an engine to incorporate custom techniques that can enhance the visual fidelity of a scene. In recent years, there has been rapid development in offline rendering and visualization methods and, along with the development of computer systems, those methods manage to represent the real world with great accuracy. However, when it comes to real time rendering the results can vary, since the process of graphics rendering needs to occur in significantly less time. This thesis examines how modern game engines face this challenge by analyzing their different aspects, including their graphics rendering capabilities, as well as by investigating the ability to implement custom graphics rendering algorithms, in particular screen space ambient occlusion and screen space directional occlusion.

SUBJECT AREA: Computer Graphics

KEYWORDS: graphics rendering, game engines, global illumination, shader programming

ΠΕΡΙΛΗΨΗ

Το αντικείμενο της πτυχιακής εργασίας αφορά στις πιο διαδεδομένες σύγχρονες μηχανές παιχνιδιών καθώς και στις δυνατότητές τους σε ό,τι αφορά την απόδοση τρισδιάστατων γραφικών σε πραγματικό χρόνο.

Παρά τις προόδους στην τεχνολογία απόδοσης τρισδιάστατων γραφικών, οι σύγχρονες μηχανές παιχνιδιών καλούνται ακόμα να αντιμετωπίσουν την πρόκληση του να παρέχουν υψηλής πιστότητας εικαστικό αποτέλεσμα σε όσο το δυνατόν λιγότερο χρόνο. Στα πλαίσια αυτής της πτυχιακής, ερευνώ τις τρέχουσες δυνατότητες απόδοσης γραφικών πραγματικού χρόνου διαδεδομένων μηχανών παιχνιδιών, καθώ και τρόπους να επεκταθεί η σωλήνωση απόδοσης γραφικών μιας μηχανής, ώστε να ενσωματωθούν προσαρμοσμένες τεχνικές οι οποίες μπορούν να ενισχύσουν το εικαστικό αποτέλεσμα μιας σκηνής. Στα πρόσφατα χρόνια, έχει υπάρξει ραγδαία ανάπτυξη σε μεθόδους απόδοσης γραφικών και οπτικοποίησης και, μαζί με την ανάπτυξη των υπολογιστικών συστημάτων, αυτοί οι μέθοδοι συμβάλλουν σημαντικά στην απεικόνιση του πραγματικού κόσμου με μεγάλη πιστότητα. Ωστόσο, σε ό,τι αφορά την απόδοση γραφικών πραγματικού χρόνου, τα αποτελέσματα μπορεί να διαφέρουν, καθώς η διαδικασία της απόδοσης οφείλει να πραγματοποιείται σε σημαντικά λιγότερο χρόνο. Αυτή η πτυχιακή εργασία εξετάζει τον τρόπο που οι σύγχρονες μηχανές παιχνιδιών αντιμετωπίζουν αυτή τη πρόκληση αναλύοντας τις διαφορετικές πτυχές τους, συμπεριλαμβανομένου των δυνατοτήτων απόδοσης γραφικών, καθώς και διερευνώντας το ενδεχομένο να υλοποιήσει κανείς προσαρμοσμένους αλγόριθμους απόδοσης γραφικών, δοκιμάζοντας πιο συγκεκριμένα τις τεχνικές “screen space ambient occlusion” και “screen space directional occlusion”.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Γραφικά Υπολογιστών

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: απόδοση γραφικών, μηχανές παιχνιδιών, έμμεσος φωτισμός, προγραμματισμός φωτοσκιαστών

ACKNOWLEDGMENTS

I would first like to thank my advisor, assistant professor Dr Maria Roussou for her support on the realization of this thesis, her help with making the field of video game development more accessible in the department and all the assistance she has offered me over my years of study, both educationally and professionally.

In addition, I would like to thank Dr George Drettakis for his valuable assistance in the selection of the thesis subject and providing his expertise in the field of computer graphics. Finally, I would also like to thank Dr Theoharis Theoharis for his contribution to the content of the thesis as well as for his much-appreciated introduction to the subject of graphics rendering and visualization.

TABLE OF CONTENTS

TABLE OF FIGURES

1. INTRODUCTION

1.1 Scope of the Thesis

The main subject of this thesis investigates the capabilities of modern game engines as far as real time 3D graphics rendering is concerned. Specifically, on the one hand, some of the more widespread game engines of the most recent years are presented focusing on their general use with emphasis on the field of graphics. On the other hand, the thesis covers the process of incorporating graphics and visualization techniques and algorithms inside the environment of a game engine in order to extend the overall graphics pipeline and to compare the integrated techniques with existing built-in methods.

1.2 Contributions

The thesis is divided in two main parts, each with its respective purpose:

- The first part presents and categorizes some of the more common game engines. The analysis is based on their overall use with an emphasis on the field of graphics rendering and visualization. The aim of this section is to give the reader an introduction to the modern software used in video game development in terms of usability, capabilities and end result.
- The second part demonstrates the process of extending the graphics pipeline by implementing a real time global illumination approximation algorithm in the environment of Unity 3D. The goal of the second part is to establish the extensibility of a game engine's graphics workflow.

1.3 Structure of the Thesis

The structure of the thesis involves two main sections each with its respective purpose:

The first section (Chapter 2) provides a general presentation of specific game engines and their characteristics, in particular Unreal Engine 4, Unity 3D and GameMaker Studio.

The second section (Chapter 3) contains a presentation of the implementation of a real time global illumination technique (screen space directional occlusion) inside the Unity 3D game engine's environment, as well as a comparison of the method with alternative methods that work in real time or via pre-processing. Specifically, it provides an insight to both direct and indirect illumination models (Section 3.2.1) along with a more detailed presentation of the screen space ambient occlusion and screen space directional occlusion algorithms (Section 3.2.2 and Section 3.2.3 respectively). Finally, the section goes through the steps of implementing both screen space effects in Unity 3D (Section 3.3).

2. ANALYSIS OF MODERN GAME ENGINES

2.1 Introduction

This part of the thesis contains a presentation of some of the most widely used video game engines at the time of writing of this document. More specifically, the game engines that are studied are:

- Unreal engine 4
- Unity 3D
- GameMaker Studio

Each game engine's analysis relies on:

- Notable elements of the game engine
- The engine's behavior scripting system
- The engine's capabilities as far as real time graphics rendering is concerned
- Advantages and disadvantages of the engine, in general as well as in comparison with other game engines.

2.2 Theoretical background

2.2.1 Game engines

A game engine can be described as "a framework comprised of a collection of different tools, utilities, and interfaces that hide the low-level details of the various tasks that make up a video game" [1]. In most cases, a game engine is composed of certain subsystems that implement the different aspects of a video game. More specifically, the most common structural elements of a game engine are as follows:

- Main logical core: The subsystem that undertakes the basic logic necessary for the execution of the video game. Additionally, it is usually responsible for the connection and coordination of the rest of the engine's subsystems. The application's behavior is extended through the logical core by the engine's user via scripting.
- Graphics rendering engine: This is the subsystem responsible for the video game's real time graphics rendering. In most cases, graphics rendering engines use one or multiple Application Programming Interfaces (APIs), like Direct3D, OpenGL or Vulkan.
- Audio engine: The audio engine is the subsystem which consists of methods and algorithms for loading, modifying and outputting audio (music and sound effects) through the client's audio system.
- Physics engine: The subsystem which provides functions that can be applied for realistic physics simulations inside the application. Specifically, it can emulate physical forces, collisions or more complex systems, like spring joints.

2.2.2 Popular game engines

In the last few years, game engines have been more and more accessible to the public. While companies used to develop proprietary technology for the design and implementation of interactive applications, nowadays there are organizations or companies that develop game engines whose primary goal is to be used by the community. At the time this thesis is being written, the more widespread game engines that are available to the community are Unreal Engine 4, Unity 3D and GameMaker Studio.

2.2.3 Game engine rendering elements

The rendering capabilities analysis of the game engines includes references to graphics rendering concepts that include:

- Shaders:
  Compute programs that are responsible for the real time rendering of 3D objects and the application of post-process effects. They contain commands that define the visual properties of a material or an effect.

  Those programs can be developed using a shading programming language like CG (C for Graphics), GLSL (OpenGL Shading Language) or HLSL (High Level Shader Language), or via a visual node-based interface, like Unreal Engine’s material editor or Shaderforge, a third-party plug-in for Unity 3D.

- Physically based rendering:
  A rendering pipeline that uses accurate physics concepts and calculations to simulate the physical properties of real-life objects as faithfully as possible. Those concepts include:
  - Energy conservation: concept which ensures that an object does not reflect more light than it receives.
  - Microsurface: property that determines how rough or smooth the surface of a material is.
  - Specularity and metalness: properties that contribute to the amount of reflectiveness a material exhibits.

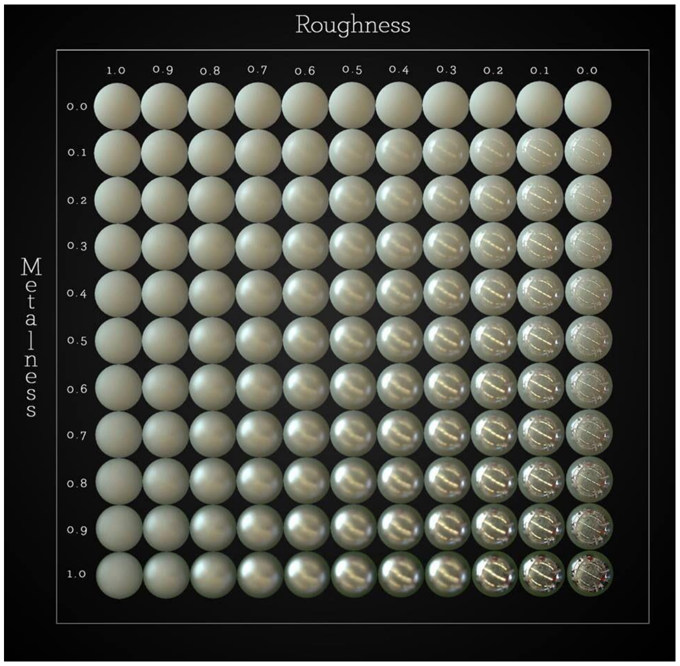


Figure 1: Demonstration of PBR material properties

- Forward and deferred shading:
  Modern video game engines nowadays incorporate programmable graphics pipelines into their graphics engines which can suit different use cases. The most popular rendering pipeline techniques are the forward and the deferred rendering techniques.

  - Forward rendering: In this rendering path, the shading calculations on all the geometries on the screen are passed sequentially to produce the final image. The calculations needed for each model's vertices and lighting take place on a per-object basis and the result is pushed down the pipeline separately. As a result, the rendering performance heavily depends on the amount of geometry and dynamic lights in a scene.
  - Deferred rendering: Conversely, in the deferred rendering path, the final shading is deferred until all the geometries have passed down the pipeline and, after that, all the shading is applied in screen space. The objects write information about their color, lighting, normal vectors etc. into separate buffers (called G-buffers) which are then used to compose the final render.

  Both techniques have their advantages and disadvantages, therefore the usage of each rendering path depends on the needs of the project and the user. Deferred rendering is usually more expensive than forward, however it can support more dynamic lights, as it does not require a separate render pass for each light, like the forward rendering path. On the other hand, deferred rendering in some cases does not have support for transparency or traditional anti-aliasing techniques and it is not suitable for development for low-end or mobile devices due to its additional computational overhead.

2.3 Unreal Engine 4

2.3.1 Introduction

Unreal Engine was developed by Epic Games, and it has always been one of the most popular choices for video game development by large studios that had the financial capabilities to license it. While the engine started as an in-house proprietary software for the development of Epic Games' "Unreal" game, after a few years Epic was licensing the engine to external studios and gave the community the tools to create different mods, thus making the engine more extensible and improvable over time.

While Unreal Engine was open to the community of modders even until its third major release, the ability to publish and sell games made using Unreal Engine was restricted to people and studios who had bought a relevant license. In November of 2009, however, Epic released Unreal Development Kit (UDK), which was a free version of the Unreal Engine's 3 software development kit (SDK).

In March 2014 Epic released the latest major version of the engine, Unreal Engine 4. A year after its release, the game engine became available to the public for free, along with all future updates and its source code written in C++.

The availability Unreal Engine 4 to the community, along with the advanced real time graphics rendering capabilities it offers, made the engine prevalent amongst a significantly wider audience, along with smaller game development studios and teams. Simultaneously, the engine has been used extensively for other purposes besides game development, such as architectural and interior visualizations as well as graphics rendering for film productions.

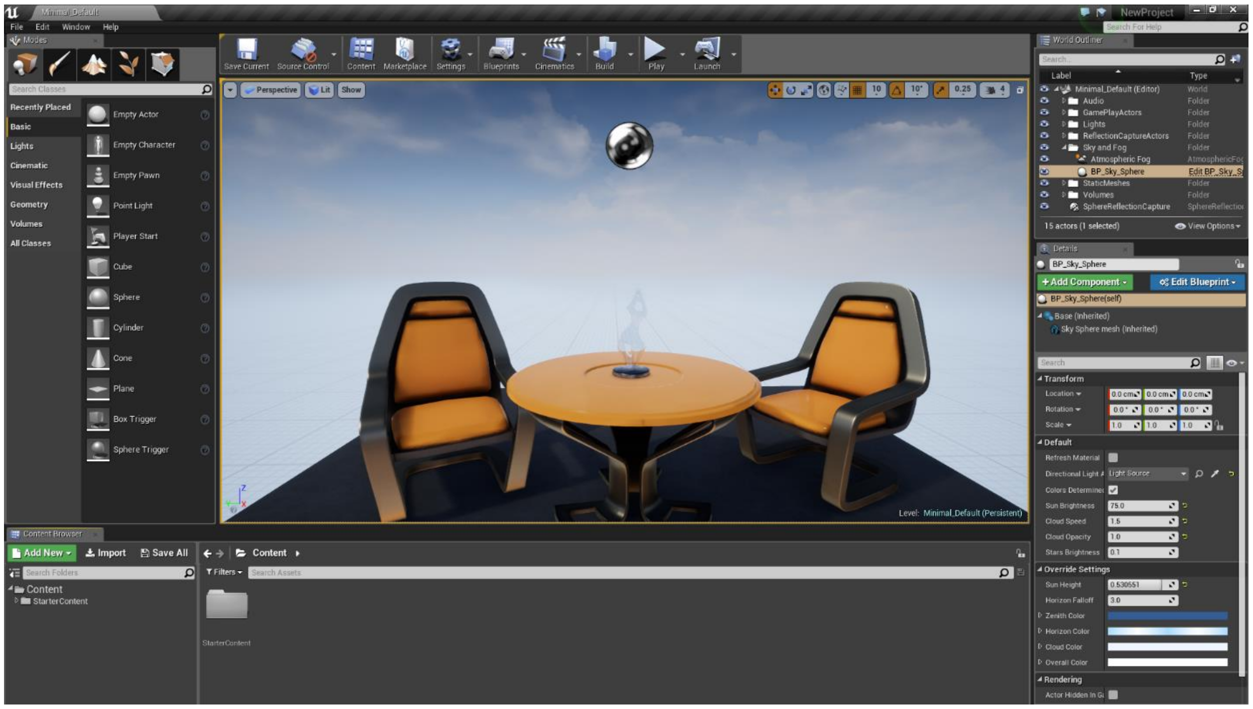


Figure 2: The main editor window of Unreal Engine 4

2.3.2 Scripting

As far as scripting to determine in-game behaviors is concerned, Unreal Engine 4 provides two main options to its users:

- Programming using C++ coding:

  Coding scripts using C++ in Unreal Engine 4 is possible through Microsoft's Visual Studio IDE. As the engine's source code is written in C++, implementing new classes directly extends the engine's main core and the newly defined behaviors get compiled along with the engine itself. As a result, new classes are immediately intertwined with the software's built-in classes and behaviors, allowing for easier creation of tools and plugins.

- Behavior designing using the "Blueprints" visual scripting system:

  With the third iteration of Unreal Engine, Epic introduced a visual scripting system called "Kismet" which featured a node-based interface. The "Blueprint" visual scripting system in Unreal Engine 4 builds upon the "Kismet" system, working in a similar fashion: For each level there is a "Level Blueprint" and each object which executes a behavior utilizes a "Blueprint" that determines the functionality of said behavior.

  The "Blueprint" system is capable of creating any desirable behavior as it accesses all the tools and concepts of the game engine. It is also very flexible, as new nodes can be created via C++ code, to allow for more custom solutions depending on the project.

By using a node-based visual solution, the "Blueprint" system allows non-programmers to easily iterate upon the gameplay behaviors of a project and define new ones.

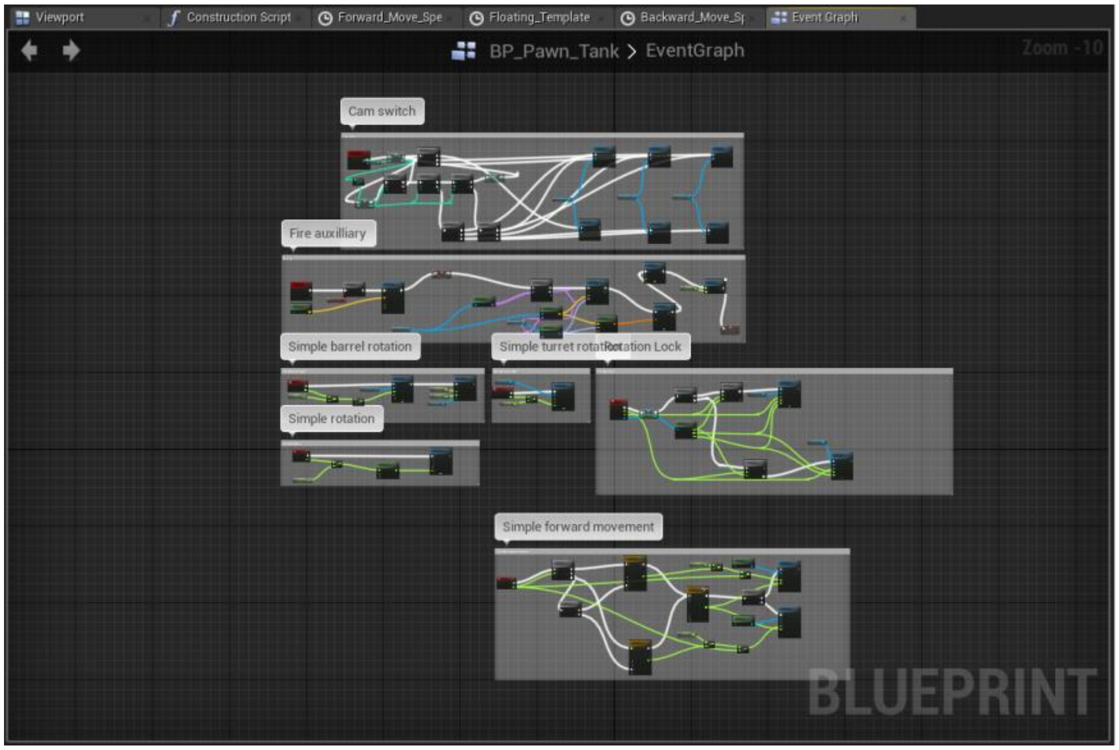


Figure 3: Unreal Engine's "Blueprint" system example

2.3.3 Rendering capabilities

Unreal Engine 4 is considered one of the best game engines in term of graphical fidelity, as it comes bundled with a plethora of built-in effects and techniques that contribute to realistic rendering of physically-based materials and lighting conditions.

The engine uses the deferred rendering path by default, however it provides the option of using forward shading as an experimental feature.

- Materials:
  In a similar fashion to the "Blueprint" scripting system, Unreal Engine 4 provides a visual material editor which can be used to generate general-purpose materials for the in-scene game objects as well as more complex, case-specific image effects. The node-based material editor is similar to those in 3D modeling software, like Autodesk's 3D Studio Max, which gives the freedom to artists and designers to easily generate material and shader instances without the use of code.

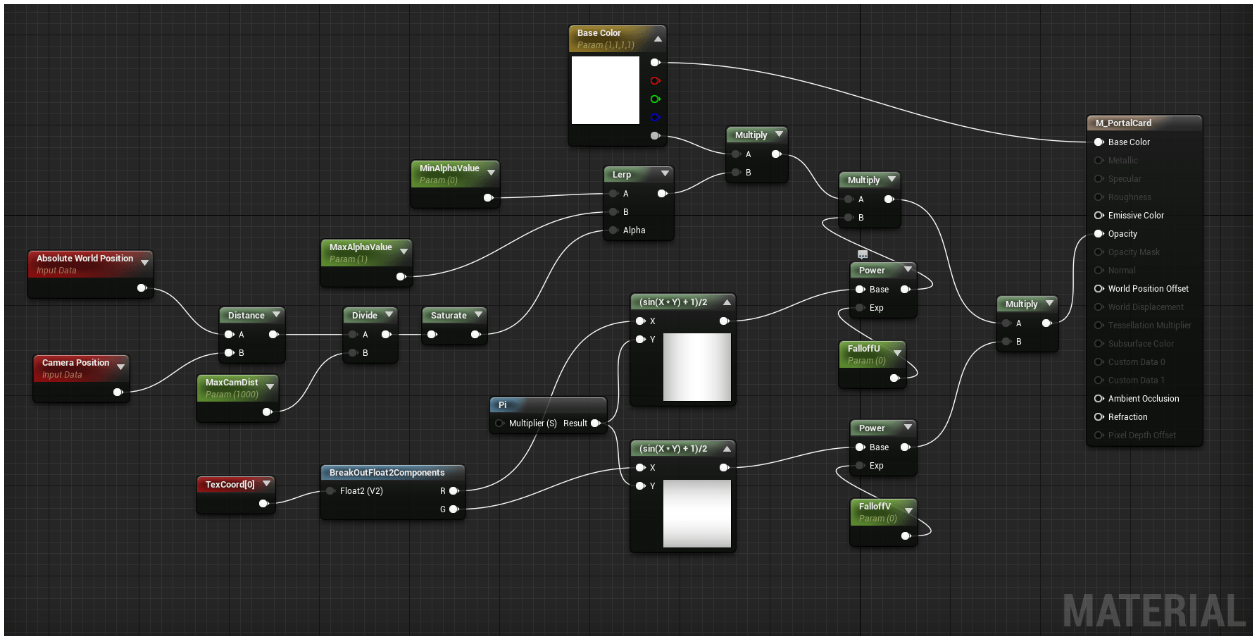


Figure 4: The material editor of Unreal Engine 4

The material editor system is powerful enough to generate complicated object materials as well as post processing effects, through a large library of built-in nodes and math functions. The output of a material is in its essence a shader, instances of which can be applied on objects or on the camera through post-process volumes.

Some of the engine's notable material features are:
- Physically based rendering supporting both metallic and specular/roughness pipelines
- Built-in translucency and sub-surface shading
- Tessellation
- Refraction

- Lighting:
  Unreal Engine 4 has support for multiple lighting modes and effects. Specifically, the engine has support for three types of lights:
  - Movable lights, which can be moved and modified dynamically on runtime, updating the shadows of the scene objects in real time.
  - Stationary lights, which, while they are intended to stay in one position, they can have their properties modified, such as brightness and light color, dynamically.
  - Static lights, which are baked prior to gameplay and all the lighting and shadow information is stored in lightmaps.

It is worth mentioning that Unreal Engine 4 uses the "Lightmass" system to precompute the global illumination to which static and stationary lights contribute. This method is capable of computing ambient occlusion, masked and translucent/colored shadows as well as color bleeding from surfaces with more saturated colors.

The engine also has built-in support for fog effects, such as exponential height fog and atmospheric fog, which gives an approximation of light scattering through the atmosphere. Those effects allow for visually interesting volumetric effects and they significantly contribute to the graphics fidelity of a scene.

There is also support for reflection capture volumes that are used to bake static reflection cubemaps that can be used from objects in the volume as an additional chromatic information.

- Post processing:
  One of the factors that contribute the most to Unreal Engine's graphics is the built-in post processing effects. Some of the most significant effects are:

  - Bloom
  - Screen Space ambient occlusion
  - Tone mapping
  - Color grading
  - Screen space reflections
  - Depth of field
  - Motion blur
  - Auto-exposure
  - Anti-aliasing

  The post-processing effects can be applied globally in a scene or by using post-process volumes, applying the effects only when the main camera of the scene is inside the corresponding volume area. The post-processing system of Unreal Engine 4 can also be expanded upon with custom image effects that can be created through its material editor.

- New advances:
  In 2018's Game Developers Conference, Epic presented some of the newer technological advancements as far as the graphics of the engine are concerned, which have not been released at the time of writing. The more noteworthy of them include:

- Improved volumetric fog and volumetric lights

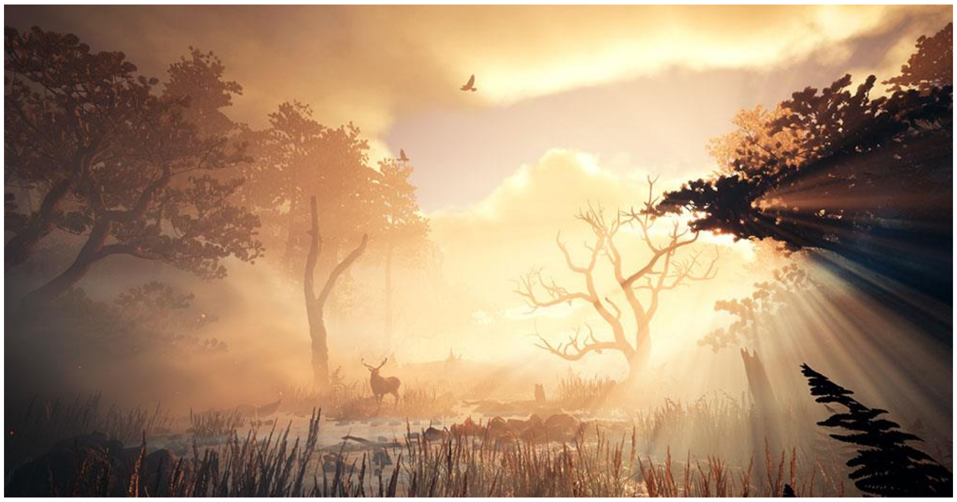


Figure 5: Volumetric fog and lighting in Unreal Engine 4

- Advancements in high fidelity character rendering

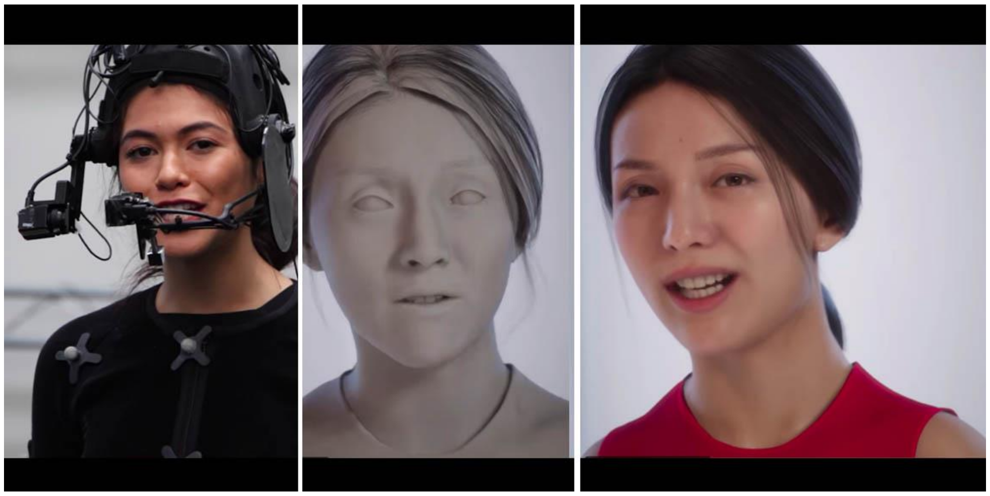


Figure 6: Character rendering demo in Unreal Engine 4

- Real time ray-traced effects, such as shadows, reflections and depth of field, which use the new NVIDIA real-time ray tracing GPU hardware.

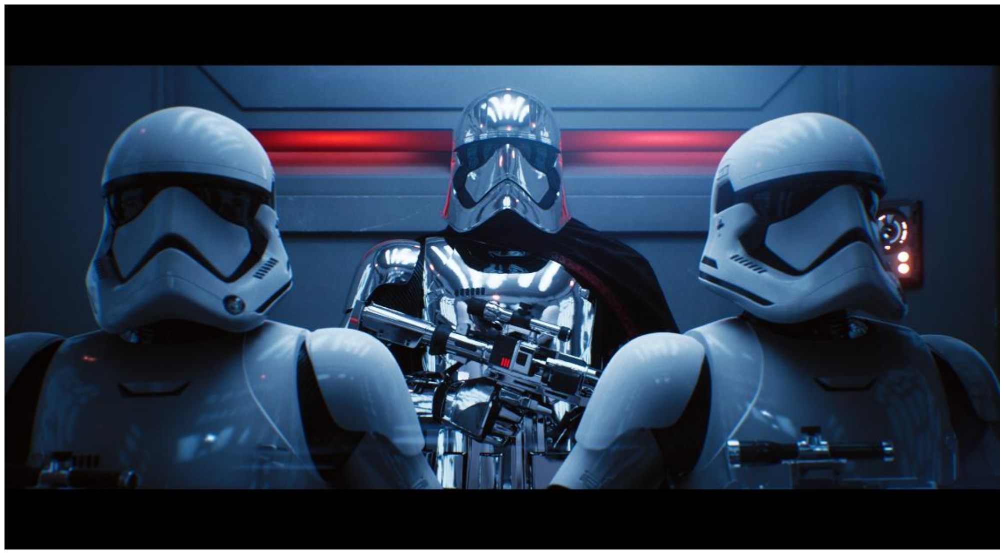


Figure 7: Real time ray-traced effects demo in Unreal Engine 4

2.3.4 Advantages and disadvantages

- Unreal Engine 4 advantages:
    1. The engine works uses node-based interfaces for numerous tasks, including behavior definition and material creation, making it suitable for non-coders, like artists and designers, to iterate upon the gameplay and visuals of the game.
    2. Unreal Engine 4 produces high-fidelity graphics out-of-the-box, without the need for custom or third-party solutions. Therefore, it is easier for designers to create realistic environments without the need for technical knowledge.
    3. The game engine has a large number of built-in modules and tools that cover a plethora of use cases. Those tools include, but are not limited to:
        - Systems for building AI agents such as behavior trees, navigation mesh generation and sensory systems.
        - Tools for greyboxing level prototypes.
        - In-engine modules for animation handling, particle systems creation and cutscene direction.
    4. Due to the engine’s popularity, there are many online resources that help new users get accustomed to the interface.
- Unreal Engine 4 disadvantages:
    1. Because of its modular nature, Unreal Engine 4 has a steep learning curve for new users.
    2. While there is a large community of Unreal Engine 4 users, the fact that the engine was not free until 2015 did not allow it to grow a community in the size of other engines, like Unity 3D. As a result, there are not as many resources and references as with other engines, and not much attention has been given to the engine’s official documentation.

3. A significant drawback to the engine’s variety of tools and modules is its high complexity, which can be intimidating for new users that want to learn how to use it.
4. Even though the “Blueprint” system is very powerful, it does not facilitate the creation of custom tools that extend the engine’s editor. For more in-depth modifications, one would have to use C++, which makes the learning curve of the engine even steeper.
5. While there is a marketplace of third-party plugins for Unreal Engine 4, it is still growing, and it is not as rich as similar stores of other engines like the Unity Asset Store.
6. Unreal Engine 4 provides tools for 2D game development, however, due to it being more suited for realistic 3D video games, the engine provides far more features than what a 2D game would need, increasing the complexity and performance costs.
7. Although the “Blueprint” visual scripting system is intuitive and useful for non-coders, it can be inconvenient for programmers as larger “Blueprints” might have a confusing, non-linear structure that requires additional effort to get converted to a state that is easy to understand and debug.

2.3.5 Conclusion

Unreal Engine 4 is a very technologically advanced game engine that is suitable for both large commercial productions as well as smaller applications. It excels at real time graphical fidelity and provides rich toolsets for users to develop a plethora of different applications. Epic Games’ system of improving the engine based on commercial titles built with it greatly contributes to the engine’s flexibility and adaptability that match the needs of real commercial products. However, the modularity and variety of different tools and workflows greatly contribute to its complexity and steeper learning curve, making it more difficult for new users to get accustomed with the engine’s capabilities. Furthermore, while Unreal Engine 4 supports development for mobile and 2D game development, it is not as suited for such applications as much as other game engines due to its focus on high-quality 3D graphics rendering.

2.4 Unity 3D

2.4.1 Introduction

Unity 3D is a cross-platform game engine and IDE developed by Unity Technologies. Since its first release in 2005, Unity has been one of the top choices for smaller or independent studios as well as for education purposes, as Unity Technologies provided a free version of the engine to the public. The free versions of Unity always had some limitations compared to the paid ones, however it was still possible for developers to create and publish small games. Additionally, what facilitated the engine’s popularity was the fact that it could export to numerous platforms, such as iOS, Android, Flash, Windows, OS X, Linux as well as consoles like PlayStation, Xbox and Wii.

While for the first 4 major releases of Unity the free versions did not give the users access to the engine’s full features, Unity Technologies still managed to achieve its goal of democratizing video game development. The release of Unity 5 in March 2015 of the free version of the engine gave access to all its technical features, while also introducing major updates in the editor and its technological capabilities. There are still paid licenses for the Unity engine, but instead of unlocking restricted features, they provided services that mostly relate to cloud computing and networking.

After Unity 5, the versioning numbering system changed to match the year of each version's release, with the most recent one being Unity 2018.1 at the time of writing.

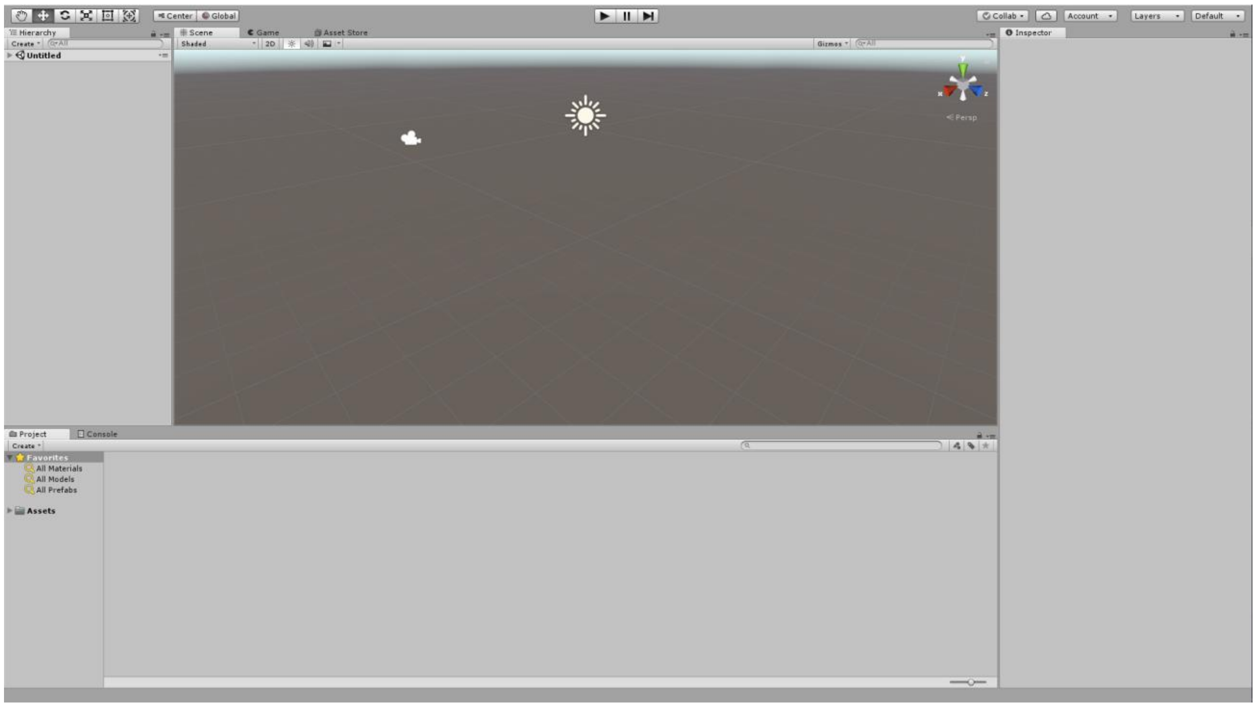


Figure 8: The Unity 3D editor window

2.4.2 Scripting

The game scripting in Unity is done via Mono, which is an open-source implementation of Microsoft's .NET framework. The available programming languages for coding were C#, UnityScript (a JavaScript-inspired language), and Boo (a Python-like programming language). However, Boo was deprecated with the release of Unity 5, while Unity Technologies announced that starting with the release of Unity 2017.2 UnityScript support will be removed from the editor. Therefore, it is now possible to code behaviors only using C#.

Until version 2017, Unity has been using the .NET 3.5 framework while offering the option to use .NET 4.0 as an experimental feature. In Unity 2018 .NET 4.0 has been integrated as a stable scripting runtime option.

While Unity does not offer a visual scripting solution, like Unreal Engine's "Blueprint" system, there is a variety of third-party solutions that provide visual interfaces for behavior scripting as well as shader creation, similar to UE4's material editor. It is also noteworthy that most of the Unity editor's classes are exposed, allowing users to easily create custom tools and modules that better suit their project's needs.

2.4.3 Rendering capabilities

Even though Unity was never known for its built-in rendering capabilities in the same manner as Unreal Engine 4 or other game engines, the big advantage of the Unity 3D engine is the flexibility it provides. Contrary to other engines, Unity 3D was always open to be built and expanded upon, though without being open to be modified by its users. That philosophy applies to most of its features, including its graphics rendering engine.

Unity 3D provides both forward and deferred rendering options, with the forward rendering path being the default option.

- Materials:
  Unity provides a wide variety of different type of built-in material shaders that are useful for a plethora of different use cases. The most notable of the built-in shaders is Unity's standard shader, which was introduced in Unity 5.0. The standard shader uses physically based rendering techniques to realistically represent different kind of real-life materials, using both a specular and metallic workflow. More specifically, this shader has support for:
    - Opaque, transparent, fade and cutoff rendering modes
    - Both the forward and the deferred rendering path
    - Specular or Metallic maps and glossiness
    - Normal maps
    - Parallax using height maps
    - Occlusion maps
    - Emission
    - Detail albedo and normal maps
    - Lightmapping

  While using all of its features can make a standard shader instance computationally expensive, when a project is built the unused properties of the shader are discarded, thus optimizing the performance of the materials that use the shader.

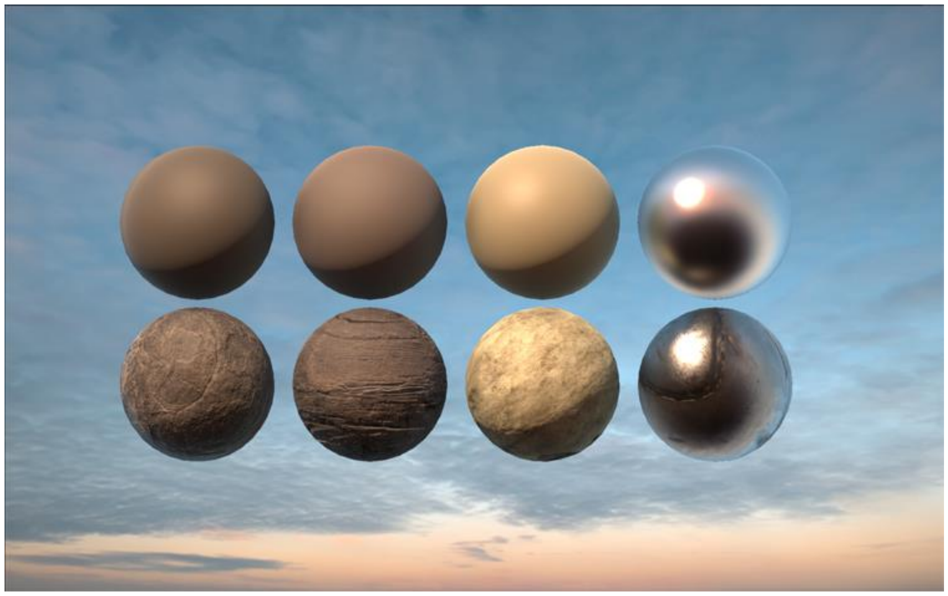


Figure 9: Different materials using Unity's standard shader

With the release of Unity 2018 and the introduction of Scriptable Render Pipelines (SRPs), Unity expanded even further upon the features of the standard shader. More details on the new features are included in the "Scriptable Render Pipelines" section.

- Lighting:
  Unity offers a wide variety of options and tools for lighting that contribute to the visual result of a scene. At a higher level, the engine offers the following lighting features:

  - Dynamic lights: Unity offers support for dynamic directional, point and spot lights with real time soft and hard shadow casting using shadowmaps.
  - Static lights: Like Unreal Engine 4, Unity 3D can bake lighting information for static objects onto lightmaps either using the "Enlighten" system, or the newly added "Progressive lightmapper". Static lights also add support for baked global illumination, precomputed real time global illumination as well as ambient occlusion.
  - Reflection probes: Unity uses reflection probes to capture a spherical view of their surroundings to store them as cubemaps. Other objects in the area of the reflection probe can use the generated cubemap with their material's reflection properties, giving the objects color information that convincingly matches their surroundings.
  - Light probes: Light probes store baked light information in a similar fashion as lightmaps, with the main difference being that they store information about the light that passes through empty space in a scene, instead of the light that illuminates surfaces. Light probes use a compact representation of low-frequency lighting (spherical harmonics).

  It is noted that Unity 2018 introduced new features in the lighting field too, which are also covered in the "Scriptable Render Pipelines" section.

- Shader creation:
  One of the main advantages of Unity is that it allows users to create shaders using code via its system called "ShaderLab". The shader coding is similar to the material creation in Unreal Engine 4, however it is more flexible and powerful since it uses low-level shader programming. The shader coding is done through CG, HLSL or GLSL which is nested inside Unity-specific directives that are used to connect the shader with the Unity editor. While CG is no longer officially supported by NVidia, it is the most common choice for shader creation in Unity. The lack of support for CG does not affect the shader creation, as the shaders get compiled to HLSL or GLSL later in the pipeline.

  Unity offers support for 4 built-in types of shaders:
  - Unlit shaders: The simplest form of a shader which only returns a solid color and does not account for lighting or special types of textures like normal or height maps.
  - Surface shaders: Those shaders provide a level of abstraction about the advanced PBR lighting features and allow the user to modify the material's properties before the lighting calculations are applied to it. Surface shaders

also allow for the creation of custom lighting models that work with Unity's lighting systems without any special setup, providing more control over the visual result while abstracting the technical details of the implementation.

- Image effects shaders: While Unity has a separate option for the image effects shader, their syntax is almost identical to that of unlit shaders. The reason for this is that image effect shaders apply effects directly to the camera's output as post-processing effects by treating the camera's view as a quad. Even though their structure follows that of an unlit shader, actually applying the effect to the camera requires a C# script.
- Compute shaders: Although compute shaders are not exclusively used for graphics rendering, they are certainly noteworthy. They are written in DX11 HLSL and they can be used for massively parallel general-purpose GPU (GPGPU) algorithms or to accelerate parts of the graphics rendering workflow. Examples of compute shader use include GPU accelerated particle systems as well as complex physics simulations.

Even though creating shaders using code offers increased flexibility, it is not as intuitive or user-friendly as a visual solution because of lack of sufficient documentation due to shaders' technical nature. However, there are popular third-party solutions like "Shaderforge" and "Amplify Shader Editor" that provide a node-based interface for shader creation, similar to that of Unreal Engine's material editor. Additionally, with the release of Unity 2018, Unity Technologies released a preview version of a visual shader creation tool called "Shader Graph" for the lightweight scriptable render pipeline.

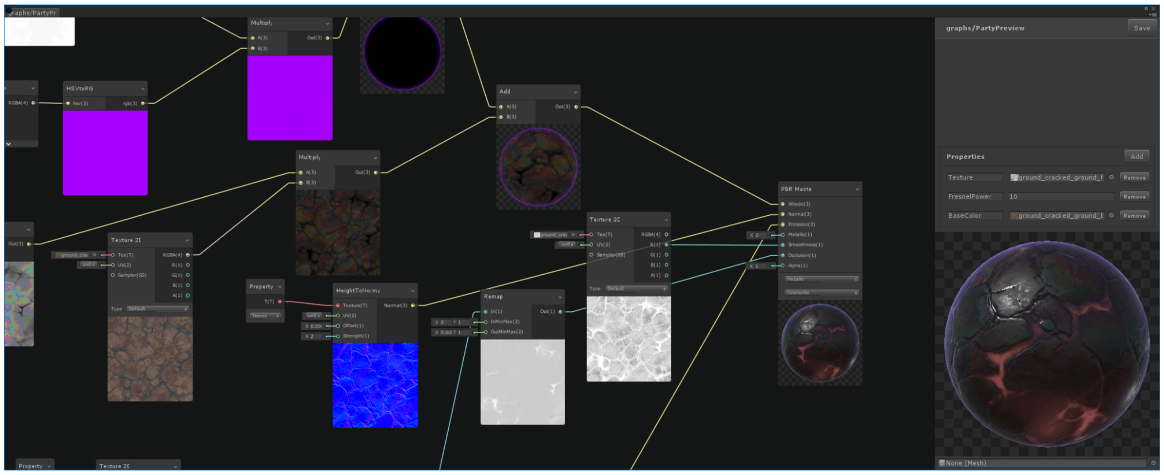


Figure 10: Unity's "Shader Graph" system

- Post processing effects:
  Unity 3D does not use and apply post-processing effects by default in the same manner as Unreal Engine 4. However, Unity provides an external post-processing package that can be integrated into any project. That package is named "post-processing stack" and can be used to apply different image effects in an efficient manner. The most recent version of the post-processing stack has the following features:

- Post-processing profiles that are stored as files. These profiles store which effects will be used by the camera as well as their parameters, allowing the usage of the same effects across multiple scenes.
- Support for post-processing volumes that can work either globally or locally in the same way as Unreal Engine's post-process volumes, where the effects are applied only when then camera is inside a specified area.
- Ability to extend the stack with custom effects by using a C# SDK and shaders written in HLSL. Other custom image effects can be applied outside the post-processing system, however, integrating the effects with the post-processing stack provides a more flexible and efficient workflow.

The post-processing stack comes with the following effects built-in:
- Bloom
- Chromatic aberration and vignette
- Screen space reflections (for the deferred rendering path)
- Color grading and tonemapping
- Depth of field
- Screen space ambient occlusion
- Auto-exposure
- Anti-aliasing
- Motion blur
- Lens distortion and grain

- Scriptable Render Pipelines:
  With the release of Unity 2018, Unity Technologies integrated the "Scriptable Render Pipelines" system in the Unity 3D game engine, therefore exposing the graphics rendering pipeline of the engine and allowing users to create rendering workflows that best suit their project's needs.

  The scriptable render pipelines determine three main aspects regarding the graphics workflow of the project:
  - The culling technique: controls the criteria according to which Unity decides whether an object will be rendered or not
  - Rendering: involves the way each object will be rendered at a low level. The user can define how the sorting of the objects will be performed, the way the objects will be dynamically batched, the preparation and dispatching of draw calls as well as the way GPU receives and processes the draw calls
  - Post processing effects: scriptable render pipelines give the user freedom over the effects that will be applied after the scene is rendered

  To demonstrate the new feature, Unity includes two built-in scriptable render pipelines:
  - Lightweight Render Pipeline (LWRP): this render pipeline is designed to target mobile devices and mid to low-end devices and, therefore, has better performance than the Unity built-in pipeline. It only supports forward rendering; however, it performs single-pass forward rendering with only one real time

shadow light. As a result, all the lights in the scene are rendered in a single pass instead of performing an additional pass per pixel light like the built-in pipeline does. Additionally, the LWRP introduced the "Shader Graph" tool which provides a visual shader authoring workflow, as mentioned above.

- High Definition Render Pipeline (HDRP): the high definition render pipeline targets high-end PCs and consoles by prioritizing visual quality. This render pipeline provides a plethora of different features that contribute to a more realistic representation of 3D objects. Some of those features are:
    - New material types: the new PBR standard shader of the HDRP includes support for anisotropic lighting, iridescence as well as translucency, subsurface scattering and decals.
    - New lighting features: the HDRP adds support for more control and flexibility over the properties of dynamic and static lights, physical units of light intensity for more coherent and consistent lighting as well as dynamic area and line lights. There is also support for colored light "cookies" and modifying the color temperature of the light.

Both lightweight and high definition render pipelines, are in preview mode at the time of writing and more features are to be added in future versions.

2.4.4 Advantages and disadvantages

- Unity 3D advantages:

1. Unity Technologies provide thorough documentation as well as official resources and tutorials that significantly contribute to the engine's ease of use.
2. Since the engine was available to the community since its first releases, it has developed one of the largest active game development communities, providing additional third-party learning resources. Additionally, due to the size of the user community, the Unity asset store is increasingly growing, providing a great variety of third-party assets and tools.
3. Because Unity 3D has most of its editor C# classes exposed, it greatly facilitates the creation of custom tools via extending the editor, allowing for custom workflows that suit the developers' needs.
4. Unity Technologies always prioritized cross-platform deployment, and, during the period of writing, the Unity engine can support over 25 platforms, including mobile platforms, VR and AR platforms, desktop computers, consoles and the web.
5. With the release of Unity 4.3, Unity Technologies integrated a 2D-specific toolset in the engine which has since been updating in tandem with the rest of the engine's features. These tools provided a powerful and flexible 2D workflow in Unity, greatly facilitating the creation of 2D games for mobile and desktop platforms.
6. Even though the engine lacks built-in visual authoring tools, the engine's simplicity along with the availability of online resources significantly contributes to its smooth learning curve.

- Unity 3D disadvantages:

  1. Unlike other high-end game engines, the Unity 3D engine does not use realistic 3D graphics rendering techniques by default and it takes more effort by the developer to achieve a better visual result.
  2. New major updates of Unity 3D versions are released quite regularly, and, in most cases, larger projects may have compatibility issues with the new engine versions, making it difficult to support and maintain a project while also taking advantage of newly released features.
  3. Due to the lack of case-specific built-in tools and systems, rapid prototyping with the Unity engine is not as easy as with other video game engines.
  4. The lack of visual scripting or visual shader creation systems make Unity less accessible by people with no programming background, like artists and designers. To accommodate non-coding developers, it is necessary to develop new tools specifically for the project or to use a third-party solution.
  5. Unity 3D was designed to cover every type of game, and therefore be as generic as possible. While this element contributes to its flexibility and extensibility, it also means that there are no convenient case-specific tools built inside the engine by default. Therefore, while the engine operates in a very high level, its functionality is closer to that of a framework, in the sense that it provides generic tools for multiple use-cases, instead of various case-specific toolsets.

2.4.5 Conclusion

Unity 3D is a game engine specifically designed to allow the community to create an enormous variety of interactive applications. Unity Technologies invested into making the engine as flexible as possible by allowing users to easily extend upon the engine’s editor and create custom toolsets and workflows. While the engine is not equipped with a high-quality rendering pipeline or case-specific toolsets, it is built like a framework, allowing the user to use the engine in a way that fits the needs of their project. Its active community, sufficient learning material and simplicity allow inexperienced users to quickly become accustomed to the engine’s interface. Additionally, while the engine requires more effort to be adjusted to the needs of a specific project, it can be configured to cater to the needs of every type of interactive application, from a high-quality 3D game to a simpler 2D mobile game.

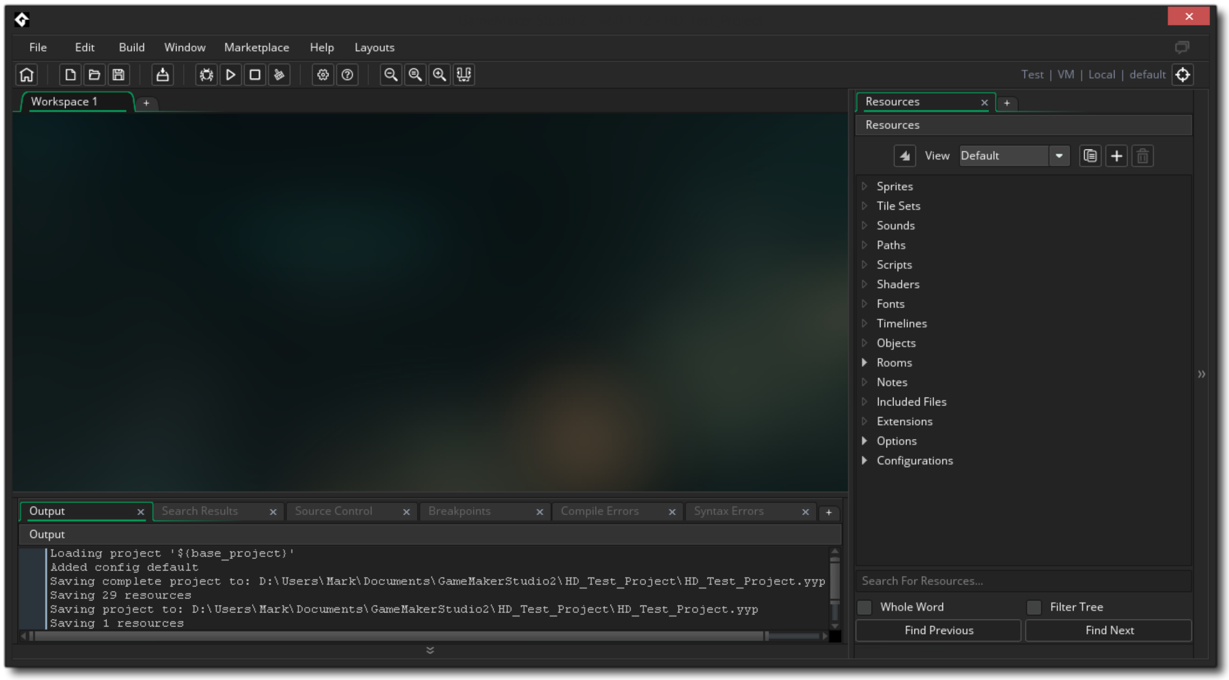
2.5 GameMaker Studio

2.5.1 Introduction

GameMaker Studio is a cross-platform game engine developed by YoYo Games. The engine has a rich history regarding its development, as its first release (by the name of “Animo”) dates back to 1998. Even if the engine was never entirely free, it managed to gain a lot of popularity among the game development community due to its ease of use and directness regarding game prototyping and designing.

Some of the features that contributed to the popularity of the engine include the built-in tools for 2D game development. Users are able to design assets and animations in editor as well as use an intuitive workflow for level designing via tile-mapping tools.

In March 2017 YoYo Games released GameMaker Studio 2 which offered a completely redesigned IDE as well as a plethora of different editor and runtime features.



```
Loading project '${base_project}'
Added config default
Saving complete project to: D:\Users\Mark\Documents\GameMakerStudio2\HD_Test_Project\HD_Test_Project.yyp
Saving 29 resources
Saving project to: D:\Users\Mark\Documents\GameMakerStudio2\HD_Test_Project\HD_Test_Project.yyp
Saving 1 resources
```

Figure 11: GameMaker Studio 2 editor

2.5.2 Scripting

GameMaker Studio provides both a visual scripting solution and a custom scripting language.

- The visual scripting tool is called “Drag and Drop” (DnD) and while it can perform a variety of common tasks, it is not as flexible as the visual scripting solutions met in other game engines. With the release of GameMaker Studio 2, the visual scripting system has been significantly enhanced, allowing for more tasks to be executed via the tool.
- The main way to program behaviors in GameMaker Studio is via its custom scripting language called GameMaker Language (GML). GML is an imperative and dynamically typed language with a syntax that resembles that of C or JavaScript. GML can be used both for behavior design as well as for editor extensibility and custom tools. All the coding can be done through the editor itself, without the need for an external text editor or IDE.

2.5.3 Rendering capabilities

GameMaker Studio is primarily built around 2D game development; therefore, its graphics engine is built around 2D graphics rendering, supporting both raster and vector graphics along with 2D skeletal animations.

Furthermore, GameMaker Studio allows for the creation of custom shaders using GLSL or HLSL. Those shaders can either be used on separate objects or they can be applied to the main camera as post-processing effects.

The engine supports the use of 3D graphics but only through a low-level graphics framework that includes the use of vertex buffers and matrix functions. As a result, there are no built-in 3D graphics features, and games that use 3D graphics must be built from the ground-up with limited support.

2.5.4 Advantages and disadvantages

- GameMaker Studio advantages:

    1. GameMaker Studio’s workflow is focused on gameplay and mechanics, making it ideal for rapid prototyping and experimentation.
    2. The familiar structure of GML along with the visual scripting tool significantly facilitate the learning and use of the engine.
    3. For specific types of art assets (i.e. pixel art sprites), GameMaker Studio’s built-in sprite editor allows for faster iteration on the art assets and animations without the need for external software.
    4. GameMaker Studio’s scripting language works very intuitively and is quite flexible, both in its syntax as well as in its functionality.
    5. The engine’s popularity for 2D games contributed to the development of its active community, providing users with a variety of third-party resources and external tools via the engine’s marketplace.
- GameMaker Studio disadvantages:

    1. The engine was designed around 2D workflows, therefore GameMaker Studio is not suited for 3D game development, even if it supports 3D graphics rendering at a very low level.
    2. While YoYo Games provide the engine for free, the free version of GameMaker Studio is very limited and mostly serves as a trial for the engine.
    3. The engine does not provide flexible modules for animations, particle systems or audio management like other game engines. Most of that functionality is usually covered by custom tools written with GML or by third-party solutions.

2.5.5 Conclusion

GameMaker Studio is a game engine that is focused on creativity and experimentation. It has a rich toolset that is dedicated to rapid gameplay prototyping while also abstracting technical details. While it lacks sufficient support for 3D graphics rendering for inexperienced users, it provides intuitive tools for 2D game development that facilitate asset creation and iteration. Furthermore, the engine is a popular choice for beginners, as the simplicity of its custom scripting language and the user-friendly interface provide a smoother learning curve for new users and an overall gentle introduction to the process of game development.

2.6 Conclusion

Video game development has been rendered more and more accessible over the last years, with companies focusing on making their technology available to the public. As a result, numerous game engines and frameworks have been open and available to the community, such as Unreal Engine 4, Unity 3D and GameMaker Studio. Each engine offers different tools and capabilities that suit different needs, depending on the project. Unreal Engine 4 offers high visual fidelity and a rich built-in toolset for larger productions as well as smaller projects. Unity 3D can fit every type of project due to its versatility and extensibility, while its rich community contribute to its ease of use. Finally, GameMaker Studio is a complete solution for 2D game projects across multiple platforms, giving the user the tools to focus on the creative process of game development.

3. SCREEN SPACE DIRECTIONAL OCCLUSION IMPLEMENTATION

3.1 Introduction

This part of the thesis focuses on the technical aspect and the process of implementing a custom graphics rendering technique in a pre-existing game engine. Specifically, it presents in detail the implementation process of screen space directional occlusion, as described in the paper “Approximating Dynamic Global Illumination in Image Space” [2], inside the environment of Unity 3D. This section discusses both the technical and theoretical characteristics of the method described in the paper, as well as the implementation methodology followed to integrate the technique in Unity’s rendering pipeline.

3.2 Theoretical background

3.2.1 Direct and indirect illumination

Illumination techniques play a significant part in the realistic rendering of modern computer graphics. In order to be faithfully implemented, illumination techniques are based on relevant laws of optics. In the context of real time applications that feature 3D graphics, illumination models are distinguished into two main categories: direct (or local) illumination and indirect (or global) illumination.

- Direct (local) illumination:
  Direct illumination consists of practical illumination models that are used to produce acceptable illumination effects at a low computational cost, making them suitable for real time applications [3]. These models are applied on a per-object basis without necessarily taking into consideration the rest of the scene, resulting in fast, sufficient but not entirely realistic or accurate shading.

  Some of the most popular direct illumination models include:
  - Lambert reflectance (or diffuse light): One of the simplest and most commonly used lighting models. Lambert reflectance is view independent, therefore the surface’s brightness does not vary based on the viewpoint; it assumes that light is reflecting equally in all outgoing directions.
  - Half-Lambert (or diffuse wrap): A modification of the Lambert reflectance model used in the original Half-Life game by Valve. It ensures that the non-illuminated surfaces of an object are not completely dark as to not lose their shape. For that reason, this illumination model is not considered to be close to a physically accurate model.
  - Phong lighting: Most commonly used to create specular effects, Phong lighting is based on the assumption that a surface reflects light using a combination of diffuse and specular reflection.
  - Blinn-Phong lighting: A modified version of the Phong lighting model, it optimizes the original technique by not calculating the light vector every frame for the specular reflection of a surface but, instead, calculating the vector in the middle between the light direction and the view direction, resulting in much smoother specular reflections.

  - Minnaert lighting: A lighting model originally designed to replicate the shading on the surface of the moon, this model is good for simulating porous or fibrous surfaces.
  - Oren-Nayer lighting: This model is a reflectivity model for diffuse reflection on rough surfaces, used as a simple way to approximate light on a rough, but still Lambertian, surface.

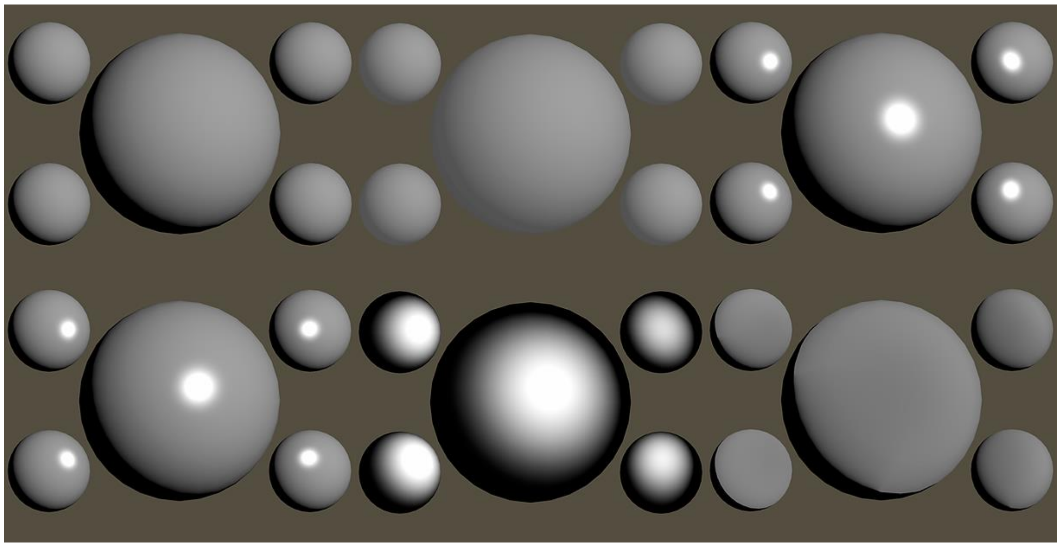


Figure 12: Direct lighting models taken from [4]. Top row from left to right: Lambert diffuse, Half-Lambert, Phong Lighting. Bottom row from left to right: Blinn-Phong lighting, Minnaert lighting, Oren-Nayer Lighting.

- Indirect (global) illumination:
  Global illumination algorithms deal with the realistic computation of light transport in a 3D scene [5]. Global illumination techniques do not take into account just the direct illumination, but they also calculate the contribution of the indirect illumination that occurs from the physical properties of light, such as diffuse inter-reflection or caustics resulting from refracted rays.

  The techniques that are used for global illumination greatly contribute to the photorealism of a 3D scene, however they can be significantly more computationally expensive than direct lighting techniques and, therefore, are not commonly used in real time applications. Most game engines incorporate off-line procedures such as baked global illumination via lightmaps or precomputed real time global illumination.

  Some elements that derive from global illumination techniques and contribute to the visual fidelity of a 3D scene are:
  - Bounced light: light bouncing and transferring chromatic information to surrounding surfaces.
  - Ambient occlusion: technique that provides soft shadows by darkening surfaces that are partially visible to the environment.
  - Caustics: the envelope of light rays reflected or refracted by a curved surface.

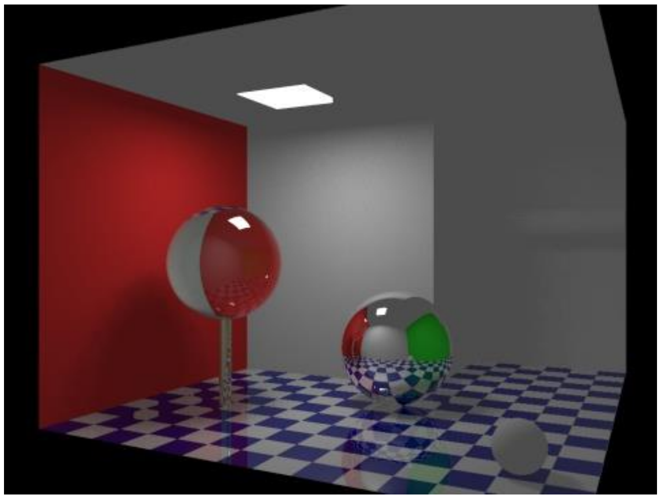


Figure 13: Scene rendered without global illumination

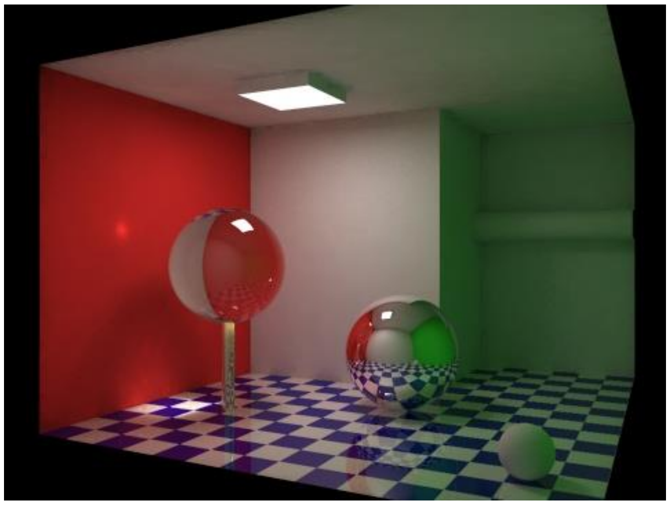


Figure 14: Same scene as fig. 13, rendered with global illumination. The scene illustrates color bouncing from walls onto other surfaces (specifically the red and green walls, see transfer of color to the ceiling), ambient occlusion (more noticeable near wall corners) and caustics cast from the glass sphere on the left of the scene onto the red wall

Some of the most popular global illumination algorithms include:

- Ray tracing: In ray tracing, there is a ray that travels the three-dimensional scene along the line of sight starting from the camera focal point and passing through each pixel registering what the observer sees along this direction. As the ray collides with geometric entities it is reflected, refracted, attenuated and absorbed, depending on the material properties of the object [6]. To achieve more complex effects, such as diffuse light bounces or caustics, stochastic Monte Carlo integration must be used, in algorithms such as path tracing or photon mapping.

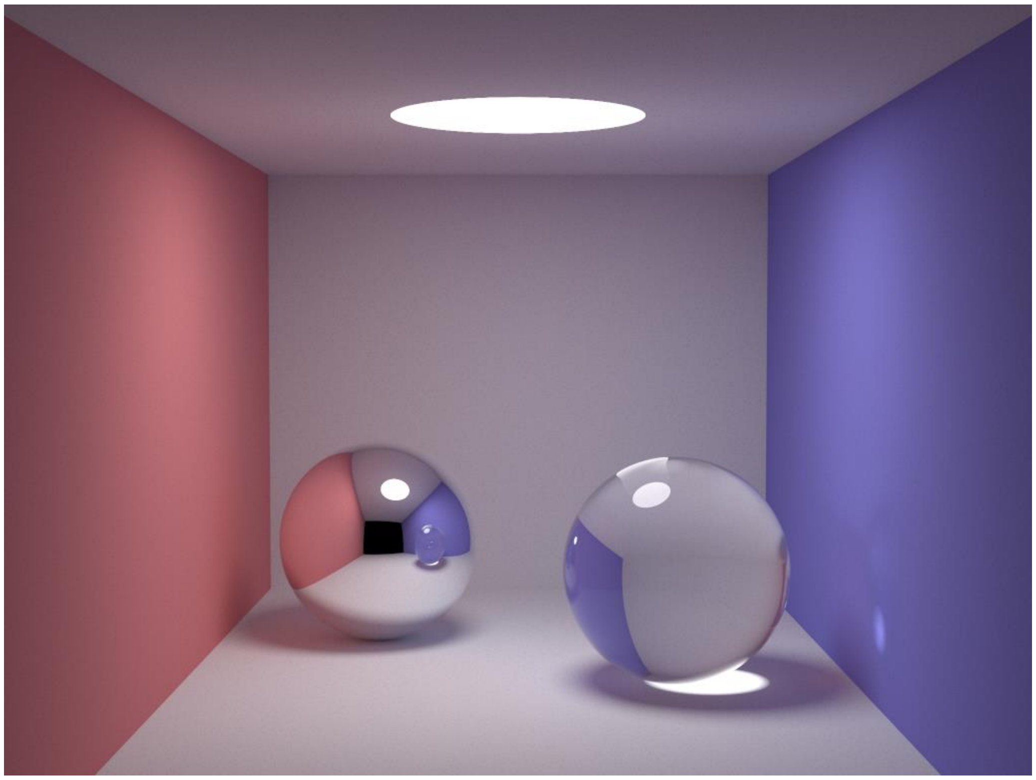


Figure 15: Ray traced global illumination scene, taken from http://www.kevinbeason.com/smallpt/

- Radiosity: Radiosity algorithms account for paths which leave a light source and are then reflected diffusely a number of times before hitting the eye [7]. This technique is used by many modern game engines, including Unity 3D via the “Enlighten” system.

  There have been different implementations of the radiosity algorithm that significantly increase the calculation speed of the method. One of those improved versions is “Instant Radiosity” [8] which greatly reduces the procedure time by using a quasi-random walk technique along with a jittered low discrepancy sampling method. An extension of the method called “Bidirectional Instant Radiosity” [9] was published in 2006, based on the concept of building and efficiently combining several estimators in order to find a set of virtual point light sources that are more relevant to the user’s viewpoint.

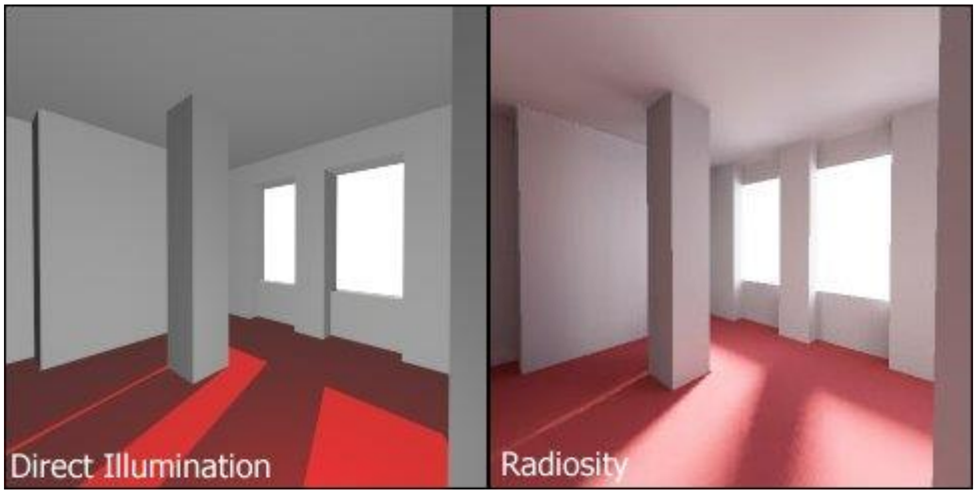


Figure 16: Radiosity demo, taken from https://en.wikipedia.org/wiki/Radiosity_(computer_graphics)

- Ambient occlusion: Method that is used to calculate how exposed each point in a scene is to ambient lighting. The idea of the technique lies in computing the obscurance of a given point, by examining other points in its vicinity [10].

  Due to the method's simplicity, there have been multiple algorithms that compute ambient occlusion in real time and are, therefore, applicable in the context of a real time application. Some of those methods include:
  - Screen space ambient occlusion (SSAO): Ambient occlusion technique developed by Vladimir Kajalin while working at Crytek. It works by using the camera's depth buffer to calculate the difference between the depth of a pixel on the screen and the depth values of the pixels surrounding it. The technique is described in detail in section 3.2.2.
  - Horizon-based ambient occlusion (HBAO) [11]: Introduced by L. Bavoil, M. Sainz and R. Dimitrov, the technique uses the depth buffer in a similar manner as the aforementioned SSAO method. The main difference lies in the sampling method of the pixels surrounding the examined pixel, as HBAO uses a dynamic sampling method to get the depth of the pixels by calculating the horizon angle and rotating the sampling hemisphere accordingly.
  - Voxel ambient occlusion (VXAO): An ambient occlusion technique based on Nvidia's Voxel Global Illumination method. The technique turns the scene's geometry to voxels on every frame and calculates the occlusion by assuming that all space emits a uniform light occluded by opacity voxels. It then uses a diffuse cone tracing pass to compute the ambient occlusion similarly to the Voxel Global Illumination method.

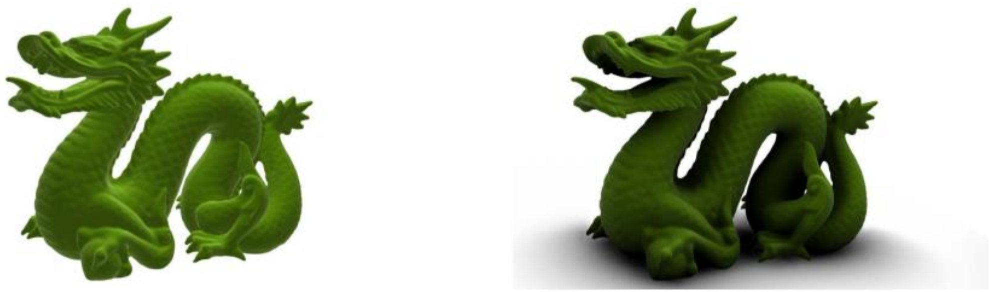


Figure 17: 3D model without ambient occlusion (left), and with ambient occlusion (right). The technique demonstrated is HBAO.

Due to the high computational cost of the global illumination algorithms, multiple techniques have been developed that attempt to recreate photorealistic properties as efficiently as possible. These techniques are designed to be used in real time applications like video games, however, in order to generate results in real time, the techniques tackle specific problems by providing approximations of global illumination elements. Some of those techniques are:

- Reflective Shadow Maps [12]: an extension to the standard shadow mapping technique, reflective shadow maps consider every pixel to be an indirect light source, therefore approximating indirect illumination to the 3D scene.

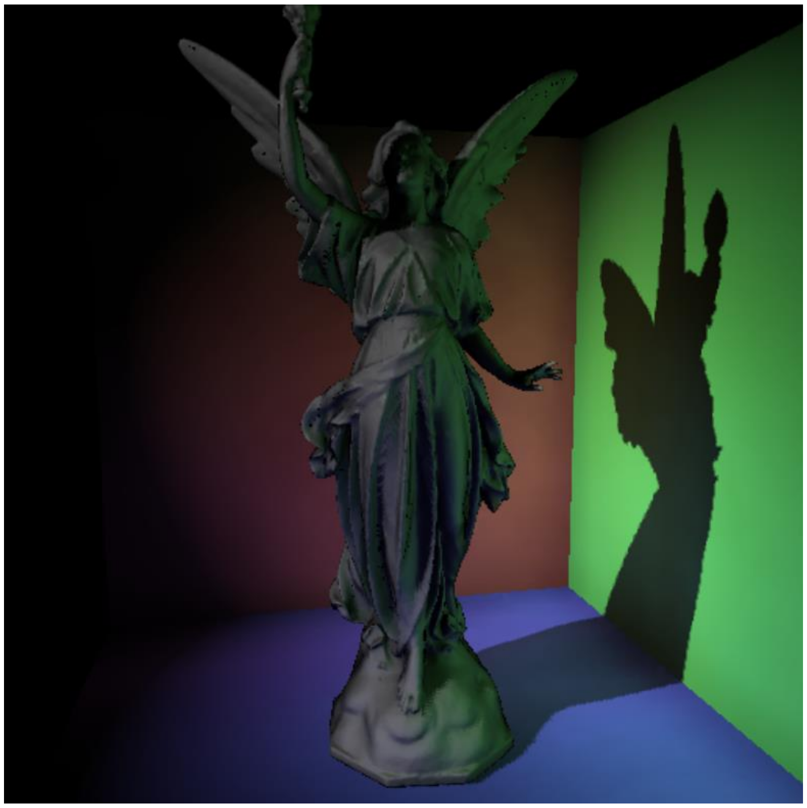


Figure 18: Reflective Shadow Maps

- Screen space directional occlusion (SSDO) [2]: SSDO is an extension to the SSAO technique. It computes both the ambient occlusion in a scene as well as an indirect illumination bounce to nearby surfaces, providing a presentable global illumination approximation in real time. The technique is discussed in more depth in section 3.2.3.

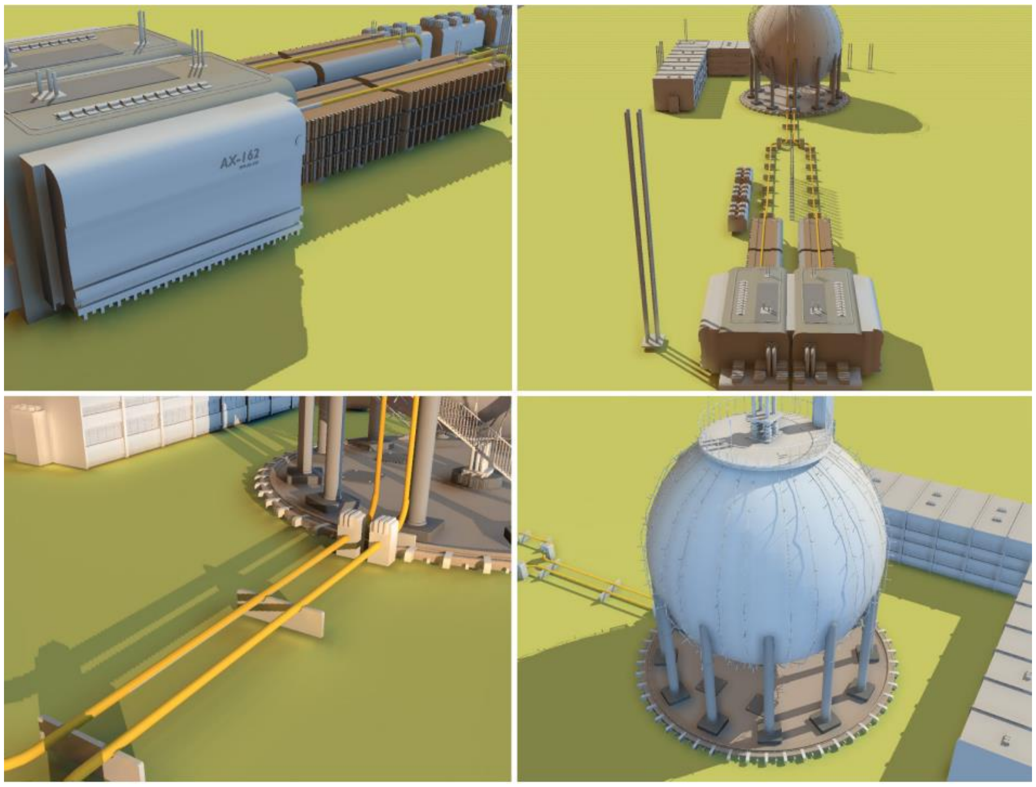


Figure 19: SSDO

- Image-based lighting (IBL) [13]: A technique used to illuminate scenes and objects using chromatic information from panoramic and high dynamic range image textures that are projected onto a sphere. When used effectively, objects reflecting the projected image appear more realistic and share more consistent indirect illumination.

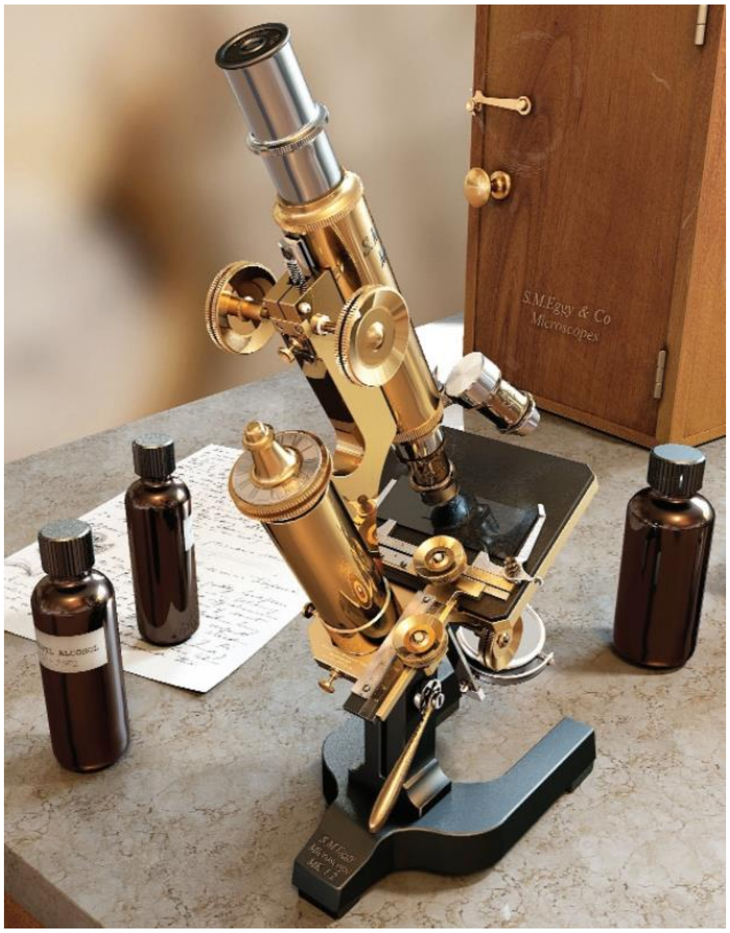


Figure 20: Scene rendered in Arnold rendering system, illuminated by an omnidirectional image of a kitchen interior, providing consistent indirect lighting and reflections.

3.2.2 Screen space ambient occlusion

In 2007, Crytek published a variety of methods that were implemented in the second release of the Cryengine game engine [14]. One of those methods was Screen Space Ambient Occlusion (SSAO), which is an algorithm to compute ambient occlusion in a 3D scene at run-time in screen space.

SSAO uses the depth of the camera to compute the occlusion factor of every pixel [10] by randomly sampling the surrounding pixels and comparing their depth values. As described in [14], the algorithm for the SSAO effect is as follows:

- For each pixel (p) in the scene:
    - Calculate the depth and the normal vector (N).
    - For each randomly sampled neighboring pixel (p_s):
        - Get the sample's depth and the vector between (p) and (p_s), (V).
        - Calculate the depth difference (d) between (p) and (p_s).
        - Calculate the occlusion using the following formula:
          occlusion = max(0, dot(N, V)) * (1 / (1 + d))
          and add it to the sum of occlusion factors (o).
    - Divide (o) by the number of sampled pixels.

This technique outputs a version of the original image that is ambiently lit and shaded accordingly. However, without any extra processing, the shaded parts introduce noise to the final image. Therefore, it is common practice to apply a blur pass to the SSAO output before blending it with the final image.

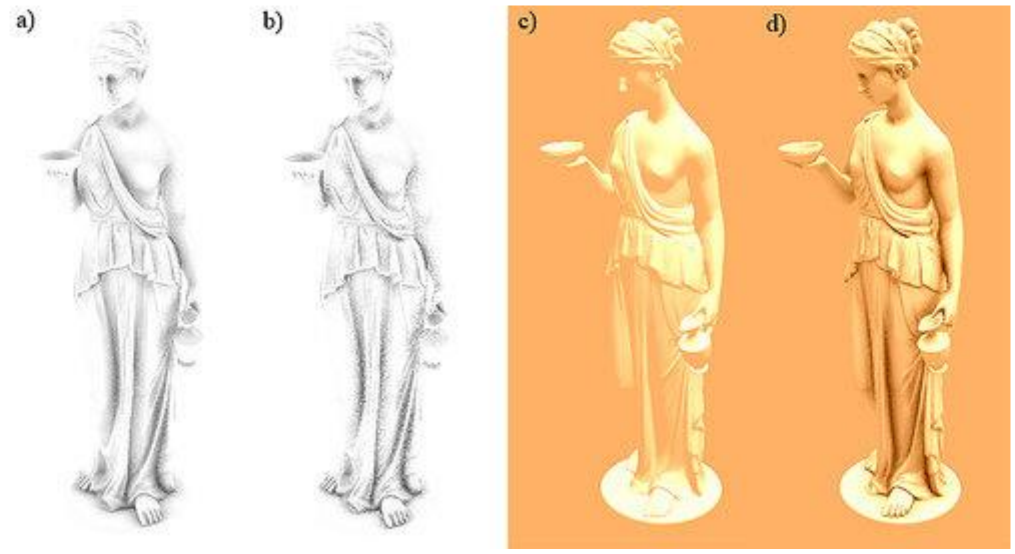


Figure 21: a) Raw SSAO output using 16 samples b) Raw SSAO output using 8 samples c) Directional light only d) Directional light + SSAO using 2 passes with 16 samples each

While the SSAO technique enhances the visual result of a scene, it has some restrictions:

- In some cases, the SSAO algorithm can present artifacts in a scene.
- Since the effect is calculated in screen space it is not consistent as the scene camera moves.
- It does not output physically accurate results, as the ambient occlusion is not affected by any direct or indirect light sources in the scene.

3.2.3 Screen space directional occlusion

T. Ritschel, T. Grosch and H. Seidel presented the paper “Approximating Dynamic Global Illumination in Image Space” in 2009 [2] where they introduced “screen space directional occlusion” (SSDO), a screen-space algorithm to approximate global illumination techniques like ambient occlusion and indirect light bounces.

SSDO is an extension of the SSAO method with some major differences:

- SSDO takes the direct illumination direction into consideration while computing the occlusion factor of the scene.
- The SSDO technique implements an indirect illumination bounce to approximate the color bleeding that occurs through global illumination.

The technique is more easily implemented in an environment that utilizes deferred rendering, as the access to lighting and color information through G-Buffers significantly facilitates the necessary calculations.

SSDO is implemented through the following steps:

- For each pixel (p) on the screen:

- Calculate (p)'s depth, normal vector, world-space position (w), and diffuse color. The color and lighting information can be acquired through the G-Buffer provided by the graphics rendering environment. The world-space position can be either derived from a world-space position texture or it can be reconstructed from the depth information of (p).
- For each randomly sampled neighboring pixel (p_s):
    - Get (p_s)'s depth, normal vector, world-space position (w_s), diffuse color (c_s) and lighting information (L_s) in a similar way to (p).
    - Calculate the occlusion factor (o_s) as mentioned in the SSAO algorithm.
    - Calculate the distance (d) between (w) and (w_s).
    - Calculate the angle between the normal vector of (p) and the sampling direction (θ_1).
    - Calculate the angle between the normal vector of (p_s) and the sampling direction (θ_2).
    - Multiply (o_s) by the lighting value of the pixel (L_s) so that illuminated areas are not affected as much by the occlusion. Then add (o_s) to the sum of occluding factors (o).
    - Calculate the indirect illumination factor using the formula:
      $L_{indirect} = c_s * L_s * (1 - o_s) * \theta_1 * \theta_2 / d * d$
      and add it to the sum of indirect illumination factors (L).
- Divide (L) and (o) by the number of sampled pixels.

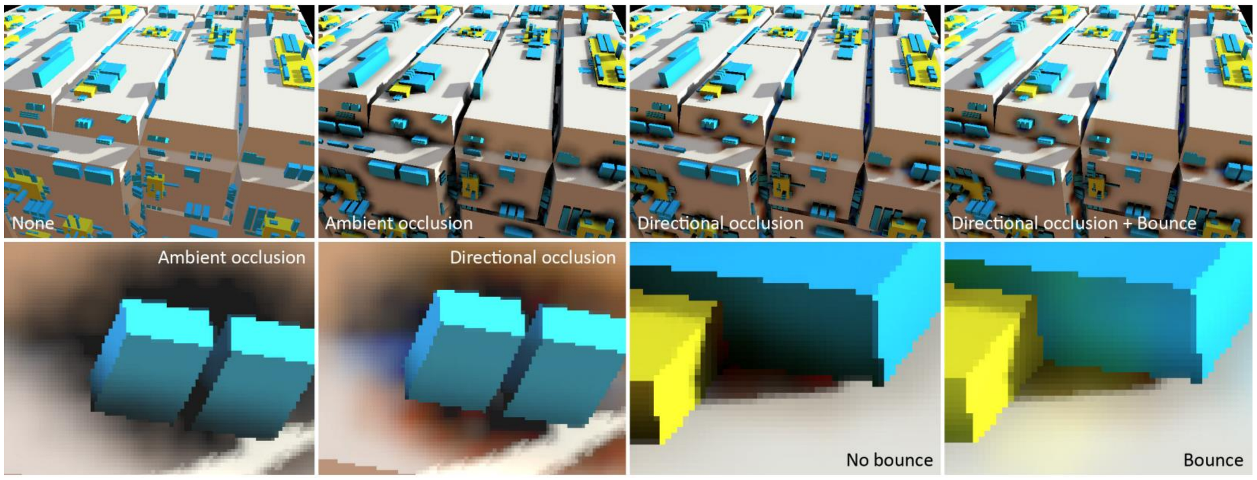


Figure 22,Comparison of directional occlusion with ambient occlusion

Similar to SSAO, the result of the algorithm above outputs a noisy result which needs a blurring pass in order to be blended with the final image. The blurring algorithm chosen for the blur pass needs to utilize the scene's depth and normal information in order to preserve geometric information, like the technique discussed in [15].

The paper describing SSDO suggests that the effect be used alongside other global illumination techniques like ray-tracing and radiosity algorithms which provide a better indirect lighting approximation. The other techniques can be pre-computed offline for static objects, so that there is minimal additional computational overhead on runtime, and the SSDO algorithm can be then applied so that dynamic objects also contribute to the global illumination approximation in the scene.

SSDO presents the following limitations:

- As SSDO follows the same sampling scheme as SSAO, it inherits the limitations of the technique.
- Since the indirect bounce is computed in screen-space, objects that occupy less screen space in certain viewpoints can have a biased indirect illumination result, which is not an accurate representation of global illumination. The paper suggests a multi-camera setup to capture and combine the indirect illumination coming from an object through multiple angles.

3.3 Implementation methodology

3.3.1 Introduction

This section includes a detailed overview of the methodology that was used to implement and integrate the SSDO technique into the rendering pipeline of Unity 3D.

3.3.2 Methodology overview

The main steps taken for the SSDO implementation are as follows:

- SSAO implementation
- SSDO implementation
- Multi-pass shader configuration
- Refactoring, additional features and optimization

Each step of the process contributed to the further understanding of the engine's rendering pipeline as well as to the overall visual result. Both SSAO and SSDO were implemented by following the respective technique's description as described in 3.2.1 and 3.2.2 and adjusting them to the context of Unity's custom shader creation workflow.

3.3.3 Implementing Screen Space Ambient Occlusion

Since the SSDO technique relies heavily on SSAO, it was a reasonable step to first attempt an implementation of the SSAO algorithm in Unity 3D. The effect was created so that it works with both the forward and deferred rendering paths as it does not rely on G-Buffers to receive necessary information. For this implementation, only the camera's depth and view-space normal were needed. That information is stored in the camera's depth and normal texture and, thus, retrieving that information was a task that required sampling a single texture.

A method called "GetPixelValue" was created in order to get a pixel's depth and view-space normal vectors. The function gets a pixel's screen coordinates and returns a 4-dimensionan vector that stores the normal information in its x, y and z components and the pixel's depth in its w component.

Using a loop in the fragment shader, random samples were picked to examine their ambient occlusion contribution. Initially, the samples were picked by sampling a colored white noise texture, where each texel's color information corresponded to a direction by considering the red, green and blue channels of the texel's color as the x, y and z components of the direction's vector respectively. The initial implementation of that method presented a very structured sampling pattern; therefore, it was later changed to a method utilizing sampling from a spherical kernel containing uniformly distributed points. [16]

Using the “GetPixelValue” method the view-space normals and depth are calculated for the pixel rendered by the fragment shader as well as for every pixel sampled from the spherical kernel. The depth values are used to determine the depth difference between the occludee and the sampled pixel, while the normals are used to calculate the angle between the two pixels to utilize the SSAO formula:

$$occlusion = \max \frac{\left(0.0, dot(N, V)\right)}{1.0 + d}$$

where “N” is the occludee’s normal values, “V” is the vector between the occludee and the sampled pixel and “d” is the depth delta between the occludee and the sampled pixel.

The occlusion value is aggregated to a total sum of occlusion factors which is afterwards divided by the number of sampled pixels.

The code for the SSAO shader and the script accompanying it can be found in sections 5.1.1 and 5.2.1 respectively.

3.3.4 Implementing Screen Space Directional Occlusion

Since SSDO shares many similarities to SSAO, the implementation used the SSAO shader as the basis. Most of the extra computations happen in the sampling loop, therefore the core of the fragment shader remains unchanged and the random sampling technique is the same as the one used in SSAO. The additional elements that are needed for SSDO can be calculated along with the occlusion factor, as they also depend on information derived from the sampled pixel.

The necessary information that is needed to calculate the indirect illumination contribution is:

- The world-space position of the current pixel.
- The world-space position of the randomly sampled pixel.
- The diffuse color of the current pixel.
- The diffuse color of the randomly sampled pixel.
- The lighting information of the randomly sampled pixel.

Since Unity 3D does not support world-space position textures, the position for both the current and the sampled pixel must be reconstructed using their depth information and a process called “inverse projection”.

In order to access the color and lighting information of the scene, the shader needs to sample the corresponding G-Buffers and, therefore, this implementation of SSDO only supports the deferred rendering path.

Unity 3D provides four (or in some cases five) G-Buffer textures in its rendering pipeline:

1. _CameraGBufferTexture0: Holds the diffuse and occlusion information.
2. _CameraGBufferTexture1: Holds the specular color and roughness information.
3. _CameraGBufferTexture2: Holds the world-space normal vectors.
4. _CameraGBufferTexture3: Holds the illumination information, including emission, lighting, lightmaps and reflection probes.
5. _CameraGBufferTexture4: If the “Shadowmask” or “Distance Shadowmask” modes for mixed lighting are used, this buffer holds the light occlusion values.

The color contribution of each sampled pixel for the indirect bounce depends on the diffuse color of the surface (sampled from G-Buffer 0), the light color the surface receives as well as the emissive color it emits (both sampled from G-Buffer 3). Therefore, the formula chosen to determine the color of the indirect illumination is as follows:

$$C_{indirect\ illumination} = \left(C_{diffuse} + C_{direct\ illumination}\right) \cdot C_{direct\ illumination}$$

The multiplication with the direct illumination color is needed as the indirect light bounces off surfaces that receive more direct illumination. It is also necessary to add it to the diffuse color, as the same information holds the emissive color of an object. Therefore, if an object has a dark diffuse color but it serves as a source for emissive color, that color should contribute to the indirect illumination. The diffuse color of the current pixel is also sampled from G-Buffer 0 outside the sampling loop in order to be multiplied with the indirect illumination color.

The distance between the world-space position of the current pixel and the sampled pixel is then calculated, as well as the two cosines that are needed for the calculation of the radiance from the surrounding geometry:

1. The cosine between the sampled pixel's normal vector and the vector between the current pixel and the sampled pixel.
2. The cosine between the current pixel's normal vector and the vector between the current pixel and the sampled pixel.

The indirect illumination is then calculated using a formula based on the formula described in [2]:

$$L_{ind} = \frac{C_{indirect\ illumination} \cdot \cos\theta_1 \cdot \cos\theta_2}{d^2}$$

where $\cos\theta_1$ and $\cos\theta_2$ the cosines mentioned above and d the distance between the world-space position of the current pixel and the sampled pixel.

As with the occlusion factor, the indirect illumination is also added to a total value which is then multiplied by the current pixel's diffuse color and then divided by the number of random samples.

The complete code of the SSDO implementation can be found in the appendix.

3.3.5 Multi-pass shader configuration

While the processes for both SSAO and SSDO resulted in the expected output, neither of those effects can be applied in a real time application, since these shaders produce a lot of noisy artifacts when used on their own, due to the random sampling pattern they use. Therefore, it is essential for the overall visual result that the output of each technique is pre-processed before it is applied to the final image. In the case of both SSAO and SSDO, the necessary pre-processing was a filtering (blurring) pass to reduce the noisy artifacts generated from the sampling process.

The process of adding a blur pass was challenging at first, as Unity does not provide a way to perform the SSAO and SSDO computations and blur the result in just one pass. The solution for this problem was the utilization of a multi-pass shader setup.

The concept behind the multi-pass configuration is simple:

- Define multiple passes in the shader, each with a specific functionality and purpose.
- Define a pipeline in the C# script that applies the effect to the camera which describes the sequence in which the effects are applied using temporary render textures.

Three main passes were needed for both the screen-space ambient occlusion and the screen-space directional occlusion shaders:

1. A pass to calculate the corresponding effect (SSAO or SSDO respectively).
2. A pass to blur the result of the previous pass.
3. A composition pass that combines the blurred result with the original rendered image (before the effects were applied to it) to provide the final image.

Those passes also determined the sequence followed by the C# scripts:

1. Render the effect (SSAO or SSDO) and store it in a temporary render texture (RT_1).
2. Blur (RT_1) and store the result in a new temporary render texture (RT_2).
3. Combine the original rendered image with (RT_2) and return that as the final image on the screen.

The blur pass that was used is a geometry-aware technique [17] that preserves hard edges and therefore does not alter the overall shape of the geometry in a scene. This effect is achieved by sampling the camera’s depth and normal vectors texture for every pixel processed in the fragment shader as well as its neighboring pixels and applying a blurring factor that depends on the difference of the pixels’ depth and normal vectors.

The composition pass depends on each technique. As far as the screen-space ambient occlusion is concerned, the final SSAO texture after the blur pass consists of a grayscale image where the darker areas illustrate the more occluded areas in the scene. The effect is composited with the original image by multiplying the two textures, so that the more occluded areas will be darker in the final image and the areas that did not receive any occlusion will remain unchanged.

Regarding the SSDO, the texture passed in the composition step contains information for both the directional occlusion and the color of the indirect bounce by storing the color information in the red, green and blue channels of the texture and the directional occlusion value in the alpha channel. The final image is then composited by adding the indirect illumination of the SSDO to the original image and multiplying the result by the occlusion factor.

3.3.6 Refactoring, additional features and optimization

To further enhance the usability and the performance of both the SSAO and SSDO effects some additional features needed to be implemented in the shaders and the respective scripts:

- A fourth pass was added to each effect to provide for a debug visualization of each effect, that can be toggled via the C# script. For the SSAO it shows the ambient occlusion contribution on top of a white background without any chromatic information. For the SSDO, the debug visualization shows both the ambient occlusion contribution in a scene and the indirect illumination that bounces off illuminated surfaces.

- Additional optimizations for the SSAO and SSDO were performed concerning the blur pass. Since the temporary render texture that was a result of the SSAO or SSDO algorithm is getting blurred to reduce the noisy artifacts, it is possible to reduce the resolution of said render texture in order for the blur pass to perform less calculations. Since the result of the operation is blurred, as long as the downscaling of the resolution is not too extreme, the visual output of the blurring pass does not get significantly altered, while there can be a very noticeable performance gain.
- For improved readability, the SSAO and SSDO shaders are separated into segments, where every segment has its own vertex function (if it is not sharing one with another segment) and its own fragment function. Furthermore, the passes of each shader are decoupled from the corresponding vertex and fragment functions to better illustrate the separate passes and allow for further code refactoring, like keeping each shader segment into a separate CG include file.
- The ambient occlusion sampling radius of the SSDO effect was decoupled with the sampling radius of the indirect illumination so that they could be modified separately. While this separation introduces a small computational overhead, it provides more flexibility as far as the desired results are concerned, since that way, the occlusion amount can vary from the indirectly illuminated area.

3.4 Results

3.4.1 Introduction

This section contains screenshots that demonstrate the SSAO and SSDO implementations in different scene environments. The breakfast room, cathedral, conference room, Cornell box and fireplace room models downloaded from Morgan McGuire's Computer Graphics Archive (https://casual-effects.com/data).

All the indoors scenes are using Unity’s pre-computed real time global illumination to demonstrate how the SSAO and SSDO techniques contribute to the visual outcome of a scene that already uses other, non-dynamic global illumination methods.

The properties of each effect have been modified to suit the needs of each scene, and they have been set to maximum quality in order to better illustrate the capabilities of both SSAO and SSDO.

No post-processing has been added to the scenes besides FXAA anti-aliasing to compensate for the lack of MSAA anti-aliasing due to the use of the deferred rendering path.

3.4.2 SSAO

Below are screenshots taken from inside Unity 3D using the implementation of the SSAO technique:

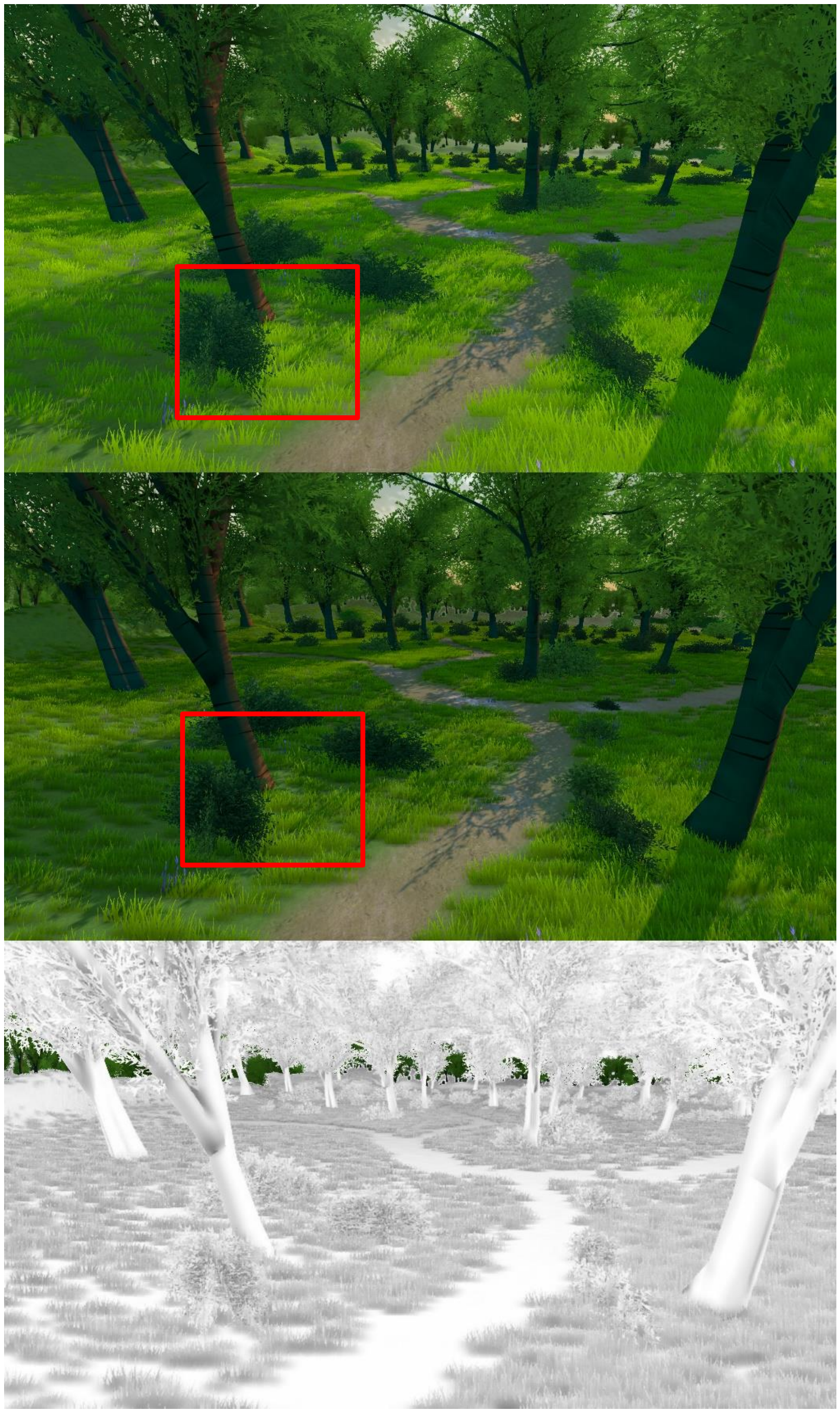


Figure 23: Forest scene. From top to bottom: Scene without SSAO, scene with SSAO, SSAO debug view showing only the effect's contribution.

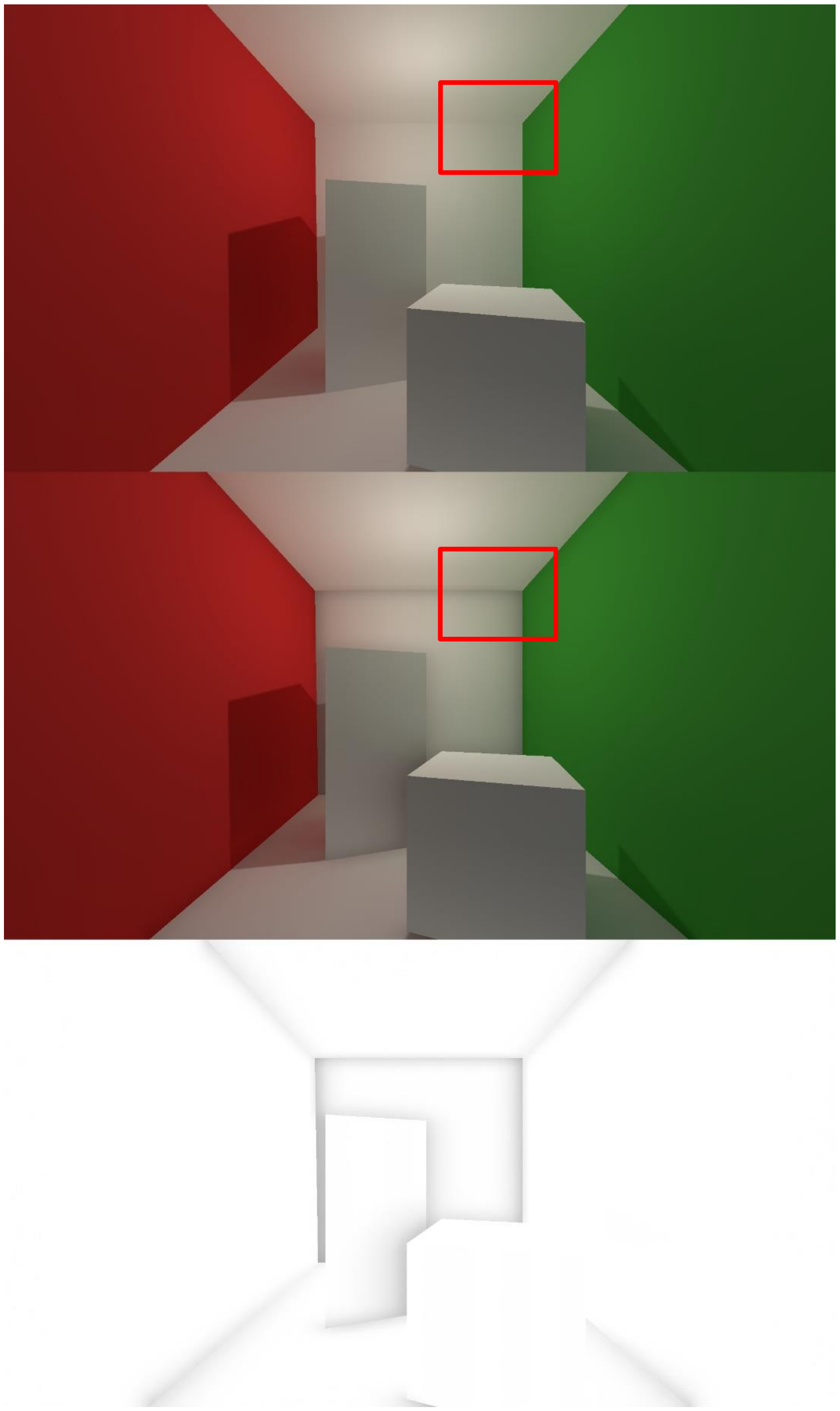


Figure 24: Cornell box. From top to bottom: Scene without SSAO, scene with SSAO, SSAO debug view showing only the effect's contribution.

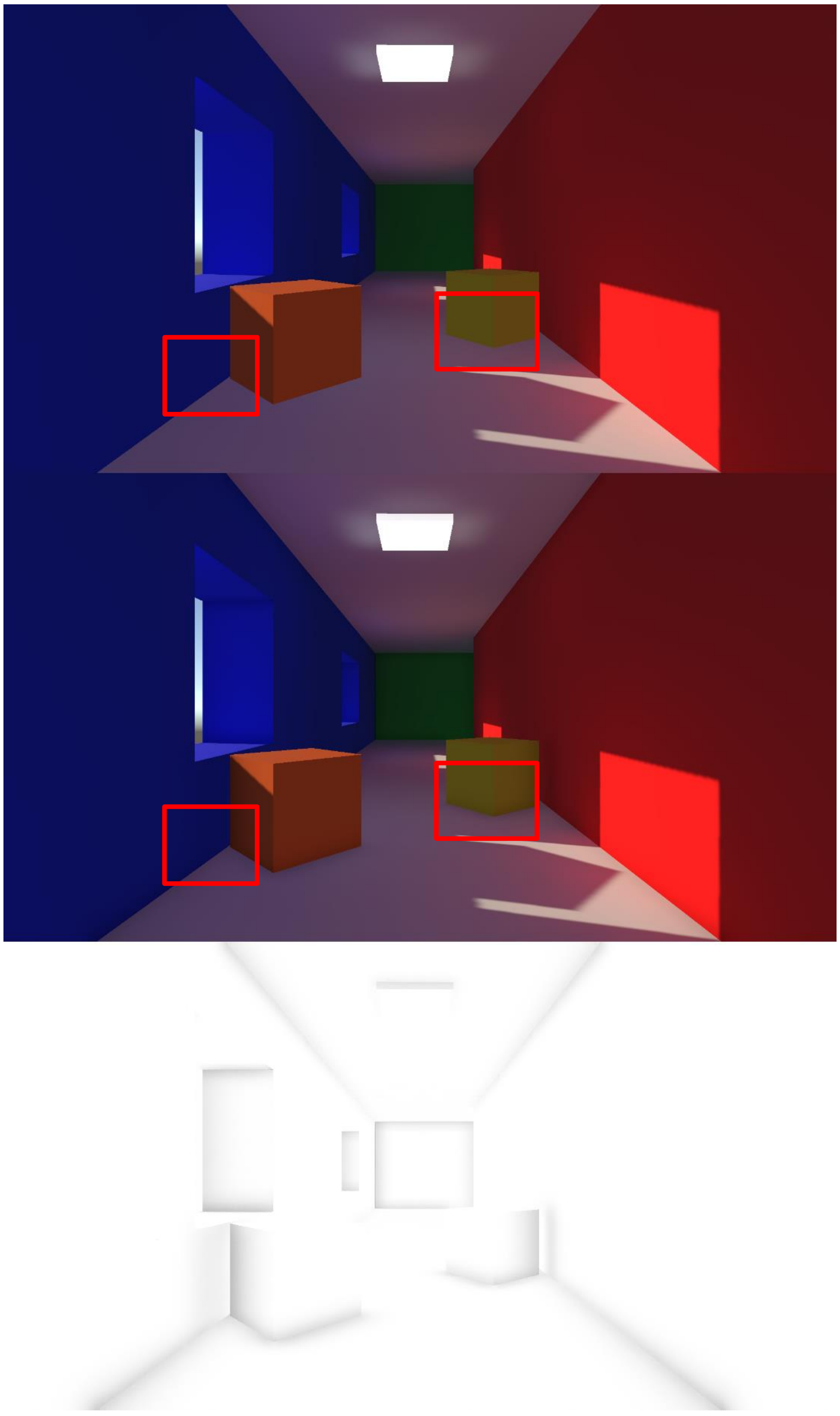


Figure 25: Colored hallway. From top to bottom: Scene without SSAO, scene with SSAO, SSAO debug view showing only the effect's contribution.

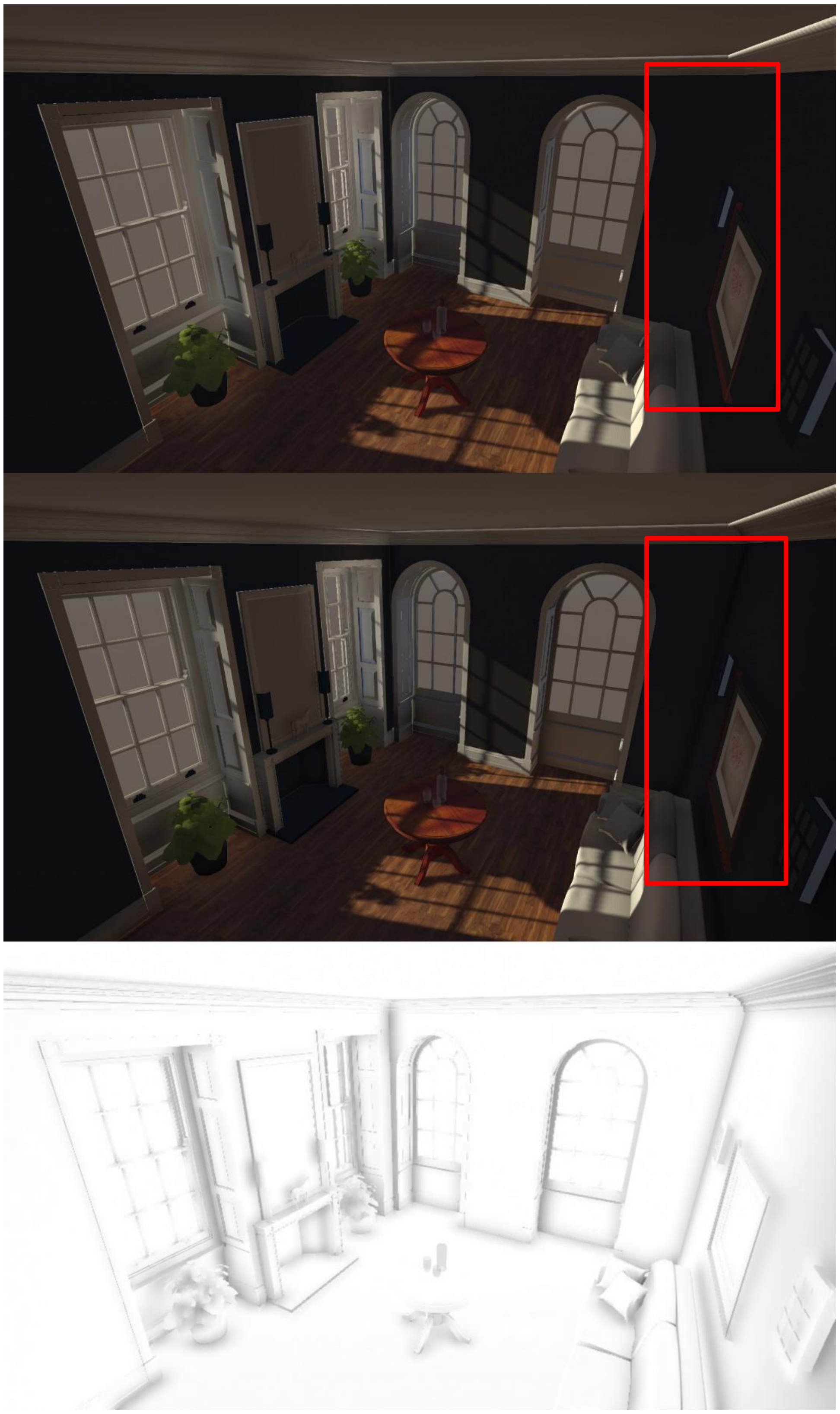


Figure 26: Fireplace room. From top to bottom: Scene without SSAO, scene with SSAO, SSAO debug view showing only the effect's contribution.

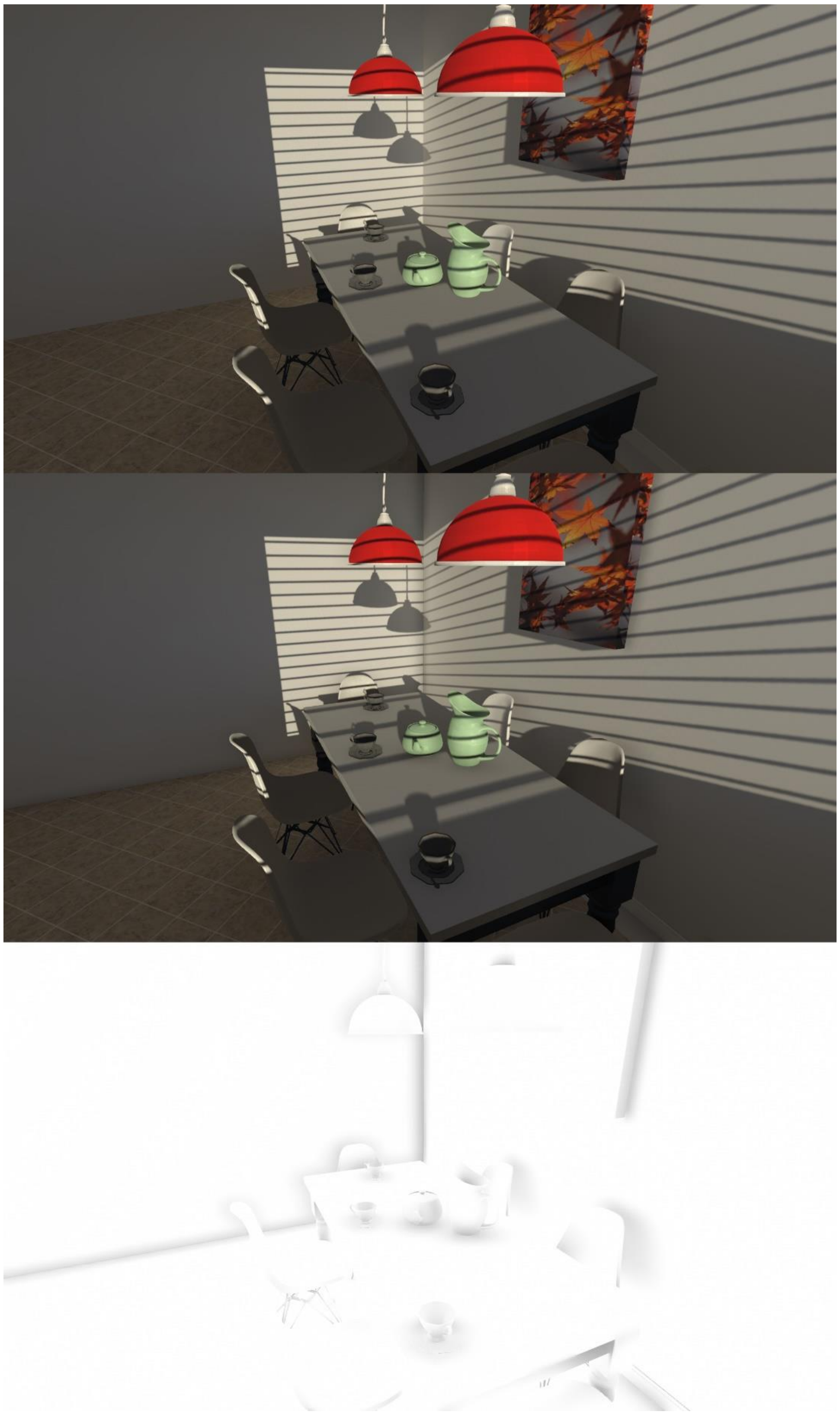


Figure 27: Breakfast room. From top to bottom: Scene without SSAO, scene with SSAO, SSAO debug view showing only the effect's contribution.

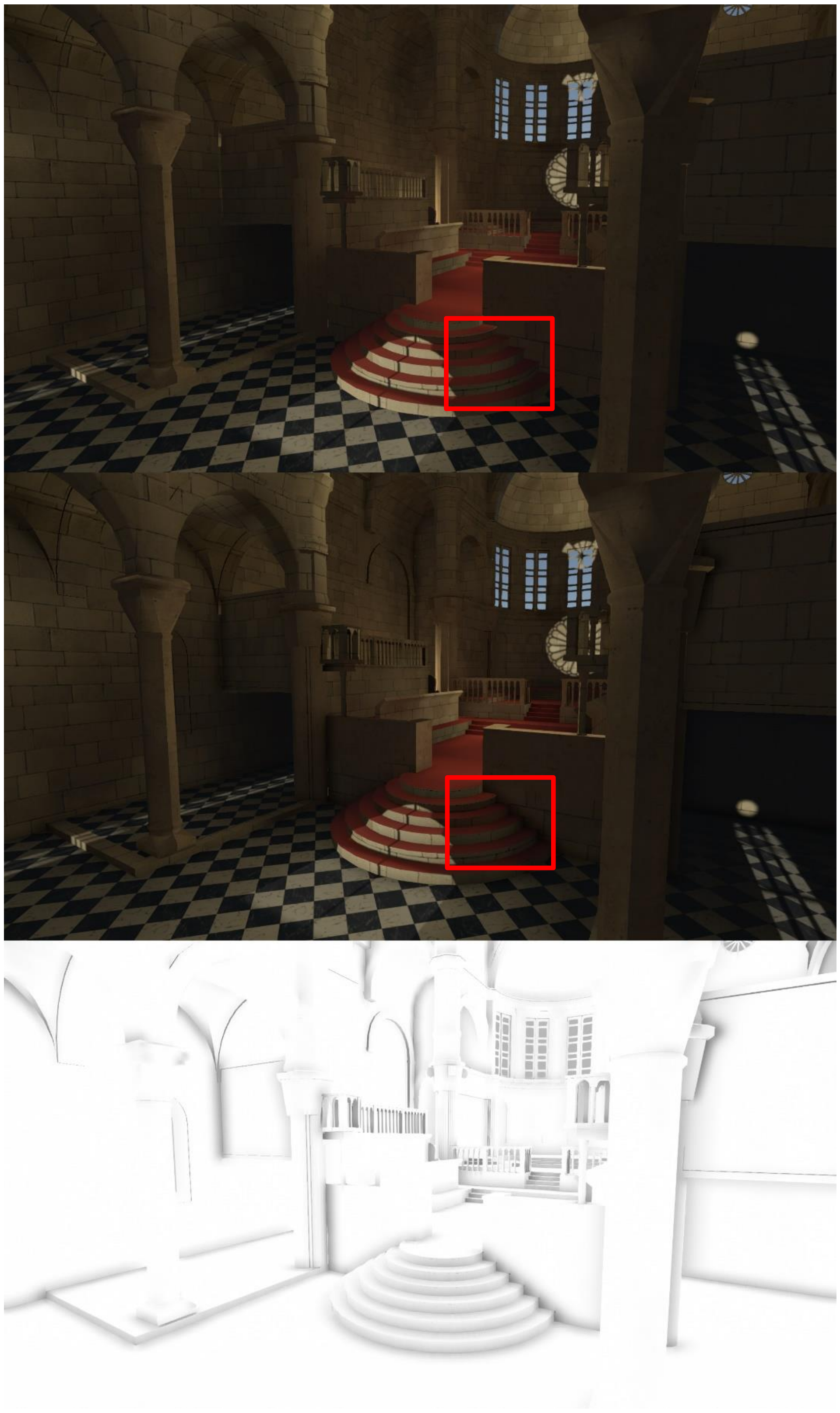


Figure 28: Cathedral. From top to bottom: Scene without SSAO, scene with SSAO, SSAO debug view showing only the effect's contribution.

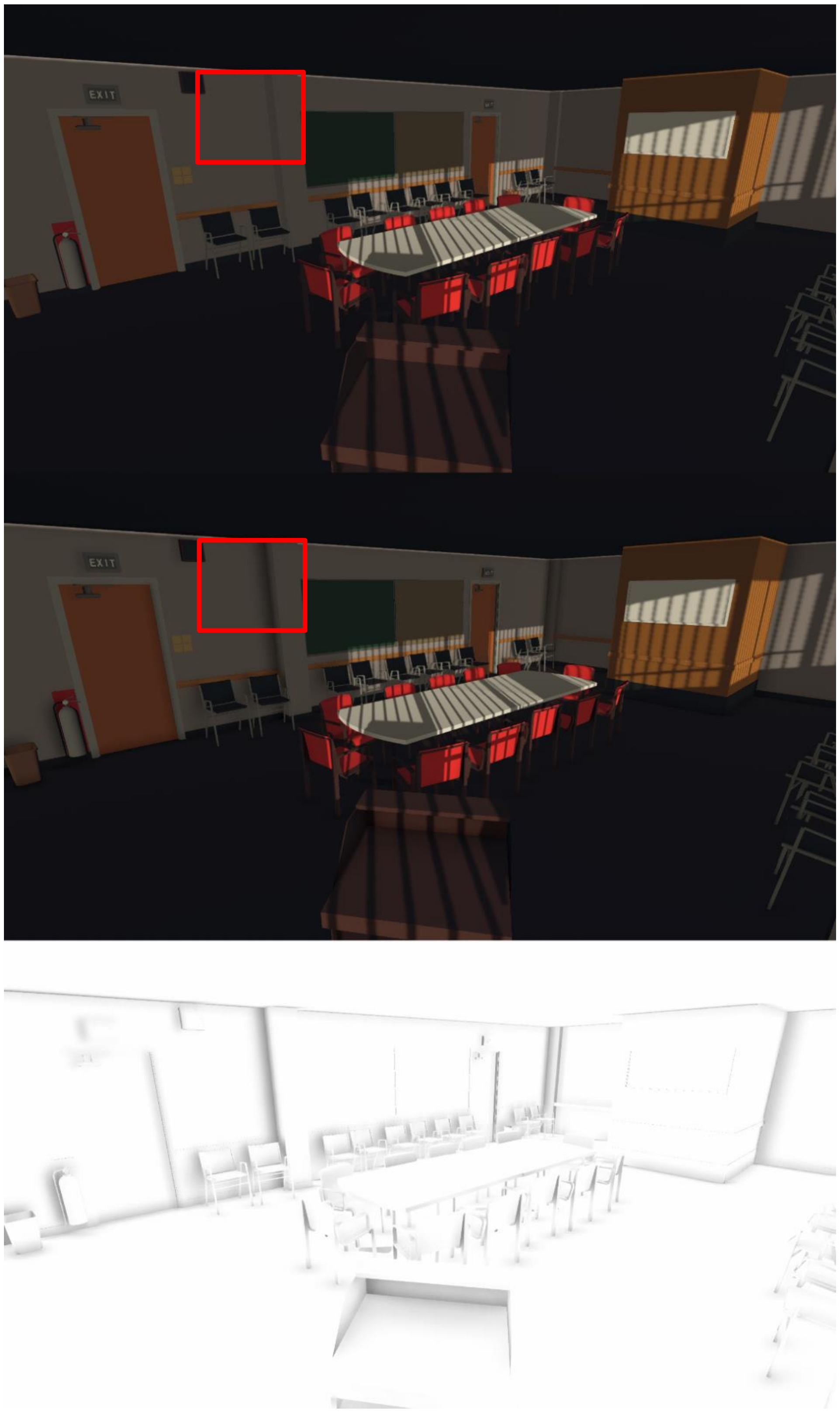


Figure 29: Conference room. From top to bottom: Scene without SSAO, scene with SSAO, SSAO debug view showing only the effect's contribution.

3.4.3 SSDO

Below are screenshots taken from inside Unity 3D using the implementation of the SSDO technique:

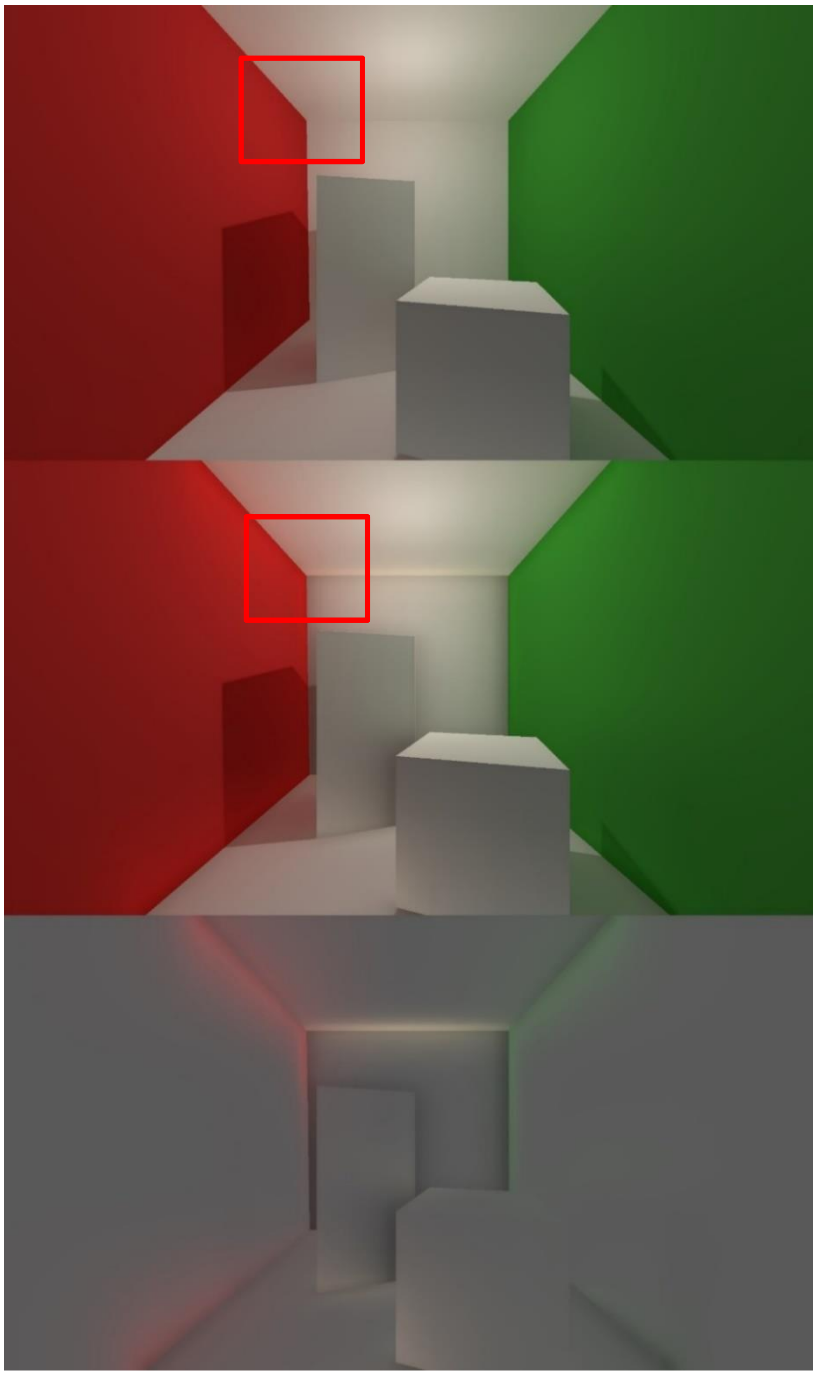


Figure 30: Cornell box. From top to bottom: Scene without SSDO, scene with SSDO, SSDO debug view showing only the effect's contribution.

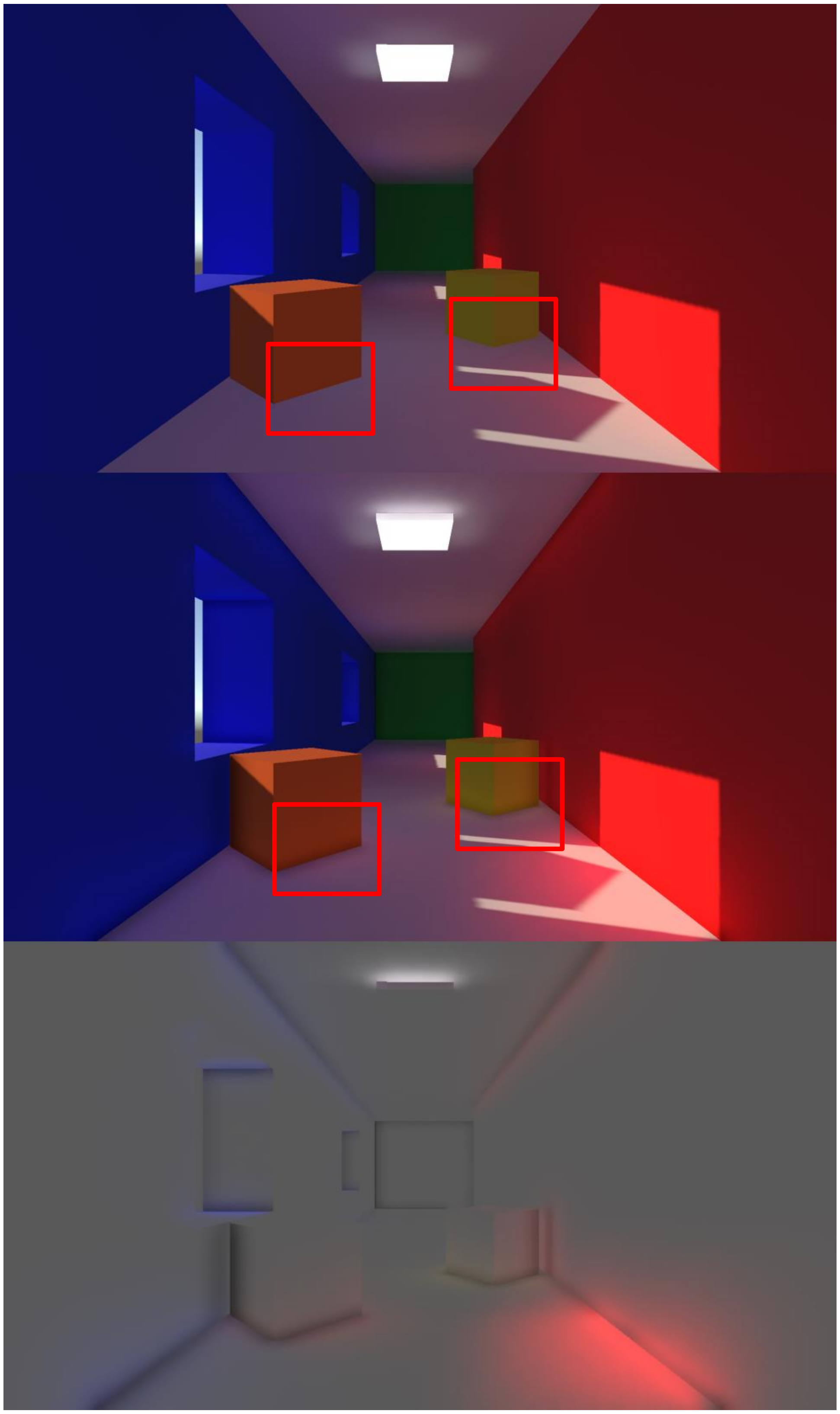


Figure 31: Colored hallway. From top to bottom: Scene without SSDO, scene with SSDO, SSDO debug view showing only the effect's contribution.

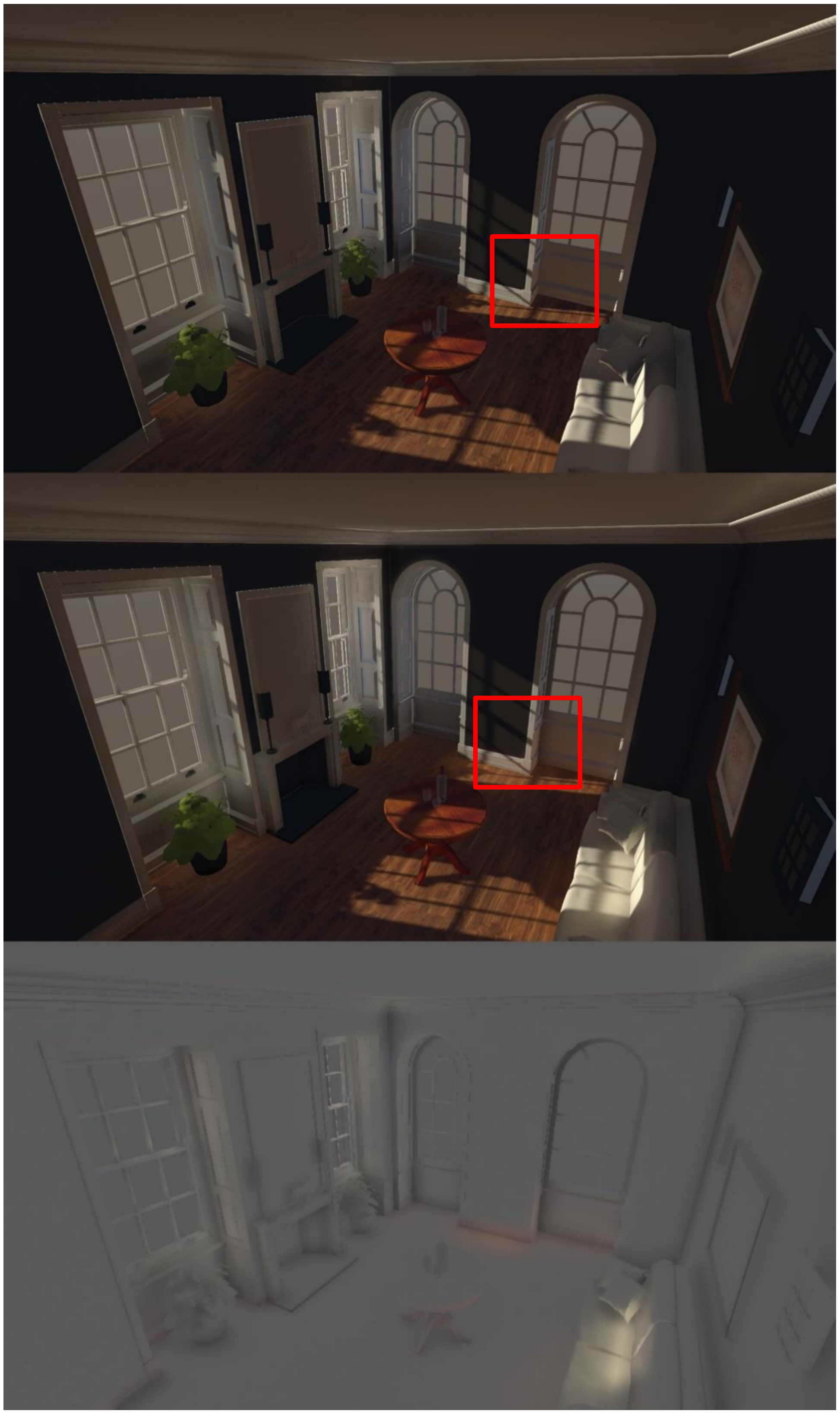


Figure 32: Fireplace room. From top to bottom: Scene without SSDO, scene with SSDO, SSDO debug view showing only the effect's contribution.

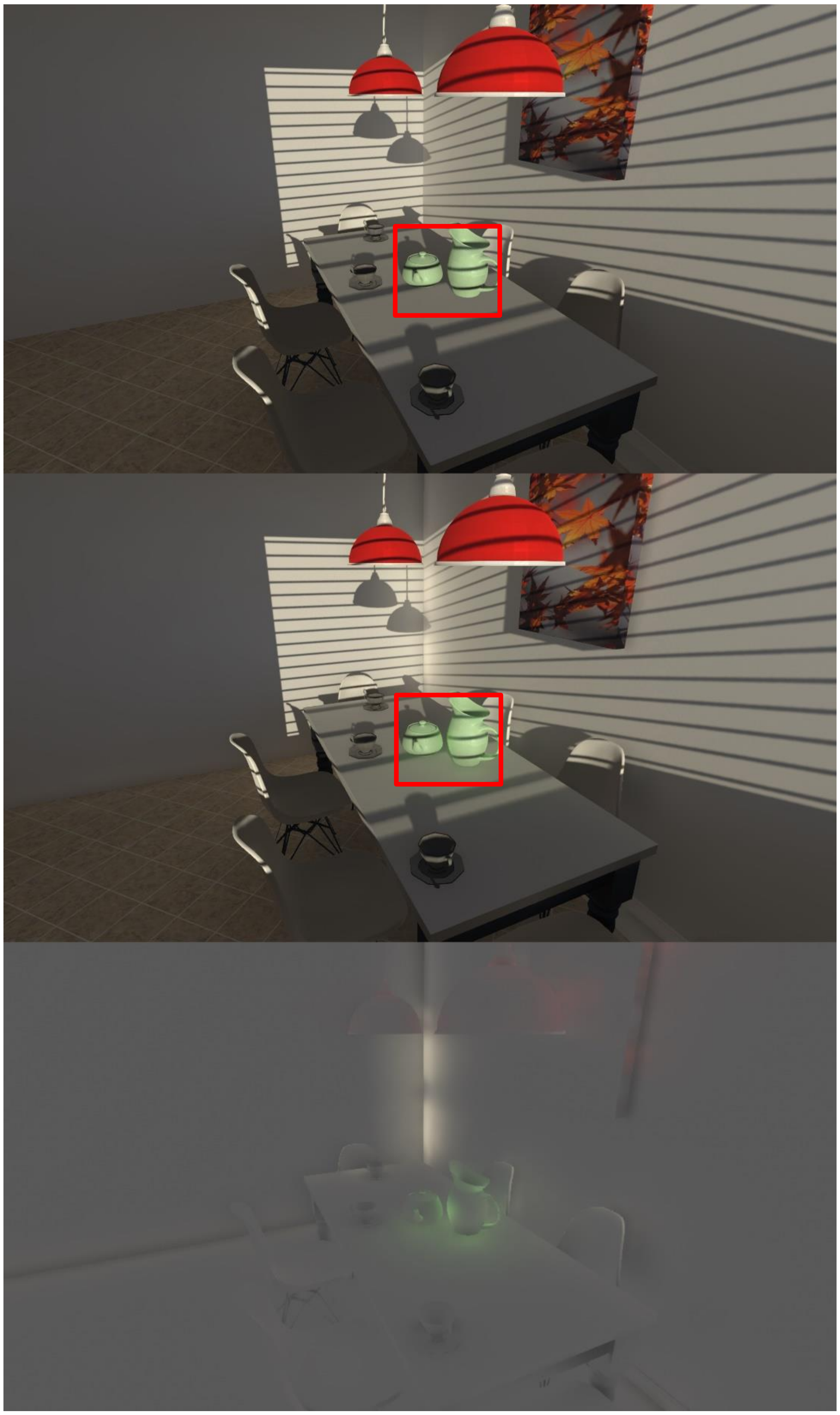


Figure 33: Breakfast room. From top to bottom: Scene without SSDO, scene with SSDO, SSDO debug view showing only the effect's contribution.

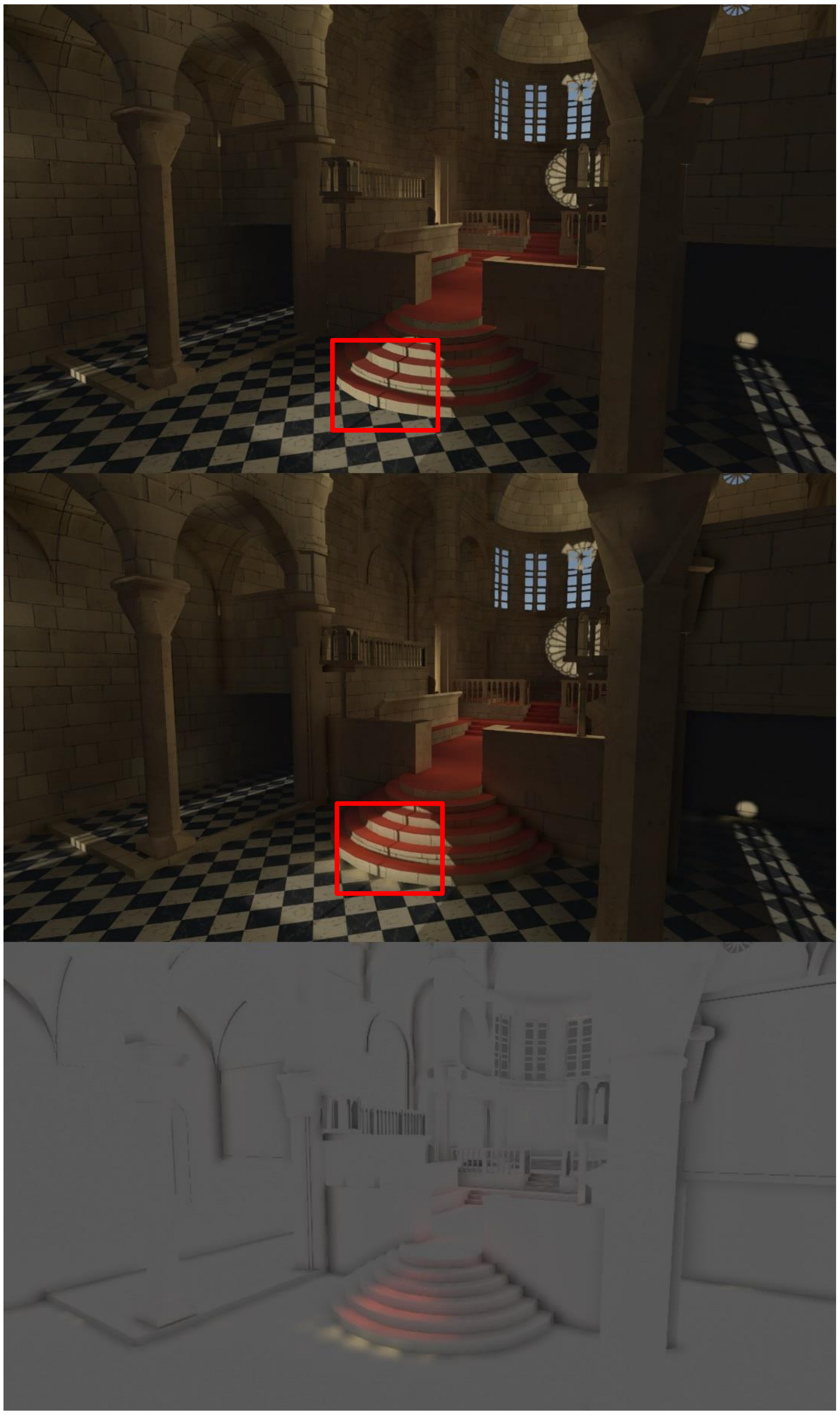


Figure 34: Cathedral. From top to bottom: Scene without SSDO, scene with SSDO, SSDO debug view showing only the effect's contribution.

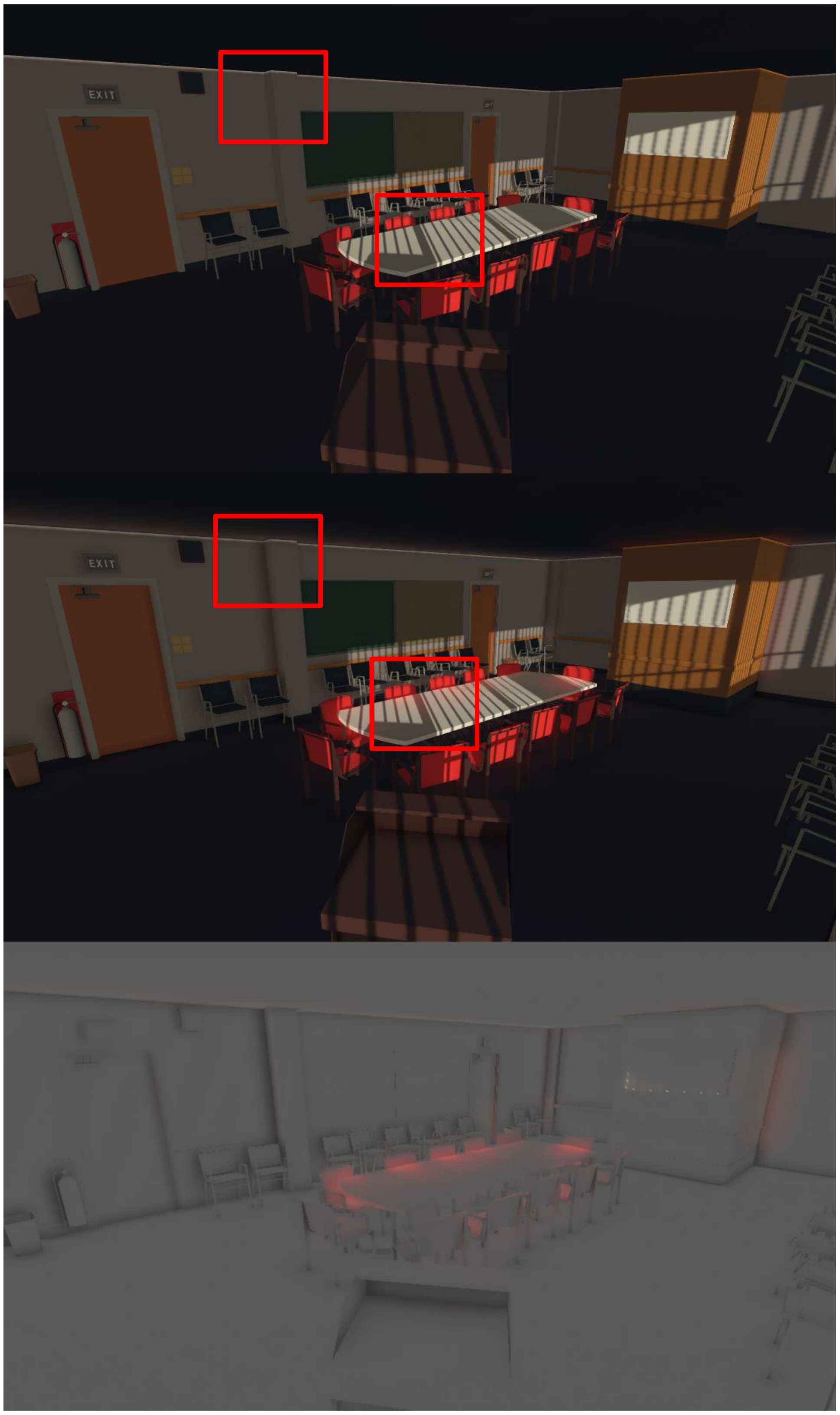


Figure 35: Conference room. From top to bottom: Scene without SSDO, scene with SSDO, SSDO debug view showing only the effect's contribution.

3.5 Conclusion

Multiple techniques have been developed to enhance the visual fidelity of real time graphics in modern game engines. Their challenge is to strike a balance between providing accurate, physically-based results and being efficiently implemented in a real time graphics rendering pipeline. A significant factor of photorealistic graphics rendering lies in the illumination techniques that, in a real time rendering context, can either be techniques for direct (local) or indirect (global) illumination in a scene. Direct illumination algorithms are faster but do not take the whole scene into account, while indirect illumination techniques offer more realistic results but are more expensive and are thus used to pre-calculate illumination information for static objects in the scene. However, there have been algorithms that can be executed in real time and offer an approximation of a specific global illumination aspect, like ambient occlusion or radiosity. Two of those techniques are screen space ambient occlusion and screen space directional occlusion, which are used to calculate ambient occlusion and indirect bounced illumination in real time respectively. Using a modern game engine, like Unity 3D, it is possible to implement those techniques to extend the graphics pipeline and apply them alongside other real time or offline illumination algorithms.

APPENDIX

Shader code

SSAO shader code

```
Shader "Dissertation/SSAO"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
    }

    CGINCLUDE

        #include "UnityCG.cginc"

        struct v2f
        {
            float2 uv : TEXCOORD0;
            float4 vertex : SV_POSITION;
        };

        v2f vert (appdata_img v)
        {
            v2f o;
            o.vertex = UnityObjectToClipPos(v.vertex);
            o.uv = v.texcoord;
            return o;
        }


        float _Bias;
        float _Power;
        float _SampleCount;
        float _Radius;

        sampler2D _MainTex;
        sampler2D _CameraDepthNormalsTexture;

        float4 GetPixelValue(in float2 uv) {
            half3 normal;
            float depth;
            DecodeDepthNormal(tex2D(_CameraDepthNormalsTexture, float2(uv.x, uv.y)), depth,
normal);
            return fixed4(normal, depth);
        }

        float nrand(float2 uv, float dx, float dy){
            uv += float2(dx, dy + _Time.x);
            return frac(sin(dot(uv, float2(12.9898, 78.233))) * 43758.5453);
        }

        float3 spherical_kernel(float2 uv, float index){
            // Uniformly distributed points
            // http://mathworld.wolfram.com/SpherePointPicking.html
            float u = nrand(uv, 0, index) * 2 - 1;
            float theta = nrand(uv, 1, index) * UNITY_PI * 2;
            float u2 = sqrt(1 - u * u);
            float3 v = float3(u2 * cos(theta), u2 * sin(theta), u);
            // Adjustment for distance distribution.
            float l = index / _SampleCount;
            return v * lerp(0.1, 1.0, l * l);
        }


        fixed4 frag_ssao (v2f i) : SV_Target
        {
            float2 uv = i.uv;
            fixed4 col = tex2D(_MainTex, i.uv);

            float4 norz = GetPixelValue(uv);
            float depth = norz.w * _ProjectionParams.z;
            float scale = _Radius / depth;

            float ao = 0.0;
            for(int j = 0; j < _SampleCount; j++)
            {
                float3 randNor = spherical_kernel(uv, float(j));
                if(dot(norz.xyz, randNor) < 0.0)
                    randNor *= -1.0;
                float2 off = randNor.xy * scale;
                float4 sampleNorz = GetPixelValue(uv + off);
                float depthDelta = depth - sampleNorz.w * _ProjectionParams.z;

                float3 sampleDir = float3(randNor.xy * _Radius, depthDelta);
                float occ = max(0.0, (dot(normalize(norz.xyz), normalize(sampleDir)) - _Bias) /
(length(sampleDir) + 1.0));
                ao += 1.0 - occ;
            }
            ao /= float(_SampleCount);
            return pow(ao, _Power);
        }

        //-------------------BLUR-----------------

        float4 _CameraDepthNormalsTexture_ST;
        sampler2D _SSAO;
        float3 _TexelOffsetScale;

        v2f vert_blur (appdata_img v)
        {
            v2f o;
            o.vertex = UnityObjectToClipPos (v.vertex);
            o.uv = TRANSFORM_TEX (v.texcoord, _CameraDepthNormalsTexture);
            return o;
        }

        inline half CheckSame (half4 n, half4 nn)
        {
            // difference in normals
            half2 diff = abs(n.xy - nn.xy);
            half sn = (diff.x + diff.y) < 0.1;
            // difference in depth
            float z = DecodeFloatRG (n.zw);
            float zz = DecodeFloatRG (nn.zw);
            float zdiff = abs(z-zz) * _ProjectionParams.z;
            half sz = zdiff < 0.2;
            return sn * sz;
        }
```

```
half4 frag_blur( v2f i ) : SV_Target
{
    #define NUM_BLUR_SAMPLES 4

    float2 o = _TexelOffsetScale.xy;

    half sum = tex2D(_SSAO, i.uv).r * (NUM_BLUR_SAMPLES + 1);
    half denom = NUM_BLUR_SAMPLES + 1;

    half4 geom = tex2D (_CameraDepthNormalsTexture, i.uv);

    for (int s = 0; s < NUM_BLUR_SAMPLES; ++s)
    {
        float2 nuv = i.uv + o * (s+1);
        half4 ngeom = tex2D (_CameraDepthNormalsTexture, nuv.xy);
        half coef = (NUM_BLUR_SAMPLES - s) * CheckSame (geom, ngeom);
        sum += tex2D (_SSAO, nuv.xy).r * coef;
        denom += coef;
    }
    for (int q = 0; q < NUM_BLUR_SAMPLES; ++q)
    {
        float2 nuv = i.uv - o * (q+1);
        half4 ngeom = tex2D (_CameraDepthNormalsTexture, nuv.xy);
        half coef = (NUM_BLUR_SAMPLES - q) * CheckSame (geom, ngeom);
        sum += tex2D (_SSAO, nuv.xy).r * coef;
        denom += coef;
    }
    return sum / denom;
}

//--------------COMPOSITION----------------

fixed4 frag_composition(v2f i) : SV_TARGET {
    return tex2D(_SSAO, i.uv) * tex2D(_MainTex, i.uv);
}

//--------------DEBUG----------------
fixed4 frag_debug(v2f i) : SV_TARGET {
    return tex2D(_SSAO, i.uv);
}
```

```
ENDCG

SubShader
{
    // No culling or depth
    Cull Off ZWrite Off ZTest Always

    //0 : SSAO Pass
    Pass
    {
        CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag_ssao
        ENDCG
    }

    //1 : Blur Pass
    Pass
    {
        CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag_blur
        ENDCG
    }

    //2 : Composition
    Pass
    {
        CGPROGRAM
            #pragma vertex vert_blur
            #pragma fragment frag_composition
        ENDCG
    }

    //3 : Debug
    Pass
    {
        CGPROGRAM
```

```
                #pragma vertex vert_blur
                #pragma fragment frag_debug
            ENDCG
        }
    }
}
```

SSDO shader code

```
Shader "Dissertation/SSDO"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Noise("Noise", 2D) = "white" {}
        _Bias("Bias", Range(0, 1)) = 0.6
        _Power("Power", float) = 5
        _SampleCount("Sample count", range(4, 32)) = 16
        _IlluminationRadius("Illumination radius", Range(0, 5)) = 0.01
        _AORadius("AO radius", Range(0, 1)) = 0.1
        _IndirectScale("Indirect scale", float) = 0.1
        _EnvScale("Environment scale", float) = 0.1
        _DistOffset("Distance offset", float) = 0.1
        _DistScale("Distance scale", float) = 0.1
    }

    CGINCLUDE

        #include "UnityCG.cginc"

        float _Bias;
        float _Power;
        float _SampleCount;
        float _IlluminationRadius;
        float _AORadius;
        float _DistOffset;
        float _DistScale;


        sampler2D _MainTex;
        sampler2D _CameraDepthTexture;
        sampler2D _CameraDepthNormalsTexture;
        sampler2D _CameraGBufferTexture0;
        sampler2D _CameraGBufferTexture3;
        float _IndirectScale;
        float _EnvScale;

        float4x4 _InverseView;

        struct v2f_ssdo
        {
            float4 position : SV_POSITION;
            float2 texcoord : TEXCOORD0;
            float3 ray : TEXCOORD1;
        };

        // Vertex shader that procedurally outputs a full screen triangle
        v2f_ssdo vert_ssdo(uint vertexID : SV_VertexID)
        {
            // Render settings
            float far = _ProjectionParams.z;
            float2 orthoSize = unity_OrthoParams.xy;
            float isOrtho = unity_OrthoParams.w; // 0: perspective, 1: orthographic

            // Vertex ID -> clip space vertex position
            float x = (vertexID != 1) ? -1 : 3;
            float y = (vertexID == 2) ? -3 : 1;
            float4 vpos = float4(x, y, 1, 1);

            // Perspective: view space vertex position of the far plane
            float3 rayPers = mul(unity_CameraInvProjection, vpos) * far;

            // Orthographic: view space vertex position
            float3 rayOrtho = float3(orthoSize * vpos.xy, 0);

            v2f_ssdo o;
            o.position = vpos;
            o.texcoord = (vpos.xy + 1) / 2;
            #if UNITY_UV_STARTS_AT_TOP
                o.texcoord.y = 1-o.texcoord.y;
            #endif
            o.ray = lerp(rayPers, rayOrtho, isOrtho);
            return o;
        }

        float3 ComputeViewSpacePosition(v2f_ssdo input)
        {
            // Render settings
            float near = _ProjectionParams.y;
            float far = _ProjectionParams.z;
            float isOrtho = unity_OrthoParams.w; // 0: perspective, 1: orthographic

            // Z buffer sample
            float z = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, input.texcoord);

            // Perspective: view space position = ray * depth
            float3 vposPers = input.ray * Linear01Depth(z);

            // Orthographic: linear depth (with reverse-Z support)
        #if defined(UNITY_REVERSED_Z)
            float depthOrtho = -lerp(far, near, z);
        #else
            float depthOrtho = -lerp(near, far, z);
        #endif

            // Orthographic: view space position
            float3 vposOrtho = float3(input.ray.xy, depthOrtho);

            // Result: view space position
            return lerp(vposPers, vposOrtho, isOrtho);
        }

        float3 ComputeViewSpacePosition(v2f_ssdo input, float2 uv)
        {
            // Render settings
            float near = _ProjectionParams.y;
            float far = _ProjectionParams.z;
            float isOrtho = unity_OrthoParams.w; // 0: perspective, 1: orthographic

            // Z buffer sample
            float z = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, uv);

            // Perspective: view space position = ray * depth
            float3 vposPers = input.ray * Linear01Depth(z);

            // Orthographic: linear depth (with reverse-Z support)
        #if defined(UNITY_REVERSED_Z)
            float depthOrtho = -lerp(far, near, z);
        #else
            float depthOrtho = -lerp(near, far, z);
        #endif

            // Orthographic: view space position
            float3 vposOrtho = float3(input.ray.xy, depthOrtho);

            // Result: view space position
            return lerp(vposPers, vposOrtho, isOrtho);
        }

        half4 VisualizePosition(v2f_ssdo input, float3 pos)
        {
            half3 color = pow(abs(cos(pos * UNITY_PI * 4)), 20);
            return half4(color, 1);
        }

        float3 GetWorldPosition(v2f_ssdo input) {
            return mul(_InverseView, float4(ComputeViewSpacePosition(input), 1)).xyz;
        }

        float3 GetWorldPosition(v2f_ssdo input, float2 uv) {
            return mul(_InverseView, float4(ComputeViewSpacePosition(input, uv), 1)).xyz;
        }

        float4 GetPixelValue(in float2 uv) {
            half3 normal;
            float depth;
            DecodeDepthNormal(tex2D(_CameraDepthNormalsTexture, float2(uv.x, uv.y)), depth,
normal);
            return fixed4(normal, depth);
        }
```

```
float nrand(float2 uv, float dx, float dy){
    uv += float2(dx, dy + _Time.x);
    return frac(sin(dot(uv, float2(12.9898, 78.233))) * 43758.5453);
}

float3 spherical_kernel(float2 uv, float index){
    // Uniformly distributed points
    // http://mathworld.wolfram.com/SpherePointPicking.html
    float u = nrand(uv, 0, index) * 2 - 1;
    float theta = nrand(uv, 1, index) * UNITY_PI * 2;
    float u2 = sqrt(1 - u * u);
    float3 v = float3(u2 * cos(theta), u2 * sin(theta), u);
    // Adjustment for distance distribution.
    float l = index / _SampleCount;
    return v * lerp(0.1, 1.0, l * l);
}


fixed4 frag_ssdo (v2f_ssdo i) : SV_Target
{
    float2 uv = i.texcoord;

    fixed4 col = tex2D(_MainTex, i.texcoord);
    float4 norz = GetPixelValue(uv);
    float depth = norz.w * _ProjectionParams.z;
    float illuminationScale = _IlluminationRadius / depth;
    float aoScale = _AORadius / depth;

    float ao = 0.0;
    fixed4 e = fixed4(0,0,0,0);
    float3 worldPos = GetWorldPosition(i);
    for(int j = 0; j < _SampleCount; j++)
    {
        float3 randNor = spherical_kernel(uv, float(j));
        if(dot(norz.xyz, randNor) < 0.0)
            randNor *= -1.0;
        float3 ray = normalize(randNor);

        float2 illumOffset = randNor.xy * illuminationScale;
```

```
                float2 aoOffset = randNor.xy * aoScale;
                float4 aoSampleNorz = GetPixelValue(uv + aoOffset);
                float4 illumSampleNorz = GetPixelValue(uv + illumOffset);

                fixed4 diffuseCol = tex2D(_CameraGBufferTexture0, uv + illumOffset);
                fixed4 lightCol = tex2D(_CameraGBufferTexture3, uv + illumOffset);
                fixed4 lightColAO = tex2D(_CameraGBufferTexture3, uv + aoOffset);
                fixed4 sampleColor = (diffuseCol + lightCol) * lightCol;

                float3 distVec = (GetWorldPosition(i, uv + illumOffset) - worldPos) * _DistScale;
                float dist2 = dot(distVec, distVec) + _DistOffset;
                float cos1 = saturate(dot(illumSampleNorz.xyz, -ray));
                float cos2 = saturate(dot(norz.xyz, ray));

                float depthDelta = depth - aoSampleNorz.w * _ProjectionParams.z;

                float3 sampleDir = float3(randNor.xy * _AORadius, depthDelta);
                float occ = max(0.0, (dot(normalize(norz.xyz), normalize(sampleDir)) - _Bias) /
(length(sampleDir) + 1.0));
                ao += (1.0 - occ * (1 - lightColAO));
                e += sampleColor * _IndirectScale * cos1 * cos2 / dist2;
            }
            ao /= float(_SampleCount);
            e = tex2D(_CameraGBufferTexture0, uv) * e * _EnvScale / float(_SampleCount);

            return fixed4(e.xyz, pow(ao, _Power));
        }

        //-------------------BLUR-----------------

        struct v2f
        {
            float2 uv : TEXCOORD0;
            float4 vertex : SV_POSITION;
        };

        v2f vert (appdata_img v)
        {
            v2f o;
            o.vertex = UnityObjectToClipPos(v.vertex);
            o.uv = v.texcoord;
            return o;
        }

        float4 _CameraDepthNormalsTexture_ST;
        sampler2D _SSDO;
        float3 _TexelOffsetScale;

        v2f vert_blur (appdata_img v)
        {
            v2f o;
            o.vertex = UnityObjectToClipPos (v.vertex);
            o.uv = TRANSFORM_TEX (v.texcoord, _CameraDepthNormalsTexture);
            return o;
        }

        inline half CheckSame (half4 n, half4 nn)
        {
            // difference in normals
            half2 diff = abs(n.xy - nn.xy);
            half sn = (diff.x + diff.y) < 0.1;
            // difference in depth
            float z = DecodeFloatRG (n.zw);
            float zz = DecodeFloatRG (nn.zw);
            float zdiff = abs(z-zz) * _ProjectionParams.z;
            half sz = zdiff < 0.2;
            return sn * sz;
        }

        fixed4 frag_blur( v2f i ) : SV_Target
        {
            #define NUM_BLUR_SAMPLES 4

            float2 o = _TexelOffsetScale.xy;

            fixed4 sum = tex2D(_SSDO, i.uv) * (NUM_BLUR_SAMPLES + 1);
            half denom = NUM_BLUR_SAMPLES + 1;

            half4 geom = tex2D (_CameraDepthNormalsTexture, i.uv);
```

```
            for (int s = 0; s < NUM_BLUR_SAMPLES; ++s)
            {
                float2 nuv = i.uv + o * (s+1);
                half4 ngeom = tex2D (_CameraDepthNormalsTexture, nuv.xy);
                half coef = (NUM_BLUR_SAMPLES - s) * CheckSame (geom, ngeom);
                sum += tex2D (_SSDO, nuv.xy) * coef;
                denom += coef;
            }
            for (int q = 0; q < NUM_BLUR_SAMPLES; ++q)
            {
                float2 nuv = i.uv - o * (q+1);
                half4 ngeom = tex2D (_CameraDepthNormalsTexture, nuv.xy);
                half coef = (NUM_BLUR_SAMPLES - q) * CheckSame (geom, ngeom);
                sum += tex2D (_SSDO, nuv.xy) * coef;
                denom += coef;
            }
            return sum / denom;
        }

        //--------------COMPOSITION----------------

        fixed4 frag_composition(v2f i) : SV_TARGET {
            fixed4 ssdo = tex2D(_SSDO, i.uv);
            half ao = ssdo.a;
            return (fixed4(ssdo.xyz, 1) + tex2D(_MainTex, i.uv)) * ao;
        }

        //--------------DEBUG----------------
        fixed4 frag_debug(v2f i) : SV_TARGET {
            fixed4 ssdo = tex2D(_SSDO, i.uv);
            half ao = ssdo.a;
            return (fixed4(ssdo.xyz, 1) + fixed4(0.1,0.1,0.1,0.1)) * ao;
        }

    ENDCG

    SubShader
    {
        // No culling or depth
        Cull Off ZWrite Off ZTest Always
        //0 : SSDO Pass
        Pass
        {
            CGPROGRAM
            #pragma vertex vert_ssdo
            #pragma fragment frag_ssdo

            ENDCG
        }

        //1 : Blur Pass
        Pass
        {
            CGPROGRAM
                #pragma vertex vert
                #pragma fragment frag_blur
            ENDCG
        }

        //2 : Composition
        Pass
        {
            CGPROGRAM
                #pragma vertex vert_blur
                #pragma fragment frag_composition
            ENDCG
        }

        //3 : Debug
        Pass
        {
            CGPROGRAM
                #pragma vertex vert_blur
                #pragma fragment frag_debug
            ENDCG
        }
    }
}
```

C# code

SSAO C# code

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[ExecuteInEditMode]
[RequireComponent (typeof (Camera))]
public class SSAO : MonoBehaviour {

    public Shader ssaoShader;
    [Range (4, 32)]
    public int sampleCount = 16;
    [Range (0.01f, 1.0f)]
    public float radius = 0.3f;
    [Range (0.0f, 1.0f)]
    public float bias = 0.6f;
    public float power = 1;
    [Range (0, 4)]
    public int blur = 2;
    [Range(0, 1)]
    public int downsampling = 0;
    public bool debugView = false;

    private Material ssaoMaterial;
    private bool supported = true;

    private static Material CreateMaterial () {
        Shader shader = Shader.Find ("Dissertation/SSAO");
        if (!shader) {
            return null;
        }
        Material m = new Material (shader);
        m.hideFlags = HideFlags.HideAndDontSave;
        return m;
    }
    private static void DestroyMaterial (Material mat) {
        if (mat) {
            DestroyImmediate (mat);
            mat = null;
        }
    }

    void OnEnable () {
        GetComponent<Camera> ().depthTextureMode |= DepthTextureMode.DepthNormals;
    }

    void OnDisable () {
        DestroyMaterial (ssaoMaterial);
    }

    private void Start () {
        if (!SystemInfo.supportsImageEffects || !SystemInfo.SupportsRenderTextureFormat
(RenderTextureFormat.Depth)) {
            supported = false;
            enabled = false;
            return;
        }
        CreateMaterials ();
        if (!ssaoMaterial) {
            supported = false;
            enabled = false;
            return;
        }
        supported = true;
    }

    private void CreateMaterials () {
        if (!ssaoMaterial && ssaoShader.isSupported) {
            ssaoMaterial = CreateMaterial ();
        }
    }

    [ImageEffectOpaque]
    void OnRenderImage (RenderTexture source, RenderTexture destination) {
        if (!supported || !ssaoShader.isSupported) {
            enabled = false;
            return;
        }
        CreateMaterials ();

        RenderTexture rtAO = RenderTexture.GetTemporary (source.width, source.height, 0);
        ssaoMaterial.SetFloat ("_Bias", bias);
        ssaoMaterial.SetFloat ("_Power", power);
        ssaoMaterial.SetFloat ("_SampleCount", sampleCount);
        ssaoMaterial.SetFloat ("_Radius", radius);

        bool doBlur = blur > 0;
        Graphics.Blit (doBlur ? null : source, rtAO, ssaoMaterial, 0);
        if (doBlur) {
            RenderTexture rtBlurX = RenderTexture.GetTemporary (source.width >> downsampling,
source.height >> downsampling, 0);
            ssaoMaterial.SetVector ("_TexelOffsetScale", new Vector4 ((float) blur /
source.width, 0, 0, 0));
            ssaoMaterial.SetTexture ("_SSAO", rtAO);
            Graphics.Blit (null, rtBlurX, ssaoMaterial, 1);
            RenderTexture.ReleaseTemporary (rtAO);

            RenderTexture rtBlurY = RenderTexture.GetTemporary (source.width >> downsampling,
source.height >> downsampling, 0);
            ssaoMaterial.SetVector ("_TexelOffsetScale", new Vector4 (0, (float) blur /
source.height, 0, 0));
            ssaoMaterial.SetTexture ("_SSAO", rtBlurX);
            Graphics.Blit (source, rtBlurY, ssaoMaterial, 1);
            RenderTexture.ReleaseTemporary (rtBlurX);
            rtAO = rtBlurY;
        }

        ssaoMaterial.SetTexture ("_SSAO", rtAO);
        if (debugView) {
            Graphics.Blit (source, destination, ssaoMaterial, 3);
        } else {
            Graphics.Blit (source, destination, ssaoMaterial, 2);
        }
        RenderTexture.ReleaseTemporary (rtAO);
    }
}
```

SSDO C# code

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[ExecuteInEditMode]
[RequireComponent (typeof (Camera))]
public class SSDO : MonoBehaviour {

    public Shader ssdoShader;
    [Range (4, 32)]
    public int sampleCount = 16;
    [Range (0.01f, 5.0f)]
    public float illuminationRadius = 0.3f;
    [Range (0.01f, 1.0f)]
    public float AORadius = 0.3f;
    [Range (0.0f, 1.0f)]
    public float bias = 0.6f;
    public float power = 1;
    public float indirectScale = 0.1f;
    public float environmentScale = 0.1f;
    public float distanceScale = 0.1f;
    public float distanceOffset = 0.1f;
    [Range (0, 4)]
    public int blur = 2;
    [Range (0, 4)]
    public int downsampling = 0;
    public bool debugView = false;

    private Material ssdoMaterial;
    private bool supported = true;

    private static Material CreateMaterial () {
        Shader shader = Shader.Find ("Dissertation/SSDO");
        if (!shader) {
            return null;
        }
        Material m = new Material (shader);
        m.hideFlags = HideFlags.HideAndDontSave;
        return m;
    }

    private static void DestroyMaterial (Material mat) {
        if (mat) {
            DestroyImmediate (mat);
            mat = null;
        }
    }

    void OnEnable () {
        GetComponent<Camera> ().depthTextureMode |= DepthTextureMode.DepthNormals;
    }

    void OnDisable () {
        DestroyMaterial (ssdoMaterial);
    }

    private void Start () {
        if (!SystemInfo.supportsImageEffects || !SystemInfo.SupportsRenderTextureFormat
(RenderTextureFormat.Depth)) {
            supported = false;
            enabled = false;
            return;
        }
        CreateMaterials ();
        if (!ssdoMaterial) {
            supported = false;
            enabled = false;
            return;
        }
        supported = true;
    }

    private void CreateMaterials () {
        if (!ssdoMaterial && ssdoShader.isSupported) {
            ssdoMaterial = CreateMaterial ();
        }
    }
    [ImageEffectOpaque]
    void OnRenderImage (RenderTexture source, RenderTexture destination) {
        if (!supported || !ssdoShader.isSupported) {
            enabled = false;
            return;
        }
        CreateMaterials ();

        RenderTexture rtSSDO = RenderTexture.GetTemporary (source.width, source.height, 0);
        var matrix = GetComponent<Camera> ().cameraToWorldMatrix;
        ssdoMaterial.SetMatrix ("_InverseView", matrix);
        ssdoMaterial.SetFloat ("_Bias", bias);
        ssdoMaterial.SetFloat ("_Power", power);
        ssdoMaterial.SetFloat ("_SampleCount", sampleCount);
        ssdoMaterial.SetFloat ("_IlluminationRadius", illuminationRadius);
        ssdoMaterial.SetFloat ("_AORadius", AORadius);
        ssdoMaterial.SetFloat ("_IndirectScale", indirectScale);
        ssdoMaterial.SetFloat ("_EnvScale", environmentScale);
        ssdoMaterial.SetFloat ("_DistScale", distanceScale);
        ssdoMaterial.SetFloat ("_DistOffset", distanceOffset);

        bool doBlur = blur > 0;
        Graphics.Blit (doBlur ? null : source, rtSSDO, ssdoMaterial, 0);
        if (doBlur) {
            RenderTexture rtBlurX = RenderTexture.GetTemporary (source.width >> downsampling,
source.height >> downsampling, 0);
            ssdoMaterial.SetVector ("_TexelOffsetScale", new Vector4 ((float) blur /
source.width, 0, 0, 0));
            ssdoMaterial.SetTexture ("_SSDO", rtSSDO);
            Graphics.Blit (null, rtBlurX, ssdoMaterial, 1);
            RenderTexture.ReleaseTemporary (rtSSDO);

            RenderTexture rtBlurY = RenderTexture.GetTemporary (source.width >> downsampling,
source.height >> downsampling, 0);
            ssdoMaterial.SetVector ("_TexelOffsetScale", new Vector4 (0, (float) blur /
source.height, 0, 0));
            ssdoMaterial.SetTexture ("_SSDO", rtBlurX);
            Graphics.Blit (source, rtBlurY, ssdoMaterial, 1);
            RenderTexture.ReleaseTemporary (rtBlurX);
            rtSSDO = rtBlurY;
        }

        ssdoMaterial.SetTexture ("_SSDO", rtSSDO);
        if (debugView) {
            Graphics.Blit (source, destination, ssdoMaterial, 3);
        } else {
            Graphics.Blit (source, destination, ssdoMaterial, 2);
        }
        RenderTexture.ReleaseTemporary (rtSSDO);
    }
}
```

REFERENCES

[1] A. Sherrod, Ultimate 3D Game engine Design & Architecture, Charles River Media, 2007.

[2] Ritschel, Seidel and Grosch, Approximating Dynamic Global Illumination in Image Space, 2009.

[3] Theoharis, Papaioannou, Platis and Patrikalakis, "Chapter 12, Illumination Models and Algorithms," in Graphics & Visualization, Principles and Algorithms, 2015, pp. 367-427.

[4] J. Stevens, "Shader lighting models," [Online]. Available: http://www.jordanstevenstechart.com/lighting-models.

[5] Theoharis, Papaioannou, Platis and Patrikalakis, "Chapter 16, Global Illumination Algorithms," in Graphics & Visualization, Principles and Algorithms, 2015, pp. 565-613.

[6] Theoharis, Papaioannou, Platis and Patrikalakis, "Chapter 15, Ray Tracing," in Graphics & Visualization, Principles and Algorithms, 2015, pp. 529-564.

[7] J. Kajiya, "The Rendering Equation," in Computer Graphics (SIGGRAPH 86 Conference Proceedings), 1986, pp. 143-150.

[8] A. Keller, "Instant Radiosity," 1997.

[9] B. Segovia, J. C. Iehl, R. Mitanchey and B. Péroche, "Bidirectional Instant Radiosity," in Eurographics Symposium on Rendering (2006), 2006.

[10] S. Zhukov, A. Iones and G. Kronin, "An ambient light illumination model," in Eurographics Rendering Workshop, 1998, pp. 45-56.

[11] L. Bavoil, M. Sainz and R. Dimitrov, "Image-Space Horizon-Based Ambient Occlusion," 2008.

[12] C. Daschbacher and M. Stamminger, "Reflective Shadow Maps," in Proceedings of the 2005 symposium on Interactive 3D graphics and games, 2005.

[13] P. Debevec, "Rendering Synthetic Objects into Real Scenes: Bridging Traditional and Imagebased," in Proceedings SIGGRAPH 98, 1998.

[14] M. Mittring, "Finding Next Gen - CryEngine 2," in Advanced Real-Time Rendering in 3D Graphics and Games Course - SIGGRAPH 2007, 2007.

[15] B. Segovia, J. C. Iehl, R. Mitanchey and B. Péroche, "Non-interleaved Deferred Shading of Interleaved Sample Patterns," in SIGGRAPH/Eurographics Graphics Hardware, 2006.

[16] "Sphere Point Picking," [Online]. Available: http://mathworld.wolfram.com/SpherePointPicking.html.

[17] "Bilateral Filter on Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Bilateral_filter.

[18] "Marmoset," 2015. [Online]. Available: https://www.marmoset.co/posts/basic-theory-of-physically-based-rendering/.

[19] "Renderwonk," 2010. [Online]. Available: http://renderwonk.com/publications/s2010-shading-course/.

[20] "Unity," 2018. [Online]. Available: https://unity3d.com.

[21] "Unreal Engine," 2018. [Online]. Available: https://www.unrealengine.com.

[22] "YoYo Games," 2018. [Online]. Available: https://www.yoyogames.com.

[23] M. Bunnel, "Chapter 14. Dynamic Ambient Occlusion and Indirect Lighting," in GPU Gems 2, Addison Wesley, 2005, pp. 223-233.

[24] R. Bonet Torres, "Unity SRP Overview: Scriptable Render Pipeline," 2018. [Online]. Available: https://www.gamasutra.com/blogs/RubenTorresBonet/20180419/316713/Unity_SRP_Overview_Scriptable_Render_Pipeline.php.